FastTrak

HAMMER Technologies

NOW IT'S YOUR TIME...

USER MANUAL

CREATE YOUR OWN GAMES!

User Manual

Written By : Daniel Navarro Medrano

UK Edited By: Leo Zullo, Jon Silvera & Mike Green

Published By: FastTrak Software Publishing

FOREWORD

Thank you for purchasing DIV Games Studio. The product is the result of over two years of continuous hard work by a team of people who have done their best to make their dream come true.

Programming has a bad reputation of being a complicated and dark field, even more so when it comes to games. The reason for such an image is the lack of information and the secrecy of the alleged geniuses who have the knowledge. We are not geniuses. We make games because we like them and we would like to share with you the information we have about our hobby.

One day someone imagined a tool to create computer video games which would contain all the necessary utilities.

Everything started with the idea of inventing a new programming language, the first specifically designed for the development of video games. Many options were considered, and all existing programming languages were thoroughly studied with the aim of coming up with a new concept. A language which could be accessible to beginners and at the same time powerful enough to achieve professional developments.

The result of that search is the ***DIV programming language***: a compromise between several options. The main problem was to determine which particular features video games had in common which are not shared by the rest of the programs. This was essential in order to design a language which would make video games development easier.

Video games are probably the only type of computer programs that share less things in common with one another. Therefore the first decision had to be about the type of video games that the language was to include since only a general purpose language would be able to include every imaginable game.

Did this challenge have any sense then? Well, it seems that it did. The constant features of video games are the use of graphics on the screen and the programming of movement, animation, sound and so on. DIV is focused towards these common features.

However, DIV Games Studio is much more than a programming language. It is a graphics environment where you can start developing a game and finish it without ever exiting DIV. We have tried to put everything into it that our imagination could conceive, and we almost did. I say almost because quite a few ideas have had to be left out. The only consolation we have is that perhaps tomorrow, in a new version, those ideas could be added.

But that's something that right now is out of our hands. Now it is up to you. It's up to you to become captivated by DIV Games Studio, and to really enjoy and utilise our work. That was the hope that encouraged us to make it all the way here.

Our aim was to develop a simple tool to use, so that it only requires the user to have a passion for video games.

We have put everything we have into DIV and the product is intended to be easy to use, professionally finished and at a very competitive price.

We think that DIV is the best way to learn how to program and at the same time the most entertaining one.

If there isn't a creator in you, we won't be able to make you develop incredible graphics and programs which could become amazing games. But if there is that creator in you, we will provide you with everything you need to make it come out.

You may be surprised by the great number of sample games that are included with DIV Games Studio. The reason why we have included all these games is that we feel that the only way to really learn something is to see it working. Those of us that have been working in programming for some time now are addicted to it and all of us have learned by watching others.

The games that we included are not very complex - they are just examples of some of the things you can do with DIV. None of these games use even 20 % of the real possibilities DIV provides. We have tried to represent most of the typical styles of video games and you can learn a lot from them. Although the truth is that we have so much fun developing them that the only reason we haven't made more is because we weren't allowed to (budgets, you know).

What can I expect to get from DIV Games Studio? That's a good question. Well, we think that at first you should just take it as a game, an adventure, you should investigate whatever you feel like, try plenty of things and if eventually you decide to make some kind of serious work, even professional, you can be sure that DIV will not disappoint you.

Perhaps you'll be responsible for the next successful hit. That's up to you now. We will be waiting for you.

Daniel Navarro Medrano

CONTENTS:

Chapter 1
Introduction

1

CHAPTER 1: Introduction

This first chapter will give you directions on how to install the program onto your computer and how to configure it so that it works optimally. The chapter also presents the basic concepts that will allow you to get to grips with the program from the very beginning.

1.1 Installation of DIV Games Studio

To function correctly this program needs the following minimum system requirements.

Program technical requirements

- 486 processor or higher (Pentium recommended).
- MS DOS or Windows™ 95/98 operating system.
- 8 MB RAM memory (16 MB recommended).
- 30 MB free space in the hard disk (146 MB recommended).
- 2X CD-ROM or higher.
- Microsoft™ compatible mouse.
- SVGA Graphics card.
- Recommended: a sound card compatible with Sound Blaster™ or Gravis Ultrasound™.

The program could possibly be installed in PC's with lower processors (from a 386 with 4 MB memory), but the results will certainly be unsatisfactory.

This is a program of development and therefore you must always have additional space in the hard disk for the developments. You will need it to create, compile and run the programs. We recommend not to use the program if the free space in the hard disk (once the program has been installed) is less than 4 MB. Otherwise some options of the program may not work or run incorrectly.

Installation

Installing DIV is very simple. All you need is to execute the program INSTALL.EXE from the CD-ROM and indicate the drive, directory and type of installation you wish to do.

MS-DOS users

First enter the drive unit which corresponds to your CD ROM drive, for instance, if your CD-ROM unit is **D**, then type:

D:

Then, press **Enter**, and after that execute the installation program using the following statement:

INSTALL

Next, you press **Enter** again and the screen of the installation program presentation will be displayed.

Windows 95/98 Users

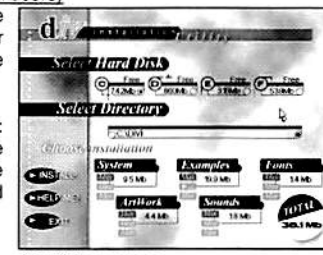
You can execute the program in different ways, for example, by clicking on the icon **MY COMPUTER**, and then choosing the CD-ROM where DIV Games Studio should be showing (both times by a double click on the left side of the mouse).

The program **INSTALL.EXE** will appear in the content of the CD-ROM unit. Double click on that program and the screen of the installation program presentation will be displayed.

Once the installation program has been executed (all users)

This installation program is controlled through the mouse. Click anywhere on screen with the mouse or press a key and the installation main screen will be displayed.

The screen will be shown divided in several parts: the hard disk chosen to install the program, the name of the target directory (or the folder for the program), the selection of installed components and the program options.



Installation screen

Follow these steps to install the program:

- If your PC has got more than one hard disk, choose one to install the program. To do this, you have to click on its' icon with the mouse.
- After that, if you do not want the program to be installed in a directory (folder) called **DIV**, enter the name of the directory in the center box text of the window (where the blinking cursor appears). If it doesn't matter to you which directory the program is installed, you can leave it in the directory **DIV**, which is the programs default.
- After selecting the components which are going to be installed in your computer, you must keep in mind that the space these components take (shown in the lower right side of the screen as **TOTAL**) must be less than the free space on the hard disk you have selected (shown underneath the hard drives' icon).

Choosing the installation

The program archives are grouped into five categories: **System**, **Examples**, **Fonts**, **Artwork** and **Sound**.

All of them have three small buttons that indicate the type of installation that can be performed: install all the archives of these category (**Max**), install a good combination between performance and space taken (**Med**) or install the minimum possible number of archives (**Min**). By default, the program will suggest the maximum installation for all the categories.

For all these categories the space required for the installation chosen is expressed in **MB** (megabytes of hard disk space). These five categories refer to the following groups of archives:

System: it refers to the DIV main and generic archives. Most of them are included in the minimum installation since they are essential to run the program. The installations Medium and Maximum include, apart from these, a higher number of extras, such as wallpapers and colour palettes.

Examples: here are the game examples which can be installed in the Games directory. They cannot be run directly from the CD-ROM and you will only be able to see those you install.

Selection of games examples

Since the games take a lot of space, you can click on the EXAMPLES to get a window where you can specify which particular games you would like to install. If you don't install all the games at this time, you may install them later. We recommend the installation of at least the tutorial ones which will be very useful to learn how to use DIV language.

In case you would like more information about the games for the purposes of installation, you can find a description of the main games at the end of this chapter.

Fonts: in this category are the different kinds of fonts available to write text within the games. The Minimum installation will install a small selection of fonts while the Maximum will install all available types.

Artwork: these archives include the graphics for games, some of DIV's own games examples and many other new graphics that you can use in your games without the need to draw them.

Sounds: this last category refers to archives containing sound effects ready to be used in the games. If you don't have a sound card, it will be better for you not to install them (thus choosing the Minimum installation) because you will be able to install them in the future if you decide to purchase a sound card (which we strongly encourage).

Recommended installation

If you have a big hard disk with a lot of free space, we recommend the Maximum installation (complete installation of the program).

If the space in your disk is more limited, we recommend the following installation:

System: Select the maximum installation for this category.
Examples: pick the ones you like best (but include the tutorials).
Fonts, Artwork and Sound: Select the minimum installation.

Whenever you would like to access the letter fonts, the graphics or the sounds for the games, you can read them directly from the CD-ROM of DIV Games Studio, selecting the corresponding unit and directory (**DATA \ IFS** for fonts, **DATA \ MAP** for artwork and **DATA \ PCM** for sound).

- Once all these elements have been selected, click on **INSTALL** to begin the program's installation. This operation will take a few minutes, depending on your PC and of the type of installation chosen.
- Finally, the program will indicate that the installation has been completed; then click **ACCEPT** to exit the installation program.

Note: You can always install the program again whenever you wish to add new elements (such as sample games that were not originally installed). To do that you need to follow this same process. You will not lose any work already performed in DIV, with the exception of the example programs which will resume to their original state (if you install them again) and therefore you could lose all changes you have made to them.

DIV Games Studio Execution
The program is already installed in your computer. All you have to do now is to execute it. For that, please follow the instructions below:

Users of MS-DOS

- Go to the hard disk where you installed the program. For instance, If it is **C**, you can do it by typing **C:** and pressing **Enter**.
- Go to the directory where the program was installed. For example, if it is **DIV** type **CD DIV** and press **Enter**.
- Now execute the program by typing **D** and pressing **Enter**.

Note for advanced users: if you include the directory of the program in the environment variable **PATH** (in the file **autoexec.bat**), you can execute DIV Games Studio from any directory. When you exit the program, it will be back again to the original directory.

Users of Windows 95/98

First of all, close the window of the program of installation, which you don't need now, and do as follows:

From the icon **MY COMPUTER** access the unit and folder where you installed the program (as you did when you executed the installation program). Once you are in the program folder, click twice on **D.EXE** to execute the program.

In the program folder you will also see an icon that can be dragged onto your desktop to create a shortcut. This will save you time in opening up DIV Games Studio.

Note: we recommend not to use **ALT+TAB** to go to another application from **DIV**, because sometimes, Windows will not restore the screen correctly when you return to the program later. In this case, press **ALT+X** and then **Enter** to exit the program and then execute it again (this way you will not lose the information when performing this operation).

Installation likely problems

You should not find have any problems in installing the program if you correctly follow the steps above.

Should you have any problems, please check the **minimum requirements** for the program stated earlier. If you have any doubts about any of the elements, please contact your computer technical service or supplier.

If you are using Windows and you have any doubts about the installation process, we strongly advice to look at Windows Help to solve any doubts you may have.

Important: you may have some problem reading the CD-ROM with the reader unit. If this happens, just wipe the DIV Games Studio CD surface using a clean and dry cloth and try installing the program again.

If you were not able to install the program and you have plenty of free space in your hard disk (enough for the program Maximum installation), you could try a manual installation of the program. Follow these steps:

MS-DOS users

Go to the CD-ROM unit using (if your CD-ROM unit is **D**):

```
D:
```

AND press **Enter**. Then type the following commands (if your hard disk unit is **C**):

```
XCOPY DATA\*.* C:\DIV\*.* /S
C:
CD DIV
DEL INSTALL*.*
```

You must press **Enter** after each one of those commands. If you have enough free space available the program will be installed with no problem. To run it, introduce the following command:

```
D /SAFE
```

For future executions of the program, just follow the regular instructions.

Users of Windows 95

Open a session of MS-DOS (either by clicking twice on its icon or from the menu start \ programs \ MS-DOS) and follow the instructions for MS-DOS users. For future executions, you may follow the instructions of your operating system, taking into account that the program will be installed in the folder **DIV** of your hard disk.

Note: if you are not able to install the program after trying all of these options, please contact the **FastTrak Technical Support Line** on 01923 495497 from **Monday** to **Friday** between **9:00 am** and **5.30 pm** local time. Or e-mail technical support at www.div-arena.com.

Important: no questions about the programming language will be answered by this service. For information and tips on programming refer to this book, the built in help (F1) and the resource centre provided at www.div-arena.com.

Problems with the mouse

If your mouse jumps on the screen instead of moving smoothly, you are using an obsolete or incorrect kind of mouse (a mouse driver which is not updated). You can solve this problem by doing one of the following:

- Contact your equipment supplier in order to get an updated mouse driver.
- Replace the program resolution with another one where this problem doesn't occur (item 1.3 of this book explains how to do this).
- (MS-DOS, only for advanced users). Comment the line of the **autoexec.bat** file where you load the mouse driver. You can do this from DIV itself. Load this file (placed in the root directory of your starting hard disk) with F4 and add the word **REM** at the beginning of the line which loads the mouse driver (this line generally ends with ...**mouse.com**). Then press **F2** to save the file again, close it (by clicking on the upper left icon of the text window), exit DIV (**ALT+X**) and restart the system. If you had any problems with any other program that needs this device, please edit the same file again and delete the word **REM** you added.

Sound Set-up

Normally the audio system of DIV Games Studio is automatically configured and the user doesn't need to introduce any parameters.

If you have a 16 Bit sound card and you cannot hear the sound effects of the sample games when running them, you can try to configure your sound card by proceeding as follows:

First make sure the sound system is not activated; for that use option **Open sound...** from the **sound menu**, and load a **PCM** archive (from the LIBRARY directory or from any of the sample games). The sound effect will appear represented in a small window inside DIV. If you click on it, you should be able to hear the effect. If you don't hear it, check the volume level, the speakers and the wires of your system.

In case of unsuccessful configuration:

- Press **F4** to open a program and load the archive **SETUP.PRG** which is in the **DIV** directory **SETUP** (click on the directory **..** (2 dots) to go up a level and then select this directory).
- Once the program has been loaded, a new text window where it will be contained will appear. Now press **F10** to execute it. The sound setup will be displayed.
- Enter your card parameters and click on the button **Save & Exit** and you will get right back to DIV environment. Then close the window with the program **SETUP.PRG** and try the sound system again.

If even after doing that, you were not able to hear the sound effects of the program, the reason could be that your sound card is not a **100%** compatible with the **Sound Blaster** or **Gravis Ultrasound** family. In that case, please contact your equipment supplier.

1.2 Introduction to the graphic environment

Basic Concepts

This first section briefly shows the **basic functioning** of the windows environment as well as many of the **terms** that will be used later on to describe more advanced functions. We recommend you to read it even if you are an expert user of other types of environments.

Every time you enter DIV Games Studio a dialog box will appear. From then on, the program can be controlled with the mouse.

About Box

Mouse Pointers

The mouse can be found on the screen thanks to a graphic called the **mouse pointer** which shows different forms depending on the action to be done by clicking on the left button of the mouse.


Normal Pointer: the **mouse standard pointer,** it's used to activate menu options, buttons, boxes and so on.


Hand Pointer: it tells us when we are passing over a special object, generally over a **movable object**, which can be taken to another position on the screen.


Hourglass Pointer: it appears when the system is executing some kind of internal process or calculation. It tells us that **no action can be done** with the program until such a process or calculation is finished.


Minus sign Pointer: it indicates the possibility of **minimising** a window, i.e. of temporarily reducing its size in order to leave open space in the desk.


Cross Pointer: it allows us **to close** a window. Depending on the kind of window the system may ask for confirmation before closing it.


Plus sign Pointer: it indicates the possibility to **maximise** a window; this is the opposite action to minimise and allows to return a window which was reduced to its original size.


Up Pointer: when the mouse is over certain controls, this pointer indicates the possibility to display or to **access to the previous elements in a list.** It also indicates the possibility of **bringing a window to the front.**


Dragging pointer: this graphic will appear when **an object is being dragged;** the pointer must be moved to the place of destination (to the wallpaper, the dustbin, a map, a file, etc.) and then the left button of the mouse must be released in order to leave the object there.


Down Pointer: opposite to the up pointer, this one indicates when we are passing over a control which allows us to display or to **gain access to the following elements in a list.**


Right Pointer: it indicates that there is a control **to move a display zone,** such as a text window, **to the right.**


Left Pointer: it will be displayed over the controls **to move a display zone to the left.**


Diagonal Arrow Pointer: it indicates the existence of a button **to change the size of a window.**


Horizontal Arrow Pointer: this graphic indicates the possibility of **moving a display window to the sides.**

Vertical Arrow Pointer: this graphic will appear over the **vertical slide bars**, generally relating to a list of elements; when you click on these controls, the elements in the list will slide.

Window Pointer: it appears on the controls that permit **to enlarge a window** unto its maximum size. When the window has already been enlarged to that size, this pointer shows the possibility of putting them back to their former size.

Forbidden Pointer: this graphic indicates that the **menu option is deactivated**, generally because it interacts with a certain object which is not loaded or selected in the program.

Click on the button "Accept" by placing the mouse pointer over it and by clicking with the left button of the mouse. Any time a dialog box appears, the following keys can be used:

ESC - To cancel the dialogue.
TAB - To choose the selected control
Enter - To activate the control that has been selected.

The controls that can be selected in a dialog box are the buttons and the text boxes.

Moving windows

The dialog boxes, like the rest of the windows, can be moved to any position on the screen by clicking on the title bar and dragging it to the new position.

Title bar. It is the upper side of the windows and shows the name of the window in white against a blue background.

Dragging. This term is used in the graphics environment when we click on an object with the left side of the mouse, we move it to a new position and then we release the mouse button.

When a new window is created, the system will place it in the part of the screen where it is considered that it hides the least information. The dialog boxes are an exception: they always appear in the middle of the screen because they require direct attention from the user.

If we would like the system to place a window automatically, we need to click twice on the title bar of the window. If the system finds a better position for the window, it will move it there.

Types of windows

Windows can be classified according to different criteria; **depending on their state**, they can be grouped in:

Foreground: they show the title bar and information and they are **highlighted** (those windows which are not covered totally or partially by others). These are the **only windows on which actions can be performed.** They can also be divided in two groups:

Active windows: unless the colour configuration has been changed, they have the title bar highlighted with white letters against a blue background.
Inactive windows: Their title bar is in black letters on a dark gray background. To activate one of these windows, all you need is to click on them.

Background: same as the foreground windows, except that these ones are **darkened** because they are at least partially covered by other windows. To interact with these windows it's necessary to bring them to the foreground by clicking on them. These background windows can also be **active or inactive** depending on the colour of its title bar (which is now darkened).

Icons: icons are windows that have been minimised, i.e. reduced temporarily. They do not display any drawing, only a plus sign and the window title. They will not be darkened when they go to a background.

Windows can also be classified **according to its function**:

Dialog boxes: they appear in the middle of the screen. They cannot be minimised (i.e. they cannot turn into an icon). They put the rest of the windows in a background (even if they are not covered). There are three kinds of dialog boxes.

Information windows: boxes like the one you find when you enter DIV Games Studio, with only one button, **accept,** used to continue the execution once the message has been read.

Interactive Dialogues: used to ask the user for information. There is a great variety of them and they will be seen in their respective options. They usually have at least two buttons, one to **accept** and the other to **cancel**.

Error Boxes: same as the information windows, except that the title bar is displayed in white against red. They inform about any problem that has occurred.

Conventional windows: they only leave in the background those windows they cover. They can be minimised at any time and they can be displayed anywhere on the screen. The main types of windows are shown below (the rest will be seen later):

Options menus: a **list of options** that take to other menus, windows or dialogues when clicking on them. All menus start from the main menu. Some menus have some options deactivated (in this case a forbidden pointer appears when the mouse is over them) because they interact with a specific type of window and no window of this kind is active (it will be necessary to create one or to load it). Chapter 2 shows all tools that can be accessed through the options menus.

Options Menus

Programs: we create the programs in these windows. They are text editor windows. In order to edit a program its' window must be activated (only one of the programs loaded can be activated). The text editor is very similar to other text editors. Appendix D offers a summary of the editing commands that are available. These windows are controlled from the programs menu and the edit menu. To get **help about a word of the language** in a program the blinking cursor (not the mouse pointer) must be placed over that word and then press **F1**. The size of these windows can be changed by clicking on their **lower right button** and dragging them with the mouse.

A Program Window

Maps or graphics: these windows contain a map (bitmap) or a graphic (a drawing used in a game). They are controlled through the Map menu. They can be loaded from a **MAP** archive (own format), or imported from a **PCX** or **BMP** archive, or they can be saved in any of these formats. To edit graphics, we must **double click** on these Windows with the mouse left button, thus entering in the graphic editor (described in chapter 3). These graphics **can be dragged** to the wallpaper (to make a copy), to another graphic (to insert them), to a file of graphics (to include them) or to the dustbin (to delete them). Maps can have any kind of size, the only limit is the available memory:

A Map or Graphic

Files of graphics: these windows always correspond to a **FPG file** in the disk. They are libraries or **graphics collections** used in the games. They are useful because that way you don't have to load a great number of maps separately in a game. They have two basic modes of functioning which are activated through boxes in their lower part. The first mode is **Tag/Drag** and is used either to tag and untag a series of graphics on which we would like an operation to be done or to drag graphics out of the file (to the wallpaper, to other files, maps or to the dustbin). The second mode is **Info**, and is the mode where you can edit the **codes of the graphics and their descriptions** or simply display its contents. The graphics files are controlled through the Files Menu.

A Graphics File

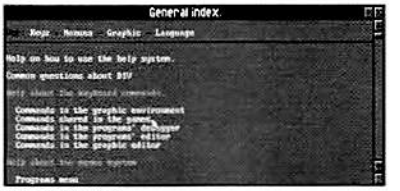
Fonts: the fonts or types of letters are small windows that represent a certain style of writing. If you click on them you will be able to see a sample of the font. They correspond to **FNT** (own format) and are controlled through the font menu from which the **fonts generator** can be accessed. This is the tool used to create new letter fonts. The fonts are used to write text in the drawing programs and in the games.

Fonts

Sound effects: they appear on the screen as **PCM's** (*pulse code modulation*) in small windows. They are controlled through the Sounds Menu and they can be imported from **WAV** archives. In the DIV directory you will a library with almost 1000 sound effects all ready for use in the games. To hear a sound effect you need a 16 Bit sound card which is correctly configured and then you need to click on one of these windows.

A Sound Effect

Help windows: they are controlled mainly by the mouse but you can also use the **cursors**, the **page down and page up keys** and **Backspace** to **return to the previous page** (this key is the one we use to delete and is situated right above the **Enter or Return** key). The help windows show the text in three colours: the main text in **black**, the highlighted text (in bold type) in **light gray** and the texts which refer to other help pages in **white.** To access those pages you must click on the references. The help windows also show **examples** which can be accessed by clicking on them and executed by pressing **F10** or aborted by pressing **ALT+X**.



A Help Window

(*) the dustbin window must be activated first from the menu system using the relevant option.

Windows Basic Controls

The main controls used in the windows are:


Minimise Button. First button in the title bar of the conventional windows. The dialog boxes don't have this button. It is used to minimise the windows, i.e. to turn them into an icon. Windows are minimised when free space is needed on the desktop window. An alternative is to close those windows without losing the information they contain.


Maximise Button. Only present in the icons. It is used to maximise the windows, restoring it to its complete size from the icon. The windows will go back to its original position unless they cover another window there and free space is available somewhere else on the desktop. Icons can be moved to anywhere on the screen by clicking on its name and then dragging. A double click on the icon name will make the system find the best possible position for the icon in the desk.

Close button. Used to close the windows, i.e. to eliminate them. It is present in all windows and dialog boxes but not in the icons since these ones must be maximised to be closed. In the windows which contain information that can be lost, confirmation will be asked before closing them. For the rest of the windows, such as menus, this confirmation will not be asked, since they can be created again any time.

Enlarge Button. It is displayed in the program windows and used to adjust the size of the window to its maximum. To put an enlarged window to its original size you have to click on this same button. This action can also be performed by pressing **Control+Z**, in which case, if there were several windows of the program, the one among them which is active will be enlarged (or minimised if already enlarged). To activate a window, all you need is to click on it with the mouse.

Rescaling Button. This plain button is displayed in the program and help windows. It is used to change the size of a window manually. Click on it with the left side of the button and, without releasing it, move the mouse until the window has the size you wish. The help windows can only be rescaled vertically (to add or to suppress lines but not columns). The program windows will never be over the limit of 80 columns per 50 lines.

Slide bar. This bar is always associated to a display window, such as a list of elements, and indicates the possibility of moving or sliding the contents of that window. In both of its ends there are two direction buttons used to slide the list of elements little by little. A movable rectangular indicator placed between both buttons shows the part of the list being displayed. By clicking on the bar the list will quickly slide until it gets to the chosen position.

Text Button. This button is displayed in the dialogue boxes and its function depends on the text written on the button. The most common ones are the buttons **Accept**, to validate the action of the dialog box and **Cancel**, to cancel the action. Very often this button will be the same as the button for closing the box described above and displayed in the upper right corner. If you wish to see the function of a specific text button, you must access the explanation of its dialog box. Most of them are described in chapter 2 of this book. Any text button can be selected by pressing once or twice on **TAB**, and can be activated with **Enter** (the button will be shown with a dark edge when selected).

Switch. The switches are options of the program which can be activated or deactivated. The text at the side of the switch refers to the option or feature of the program whose state can be established. Such features will be activated when inside the switch there is a black point. To activate or deactivate these options all you need is to click on the switch or on the text: it is **not** possible to modify a switch through the keyboard commands.

Text box. Used to ask for any kind of numeric or alphabetical information, but they can also be selected by using **TAB**. Once they are selected, a dark edge will be shown in the box and you will be able to start writing directly or otherwise press **Enter** to go to edit mode. Once the text has been edited, the key **Enter** will validate the new text and the key **ESC** will cancel the edition making the text box recover its previous content. A box can be edited by clicking on it; the first click will activate the box keeping the previous text, the second click will delete it.

The Menu System

The Menu System is a group of windows with options that derive from the window called **Main Menu**. Once it has been activated, a menu will remain on the screen until it is closed or minimised. Any menu can be created again from the main menu when we need it.

The Main Menu

When you refer to an option, you must indicate the name of the menu and then that of the option with an inverted bar between them. For instance, the option **Programs \ New..** refers to the first option of the menu of programs, used to create a new program.

Some menu options can be reached through shortcuts, i.e. combinations of keys that permit to do the action directly, even if you have not opened the menu. In these cases, the key combination is always shown in the menu itself, next to the text of the option.

To exit the environment, press **ALT+X** and a confirmation box will appear. To exit quickly without having to confirm, press **ESC+Control**. Windows users must be careful with the order in which they use this combination: first, they have to press **ESC** and, without releasing it, then press **Control.** If they use this combination the other way around (i.e. **Control+ESC**), it will take them to the start menu of the Windows system.

The contents of DIV Games Studio's desktop will be restored in the next execution of the program exactly as you left it when you exited, and therefore **information will never be lost when you exit** the environment.

Here is a summary of the functions of the most important menus you can access from the main menu. Chapter 2 gives a detailed description of the functions of these menus.

Programs. This menu allows to load programs, to execute them, to debug them or to create new programs. Programs are shown in the edit windows. From this menu we access the **Edit** menu, where the basic commands of text editing are shown (cut, paste, searching, replacements, etc).

Palettes. The colour palette is the whole 256 colours used by the game. From this menu you can load palettes (from multiple archive formats), record, edit (to create new palettes), give them an order, fuse them, etc. All graphics used simultaneously in a game (in the same screen or phase) must have been created with the same colour palette. When there is a graphic in the desktop window and you would like to load another one with a different palette, the system will make you adapt one of them to the palette of the other one. Chapter 3 gives a detailed explanation about the use of palettes.

Maps. Maps or graphics/sprites are the backbone of the games. This menu allows to work with archives **MAP**, **PCX** or **BMP**, to create both the graphics/sprites of the characters and the scenery of the games. Graphics of these files can be loaded, recorded, edited, copied and so on. To edit them you can use the option **Edit map** or double click with the mouse. From this menu you can also access the Explosion Generator.

Files. The FPG files are files that contain libraries or complete collections of graphics. They are used in the games to load many graphics in one go. In order to put graphics into a file, the map windows have to be dragged onto the File window. This menu gives access to the basic options of the Files.

Fonts. The letter fonts refer to the **FNT** archives which contain game fonts, ready to be used in the games. This menu allows you to perform the basic options with these archives, among them, to access the font generator which is used to create new fonts. The last options of this menu allows you to export fonts to graphic maps to retouch them manually in the graphic editor and them to import them back again into a font archive.

Sounds. This menu allows you to load sound effects from **PCM** archives and to hear them so that you can identify the suitable sound effects for a specific game among the ones in the DIV Games Studio sound library. These effects can also be imported from **WAV** archives, which is the windows standard format.

System. The system menu gives access to the generic tools of the environment and to the configuration options. From this menu the colours, fonts and video mode used in the environment can be defined. The configuration options are described next.

1.3 Configuration of the environment

Setting a videomode

The option **System \ Videomode...** gives access to a dialog box that allows us to modify the resolution used by the DIV Games Studio's graphic environment.

Videomode Selection Box

The video resolution is indicated as the number of pixels (horizontally and vertically measured) existing on screen in that mode. They vary between 320x200 (low resolution) and 1024x768 (highest resolution). It is necessary to click on the list appearing in the box and then, on the button **Accept** in order to select a new resolution.

Important: Some of these videomodes can incorrectly be displayed in some computers. In these cases, the first thing to do is to exit the environment by pressing the **ESC+Control** combination and then, re-enter DIV in the fail-safe mode. For that, the following command must be executed from the commands line of **MSDOS** and in the directory (folder) in which the program has been installed :

D.EXE /SAFE

Thus, the environment will be entered in low resolution (in 320x200, the most compatible mode). To position in the program's directory from the commands line, the following statement must be executed (supposing that the program has been installed in unit **C**, in the directory **DIV**):

C :
CD \DIV

In those computers in which the videomode is not compatible with the standard **VESA**, a **vesa driver** must be used. For that purpose, you must contact the supplier or the after-sales service of your hardware equipment (a **driver** is a short program that must be installed in the computer to give support to some devices such as, in this case, the monitor).

Due to problems of incompatibility of the **mouse driver**, it is possible that in some computers, the mouse pointer jumps. In these cases, it is necessary to update that **driver** or use another videomode for the environment, otherwise it won't be possible to correctly work in the graphic editor.

Two switches allow us to select the font used by the system:

Small font. In this mode, all the windows, menus and boxes will be seen in a smaller size, being appropriate for low resolution .

Big font. This mode **can only be activated in SVGA resolutions** (from 640x480) and it is the appropriate for these resolutions in 14 inch monitors.

The font used in programs and help windows is defined in the configuration window later described.

Configuring the wallpaper

To establish the desktop's wallpaper, it is necessary to access the option **System \ Background...**, which will invoke the following dialog box:

Wallpaper configuration

First, the button '...' indicating '**Source**' must be clicked to select an archive **MAP** with the graphic aimed to be used as background. If the aim is to use an archive **PCX** or **BMP** as a background, it must be first loaded with the option **Maps \ Open map...** and then, saved (**Maps \ Save as...**) indicating an archive name with the extension **.MAP**.

The appearance of this graphic will be indicated through the following switches:

Mosaic. If this switch is activated, the game will be displayed in its original size, not re-scaled, and if it is smaller than the screen, the picture will be repeated until the screen background is filled with it. When it is disabled, the picture will be re-scaled (expanded or reduced) to fit the screen size.

Colour / Monochrome. Out of these two switches, only one can be activated. The first one indicates that the picture will be taken in colour, adapting to the colour palette active in the environment (it normally implies a loss of quality for the picture). If the second switch is activated, the picture will be adapted to a colour range of the palette, which will be defined with three components: **Red, Green** and **Blue**.

These components, ranging from **0** to **9**, will be modified with the buttons '**-**' and '**+**'. They will only be applied when the switch indicating **Monochrome** is activated, and they define the lighter colour of the range. Some examples of combinations of these values and the resulting colour are now shown:

- Red=9, Green=9 and Blue=9 will define the **white** colour as the lightest one in the range (the wallpaper will be shown in black and white).
- Red=0, Green=0 and Blue=9 will define the **pure blue**. The wallpaper will be shown in different blue shades.

- Red=4, Green=0 and Blue=0 will define the **dark red** colour as the lighter one of the range. That is to say, the wallpaper will be seen in dark red shades.

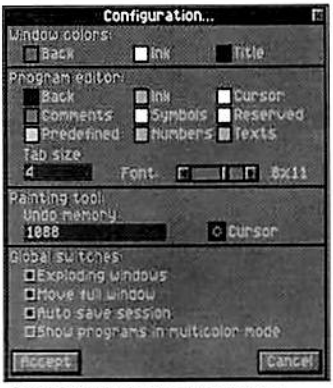
- Red=9, Green=9 and Blue=0 will define the colour **yellow** (red+green). Then, the background will be seen as a colours range defined between the black and yellow colours.

The resulting ranges must be adapted to the colours available in the palette, some of these ranges look better than the rest. Consequently, before finding a colour range compatible with the palette, several tests must normally be carried out.

The changes made in the wallpaper won't be visible until the dialog is finished by clicking on the button **Accept**.

The configuration system

Through the option **System \ Configuration...**, a dialog box will be accessed. From it, many aspects of the graphic environment can be defined.



Box To Configure The Environment

This dialog is split into several sections, shown next. These options won't take effect until the dialog finishes.

Window colours. This first section establishes the colours used by the environment. The background, ink and bar colours may be changed. In order to change one of these colours, it is necessary to click on the box with the colour, and a dialog box will appear that will show all the colours available in the active palette. The currently selected colour will be shown inside this box with a mark. To select another colour, it is necessary to click on it first, and on the button **Accept** later. The system doesn't only use these three colours but, from them, it generates other intermediate colours for texts, cursors, buttons, etc.

Program editor. The appearance of the programs edit window is established in this section. The three basic colours (windows background, characters ink and cursor colours), as well as the edit font (the letter type's size) can be selected from 6x8 pixels to 9x16 pixels. All the edit fonts are fixed spacing. To select another font, the buttons of the scrolling bar must be used. The text blocks tagged inside the editor will be seen with the ink and background colours exchanged.

Painting program. In this section, it is possible to define the quantity of memory reserved to undo options in the graphic editor and the mouse pointer used in the edit. The amount of memory is specified in Kbytes; by default, it equals 1088Kb (a little bit more than one MB). It is not necessary to modify this value, unless a task can not be performed in the graphic editor because there is not enough undo memory (in that case, the program will report it). The bigger the undo memory's reserved space is, the smaller the memory space available in the system will be for the rest of tasks. It is possible to select among three different pointer sets by clicking on the pointer graphic shown in this box.

Global options. Three switches appear in the last section. When they are activated, they will define the following characteristics:

Exploding windows. Indicates that all the windows' offsets must be displayed on opening, closing, minimising, etc. If this option is disabled, the environment will lose effectiveness, but it will answer more quickly.

Move complete windows. Indicates that, on dragging the windows to a new position, the final result must be seen all the time. The windows that are progressively uncovered will pass to the foreground and those that are hidden will pass to the background. It may be advisable to disable this option in slower computers.

Save session always. Indicates that, on exiting DIV Games Studio, the state of the desktop and all its objects (programs, maps, sounds, etc.) must be saved. If this option is disabled, the user will enter and exit the environment more quickly. However, the user risks losing works that have not been saved on exiting the environment.

If the configuration window is closed or the **ESC** key is pressed, all the changes made in it will be lost, restoring the values of the previous configuration.

1.4 Execution of the sample games

General instructions

This section will explain how to execute the sample games of DIV Games Studio and will also give you the instructions for all of them.

All these games are simple examples and therefore even though they are complete games most of them are very easy or very short. The reason for this is that we have tried to show the techniques they use and the way the programs are made and not to give endless lists where the user would get lost. Nevertheless we have put a lot of enthusiasm into these games and we hope you find them enjoyable (we like them very much...).

To execute any of the examples, first you have to load the program by using the option **Programs \ Open program...**, which will make a window appear containing the list of the program loaded.

We encourage the user to examine the programs and to try to make changes in them... ***this is one of the best ways to learn***. When you are over any reserved word, constant, variable, function, etc. of the language, you can press **F1** to see a help page about that item. If the help about the item doesn't appear (and what appears is the general index), it means that the item is not a name typical of the DIV language but a constant, variable or exclusive process of the game (processes are like functions which direct the performance of the graphs or of the '*sprites*', in the games).

We recommend to start with a simple game, like **STEROID.PRG** (*Steroid*, a version of the famous Asteroids) which although it is not technically very advanced or graphically very spectacular, it's one of the easiest ones to understand. Later on we can go to more complex ones such as **MALVADO.PRG** (*The castle of Dr. Malvado*, platforms) or **FOSTIATO.PRG** (*Fostiator*, a fighting arcade game in the traditional style). These will teach you many more things.

Texts starting with the symbol // (double bar) are explanation comments, they are not part of the program but clarifying notes about the running of the program. These comments are usually vital to be able to understand how the programs work since they can be placed anywhere in the program.

In order to go to one of the program processes (one of the blocks containing programs to control a graph or a *sprite* of the game) you have to press **F5** and select the name of that process by using the mouse.

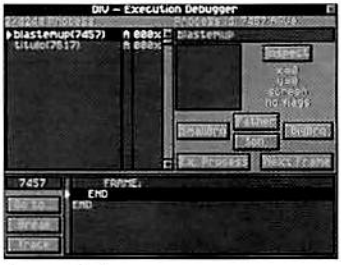
To execute one of the loaded games, you have to click on its window with the mouse and them press **F10** (you can also do this using the option **Programs \ Execute.**

Instructions for these games vary from one to another, but most of them allow to exit by pressing **ESC** and are mainly run with the **cursors** keys and with **Control**.

The **Pause** key can be used in all the games to stop its running momentarily.

All programs can be aborted at any point just by pressing the key combination **ALT+X**.

The most curious of minds can go into the games by pressing **F12** (from the game itself, when this is being executed). This key allows to access the **program debugger**, which is a tool designed to execute the games step by step. This way you can watch all processes and modify its variables (if you find the appropriate variable, you can change all the game parameters, the phase number, the lives,...). Of course at this point we don't guarantee the results you get.

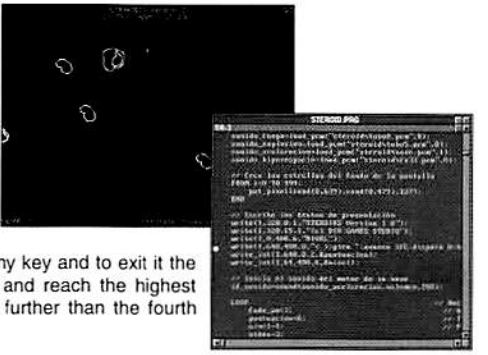


Program Debugger

Let's play!...

EXAMPLE GAMES

STEROID v1.0



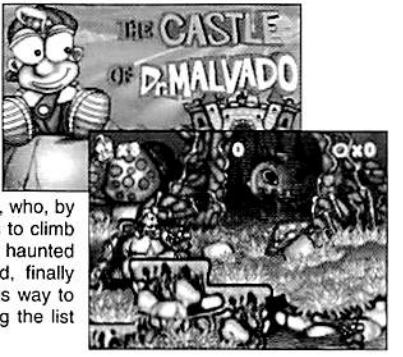
The galactic spaceship Caesar-Julius (the blue triangle) has got lost in a field full of immense fusion asteroids (the drawings of yellow lines) and it must make its way through them by destroying them with its neo-electrons laser-phaser. There have been more versions of this traditional game than what anyone can remember.

To start the game you have to press any key and to exit it the key **ESC**. You have three lives to try and reach the highest possible level (it's really difficult to go further than the fourth level and even to get to this one).

It is controlled by the following keys:

Right: Clockwise rotation.
Left: Counter-clockwise rotation.
Up: Accelerating (to brake you have to accelerate in the other sense and this is more difficult than it should be).
Space: Neo-electrons laser-phaser shot.
H: Hyperspace (it takes the spaceship to another position in the screen when collision with an asteroid is imminent).

THE CASTLE OF DOCTOR MALVADO



The hero of this colourful platform adventure is Jack, a chubby boy who for some reason is very mad at the wicked Doctor Malvado. Jack has to get to him trough multiple obstacles and through beautiful landscapes to quench his thirst for revenge.

The game is divided into three parts. First, Jack has to go through the woods and get inside the castle to battle the son of the Doctor, who, by the way, is even fatter than him. Then, he has to climb one of the castle towers avoiding spiders, haunted pumpkins and other hair-raising objects. And, finally Jack has to fight Malvado and find the devious way to beat him, either by playing a lot or by studying the list of the program to see what he does.

Jack has only two extra lives to get to the end and since they are not at all enough, he has the opportunity of picking up new lives along the way by collecting ten coins from the many which are available.

If you are fond of platform games, you will find this one a fun and especially difficult challenge and if you don't like them we are sure this game will get on your nerves.

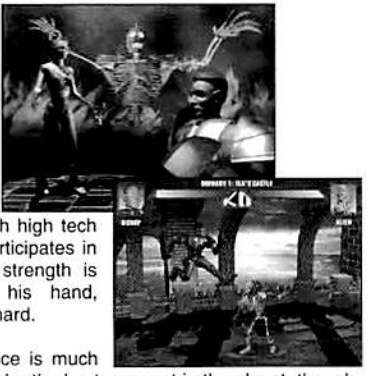
You can play with the joystick or using the following keys:

Left and right: to move Jack to both sides.
Space or Control: Jump.
ESC: Abort the game or exit it.

The hero may go up many objects in the map, for example, the mushrooms; all he needs is patience and ability. Enemies can only be killed by jumping on them, same thing for Doctor Malvado's son. But how do you kill Doctor Malvado himself ?

FOSTIATOR

This is the ultimate combat between the best bionic person and super human warriors. It is a very simple – a typical fighting game but it is great fun and has ample 'gore' (very generous when blood is concerned). It consists of three round fights between two of the following warriors:



Alien: A 612 BC Alien warrior who is now somewhat rusty. He returns to live in 1992 to be in the Olympic Games in Barcelona but they wouldn't allow him to participate and now he is in a very bad mood. His strength is medium-low, but he is very skilled in the use of a claw. Something like good old Freddy which would raise even Casper's uncles' hair.

Bishop: he is the good guy: a bionic warrior with high tech parts. More handsome than Barbie's Ken, he participates in the fights only to get a hard man face. His strength is medium-high and his main weapon the anvil in his hand, ready to make the rest of the fighters' faces look hard.

Ripley: the only girl in the game. Her intelligence is much higher than that of the other fighters. She used to be the best gymnast in the planet, though she also was a waitress (for two months), taught little kids, programmed videogames, etc. She is the least strong one but she is very hard to beat because she uses the sharp Ra baton - excellently.

Nostromo: one of the immortals who has not fought Christopher Lambert yet. He is certainly immortal but he can be knocked out like the rest. He is rude and he bases all his strategy in the blows with his double-edged battle axe. He is the strongest one and this gives him a good chance against the others.

The game features have to be set in the option selection screen by using the **cursors** to select the options and pressing **Enter** to activate them. Up to bottom, the following options can be set:

Control of players. You can select who controls the first player and who controls the second. This way you can select a demo (computer against the computer), a player against the computer or a game for two players.

Level of difficulty. Three levels of games can be established. Even in the most difficult one it is possible to beat all the fighters, of course.

Blood Level. You can omit blood (but the game loses much of its attractiveness) or set the blood level to: normal (which is already a lot of blood) or excessive (you can work it out).

First and second fighter. Any of the four fighters can be selected as the first or second player. You can choose the same one for the first and second player; in this case you will see the first one coloured and the second one in black and white.

Game scenery. You can choose between three scenarios for the battle: the castle, the cave or the desert (this makes the fights more attractive).

The game keys are shown in the control option of the players (the first one). By using these keys you can advance, go back, jump or get down; you also have a blow key which depending on the action will give one blow or another.

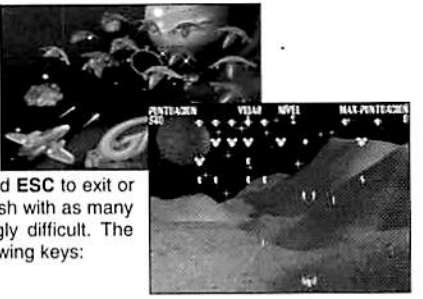
Blow while stopped. Punch (or traditional blow).
Advance and kick. Short and quick kick.
Go back and kick. Super rotating kick.
Jump and kick. Kick in the air.
Crouching punch. A punch while crunched (a low blow).

Other blows can be performed...but first you'll have to program them.

Game strategy
This game is not only about hitting. You have to keep in mind that any time you use the same kick or punch it **loses effectiveness**, either the other fighter gets it or not. Defense is as important as attacking and thus all blows have to be used to get the best out of each fighter and also the same blow or kick mustn't be repeated many times (otherwise it won't do anything to the opponent in the end). Then the best thing is to try to hit the opponent every time we use a blow or else our fighter would get tired for nothing and will be vulnerable.

The easy mode is probably quite easy, but to win in the difficult mode is a much more interesting challenge and so are the two player games.

GALAX



The US President, decorated with a Medal of honour when he was young, has to defend the Earth from the outerspace invaders. He is on board a state of the art fighter. This is a version of the typical space invaders game in which a single aircraft battles hundreds of enemies.

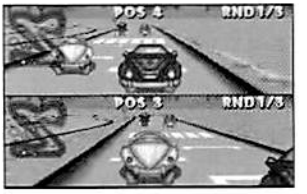
You can press any key to start the game and **ESC** to exit or to abort a game. You have three lives to finish with as many enemies as possible and this is increasingly difficult. The fighter-plane can be controlled with the following keys:

Left and Right: Move the aircraft laterally.
Space: Laser shot.

To increase game difficulty: you cannot shoot again until the previous shot has collided with an enemy or it gets out of the screen. During the game, the maximum score you have got will be shown. We have not been able to go over the fifth level.

SPEED FOR DUMMIES

Probably it is not a very realistic simulation, but under its childish appearance the four ferocious racing cars hide High Horse Power engines.



It is a hard race where second gets nothing (actually the winner doesn't get much either). In the first screen, you can select, with the **cursors** and **Enter**, to start a game with one or two players (the screen divides into two), to set the race options and to exit the game.

The race options which can be defined are the following:

Difficulty Level. You can choose between three levels.
Race scenery. You can choose between a forest or a desert.
Number of laps. It can be 3, 6 or 9 laps to the circuit.

Once in the race, the keys for both players are:

Player 1 - the keys of the cursors.
Player 2 - **R**, **T** to turn and **Q**, **A** to speed up.

To abort a game you can use the key **ESC**. The games with two players will not finish until both cars have crossed the finish line.

When a car tries to get out of the circuit it will bounce and stop. Thus you will never be the winner if you are pressing the accelerator the whole time. The fun of the game is in learning the circuits so that you are able to take each turn as fast as possible. Probably in the first game you don't make a single right turn, but this is normal.

NOID

This is another adaptation of a game that has had many versions. I imagine that whoever had the idea of a game with a ball which goes breaking a wall never thought that his idea could go this far. He probably thought that it was kind of stupid, but he was wrong.

Some smart chappy came up with the fantastic idea to add sense to the game: the wall had to be knocked down because a galaxy depended on it. But this game doesn't need a reason, only skill and even more patience than you need to finish programming videogames once you start.

To start playing all you need is to press to knock down some bricks with the following keys:

Left and Right: to move the cybernetic racket where the ball will bounce (if you can hit it, of course).
Space: to release the ball and shoot when you get the laser racket.
ESC: Finish a game or exit the game.

There are a great number of walls you can tear down, also different kinds of bricks and a series of capsules whose effect must be learnt as you play. The capsules can be good or bad depending on how lucky you get.

Not all the bricks can be broken, so you must not insist on tearing them all them down if you see that after hitting them hundreds of times they are still in the screen.

NOID lacks any strategy whatsoever and thus intelligence is not required to play (how lucky we are).

ALIEN SUPRIMER

Commander Kzygürhypsñü, these are your goals in this mission: "you must...beep...use A.S. (Alien Suprimer, an alien robot to suppress other intelligent or human civilisations)... to destroy a few...bip...inhabitants of the Earth (stop). This is a training mission, exclusively for sport (stop). Have a good hunting trip, commander, and a good time, bip".

If you like action in the style of the A-team, you are going to love this game. Its a vertical space-invaders killer (in this case it is people from Earth that we kill). You face a great number of units belonging to a professional army. A difficult mission, no doubt about it.

Both the options and the game are controlled with the **cursors** and the **Control** key. Several ambushes have to be overcome, each one harder than the one before, until you get to the final enemy, a super giant you will try to beat.
To win in this game, you have to advance slowly and cautiously. If you go too fast, the enemy's ambushes will accumulate and they will make roast chicken out of the poor commander who will leave 13 lonely alien widows.

You have five land to land missiles to complete the mission (they are useless against helicopters). You can shoot the missiles during the game by pressing **ALT**. If you really want to make it all the way to the end you better keep the missiles for the final enemy. The laser shots are endless, so the worst thing that can happen if you use them too much is that you may have to buy another **Control** key.

PUZZLE 'O' MATIC

Solving 35 piece puzzles may be easy even for Nostromo. But if you don't have much time to do it, they can be a challenge even for a videogames programmer.

This game, that gets you addicted like many others, challenges you to solve five puzzles each in a shorter time than the previous one. In the options menu can be used the following keys:

F1: To start a game.
ESC: Exit the game.

Once you have started the first game, you'll have 250 seconds to solve the first puzzle; this can seem like a long time for 35 pieces but I bet you won't be able to do it the first time. To move the pieces, you have to drag them with the **left button of the mouse** and turn them with the **right button.**

When a piece is placed in the right orientation and right next to the right position, the program will take it down to its place. Then you have to worry about the pieces which haven't been placed yet.

Of course the pieces will come in a different orientation and position in each game. For the second level, you'll have 200 seconds plus the time left from the first level, in the third level you'll have 150 seconds and so on.

The game keeps the best times of each game and invites you to beat them in the next games.

Once you are familiar with the shapes and if you concentrate a little bit, it won't be difficult to complete them. If you think solving puzzles is for boring people, this game will change your opinion. Try it.

BLAST'EM UP

The bionic warrior Bishop used to destroy whole groups of enemies in his space fighter. When his craft got too old and he couldn't get a new one, he started to wrestle. Now this horizontal space-invaders killer evokes the space odysseys of your youth.

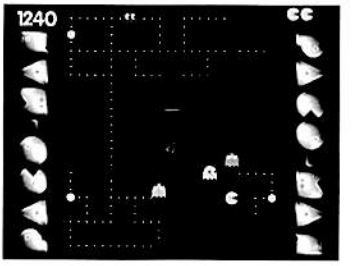
In the title screen you have to press key **1** to start a game. During the game, the spaceship fighter will be controlled with the **cursors** and shots will be made with the **Space** key. You have three lives to advance until you beat the final enemy.

Each time a whole group is eliminated, a capsule with a letter or with a small metal molecule will appear in the screen. If the word 'BONUS' is completed with the capsules of letters, you'll get rewarded with an extra life. And each time you take one of the metal molecules, the power of the shot will be increased.

If the fighter collides with a shot or with an enemy it will be destroyed and will lose one of its lives. When you get a new life, a magnetic field will protect the fighter from the enemies for a few seconds and then it will disappear.

If Bishop was able to do it, we are sure it is easy to do it.

PACO-MAN



Fatty Paco, 38 years old, it's still afraid of the bad guys. And it is not surprising, since there are four bad guys who have something very personal against him. If you don't know this game, you haven't been involved with videogames for very long because this is the best known classic videogame. Paco-man is a conversion of an arcade videogame which was extremely popular a few years ago.

This game gets you hooked with the challenge of eating as many points as possible and in turn trying not to get eaten. These four little ghosts will give you trouble along the 10 levels of increasing difficulty.

There are not spectacular graphics, or shots or explosions. The only weapon Paco has is four magic pills, in the corner of the screen that for a few seconds allow him to eat the ghosts instead of being eaten by them.

The ghosts' intelligence is used to surround Paco and it will increase more and more in each level. You have two extra lives and two more when you get to 10.000 and 50.000 points.

You learn new tricks as you go on playing, for example that when you eat several ghosts one after another you get double score.

You must press **Space** in the initial screen to start playing and you can control the packman with the **cursors**.

If it gets too hard, there is a little trick to apply when the ghosts are getting too close: press the **Space** key and Paco, gathering all the energy he has left will run faster. Without this small advantage it would be quite difficult to survive level 10.

WORLD BOTTLE CAPS CHAMPIONSHIP

The good thing about having a bottle caps race is that any adult that used to play this game in the street (with real bottle caps) when he was a kid can enjoy the game again now and not feel embarrassed by it.

What is a bottle caps race? You design a winding circuit on the sand and compete with other friends to see who takes a bottle cap by hitting it with the finger to the finish line first.

This world championship will be run on an Olympic sand circuit, designed by the best graphic artist at Hammer Technologies. The game is an original mix between a race car game and a golf game.

From the main menu you can use keys from **1** to **3** to select one of the following three options respectively.

Practice. To be able to be good at controlling the bottle cap and learn the circuit. You won't get anywhere without practice or effort, just like in real life.

Compete two players. This is one of the greatest challenges to have fun with a friend. The first one that gets to the finish line will be the winner.

Compete against the computer. if your friends think they have better things to do, leave them. The computer will challenge you to complete the circuit before it does it (and the computer is specially wicked).

The game control is done with the following keys:

Left and Right. Choose the shot angle for the bottle cap.

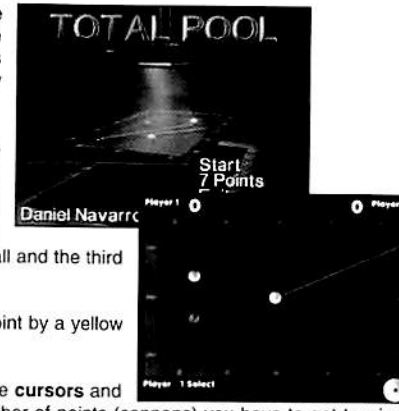
Up and Down. To raise the bottle cap. This way you will be able to make faster turns; it is hard but once you learn how to do it you'll earn plenty of time since you will be able to make the turn with a single move.

Space or Enter. To shoot. The more you keep the key pressed the more power the cap will get. The power is indicated by a score keeper placed in the upper right side of the screen.

When two caps are playing, one against the other, each one will have a turn. At the beginning of the game, who starts first will be determined at random.

If in a shot a cap gets out of the circuit, it will have to go back to its original position as a penalty. So you better get accurate.

TOTAL BILLIARDS



European billiards with an aerial view of the table. Two players compete and each one tries to reach a certain number of cannons before the other. It is not possible to play against the computer in this game.

The game is played in a table with no holes and a cannon is achieved when the ball touches the other two. In the original game, both players use the same white ball, another plain one and another with a point. Both players must try to hit the opponent's ball and the third ball, the red one, in whatever order.

This game replaces the white ball with the point by a yellow ball. The rest remains the same.

In the options menu of the game, by using the **cursors** and **Enter**, or the mouse, you can select the number of points (cannons) you have to get to win or you can start the game.

The first turn is always for the white ball. Pressing the **left button of the mouse** you can use three modes, all of them necessary to make a shot:

Aiming. This mode can set the shot angle by moving the mouse laterally. A blue circle gives you some help, showing where the ball will hit for the first time

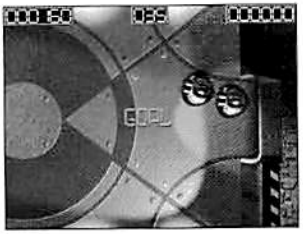
Spin. This mode can select, in the lower right side of the screen, where you want to hit the ball with the cue. To do this you use again the mouse and you may now move it in any direction. If you are able to control the spin of the ball you'll be able to get many cannons.

Shot. This last mode is to make the shot. To do that you have to move the mouse vertically, keeping in mind that the program will detect how hard you have hit according to the vertical speed of the mouse.

In order to adjust the parameters correctly, you may access any of these modes as many times as necessary (pressing the mouse button several times).

You can abort the game by pressing the key **ESC**. This a game that challenges your logic and your skill.

HELIOBALL



Isn't it weird that in the far future people still find it fun to see how two opponents score goals one against the other. The only problem is that they cannot find anybody to run up and down a field: money has disappeared and the professional sportsmen are not going to play for free.

Luckily the multinational company Hammer Technologies, which in this century builds high tech hardware has solved this problem by constructing weightless *hovercraft* machines for mass sports.

Soccer is called Helioball now and two hovercrafts play each other. Now the eleven players of each team are inside their team *hovercraft* cabin, watching the game live on a Hammer panoramic TV (the best one) at the same time they are playing. The *hovercraft* are immense, the eleven players are not packed in there, their cabins are something like a small cinema and even have sauna (so that the players can give the impression of being tired).

This game allows you to travel to the future and control one of this floating monsters which push a mechanic ball (guess what brand it is?) to the goal. The other *hovercraft* will try of course to push the ball to the other goal.

In the title screen, you can press any key to go to the options screen (except **ESC**, which exits the game).

The **mouse** selects options as important as who controls each of the *hovercraft* or the game time (depending if you are in a hurry). If both teams are controlled by the computer, the game will be shown on TV (demo mode) and to switch channels (go back to the menu) you have to press **ESC**. You may have two players with the screen divided. To start the game you have to click over the button start.

The *hovercraft* control keys for both players are:

Player 1. Cursors keys.
Player 2. Q, A up/down and **R, T** left/right.

Everything is legal during the game. The most intelligent hovercraft (the one which scores more goals) will normally win.

SOCCER

This game has to be here. It a very simple adaptation of soccer. There are many rules which are not here: cards, penalty kicks, walls and so on.

Two teams play against each other in this game. You can just watch the match (selecting the computer as control of the first team) or play against the machine using the joystick or the keyboard. A key (or the mouse) must be pressed to access the options screen.

Once in the screen, the program will always be controlled through the mouse. In the screen you may select the type of control, the team colours, the screen resolution as well as the possibility of changing the teams and the players names.

To change the type of control, you have to click on the icon on the upper left corner. By default there is a keyboard. The only other possibilities are the joystick and a demo by the computer.

To change the team's colour, you have to click over that you want to change: socks, shorts or shirt bands.

To modify the screen resolution, you have to press on the bar where the resolution is specified and the program will alternate two possibilities: high resolution (640x480) and low resolution (320x200).

To change the names of the teams and of their players you will access to another screen where you may edit them. The game will save the colours and names set-up for future executions.

There are also two buttons in the lower part, one to each side of the screen. The left button is to go back to the introduction screen (from this screen you may exit the game by pressing **ESC**) and the right button to start a game.

Names editing screen
To change a name you have to click on it. A blinking cursor will appear and you may then introduce data by clicking with the mouse over the keyboard in the lower part or otherwise via keyboard. To end you have to click with the right button of the mouse or press **Enter**.

To exit this screen, you have to click over the button on the lower left side.

Controls during the game
To control the selected player you use the keyboard or the joystick. The keyboard controls are:

Cursors. To move players.
Control. To shoot or tackle another player.
ALT. To pass (with ball possession).

When you use the joystick, the first button will be used to shoot and the second to pass.

For the throws, goal kicks, corner kicks, or center kicks you can use the player you want to pass the ball by using the usual controls for moves.

CHECKOUT

Checkout is a chessboard game, fun even for those people who don't like chessboard games. It is a new game that has been created for DIV Games Studio. So nobody has an advantage when playing.

It is a very simple game which uses artificial intelligence techniques to calculate the moves the computer makes. This makes it one of DIV most advanced examples. So you shouldn't try this list until you are experience enough in the creation of programs.

In a board of 6x6 squares, two teams of pawns each starting from one corner have to try to get to the opposite corner. The games are quite dynamic since they are over in a relatively low number of moves.

The game is divided into 5 levels, each of them with one more pawn at each side. You have to go over through these levels untill you beat the computer in the last level.

You always start the first level with six pawns at each side. Every time you pass a level you go to next but if you lose you have to go back to first level. If you lose in the first level, the game will be over (this will happen at least the first time you play).

The game is always controlled through the **mouse**. In the first screen you can select **start** a game, see the game **instructions** (the rules) or **exit** it.

At the beginning of the game you can choose if you want to play with white or with black. White always starts the game.

To play you select a piece (by clicking on its square) and then to choose where you want to move it to. The computer only allows to select the pieces which can be moved and the moves which are legal according to the rules.

The computer will never play as it did before; it will change its style and he never makes mistakes.

If you like the challenges which require a little bit of concentration, you'll have great fun with this game. You must not get frustrated at first, it's absolutely normal to lose in the first three or four games (or five or six or seven...).

Chapter 2
The Menu System

2

CHAPTER 2: The Menu Sytem

This chapter will describe all the program's options accessible from the main menu, and separated in their different options menus. This chapter has been designed as a text to consult, for the user to obtain quick information about the programs options.

2.1 The programs menu

This menu controls all the issues related to the listing of the programs, their edition and their execution. It allows us to open and save listings, execute a code or edit it.

Programs Menu

The whole list of available keys to edit a program can be seen in **Appendix D**. When there is more than one program window, it will only be possible to edit in the selected window. To select another one, it is necessary to click on it.

Each of the options of this menu is now described.

New...
This option is used to start a new program (to start writing a listing). The first steps are described in Chapter 5. A dialog box, in which the name of the new program must be specified, will appear. Thus, the archive name for the program must be specified (up to eight characters may be used). The extension **PRG** will be automatically added. If an existing archive name is specified, the program will ask you whether you want to overwrite the program. If you answer **Accept**, the existing program will be overwritten. Finally, a new empty edit window will appear, ready for the new program.

Open program... (Direct key: F4)
This option is used to load a program. The name of the archive (with extension **PRG**) containing the program must be specified in a dialog box, that may be selected either by clicking on it or typing its' name. When the program is loaded into the environment, an edit window will appear with the program's listing. The **F10** key may be pressed to execute the program.

Close
Closes the selected edit window. The program will ask for confirmation. If you have modified the program and it has not been saved, choose **Cancel** and press the **F2** key to save the program before closing it, as the program is not automatically saved on closing the window. Therefore, if a listing is modified and then closed without having previously been saved, the changes will be lost.

Save (Direct key: F2)
Saves the contents of the selected edit window in the corresponding archive of extension **PRG** or, in others words, in the indicated archive on creating or opening the program. The mouse pointer will depict an hourglass for a while, to indicate that the task is being performed. If the aim is to save the program with a different name, in another archive, the option **Save as...** of this menu must be used.

Save as...
Saves the selected edit window in another archive (with a different name). A dialog box will be open, specifying the new archive with extension **PRG**. If the chosen program's name already exists, the program will ask for confirmation before overwriting the program with the contents of the edit window. In fact, the extension **PRG** is not obligatory, but advisable.

Edit menu...
This option gives access to a new options menu, the programs' edit menu. The options of this menu are explained in section **2.2** of this book, where the basic commands of the program's editor are shown.

Execute (Direct key : F10)
Compiles and executes the program of the selected edit window. If the program contains any error, the system will place the edit cursor on it, reporting a descriptive message in a dialog with two options:

Compiling A Program

- **Accept**. This button must be clicked on when the error message has been understood, in order to return to the edit window and correct it.

- **Help**. When the error message reported by the system is not understood, then this button will create a help window in which the found error will be explained in great detail.

When **Accept** is clicked on, returning to the program's edit, it is still possible to obtain expanded help about the error by pressing the **F1** key.

When the program has no errors, the system will be able to finish its compilation and execute it. To return from a program, it is always possible to press the **ALT+X** key combination.

Compile (Direct key: F11)
Compiles the program of the selected edit window. The program won't be executed: this option is only used to verify whether a program has any errors. If any error is found, then it will be reported in the same way as in the previous option, being possible to access the expanded help about it, if necessary.

Debug a program (Direct key: F12)
Debugs the program of the selected edit window step by step. To **debug** a program means to execute it little by little, statement by statement or frame by frame, in order to verify all that is being done, how the processes are created, how the variables change their value, etc.

This option is normally used when a program does not work in the expected way, for any reason. Thus, it is possible to find the exact point in which the error of a program is found.

The **Debugger of programs**, described in section **2.10** of this book, will be accessed with this option. It is also possible to access it in a program's ordinary run time by pressing the **F12** key. However, in this case, the program will be debugged from the point in which it has been interrupted, and not from the beginning.

The listing of the executed program will be seen inside the programs debugger, but it won't be possible to modify it. This is an advanced tool, that shouldn't be used until the fundamentals for DIV programming are correctly understood.

Create installation...

The last option of the programs menu is used to create installation disks of the games created in DIV Games Studio automatically. It will be possible to distribute the games with these disks and independently install them in other computers, **without requiring DIV Games Studio for their execution**.

Installation Configuration

About royalties. Important

It will be possible to distribute the games without restrictions, as Hammer Technologies and FastTrak Software **have no royalty or copyright** in this case. Thus, it is possible to use the programs created in DIV Games Studio with any purpose, including the sale of their copyrights to distribution enterprises, direct sale, shareware, freeware, etc.

At the same time, authors are not obliged to include in their games any mention either to DIV Games Studio, to Hammer Technologies or to FastTrak Software. However if you choose to you may include the DIV Games Studio logo or a reference to it.

The archive **DIV32RUN.EXE** is the only DIV component required in the installations. This archive contains the dynamic link library with the internal functions used in the programs. This archive will be always included by the system in the performed installations. **DIV32RUN.EXE** is a **freeware** program, of free distribution, whose lastest version can always be found in DIV Games Studio's WEB page at www.DIV-ARENA.com.

The user is also allowed to create and distribute utilities or auxiliary tools for this program freely. With this purpose, the format of the archives used by DIV Games Studio is shown in **appendix E**.

In order to create the installation program, the system must first compile the program to verify that it does not have any errors and generate the game's executable archive (an archive with extension **EXE**).

Once it has been compiled, a dialog box will appear in which it is possible to configure the installation program. Two switches will appear in the upper part of this dialog box. The one placed to the right allows us to define whether subdirectories must be created in the installation (otherwise, all the game's archives will be installed in a single directory). The one placed to the left defines whether a sound system setup program must be included.

Normally, this setup program is not necessary. If it is included, the system will add the last **SETUP_PROGRAM** compiled in the environment. By default, this program is **SETUP.PRG**, stored in the directory **SETUP** of DIV Games Studio, just in case the aim is to modify or adapt it to a specific game.

In order to customise the installation, after the two upper switches this dialog box will allow us, through a series of text boxes, to input the following information:

Name of the installed program. The name of the archive PRG will appear by default.
Name of the programmer. To input any message of the "Program by ..." type.
Copyright and name of the firm. Any message may be input.
Unit or directory where the installation program must be created. It If it is to be on diskette then it is necessary to prepare enough formatted diskettes as to save the compressed game on them.
Installation directory by default. This is the directory where the game will try to be installed by default, unless the user changes it.
Message that will be used as loading instructions. The system will suggest a proper message that indicates how the game must be executed after its installation.

Finally, the **Accept** and **Cancel** buttons appear to start creating the installation program or to exit the dialog without creating them.

The installations created in this way will use the **Universal installing system** of DIV Games Studio, that is a **freeware** program, of free distribution. If the user wishes to use other installation systems, the best way to do so is to start from a copy of the game installed with this Universal installation system.

In the installed programs, there are no DIV windows reporting errors. At the same time, it is not possible to access the **Programs' debugger** on pressing the **F12** key. If an execution error appears in an installed program, the program will stop, returning to the system.

2.2 The edit menu

This menu has a series of essential options to edit, indicating direct keys in all of them. Therefore, once they have been learnt, it won't be necessary to access this menu.

There are more edit commands together with those shown here. They appear in the section **Programs' editor** in **Appendix D** (keyboard commands).The options included in the edit menu are the following ones:

Edit Menu

Delete line (Direct key: Control+Y)
Deletes the program's line on which the edit cursor is placed.

Tag block (Direct key: ALT+A)
Tags the beginning or the end of a text block. In order to tag a block, it is necessary to tag both ends with this command. The order makes no difference. Tagging blocks of the standard EDIT is also allowed, with the shift key and with the cursors.

Untag (Direct key: ALT+U)
Untags the text block. The blocks selected with the previous command are persistent. That is to say, they remain until they are deleted or untagged. The blocks of the standard EDIT are untagged by themselves, on moving the cursor.

Copy (Direct key: ALT+C)
Copies the tagged text block from the current position. First, it is necessary to tag the original block and place the cursor in the position where it is aim to insert it. A copy of the text will be made, remaining tagged just in case the aim is to make more copies.

Move (Direct key: ALT+M)
Moves the tagged text block to the current position. This task is similar to the previous one, with the proviso that the original text will be deleted after being copied to the current position. The moved text will also remain tagged. Thus, it is possible to continue to move or copy it.

Delete (Direct key: ALT+D)
Deletes the tagged text block. It won't be possible to undo this task. Thus, it must be used carefully. If a big block is accidentally deleted, it is advisable to **Open** the program again (without saving the existing copy), tag the deleted block in the new loaded window, and copy it to the window in which it has been accidentally deleted.

Go to... (Direct key: F5)
Directly goes to one of the program's processes. That is to say, it places the edit cursor of the program's window at the beginning of one of the processes.

On using this command, a dialog box will appear, containing a listing of the processes found in the program. It is possible either to select one through the mouse or to press the **ESC** key to remain in the current position.

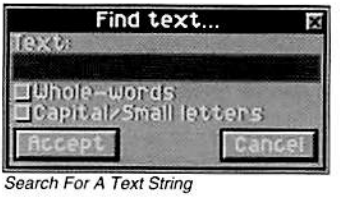
List Of Processes

A switch will allow us to indicate the order in which the list of processes is shown, following either the order of appearance in the program, or the alphabetical order.

To some extent, processes are similar to the functions of other languages, but in DIV they are aimed at controlling the games' graphics (also named sprites). That is to say, they are basically blocks of the program that determine the performance of a graphic, or a kind of graphics, of the game.

Search... (Direct key: ALT+F)

Searches for a text string in the program. This option will be used to locate a point of the program or a name inside it. A dialog box with a text will appear. The sequence of characters that is intended to be found must be inserted in this text.



Search For A Text String

This dialog contains the two following switches to define the type of search that must be performed:

Whole words. Indicates (when activated) that the inserted string must be searched for as a complete word and not as a part of a bigger word (for instance, 'word' must not be found if you write 'wo').

Upper case/lower case letters. Indicates that the text must exactly be found as it has been written, distinguishing between upper case and lower case letters.

In this box, **Accept** must be clicked on in order to start the search for the text. The search will be always carried out from the current position of the edit cursor. Therefore, if the aim is to search for the text through the program, first the cursor will have to be placed at the beginning of it, in line 1 and column 1 (this can be done by pressing **Control+Pg.Up** and then **Home**).

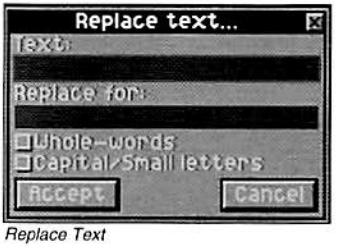
Repeat (Direct key: ALT+N)

Repeats the last search. It searches again for the last text string that was searched for, from the current position. That is to say, it searches for the next match in the program.

When there are no more matches of the searched text in the rest of the program's lines, the system will report it.

Replace... (Direct key: ALT+R)

Replaces a text string by another one in the program. This option can be used, for instance, to replace a name (of a variable, process, etc.) by another one.



Replace Text

A dialog similar to the one used to search for a text will be shown. But in this case, there will be a second text box in which it is necessary to indicate the text that is going to replace the indicated text in the first one.

The two previous switches appear again, fulfilling the same function. In order to replace a name by another one, it is advisable to activate the **Whole words** switch.

The replacement of the text will be carried out from the current position of the edit cursor in the program, ignoring the previous matches. When the text to be replaced is found, before the replacement, the program will offer the following options in a dialog (highlighting the found text in the program's window):

Yes. To replace the text in the highlighted position and continue to search for.
No. Not to replace the highlighted match, but continue to search for.
All. To automatically replace the rest of the matches, including the one that is highlighted currently.
Cancel. To cancel the text replacement's task (no more replacements will be performed).

To press the **ESC** key or to close this dialog will be interpreted by the program as if the aim was to finish the replacement task.

2.3 The palettes menu

This menu controls the tasks dealing with the colour palette that DIV Games Studio is using. It allows us to load and save palette archives (with extension **PAL**), edit it, put it in order, etc.

Palette Menu

The essential concepts about the colour palette are included in chapter 3. In the environment, there is always only one active colour palette. The last option of this menu (**palettes \ show palette**) must be used to display it. It is advisable to have the active palette visible in the environment to get orientated when it comes to performing tasks with it.

It is advisable to define, at the beginning of a new project (game), what is the colour palette to be used as, during the game, all graphics displayed on the same screen must have been created with the same colour palette. Otherwise, they would be displayed incorrectly.

Open palette...
Loads the colour palette from an archive. On activating it, a dialog box will be opened, in which it is necessary to indicate an archive of one of the following types: palettes (**PAL**), graphics files (**FPG**), bit maps (**MAP, PCX** or **BMP**) or letter fonts (**FNT**). If there are graphics loaded in the desktop of DIV Games Studio, the system will ask whether they must be adapted to the new loaded palette. If the answer is **Accept**, they can lose quality. Therefore, the alternative is to close them first and reopen them later.

Save palette...
This option allows us to save an archive with the current colour. A dialog box will be opened. Inside it, it is necessary to indicate the archive name in which the palette is going to be saved (for that, archives with the extension **PAL** will be used). It is not possible to save the colour palette in graphics files, maps or fonts. Moreover, information about the colour rulers defined inside the graphic editor is stored in the palette archives .

Edit palette...

On choosing this option, a dialog box with the palette editor will be opened. Inside it, it is possible either to modify the current palette of the system or to define a new one. While the palette is edited, the colours of the desktop may appear incorrect. This problem is normal and it will be resolved on exiting this dialog box.

All the colours of the active palette appear occupying the greatest part of the box. A small square will tag one of these colours; this is the colour selected for editing. In order to select another colour, click on it. The following information about the chosen colour appears in the right part of the box:

Palette Editor

Number of the colour's order within the palette. This number appears as a decimal between 0 and 255, and as a hexadecimal, between 00 and FF.

The **red, green and blue components** of the colour, which will be numbered from 0 to 63.

Three **vertical scroll bars**, that can be used to modify the three components of the colour, in the same order as in the previous case.

Three switches with the following function appear in the lower part of the box:

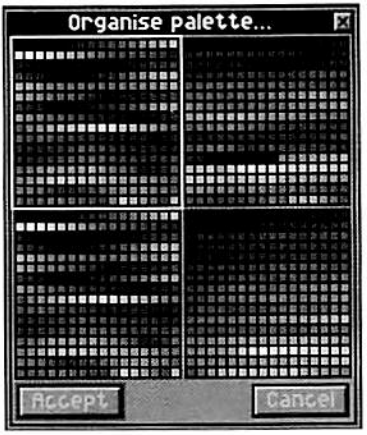
Range: It is used to define a gradual colours' range. It is necessary to press on the first colour of the range and define it with the scroll bars. Then, do the same with the last colour of the range and finally (with the last chosen colour) select this switch and click again on the first colour. The system will define the intermediate colours with a gradual range.

Copy: To copy a colour to another position of the palette, it is necessary first to choose the colour, then this switch and, finally, the destination position.

Change: It works in a similar way to the copy function, with the proviso that it will only exchange the origin and destination colours, instead of copying the origin colour to the destination one.

Finally, three buttons allow us, from left to right, to validate the performed palette edit, undo the last modification and cancel the edit (undo all the changes made since the editor was entered).

Arrange palette...
Allows us to arrange colour palettes. This option is useful to edit maps coming from other graphic or rendering programs, and having a colour palette out of order. This task will facilitate these maps to work inside the graphic editor, making it easier to locate the colours. A dialog box will show four possible orders of the palette's colours. The order considered more appropriate must be clicked on. This dialog box finishes with two buttons that allow us to confirm this selection or cancel it.



Palette Organiser

Merge palettes...
This is an advanced task, that allows us to create one colour palette from two. It must be used when the aim is to use graphics with different palettes simultaneously in a game. First, one of the palettes must be loaded in the environment and then, this option must be used to select an archive with the other one.

The system will create a mixed palette from both. After this task, the best thing to do is save this palette in a palette archive (**PAL**) to have it located and not to lose it. After that, the graphics must be loaded again with the previous colour palettes, indicating that their palettes must not be loaded, but they must be adapted to the palette of the system. Finally, once it has been verified that the graphics of both palettes look good with the new palette, they are saved again in their respective files.

Display palette
Displays the active colour palette in the environment. This option will create a new window in the desktop, where it will be permanently displayed. Only one active colour palette can exist at each moment.

Display Palette

2.4 The maps menu

This menu controls the tasks related to the maps. The maps, also called graphics or bitmaps, are simply frames of any size that can be used in the games as backgrounds, sprites, objects, screens, etc. This menu gives access to the loading and saving of maps in the disk, as well as to the graphic editor, which is the tool used to create or modify these frames. Many maps can be handled, loaded in the desktop, etc. The only limitation is that all of them must use the same colour palette. If two maps have to use different palettes, one of them must be closed before opening the other.

Maps Menu

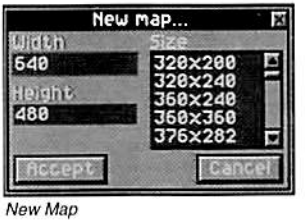
Besides the options of this menu, it is possible to perform some other tasks with the maps windows. In order to drag

them, it is necessary to click on them with the left mouse button and, without releasing it, to move the mouse to the destination position. These tasks are the following ones:

Make a copy of the map. Drag the window to the desktop's wallpaper (to a screen zone in which there is no window).
Paste a map on another one. Drag a map window to another one. The graphic editor will be entered, being then possible to paste the map in the desired position and conditions.
Close a map. Drag the window to the bin (first, the bin must be activated with the option **system \ bin**). The system will close the map without asking for confirmation.
Include a map in a graphics' file. Drag the map window to the file window.

Several options of this menu only interact with one map, even if there are a lot of them loaded. The action will be performed on the map selected among all of them. In order to select a map, the left mouse button must be clicked on it.

New...
Creates a new window with a graphic map. The only necessary information to create a new map is to know its width and height in pixels, which will be requested in a dialog box. Any size, from 1x1, is valid with the only limitation of the available memory in the system. After having selected the size and clicked on **Accept**, a new map window will appear empty (in black) in the desktop. In order to edit the contents of this window, it is possible to double-click on it with the left mouse button.



New Map

Open map...
Loads a map from a disk archive. The archives of extension **MAP** are the own format of DIV for the maps but, with this option, other archives with extension **PCX** and **BMP** can be imported, providing that they are frames in **256 colours**.

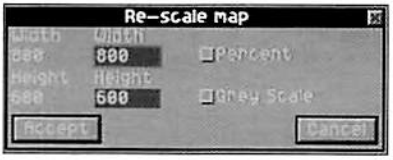
Close
Closes a window with a map. The system will ask for confirmation before closing a window with a map. If the aim is to save the contents of the window, the **Save...** option must previously be used in order to update the disk archive that contains the frame. Once a map has been closed, the only way to recover it is to load the archive again (if it has been saved previously).

Close all
This option closes all the map windows loaded in the desktop, **except those that have been minimised**. Previously, the system will ask for confirmation. This option is useful when the desktop is filled with maps that have been included in a graphics file, or maps of the generator of explosions, to prevent them from having to be closed individually.

Save
Saves the current contents of the selected map in its archive. The mouse pointer will momentaneously depict a hourglass, to indicate that the task is being performed. If the map had no previous archive, the system will show a dialog box in which, a name for the new archive must be specified. The **Save as...** option of this menu must be used to save a map in another archive.

Save as...
This option allows us to save the contents of the selected map window in a new disk archive. A dialog box, in which the archive name must be specified, will appear. Archives **MAP, PCX** or **BMP** can be saved, depending on the extension given to the archive name. By default, archives **MAP**, the own format of **DIV**, will be saved. Only this format can be used in the programs created in **DIV** and it is not possible to load frames of archives **PCX** or **BMP** in them.

Re-scale...
Re-scales the selected map to a new size. That is to say, increases or reduces the frame's size. The new size must be included in a dialog box that will appear with two switches:



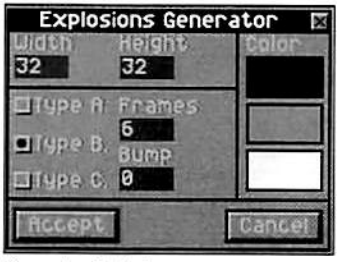
Re-Scale A Map

Percentages. By default, the new size will be specified in pixels. But it can be indicated in percentage, as a integer, if this switch is activated. For instance, 100 is the original size, 50 is half and 200 is double.
Ranges of gray. The new copy of the map will be created in colour by default. However, this switch may be activated to create it in black and white.

The rescaling data for the horizontal and vertical axes must be included in the texts boxes **Width** and **Height**. **Accept** must be clicked on after the indication of these values, and the system will create the new map.

Edit map
Edits the selected map. The user will directly enter the graphic editor. This task can also be performed by double-clicking on the map window with the left mouse button. The right mouse button or the **ESC** key must be pressed to exit this editor.

Explosions generator
This option gives access to a dialog box to automatically generate a series of frames for an explosion. Each of the frames will be created in an independent map. The maps will be created with the colour palette active in the environment. Later, if the aim is not to preserve the created maps, it can be advisable to use the **Close all** option to eliminate them.



Generates Explosions

Two text boxes appear in the upper part of the dialog. The **Width** and **Height** of the maps to be generated must be included in pixels in both boxes. The greater these maps are, the longer the generator will take to complete the sequence of frames producing the explosion.

By clicking on the boxes appearing below the text **Colour** a dialog will be displayed with the colour palette, and thus, it will be possible to select the initial (the darkest one), intermediate and final (the lightest one) colours of the explosion, respectively. The generator will use as many intermediate colours as are found in the active palette.

Three switches named **Type A**, **Type B** and **Type C** allow us to choose among three different algorithms to generate the explosion, in order of complexity: the first type is the easiest and more homogeneous one, and the last type is the most complex one.

Next to these switches, two other text boxes appear. The number of frames that the complete sequence of the explosion must comprise, have to be included in the first one, named **Frames**. The second text box defines the effect of **Bump** (Granulation) applied to the explosion. The higher this value is, the louder the "noise" or "vibration" of the explosion will be.

Each time an explosion is generated, it will be different, even if exactly the same parameters are included in this dialog box. For that reason, it is advisable to carry out several tests until the definitive sequence is obtained.

The possibilities of animation of the graphic editor can be used to display the animated explosion. For that, it will be necessary to arrange the windows containing the explosion frames (placing them one on each other, in order), editing the first one (double-clicking on it) and, once the editor has been entered, use the **TAB** and **Shift +TAB** keys to see the animation. The **Z** key can be used to vary the scale at which the animation is displayed and **ESC** to exit the graphic editor.

2.5 The files menu

This menu controls the tasks performed with the graphics files, such as to create new files, or to load (open) their maps. The **graphics files FPG** are archives containing libraries or complete collections of maps.

Files Menu

The only actions of files that can not be performed through this menu are the following ones:

Add maps to a file, This can be done by dragging the maps windows to the file's window.
Copy maps from a file to another one. For that, they must be dragged from the list of a file to the list of the other one.

Files are not loaded in the computer's memory, as they normally occupy a lot of space. On the contrary, their information always remains in the **archives with extension FPG** of the disk that contain them. Therefore, there is no option to save the files, as **they are always stored**; that is to say, what is displayed on screen is exactly the same as what is stored in the archive of the disk.

Important: All the maps of a file must use **the same colour palette.** Those maps using a different palette must be adapted to the palette of the file, in order to be included in it.

New...
Creates a new file FPG on the disk to store graphics maps with the active palette in the environment. The archive name must be specified in a dialog box. The system will

automatically add the extension **FPG** to it. If the name of an existing file is specified, it will be asked whether the aim is to overwrite it. If the answer is yes, all the maps previously stored in the file will be deleted.

Open file...
Opens a new window from a FPG file. The name of an archive with extension FPG must be selected in the dialog box of archives managing. If the file has a palette different from the one that is active in the environment at that moment, as options the system will offer either to load the new palette or to close the file. It is also possible to open the file without loading its palette but it is not recommended, unless you are an expert user.

Close
Closes the selected file window. As the contents of the files is always stored on disk, their information (the maps) won't ever be lost. For that reason, the system won't ask for confirmation in order to close a file, as it can always be recovered by opening it again.

Save as...
This option allows us to save the selected file with a different name (in another archive FPG). That is to say, a copy of the file with a new name will be made. A dialog box will appear. The name of the new archive FPG must be included in this dialog box. Once again, if the name of an existing archive is indicated, its contents will be replaced by the contents of the file selected in the environment.

Load tags
Loads the maps tagged in the selected file in the desktop. On using this option, the switch **Tag/Drag** of the window's file must be tagged, being thus possible to previously tag the maps intended to be loaded (opened) from the list included in the file (by clicking on them with the mouse). It is possible to untag maps by clicking on them again. As many maps windows as tagged in the file will be created. This option is useful to modify these maps in the graphic editor and then, to drag them again to the file.

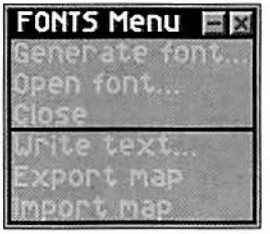
In order to load maps individually, one by one, they can directly be dragged from the file window to the desktop.

Delete tags
Deletes the tagged maps in the selected file. This option is used to eliminate maps from a file when they are not necessary any longer. The maps have to be tagged previously, exactly the same as in the previous option. Once these maps have been deleted, it won't be possible to recover them.

To individually delete maps, they can be dragged from the file window to the system bin (which is shown through the option **system \ bin**).

2.6 The fonts menu

This menu allows us to perform actions related to the letter fonts of the games (or letter types), including the access to the generator of fonts used to create new fonts. The letter fonts are used inside the graphic editor and in the games, to write texts.



Fonts Menu

At the same time, the fonts are shown in small windows that indicate the set of characters they have defined. The characters are divided into the following groups: numeric digits, upper case letters, lower case letters, symbols and expanded characters.

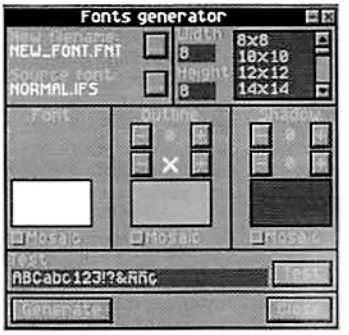
To visualise a sample of the characters contained in a font, it is necessary to click on its window with the left mouse button.

The fonts are saved in the archives disk **FNT** and adapted to the palette of the environment when it is loaded and when the active palette is changed. In order to see a font with its original palette, the **palettes \ open...** option must be used, indicating the name of the archive FNT.

The fonts windows always correspond with a FNT archive. Therefore, there is no option to save the fonts, as they **are always saved**. That is to say, what is shown on screen is exactly what is saved in the disk archive.

Generate font...

This option will create a new window with the generator of letters fonts. New types of letter for the game can easily and quickly be created with this tool. The following steps must be followed to generate a new font:



Font Generator

- First, select the new archive FNT in which the created letter font must be saved. For that, it is necessary to click on **New font**, in the button marked ellipsis. A dialog box will appear to include the archive name. By default, the font will be generated in the archive **NEWFNT.FNT** valid to carry out tests.

- Second, choose the source font. A dialog will be displayed by clicking on **Source**, in order to select the font type. One of the archives with extension **IFS** of the list must be selected. A sample can be seen in the dialog itself by clicking on any of them. Once the desired type has been found, the **Accept** button must be clicked on.

- In the upper right part of the box, two input text boxes allow us to define the size of the font characters. Their **Width** and **Height** must be defined as any number ranging from 16 to 128. For sizes smaller than 16x16, only the numbers shown in the adjoining list (8x8, 10x10, 12x12 or 14x14) can be used. They can be selected on clicking on the list.

For fonts bigger than 16x16, two different values for width and height can be indicated, becoming distorted the original appearance.

- In the section **Font** (left central) either to select the ink colour for the font by clicking on the colour box or **to drag a map with the texture** to this box. If a texture is in this way defined as a filling of the letters font, it will be possible to select in the **Mosaic** switch whether this texture must cyclically be repeated. By default (without the switch activated) the texture will be re-scaled to the size of the font characters. That is to say, in order to use a texture in a font, it is first necessary to open or create the map with the texture and then, to drag it to the fonts generator.

- In the section **Outline** (middle central) it is necessary to define whether an outline must be created in the fonts characters. If the answer is yes, the outline's width in pixels must first be specified, by using the buttons - and +. Next, the direction of the outline's lighting can be specified where a **X** appears between other two buttons - and + (the symbol **X** indicates that the outline is not going to be lighted). Finally, the colour or texture of the outline will be specified in the same way as in the previous section.

- In the section **Shadow** (right central), the shadow of the characters is defined. First, it is necessary to indicate its horizontal shift (in pixels), keeping in mind that the positive numbers are to the right, and then the vertical shift (now, the positive numbers are in the lower part). If both shifts are left at zero, no shadow will be created. Later, it is necessary to choose the colour or texture for the shadows again. A dark colour is normally specified as the filling of the shadows, but **not the first colour of the palette**, as this is the transparent colour and consequently, it is not displayed.

- With the button called **Test**, it is possible to see a sample of the font at any moment. Thus, the previous parameters can be adjusted better. The text for the demonstration can be input in the text box next to this button.

- Finally, the **Generate** button must be clicked for the system to create the new characters font in colour in the indicated **archive FNT.** Then, a confirmation dialog will appear. In it, it is necessary to indicate which are the five sets of characters that must be included in the font, activating the corresponding **switches**. All the switches must be activated to create a complete font. For instance, if only the numeric digits of the font are going to be used, it is better to create it only with these characters, as the archive FNT will thus occupy less space on the disk and in the computer's memory.

Once the process has finished, a new font window will appear in the environment's desktop The font generator remains in the desktop until its window is closed.

The button **Close** located in its lower right corner can be clicked to close the generator or cancel the process to create a font. However, if the aim is to use it later, it will be better to minimise the generator's window. Thus, the information inserted in it up to then won't be lost.

Open font...
This option allows us to open a FNT archive with a characters font previously created to perform some tasks with it or to use it in the graphic editor. A dialog box to manage archives will allow us to select the archive. Then, a new font window will be created in the desktop. A sample of the font may be displayed by clicking on this window.

Close
Closes the selected font window. Confirmation won't be asked for, as the contents of these windows is always updated in the disk. Therefore, the font can always be recovered by loading the FNT archive again. Inside the graphic editor, in the option labeled write text, the font of the programs editor will be used when there is not any font in the desktop of the environment.

Write text...
Creates a new map with a text written with the selected font. A dialog box will appear. The text that must contain the graphic map has to be inserted in this dialog. The system will create a new map window with this written text. This option won't be available if there is no selected font window in the desktop. This option is useful when the aim is to use a characters' font in a game only to display one or two messages, as the graphics with these texts will occupy less space than the complete characters font.

Write Text

Export map
The options of exporting and importing a font from a map will allow us to manually retouch the fonts created with the generator in the graphic editor. This option will create a new map window with all the characters of the selected font. Then, to edit the characters of the font, it is necessary to double-click on the map window with the left mouse button, entering the editor. It is very important to respect the external margins of the characters. That is to say, each character can be retouched, but respecting its limits. The characters not included in the font will be represented in this map with a pixel of colour 0 (the first colour of the palette, that is to say, the transparent colour).

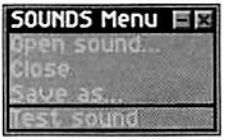
Import map
This option allows us to obtain again the **archive FNT** from the map with the characters. If the map is not a font with all its characters, this option won't be able to obtain the archive FNT from it.

The maps with fonts have a format that must be respected, with a colour defining the external margins (any colour not used in the font may be chosen) and 256 squared boxes with the characters defined in the font, arranged according to their ASCII code, indicating the undefined characters as 1 by 1 pixel boxes.

If, during the characters' edit in the graphic editor their margins have been modified, this option won't be able to obtain the font again. The system verifies that there is an external colour in this map, and inside it, 256 boxes arranged from left to right, with a margin between them of at least one pixel (of the external colour).

If these rules are observed, even letter fonts may be created from zero in the graphic editor in order to import them later with this option.

2.7 The sounds menu



Sounds Menu

This menu allows us to control the sound effects' windows and perform the essential tasks with them, such as loading effects, saving them with a different name or listening to them.

The sound windows always correspond with an archive PCM of the disk. Therefore, there is no option to save the sounds, as they **are always saved**. That is to say, what is shown on screen is exactly what is stored in the disk archive.

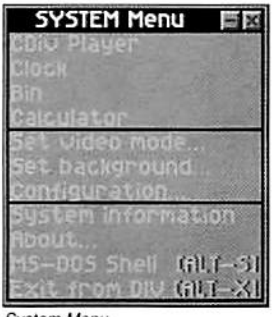
Open sound...
Loads a sound effect from a disk archive. A dialog box will allow us to indicate the name of an archive with extension **PCM** or **WAV**. However, when an archive **WAV** is loaded, the system will automatically create another archive with the same name but with extension (and format) **PCM**, as it is the only allowed format for the sound windows. Then, the new sound window showing the contents of the archive **PCM**, to which it represents, will be created.

Close
Closes the selected sound effect. Confirmation won't be asked for, as the contents of the window is always updated in the disk archive (the sound effects can not be edited in this first version of the program). For that reason, this window can always be recovered by reopening the disk archive.

Save as...
Saves the selected sound effect in another archive, with another name. A new dialog box, in which the name of the new archive PCM must be input, will appear. It won't be possible to export sounds as WAV files. If the name of an existing archive is specified, the system will ask for confirmation before overwriting the contents of this archive.

Test sound
Emits the sound effect through the computers' audio system. Then, the sound effect selected in the desktop will be heard. This action can also be performed by clicking on the sound window.

2.8 The system menu



System Menu

Actions dealing with DIV's windows' environment are controlled from this menu. Thus, it is possible from this menu to configure their appearance (videomode, fonts, colours, wallpaper, etc.), access auxiliary utilities, ask for information about the system or exit it.

CDiv Player

This option will show the CD-Audio Player on screen. The reproduction of music CDs in the computer's CD-ROM driver can be controlled with this tool. Information about the selected song and the time is shown in the upper part of the CD Player's window. The control buttons, from left to right, allow us: to skip back the previous song, rewind, stop, play, fast forward and jump ahead to the next song.

CDiv Player

The reproduction of a song will go on until the end of the CD. If the CD Player window is closed, once the reproduction has started, it won't stop. However, the CD Player can also be minimised.

Clock

This option will display a clock showing the current time on screen. This clock will appear in a window that can be left active all the time, dragged to another position or closed. If the window is minimised, the text of its icon will continue to show the updated time.

Clock

Bin

This option will show the bin window in the desktop. The bin is used to delete maps windows or files windows' maps dragged to it. Confirmation won't be asked before deleting the maps dragged to the window.

Videomode... / Wallpaper... / Configuration

These three options allow us to configure the appearance of the windows' graphic environment, and they are explained in section 1.3 of the first chapter in this book.

System information

This option will show information about the resources available in the system in a dialog box. This information will be related to the available free memory and the occupied memory by the loaded map windows. It will also indicate, in percentage of resources, how many objects, maps and windows of any kind are loaded in the desktop out of the total that can be loaded.

Information About The System

About ...

This option shows the introductory dialog box containing the information about the current version of the program.

Shell MS-DOS (Direct key: ALT+S)

Executes a session of the MS-DOS operative system, without exiting DIV. It will be possible to use commands of this system and programs that only require base memory. **EXIT** must be keyed to finish the session of this system.

Exit DIV (Direct key: ALT+X)
Exit the DIV Games Studio's environment. The system will ask for confirmation before exiting definitively. By default, the system will save the contents of the desktop for following executions of the environment. Therefore, **information** that has not been saved before exiting **won't be lost**.

Exit Screen

2.9 Help option

The index of this help hypertext of the environment can be accessed from the main menu by selecting the last option.

The help windows are controlled with the mouse, by using the vertical scrolling bar placed to its right to move through each of the help pages. For that, the following keys can also be used:

Up / down cursor. To shift the text one line up or down.
Pg.Up / Pg.Dn. To shift the help text page by page.
Backspace. To go to the previous help page (this key is used to delete the previous character).

The text is shown in the help windows in three colours:

Black. This is the main part of the help. Most of the text is shown in black in the help pages.
Gray. Those texts aimed to be enhanced as they contain specially important information (such as the **bold typed** text of a book) will appear in gray colour.
White. The texts or words referred to concepts or terms explained in another help page will appear in this colour. To access this page, it is possible to click on them, and then, to use the **Backspace** key to return to the previous page's point in which the reading was stopped.

Besides texts, **frames** or **examples** may appear in the help windows. The examples are programs starting with a **blank** line and containing **dark blue** texts. To extract the examples from the help windows, it is necessary to click on their initial blank line. Then, they can be compiled and executed by pressing the **F10** key and to abort their execution, it is necessary to press **ALT+X**.

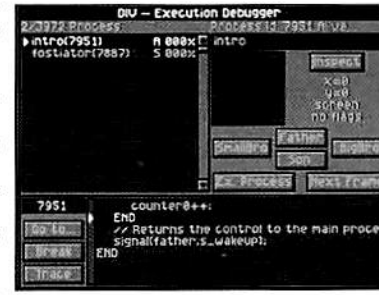
A plain button appears in the lower right corner of the help window. This button can be used to expand or reduce the window's size, by clicking on it and dragging it up or down.

2.10 Programs debugger

The programs debugger is an advanced tool, whose use first requires a correct understanding of all the programming concepts explained from chapter number 5 of this book.

The debugger is a dialog box that can be activated in programs' run time for one of the following reasons:

- The program was entered with the option **programs \ debug program**.
- The **F12** key was pressed in a program's run time.
- An **error of execution** arose in the program.
- A **debug** statement was found in the executed program.



Programs Debugger

This tool allows us to execute the program statement by statement, verifying the value taken by the different program's data when necessary. It is useful because, on checking the program's execution step by step, it can find the mistakes eventually made by the programs.

As it is a dialog with a great deal of information, each of its sections are now described separately.

Upper information line

Two messages are reported in the upper part of the window. To the left, there is one indicating the **number of processes active in the program** out of the **total that can be created**. For instance, if it reports 23/3201, it means that there are 23 processes active in the program and that up to 3201 could be created before using up the available memory for processes.

The maximum number of processes vary from some programs to others, depending on the number of their local and private variables.

The **identifying code** of the process selected in the list, as well as its current state (normal, killed, asleep or frozen) are indicated to the right.

List of active processes

This list appears in the upper left part of the debugger with a scrolling bar to its right. All the active processes in the program are shown in it. Active processes are all those processes that have been created and that still have not been disappeared. The following information appears for each process:

The process name in the program.
Its identifying code in brackets (occasionally, there is no space to put it entirely).
A letter indicating its state (**A**-Normal, **K**-Killed, **S**-Asleep and **F**-Frozen).
The percentage of accomplished execution for the following frame.
The scrolling bar must be used to move through the processes' list.

Important: One of the processes appears with a white tip arrow pointing out its name. This is the process that **is being executed** in the program currently. Therefore, the next statement of the program will belong to this process.

One of the processes appears tagged with a black band. This is the process about which information is shown in the right part of the window (close to this list of processes). This process may be selected with the mouse, by clicking on the list.

It is very important to distinguish between the process in execution and the process about which the information is shown, as they don't have to be equal necessarily. For information about the process in execution to be shown, it is necessary to select it (that of the white arrow) by clicking on the list with the left mouse button.

Information box about the indicated process
To the right of the previous list, information about the program tagged with a black band in the list (not the process in execution) is shown. Its identifying code and its state was shown in the uppermost line.

The complete process name is shown in a dark background's box. Below it there is another box with the graphic of this process (when it is bigger, it will be reduced to fit this box).

The button **See data** appears to the right of the graphic. This button allows us to access another dialog box in which all the data of the process can be consulted and modified. It will be later explained in the section **Inspecting data**.

The (**x**,**y**) coordinates of the process, the system of coordinates used by it (referred to the **screen**, to a **scroll** or to a **mode 7**) and the mirrors or transparencies applied to the graphic of the process are always shown following this button.

Finally, four buttons allow us to access the **father** process (the process that called the selected one), the **son** process (last process called by the selected one), the younger brother (**smallbro**, the last one called by the father before it) and the elder brother (**bigbro**, the following one called by the father after it). If these buttons don't lead to any other process, that is because there is no process with that relationship.

Partial execution controls
Two buttons called ***Exec.Process*** and ***Next FRAME*** below the previous information box allow us to execute the program partially.

Execution of the process. This first button allows the program to continue just to the end of the process currently under execution (the one pointed with the white arrow in the list). All its statements will be executed until it reaches the next **FRAME** (until the process is ready for the next frame of the game).

Next FRAME. The second button will execute the program to its next frame, first executing all the pending processes and displaying the next frame of the game (in the debugger's background). The debugger will stop in the first statement of the first process to be executed in the new frame. It is possible to displace the dialog box with the debugger (by dragging its title bar) in order to contemplate the result of the previous frame of the game.

Debugging box of the program's listing

The code of the program is shown in the lower part of the debugger. The identifying code of the process under execution (again, the one pointed with the white arrow in the list) appears in the left upper corner. Below it, there are three buttons and, to its right, the code window.

In the **code window**, another white arrow indicates the line including the next statement to be executed by the process. It can be noticed how the statement also appears highlighted in white from the rest of the code.

This window's contents can be displaced with the **cursors** keys. The program's lines can be tagged with a black band. Nevertheless, it is not possible to modify the program from the debugger. Indeed, to modify the program it is necessary to finish its execution (which can be done by pressing **ALT+X**) and return to the editor of the environment.

The first button called **Process** allows us to go in the code window to one of the processes of the program directly. A list containing all the processes found in the program will appear, being necessary to select the desired process with the mouse. However, it won't change the process currently under execution, which will continue to be the same.

The second button allows us to establish a **Breakpoint** in the program. For that, it is first necessary to tag the line of the listing with the black band. On reaching this line (with the cursor), the program must stop. Then, this button must be activated making the line appear in red.

Breakpoints can not be established in all the lines of the program, but only in those for which the executable code has been generated (in which any action is performed).

Many breakpoints can be established in the program. To execute the program until it reaches one of these points, suffice will be to close the debugger or press the **ESC** key.

To **disable** a breakpoint, it is necessary to select the line and click on the same button again.

The last button, **Debug**, is the one that really allows us to debug the program statement by statement. Every time it is clicked on, one of the program's statements will be executed. When a process finishes its execution, or completes a frame, you will pass to the first statement to be executed of the next process.

Inspecting data

By clicking on the button **Inspect** of the programs' debugger it is possible to access this other dialog box, in which the values of the program's data can be consulted (and even modified) in the point in which it has stopped, normally with the aim of carrying out tests in it.

Dialog Box To See The Data

Most of this box is occupied by the data list. Each of them is shown with its name and numeric value. This list always appears in alphabetical order.

The data set appearing in this list can be selected through a series of switches. The two upper switches define the two following sets.

Predefined. When this switch is activated, all the data predefined in the language will be included in the list. Thus, it will be possible to access the predefined local data (such as *x, y, angle, size* ...), the predefined global data and the predefined constants.

Defined by the user. This switch selects all the new data defined in the program. These are the specific constants, variables, tables and structures of every program.

Besides selecting the data depending on whether they are predefined or new, they can be selected according to the context in which they have been declared, with the following switches.

CONST. This switch is used to include the constants in the list, even if the constants are not data, but synonymous of a numeric value. Therefore, they can not be modified.

GLOBAL. On activating this switch, all the global data (accessible by all the processes) will be included in the list.

LOCAL. When this switch is activated, the local data (that is to say, the data that all the processes of the program have) will be included in the list.

PRIVATE. This switch selects the specific data of the process tagged in the debugger window to include them in the list. These data are exclusively for the program's internal use.

The list of data can be displaced with the vertical scrolling bar or with the **cursors** and **Pg.Up / Pg.Dn** keys.

The button **Change** allows us to modify the value of the selected data; only the constants can not be modified. A new dialog will appear with a text box in which the new value of the datum must be input. Any datum of the list can be selected with the **cursors** or clicking on it.

Below this button, there are other two buttons with the symbols - and +. They are used to **modify the index of tables and structures**, which can also be done with the **right cursor** and **left cursor** keys. The table or structure whose index is intended to change must previously be selected in the list. This is the way to observe or modify any element of a table or structure

Finally, a series of buttons appears in the lower part of this dialog. These buttons, mentioned below, allow us to display the value of a datum in a specific way:

Angle. This button allows us to display the datum as an angle. The angles are specified internally (in the programs) in degree thousandths. The value of the datum will be displayed as an angle in degrees and radians.

Process. If the datum is the **identifying code** of a process, on selecting this display filter, the name of this process will appear in the list as a value of the datum.

Text. When the datum is a text or a pointer oriented to a text (to a literal of the program), that text will be displayed in the list by clicking on this button.

Boolean. If a datum contains a logical value, on applying this filter to it, in the upper list will be shown whether it is ***false*** or ***true***. In the language, on evaluating them as logical conditions, the odd numbers are considered true, and the even numbers are considered false.

Once the display filter of a datum has been established, it will remain during the rest of the program's execution. The same button must be double-clicked to display again the contents of the datum as a numeric value.

Chapter 3
Graphics Editor

3

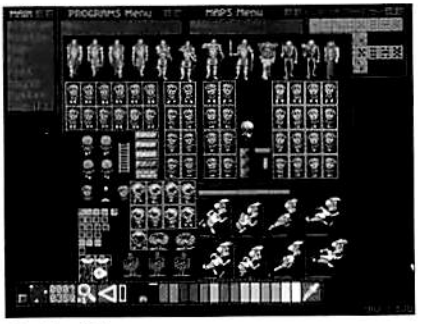
CHAPTER 3: The Graphic Editor. First Steps

3.1 General Concepts

The graphic editor is described in chapters 3 and 4. This tool is used to create the graphics of the games. The first chapter offers a program's generic view and explains all the concepts and terms necessary to understand how it works.

The graphic editor is the tool used to paint the graphics of the games. Any picture may be created with the mouse, the colours of a palette and a little skill. Painting with the computer is quite different from painting on a sheet of paper. Less accuracy and skill, but more patience are required.

The editor only works with maps windows (those that any kind of graphic contains), and the easiest way to enter it is by double-clicking on a desktop's map with the left mouse button. Once the editor has been entered, the picture is expanded, the mouse pointer will change and a horizontal tool bar will appear with several icons and colours. The editor can be exited in different ways: by closing the tool bar, by pressing the **ESC** key or with the right mouse button.



Graphics Editor

This graphic editor is a very comprehensive tool. However, it can not replace many of the commercial programs specifically designed to create graphics. For that reason, it is possible to import maps in **PCX** or **BMP** formats (that are widely spread among the graphic utilities). Thus, those who want to use other tools to create graphics are allowed to use the frames created with other programs in DIV Games Studio.

Despite that, it is advisable to read these chapters on the graphic editor, as many tasks will be easier and faster from DIV Games Studio than from other programs.

If you do not have other programs to create frames, do not worry, as DIV's graphic editor contains many powerful paint tools. The features shared by many of them are better than those of specific paint programs. Thus, it won't be necessary to use any other program to create the games' frames and graphics.

Before accessing this chapter, which describes the basic painting concepts and terms, it is necessary to know the use of maps and maps files described in sections 2.4 and 2.5.

For instance, to carry out the first tests, load the colour palette by default (**palettes \ open...** indicating the archive **div.pal**) and create a new map (**maps \ new...**) of any size in pixels, such as **200x200**, and access the editor by double-clicking on this map.

A quick glance at the editor

The tool bar is provided with a little title bar to the left with which, it is possible to drag it at any screen position, preventing thus the painting zone on which you are working from being hidden.

Tool Bar

The first icon appearing depicts three dots (this is the dotting bar). Click on this icon to open a dialog showing the different icons of the available modes. Select the upper right icon of this box to use the pen bar.

The picture will appear in black. To start painting, first select a colour, either within the bar's range (at that moment of gray colours), if you wish, or any colour from the palette by pressing the **C** (colour) key to make the colour selection's dialog appear. In this box, you can select either a colour from the main palette or, by clicking on the right part, the range that must appear in the tool bar.

The pointer coordinates in respect of the picture appear to the right of the pen icon. Then, an icon with a magnifier to vary its zoom percentage is displayed; this can also be done by pressing the **Z** (zoom) key.

You can try out other icons and tool bars. Once you have painted something, use the **Backspace** key (placed above **Enter**) to go back in the creation of the picture (undo) and then, **Shift+Backspace** to advance (redo).

3.2 Colour palettes

The colour palette is the basis of any picture. However, as far as games are concerned, it will only be possible to simultaneously display **256 different colours** on screen. These are the colours chosen with the colour palette at first. It is very important to know that all the pictures that are going to be used at the same time in a game must have been created with the same colour palette. The games can use several palettes, but **only one can be active** at each moment. Therefore, only the graphics using the active palette will correctly be seen at that moment.

It is also possible to **adapt** a graphic created with a colour palette to another one. For each pixel of the original graphic, the system will look for the most similar colour in the new palette, replacing it. Nevertheless, as similar colours aren't occasionally found, the pictures normally lose quality when they are adapted to another palette.

There is always just one active palette in the DIV environment. All the maps of the desktop must use this palette. For that reason, when the aim is to open a map (or a maps file) that uses a different palette, the system will ask you first whether the new palette must be activated.

Important: If there is unsaved information in the loaded maps, answer **Cancel** (not activating the new palette, as this task could damage the loaded maps), close the map (or filed) recently loaded, save all the maps loaded in their respective archives, close them and then, reopen the new map.

If you answer **Accept** and the new palette is activated, the system will ask you whether you wish to adapt the maps previously loaded to the new palette. If you do not want to do so, then close the loaded maps, without saving them previously (as they would be incorrectly saved, with a colour palette different from theirs). In short:

Dialog Box

- In order **to adapt the open map to the active palette in the environment**, answer **Cancel** to the question activate the new palette. If you are not satisfied with the performed adaptation, close the new map without saving it.
- In order **to adapt the loaded maps to the palette of the open map**, answer **Accept** to the question activate the new palette and **Accept** to the question adapt the loaded maps. If you are not satisfied with the performed adaptation, close the previous maps without saving them.
- In order **not to adapt any map to another palette**, first save and close the loaded maps in the desktop and then, open the new one, answering **Accept** to the question activate the new palette.

Once you start a project, you must decide which colour palette you are going to use for it. If it makes no difference, then by default use the palette of DIV Games Studio (**div.pal**). Otherwise, you must use the palettes editor (**palettes \ edit palette...**). Once you have created a new palette, it is advisable to save it in a disk's archive, as it can be recovered from this archive if the palette is later modified or accidentally changed.

DIV Games Studio contains a library with many graphics ready to create new games. However, keep in mind that many of these graphics use different palettes. Therefore, you will have to adapt some of them to the palette of others, in order to put them together in a game. The best option is, once your own palette has been defined, to adapt the graphics of the library (the graphics that you wish to use in the game) to this palette.

The option **palettes \ merge palettes...** allows you to create one palette from two, including the most characteristic colours of both palettes in the resultant one. This is a very good option to use graphics with several palettes, to create a mixed palette and then to adapt all the graphics to it. It is also possible to create a palette from more than two, merging them in pairs until only one is left.

The palettes menu that allows us to access these and other options has been described in section 2.3 of this book.

3.3 Transparent colour

From now on, many references will be made to the **transparent colour**. This is the first colour of the palette. The colours in the palette are always shown in order, in 16 lines, from the uppermost line with colours ranging from 0 (left) to 15 (right), to the lowest line with the colours ranging from 240 to 255. Thus, the transparent colour is the upper left one in the palette.

This colour is normally **black** and it is the colour that all the pixels of a map have when they are created. This tonality can be modified (in the palettes editor), but at first, it is not advisable to do so.

The name **transparent** stems from the fact that the graphics' pixels painted in (or left in) this colour **won't appear on painting the graphics in the game**. That is to say, the zones of the graphics that **are not opaque** are painted in this colour. Without a transparent colour, all the graphics should be squared, such as the maps containing them.

Other utilities also name this colour **background colour or mask colour**.

In the graphic editor, it is possible to recognise the transparent colour because, **on pressing the B key**, this colour changes from black to an intermediate gray. This task is performed on many occasions to clearly observe the graphics' outlines (to be able to recognise their transparent zones). To change the transparent colour to black, it is necessary to press the **B** key again.

Therefore, it is advisable to have **two black colours** in the palette. On the one hand, colour 0 that will be the transparent one. On the other hand, a black colour (that can be placed at any position in the palette) to paint the zones of the graphics that must **really be black** and not transparent. To paint in this black, the transparent colour must previously be highlighted by pressing the **B** key as, otherwise, nothing will be seen. This **opaque black** is colour number 240 of the palette by default of DIV (that of the lower right corner).

A little experiment can be carried out to verify the effect of the transparent colour, which is as follows:

- Create a new map and paint something in it (some colour doodles).
- Exit the editor and creates a copy of this map (drag the map window to the wallpaper).
- And finally, drag the copy that has been created (the new map window) to the original map.

The editor will be entered to copy the graphic on itself. On moving it with the mouse, it will be possible to observe how the zones that were left in transparent colour do not appear now. It is possible to force the painting program to paint the zones of transparent colour in its original black, by selecting the icon depicting a little man in the tool bar.

The transparent colour won't ever be painted in the games. Therefore, when the aim is to show black parts of a graphic, they must be painted in **opaque black**.

3.4 Basic controls

Almost all the tool bars have a series of common commands, which are now described. At the same time, this section also explains how the paint program generally works.

Control is basically performed through the mouse, but in the painting zone, when **precision** in movements is required, the pointer can also be moved with the **cursors** or the following keys:

Q, **A** Move pointer up / down.
O, **P** Move pointer left / right.

The **spacebar** can also be replaced by the left mouse button, when the pointer is controlled through the keyboard. That is to say, in most bars, this key will allow us to paint in the selected colour.

Control through keyboard is performed pixel by pixel. To move faster, it is necessary to press the **Shift** key simultaneously. If the **Num. Lock** key of the keypad is activated, the cursors will move the pointer 8 by 8 pixels. To move it one by one pixel, this key must be disabled.

In most bars, it is possible **to take a colour from the picture** if the **Shift** key is held pressed while the **spacebar** is pressed or the **left mouse button** is clicked on a pixel of the edited map.

Besides the mouse, the **W / S keys** may be used to choose a lighter or darker colour of the selected range. Moreover, if the **Shift** key is held pressed, then the range appearing in the bar (instead of the colour) will be chosen. These tasks can also be performed if **Control** is held pressed while the **cursors** are used, selecting the colour of the range with the **left** or **right** arrow keys and the range, with the **up** or **down** arrow keys.

To select the **transparent colour** momentaneously, it is possible to press the **0** (zero) key. If later this key is pressed again, the previous colour will be restored. It can also be done by clicking on the **black rectangle** placed just before the colours range of the bar (to its left).

To undo actions, we have already mentioned the **Backspace** key, which is used together with **Shift** to redo the undone actions. The **undo icon** appears as an arrow to the left, next to the magnifier icon (the edit zoom).

The use of the **Z** key to vary the zoom percentage was also mentioned (it is necessary **to point the zone to be expanded with the mouse** when this key is pressed). When the maps are edited expanded, they won't fit on screen on many occasions. To move through the complete map, it is simply necessary **to move the mouse towards the edge of the screen.**

The screen's **shift can be blocked** by clicking on the coordinates of the tool bar. They will change their **light gray** colour for a **dark gray**, blocking the shift of the expanded zone. To unblock the shift, it is necessary to click on the coordinates again.

The dialog boxes displayed from the tool bars can be exited in many ways:

- By closing the box (with its upper left button).
- By pressing the **ESC** key.
- By pressing the left mouse button.
- By selecting the icon that activated it again.
- By clicking on the edited map.

A summary with all the available keys in the **graphic editor** can be found in **Appendix C** of this book .

3.5 Generic icons

As it has been already mentioned, the first icon of all the tool bars is that indicating the painting mode. Each bar corresponds with a painting mode and therefore, has its own initial icon. The bars are individually described in chapter 4. To access these bars, together with clicking on the first icon, the **function keys** can also be used. The list of keys, as well as a summary of the tool bars' function is now shown.

F2: **Pen**, for hand drawing sketches and outlines.
F3: **Straight lines**, to create diagrams and geometric figures.
F4: **Multiline**, variation of the previous bar for stringed lines.
F5: **Bézier curves**, to trace outlines and curved lines.
F6: **Multicurve**, variation of the previous bar for stringed curves.
F7: **Rectangles**, to create squared or rectangular boxes or frames.
F8: **Circles**, to create circumferences, circles or ellipses.
F9: **Spray**, tool for retouching and artistic finish.
F10: **Filling**, to fill several types of surfaces.
F11: **Blocks edit**, tool to manage graphic blocks.
F12: **Undo**, specific bar to do and undo actions.
Shift+F1: **Text**, to write texts inside the edited maps.
Shift+F2: Bar to position **Control points** inside the maps.
Shift+F3: **Dotting** bar, to edit little graphics accurately.

Painting Tools

The **Colour sampler** is another of the icons shared by many bars, which normally appears after the colours range and which, on clicking on it, momentaneously leads to another little bar used to select a map colour.

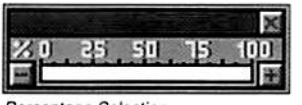
The coordinates, the magnifier and the colour on which the mouse pointer is placed in the map are shown in the sampler bar. This colour will be taken to paint on clicking on the map, automatically returning to the previous bar.

Colour Sampler Bar

Keep in mind that, to choose a map colour, it is also possible to click on it by holding the **Shift** key pressed at the same time.

A great number of bars show the **percentage** icon after the colour sampler. The **amount of ink** added to the picture on painting is controlled through this icon.

A dialog box with a ruler will appear on clicking on this icon. It is necessary to click on this ruler to establish the percentage. A percentage equal to 100% (the value of most bars, by default) will imply to paint it in completely solid colours. That is to say, they won't mix up with the previous picture.



Percentage Selection

Very good results may be obtained if you learn how to use this technique properly, as it allows us to retouch a part of a graphic (for instance, with the pen bar) by **adding** a small quantity of one colour. It is also useful to paint with a low percentage of **black** colour to get parts of the picture gradually darker, or with **white** colour to get them lighter.

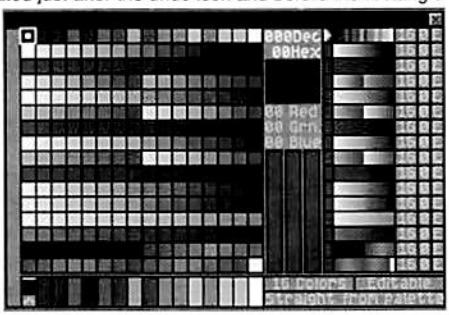
The icons that can appear following these two icons in the different tool bars are normally **mode icons**, which offer several ways to use this tool. Generally, on using them a new dialog is displayed with new icons, each of them showing a different way to use this tool.

When a **second pair of coordinates** appears in the right part of a bar, they will normally indicate the width and height of the object that is being painted with the tool. Thus, it is possible to perform precise measurements.

3.6 Colour ranges

To access the dialog box labeled **Creation and selection of colours range** described now, it is necessary to press the **C** key or to click with the **left mouse button** on the selected mouse in the bar (it is the rectangle located just after the undo icon and before the rectangle of the transparent colour). A **colours range** is simply a sequence of colours of the palette used to paint in different shades of a colour (a sequence of greys, reds, etc.).

A rather big box will appear with many colours. To return to the picture, it is necessary either to press the **C** key or to click on the selected colour again. This box is split into three big zones: the colour palette (upper left part), the range editor (lower part) and the list of colour ranges (upper right part).



Colour Range Selection

The colour palette
This section is simply used to select colours of the palette, by clicking on it. Information about the selected colour or about the colour on which the mouse is placed, if it is inside the palette, appears to the right.

Information deals with the colour number in decimal, in hexadecimal, a colour sample and the percentages of red, green and blue comprising the colour.

From this box **it is not possible to modify the palette**, as this task can only be performed from the palettes menu, outside the graphic editor.

The range editor
Up to 16 different colour ranges may be defined for a palette. The selected range appears in the tool bar and can be edit with this tool.

Note: The ranges editor **does not modify the colours** of the palette, but it only **arranges them again**, in order to create with them colours' sequences useful to paint.

An exact copy of the range located in the tool bar appears to the left and three rectangular buttons that change their value when pressed appear to the right, having the following functions:

- The first one (upper left) defines the number of colours comprised by the range, that can be changed between **8**, **16** or **32** colours.
- The second one (upper right) defines whether the range is **editable** or **fixed**, that is to say, whether the range is going to be modified or is going to remain in its current state. When the range is **editable**, one or several small **icons with a gray up arrow** appear in it. These icons are used to establish the range colours.
- The last button (lower) defines the mode in which the aim is to define the range. The available modes are as follows:

    - Straight from the palette. In this mode, the first colour of the range is defined, while the other colours are sequentially taken from the palette.
    - Edit every colour. In this mode, each of the range's colours can individually be defined, by selecting them from the palette and assigning them later at each position of the range.
    - Edit every two colours. This mode is similar to the previous one, with the proviso that it will only be possible to define one colour out of every two. The intermediate colours will be defined by the system with the colour of the palette closest to the average of its two adjacent colours.
    - Edit every four or eight colours. These last two modes are practically identical to the previous one, with the proviso that they even define less colours. They are used by the system to search for the intermediate ranges between every two colours, automatically defined.

Once a range has been defined, it is advisable to **fix** it to prevent it from accidentally being lost. When the palette is saved in an archive, the information about the colour ranges used in it is also saved in this archive.

The list of colour ranges

The 16 colour ranges that can be defined appear in the upper right part of the dialog box. This list is very useful, as it allows us to directly select the colour range in which the user is interested at each moment in order to paint.

It can be noticed that a small white arrow indicates which is the selected colours range. It also appears in the ranges editor and the tool bar.

Three values are shown following each range, defining its characteristics: **number of colours**, **mode of definition**, and whether it is **Editable** or **Fixed**.

3.7 Use of colour masks

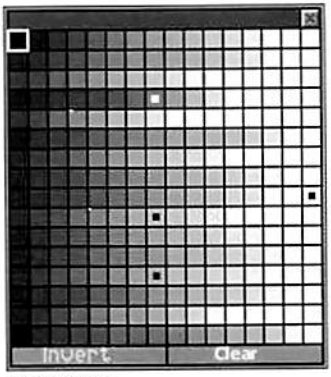
The use of colour masks is an advanced technique. It is used to protect parts of the picture that must not be modified, a task that can be performed by selecting a set of the palette colours, over which it won't be possible to paint.

For instance, if the black colour is defined as a mask in a map with most of its background painted in black and with a picture at the center, then the black colour will be protected, so that it is not possible to paint on it. Therefore, it is possible to paint the graphic keeping inside its outline, as the paint tools won't paint outside the graphic.

It is necessary to press the **M** key or to click with the **right mouse button** on the selected colour of the ruler, in order to access the dialog of masks definition (the same as to activate the dialog of colour ranges, but with the other mouse button).

A dialog containing all the palette's colours will be displayed, with two buttons appearing in the lower part. The colours aimed to be masked can be selected and unselected by clicking on the palette with the left mouse button. A little square appearing at the center of a colour will mean that it is not possible to paint on this colour in the map. If all the colours are selected, it won't be possible to paint with any tool in the map.

It is also possible to select (mask) colours by clicking **on the edited map** (while the masks dialog is open).



Masks Dialog

The two buttons located in the lower part of this dialog fulfill the following functions:

Invert. It is used to invert the state of selection of all the palette's colours. If, for instance, the aim is to paint only in one colour, it will be easier to select this colour and then, to reverse the selection by pressing this button rather than selecting the rest individually.

Clear. It is used to take the defined masks out. In other words, to unselect all the colours in this dialog.

Note : It is commonplace, after having masked some colours, to forget it and, later, to notice how the paint program is not properly working. That is explained because the masks must first be taken out (which can be done by pressing the **M** key and then, clicking on the button **Clear** in the dialog).

A practical example

- Creates a new map (**maps \ new...**), enter the editor (by double-clicking on the map), select the Spray tool (which is accessed by pressing **F9**) and paint something, using several colours at the center of the map.
- Access the masks dialog by pressing **M** and select the **transparent colour** (the first colour of the palette, in the upper left corner) to hide it.
- Exit the editor by pressing **ESC** and create a copy of the map, dragging it to the wallpaper.
- Drag the copy recently created to the window containing the original map, in order to copy the graphic on itself.

If you follow these steps, you will notice how, on moving the graphic, it is not displayed against the black background, as this colour has been masked to prevent it from being modified. Don't forget to take the mask out in order to continue to paint normally.

Chapter 4
Graphics Editor

4

CHAPTER 4: The Graphic Editor.

This chapter describes the specific functions of every painting tool bar and some of the techniques that can be used to make some tasks easier or to obtain better results with this graphic editor.

It is advisable to read this chapter with the computer before you, in order to practice the techniques described in it at the same time.

4.1 Dotting and pen bars

These are the two basic painting tools to initiate the pictures or to define the small details.

Dotting Icon

The **dotting bar** is used to put and remove pixels in a map. To **remove** pixels means to put them in transparent colour again. This mode is normally used to create very little details of graphics accurately.

It is normally used by controlling the pointer with the keyboard (using one of the two sets of keys: **cursors** or **QA,OP**). The keyboard is also used to select colours (with **control** and the **cursors**, or with **W,S** if the second set of keys is used) and to put and remove pixels (**space bar**). The **Shift+Space bar** is also normally used to take a colour of the picture itself.

This way to paint pixel by pixel allows us to create graphics with great. Precisely for that reason, the dotting bar, instead of the pen bar, is used, as the first one facilitates to remove a pixel by clicking again on the same position, when the pixel has been put incorrectly or mistakenly.

Pen Icon

The **pen bar** is less complicated and is normally used with the mouse. It is the basic painting tool par excellence, and it is used to initiate most of the pictures or sketches. Better results will be obtained if you work with a very high zoom percentage, as it will be easier to perceive the details in this way.

It is not possible to vary the stroke thickness, which always equals one pixel. Thicker strokes of other tools are only useful to paint doodles instead of pictures, creating the negative habit of painting quickly and bad.

Percentage Icon

The **Percentage** icon, which allows us to define the ink quantity, appears in this bar. Good results will easily be obtained with a high zoom percentage, one pixel stroke and a low ink percentage. The **percentage** initially equals 100%, while the picture is defined, using lower percentages for the final finish.

There is another utility available through the pen bar: to **smooth**. This is obtained by holding the **D** key pressed while the picture is painted with the pen. In this mode **the user won't**

paint in the selected colour (no matter which one it is), but the map colours will gradually be modified to mitigate the sudden colour changes and to prevent some pixels from excessively standing out.

Don't try to start a picture with low ink percentages, smooths, sprays, etc. These artistic finishes must be left for the end. At the beginning, the picture's outlines and sections must be defined with solid and clean strokes, starting with the colouring, bright, etc. once they have been retouched.

To retouch outlines until they look good, it is normally necessary to make several attempts, clearing the inaccurate strokes (by painting with the background colour or using the **undo** command), to correct them on time.

4.2 Bars for lines and multilines

These two bars allow us to draw straight lines and they are used to define the most geometric parts of the initial sketches or very detailed finishes.

Lines Icon

The colour must first be selected, double-clicking later on both ends of the line. When the first end has been defined, the line will be seen in the way it will be drawn when the second end is defined. To cancel the task at that moment, press the **right mouse button**. To cancel the line once the two ends have been defined, use the **undo icon** (or the **Backspace** key).

To the right of these bars, two numeric values appear: the **width of the drawn line** (higher) and its **height** (lower). These values only appear when the first end of the line has been defined, to help us to measure. The **cursors** are normally used to position the line ends for the precise adjustments.

These bars contain all the conventional icons, among them that of **percentage** which, in this case, is only used to perform specific touches.

Multilines Icon

The **multiline** bar is practically identical to the previous one, with the proviso that the former allows us to draw several interwoven lines. Once the last end has been defined, the **right mouse button** must be pressed to finish the multiline.

Finally, it is also possible to use **smooth** with the **D** key. Smooth can be used for many effects; for instance, to paint anti-aliasing lines, as shown in the following practical exercise:

- Create a new map. Enter the editor, choose the lines tool (F3) and selects the white colour.
- Draw a line from the (10,10) coordinates to the (68,47) coordinates; use the cursors to adjust the coordinates. You will see how the line has an aliasing appearance (the pixels are seen a lot).
- Now draw a second line from the (10,9) coordinates to the (68,46) coordinates (one pixel upper). But, on defining the second end, hold the **D** key pressed.
- Perform the same task one pixel lower than the original line. You will see how, on creating two smoothed lines above and below the original one, you have succeeded in hiding its aliasing appearance .

4.3 Bars for curves and multicurves

The curves bars allow us to accurately do curved strokes in the initial sketches, without requiring a steady hand (unlike drawing with a pen).

Curves Icon

The use of the **curves bar** is similar to that of the lines bar, with the proviso that the former creates **bézier** curves instead of straight lines. The initial and final ends of the curve must first be created (the same as it happens regarding straight lines). Then, two other points must be defined. The first one, indicating the steepness of the curve in the initial end (to which the curve goes up) but the farther this point goes from the initial end, the sharper will the steepness with which the curve emerges from this end be. The second one will specify the final end's steepness in a similar way. The curve will be defined after the left mouse button has been pressed four times.

Multi curves Icon

The **multicurves bar** works in a way slightly different way. It is very useful when it comes to defining curved outlines of a graphic. A complex curve is defined by sections, with **splines**.

The first section will always be defined as a **straight line**, establishing its initial and final ends. Then, different points will be defined along the trajectory of the outline. They will be linked by the computer, automatically creating a continuous curve.

It is very important to use the **+** and **-** keys **of the keypad** to vary the **strength** with which the curve emerges from each point of the trajectory. That is to say, the destination point and the initial steepness of the section must be established for each new section. If the intensity is reduced to the minimum, this tool will practically work in a way similar to the multilines bar.

These two curves bars also allow us to adjust the ink **percentage** and use the **smooth**, as it happens regarding the previous bars.

4.4 Bars of rectangles and circles

These bars allow us to paint the most basic geometric figures. They can be used as colour filters, among many other applications.

Rectangles Icon

The **rectangles bar** is used in a way similar to the lines bar, selecting the colour first and defining the two corners of the rectangle by double-clicking the left mouse button on the map. The **width** and **height** in pixels of the rectangle that is being defined appear to the right of the bar.

Mode Selector (Rectangles)

A new icon, placed to the left of the rectangle's size, allows us to select two modes with which this bar paints rectangles or solid boxes (by default, boxes will be painted).

To paint squares (or squared boxes), it is necessary to hold the **Control** key pressed while the second end is defined. Thus, the width and height will be forced to be equal.

This bar also allows us to establish the ink **percentage**. When solid boxes are painted with an ink percentage less than 100% against a background picture, the effect of applying a **colour filter** on that picture will be obtained. On defining the box, it is possible to vary the ink percentage (by clicking on the percentage icon) until the desired effect is reached.

Circles Icon

The **circles bar** works in a way similar to the rectangles bar, as it also allows us to paint circles (filled) or circumferences (not filled), apply filters with the percentage, etc. The circles are defined with **two radii** (horizontal and vertical, respectively), creating kinds of ellipses that are but flattened circles. To create perfect circles, with similar radii, it is necessary to hold the key **Control** pressed, as it happens regarding the rectangles.

Mode Selector (Circles)

Besides choosing between circles or circumferences, the mode selector icon allows us to define them in two ways:

- From corner to corner, defining the upper left corner and the lower right corner of the box that embraces the circle.
- From center to the radius, defining the circle's central point and then, its radius length (or its radii length, if **Control** is not pressed).

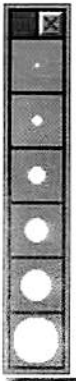
The mode selector icons of both bars appear in a little dialog that is displayed from the tool bar, and their pictures are self-explanatory.

4.5 Spray bar

This tool bar allows us to use the computer version of a paint spray (an aerosol or diffuser) to carry out artistic and irregular finishes. Its use is almost as easy as that of a real spray. It is necessary to select the colour and click on the map at the same time that the pointer is moved.

Spray Icon

It is **essential** to use the **percentage** icon of this bar to adjust the paint quantity expelled from the spray. By default, the opacity percentage is adjusted at 25%. To obtain precise finishes, it is better to use very low percentages (even less than 25%). Thus, it is possible to retouch the picture little by little.



Thickness Selection

A new icon appears in the right end of this bar. This icon is used to adjust the spray's **stroke thickness**, which is now necessary. By clicking on this icon, a new dialog will be displayed, showing the available thickness; select one by clicking on it.

As usual, better results will be obtained with a high zoom percentage (**x4** or **x8**) with a small spray and with a low ink percentage. It is also important to learn how to use the colour masks (described in section 3.7) together with the spray in order to keep inside the zones on which the aim is to apply the effect.

The spray has several interesting applications, related to retouching rather than to painting. For instance, by using the spray on a graphic while the **D** key is held pressed, it is possible to **smooth** the graphic irregularly.

Another example: it is possible to apply **irregular bright** with the spray on graphics already painted, by selecting the white or black colour (depending on whether the user wishes light or shade) and adjusting the ink percentage to the minimum.

4.6 Filling bar

This is one of the most useful bars of the graphic editor if you learn how to use it properly; thus, it requires practice. Its appearance is similar to that of the previous bars, but its application is quite different.

Filling Icon

The **filling bar** is used to fill the parts of the pictures with colours. It works in four ways, that must be distinguished. They are activated by clicking on the icon placed to the right of the bar and selecting the filling mode with one of the icons that are displayed.

Conventional filling

The first mode (corresponding with the icon selected by default) is shared by most of the graphic tools. One colour is selected and a part of the picture is clicked. With the selected colour, the program will paint all the pixels of the picture whose colour is the same as that of the pixel that has been clicked on and which are **attached** or **joined** to it.

Conventional Filling Icon

Example: A test can be carried out in a new map, painting an irregular, closed outline with the pen tool, selecting a filling colour and clicking within the outline of this filling mode. If the outline is closed, its inside will become filled with the selected colour. The same operation can also be done regarding the external part of the map.

This tool is used to colour sketches. When the filling is going to be used, care must be taken to keep all the sections to be filled **closed** since, if there is any hole or fissure in its outline, the filling **will leave to the external part** of the section.

Diagonal filling

Diagonal Filling Icon

This mode is similar to the previous one, but with an important difference. In the conventional filling, when the pixels **attached** to the original one (to the pixel in which the filling starts) are selected, only the pixels from which **it is possible to reach the original pixel** are taken into account, without passing through a pixel of a different colour, and with individual movements in **straight line** (passing from one pixel to the next one by one of the four sides).

This filling mode considers attached pixels those of the same colour united by a **diagonal**, as well as those united by one **side**.

Example: In a new map, paint a square in any colour with the rectangles tool (only the edge, not filled with colour). In the same colour, paint a circumference (not filled) inside the square. Now select another colour with this filling mode and click on the circumference's inner part. You will see how the filling **has left** it by the corners of the circumference (by the diagonals) but not of the square (as it can not be left by any side or diagonal of any pixel).

The usefulness of this filling mode lies on the possibility of changing the colour of pixel line graphics, as they deal with pixels attached by sides as well as by diagonals, and the conventional filling could not go through them.

Example: In a new map, paint any doodle with the pen tool (in a single colour). Now select another colour and, with this filling mode, click on a pixel of the doodle; you will see how it is filled. You can carry out the same test with the conventional filling mode, noticing how, in this case, only one of the segments shaping the doodle is filled (as the conventional filling doesn't leave by the diagonals).

Filling To A Limit Icon

Filling to a limit

The third filling mode is the most powerful one, but not the most useful one. Instead of filling a colour with another one (like the two previous modes), it fills with one colour all the colours it finds, until it reaches an edge of the same colour.

Example: In a map with several pictures, select a colour not used by them (purple, light green, etc.) and, on them, paint a closed outline with the pen, leaving a part of the pictures inside the outline and another part outside. Finally, with the same colour select this filling mode and click on within the outline. The inner outline will be filled, no matter the pictures contained in it.

If several graphics are painted against a black background map, this filling mode is selected (with the black colour of the selected background) and one of the graphics is clicked on, being deleted. It happens because the program has been filling with the black colour until it has found an edge of the same colour (the graphic's exterior).

Filling With A Gradient Icon

Filling with a gradient

It is the filling mode most esteemed by the graphic artist and, at the same time, one of the most difficult to find in other painting programs. It requires technical skill, and terrific results may be obtained with it.

In this mode, a section is filled with a colour with a colours' **gradient** of the **selected range** in the tool bar (explanation about how to define and select these ranges was included in section 3.6). **Lighter or darker colours of the range will be used, depending on the lightness of the colours delimiting the filled section.**

Example: In a new map, select a uniform colours' range (for instance, any of those defined in the DIV's palette by default). Now paint a circumference (and not a filled circle) with one of the darker colours of the range, about 50 by 50 pixels (not very big) and, within it, paint a small circle (filled) with the lightest colour of the range. Finally, select this filling mode (with the range of colours in the tool bar) and click on between the circle and the circumference.

A **new tool bar** will appear with two icons: a single arrow and a double arrow. You must click on one of these two icons in order to start the gradient. The **single arrow** advances a step and the **double one**, several. When you are satisfied with the result, press **the right mouse button** to go back to the filling bar.

It is advisable to carry out several tests, filling in different ways, to observe the results that can be obtained. Take into account that the gradient allows us to have light and dark colours in the outline of the filled zone. What is really achieved with this mode is to fill a section with the average of the colours defining its outline, taking the average of its lightness.

The bigger the zone to be filled is, the longer the effect will take. Therefore, you must try to divide the big zones into small ones, if possible.

Not always are better results obtained the longer the gradient is being applied. Indeed, on many occasions when it is excessively applied, the outlines of the section stand out too much. Thus, the result looks worse.

There are not many fixed rules to obtain good results with gradients. To a large extent, good results depend on **uniformity**, **number of colours of the range** (it is better to use 32 colour ranges, if possible) and the section to be filled. Practicing with this tool is indispensable to master it.

Note: The bar of blocks edit described next can also be used to carry out other very effective types of colour filling.

4.7 Blocks edit bar

This is the main and most important bar of the graphic editor. It is mainly used to select zones of the picture and allows us to access other tool bars to perform different tasks with these zones.

Block Edit Icon

It is the graphic tool to cut and paste. But it can be used for other tasks such as to rotate, scale, lighten, darken, soften, invert, etc.

The first action to perform will always be **to select the zone** with which the aim is to work. In the right part of the bar, an icon depicting **a dotted square**, indicates the mode in which this selection is going to be performed. By clicking on this icon, a dialog with six icons will appear in the right part of the bar. These six icons indicate the possible modes to select a zone of a graphic.

Learning to select parts of the maps in these six modes will be very helpful to gain skill to work with the graphic editor. Therefore, the **way to select zones** in each one of these modes will be now described. The tasks that can be performed with these zones once selected will be explained later. You can try out all these modes on a map with some pictures. Once the zone has been selected with a mode, press the **ESC** key to try out to select with another mode.

Block Selection

Box selection

Block Selection Icon

This is the easiest way to select a zone of the edited map. It deals with tagging a simple box (or rectangle) of the map. To do so, it is necessary to click on two of its ends with the left mouse button.

Once the initial box has been defined, it can be adjusted by clicking near its corners to place it in a new position, or by clicking near the central point of one of its four sides to increase or reduce the box by this side.

Filling selection

Filling Selection Icon

This selection is very versatile. It allows us to select a zone of a map by selecting a set of colours from one or several points. It is necessary either to click several times on the zone to be selected, or to move the mouse pointer through it with the left mouse button pressed, until the entire zone is selected.

In order to understand its performance well, practice is required. Once the zone of the map has been selected, it is possible to add more colours or zones. This selection is not always appropriate but, on many occasions, it allows us to select complex zones quickly. It is very similar to the **magic wand** of other painting programs, as they name it.

If, for instance, a yellow pixel is clicked on, all the yellow pixels united to the original one will be selected. Later, if a red pixel is clicked on, all the yellow or red pixels united to the original ones (both the original red and yellow) will be selected. And so on.

Polygonal selection

Polygonal Selection Icon

The polygonal selection is very common in the painting programs. It deals with selecting the zone by drawing its outline. It can be done in two ways: defining several points (clicking with the left mouse button) around the zone that the computer will unite with straight lines, or freehand drawing its outline (moving the pointer with the left button pressed).

The selection won't be completed until **the outline is entirely closed**. This can also be done in two ways: reaching the initial point of the outline again, or pressing the **right mouse button** (or the ESC key) for the computer to unite the initial and final ends of the outline.

Once the polygonal selection has been closed, it can not be modified. Therefore, if it has been incorrectly defined, it will be necessary to press the **right button** (or the ESC key) again to **untag the zone** and thus be able to define the selection again.

Several boxes selection

Multibox Selection Icon

This is another quick mode to select parts of a map. First, a simple box is defined, like in the first mode. But, once it has been defined, instead of adjusting it, more boxes may be defined. The union of all the tagged boxes will comprise the selected zone.

It is used to quickly tag pictures that can not be tagged with a simple box because a part of another picture that is not intended to be selected would be included in it. For instance, a "L"-shaped graphic could easily be selected with two boxes.

Automatic Box Selection

Automatic box selection

This mode of selection and the following one are the quickest modes. They can only be used to select a picture inside a map with several separate pictures. It is essential that the map's main background has a **transparent colour**. That is to say, that the pixels external to the pictures are of the first palette's colour (number 0).

If any pixel of a picture of the map is clicked on, the **minimum box containing** it (a rectangular zone containing this graphic) will be instantaneously selected. The pictures must be separate enough one from other so that it is possible to define a box which doesn't invade the adjacent graphics.

If, once the selection has been defined, the same picture is clicked on again, **the box will increase one pixel by** its four sides. That is to say, it will include the picture with one pixel added margin. If it is clicked on once again, it will either recover its origins size, or increase even more if it has found another part of the graphic near the original one.

On the contrary, if another picture of the map different from the one initially selected is clicked on, the original one will be unselected and the new graphic will be selected.

Automatic Filling Selection

Automatic filling selection

This selection is similar to the previous one, but using filling techniques, instead of boxes. It is as if, in the previously mentioned filling selection, all the colours except the transparent one were preselected. That is to say, on clicking on a picture's pixel **all the pixels united to the original whose colour is not the transparent one, will be included.**

This is a really powerful selection tool. It can make the work with the graphic editor easier. Moreover, once the initial selection has been defined, other selections may be added to it. That is to say, as many pictures of the original map as necessary can be selected.

Controls common to all the selection modes

- All these selections, once defined, may be shifted through the screen if the **Control** key is held pressed, clicking at the same time with **the left mouse button** on a position of the edited map.

- To cancel any selection (untag it), it is simply necessary to press the **right mouse button** or the **ESC** key. Then, the edit bar will be returned to its original state, being possible **to define the selection** again from the beginning, change the selection mode, or exit this tool bar.

- As long as a zone is selected, it is always possible to consult its **width** and **height** in the right end of the edit bar.

- As long as the selection is being defined, it may be useful to vary the zoom percentage by using the **Z** key, and thus observing better which are the pixels that are being included in it.

Once the selection has been defined

Three new icons will appear in the bar when a zone of the map is selected. These are the **move, effects** and **cut and paste to window** icons, from right to left respectively.

Icon To Cut

The **cut and paste to window** icon is the easiest one of the three. On clicking on it, **a new map window will be created** in the desktop **and the selected zone will be pasted** in it. Inside the graphic editor, it will be only possible to notice how **the selection will be untagged** but, on exiting it, the new map window will appear in the environment.

This icon is really useful, as it provides great flexibility in the operations with graphic blocks, for instance:

- **Copy sections from some maps to others**. Taking the selection out to a new map and then, dragging it to the map in which it is intended to be copied.
- **Edit graphics in bigger maps**. For instance, creating a 20 by 20 pixel picture in a map of similar seize is not very comfortable. It is easier to create this picture in a 200 by 200 pixel map that allows us to have several copies, textures and samples of it, in order to take it to the 20 by 20 pixel map, when the picture is finished.
- **Take a graphic or texture of a map out**. The maps are normally used to create many pictures, not only one. Finally, in order to include these graphics in the game, they will be selected, cut and paste to individual maps and stored in a **file FPG** which can be loaded in the game.

If the **move** or **effects** icons are clicked on, two new tool bars will be accessed, to move and copy the selected zone or to apply graphic effects, respectively. These two new bars are now described.

Icon To Move

Move the selected zone

Once a zone of a map has been selected, by clicking on the move icon (depicting a hand on a little man), the bar to cut, copy and paste graphic blocks will be accessed.

Note: This is also the bar that appears when a map is dragged to another one in the environment's desktop.

In it, the selection will be moved through the map with the mouse pointer (a hand). To **copy** the block to a new position, it is necessary to press the **left mouse button**. To hide the graphic of the mouse pointer (to see the appearance of the block in a position, without the hand above it) it is possible to press the **H** key.

The usual commands, such as the **right mouse button**, or the **ESC** key may be used to return from this bar.

Bar To Move

In this bar there are many new icons. The zoom and undo icons are displayed after the coordinates, like in most of the bars, appearing the following ones to its right:

Opacity / Semiopaque. On clicking on this icon, there will toggle between two ways to copy the graphic: opaque (by default) or semiopaque. The only limit when it comes to copying the semiopaque graphic (with an effect of transparency), is that not always are the colours necessary to create the effect found in the palette, thus using the closest ones.

Transparent colour. This icon must be clicked to prevent the first colour of the palette from being shown as **transparent**. That is to say, when the aim is to move the selection like a

compact rectangular block, with no hollows. It is used to prevent the copied block from being mixed up with what there is in the map's background.

Horizontal flip. The third icon will horizontally flip (mirror) the block. If it is clicked again, the block will be restored to its original position.

Vertical flip. This icon is complementary to the previous one. On clicking on it, the graphic will be vertically flipped.

Angular rotation. On clicking on this icon, a **new bar** used to rotate the block will appear. It works easily. First, it is necessary to place the block in any position of the map and press the **left mouse button**. Now, moving the mouse around the graphic, its new angle will be defined. The **left mouse button** must be pressed again to establish the angle, while the **right mouse button** must be pressed to cancel the rotation.

Block scaled. This is the icon to change the size of the block (increasing or reducing it). It works practically equal to the rotation icon. A **new bar** also appears. First, it is necessary to click on the map's position. Then, moving the mouse, the zoom percentage is selected to establish it finally, by pressing again with the **left button** in the map.

Delete original selection. The last icon will delete the original zone in the map. **The colour selected in the bar of blocks edit will be used to delete**. Therefore, if the aim is to delete with the **transparent colour**, it must first be selected (before entering the move bar). This icon is used to **move a block**. To access this bar, it is first necessary to select the zone, and then, the move icon. Then, the block is deleted from its original position with this icon and finally, it is copied in the new position.

All these icons are compatible. That is to say, it is possible to use all the necessary effects to obtain the desired result.

It can be noticed that the icon to delete the original selection can be used as a **filling** tool, selecting the colour in the tool bar and the zone to be filled (with any of the selection mode) so that, on clicking this icon, the selection is filled with the chosen colour.

Effects Icon

Apply effects to the selection

Once a zone of a map has been selected, by clicking on the effects icon (with the letter **FX**) the tool bar will be accessed to apply effects to the selected block.

The pixel line delimiting the selection will continue to be seen in this bar. To return from this bar, it is also possible to use the **right mouse button** or the **ESC** key.

Effects Bar

New icons also appear in this bar from the undo icon (which must be used when, with an effect, the desired result is not obtained). These icons are now described, from left to right:

Pass to the selected range. All the colours of the selected zone will be changed into the colours of the range selected in the tool bar (edit bar). For that reason, the colours' range must be selected before entering the effects bar. The change is made depending on the levels of brightness of the selected pixels and the range's colours.
Reverse colours. Creates the negative of the selected zone, changing the light colours for the dark ones, and viceversa.

Create an edge. First, it is necessary to select a graphic with, at least, a margin of one pixel, and the colour with which the edge is intended to be created. Then, on selecting this icon, an edge of that colour will be created through all the graphic's outline.

Lighten. Lightens all the selected pixels. That is to say, it slightly increments its lightness. The only limitation are the available palette's colours.

Darken. This operation is opposite to the previous one. It subtracts lightness from all the pixels of the selected zone. If the zone is excessively lightened or darkened, it is necessary to use the undo icon, and not the opposite operation as, in this case, the colours of the original picture would be more and more impoverished.

Soften. The icon located in the right end allows us to soften all the pixels of the selected zone. It must be used for effects in very specific zones as, if was applied on complete pictures, they would look blurred. To avoid the excessive aliasing of some graphic's pixels, it is better to use the tools to soften specific zones (tools such as the pen or spray, by holding the **D** key pressed).

4.8 Undo bar

Undo Icon

The undo bar, that is accessed through the icon depicting a double arrow or through the **F12** key inside the editor, is used to undo or repeat all the actions performed in the graphic editor.

The undo task can be performed from practically all the bars, with the **Backspace** key, and to repeat actions, it is necessary to hold the **Shift** key pressed while the same key is pressed.

Therefore, it is not normally necessary to access this bar. However, it will be more comfortable to do so when many actions must be undone as there are four buttons to undo and repeat actions at two different speeds. The icons of this bar are self-explanatory: left arrows are used to undo and right arrows are used to repeat (redo).

Undo Bar

Once a picture has been created in a map, it is very funny to use this bar to undo and repeat the work, as if it was a video.

The **undo memory** establishes the limit of actions that can be recovered; it can be defined in the configuration window (with the option **system \ configuration...**). By default, the limit is 1088Kb, which will almost always be enough. However, if you are working with very big maps, it can be advisable to increase this limit.

Important: When you are working on several maps, it is vital to know that it will only be possible to undo the actions dealing with the last map on which you have worked. For instance, suppose that there are two maps and you enter the first one to paint a circle and then, the second one to paint a square. You won't be able to undo the circle, unless you first undo the square. That is to say, the actions can only be undone in the inverted order in which they have been done.

4.9 Text bar

Text Icon

The text bar is used to **write texts** in the maps with the fonts of the archives **FNT**, or with those created with the **Fonts generator** (explained in the fonts menu, section 2.5 of this book).

The appearance of this bar is practically identical to the rest of tools. Any activated **window of letters font** must exist in the environment to write with a font (a letter type) (see the fonts menu).

If there isn't any window of this type, it will be possible to write with **the letters font used by the programs editor** (they can be selected in **system \ configuration...** among several font sizes for the editor). In order to write with the editor font, it is necessary first to select the colour for the letters and, then, to click on any part of the map (with the **left button**) to input the text. The **ESC** key or the **right mouse button** must be pressed, once the text has been input.

If there is an activated font window, then it will be used to write. In this case, **the transparent colour must be selected** to write a text. If another different colour is selected, the **font coloured** in it will be shown. That is to say, the first colour of the palette must be selected to use the natural colours of the font.

While the text is input, the cursor can be positioned by clicking on another different part of the map.

When a text is being input, the previous character may be deleted with the **Backspace** key. To go to the following line, press the **Enter** key.

Note: This bar also has the ink **percentage** icon, which can be used to write translucent texts, instead of opaque ones. Thus, attractive labeling effects can be obtained, on writing on different textures.

4.10 Control points bar

Control Points Icon

The last accessible tool bar is not a painting bar. It is used to define **control points** inside the maps to be used in the games, in order to locate some positions of these maps.

There are different applications that can be given to these points inside the DIV programs.

This control points bar allows us to place up to 1000 different pixels inside a graphic. Each of them will be identified by a number (from 0 to 999).

Control Points Bar

To place one of these pixels, suffice will be to select the pixel number with the **left arrow** and **right arrow** icons and then, to click on the graphic.

To **delete** (unselect) a control point, it is necessary to click on it in the map once again.

The only control point used by the system is control point **number 0** (the first one). This point defines which is the **virtual center** of the graphic and it has many applications inside the language.

When control point number 0 is not defined, the system will work as if the **virtual center** of the graphic was its **real center** (a point placed half the width and height of the graphic).

Note: Once the control points have been defined, for them to have effect inside a program, it will be necessary either to save the **map** (archive **MAP**) or to include it again in the graphic file (archive **FPG**), by dragging it to this archive (depending on which graphic is loaded in the program).

4.11 Animations edit

The DIV graphic editor also allows us to create and edit animation sequences. The frames sequence will be created in a series of maps **all of them of a similar size in pixels**. That is to say, if the aim is to create an animation of 100 by 100 pixel size and 8 **frames**, the eight 100 by 100 pixel maps must first be created in the desktop.

The easiest way to create several maps of the same size is to create the first one (with the option **maps \ new...**) and then, to drag it to the wallpaper several times.

Then, **the maps windows must be arranged**. This is done by simply superimposing the windows on top each other. The last one is placed in any part of the desktop and, on it, the penultimate, and so on. It is not advisable to put each window **exactly on top** the previous one. Rather, it must be slightly displaced, for instance, a little to the right and somewhat lower.

Superimposed Windows

Important: If there are more maps of the same size in the desktop (continuing the previous example, another map of 100 by 100 pixels that must not be a part of the animation), **their windows must be minimised** for the program to know that they must not be included in the animation. Once the animation has been edited, it will be possible to **maximise** these maps again.

The animation can be edited once the windows have been put in order. For that, only the upper window among those comprising the animation must be edited (that is to say, the window that is over the rest) by double-clicking on it. And, once the graphic editor has been entered, it will be possible to pass to the following frame by pressing the **TAB** key, and to the previous one, with the **Shift+TAB** combination.

Practical example
Continuing with the same example, once the eight 100 by 100 pixel maps (all of them empty, in black) have been created, they will be put in order, as it has been explained (it

doesn't matter the order they are put in, as all of them are empty; thus, anyone can be first, second the last one, etc.).

Then, double-click to edit the upper map (the last one that has been placed). Now, select any colour and the pen tool. Draw a big 1 in the map (don't mind about the number's appearance). Now, press **TAB** and draw a 2 in the second frame. Press **TAB** again and draw a 3, ... and so on, up to number 8.

The animation is already created. You can use the **TAB** key (pressing **Shift** or not) to display the animation. If you hold this key pressed, you will see the whole series of the animation. Use the **Z** key to vary the zoom percentage in which the animation is seen. You can continue to edit any of your frames.

Note: If you exit the **graphic editor** in a frame that is not the first one, for instance, in the fourth one, you will see that the windows are put **in another order**, with the fourth frame over the rest. Don't worry, they are still in order. But now, **when you want to re-edit the animation, click on the uppermost frame**, the fourth one, so that they don't get out of order.

Some pieces of advice about animations

Normally, the best way to create an animated graphic is first to create a sketch of the animation, drawing some lines representing the graphic. When this animation looks good, edit the first frame and paint the definitive graphic in detail, on the sketch. Later, use the same techniques and tools to complete the rest of frames.

When long animations (made up of many frames) are retouched, occasionally it is advisable to focus on a specific part of the animation. For that, minimise the rest of frames

The best way to save the animations in the disk is through a **file FPG**. If you don't have any file yet, create a new one (**files \ new...**). Then, drag the first frame to the file window, input a code and describe it. Next, do the same with the rest, in order (the program will suggest you consecutive codes for the rest of frames). When you wish to work again on this animation, tag all the frames in the file window and use the option **files \ load tags**.

If the animation is not made up of many frames, and they are not very long, you can use a trick to arrange the animation's frames without having to put the maps one on each other. Thus, **pass the mouse pointer over all the maps in inverted order** (first over the last one, then over the penultimate one, etc.) until you reach the first one. Then, enter the graphic editor by double-clicking and the frames will be similarly arranged. The only problem is that they will get out of order if you exit the graphic editor and move the mouse pointer over the desktop in a disorderly manner.

4.12 Tricks and advanced drawing techniques

This last section dedicated to the **graphic editor** deals with the description of some tricks and techniques used by the graphic artists of Hammer Technologies to obtain better results in the graphics of the games, obviously adapted to the possibilities of DIV Games Studio's graphic editor.

We recommend you to read this section only when you have a perfect knowledge about all the possibilities of this editor.

Use of scaled
The first technique deals with painting graphics in double size and then, halve them. Thus, they will gain quality and definition.

For instance, to create a 100 by 100 pixel picture, it is done first 200 by 200 pixels size and then, it is re-scaled to its real size (with the option **maps \ re-scale...**).

For instance, try to paint a picture with white strokes, against a black background, split into little closed sections. Then, colour these sections (with the normal filling tool, in different colours), delete the white strokes (with the diagonal filling tool and the black colour) and finally, halve the map.

Another curious technique. In the previous picture, instead of deleting the white strokes, select them with the block edit tool (with the filling selection, by clicking on these strokes) and apply the softening effect on them several times (with the effects bar)...

Use of masks to replace colours
Occasionally, the aim is to replace a map's colour by another different one. This can be done as shown below.

For instance, create a map with several doodles by pen in different colours. Now, to replace one of the colours by another one, select the masks dialog (key **M**), choose the colour to be changed (for that, you can click on the map) and click the button **Invert** (to protect against writing all the colours, except the one you intend to change). Finally, select the rectangles tool (filled boxes), choose the new colour and paint a rectangle occupying the entire map (from corner to corner) ... and don't forget to remove the mask to go on working!.

Use of the windows
Rather than a technique, this is a piece of advice. Always use the possibilities of the DIV environment to back up the work. When you are going to "improve a graphic", it can be advisable to select it and cut and paste it in a new map window. Thus, it will always be possible to recover the original graphic "if the improvement doesn't improve". When these security copies don't exist any longer, close the windows.

Use the keys to adjust blocks
When you are creating a mapped background for a game, that is to say, a décor from basic blocks that are repeated several times, use the cursors to correctly adjust the blocks, as they will always be more secure than the mouse as far as movements are concerned. Remember that the **Num. Lock** key of the keypad must be disabled for the cursors to move the block pixel by pixel.

Creating colours ranges
A proper definition of the colours in a game can make the work inside the graphic editor easier.

When you need to carry out a **transition from a colour to another one** in a picture, access the colours range editor (key **C**) and select one of the ranges that you are not using to redefine it. Define a range of **16 or 32 colours, editable every 8 colours**. Then, place one of the colours in one of the ranges positions (by clicking on the little icons with a gray up arrow) and the other one in the following position (eight colours farther to its right or left). For you, the program will search for **the best possible transition** in the palette from one

colour to the other one (the intermediate colours between both). It will be useful to find transitions between different colours, such us from red to blue, from green to brown, etc.

Always carry out tests by selecting colours from the range and reassigning them to another position of it (first clicking on a range colour and then, on one of the little gray icons), to obtain different sequences of colours that can be useful for you.

Redefine the transparent colour to avoid aliasing

Aliasing implies that the pixels of a graphic's outline stand out too much. If the game is going to be developed against a background of a specific colour, or against some specific shades of colour, it is possible to create graphics whose outlines are better hidden in the background with the technique explained here.

For instance, create a new map of 80 by 80 pixels and paint a graphic (e.g., two separated circles, the first one in white and the other in dark colour). Leave a free margin of one pixel, at least, around the graphic (empty in black colour). Supposing now that this graphic must appear against a blue background representing the sky in the game, follow the steps described below:

- Enter the palettes editor (**palettes \ edit palette...**), select the transparent colour (the black colour located in the upper left part) and modify it so that it becomes the colour of the alleged game's background (the blue you prefer, for instance, 25% of green and 50% of blue will create a sky blue). Now click on **Accept** and answer **Cancel** to the question of whether you wish to adapt the maps.

- Edit the graphic, you will see the graphic against the blue background, with its outline aliased. In the masks dialog (key **M**), hide the transparent colour (click on the first palette's colour, the blue one). Thus, you avoid to modify the exterior of the graphic.

- Now, to soften the aliasing, select the pen, expand the picture (with the magnifier, zoom by 8) and carefully pass it along the graphic outline with the **D** key (smooth) pressed. The colours of the graphic outline will approach the background colour. Continue to apply the effect as long as necessary.

- Finally, remove the mask (once again clicking on the transparent colour in the masks dialog), return to the palettes editor, redefine the transparent colour to black, click on **Accept** and again, answer **Cancel** to the question so that the maps are not adapted.

The transparent colour can be defined when you want to see the outline of the graphics against a specific colour. Keep in mind that you can always stress the transparent colour with the **B** key inside the graphic editor.

Filling with textures

To fill a graphic with a texture, instead of with a solid colour, you can move the selections with the **Control** key, for instance:

- In a map, put a texture on one side (if you have no texture, you can quickly create one with the spray bar) and, on the other side, paint a circle of any colour (filled). The texture must be greater that the circle.

- Now, to fill the circle with the texture, use the block edit bar and select the circle (with the filling selection mode).

- With the **Control** key pressed, move the circle selection to the texture, then click on the move icon and shift the block again to the circle.

Pass graphics from a map to another one with TAB

The possibility given by animations to change the map, inside the graphic editor, by pressing the **TAB** key, can be used for many other tasks.

If you have several maps **of the same size**, with different graphics in each one, you can select a graphic in one of the maps, click **the move** icon in the edit bar and then, use the **TAB** key to copy the graphic to another map.

It can also be used to obtain the filling with textures explained in the previous section, having the texture in another map. Select the graphic to be filled in one of the maps, press **TAB** to pass the selection to the map with the texture, cut in it the block (with the **move** icon) and return to the map with the graphic by pressing the same key again.

Summary

After all we've mentioned plus the possibility of merging palettes, the generator of explosions, the letters fonts and managing filling with gradient, the blocks operations, the re-scaled, the ranges editor, ..., you can consider yourself a professional graphic artist, providing that you are not daltonic, unfortunately.

Chapter 5
Creating Programs

5

CHAPTER 5: Creating Programs. Basic Concepts

This chapter describes many of the terms related to programs which are essential to understand how to create programs. We recommend you to read it even if you are an expert in programming or know a lot about it. Even if you are able to recognise the concepts that are explained here, you should learn the terms which are used to name them in the DIV language.

If you have never made a program, it could be hard to understand some of the terms that are explained here, but you must not worry, you'll be able to understand them later on once you see them used in practice.

5.1 Definition of a program

Basically a program is a series of written orders that the computer must **execute** one after the other so that the expected results are achieved.

Programs are actually the group of orders, data, graphics and sounds, etc. which produce the final result, but in DIV programs will be referred as the **list** of programs, that is, what we must type in the edition windows **in order to create the code which governs what the videogame does.**

A DIV program consists of three big parts: one to define the program **data,** another to define the game main orders (orders are called **statements** in the program) and another to define the orders of the different type of processes.

For **processes** we understand the different items inside a game, i.e. the graphics in motion (also called **sprites**). For example, in a space-invaders game, the spacecraft driven by the player is a process, each enemy is another process, each shot is another process and so is each explosion, etc.

Appendix A illustrates the generic outline of a program in DIV language.

5.2 Definition of data

Data are the key to programming. They are a very easy concept but somehow difficult to describe. If you already know the basis of programming then all you have to know is that in the DIV language for **data** we understand the programs **variables, tables** and **structures**. It is a generic concept which includes those three.

If you have never created a program, you should know that computers have a memory where they can store values. **Data** are references to specific positions of the computer

memory containing a **numeric value** which is used in a program. These data are given a **name**, let's say for instance we call a datum **"counter"**, and this name is used in the program to refer to that numeric value.

A program can give a specific value to the datum called **"counter"**. To give the datum the numeric value **123**, the program will use a **statement** (order) like the one below:

```
counter = 123;
```

Programs can also check the value of the data or use them in expressions; for instance, to give a datum called **"mydatum"** the value of the **"counter" plus 10**, we will used the following kind of statement:

```
mydatum = counter + 10;
```

Before the computer executed this statement, **mydatum** could have any numeric value but after the statement was executed, mydatum will have exactly the value that results from adding **10** to the numeric value the datum **"counter"** has at that time.

Definition of data

Data can be classified as **pre-defined** or **defined in the program**.

Pre-defined data are names already reserved in the language to refer to certain numeric values. For example, all processes (graphic items of the games) have two pre-defined data called **x** and **y**, where its coordinates (the situation of the drawing of the process on the screen). Therefore it is not possible to create more data called x or y, since these two names refer to the pre-defined data in which the coordinates are stored.

The data defined in the program are the new data used by each program to make calculations, etc. The part of the programs which declare new data are similar to the following one:

```
GLOBAL
    newdatum1 = 33;
```

This will mean this program is reserving a position in the computer memory to contain a numeric value, which will initially be 33 and that the program will refer to such position with the name "newdatum1". Program data names are always names the programmer makes up. We can define as many different data in a program as necessary.

Types of data

This kind of data, such as **"counter"**, **"mydatum"** or **"newdatum1"**, are called **variables** and are the simplest data, that is a **name associated to a numeric value which is stored in a position of the computer memory.**

But there are other types of data which are more complex: the **tables** and the **structures**.

A **table** is a list of **variables**, i.e. it is a **name** associated not with a single memory position but with as many as it is necessary. To define a table in a program, we make a declaration like this:

```
GLOBAL
    mytable1[ 3 ] = 0, 11, 22, 33;
```

In a program, the above lines will declare "**mytable1**" as a list of 4 variables. You must take into account that we always start counting from the position 0, and that this table has up to position 3, as the number between the symbols [ ] (called **square brackets** and which must not be taken for brackets) indicates. Four positions of the memory will be reserved for "**mytable1**", and these positions will initially (before the program is started) be given the values 0, 11, 22, and 33.

When one of these memory positions have to be checked or modified in a program, the name of the table must be indicated and also a numeric value to specify which table position is to be checked or modified must be given between the square brackets.

For example, a statement to put value **999** in position **1** of "**mytable1**" (which had **11** as original value) would be as follows:

```
mytable[ 1 ] = 999;
```

Structures are the most complex data. They are like a file box (think of a filing cabinet containing information about a person in each file) which has a number of **notes** (like the name, address, telephone, identity number, of each person) in each file.

Structures are the filing cabinets and each of the files is called a **record**, and to each note within a file is called a **field**. For example, in a game the following structures could be defined to keep information about the position on the screen of three enemies:

```
GLOBAL
    STRUCT position_enemies[ 2 ];
        coordinate_x;
        coordinate_y;
    END = 10,50,  0,0,  90,80;
```

This file box would be called **position_enemies**, would include 3 files (with the numbers **0**, **1** and **2**, as the number **2** in **square brackets** shows), and each file would have two notes, the horizontal and the vertical coordinate of the enemy.

Technically speaking, **position_enemies** is a **structure** with **3 records**, each one with **2 fields**, that is **coordinate_x** and **coordinate_y**.

The list of values which goes next will place the first enemy at the coordinates **(10, 50)**, the second one at **(0, 0)** and the third one at **(90, 80)**.

For this structure **position_enemies**, **6 positions** will be reserved in the computer memory (3 records x 2 fields). Later on, when you wish to access one of these numeric values in the program, you will have to indicate the name of the structure, the record number and the field name. For instance, to introduce the value 66 in the memory position where the vertical coordinate (**y**) of the second enemy is kept (that is the 1 record, because the first enemy coordinates are in the 0 record), the following statement would be used:

```
position_enemies[ 1 ] . coordinate_y = 66;
```

In fact, it is very similar to tables except that after the record number you have to indicate the symbol "**.**" (a dot) and the name of the field.

Data scope
Data can still be classified according to another criteria: their scope. In this case, we find three kinds of data: **global**, **local** and **private**.

The **global** data scope covers the entire program; this means that whenever we define a **global datum**, this will be a memory position (or several positions if the datum is a table or a structure instead of a variable). This memory position can be accessed from any point in the program.

Any point in the program means any area of statements. There is one area of general statements in the main program and then each type of process has its own area of statements.

All **areas of statements** can be identified because they start with the word **BEGIN** and finish with the word **END**.

Therefore, when we say a datum is **global** it means that this datum (the same memory position) can be used in the main program as well as in any of the processes.

The **local** data are **names which actually refer not to a single datum** but to several. Among this kind of data are the above mentioned **x** and **y** used to keep the coordinates of the processes.

For example, there could not be a single memory position reserved to the datum **x**, since each process of the program (each game item) will have a different horizontal coordinate, a different value in **its x** variable. That's the reason why there will be a memory position reserved for the **x** variable of each process.

Thus when we use the name **x** in a program, we will access a different numeric value depending of which process uses it. Each process will access its own **x** coordinate by the **local variable x**.

The **global data,** on the other hand, always refer to a single value. For example, a global datum of a game could be the player's **score.** The name **"score"** will always refer to this value, no matter which process uses it. This way any process would be able to modify the player's score and all the processes would access to the same memory position using this name.

Finally, the **private data** are those which are **used exclusively in one process of the program**. For instance, if a game item requires a variable to count from **0** to **9** this counter will be kept in **private variable**, since the rest of the processes do not need to use this value.

That means that in a process we define a private datum called **"mycounter"**, this name **will not mean anything within another process.**

Summary
For example, when we speak of the **pre-defined local variable angle** we will be referring to a datum which is identified with the **name "angle"**, and which has the following characteristics:

- It is a **variable**, that is, a name which refers to **a single numeric value**, and not to a list of them or to a records structure.
- It has a **local** scope, and therefore all processes will have **their own angle variable**, each one of them with its numeric value (these can be the same or different).
- It is a **pre-defined** datum, this means that it is not exclusive of a program: **all DIV programs have this datum a predefined.**

In addition to this all we need to know now is that this is the datum in which the processes of the games defined their **angle** to three decimal places.

5.3 Numeric Values and Expressions

In the DIV language, the data can have whole numeric values within the range (**-2147483648** ... **+2147483647**). These are the numeric values for which there is space in a memory position of the computer.
DIV does not use numbers with decimals. It is not possible to specify **1,5**, a value will pass from **1** to **2** directly without intermediate values. It is also important to know that we must not use commas to separate the thousands, for example, twenty three thousand and forty should be written **23040** and not **23,040**.

When we need to do a more accurate calculation using a less numeric than the units, then we have the data used to count in **tenths, hundredths or thousandths**, and so on. This is called **fixed comma**. For example, if we use the local variable **x** (**x** coordinate) of a process in **tenth parts,** we will be able to specify the coordinate **1.5 units** by defining the value of **x** as **15 tenth parts,** which equals to it.

Every time a program expects a **numeric value** at a specific point, a numeric expression can also be specified.

Numeric Expressions
An **expression** is basically a mathematical formula which links one or more **operands** (**x**, **2**, **-7, counter**, ...) by several **operators** (*, **AND**, +, /, ...); examples of expressions would be: **2**, **2+3 or (x*4)/-3**.

As values we can only use integers within the range specified above and the result of the expression will also be truncated within that range.

The **operands** which can be used in a expression are, besides the numbers, all the data, functions and items of the programs.

The **operators** which can be used in a expression are the following: (synonyms of the operator are shown in brackets, if it has any)

- the basic **arithmetical operators** : **( )** Brackets, **+** Addition, **-** Subtraction (or negation of the sign), * Multiplication, **/** Division and **MOD** Module (**%**).

- the **logical operators**: **NOT** Binary and logical Negation (**!**), **AND** binary and logical (**&**, **&&**), **OR** binary and logical (**|**, **||**), **XOR** (Or exclusive) (**^**, **^^**).

- the **comparison operators**: **==** Comparison, **<>** Different (**!=**), **>** Bigger than, **>=** Bigger than or equal to (**=>**), **<** Less, **<=** Less or equal to (**=<**).

- the **pointers operators**: **OFFSET** Direction or slide (**&**), **POINTER** Operator of addressing (*, ^, [ ]).

- the **operators of increments**: ++ Operator of increment, -- Operator of decrement.

- the **binary operators**: >> Rotation to the right, << Rotation to the left.

Note: the function of each of these operators can be learnt later on. You don't need to worry about it now.

5.4 Definition of a constant

Constants are a type of **names** (like the ones given to the data) used as **synonyms of numeric values**. For instance, we could use a constant called **maximum_height** as a synonym of the **numeric value 100**.

The **difference between a constant and a datum** is that no computer memory position is reserved to keep the value of a constant since such value never changes (**maximum_height** will always be **100**, once it has been so defined). Thus DIV will replace (when running a game) every constant with their respective value.

This means it would be the same to use **maximum_height** as to use **100** in the program. Constants are used to see more clearly the list of a program. In the example given, the constant will report that the number **100** used in the program is the **maximum height** (of any object or item in any game).

There are also **pre-defined and defined constants in the programs**. An example of **pre-defined** constant could be **min_int,** a synonym of the smallest numeric value a datum can take in the language (**-2147483648**), or **max_int,** synonym of the largest numeric value (**+2147483647**). A **defined constant in a program** would be the one used for the example above **maximum_height**.

When are constants used

Let's say that in a game several times we set **3** as **the maximum number of lives the hero has.** When we wish to raise or reduce that number, we would have to search and replace that number for the program and we run the risk of replacing another number **3** which could be in the program for something else.

The alternative to this is to state a constant which we could call for example **maximum_lives** as a synonym of the numeric value **3** and to use such constant in the program instead of the number. Then, when we wish to change this value, all we'll have to do is replace that number in the constant statement **maximum_lives**.

Once a constant has been given a value, this value cannot be modified later in the program.

5.5 Names

A name is a sequence of alphanumeric characters used to **identify** an item in the program, such as the name of a process, of a constant or of a variable. These names can be created with the following characters:

Symbols: **_ # ª º $ P ƒ £ ¥ ¢**
Digits: **0123456789**
Letters: **abcdefghijklmnopqrstuvwxyz**
Letters (extended): **ñ ç æ ãäàåá êëèé îìí ôöòó ûüùú ÿ**

Rules to build new names in a program

- the sequence of characters must only contain characters included in the above list (except for the capital letters corresponding to the small letters of the list). The compiler doesn't care for differences between capital and small letters (**ABc** or **abC** are the same name for it).

- Inside each sequence we cannot leave blank spaces, i.e. for the compiler it is not valid as a name **enemy aircraft**. It will take that as two names. In this case, the name can be stated as **enemy_aircraft**, and we will use the underlining character (low dash) instead of a space.

- A name cannot start with a numeric digit like, for example, **9a.** However, after the first character every necessary digit can be included in the name (**a9** for instance would be a valid name).

- The name cannot be the same as one of the **pre-defined items** of the language. To check this out, you can go to the **glossary of terms**, or otherwise you can put the edit cursor on a name of the program and press **F1**; if that name corresponds to an item pre-defined in the language, the system will show a help page about it.

- The same name cannot be used for two different items. For instance, we cannot give a constant the name **thing** and, then, state a process as PROCESS **thing**(x,y);.

5.6 Items pre-defined in the language

Many names and symbols used in the programs have a purpose clearly defined in the programming language. They are the pre-defined items.

All programs respect a generic structure called syntax. This structure is visible trough several pre-defined items which appear in specific positions in the program.

Reserved Words

Reserved words are **names** which have been reserved as **key words** of the programming language. For instance, **PROGRAM** is a reserved word which always defines the beginning of a program.

Reserved words don't have a difference between capital and small letters. Thus **PROGRAM**, is as valid as **program**, or as **Program**, ...

Nevertheless, all examples in DIV Games Studio show the words in capital letters. The only reason for this is to help you recognising them more easily.

Pre-defined Data

Pre-defined data are the variables, tables and structures which all the programs created through DIV are going to have. These are basically used to control the different computer devices and games graphics.

There are **global** data and **local** data (there are not pre-defined private data, these must be defined in the programs themselves).

Global data are the information which every program needs. They are mainly used to control devices such as the screen, the mouse, the joystick, the keyboard or the sound card.

Local data are the information **about the processes** of the games. The system needs to know things such as which is the graphic of a process, what position it is on, its size, its angle, its depth plane, etc. These data must be indicated in **local pre-defined variables** of the processes. They all have a valid value default and therefore only those values we wish to change will have to be set.

Pre-defined data of the DIV language are described in **Appendix C** of this book.

Symbols

Symbols are just characters or combinations of characters which also have a specific meaning within a program.

In contrast with the reserved words, symbols are not names. They never used any of the characters which can be part of the names (those described at 5.5).

Symbols are for example all the operators which can be used in the numeric expressions.

Symbols are always the same for all programs. There are not defined symbols in the programs. The items which can be defined in the programs will always be identified by names, either constants, data or processes.

5.7 Statements

In this chapter we talk a lot about numeric values, data and expressions of the programs. But programming is really done through statements. Statements are the orders you give the computer for each game. We have the following kind of statements:

- The **assignments** are statements used for calculations. A numeric expression is evaluated and the result is assigned to the datum on the left of the equation.

- The **conditional statements** permit to verify if a condition is fulfilled (for example if a process is at some specific coordinates, if the energy equals zero, etc.) and, in this case, to execute another group of statements.

- The **loop statements** allow to repeat a group of statements for a certain number of times or until a condition is met.

- The **calls to a function** are the statements which really allow to execute "visible" actions within a game, such as to put a graphic on the screen, to read the keyboard, to make a sound, to change the video mode, etc.

- The **calls to a process** are very much alike the calls to a function, only that they call one of the **PROCESS** blocks of the program itself. Call to processes are generally made to add a new process to the game (an item such as a bonus, an explosion, an enemy, etc.)

- The **control statements** are a number of generic statements, such as **FRAME,** which allow to visualise next frame in the game, or **RETURN,** which allows to finish a process, or **CLONE,** which gives you the possibility of creating a copy of a process.

Generally in a process there are a **loop statement** and a **FRAME** statement. The loop is used by the process to execute a number of orders in all the game frames and the **FRAME**, which appears like one of the statements inside the loop, is used to visualise the process graphic in the following frame (next frame). For example:

```
LOOP
    x = x+1;
    FRAME;
END
```

The LOOP ... END loop is the simplest; it indicates that the statements inside it must be repeated indefinitely, once and again. The statement **x = x+1;** is an assignment used to put in the **local variable x** (horizontal coordinate of the process) the result of the expression **x+1**, i.e. it adds one to this local variable and therefore moves the graphic of the process one point to the right. And the finally, the statement **FRAME,** will indicate the point in which the process graphic has to be visualised in a frame of the game.

5.8 Conditions

Conditions are **expressions** usually like the following ones:

```
x<320
size==100 AND graphic>10
y==0 OR (x>=100 AND x<=200)
...
```

They are used within some statements to check the program data. For that purpose the comparison operators are used. These are the following:

```
== Comparison of equality.
<> Comparison of difference (also valid!=).
>  Comparison of bigger than.
>= Comparison of bigger than or equal to (also=>).
<  Comparison of less.
<= Comparison of less or equal to (also=<).
```

Brackets and logical operators can also be used to link several check-outs within a condition. The logical operators are:

OR	it compares that at least one of the two expressions is true.
AND	it compares that two expressions are true.
XOR	it compares that only one of two expressions is true.
NOT	it compares that the following condition is not met.

Next, some conditions formulated with these symbols and operators are shown, together with a description in regular language.

x>0
this condition verifies that **x** is a bigger than number than **0**.

x>=0 AND x<=9
it verifies that **x** is a number between **0** and **9**.

y==x OR y==2*x
it verifies that **y** is equal to **x** or to **2*x**.

x<>y AND NOT y==0
it verifies that **y** is different from **x** and that **y** is not **0**.

(x<0 OR x>9) AND y==z
it verifies that **and** equals **z** and also that **x** is not between **0** and **9**.

5.9 Comments

A comment is a clarifying note about the program. Comments are not necessary for the program to work correctly. There are two kinds of comments:

- Of a single line: they start with the symbol **//** and finish at the end of the line in which they are defined.
- Of several lines: they start with the symbol **/*** and finish with the symbol ***/**.

Next is an example program with several comments.

```
PROGRAM my_game; // Comment of a single line.
/*
This is an example of a comment of several lines, in which we can have as  many
clarifying notes about the programs as we like.
*/
BEGIN
   FRAME;
END // the main program ends.
```

All the texts included in a comment are ignored by the compiler. It is possible to put as many comments as necessary in a program and at any point of the program. The comments starting with /* and finishing with */ (called comments of several lines) can also start and finish in the same line.

5.10 Functions

Functions are a number of **names reserved** in the language and used as statements to do many actions in the programs.

The functions which are available in the language are described thoroughly in **Appendix B.** Here is a summary of the available functions classified according to their task and a brief description of each one of them.

All functions are used by simply specifying in a point of the program its name and, in brackets, the values they require to specify the exact function they perform.

Functions of interaction between processes

collision() - It detects the collision between two processes.
get_angle() - It gets the angle towards another process.
get_dist() - It gets the distance to another process.
get_distx() - It gets the horizontal distance of an angle and a distance.
get_disty() - It gets the vertical distance of an angle and a distance.
get_id() - It gets identifying codes of a type of processes.
let_me_alone() - It eliminates the rest of the processes.
signal() - It sends a signal to another process (for example, to eliminate a process).

Mathematical functions

abs() - It gets the absolute value.
advance() - It advances the process coordinates in their angle.
fget_angle() - It gets the angle between two points.
fget_dist() - It gets the distance between two points.
near_angle() - It gets an angle near another in a given increment.
pow() - It raises a number to a power.
rand() - It gets a number within a range randomly.
rand_seed() - It initiates a series of random numbers.
sqrt() - It gets the square root.

Graphic functions

clear_screen() - It clears the screen
get_pixel() - It gets the colour of a pixel on the screen.
map_block_copy() - It copies a block of a map in another.
map_get_pixel() - It gets the colour of a pixel in a map.

map_put() - It puts a graphic in a map.
map_put_pixel() - It puts a pixel in a map.
map_xput() - It puts a graphic in a map, with effects.
put() - It puts a graphic in the screen background.
put_pixel() - It puts a pixel in the screen background.
put_screen() - It puts a map as screen background.
xput() - It puts a graphic in the screen background, with effects.

Music and sound functions

change_sound() - It changes the sound parameters.
is_playing_cd() - It reports if the CD is playing.
load_pcm() - It loads a new sound effect.
play_cd() - It initiates playing of a CD-Audio.
set_volume() - It sets the volume of the mixer.
sound() - It emits a sound effect through the card.
stop_cd() - It stops the CD-Audio playing.
stop_sound() - It stops a sound effect.
reset_sound() - It resets the system of sound.
unload_pcm() - It unloads a sound effect.

Entering functions

get_joy_button() - It gets the state of the buttons of the joystick.
get_joy_position() - It gets the position of the axes of the joystick.
key() - It gets the state of a key.

Functions to use the palette

convert_palette() - It converts the palette of a graphic.
fade() - It starts a fading effect on the screen.
fade_off() - It fades off the screen.
fade_on() - It fades on the screen.
load_pal() - It activates a new colour palette.
roll_palette() -It makes a colour roll with the palette.

Functions for scroll and mode-7

move_scroll() - It updates the coordinates of a scroll.
refresh_scroll() - It updates the background of a scroll.
start_mode7() - It activates a mode-7 window.
start_scroll() - It activates a scroll window.
stop_mode7() - It stops a mode-7 window.
stop_scroll() - It stops a scroll window.

Functions to print texts

delete_text() - It deletes a text from the screen.
load_fnt() - It loads a font for letters.
move_text() - It moves a text to another position.
write() - It writes down a text on the screen.
unload_fnt() - It unloads a font for letters.
write_int() - It writes down a numeric value on the screen.

Functions for animations

end_fli() - It ends up an animation.
frame_fli() - It shows next frame of an animation.
reset_fli() - It resets an animation.
start_fli() - It starts an animation FLI/FLC.

Screen regions functions

define_region() - It defines a region or window on the screen.
out_region() - It reports if a process is outside a region.

Information about graphics functions

get_point() - It gets the position of a control point in a map.
get_real_point() - It gets the real position of the control point.
graphicic_info() - It gets information about a map.

Initialising functions

set_fps() - It defines the number of frames per second in the game.
set_mode() - It defines the video mode.
load_fpg() - It loads a FPG file with graphics.
load_map() - It loads a map.
unload_fpg() - It unloads graphics from a FPG file.
unload_map() - It unloads a map.

Data recording functions

load() - It keeps the value of a series of data in a file.
save() - It saves the value of a series of data.

System functions

exit() - It exits from the game.
system() - It executes an external command of the system.

Note: You will learn the use of all these functions as you go on creating programs. All of the functions are not at all used in every game; each game uses only those it needs depending on the technique and on the effects it is going to offer.

You can obtain help in the environment itself about any of these functions just by typing its name in a window of a program and then pressing **F1**.

5.11 Processes

This chapter has already dealt with processes. There is a difference between the blocks **PROCESS** of the programs, which define the performance of a **specific type of processes,** and the **processes** of the game while running, which are objects of the game whose performance is governed by one of the blocks **PROCESS** of the program, depending on its type.

```
PROCESS <name of the process > ( <list of parameters > )

PRIVATE // Declaration of private data, if there are any.
  <declaration of datum >;
  ...

BEGIN // Start of the statements of the processes
  <statement> ;
  ...
END // End of the process
```

The blocks which define data and statements for a type of processes must start with the reserved word **PROCESS** followed by its name (the name by which the processes of that type are going to be identified) and its **call parameter** in brackets.

Parameters are a list of data in which the process is going to receive different values. The brackets are mandatory even if the process does not have parameters.

After this heading, there can optionally be a **PRIVATE** section where the data which are going to be used exclusively in the process are declared.

And, finally, the code for the process has to be specified. This code is a sequence of statements between the reserved words **BEGIN** and **END**.

A process generally corresponds to a type of item in the game, such as a spaceship, an explosion, a shot, etc. and within the process code usually a loop is implemented. Within this loop, all the values needed to visualise such item (graphic, coordinates, etc.) and, then, by the statement **FRAME** the order to visualise the object with the set features will be given.

Chapter 6
Practical Example

6

CHAPTER 6: A Practical Example

Let's get to the point. There are still many important concepts to be explained but this chapter deals with how to create a game step by step.

This chapter gives you a little break after all those boring concepts and computer terms.

The first game is going to be a very simple version of a space invaders game. This example will help to explain the general methodology to work with DIV Games Studio.

6.1 The graphic work

The first thing you need to create a game is an idea and then to create the graphics for it. We are not going to give you the graphics because we'd like you to do all of this by yourself, so we will just give you the necessary guidelines for you to create those graphics.

Note: do the required graphs for the game (we'll tell you which) quickly. Don't worry if they don't turn up too great: it's only a test.

- Load the palette default of DIV Games Studio (**palettes \ open...** indicating **div.pal**). This palette will be used by the game. Then select **palettes \ show palette...** to see it.

- Create a new map of 40 x 40 points, to design the main spacecraft (**maps \ new...**). Enter the editor by double clicking on the map and create the drawing of the spacecraft as a filled triangle pointing up and centered in the map.

All game graphs are to be introduced in a **file FPG** which will be loaded later in the game.

- Create a new file called **test1.fpg** (with option **files \ new...**) and drag the spacecraft graphic to this file, indicating **1** as graphic code. Create the file in the directory default (the directory called **\FPG** within **DIV**).

- Now do the same to introduce in the file a **star background**. Create a map of 320 x 200 points, select the colour white, the spray tool (the smallest size) and move quickly the drawing pointer all around the map until you get a dotting more or less consistent. Then introduce this graphic in the file with the code **2**.

- Finally, create a graphic (any graphic) for the **shot of the spacecraft**, approximately of 4 x 12 points (with the graphic code **3**) and an **enemy** of the same size as the main spacecraft; all you need is to draw a filled circle (with the code **4**).

Once the four maps are in the file, you can start programming the game. You can close all the maps to free space in the desk, since all of them are kept in the file (**maps \ close all...**).

6.2 The first tests

Create a new program (**programs \ new...** indicating as name of the file **test1.prg**), an empty window edition will appear. To start with, type the following list (you can use capital or small letters)

```
PROGRAM test1;
BEGIN
END
```

This is already a correct program in the DIV language, although it doesn't do anything. From now on, the main statements of the program must be defined **between the reserved words BEGIN and END** so that the computer will receive the different commands.

The first thing is to instruct the computer to load the file where the graphics have been introduced; this will be done with the following statement (write it down after **BEGIN**):

```
load_fpg("test1.fpg");
```

This statement calls the function of the language **load_fpg()** which loads the file with the graphics for the game. Now, create a **PROCESS** block to control the main spacecraft, right after **END** of the main program.

```
PROCESS spacecraft ()
BEGIN
END
```

Then you have to state in the main program that you wish to create a process of the type spacecraft, and you call the process (right after the loading of the fpg file) with the following statement:

```
spacecraft();
```

To display the spacecraft within the program, some of its local variables must be defined in order to define the position of the graphic for the spacecraft. The code of the graphic of the process must be indicated in the **graphic** variable, and its screen coordinates must be indicated in the **x** and **y** variables.

Then you have to use the statement **FRAME** to display the frames and some kind of loop to do this several times. Otherwise if the process visualises only one frame, the program will finish too soon. For instance, let's use the loop **LOOP ... END**, so that the statement **FRAME** is always repeated.

Taking into account that the graphic of the spacecraft is number **1** , if it is placed right in the center of the screen (in the coordinates **(160, 100)**, since the screen default will be 320 x 200 points), the spacecraft process would be as follows:

```
PROCESS spacecraft ()
BEGIN
    graph = 1;
    x = 160;
    and = 100;
    LOOP
        FRAME;
    END
END
```

Given that several statements can be put in the same line and that the spaces to separate names and symbols are not necessary, the three space variables could be defined in a single line such as the following one:

```
graph=1; x=160; y=100;
```

You can debug the program step by step now to see how the computer executes all the statements of that program; to do that press **F12** and, once you are in the **program debugger** (described at **2.10**), press the button **Debug** several times. To end press **ESC** (to exit the debugger) and **ALT+X** to exit the game.

You have already created a program which creates a process and shows its graphic on the screen.

6.3 Moving the spacecraft

To move the spacecraft, all the keys of the cursors will be used. The function **key()** can be used to check if a key is pressed. We will also need a conditional statement to specify that the spacecraft must only move when certain keys are pressed. Include these instructions right before the instruction **FRAME** of the spacecraft.

```
IF (key(_right))
    x = x+1;
END
```

This statement indicates that when the "right" curser key is pressed, **1** must be added to the horizontal coordinate of the spacecraft.

The statement **x=x+1;** will add **1** to the local variable **x**. This can also be done with the statement **x+=1;** which is an abbreviation of the one above. The conditional statement **IF()** ... **END** will execute the statements inside it when the condition between brackets is fulfilled.

Then, the statement above could have also been specified in a single line such as:

```
IF (key(_right)) x+=1; END;
```

You can try to execute the example with the key **F10**. Press **right cursor** to move the spacecraft point by point in that direction and then **ALT+X** to return to the desk.

You can complete the motion of the aircraft adding these three statements:

```
IF (key(_left)) x-=1; END
IF (key(_down)) y+=1; END
IF (key(_up)) y-=1; END
```

If you run the program now you will see that you are able to move the spacecraft around the screen, even using the diagonals.

To move the spacecraft faster, you can replace the **1** of the lines above by another number, for example, let's replace them by **4**. And to move the spacecraft with the joystick (if you have one), you can replace **key(_right)** with **joy.right**, **key(_left)** with **joy.left**, etc.

6.4 Creating more processes

Now you can create the processes for the spacecraft shots. Write the following block after the block which controls the spacecraft.

```
PROCESS shot(x, y)
BEGIN
   graph = 3;
   LOOP
        y -= 16;
        FRAME;
   END
END
```

You can see an important difference with the spacecraft process. Now, the variables **x and y** are specified between brackets. These are called **parameters** and what they mean is that when we call the process shot two values must be specified in brackets and the process will take these values in its variables **x and y**. Therefore, the process cannot be called **shot();**, its coordinates will have to be indicated and it will have to be called for example **shot(160,200);**.

The code of this process defines its graphic (code number 3) and then it enters a loop **LOOP** ... **END** (an indefinite loop) subtracting **16** to its coordinate **y** (thus moving the process 16 points upwards) and shows an frame.

If you execute the program now you won't see the difference with the previous program; although there is a new **PROCESS** block, no process of this type appears in the game. The reason is that the process has not been called in any point of the program.

The shot must be called by the process of the spacecraft, for example when it detects that the **Control** key has been pressed. To do that the following statement must be used (right after the statements which detected the moves of the spacecraft):

```
IF (key(_control)) shot(x,y); END
```

This way the spacecraft will create a shot when it detects that the **Control** key is pressed and will pass as parameters the same coordinates (x,y) as for the spacecraft so the shot process receives in its coordinates (x,y) the same numeric values saved in the spacecraft variables.

If the move was programmed with the joystick instead of the keys, you have to indicate **joy.button1** and not **key(_control)**, to make shots.

You can try it pressing **F10** again. You will come to a small problem: the shots come from the middle of the spacecraft and that doesn't look right. To solve this problem you have to use a **coordinate y** for the shots (the second parameter) a less value (since the positive coordinates go downward and the negative go upward).

Try to replace the previous call by **shot (x,y-20);** (inside the statement **IF** ... **END**, of course) and the problem will be solved. The shot is 20 points higher.

When you execute the game and you shoot for a while you will find another problem: the game goes more and more slowly. This is due to the fact that there are more and more processes of the type shot active in the game and when there are 500, 1000 or more processes moving at the same time (even if you don't see them on the screen, since they are higher), the system takes longer to process them all.

You can stop the game with **F12**, calling the debugger, to check in the upper right corner the **number of processes** the program has at each time.

In order to solve this, you must modify the processes of the type shot so that they are eliminated when they appear on the screen. You can do it in several ways, for instance, replacing the loop **LOOP** ... **END** with the following:

```
REPEAT
    y -= 16;
    FRAME;
UNTIL (y<0);
```

The statement **REPEAT** ... **UNTIL()** is also a loop statement which repeats the intsructions within it several times, however it won't do it indefinitely but when the condition indicated in the brackets of **UNTIL** is complied with; i.e. until the coordinate **y** of the process is less than **0** (until the process gets out of the screen).

Now you can shoot as much as you like since when the shots get out of the screen they will also get out of the loop; therefore when they reach the **END** of their **BEGIN** they will stop running. Use the debugger again to verify it.

6.5 Adding enemies

We already have a spacecraft which shoots; now we are going to try something more difficult: to add enemies processes. We are going to create a process which, unlike the shots, goes down until it disappears trough the bottom of the screen. Insert the next block after the shots.

```
PROCESS enemy (x, inc_x, inc_y)
BEGIN
   graph = 4;
   y = -20;
   REPEAT
        x += inc_x;
        y += inc_y;
        FRAME;
   UNTIL (y>220);
END
```

This process receives three parameters: the horizontal coordinate (the vertical one will be set by the process itself with the statement **y=-20;**, twenty points over the screen) and then two values **inc_x** and **inc_y** which are two names of invented data (they are two new names, we could have used any other).

These two data will be exclusive of the processes of the enemy type. In this case they will be used to define the horizontal and vertical increment of the process per frame, i.e. the number of points which will change its coordinates **x** and **y**.

It will also set its graphic, which was number **4**; then the process will stay in a loop, and each frame will add these increments to its coordinates until the coordinate **y** is a number bigger than **220**. Thus the enemy will have for sure exited the bottom of the screen and its execution will be finished.

We'll make the general program create the enemies processes; to do that, after the call to the spacecraft process, the following statements have to be included:

```
LOOP
   enemy(rand(0,320),rand(-4,4),rand(6,12));
   FRAME;
END
```

A loop has also been created for the main program, so that each frame creates a new enemy. The **FRAME** statement is mandatory, because even if the main program hasn't got any graphic to display in this case, it must anyway pass the images so that the program is not stopped.

The function **rand(minimum value, maximum value)** receives a **number randomly** which ranges between the two provided.

This way the **enemy(x, inc_x, inc_y** process will get a horizontal coordinate at random between **0** and **320** (any position from the left to the right of the screen), a horizontal increment between **-4** and **4** (the enemy will be able to move from 4 points to the left to 4 points to the right, in each frame of the game) and a vertical increment between **6** and **12** (thus the enemy will go down the screen between 6 and 12 points per frame).

You can press **F10** to try it. And yes,..., it's true, perhaps there are too many enemies. Then we can reduce the frequency of enemies appearing. Instead of making the main program create one enemy per frame, we will specify a certain frequency of appearance.

This can be done several ways. For example, we are going to use a conditional statement to verify if the random number between **0** and **100** is less than **30**. To do that, the call to the enemy process of the main program have to be inserted in an **IF()** ... **END**, as shown below:

```
IF (rand(0,100) < 30)
    enemy(rand(0,320),rand(-4,4),rand(6,12));
END
```

Now, the call to the enemy process will not be made every time, but only in **30%** of the frames, as a general average. The reason for this is that on obtaining the numbers randomly, several of them between 0 and 30 could come out consecutively, and then many number bigger than 30, etc.

6.6 Retouching the program

To include the star background you drew before, you'll use the function **put_screen()**, indicating after the load of the file FPG in the main program the following line:

```
put_screen(0,2);
```

This function requires two parameters. The first one is the number of the file where the graphic we wish to use as background is located; the first file loaded in the program is the file **0**, the second one is **1**, and so on. Logically since only one file has been loaded in the program, this one will be the file number **0**. The second parameter is the number of the graphic (graphic code) within the file and the star background was inserted with number **2** in the file.

Now we will use a little trick to avoid seeing the enemies so much alike: we will change their size. We will define their local variable **size**, which indicates their size in percentage (by default is **100**, which is the original size) as a random number between **25** and **100**, this way some enemies will be bigger than others.

This can also be done in different ways; for instance, a new call parameter can be added in the **enemy** processes indicating that instead of three values, they must have four and that the fourth one will be in their **size** variable. The head of the process which controls the enemies will then be as follows:

```
PROCESS enemy(x, inc_x, inc_y, size)
```

But now the call has to be modified too so that the fourth value is sent to this processes as a random number between **25** and **100**. The **rand()** function will be used again and the call to the enemy process of the main program will be as follows:

```
enemy(rand(0,320),rand(-4,4),rand(6,12),rand(25,100));
```

The calls to very long processes or functions (those with many parameters), like this last one, can be split in several lines so that the program does not go too wide. This can be done by dividing the line practically at any position, for instance:

```
enemy(rand(0,320),rand(-4,4),
   rand(6,12),rand(25,100));
```

Instead of modifying the call parameters of the enemy process, the following line could have been included after **BEGIN** of the enemy block:

```
size = rand(25,100);
```

And, as a last retouching for now, we'll make the spacecraft always appear in the lower part of the screen. To do that the statements which used to allow the vertical move of the spacecraft have to be eliminated (in the **y** coordinate); these statements were:

```
IF (key(_down)) y+=4; END
IF (key(_up)) y-=4; END
```

Once these two lines have been deleted (or commented by a **//** symbol at their start), the spacecraft have to be placed in the lower part. To do that, the statement **y=100;** of the spacecraft block (which used to place the **y** coordinate of the spacecraft in the middle of the screen) will have to be replaced with the statement **y=180;** (which places the spacecraft in the lower part of the screen).

6.7 Destroying processes

This example is looking more and more like a real game. We are sure you must be willing to know how **to kill** the enemies, because it's unconvincing that the shots do nothing to them.

We will not only make the "**laser**" shots from the spacecraft to eliminate the enemies, we will also program an explosion to make its destruction more real.

To detect the collision between two processes we use the function **collision()**, which requires as parameters the indication of the **type of processes** to verify if there is a collision. Add the following line in the **enemy** process loop (anywhere in the statement **REPEAT** ... **UNTIL()**):

```
IF (collision(TYPE shot)) BREAK; END
```

The **TYPE** operator obtains the **type of process** and indicates afterwards its name. The **collision()** function will indicate if the process of the enemy is colliding with any shot, if that were the case, the statement **BREAK** will be executed.

This statement is used to **exit from a loop**, i.e., when the instruction **BREAK** is executed, the **enemy** processes will exit from their **REPEAT** ... **UNTIL()** loop, and they will reach the **END** of their **BEGIN** and end their execution. You can see how the statement **BREAK makes the loop to exit**, and it is not necessary to comply with the **y>220)** for that.

To exit the loop when the collision with the shots is detected, we could have also modified the exit condition of the loop in the sense that the loop would be repeated until the coordinate **y** were bigger than **220** or until the collision were detected; to do this the following condition will have to be modified in the **UNTIL** of the enemies:

```
UNTIL (y>220 OR collision(TYPE shot));
```

In this case the above conditional statement (**IF** ... **END**) wouldn't be necessary.

In order to program the explosion we have to create it first. To do that, you must access the explosion generator (**maps \ explosion generator**). Set the size **40** x **40** points (that is the enemy's size), **6** frames (the value by default) and select as the three colours, top down, dark red, bright red and yellow.

After pressing **Accept**, **6** new windows will appear on the desk. You have to drag them in order onto the file **test1.fpg** of the game (following the sequence of the explosion) and indicate as codes of the graphics numbers from **5** to **10.**

To display in the game, another loop is going to be done in the enemies processes, after **REPEAT** ... **UNTIL()**. Thus when they exit from it, the explosion is displayed instead of ending the program. To do this, the **graphic** have to pass all values from **5** to **10**, giving a frame with each one of them. In this case another loop statement will be used: **FROM**. This is one of the easiest and most versatile statements. In this case, add the following statement (after **UNTIL**):

```
FROM graph = 5 TO 10 ;
   FRAME;
END
```

And we already have the game explosion. The **FROM** uses a variable as a counter, from an initial value to a final one, and executes the instructions inside (those between **FROM** and **END**) for all these values of the variable. The explosions will have the same size as the enemies, they'll be small if the enemies are small or big if they are big because the variable **size** hasn't been modified and, although the graphic changes, each explosion continues to be the same process which controls the enemy, with the same data.

What are you waiting for? press **F10** again to verify the effect of the explosions.

6.8 Last minute changes

Basically the sample mini game is finished; now it's up to you to go on advancing and improvising your own modifications.

But before ending this practical chapter, we are going to suggest some interesting modifications.

For example, to improve the control of the protagonist spacecraft you can do it with the mouse. To do that the two **IF** ... **END** which controlled the keys of the cursors by the statement **x=mouse.x;** (which assigns the spacecraft the coordinate **x** of the mouse pointer) will have to be replaced. And to shoot with the mouse, replace the condition of the following

IF (the so-called **key(_control)** function) with the condition **mouse.left** (which indicates that the left button of the mouse is pressed).

Note: You can look at the next list to check how the program looks with all these changes.

You can put a mouse pointer in the program by assigning its code of graphic to the variable **mouse.graph** (for example, **mouse.graph=2**; will be a fun effect).

Important: you must not worry if a game goes too fast or too slowly, because the speed can be changed with function **set_fps()** (set frames per second). This function requires two parameters between brackets; the first one is the number of frames per second the game must give and the second the number of displays which can be omitted when the game is not run in a fast enough computer. For example, try to put the statement: **set_fps(100,10);** at the beginning of the program (after **BEGIN** of the main program)

6.9 List of the program

In case you got lost at any point in this chapter, below is the list of the program as it will be after all modifications in the version controlled by the mouse.

```
PROGRAM test1;
BEGIN
  load_fpg("test1.fpg");
  put_screen(0,2);
  aircraft();

  LOOP
    IF (rand(0,100)<30)
      enemy(rand(0,320),rand(-4,4),rand(6,12));
    END
    FRAME;
  END
END

PROCESS aircraft ()
BEGIN
  graph=1;
  x=160;
  y=180;

  LOOP
    x=mouse.x;
    IF (mouse.left)
      shot(x,y-20);
    END
    FRAME;
  END
END
```

```
PROCESS shot(x,y)
BEGIN
  graph=3;

  REPEAT
    y-=16;
    FRAME;
  UNTIL (y<0);
END

PROCESS enemy(x,inc_x,inc_y)
BEGIN
  graph=4;
  y=-20;
  size=rand(25,100);

  REPEAT
    x+=inc_x;
    y+=inc_y;
    FRAME;
  UNTIL (y>220 OR collision(TYPE shot));

  FROM graph=5 TO 10;
    FRAME;
  END
END
```

If you have been able to follow this chapter till now, we are sure you will be excited to show a copy of the program to a friend. To create this copy in a floppy you need a formatted disc (3'1/2 inches) and to follow the steps described at **2.1 / Creating installation...**

We recommend ...

You can press **F1** on any name of the system variable, reserved word of the language or program statement to obtain more information. We recommend to examine the **examples of the language functions**, which are very small programs (easier to understand than the sample games) where you can find a great number of tricks and interesting techniques.

You can go back to point **5.10** of this book to see the list of functions and thus to be able to determine those you are interested in. There are many interesting things: adding scores (text functions), sliding the screen background (scroll function), adding sound effects, three-dimensions effects (mode-7 functions), etc.

We also recommend to go back now to **2.10** to practice with the **program debugger**, using the example you have created.

And, of course, we recommend to read **chapters 7 and 8** where the rest of the necessary concepts for programs creation is described.

Chapter 7
Program Structure

7

CHAPTER 7: Structure Of The Programs

This chapter reviews the syntax of a program in detail. A summary of this syntax can be found in **appendix A.**

The most advanced concepts about the program are described in **chapter 8**, thus avoiding to mix them with the syntax. However, some of them are used in this chapter. Thus, it is advisable to consult the following chapter in order to obtain information about **identifying codes, types of processes, states of processes**, etc.

7.1 Head of the program

PROGRAM <name>;

All the programs must start with the reserved word **PROGRAM** followed by the name of the program and a symbol ; (semicolon). This head is obligatory in all the programs. Before it, only one or several comments can optionally appear.

7.2 Declaration of constants

CONST
  <name> = <numeric value>;
  ...

This section of the program is **optional**, as its purpose deals with setting a series of synonymous numeric values.

In a game, for instance, number 3 has been set in one or several points of the program as the maximum lives of the leading character. If the aim is to modify this number, increasing or decreasing it, it will be necessary to look for this number and to replace it in the program. But there is a risk of replacing other '3' numbers appearing in the program with different aims.

An alternative is to declare a constant called, for instance, **maximum_lives** as a synonymous of the numeric value 3 and use that constant in the program instead of the number. Now, if the aim is to replace this value by another one, it is done simply in the declaration of the constant **maximum_lives**.

This section then establishes a list of names that are going to represent a series of numeric constants. This section must obligatory start with the reserved word **CONST** and then, for every declared constant, its name followed by the symbol = (assignment symbol) and a

constant expression (numeric value) must appear. After the declaration of every constant, the symbol ; (semicolon) must appear.

Once a value has been assigned to a constant, it won't later be possible to modify the former in the program.

7.3 Declaration of data

In a data declaration, three different kinds of objects can appear: a **variable**, a **table** or a **structure**.

In general, a **variable** will store a simple numeric value. A **table** will store a list of numeric values. And a **structure** will store a list of records of several fields (such as a list of index cards with varied information).

Note: All the data will be declared with a name which, from that moment, will become the means to access or modify the information contained in those data.

Each data will belong to a specific area, depending on whether its declaration has been made inside the **GLOBAL**, **LOCAL** or **PRIVATE** sections. These three sections are **optional** (they may not appear in the programs if it is not necessary to declare data of these types).

It is possible to access all the **global data** from any point of the program. **Local** data belong to all the processes (every process has its own value in them). Finally, **private** data belong to a single specific process.

Note: These three sections **must always appear in this order** (GLOBAL, LOCAL and PRIVATE, when all of them appear)

Declaration of global data

```
GLOBAL
  <declaration of datum> ;
  ...
```

This section of the program is optional. Global data, that is to say, the **data that can be used from any point of the program**, are declared in this section. Therefore, a global datum can be used for all the program's processes.

The section must start with the reserved word **GLOBAL** followed by a series of **declarations of data** finished with a symbol ; (semicolon).

In general, all those data that establish general conditions of the game related to several processes are declared as global data. An example could be the score obtained by the player, that could be stored in the **score** global variable. Thus, any process of the game could increment it, if necessary.

Declaration of local data

```
LOCAL
  <declaration of datum> ;
  ...
```

This section of the programs is optional, as the local data, that is to say, the **data that all the program's processes have** are declared here, each one with its own values (such as the **x** and **y** predefined local variables determine the coordinates of all the processes).

The section must start with the reserved word **LOCAL** followed by a series of **declarations of data** finished with a symbol ; (semicolon).

In general, the important information of the processes, that is to say, **the data to be consulted or modified from other processes**, are declared as local data.

The remaining energy of a process (a spacecraft, a shotgun, the leading character, etc.) could be an example. This information could be stored in the **energy** local variable, so any process can access or modify the energy of the rest (for instance, on colliding with them, energy could be subtracted).

Note: If a datum declared as local is to be used only inside one process, then the former can be defined as a private datum.

Declaration of private data

```
PRIVATE
  <declaration of datum> ;
  ...
```

These sections of the programs are optional. Private data, that is to say, **data that are going to be used exclusively inside a process**, can be declared in this section.

This section can appear either in the main program or in any other process of the program, as the main program is also considered as a process.

This section is defined just before the **BEGIN** of the process that is going to use these data and must start with the reserved word **PRIVATE** followed by a series of **declarations of data** finished with a symbol ; (semicolon).

In general, all the data that are going to contain information necessary only for a process, as well as those that can not be accessed from any other process, are declared as private data.

Those variables that are going to be used as counters in a loop, variables to contain angles or secondary identifying codes, etc. are normally defined as private data.

Note: If you need to consult or modify a datum declared as private from another process (datum.identifier), then this datum will have to be declared local (inside the program's section **LOCAL**). Thus, all the processes will have the datum and every process can access its value or the value that this datum has in another process.

Declaration of a variable

<name>

(or, if you want to initialise it)

<name> = <numeric value>

To declare a variable inside a section, it will be enough to indicate its name inside that section. In this case, the variable will be initialised at 0 (zero).

If the aim is to initialise the variable at other values, the symbol = (assignment) will be put after the name of the variable. The constant value at which the variable is intended to be initialised will be put after this symbol.

A variable is a cell (or position) of the computer's memory to which we refer by its name and that can contain whole numeric values.

Declaration of a table

<name> [<numeric value>]

(or, if you want to initialise the table)

<name> [<numeric value>] = <list of numeric values>

(or, if it is initialised without defining its length)

<name> [ ] = <list of numeric values>

To declare a table inside a section, it will be enough to indicate its name followed by the length of the table in square brackets. In that case, all the positions of the table will be initialised at 0 (zero).

The table's length is expressed as the maximum value of its index. That is to say, all the tables range from the position 0 to the position indicated in the square brackets in their declaration. For instance, a table declared as my_table[9], will be a table of length 10 (of 10 positions, from my_table[0] to my_table[9]).

If the aim is to initialise the different positions of the table, it is necessary to put the symbol = (assignment) after the previous declaration and, after this symbol, a list of numeric values.

If the table is initialised with a list, then it is not necessary to indicate the table's length in square brackets, as the compiler will create a table with as many positions as the number of values included in the list.

A table is a series of cells (or positions) of the computer's memory that is called by its name, appearing after it, in square brackets, the number of cell inside the table intended to be accessed.

For instance, if we declare a table as the following one:

```
my_table[]=33, -1, 6, -3, 99;
```

we will be declaring a table whose name is my_table and that has 5 cells (or positions), from cell no. 0 to cell no. 4. In the previous declaration, cell 0 (my_table[0]) is initialised with the value 33, cell 1 (my_table[1]) with the value -1, etc.

The language allows us to access cell 0 simply with the name of the table (my_table), as if it was a variable, omitting the zero in square brackets that should appear after. That is to say, for the compiler, my_table[0] will be the same as my_table (the first cell of the table).

Declaration of a structure

```
STRUCT <name> [ <numeric value> ]
  <declaration of datum> ;
  ...
END

(or, if the structure is initialised)

STRUCT <name> [ <numeric value> ]
  <declaration of datum> ;
  ...
END = <list of numeric values>
```

To declare a structure inside a section, it is necessary to put the reserved word **STRUCT** preceding its name. After it, the number of records of the structure must be indicated, in square brackets.

After this head defining the name of the structure and the number of records, all the data that belongs to the structure and that will comprise its fields, will be declared. Finally, the reserved word **END** must appear to finish the declaration.

The records' number of the structure is expressed as the maximum records' number of the structure. That is to say, all the structures have from record 0 to the record indicated in the square brackets. For instance, a structure declared as **STRUCT my_structure[9]**, will be a structure of **10 records** (from the record my_structure[0] to my_structure[9]).

A structure is like an index card file (records), each of them with different written information (fields). For instance, a structure in which we could include the initial and final positions of a series of processes of a game could be as follows (an index card file with 10 cards, each of them indicating the initial (x, y) and the final (x, y) of a process):

```
STRUCT movement_enemies[9]
  x_initial;
  y_initial;
  x_final;
  y_final;
END
```

This structure, that would be accessed with the name **movement_enemies**, has ten records and four fields in each record (two coordinates that determine the initial position of the process [x_initial ,y_final], and two that determine the final position [x_final, y_final]). **Movement_enemy[0].x_final** would be used to access the **x** final of the first enemy.

The language allows us to access the record 0 of the structure simply with the name of the structure (**movement_enemies.x_final**), omitting the zero in square brackets that should come next. That is to say, for the compiler **movement_enemies[0].x_final** will be the same as **movement_enemies.x_final**.

Each field of the structure may be a variable, a table or another complete structure, with its different records and fields.

If the aim is to initialise the structure (establishing the initial values of its fields in the different records), the symbol = (assignment) must be put after the reserved word **END** followed by a list of numeric values. If the structure is not initialised in this way, all the fields will be put at 0 by default.

Keep in mind that , in order to initialise a structure, the first values will be the values of the fields of the first record, the following ones those of the second record, and so on. For instance, if the following declaration is made:

```
STRUCT a[2]
   b;
   c[1];
END = 1,2,3,4,5,6,7,8,9;
```

First, it must be taken into account that the structure **a[ ]** has **3 records** (from a[0] to a[2]) and that there are **three fields** (b, c[0] and c[1]) in each record. Then, the previous declaration will initialise the structure in the following way:

```
a[0].b=1;
a[0].c[0]=2;
a[0].c[1]=3;
a[1].b=4;
a[1].c[0]=5;
...
```

Definition of a list of numeric values

The lists of values are basically a series of numeric values separated by commas and they are used **to initialise the values of tables or structures.**

An example of a list of constants is shown below:

```
1, 2, 3, 4, 5;
```

But, besides this basic definition, the use of the operator **DUP** is allowed to repeat a series of constants a specific number of times. For instance, the following list:

```
0, 100 DUP (1, 2, 3), 0;
```

It is a list of 302 constants (0,1,2,3,1,2,3, ...,1,2,3,0). That is to say, the operator DUP (duplication) allows us to repeat the sequence appearing after it in brackets, the indicated number of times.

It is possible to nest operations DUP. For instance, the following list:

```
2 DUP (88, 3 DUP (0, 1), 99);
```

would be equivalent to:

```
88, 0, 1, 0, 1, 0, 1, 99, 88, 0, 1, 0, 1, 0, 1, 99;
```

Moreover, the omission of the operator DUP is allowed; in other words, **2 DUP (0, 1)** is equivalent to **2(0, 1)**.

The operator DUP is specially useful to initialise structures. If, for instance, the aim is to initialise the following 100 record structure:

```
STRUCT a[99]
        b;
        c[9];
        d[9];
END
```

with the fields **b** initialised at **0**, the fields **c[9]** at **1** (all its positions) and the fields **d[9]** at **2**, the following list of initialisation would be used:

```
100 DUP (0, 10 DUP (1), 10 DUP (2)) ;
```

7.4 Main code

```
BEGIN
  <statement> ;
  ...
END
```

The main code of a program starts with the reserved word **BEGIN**. After it, any number of statements may appear. The main code finishes with the reserved word **END**.

This code controls the main process of the program, which initialises the program, controls the loops of the menu and game, and finishes the program.

An example of a main code's block is now shown:

```
PROGRAM my_game; // Head of the program
GLOBAL
  option; // Option chosen in the menu.

BEGIN // Beginning of the main code.

  Set_mode(m640x480); // Initialisation.
  Set_fps(24, 4);
  // ... Loads files, sounds, etc.

  REPEAT // Beginning main loop.

    option=0; // Control loop of the options menu.
    //... Initialises the options menu.
REPEAT
      // ... Actions to be performed in the menu.
      IF (key(_enter)) option=1; END // Playing is chosen.
      IF (key(_esc)) option=2; END // Finishing is chosen.
      FRAME;
    UNTIL (option>0);
    IF (option==1) // If the playing option has been chosen.
      //... Initialises regions, scroll, etc.
      //... Creates the game processes.
      //... Loop of game's control, waiting for its end.
    END
  UNTIL (option==2); // End of the main loop.
```

```
let_me_alone(); // Finishes all the processes.

END // End of the main code.

//... Declaration of the program's processes.
```

The end of the main code's execution does not imply the end of the program's execution, as it will continue if there are alive processes. If the aim is to force the end of the program when the code finishes, it is possible to use, for instance, the **let_me_alone()** function just before the **END** that marks the main code's end, in order to eliminate the rest of the processes that may remain alive.

The execution of the program can also be finished at any of its points with the **exit()** function, which will automatically eliminate all the processes.

7.5 Declaration of processes

```
PROCESS <name> ( <list of parameters> )
<Declaration of private data>
BEGIN
  <statement> ;
  ...
END
```

A process must start with the reserved word **PROCESS** followed by its name and its call parameter in brackets. The parameters are a list of data in which the process is going to receive different values. The brackets are obligatory even if the process has no parameters.

After this head, a **PRIVATE** section, declaring data to be used by the process exclusively, may be put optionally.

And finally, the process code, that is a sequence of statements between the reserved words **BEGIN** and **END**, will be specified.

A process normally corresponds with a kind of object of the game, such as a spacecraft, an explosion, a shot, etc. Inside the process' code, a loop (in which all the necessary values to display this object, such as graphic, coordinates, etc., will be established) is normally implemented. Then, with the **FRAME** statement, the order to display the object with the established attributes is given.

```
PROGRAM my_game;
PRIVATE
  id2;
BEGIN
  id2=my_process(160, 100)
  // ...
END
```

```
PROCESS my_process(x, y)
PRIVATE
  n;
BEGIN
  graph=1;
  FROM n=0 TO 99;
    x=x+2;
    y=y+1;
    FRAME;
  END
END
```

As it can be noticed in this example, when a process is called, it returns its identifying code (that, in the example, is stored in the private variable of the main program id2).

If the aim is to implement a process in the style of the functions of other languages that returns a numeric result, then it is necessary to use the **RETURN** (<numeric value>) statement, not using the **FRAME** statement inside the process, as this statement returns to the father process (caller), returning the process' identifying code as a return value.

Parameters of a process

The parameters of a process are basically a list of data in which the process will receive different information every time it is invoked (called or used) from another process.

The processes can receive parameters in the following types of data:

A predefined local datum (such as **x**, **size**, **flags**, ...).
A local datum defined inside the **LOCAL** section.
A global datum defined inside the **GLOBAL** section.
A private datum which must not necessarily be declared inside the **PRIVATE** section.

In all these cases, it is understood that a datum may be referred to a variable, to a specific position of a table or to an element inside a structure.

As an example of the different types of parameters, a program with a process that receives five parameters different from the types respectively indicated in the previous list is now shown.

```
PROGRAM my_game;
GLOBAL
  score=0;
LOCAL
  energy=0;
BEGIN
  my_process(1,2,3,4);
  // ...
END

PROCESS my_process(x, energy, score, n)
BEGIN
  // ...
END
```


It is not necessary to declare the **n** private variable, but it could be declared in the following way (after the head of the process, and before its **BEGIN**):

```
PRIVATE
    n;
```

Note: Receiving a parameter (such as the global variable **score**) in a global data is equivalent to making the assignment (**score=3;**) and then, calling the process.

7.6 List of statements

The types of statements existing in the DIV language are shown in this section.

The statements always appear as a set, from none (which makes no sense) to as many as necessary. All the statements will sequentially be executed (the first one, the second one, the third one...), unless the statements that can control the program's flow (control, loops and break statements) determine some exceptions.

7.6.1 Assignment statement

The assignment statements are used to calculate expressions and **to assign them** to a datum.

```
<reference to a datum> = <expression> ;
```

The datum in which the result of the expression is going to be stored must be indicated, followed by the symbol **=** (symbol of the **assignment**), as well as the numeric or logical expression to evaluate when the statement is executed. After this statement, the symbol **;** (semicolon) must always be put.

In an assignment statement it is only allowed to assign values to objects such as any kind of **variables**, to **a position of a table**, or to **an element of a structure**.

It is not possible to assign a value to a **constant**, to a **function** or to a **process** or, in general, to any **numeric or logical expression**.

Now, a program with several assignments is shown.

```
PROGRAM my_game;
BEGIN
  x = x+1;
  angle = (angle*3)/2-pi/2;
  size = (x+y)/2;
  z = abs(x-y)*3-pow(x, 2);
  // ...
END
```

This is the basic form of the assignments, even if there are other symbols of assignment that, instead of assigning a new value to the referred datum, modify its value.
These are the symbols of **operative assignments**:

+= Adds to the datum the result of the expression
Example: **x=2; x+=2;** is equivalent to **x=4;**

-= Subtracts from the datum the result of the expression
Example: **x=4; x-=2;** is equivalent to **x=2;**

*= Multiplies the datum by the result of the expression
Example: **x=2; x*=3;** is equivalent to **x=6;**

/= Divides the datum by the result of the expression
Example: **x=8; x/=2;** is equivalent to **x=4;**

%= Puts in the datum the remainder of dividing it by the result of the expression
Example: **x=3; x%=2;** is equivalent to **x=1;**

&= Performs an AND (binary and/or logical) between the datum and the result of the expression, assigning it as a new datum's value
Example: **x=5; x&=6;** is equivalent to **x=4;**

|= Performs an OR (binary and/or logical) between the datum and the result of the expression, assigning it as a new datum's value
Example: **x=5; x|=6;** is equivalent to **x=7;**

^= Performs an exclusive OR (XOR binary and/or logical) between the data and the result of the expression, assigning it as a new datum's value
Example: **x=5; x^=3;** is equivalent to **x=3;**

>>= Rotates the datum to the right as many times as indicated by the result of the expression (each rotation to the right is equivalent to dividing the datum by 2)
Example: **x=8; x>>=2;** is equivalent to **x=2;**

<<= Rotates the datum to the left as many times as indicated by the result of the expression (each rotation to the left is equivalent to multiplying the datum by 2)
Example: **x=2; x<<=2;** is equivalent to **x=8;**

Within the category of assignment statements, the **increments** and **decrements** of a datum are also allowed. For instance, if we wanted to add 1 to the local variable **x** we could do it either with the **x=x+1;** or **x+=1;** statements, or with the operator of increment: **x++;** or **++x;**.

7.6.2 IF statement

```
IF ( <condition> )
  <statement> ;
  ...
END
```

(or)

```
IF ( <condition> )
  <statement> ;
  ...
ELSE
  <statement> ;
  ...
END
```

The **IF** statement is used to execute a block of statements optionally, when a **condition** is complied. In the second aforementioned variant, another block of statements will also be executed (inside the **ELSE** section) when the condition is **not** complied.
A program with several **IF** statements is now shown.

```
PROGRAM my_game;
BEGIN

  IF (key(_esc))
    exit("Good by!", 0);
  END

  IF (x>100 AND x<220)
    y=y+4;
  ELSE
    y=y-8;
  END

  IF (size>0)
    size=size-1;
  END

  IF (timer[5]>1000)
    z=1;
  ELSE
    z=-1;
  END

  // ...
END
```

It is possible to nest **IF** statements with no limits. That is to say, more **IF** statements can be put inside the part that is running when the condition is complied (**IF** part) or inside the one that is executed when the condition is not complied (part **ELSE**).

7.6.3 SWITCH statement

```
SWITCH ( <numeric expression> )
  CASE <range of values> :
    <statement> ;
    ...
  END
  ...
END
```

(or)

```
SWITCH ( <numeric expression> )
  CASE <range of values> :
    <statement> ;
    ...
  END
  ...
  DEFAULT :
    <statement> ;
    ...
  END
END
```

A **SWITCH** statement is made up with a series of **CASE** sections and, optionally, a **DEFAULT** section.

```
PROGRAM my_game;
BEGIN
  SWITCH (x)
    CASE 1:
      x=-1;
    END
    CASE 2:
      x=-2;
    END
    CASE 3:
      x=-3;
    END
    CASE 99:
      x=-99;
    END
    DEFAULT:
      x=0;
    END
  END
END
```

When a **SWITCH** statement is executed, the expression is first evaluated and then, if the result is within the range of values included in the first **CASE** section, its statements will be executed and the statement will finish. If the result of the expression is not in the first **CASE**, it will be looked for in the second, third, etc. **CASE**. Finally, if there is a **DEFAULT** section and the result of the expression has not coincided with any of the **CASE** sections, then the statements of the **DEFAULT** section will be executed.

The **SWITCH** statement of this program will change the sign of the **x** variable if it is equal to **1**, **2**, **3** or **99**. Otherwise, the statement will put the variable at **0**.

Range of values of a section CASE

A value, a range of values **minimum .. maximum** (it is important to separate these values by **two dots**, not by three), or a list of values and/or ranges separated by commas may be specified in a **CASE** section. For instance, the previous statement could have been expressed as follows:

```
SWITCH (x)
  CASE 1..3,99:
    x=-x;
  END
  DEFAULT:
    x=0;
  END
END
```

Once one of the **CASE** sections of a **SWITCH** statement has been executed, **no more sections will be executed**, even if they also specify the result of the expression, for instance, in the following statement:

```
SWITCH (2+2)
  CASE 3..5:
    x=x+1;
  END
  CASE 2,4,6:
    y=y-1;
  END
END
```

The **x=x+1;** section will be executed and then, the statement will finish and the **y=y-1;** section won't be executed as, even if the result of the evaluated expression (**4**) is included in it, it is also included in the previous section, (as **4** is within the range **3..5**).

It is not necessary to arrange the **CASE** sections according to their values (smaller to larger, or larger to smaller), but it is indispensable that the **DEFAULT** section (if it exits) is the last section. There can only be one **DEFAULT** section.

It is possible to nest **SWITCH** statements with no limits. That is to say, new **SWITCH** statements (and any other kind of statement) can be put inside a **CASE** section.

7.6.4 WHILE statement

```
WHILE ( <condition> )
  <statement> ;
  ...
END
```

The **WHILE** statement implements a **loop**. That is to say, it is capable of **repeating a group of statements a specific number of times**.

In order to implement this loop, the condition that has to be complied for the group of statements to be executed must be specified in brackets, after the reserved word **WHILE**. All the statements that necessarily have to be repeated will be put after the specification of this condition. Finally, the end of the loop will be marked with the reserved word **END** (it doesn't matter whether more words **END** appear inside the loop when they belong to internal statements of that loop).

When a **WHILE** statement is executed, the specified verification will be carried out. If the result is true, the internal statements will be executed. Otherwise, the program will continue from the **END**, that marks the end of the **WHILE**.

If the internal statements have been executed (what is called to make a loop's **iteration**), the condition will be verified again. If it is true, another **iteration** will be made (the internal statements will be executed again). This process will be repeated until it is verified that the condition of the **WHILE** is false.

If the condition turns to be false directly while a **WHILE** statement is executed, then the internal statements will never be executed.

```
PROGRAM my_game;
BEGIN
  x=0;
  WHILE (x<320)
    x=x+10;
    FRAME;
  END
END
```

In this example, the ***x*** local variable (***x*** coordinate of the process) will be put at zero and then, providing that ***x*** is less than 320, 10 will be added to ***x*** and a **FRAME** will be performed.

A **BREAK** statement inside a **WHILE** loop will immediately finish it, continuing the program from the statement next to that loop.

A **CONTINUE** statement inside a **WHILE** loop will force the program to verify the initial condition immediately and, if it is true, to execute again the internal statements from the beginning (after the **WHILE**). If the condition turns to be false, the **CONTINUE** statement will finish the loop.

The internal statements of a **WHILE** loop can be as many as desired, and of any kind, obviously including new **WHILE** loops.

7.6.5 REPEAT statement

```
REPEAT
  <statement> ;
  ...
UNTIL ( <condition> )
```

The **REPEAT** ... **UNTIL(** ... **)** statement is very similar to the **WHILE** statement and also implements a **loop**.

It must start with the reserved word **REPEAT**, followed by the statements that you want to repeat one or more times, and the end of the statement will be determined by putting the reserved word **UNTIL** followed by the **condition** that has to be complied for **the statement to finish**.

When a **REPEAT** statement is executed, the internal statements (those placed between the **REPEAT** and the **UNTIL**) will be executed first and then, the condition specified in the **UNTIL** will be verified. If it is still false, the internal statements will be executed again. The process will be repeated until the condition of the **UNTIL** turns to be true, continuing then the execution of the program after this statement.

Every time that the internal statements are executed, a loop's **iteration** has been performed. The **REPEAT** ... **UNTIL()** (the **<condition>** is complied) statement will always execute the internal statements at least once, as it always verifies the condition after their execution.

```
PROGRAM my_game;
BEGIN
  x=0;
  REPEAT
    x=x+10;
    FRAME;
  UNTIL (x>320)
END
```

In this example, the ***x*** local variable (***x*** coordinate of the process) will be put at zero and then, 10 will be added to ***x*** and a **FRAME** will be performed **until *x*** becomes a number bigger than 320.

A **BREAK** statement inside a **REPEAT** loop will immediately finish it, continuing the program from the following statement to that loop.

A **CONTINUE** statement inside a **REPEAT** loop will force the program to make the verification of the **UNTIL** immediately and, if it is true, it will execute again the internal statements from the beginning (after the reserved word **REPEAT**). If the condition turns to be true, the **CONTINUE** statement will finish the loop.

The internal statements of a **REPEAT** loop can be as many as desired, and of any kind, obviously including new **REPEAT** loops.

7.6.6 LOOP statement

```
LOOP
  <statement> ;
  ...
END
```

The **LOOP** statement implements an **infinite loop**. That is to say, it **indefinitely repeats a group of statements.**

In order to implement this loop, it is necessary to start with the reserved word **LOOP**, followed by the statements intended to be repeated continuously, putting the reserved word **END** at the end.

When a **LOOP** ... **END** statement is found in a program, all the internal statements of that loop will repeatedly be executed from this position.

In order to finish a **LOOP** loop, it is possible to use the **BREAK** statement which, on being executed inside a loop of this kind, will force the program to continue from the **END**.

The **CONTINUE** statement inside a loop will finish the current **iteration** and will start the following one (the program will go on running after the reserved word **LOOP**.

```
PROGRAM my_game;
BEGIN
  x=0;
  LOOP
    IF (key(_esc))
      BREAK;
    END
    x=x+1;
    FRAME;
  END
END
```

In this example, the ***x*** local variable (***x*** coordinate of the process) will be put at zero and then, 1 will be added to it and a **FRAME** will continuously be performed. If the **ESC** key is pressed, the **BREAK** statement will be executed, finishing the **LOOP** loop.

The internal statements of a **LOOP** loop can be as many as desired, of any kind, obviously including new **LOOP** loops.

7.6.7 FOR statement

```
FOR ( <initialisation> ; <condition> ; <increment> )
  <statement> ;
  ...
END
```

The **FOR** statement (replica of the **C language**) also implements a **loop**. After the reserved word **FOR**, three different parts must be specified in brackets, separated by symbols ; (semicolon). These three parts, that are optional (**they can be omitted**), are the following ones:

Initialisation. An assignment statement is normally codified in this part. This kind of statement **establishes the initial value** of the variable that is going to be used as a **counter of the loop's iterations** (each execution of the inner group of statements is called a loop's iteration). The assignment statement **x=0**, that would put the ***x*** variable at zero at the beginning of the loop (value for the first iteration), is an example.

Condition. A condition is specified in this part. Just before each iteration, this condition will be checked and, if it is true, the group of statements will be executed. If the condition is false, the **FOR** loop will finish, continuing the program after the **END** of the **FOR** loop. An example of condition can be **x<10**, that would allow the inner group of statements to be executed only when the ***x*** variable is a number less than 10.

Increment. The **increment of the variable used as a counter** for each iteration of the loop is indicated in the third part. It is normally expressed with an assignment statement. For instance, the **x=x+1** statement would add 1 to the ***x*** variable after each iteration of the loop.

The group of inner statements of the loop that are going to be repeated sequentially while the condition of continuance (second part) is complied, must appear after the definition of the **FOR** loop with its three parts. After this group of statements, the reserved word **END** will determine the end of the **FOR** loop.

When a **FOR** statement appears in a program, the part of the initialisation will be executed first, checking the condition. If it is true, the inner group of statements first and, the part of the increment then, will be executed, being the condition checked again, etc. If, before any iteration, the condition turns to be false, the **FOR** statement will immediately finish.

A program with a **FOR** loop containing the three parts mentioned in the previous sections is now shown.

```
PROGRAM my_game;
BEGIN
  FOR ( x=0 ; x<10 ; x=x+1 )
    // The inner statements will be put here.
  END
END
```

This loop would first be executed with the ***x*** variable equal to 0, the second one equal to 1, ..., and the last variable equal to 9. The part of the increment would be executed after this iteration, becoming ***x*** equal to 10. Then, on checking the condition of continuance in the loop (***x*** is less than 10), if it is false, the loop will finish.

As it has been mentioned, the three parts in the definition of the loop are optional. If the three were omitted:

```
FOR ( ; ; )
  // ...
END
```

Then, this loop would be equivalent to a **LOOP** ... **END** loop.

Moreover, several parts of initialisation, condition or increment can be put in a **FOR** loop, separated by commas. At first, all the initialisations will be executed. Then, all the conditions of continuance will be checked (if any of them turns to be false, the loop will finish). The inner statements and, after every iteration, all the increments, will finally be checked.

```
PROGRAM my_game;
BEGIN
  FOR ( x=0, y=1000 ; x<y ; x=x+2, y=y+1 )
    // The inner statements will be put here.
  END
END
```

A **BREAK** statement inside a **FOR** loop will immediately finish it, continuing the program from the following statement of this loop.

A **CONTINUE** statement inside a **FOR** loop will force to execute the part of the increment directly and then, to verify the condition of continuance. If it is true, then the inner statements will be executed again from the beginning. If the condition turns to be false, then the **CONTINUE** statement will finish the **FOR** loop.

A **FOR** loop is practically equivalent to a **WHILE** loop, implemented in the following way:

```
PROGRAM my_game;
BEGIN
  x=0;
  WHILE (x<10)
    // The inner statements will be put here.
    x=x+1;
  END
END
```

With the only exception that a **CONTINUE** statement, inside this **WHILE** loop, would not execute the part of the increment, while it would do so inside a **FOR** loop.

If, after the execution of the initialisation, the condition turns to be false directly inside a **FOR** loop, no inner statements will ever be executed.

The inner statements of a **FOR** loop can be as many as desired, of any kind, obviously including new **FOR** loops.

7.6.8 FROM statement

```
FROM <variable> = <numeric value> TO <numeric value> ;
  <statement> ;
  ...
END

(or)

FROM <variable> = <numeric value> TO <numeric value> STEP <numeric value> ;
  <statement> ;
  ...
END
```

The **FROM** statement is the last one implementing a **loop**. For that, a variable of the process itself that can be used as a loop counter is needed.

The reserved word **FROM** must be put before the statements that will comprise the inner group of statements. This word will be followed by the **name of the counter variable**, the symbol of assignment (=), the **initial value** of the variable, the reserved word **TO** and, finally, the **final value** of the variable. The symbol ; (semicolon) must be put after this declaration of the loop **FROM**.

The inner group of statements that is intended to be repeated a specific number of times is put after this head defining the conditions of the loop. Finally, the reserved word **END** will be put.

The first **iteration** will be performed with the **initial value** in the variable used as a counter. After this iteration, **1 will be added** to this variable (if the initial value **is less** that the final value). Otherwise **1 will be subtracted** from it. After having updated the value of the

variable, it is necessary to pass to the following iteration, provided that the value of this variable has not reached (or exceeded) the **final value** of the loop.

The reserved word **STEP** may be put as a second meaning of the **FROM** statement, after the initial and final values of the statement. This word must be followed by a **constant value** indicating the increment of the counter variable after every iteration of the loop, instead of **1** or **-1**, which are the increments that will be performed by default if the **STEP** declaration is omitted.

The following example shows a program with two loops **FROM**: one without **STEP** declaration (with increment or decrement by default) and the other with it.

```
PROGRAM my_game;
BEGIN
  FROM x=9 TO 0;
    // Inner statements ...
  END
  FROM x=0 TO 9 STEP 2;
    // Inner statements ...
  END
END
```

The first loop will be executed 10 times with the ***x*** variable. Its value will range between 9 and 0 in the different iterations. By default, 1 will be subtracted from the variable each time, as the initial value (9) is bigger than the final value (0).

In the second loop, constant 2 is indicated as the increment of the variable. Thus, the loop will be executed 5 times with the ***x*** variable, whose values will be 0, 2, 4, 6 and 8, respectively, in the consecutive iterations. As it can be noticed, no iteration will be performed with ***x*** being equivalent to 9, even if it is the loop's **final value**. By default, if 2 had not been specified as **STEP** of the loop, 1 would have been added to the ***x*** variable after each iteration.

```
PROGRAM my_game;
BEGIN
  FOR ( x=9 ; x>=0 ; x=x-1 )
    // Inner statements ...
  END
  FOR ( x=0 ; x<=9 ; x=x+2 )
    // Inner statements ...
  END
END
```

A loop **FROM** can always be performed with the **FOR** statement, as it is now shown (with two loops equivalent to those of the previous example).

Note: The **initial** and **final** values of a loop **FROM** must be different.

- If the initial value **is less** than the final value, it is not possible to specify a negative value in the **STEP** declaration.
- If the initial value **is bigger** than the final value, it is not possible to specify a positive value in the **STEP** declaration.

A **BREAK** statement inside a loop **FROM** will immediately finish it, continuing the program from the following statement to this loop (after the **END**).

A **CONTINUE** statement inside a **FROM** loop will force the program to increment the variable used as a counter immediately and then, if the final value has not been exceeded, to start the following iteration.

The statements inner to a loop **FROM** may be as many as desired, of any kind, obviously including new loops **FROM**.

7.6.9 BREAK statement

A **BREAK** statement inside a loop will immediately finish it, continuing the program from the following statement to that loop. This statement can only be put inside the following loops: **WHILE, REPEAT, LOOP, FOR** or **FROM**.

A **BREAK** statement will make the program continue its execution after the **END** or the **UNTIL** of the loop closer to the statement.

If there are several nested loops (one inside another), the **BREAK** statement will only exit the innermost loop.

```
PROGRAM my_game;
BEGIN
  LOOP
    REPEAT
      IF (key(_esc)) BREAK; END
      //...
    UNTIL (x==0);
    //...
  END
END
```

In this example, the **BREAK** statement will exit the **REPEAT** ... **UNTIL** (when the **ESC** key is pressed), but not the **LOOP** ... **END**.

Important: The **BREAK** statement is not valid to finish **IF**, **SWITCH** (or the **CASE** sections of this statement), or **CLONE** statements.

7.6.10 CONTINUE statement

A **CONTINUE** statement inside a loop will force the program to finish its current iteration and start the following iteration. We call **iteration** to each execution of the set of statements internal to a loop (the statements between a **LOOP** and its **END**, for instance). This statement can only be put inside one of the following loops:

LOOP ... **END**
A CONTINUE inside this loop will jump to the LOOP.

FROM .. = ..**TO** .. **STEP** .. ; ... **END**
A CONTINUE inside this loop will perform the increment (STEP) and, if the value indicated in the TO has not been passed, the program will continue at the beginning of the loop.

REPEAT ... **UNTIL** (..)
A CONTINUE inside this loop will jump to the UNTIL.

WHILE (..) ... **END**
A CONTINUE inside this loop will jump to the WHILE.

FOR (.. ; .. ; ..) ... **END**
A CONTINUE inside this loop will perform the increment and the comparison. If the latter is true, the program will continue at the beginning of the loop. But if it is false, the program will continue after the END of the FOR.

If there are several nested loops (one inside another), the **CONTINUE** statement will take effect only in the inner loop.

```
PROGRAM my_game;
BEGIN
  FOR (x=0,y=0;x<10;x++)
    IF (x<5) CONTINUE;
END
    y++;
  END
END
```

In this example, after the whole loop has been executed, ***x*** will be equal to 10 and ***y*** will be equal to 5 as, providing that ***x*** is less than 5, the **CONTINUE** statement prevents the ***y++;*** statement from being executed.

Important: The **CONTINUE** statement is not valid inside **IF**, **SWITCH** (or the **CASE** sections of this statement), or **CLONE** statements (as these statements do not implement loops and, therefore, they do not make iterations).

7.6.11 RETURN statement

The **RETURN** statement immediately finishes the current process, as if the **END** of its **BEGIN** was reached.

When this statement is included in the main code, it will finish the current process. But if there are alive processes, they will go on running. For instance, the **exit()** function can be used to finish a program and all its processes.

```
PROGRAM my_game;
BEGIN
  LOOP
    IF (key(_esc))
      RETURN;
    END
    FRAME;
  END
END
```

A **RETURN** inside a process will finish it, killing this process.

In this example, the **RETURN** statement will be executed by pressing the **ESC** key, finishing the program.

Use of RETURN to return a value

It is possible to design processes with a performance similar to the functions of other programming languages, that **receive a series of parameters and return a value.** For instance, a process receiving two numeric values and returning the biggest one.

For that, this statement must be used with the following syntax:

```
RETURN( <expression> )
```

It is also important not to use the **FRAME** statement inside the process, as this statement will immediately return to the calling process. When the compiler finds the **FRAME** statement inside a **PROCESS**, it directly classes it as a process, ruling out its hypothetical use as a function.

The example proposed before is shown next: an implementation of the mathematical function **max** that returns the greater of its two parameters.

```
PROGRAM my_game;
BEGIN
  x=max(2,3)+max(5,4);
END

PROCESS max(a,b)
BEGIN
  IF (a>b)
    RETURN(a);
  ELSE
    RETURN(b);
  END
END
```

After the execution of this program, the ***x*** variable of the main process will be equal to **8** (3+5).

Important: By default, if the **RETURN** statement is used without the expression in brackets or the **FRAME** statement is used in a process, its return value will be its **Identifying code** of the process.

7.6.12 FRAME statement

The **FRAME** statement is an essential part of the language. A program's mechanics described below in general terms:

- The main program starts its execution. This process may create more processes (objects of the game) at any point. All the processes may be finished at any moment, and they may create or eliminate other processes.
- The games will always be displayed frame by frame. In each frame, the system will execute all the processes existing at that moment, one by one, until each one executes the **FRAME** statement, which will indicate that it is ready for the next display (frame).
- In the preparation of each frame, all the processes will be executed in the established priority order (the **priority** local variable of the processes determines this order).

```
PROGRAM my_game;
BEGIN
  my_process();
  my_process();
  LOOP
    IF (key(_esc))
      my_second_process();
    END
    FRAME;
  END
END

PROCESS my_process()
BEGIN
  LOOP
    FRAME;
  END
END

PROCESS my_second_process()
BEGIN
  LOOP
  END
END
```

Therefore, this statement is similar to an order for the processes to be displayed.

If a process starts its execution and it neither finishes nor executes this statement, then the program will become blocked, as there is a process that is never ready for the next display. Therefore, the system won't be capable of showing the following frame.

In this program, the main process (**my_game** type process) creates other two processes (**my_process** type). From that moment, the three processes will continuously be executed, each one to their **FRAME** statement. But if the **ESC** key is pressed, then the main process will create a new process (**my_second_process** type) that will remain in a **LOOP** loop indefinitely, without executing any **FRAME**. Consequently, the program will be interrupted (the system will report such a situation after few seconds; see the **max_process_time** global variable).

Basically, all the processes that correspond with objects of a game construct a loop inside which, every frame establishes all its display values (**x, y, graph, size, angle, ...**) executing then the **FRAME** statement.

Synchronisation of processes
It is possible to use this statement with the following syntax:

FRAME(<percentage> **)**

By putting in brackets a whole percentage, from 0 to 100 or bigger, after the reserved word **FRAME**.

This figure will indicate the percentage of the following frame, completed by the process. That is to say, the absence of this percentage is equivalent to putting **FRAME**(100) (100% of the work previous to the following display has been completed by the process).

For instance, if a process executes the **FRAME(25)** statement in a loop, it will need to execute it **4 times** before it is ready for the next display (as 4*25% is the 100%).

On the other hand, if a process executes the **FRAME(400)** statement inside its loop, after its first execution, it will have completed 400% the display. Therefore, even after the display, a completed 300% of display will still be missing. For that, in the preparation of the following **3** frames the system won't execute this process, as it is ready for the display. Then, this process would be executed just once every 4 frames (unlike the example of the previous paragraph, in which it was executed 4 times every game's frame).

The processes won't reach the next display unless they give **100%, at least**. For instance, if a process always executes **FRAME(80)** statements, it will execute them twice before the first display, so it will have completed 160% (2*80%) the display. Therefore, it will have precompleted 60% (160%-100%) for the next display. For that reason, in the second display it will only require a **FRAME(80)** statement to be displayed, as this 80%, plus the remaining 60% of the first display, will be equal to a 140% completed. Therefore, it will immediately be displayed ,and a 40% will be even left to prepare the next frame.

Note: A **FRAME(0)** statement completing a **0%** of the next display only makes sense in the two following cases:

- It can be a way to force the system to execute in this point the rest of the processes having the same priority as the current one and, after them, the system will execute the latter again.
- It can also be a way to initialise functions such as **get_id()** or **collision()**, as they return some specific values for every frame. If the aim is to obtain values again, it is possible to execute a **FRAME(0)** statement that will be interpreted as a new frame by these functions.

7.6.13 CLONE statement

```
CLONE
  <statement> ;
  ...
END
```

This statement creates a new process identical to the current one, with the exception that the statement between the reserved words **CLONE** and **END** will only be executed in the new process, but not in the current one.

For instance, if any process of the program, with specific coordinates (***x, y***) and with a specific graphic (**graph**), executes the following statement:

```
CLONE
  x=x+100;
END
```

A new process will be created, identical to the former one, with the same graphic and the same values in all its variables, with the exception of the ***x*** coordinate that, in the new process, will be placed **100** pixels farther to the right.

This statement is used to create replicas of a process, dividing it into two processes (almost) similar.

```
PROGRAM my_game;
BEGIN
  // ...
  x=0;
  y=0;
  CLONE
    x=x+10;
  END
  CLONE
    y=y+10;
  END
  // ...
END
```

In this example, the **2 CLONE** statements will create **3** copies of the main process (and not 2, as it could have been expected).

On executing the first **CLONE** statement, a new process will be created. Thus, there will be **2** processes: one in (x=0,y=0) and the other in (x=10,y=0). These two processes will execute the second **CLONE** statement. The first one (the original one) will create a new process in (x=0, y=10), and the second one will create the new process in (x=10, y=10).

To create only **2** copies of the original process, the program could have been constructed, for instance, in the following way:

```
PROGRAM my_game;
BEGIN
  // ...
  x=0;
  y=0;
  CLONE
    x=x+10;
    CLONE
      y=y+10;
    END
  END
  // ...
END
```

The original process (x=0,y=0) will create one in (x=10, y=0) and the latter will create another one in (x=10, y=10). Therefore, only two copies of the original will be created.

Much care must be taken when it comes to using the **CLONE** statement sequentially or inside a **loop**, as it is necessary to take into account that the first '**clones**' may also create new '**clones**'.

This statement can be used without putting statements between the words **CLONE** and **END**. But, intending to have two identical processes with the same coordinates, the same graphic and executing the same code, seems to make little sense, at least at first.

7.6.14 DEBUG statement

The **DEBUG** statement will call the interactive **debugger** when it is executed. It is normally used to debug programs, that is to say, to find possible errors of the programs. On some occasions, it is normally put in the following points:

- Where you want to verify that a part of the program has done what was expected. After the execution of that part, **DEBUG** will call the debugger, from which it is possible to check all the active processes and the value of all their variables.

- When you are not very sure whether something can happen in a program, you can put this statement in that point to report you whether what we are expecting actually happens.

This statement is only used temporarily, until the error that is looked for is found. From that moment, the statement won't be necessary. Thus, it can be removed from the program since it has no additional effect.

```
PROGRAM my_game;
BEGIN
  // ...
  IF (x<0)
    DEBUG;
  END
  // ...
END
```

In this example, it is verified that, in a specific point of the program, the ***x*** coordinate of the process is not a negative number (less than zero). If this happens, the debugger will be called to find out why it has happened.

When this statement is executed, a dialog box appears, offering us the following options:

- To disable the **DEBUG** statement, preventing it from being activated in this execution of the program.

- To stop the program and enter the debugger, to be able to examine all the processes and their variables.

- Or to finish the execution of the program immediately, returning to its edition in the windows' graphic environment.

Moreover, if the escape key **ESC** is pressed in that box, the **DEBUG** statement will simply be ignored, and the program will continue to be executed as usual.

Note: When a program is executed from the windows' graphic environment, the debugger can be called at any moment by pressing the **F12** key.

On invoking the debugger in this way, the program will always be interrupted just before starting the processing of a new frame. All the processes to be executed will appear before the next display.

Chapter 8
Program Creation

CHAPTER 8: Creating Programs. Advanced Concepts

The last concepts essential to create programs in DIV Games Studio are described in this last chapter. For further information, consult the appendixes and the help hypertext inside the graphic environment.

8.1 Types of processes

The blocks of the programs starting with the reserved word **PROCESS** determine the performance of a specific process type. Then, when the program is executed, any number of processes of this type will be able to exist at a specific moment. Each of these processes will have a different **identifying code**, but all of them are of the same type.

```
PROGRAM my_game;
BEGIN
  // ...
END

PROCESS spacecraft()
BEGIN
  // ...
END

PROCESS enemy()
BEGIN
  // ...
END

PROCESS shot()
BEGIN
  // ...
END
```

In this example, four types of processes are defined: **my_game** (the type of the program's initial process), **spacecraft**, **enemy** and **shot**.

The number of processes of each of these types existing in the game depends on the number of calls made to these processes.

All the **spacecraft** type processes will always execute the statements defined in the **PROCESS spacecraft()** of the program.

A "process type" is a numeric code referred to the name of the PROCESS that determines how the process works during the game. This numeric code can be obtained with: **TYPE <name_of_the_process>.**

TYPE is an operator defined in the language that, applied to a process name, returns this numeric code.

For instance, **TYPE spacecraft** will be equivalent to a specific numeric constant and **TYPE enemy** will be equivalent to another one.

All the processes have a local variable containing this numeric code, which is:

reserved.process_type

Why is the process type for?
The process type is used for several things, as mentioned below:

- For the **get_id()** function that receives a process type (for instance, **get_id(TYPE enemy)**) as a parameter and returns the identifying codes of the processes of this type existing in the game at that moment.
- For the **collision()**function is similar to the previous one, with the proviso that it returns the identifying codes of the processes with which it is colliding (that is to say, the graphics of both processes are partially superposed).
- For the **signal()** function, that may send a signal to all the existing processes of a specific type.
- Or to verify, from a process' identifying code, what kind of process it is (type spacecraft, type shot, etc.).

Note: The operator **TYPE** can only be used preceding a process name of the program or the word **mouse**, to detect collisions with the mouse pointer (**collision(TYPE mouse)**).

8.2 Identifying codes of processes

A process is an object independent of the program, that executes its own code and that can have its own coordinates, graphics, etc. Processes of a program can be, for instance, a shot, spacecraft or enemy. When something similar to what is below is input inside a program:

```
PROCESS shot( ... );
BEGIN
  // statements ...
END
```

The statements that are going to execute the "**shot** type" processes (that is to say, the code ruling their performance), are specified.

As it can be noticed, more than one **shot** type process may exist in a program. Then, how can they be distinguished? Simply by their **identifying code**.

Every time a new process is created in a game, an identifying code is assigned to this process. This code is going to be the exclusive reference of the process until it disappears.

Two different processes will never have the same identifying code at the same time. However, the code that belonged to a process that has already disappeared can be assigned to a new process (something similar to what happens in relation to an i.d.).

The identifying codes are always whole, positive, odd numbers, like 471, 1937 or 10823.

All the processes have their own identifying code in **ID**, that is something similar to a process' local variable local, with the proviso that it can not be modified.

Moreover, the processes have the identifying code of the process that created them (that called them) in **father**. They have the identifying code of the last process they created (the last one they called) in **son**. And so on.

What are the identifying codes for?

Normally, all the processes need the identifying code of the other processes in order to interact with them (to see where they are, to modify them, ...).

For instance, it is not possible to subtract energy from the "**enemy** type" process, as many or none of this type of process may exist. It is necessary to have the specific identifying code of the **enemy** process from which you want to subtract energy.

A process accesses all its own variables simply by their names, such as **x**, **size** or **graph**. Thus, if the identifier of a process is known (in **son**, **father** or any variable defined by the user, such as **id2**), then it is possible to access the variables of that process, as (**son.x**, **father.size** or **id2.graph**). That is to say, the syntax to access local variables of another process is as follows:

```
<identifying_code>.<name_variable>
```

These variables can normally be used to consult them or modify them.

It is not at all possible to access PRIVATE variables of another process at any rate. In order to access a private variable of another process, it is necessary to change its declaration to the **LOCAL** section to transform it into a local variable. Then, any process will be able to access that variable just having the identifying code of the process, as all the processes will have that variable.

The identifiers have more utilities other than the access to alien local variables, such as the **signal()** function, that can send specific signals to a process if its identifying code is known (for instance, to eliminate the process).

There are also other functions, such as **collision()**, used to detect collisions with other processes. When this function detects a collision, it returns the identifying code of the process with which it is colliding. Once this code is known, it is possible to access the variables of the process and send them signals.

The **get_id()** function operates in a similar way to **collision()**, obtaining the identifying code of a process. But in this case, no collision with it is required.

8.3 Ways to obtain the identifying code of a process

All the processes have their own **identifying code** in **ID** (reserved word in the language that is equivalent to the identifying code of the process).

When a process is created (is called), it returns its own identifying code as return value, unless it has finished with a **RETURN**(<expression>). That is to say, a process will always return its identifying code when it finishes (when its **END** is reached), when it executes the **FRAME** or the **RETURN** statements without expression in brackets.

In the following example, a process (**my_process** type) is created from the main program, and its identifier is stored in the **id2** variable.

```
PROGRAM my_game;
PRIVATE id2;
BEGIN
  id2=my_process();
  // ...
END
PROCESS my_process()
BEGIN
  // ...
END
```

All the processes have the following local variables predefined with identifiers of other processes:

father - father, identifier of the process that created it (the one that made the call).

son - son, identifier of the last process created by it (last called process).

bigbro - Elder brother, identifier of the last process created by the father before creating it.

smallbro - Younger brother, identifier of the following process created by the father after having created it.

These variables can be equal to **0** if they have not been defined (for instance, **son** will be equal to 0 until a process is not created or if this process has disappeared).

The processes' identifying codes allow us to access their local variables (<identifier>.<variable>) and, as **father**, **son**, etc. are also local variables, it is possible to make combinations such as **son.bigbro** to access the identifier of the penultimate process created (as **son** is the last one; therefore, its elder brother will be the penultimate one).

Besides creation or direct relationship, there are other ways to obtain identifying codes of processes, as indicated below:

- The **get_id()** function to obtain the identifiers of the processes of a specific type (spacecraft, shot, etc.) existing at a specific moment in the game.
- The **collision()** function to obtain the identifiers of the processes with which it is colliding.

When a specific process needs to access from many others, as it is an important process such as, for instance, the protagonist spacecraft of a game, then it can be more useful to assign its identifier to a **GLOBAL** variable of the program (that can be accessed by any process at any point). Thus, any process will be able to interact with it, as it will have its identifier.

```
PROGRAM my_game;
GLOBAL
  id_spacecraft;
BEGIN

id_spacecraft=spacecraft();
  // ...
END
PROCESS spacecraft()
BEGIN
  // ...
END
PROCESS enemy()
BEGIN
  // ...
  id_spacecraft.z=0;
  // ...
END
```

In this example, at a specific point the **enemy** type processes access the **z** variable of the **spacecraft** created by the main program, using for that purpose its identifier, that is included in the **id_spacecraft** global variable.

8.4 Call to a process

<process name> (<list of parameters>)

In order to call a process, put the **name** of the process, followed by a list including as many **expressions** separated by commas as **parameters** of the process, in brackets. The brackets are obligatory, even if the process has no **call parameters**.

A call to a process will always return a value that depends on which one of the following actions is performed first by the called process.

- If the **FRAME** statement is executed, then the process will return its **identifying code**.
- If the process executes the **RETURN(<expression>)** statement, then the former will return the result of this expression.
- If the process finishes, either because the **END** of its **BEGIN** is reached or because a **RETURN** statement is executed with no expression, the process will return the **identifying code** that had. But, as the process has finished (**killed**), it is necessary to take into account that this **identifying code** can now be used by any other process created from now on.

The return value can be ignored, assigned to a variable or used inside an expression.

```
PROGRAM my_game;
PRIVATE
 id2;
BEGIN
  my_process(0,0);

id2=my_process(320,200);
  // ...
END
PROCESS my_process(x,y)
BEGIN
  LOOP
    FRAME;
  END
END
```

In this example, the main process **my_game** makes two calls to the process **my_process**, which receives two parameters in its ***x*** and ***y*** local variables.

As the process executes the **FRAME** statement, it will return its **identifying code**.

It can be noticed how the value returned in the first call to the process is ignored (it is not used at all), and how, in the second call, the **identifying code** of **my_process(320, 200)** is assigned to the private variable of the main process **id2**.

When a call to a process is made, the execution of the current process is momentarily stopped, and the code of the called process is executed, until it is returned through one of the three mentioned cases (until it finishes or executes a **FRAME** or **RETURN** statement).

If the process has finished with a **FRAME** statement, it will be displayed in the following frame according to the values established in its local variables (**x**, **y**, **graph**, ...) and, in the preparation of the following frame, this process will go on running from the **FRAME** statement.

8.5 Hierarchies of processes

When a program starts to run there is only one process: the initial process, which starts the execution of the main code's statements. But, from this moment, this process can create new processes that, at the same time can create other processes, destroy them, etc.

In order to clarify the events appearing through a program, we use a simile, treating the processes as if they were alive beings that are born and killed (when they are created or destroyed). For that reason, the following terms are established:

Father, name given to the process that has created another one (mother would have been a more appropriate name).

Son, process created by another one.

Brothers, processes created by the same father.

Orphan, process whose father has dead (as it has been either eliminated or finished).

This vocabulary may be spread as far as your imagination desires **grandfathers**, **grandsons**, **uncles**, etc.

All the processes have direct access to the identifying codes of the processes with which they have direct relationship.

Occasionally, reference is made to actions performed by "**the system**". This process, called **div_main**, controls the rest. Therefore, it is in charge of creating the initial process at the beginning of the execution, of setting the speed execution, the debugger, etc. All the processes that are orphaned become sons of this process.

The **div_main identifier** can be obtained with **get_id(0).** It can be used to send a tree signal to all the processes, but this process won't be displayed on screen , even if its **x**, **y**, **graph**, etc. variables are defined.

8.6 States of a process

Processes are the different elements of a program (objects of the game). They may experience different states on creating, destroying or receiving specific signals with the **signal()** function.

Alive or awaken process
A process is **alive** when it is running (when it is interpreting the statements located between its **BEGIN** and its **END**).

Dead process
A process is **dead** when it finishes (either because its **END** is reached in the execution, a **RETURN** is executed or because it receives a signal **s_kill** or **s_kill_tree**).

Asleep process
A process may receive the signal **s_sleep** (or **s_sleep_tree**), then becoming asleep. In this state, this process will appear to be dead. But it is not as, at any moment, it may receive a signal **s_wakeup** and return to the alive or awake states. It is also possible to kill an asleep process.

Frozen process
The signal **s_freeze** (or **s_freeze_tree**) freezes a process. In the frozen state, the process, that is still visible, remain blocked. It may be detected by the rest of the processes (for instance, in collisions), but it is not executed (it stops interpreting its code statements). It will remain in this state until it receives another signal that changes its state or that kills it.

A frozen process may be controlled (moved) by another process, directly manipulating its variables.

Notes:
When a signal is sent to a process, aiming at changing its state, this signal will have no effect before its following display (**FRAME**) is reached if the process is running. If the process is not running, then the signal will have an immediate effect.

No signal must be sent to nonexistent processes (to an identifying code that does not correspond with any process).

This signal will be ignored when the aim is to put a process in the state in which it is already.

8.7 Use of angles in the language

In the language, all the angles are specified in degree thousandths. For instance:

0 are **0 degrees** (to the right)
90000 are **90 degrees** (up)
-45000 are **-45 degrees** (down right diagonal)

If **360 degrees (360000)** are added to or subtracted from any angle, an equivalent angle is obtained. For instance, the angles -90000 and 270000 are equivalent (the angles of **-90 degrees** and **270 degrees** go both downwards)

The constant **PI** predefined as **180000**, **3.1415 radians** or, what is the same, **180 degrees**, can be used as reference. For instance, **PI/2** will be equal to 90 degrees (**90000**).

Some of the functions dealing with angles are the following ones: **get_angle()**, **get_distx()**, **get_disty()**, **fget_angle()**, **near_angle()**, **advance()**.

All the processes have a predefined local variable called **angle** which, by default, will be equal to **0**. If its value is modified, the display's angle of the graphic of the process will be changed (the graphic will rotate in the indicated degrees, from the original graphic).

8.8 About the conditions

In general, any **expression** is valid as a condition. In the language, all the **ODD** expressions are interpreted **as** true and all the **EVEN** expressions are interpreted **as false**.

```
PROGRAM my_game;
BEGIN
  IF (20*2+1)
    x=x+1;
  END
END
```

In this example, the **x=x+1;** statement will always be executed, as the expression **20*2+1** equals **41**, an **odd** number.

All the available operators are valid inside a condition. It is even possible to make assignments inside a condition (the assignments are operations that return the assigned value as a result).

All the **identifying codes** of processes are **odd** numbers, that is to say, all of them are **true**. Therefore, it is possible to implement conditions as the following one (supposing that **id2** has been declared as a variable, and **shot** is a process type of the program).

```
WHILE (id2=get_id(TYPE shot))
  id2.size=id2.size-1;
END
```

In the condition **id2=get_id(TYPE shot)** the result of the **get_id()** function is assigned to the **id2** variable. If that function has returned an identifying code, it will be an **odd** number and the condition will be evaluated as **true** (if **get_id()** does not find (more) identifiers of "type **shot**" processes, then it will return **0** (which is an **even** number). The condition will be interpreted as **false**, and the **WHILE** statement will finish.

The previous statements would decrement the **size** variable of all the type **shot** processes contained in the program.

8.9 Evaluation of an expression

It is important to know the way in which the expressions are evaluated in order to know where it may be necessary to put brackets indicating the way in which the expression is intended to be evaluated.

In the language, an expression can contain operators of different levels of priority.

In the evaluation of an expression, the operators of **priority 1** (if they exist), will always be processed first, and then, those of **priority 2**, **priority 3** and so on.

Priority 1
: **( )** Brackets, beginning and end of a sub-expression.

Priority 2
: **.** Period, operator of access to local data and structures.

Priority 3
NOT Binary and logical negation (!).
OFFSET Offset (&).
POINTER Addressing operator (*, ^, [ ]).
- Sign negation.
++ Increment operator.
-- Decrement operator.

Priority 4
* Multiplication.
/ Division.
MOD Module (%).

Priority 5
+ Addition.
- Subtraction.

Priority 6
<< Rotation to the right.
>> Rotation to the left.

Priority 7
AND Binary and logical (&, &&).
OR Binary and logical (|, ||).
XOR Exclusive Or (^, ^^).

Priority 8
== Comparison.
<> Different (!=).
> Bigger.
>= Bigger or equal (=>).
< Less.
<= Less or equal (=<).

Priority 9
= Assignment.
+= Addition-assignment.
-= Subtraction-assignment.
*= Multiplication-assignment.
/= Division-assignment.
%= Module-assignment.
&= AND-assignment.
|= OR-assignment.
^= XOR-assignment.
>>= Rotation to the right-assignment.
<<= Rotation to the left-assignment.

The operators of **priority 3** are known as **unary** operators. They do not link two operands (unlike the **binary** operators such as, for instance, a multiplication), but they just affect the value of an operator.

Inside the **unary** operators, those closest to the operand will be executed first. For instance, in the expression:

NOT -x

The operand ***x*** has two **unary** operators, the negation of sign **-** and the logical and/or binary **NOT**. Among them, the negation of sign will be executed first, as it is closer to the operand.

From **priority 4**, all the operators are **binary** and they will be executed according to their level of priority. Therefore, when in an expression there is more than one operator of the same level (for instance, a multiplication and a division, both of priority 4), they will be processed from left to right. That is to say, in the following expression:

8/2*2

The division will be executed first and then, the multiplication (it is the natural way to evaluate the expressions mathematically).

The only exception are the operators of **priority 9** (**assignment** operators), that will be evaluated from right to left (instead of from left to right). That is to say, in the expression:

x=y=0

y=0 will be processed first (**y** will be put at **0**) and then, **x=y** (**x** will also be put at **0**, as **y** will now be equal to **0**).

As it can be noticed, the assignments work like an operator. After the assignment, they return the value they have assigned as a result of the operation.

APPENDIX A

A

Appendix A: Summary Of The Syntax Of A Program

This first appendix shows a brief summary of the syntax of a program in the DIV language which could help to clarify the general structure of a program.

The syntax is shown **informally**, highlighting the **symbols** and **reserved words** which are essential to the language. The **dots** indicate that the previous declaration can be repeated any number of times. Many of the parts of the programs shown here are **optional**.

An extended and detailed syntax with examples can be found in the help hypertext of DIV Games Studio (using option **help...** in the main menu).

<Program>

PROGRAM <name> ;
CONST
  <name> = <numerical value> ;
  ...
GLOBAL
  <declaration of datum> ;
  ...
LOCAL
  <declaration of datum> ;
  ...
PRIVATE
  <declaration of datum> ;
  ...
BEGIN
  <statement> ;
  ...
END

<Declaration of process>
...

<Declaration of datum>

There are three kinds of declaration of datum:

<Declaration of a variable>
<Declaration of a table>
<Declaration of a structure>

<Declaration of a variable>

<name> = <numerical value>

<Declaration of a table>

<name> [<numerical value>] = <list of numerical values >

<Declaration of a structure>

STRUCT <name> [<numerical value>]
  <declaration of datum> ;
  ...
END = <list of numerical values >

<Declaration of process >

PROCESS <name> (<list of parameters>)
PRIVATE
  <declaration of datum> ;
  ...
BEGIN
  <statement> ;
  ...
END

<Statement>

We find the following kinds of statements:

<Statement of assignment>
<Statement IF>
<Statement SWITCH>
<Statement LOOP>
<Statement FROM>
<Statement REPEAT>
<Statement WHILE>
<Statement FOR>
<Statement BREAK>
<Statement CONTINUE>
<Statement RETURN>
<Statement FRAME>
<Statement CLONE>
<Statement DEBUG>
<Call to a function >
<Call to a process >

<Statement of assignment>

<datum> = <numerical expression>

<Statement IF>

IF (<condition>)
  <statement> ;
  ...
END

(or otherwise)

```
IF ( <condition> )
   <statement> ;
   ...
ELSE
   <statement> ;
   ...
END
```

<Statement SWITCH>

```
SWITCH ( <numerical expression> )
   CASE <range of values> :
      <statement> ;
      ...
   END
   ...
END
```

(or otherwise)

```
SWITCH ( <numerical expression> )
   CASE <range of values> :
      <statement> ;
      ...
   END
   ...
   DEFAULT :
      <statement> ;
      ...
   END
END
```

<Statement WHILE>

```
WHILE ( <condition> )
   <statement> ;
   ...
END
```

<Statement REPEAT>

```
REPEAT
   <statement> ;
   ...
UNTIL ( <condition> )
```

<Statement LOOP>

```
LOOP
   <statement> ;
   ...
END
```


<Statement FROM>

FROM <variable> = <numerical value> **TO** <numerical value> ;
  <statement> ;
  ...
END

(or otherwise)

FROM <variable> = <numerical value> **TO** <numerical value> **STEP** <numerical value> ;
  <statement> ;
  ...
END

<Statement FOR>

FOR (<initialisation> ; <condition> ; <increment>)
  <statement> ;
  ...
END

<Statement BREAK>

BREAK

<Statement CONTINUE>

CONTINUE

<Statement RETURN>

RETURN

(or otherwise)

RETURN(<numerical expression>)

<Statement FRAME>

FRAME

(or otherwise)

FRAME(<numerical expression>)

<Statement CLONE>

CLONE
  <statement> ;
  ...
END

<Statement DEBUG>

DEBUG

<Call to a process >

<name> (<list of parameters>)

<Call to a function >

<name> (<list of parameters>)

APPENDIX B

B

Appendix B: Functions Of The Language

The functions available in DIV Games Studio are described in this appendix. It is also possible to consult them, in an interactive way, in the help hypertext where, moreover, a **sample program** for every one of these functions is shown and explained.

abs(<expression> **)**

Returns:
The **absolute value** of the expression.

Description:
Calculates the absolute value of the expression passed as a parameter. That is to say, if the result of the expression is negative, it will change its sign; if it is positive, it will not change it.

---

advance(<distance> **)**

Description:
Advances the **process** in its angle (the one shown by the **angle** local variable) as many points as the **expression** (distance) passed as a parameter shows.

The distance can also be a negative number. In that case, the graphic of the process will move forward (its coordinates **x** and **y**) in the direction opposite to its angle.

Notes:
Keep in mind that the angle is specified in thousandths of a degree.

This function is always equivalent to the two following statements:

```
x+=get_distx(angle,<distance>);
y+=get_disty(angle,<distance>);
```

That is to say, this function only modifies the coordinates of the process. It is possible to use the two previous statements when the aim is to advance the process in an angle different from the one shown by its **angle** variable. It will be useful when the aim is to make the graphic of the process advance in a direction without rotating.

For instance, to make the process advance 8 points in a direction (that could be obtained in a private variable like **angle2**) but rotated to another direction, (the one showed in **angle**), the following statements would be used:

```
x+=get_distx(angle2,8);
y+=get_disty(angle2,8);
```

---

change_sound(<channel>, <volume>, <frequency>)

Description:
To use this function it is essential to have a 16 Bit sound card installed.

The use of this function only makes sense after using the **sound()** function, used to emit sounds.

Change_sound() modifies a sound that is being played through one of the **channels**, adjusting its **volume** and its **frequency** once again.

The **channel** is the **channel code** returned by the **sound()** function when it is called. The 16 channels may even sound at the same time, with the same sound or with different sounds. Therefore, every time that a sound is emitted it will possibly be emitted through a different channel.

Every **channel** has its volume and frequency levels established at all times.

The **volume** is a value between **0** (minimum volume) and **512** (maximum volume) that determines the power with which the sound of that channel will be heard.

The **frequency** is a value that affects the speed at which the sound is heard through the channel. That is to say, it controls the bass and treble of the sound. This value ranges between **0** (bass) and **512** (treble).

---

clear_screen()

Description:
Clears the screen background, that is to say, the graphics that would be drawn on it with the **put()**, **xput()**, **put_pixel()** and **put_screen()** functions.

---

collision(<process type>)

Returns:
The **identifying code** of a process or **0**.

Description:
This is the language's function used **to detect collisions** between graphics.

It verifies if the current process (executed by this function) collides with one of the **type** shown as a parameter. That is to say, it verifies if the graphics of both processes are touching, at least partially.

In case of collision, it will return the **identifying code** of the process with which the current process is colliding. Otherwise, the function will always return **0**.

If the current process collides with several processes of the specified **type**, the **collision()** function will return the rest of the identifiers in the successive calls made to it.

To obtain in this way all the **identifying codes** of the processes that collide with the current one, the **FRAME** statement must not be used between two consecutive calls to the **collision()** statement. When a process executes a **FRAME** statement, this function will return all the **identifying codes** of colliding processes from the first one.

Something similar happens if a call to the function is executed, specifying a different type of process. If, after that, collisions with the previous type are detected, this function will also return all the codes from the first one.

If the aim is to obtain the **identifying codes** of the processes of a specific **type**, even if there is no collision with them, it is necessary to call the **get_id()** function.

But if the aim is to check the proximity between two processes whose graphics are not necessarily colliding, then the **get_dist()** function will have to be used.

Notes:
When the mouse pointer is being displayed in the program (assigning the code of the corresponding graphic in the **mouse structure**), it is possible to see if the pointer collides with the current process using this function, for instance, in the following way:

```
IF (collision(TYPE mouse))
  // The process collides with the mouse pointer.
END
```

On detecting the collision with the mouse pointer, it will not be done with the whole graphic used as a pointer, but only with its main **control point** (number 0), usually called mouse **hotspot**.

Important:

This function is useful to detect collisions between graphics of the screen or of a scroll window.

It is not possible to use this function to detect collisions with processes that have no graphic (a valid code assigned to its **graph** variable) or between graphics of a mode-7 window (with its **ctype** variable assigned to the value **c_m7**).

Therefore, it is **essential** that both, the current process and the process of the specified type, have a graphic defined.

---

convert_palette(<file>, <graphic>, <OFFSET new_palette>**)**

Description:
Changes the colour map of the **<graphic>** of the indicated **<file>**.

The **<OFFSET new_palette>** is the address, inside the computer's memory, of a 256 value table where the new order of the graphic's colours will be indicated.

If the table with the new palette is like the following one:

```
new_palette[255]=0, 1, 2, 3, 4, ... , 254, 255;
```

then the graphic would not be transformed. If, for instance, in position 3 of the previous table (new_palette[3]) we put a 16 (instead of a 3), by calling this function with the **OFFSET** of this table, colour 3 would be replaced by colour 16 in the graphic.

The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the code 0 file).

If a process wanted to replace the colours of its graphic, it should be first necessary to create a table with the new colours order and then, call the function with the parameters:

convert_palette(file, graph,<OFFSET new_palette>)

Using the **file** and **graph** local variables of the process itself as parameters of the **convert_palette()** function.

---

define_region(<number of region>, <x>, <y>, <width>, <height>)

Description:
Defines a new display region inside the screen (something similar to a window). The regions are rectangular zones of the screen inside which some specific processes, scroll or mode-7 windows will be displayed.

The **region number** must range between 1 and 31. It is possible to define up to 31 different screen regions that, later can be assigned to different processes (establishing their **region** local variable at the new number) as their display window. They can also be used as a framework for a scroll or mode-7 window, indicating it in the corresponding parameter of the **start_scroll()** or **start_mode7()** functions.

Region number 0 must not be redefined, as it will always be the entire screen, a window at the (0, 0) coordinates, of the same width and height of the screen. This is the region in which all the processes will be displayed by default, as its **region** local variable is always equivalent to 0 by default.

---

delete_text(<text identifier>)

Description:
Definitively deletes a text from the screen if the **text identifier** is specified as a parameter. This identifier is a numeric code returned by the **write()** and **write_int()** functions when they are asked to write a text.

If **all_text** is specified as a parameter, all the texts will be deleted from the screen.

---

end_fli()

Description:
Finishes a **FLI/FLC** animation displayed on the screen and frees up the computer's memory that it was occupying.

Notes:
The **FLI/FLC** animations start with the **start_fli()** function.

Only one animation can be loaded in the computer's memory.

It is not necessary to wait for the animation to finish in order to unload it from the memory.

---

exit(<message>, <return code>)

Description:
The game finishes by killing all the processes immediately and going back to the operating system (or to the DIV environment) with a **message** and a **numeric code** (the one indicated in the expression of the second parameter).

The message is a text in inverted commas that will appear as a farewell message for the player when the game is over.

The **return code** is valid to use programs external to DIV Games Studio (such as BAT batch processing files), to determine the action that must be performed after the game has been executed.

When the **exit()** function is used, it is not necessary to have previously unloaded any resource, such as files, maps, sounds, etc., since the system automatically finishes all the resources.

Note:
All the programs will stop running if the **ALT+X** key combination is pressed at any moment. This operation is equivalent to forcing the execution of the **exit()** function, but without displaying any message and with the return code 0.

---

fade(<% red>,<% green>,<% blue>,<speed>)

Description:
Starts a fading of the game's palette colours until the display percentages (from 0% to 200%) of the **red**, **green** and **blue** components, shown as parameters, are obtained.

The last parameter indicates the speed at which the colours' fading will be performed. It is normally defined by a number, from 1 (very slowly) to 10 (very quickly).

If a number bigger than 64 or equal to it is indicated as speed, the fading will be performed instantaneously.

The fading will gradually be performed in the successive displays of the game (in the following frames).

If the three components are set at 0, a fading to black is carried out. If they are set at 200, a fading to white will be carried out. Finally, the original colours of the game palette will be recovered if the components are set at 100.

A value less than 100 in a component will fade its colour, whereas a value higher than 100 will saturate the colour.

Keep in mind that the fading is not carried out by executing the **fade()** function, but it is carried out in the following **FRAME** statements. While a fading is carried out, the predefined **fading** global variable will be equivalent to **true** (an odd number that, in this case, will be 1) and when the fading is over (finally obtaining the specified colour display values), this variable will become equivalent to **false** (an even number, number 0).

---

fade_off()

Description:
Carries out a fading of the screen's colours to black. The game stops until the screen becomes completely black. Carrying out a fading to black is called **to fade the screen off.**

The **fade_on()** function is used to fade the screen on again (undo the fading to black).

Note:
The **fade()** function can fulfill the same function without stopping the program or at different speeds. At the same time, it can produce other more advanced palette's effects.

---

fade_on()

Description:
Carries out a screen colours' fading to its natural situation. In the successive displays of the game (on reaching the **FRAME** statement) the colours will gradually be appearing until they are perfectly seen. This action is called **to fade the screen on.**

The **fade_off()** function is used to fade the screen off (to carry out a fading to black).

Notes:
The **fade()** function can fulfill the same function at different speeds. At the same time, it can produce other more advanced palette's effects.

All the games automatically carry out a **fade_on()** at the beginning of the execution.

---

fget_angle(<x0>, <y0>, <x1>, <y1>)

Returns:
The angle between two points.

Description:
Returns the angle existing from **point 0** (x0, y0) to **point 1** (x1, y1).

Keep in mind that the **angle** is specified in degree thousandths. The function always returns a value between **-180000** and **180000** (an angle between -180 and 180 degrees).

Any valid numeric **expression** can be defined as coordinates of both points (x0, y0, x1, y1).

Notes:
The **get_angle()** function is used to obtain the **angle from a process to another one**, instead of between two points.

The **fget_dist()** function is used to obtain the **distance between two points**, instead of the angle.

---

fget_dist(<x0>, <y0>, <x1>, <y1>)

Returns:
The distance between two points.

Description:
Returns the distance existing from **point 0** (x0,y0) to **point 1** (x1,y1).

Any valid numeric **expression** can be specified as coordinates of both points (x0, y0, x1, y1).

Notes:
The **get_dist()** function is used to obtain the **distance of a process to another one**, instead of between two points.

The **fget_angle()** function is used to obtain the **angle between two points**, instead of the distance.

This function may be used to detect collisions between processes due to their proximity, even if the **collision()** function is normally used to this purpose. The last function detects when two processes have their graphics overlapped.

For instance, with the processes displayed inside a mode-7 window (see **start_mode7()**), the **collision()** function can not be used, being necessary to obtain the distance between the processes (normally with **get_dist()**) to verify if they collide with each other (if their distance is less than a distance already determined).

---

frame_fli()

Returns:
True if the animation goes on and **false** if it has finished.

Description:
Shows the following frame of a **FLI/FLC** animation started with the **start_fli()** function. This function returns **0** if the animation has already finished.

During the program's execution, it will be possible to execute but one **FLI/FLC** animation at the same time. That is to say, it will not be possible to execute two animations simultaneously.

The animation frame will only be displayed in the following frame of the game (when the **FRAME** statement is reached). Therefore, if a loop is made and inside it the **frame_fli()** function (but not the **FRAME** statement) is called, the animation will not be displayed on the screen.

---

get_angle(<identifying code>)

Returns:
The angle towards another process.

Description:
Returns the angle from the current process (the one that called this function) to the process whose **identifying code** is passed to it as a parameter.

Keep in mind that the **angle** is specified in degree thousandths. The function always returns a value between **-180000** and **180000** (an angle between -180 and 180 degrees).

Notes:
The **fget_angle()** function is used to obtain the **angle between two points**, instead of between two processes. If the **identifying code** of the process is stored, for instance, in a variable called **id2**, then the call to the function:

get_angle(id2)

would be equivalent to:

fget_angle(x,y,id2.x,id2.y)

Obtaining the angle from the (x, y) coordinates of the current process, to the (x, y) coordinates of the process whose **identifying code** is **id2**.

The **get_dist()** function is used to obtain the **distance to another process**, instead of the angle.

---

get_dist(<identifying code>**)**

Returns:
The distance to another process.

Description:
It returns the distance between the current process (the one that called this function) to the process whose **identifying code** is passed to it as a parameter.

If the process has defined its local variable **resolution**, it is important that the process to which the aim is to obtain the distance has it defined at the same value. That is to say, if both processes use the coordinates in hundreds instead of units (with **resolution**=100), the distance between both will also be obtained in hundreds. But if the value of that variable differs in both processes, the result of the **get_dist()** function will make no sense.

Notes:
The **fget_dist()** function is used to obtain the **distance between two points**, instead of between two processes. If the **identifying code** of the process is stored, for instance, in a variable called **id2**, then the call to the function:

get_dist(id2)

Would be equivalent to:

fget_dist(x,y,id2.x,id2.y)

Obtaining the distance from the (x, y) coordinates of the current process, to the (x, y) coordinates of the process whose **identifying code** is **id2**.

The **get_angle()** function is used to obtain the **angle to another process**, instead of the distance.

This function may be used to detect collisions between processes due to their proximity, even if the **collision()** function is normally used to this purpose. The last function detects when two processes have their graphics overlapped.

For instance, with the processes displayed inside a mode-7 window (see **start_mode7()**) the **collision()** function can not be used, being necessary to obtain the distance between the processes to verify if they collide with each other (if their distance is less than the distance already determined).

get_distx(<angle>, <distance>)

Returns:
The horizontal offset of the vector (angle, distance).

Description:
Returns the horizontal distance (in the axis of the **x** coordinate) from the **angle** and **distance** (over this angle) passed as parameters. That is to say, it returns the distance covered in horizontal by the vector made by the **angle** and **length** (distance or vector module) indicated. Keep in mind that the **angle** is specified in degree thousandths and any valid numeric **expression** may specify the distance.

The function used to calculate the vertical distance, instead of the horizontal one, is **get_disty()**.

Notes:

If the aim is to advance the coordinates of the process a **distance** in a specific angle, the following statements may be used:

```
x+=get_distx(<angle>,<distance>);
y+=get_disty(<angle>,<distance>);
```

If the angle in which the aim is to move the process is the one contained in its **angle** local variable, then this operation could be performed with the **advance()** function in the following way:

```
advance(<distance>);
```

The **get_distx()** function is equivalent to calculating the **cosine** of the **angle** and multiplying it by the **distance**.

---

get_disty(<angle>, <distance>)

Returns:
The vertical offset of the vector (angle, distance).

Description:
Returns the vertical distance (axis of the *y* coordinate) from the **angle** and **distance** (over this angle) passed as parameters. That is to say, it returns the distance covered in vertical by the vector made by the **angle** and **length** (distance) indicated.

Keep in mind that the **angle** is specified in degree thousandths. Any valid numeric **expression** can specify the distance.

The function used to calculate the horizontal distance, instead of the vertical one, is **get_distx()**.

Notes:
If the aim is to advance the coordinates of the process a **distance** in a specific angle, the following statements may be used:

```
x+=get_distx(<angle>,<distance>);
y+=get_disty(<angle>,<distance>);
```

If the angle in which you want to move the process is the one contained in its **angle** local variable, then this operation could be performed with the **advance()** function in the following way:

```
advance(<distance>);
```

The **get_disty()** function is equivalent to calculating the **sine** of the **angle** and multiplying it by the **distance**, changing this result of sign, because the screen **Y** axis advances downwards (unlike the sine function).

---

get_id(<process type>)

Returns:
Either the **identifying code** of a process or **0**.

Description:
Verifies if there are active processes of the specified **type**. If there are, then this function will return the **identifying code** of one of them. If there are not, it will return 0.

If there are several processes of the specified **type**, the **get_id()** function will return the rest of the identifiers in the successive calls made to it.

Once all the **identifying codes** have been returned, the function will return 0, until a **FRAME** statement is executed again. From this moment, this function will return all the identifying codes of the processes of the specified **type** again. To obtain all the **identifying codes** of the processes in this way, the **FRAME** statement will not be used between two consecutive calls to the **get_id()** statement. In case that a **FRAME** statement is executed, this function will return all the **identifying codes** of processes again, from the first one.

Something similar happens if a call to the function is executed, specifying a different **type of process**. If, after this, identifiers of the previous type of process are asked again, this function will also return them from the first one.

Note:
The **collision()** function is used to obtain the **identifying codes** of processes of a specific type that, moreover, collide with the current process.

get_joy_button(<number of button>)

Returns:
True (1) if the button is pressed, **False** (0) if it is not.

Description:
This function requires the joystick number button (from 0 to 3) as a parameter, and returns **true** (an odd numerical value) if it is pressed at that moment.

If the button is not pressed, the function returns **false** (an even numeric value).

Some joysticks have only 2 buttons. In this case, they will be number 0 and 1 buttons. In computers with two connected joysticks, the second joystick will have buttons number 2 and 3.

Note:
There are other ways to use the joystick. The easiest one is to use the **joy structure**, since in it there are four records that continuously show the state of the joystick buttons.

---

get_joy_position(<number of axis>)

Returns:
The position of the joystick axis.

Description:
This function returns the coordinate in which the axis specified (with a number from **0** to **3**) of the analog joystick is placed.

0 axis - Main **X** axis.
1 axis - Main **Y** axis.
2 axis - Secondary **X** axis.
3 axis - Secondary **Y** axis.

The joystick coordinate may vary, depending on the type of joystick and on the computer in which it is executed. Anyhow, it is a number that usually ranges from **4** to **200**, approximately.

The **main** and **secondary** axes may be integrated in a single joystick, on some occasions (flight commands with **pedals**, etc.). In computers having two connected joysticks, joystick 1 will be the main axis and joystick 2 will be the secondary one.

Note:
There are other ways to use the joystick. The easiest one is to use the **joy** structure, when an analog reading of the joystick (its coordinates) is not required. That is to say, when it is enough to know if the joystick is at the center, to the right, down, etc.

---

get_pixel(<x>, <y>)

Returns:
The colour of the pixel (0..255).

Description:
Returns the colour that the **background screen** pixel placed in the coordinates indicated as parameters has.

The returned number is the colour's order inside the colours palette active in the program. It ranges from 0 to 255, as the palettes have 256 colours.

The pixel is exclusively taken from the background screen picture, without having into account the graphics of the processes, texts, scroll regions, etc. That is to say, only the colours put by the **put()**, **xput()**, **put_pixel()** and **put_screen()** functions will be read.

---

get_point(<file>, <graphic>, <number>, <OFFSET x>, <OFFSET y>)

Returns:
The position of the control point (in the variables whose **offset** is indicated as the last two parameters).

Description:
This function returns the **control point**, whose **number** is indicated as third parameter, to the position where it was placed in a **graphic** (of the indicated **file**).

A **control point** is a point that can be defined in the graphic editor (painting tool), in the option designed for this function.

The function needs the **address** (that is obtained with the **offset** operator) in the computer's memory of **two variables** in which it will return the ***x*** and ***y*** position of the control point.

The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the file with the 0 code).

Note:
This function returns the exact coordinates in which that control point was placed inside the graphic, without considering how this graphic is now (scaled, rotated, etc.). The **get_real_point()** function must be used to obtain the position of a control point in a scaled, rotated... graphic and referred to the screen coordinates (and not to those of the original graphic).

That is to say, this last function returns the position where a control point is at a specific moment, and **get_point()** returns the place where this point was originally located.

get_real_point(<number>, <OFFSET x>, <OFFSET y>)

Returns:
The current coordinates of the control point (in the variables whose **offset** is indicated as the last two parameters).

Description:
This function returns where, at that moment, a **control point** of the current process' graphic is in the system of coordinates used by the process itself (see **ctype** local variable), evaluating the original location of the point, the current coordinates of the process, their size, angle, etc.

A **control point** is a point that can be defined in the graphic editor (painting tool), in the option designed for this function.

The function needs the **address** (that is obtained with the **offset** operator) in the computer's memory **of two variables** in which it will return the ***x*** and ***y*** position of the control point.

The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the file with the 0 code).

Notes:
This function is normally used to have some important points of a graphic located. For instance, if we have defined a process whose graphic is a man with a gun that can be scaled, rotated or that can perform different animations, we could define a control point in the barrel gun's point-to know at any time from where the bullets must leave when shooting.

If the original graphic was inside a scroll region (see **start_scroll()**) then the returned coordinates will also refer to that scroll region.

The **get_point()** function returns the position where a **control point** was originally placed in the graphic, unlike the **get_real_point()** function, that returns its current position.

---

graphic_info(<file>, <graphic>, <information>)

Returns:
The required information about the graphic.

Description:
Returns the required information about a **graphic** of a **file**.

Information:
g_wide - The function will return the **original width** of this graphic if g_wide is put as third parameter.

g_height - The function will return the **original height** of the graphic.

g_x_center - The function will return the **x coordinate** of the graphic's center.

g_y_center - The function will return the **y coordinate** of the graphic's center.

The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the file with the code 0).

---

is_playing_cd()

Returns:
True (1) if the CD is playing, or **false** (0) if it is not.

Description:
This function is used to determine whether the CD is playing a song.

It returns **True** (an odd number) if the CD is playing a song. Otherwise, it returns **False** (an even number).

Its most widespread use is to play a song indefinitely.

Note:
The volume of cd-audio reproduction may be controlled with the **setup** structure and the **set_volume()** function.

---

key(<keyboard code>)

Returns:
True (1) if the key is pressed and **false** (0) if it is not.

Description:
Returns **true** (an odd number) if the key, indicated as parameter, is pressed at that moment. Otherwise, it returns **false** (an even number).

The input parameter will normally be the name of the key with the _ (underlining) symbol ahead. For instance, to read the [A] key, the function must be called **key(_a)**.

Access the **keyboards codes** in **appendix C.6** to see the whole list of the keyboard codes that may be used as parameter of the **key()** function.

Notes:
There are three predefined global variables that can also be used to control the keyboard, and they are the following ones:

scan_code - Code of the latest key that has been pressed. This code is a numeric value that directly corresponds with the constants of **keyboard codes**, used as parameters of the **key()** function.

ascii - ASCII code of the latest pressed key.

shift_status - variable that shows a number depending on the special or lock keys (shift, alt, control, ...) pressed at that moment.

---

let_me_alone()

Description:
Sends a **s_kill** signal to all the processes, except the one executed by this function. Therefore, all the processes, except the current one, will be eliminated.

This function is normally used from the main process, when a game is over, to delete all the processes (shots, enemies...) that remain active, and to recover the control of the program.

A call to **let_me_alone()** could always be replaced by a series of calls to the **signal()** function with the **s_kill** signal. For that, it would be necessary to know either the **types of processes** intended to be deleted or their **identifying codes**.

Notes:
To check the active processes of a program at a specific moment, you must access the debugger by pressing the **F12** key.

The **exit()** function is also used to immediately finish a program, returning to the system.

---

load(<archive name>, <OFFSET datum>**)**

Description:
Loads a data block from an archive on the disk to the program's memory.

For that, the function requires the **archive name** and the offset, inside the computer's memory, of the variable, table or structure stored on the disk (the datum offset may be obtained with the **OFFSET** operator).

The offset of the same datum specified when the archive was stored with the **save()** function must be specified.

The archive names may be given by specifying a path, **which must be the same as that used with the save() function** to store the archive. Nevertheless, **it is not necessary** to specify a path. It is important that the archive intended to be loaded has been previously created, as an error will occur if the aim is to load a nonexistent archive (even if this can be ignored, continuing the program's execution).

---

load_fnt(<archive name>)

Returns:
The loaded **font code**.

Description:
Loads an archive with a new characters font (*.FNT) of the disk (a "font" with a new set of graphic characters).

The function returns the **font code** that can be used by the **write()** and **write_int()** functions to write a text.

The archives' path can be specified with the font. Nevertheless, it won't be necessary if the archive with the letter font has been generated in the directory by default (\FNT).

The archive with the new font has to be created with the game's colour palette to be displayed correctly. Otherwise, the colours will appear changed.

---

load_fpg(<archive name>)

Returns:
Returns the loaded **file code**.

Description:
Loads an archive with a (*.FPG) file. A file means a graphics library (or collection).

An FPG archive with a graphics library may contain from no graphic to 999 graphics. Every graphic included in the library will have a numeric code, the **graphic code**, a number that ranges from 1 to 999, and that is used to identify the graphic inside the file.

It is possible to load as many graphics' files as necessary, providing there is available memory (this function has to be called several times to load several files).

The function returns the **file code**, that can be used by many functions that require a graphic. For that, it is necessary to indicate to them the **file code** in which the graphic is and the **graphic code** inside the file.

The archives' path can be specified with the graphics file. Nevertheless, it won't be necessary if the file is in the directory by default (\FPG).

Notes:
The **unload_fpg()** function allows us to free up the computer's memory occupied by the graphics' file when it is not going to be used any longer. For that, it also requires the **file code**, in order to know which file we want to unload from the memory.

It is not necessary to unload the file from the memory before finishing the program, since the system will do it automatically.

---

load_map(<archive name>)

Returns:
The loaded **graphic code**.

Description:
Loads a **MAP archive** with a graphic in the computer's memory . The function requires the archive name as a parameter, in inverted commas.

The **graphic code** is returned as return value, which is a numeric value that must be specified to use the graphic, in the **graph** variable or, in general, in all the functions requiring a **graphic code** among their parameters.

It is possible to load as many graphics as necessary. Every time one is loaded, the function will return the corresponding code (the first graphic loaded will have the code **1000**, the following one the code **1001**, etc.)

It is possible to specify the path to the archive with the graphics' file. Nevertheless, if the file is in the directory by default (\MAP), it won't be necessary.

Important:
When the **file code** to which that graphic belongs is required inside a function, the code **0** (which is the code of the first **file FPG** that is loaded in the program) must be indicated.

When different graphics have been loaded, keep in mind that if they have different palettes, every one of them must previously be activated with the **load_pal()** function, indicating the name of the file (MAP) as a parameter, before using the graphic.

Graphics created with different palettes can not simultaneously be used.

Notes:
The **unload_map()** function allows us to free up the computer's memory used by the graphic when it is not going to be used for a specific time. For that purpose, it also requires the **graphic code** to know which is the graphic to be unloaded from the memory.

It is not necessary to unload the graphic from the memory before finishing the program, as the system will do it automatically.

To load several graphics all at once in a program, they must be included inside a graphics file (FPG) and loaded with the **load_fpg()** function.

---

load_pal(<archive name>)

Description:
Loads a colour palette of the disk (from a PAL, FPG, MAP or FNT archive) defining the **256 colours** displayed on the screen.

From that moment, the game will be seen with the colours set indicated by that palette.

If, at the moment of loading the palette, the program already had a different one assigned, then a fading of the screen colours to black will be carried out. Then, the new colour palette will gradually appear in the following frames of the game.

The archive path may be specified with the palette. Nevertheless, it will not be necessary if the file is, by default, in the directory (that, depending on the type of archive, will be: \PAL, \FPG, \MAP or \FNT).

The program will automatically read the palette of the first of these types of archives loaded in the program, even if the **load_pal()** function is not used. Then, this function will be used when the program uses several different palettes to change from one to another.

Note:
A palette can not be unloaded from the computer's memory, since it does not occupy any space in the memory.

---

load_pcm(<archive name>, <cyclic>)

Returns:
The loaded **sound code**.

Description:
Loads a sound effect from a PCM archive of the disk. The archive name must be replaced by the sound effect as a first parameter. As a second parameter, the <cyclic> must be replaced by **1** if the sound must indefinitely be repeated, or **0** if it must be played only once (when it is required with the **sound()** function).

The function returns the **sound code** that must be used by the **sound()** function to play that sound through a channel.

The archive path may be specified with the sound. Nevertheless, it will not be necessary if the sound is in the directory by default (\PCM).

Notes:
The **unload_pcm()** function allows us to free up the computer's memory occupied by the sound when it is not going to be used any longer. For that, it also requires the **sound code** in order to know which sound we want to unload from the memory.

It **is not necessary to unload the sound from the memory** before finishing the program, since the system will do it automatically.

---

map_block_copy (<file>, <destination graphic>, <destination x>, <destination y>, <origin graphic>, <x>, <y>, <width>, <height>)

Description:
The **map_block_copy()** function allows us to transfer a rectangular block from a graphic to another one.

The graphic from which the rectangular region is taken is called **origin graphic** and the **destination graphic** is the one in which this block will be copied. That is to say, this function allows us to copy a part of a graphic (origin) to another one (destination). The parameters are the following ones, in order:

<file> - Both graphics must come from the same graphics file. The **file code** must be specified as first parameter (see **load_fpg()**). The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the file with the code 0).

<destination graphic> - **code of the graphic** in which the block is going to be put.

<x destination>, <y destination> - (*x, y*) coordinates at which the aim is to put the block inside the destination graphic.

<origin graphic> - **code of the graphic** from which the block is going to be taken.

<x>, <y> - **starting** coordinates of the block inside the origin graphic.

<width>, <height> - **dimensions** of the block that is going to be transferred.

This function will modify the indicated graphic, but only its copy that has been loaded in the computer's memory. The original graphic, that is stored in the **FPG** or **MAP** archives of the disk, **will remain unchangeable.** For that reason, if at a specific moment of the game the aim is to recover the original state of the graphic, it is necessary to unload it from the memory (with the **unload_fpg()** or **unload_map()** functions) and then, load it again.

Note:
When a graphic is put inside another one that is being used as a scroll region's background, it will not automatically appear on screen unless the **refresh_scroll()** function is used.

---

map_get_pixel(<file>, <graphic>, <x>, <y>)

Returns:
The colour of the pixel (0..255).

Description:
Allows us to obtain the colour of a graphic's specific pixel, as a return value of the function. For that purpose, it requires the <file code> in which the graphic is stored, the <graphic code> inside the file and the (*x, y*) coordinates of the graphic's pixel whose colour intended to be obtained.

The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the file with the code 0).

Notes:
This function is normally used to detect zones inside the graphics. This technique is called **hardness maps** and it allows us to use two different graphics, one with the picture and the second with the zones to detect, painted in different colours.

For instance, in a game with spacecraft, the zones that take energy away from the spacecraft when passing over it with a specific colour (for instance, colour 32) could be

painted in this **hardness map**. Then, the colour would be obtained from the **hardness map** over which the spacecraft is and, if it is 32, energy would be taken away from it.

That is to say, there would be two different pictures: one of them visible, with colours (the background picture over which the spacecraft moves in the game, and the other, the **hardness map** that would only be used to obtain colours from it with the **map_get_pixel()** function, identifying so the zone of the original picture over which the spacecraft is.

---

map_put(<file>, <destination graphic>, <origin graphic>, <x>, <y>)

Description:
Puts a graphic inside another one. The graphic that is going to be copied is called **origin graphic** and the **destination graphic** is that inside which the origin will be copied. That is to say, this function allows us to copy a graphic (origin) inside another one (destination).

Both graphics must be in the same file. The parameters are the following ones, in order:

<file> - **file code** with the graphics library that contains both graphics. The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the file with the code 0).

<destination graphic> - **code of the graphic** inside which the other one is going to be put.

<origin graphic> - **code of the graphic** that is going to be copied in the destination.

<x>, <y> - **coordinates inside the destination graphic** where the aim is to put the origin graphic. The center (or **control point** number 0) of the origin graphic will be located in these coordinates.

This function will modify the indicated graphic, but only its copy that has been loaded in the computer's memory. The original graphic, that is stored in the **FPG** or **MAP** archives of the disk, will **remain unchangeable**. For that reason, if at a specific moment of the game the aim is to recover the original state of the graphic, it is necessary to unload it from the memory (with the **unload_fpg()** or **unload_map()** functions) and then, load it again.

Notes:
The **map_xput()** function is a version a little more complex than the **map_put()** function, but with much more utilities. Thus, the latter allows us, moreover, to put rotated, scaled, mirror and transparent graphics.

The **map_block_copy()** function must be used to put a part of a graphic (instead of the full graphic) inside another one.

When the **map_put()** function (or any other similar) is used to modify a graphic that is being used as background of a **scroll** window, it is possible that the graphic you have put does not immediately appear on screen. To solve this problem, you must use the **refresh_scroll()** function.

---

map_put_pixel(<file>, <graphic>, <x>, <y>, <colour>)

Description:
Allows us to modify the colour of a specific pixel of a graphic. For that, the <file code> where the graphic is stored, the <graphic's code> inside the file and the (*x*, *y*) coordinates of the pixel whose <colour> is intended to set are required.

The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the file with the code 0).

This function will modify the indicated graphic, but only its copy that has been loaded in the computer's memory. The original graphic, that is stored in the **FPG** or **MAP** archives of the disk, **will remain unchangeable**. For that reason, if at a specific moment of the game the aim is to recover the original state of the graphic, it will be necessary to unload it from the memory (with the **unload_fpg()** or **unload_map()** functions) and then, load it again.

Notes:
The **map_put()** or **map_xput()** functions may be used to put a full graphic inside another one (and not only at one pixel). The **map_block_copy()** function may be used to put just a part of a graphic inside another one.

When the **map_put_pixel()** function is used to put a pixel in a graphic that is being used as background of a **scroll** window, it is possible that this pixel does not immediately appear on screen. To solve this problem, you must use the **refresh_scroll()** function.

---

map_xput(<file>, <destination graphic>, <origin graphic>, <x>, <y>, <angle>, <size>, <flags>)

Description:
Extended version of the **map_put()** function.

Puts a graphic inside another one. The graphic that is going to be copied is called **origin graphic** and the **destination graphic** is that inside which the origin will be copied. That is to say, this function allows us to copy a graphic (origin) inside another one (destination).

Both graphics must be in the same file. The parameters are the following ones, in order:

<file> - file code with the graphics library that contains both graphics. The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the file with the code 0).

<destination graphic> - code of the graphic inside which the other one is going to be put.

<origin graphic> - **code of the graphic** that is going to be copied in the destination.

<x>, <y> - coordinates inside the destination graphic where the aim is to put the origin graphic. The origin graphic is going to be copied at these coordinates, from its upper left corner.

<angle> - angle (in degree thousandths) in which the origin graphic is going to be copied; the normal angle is **0**.

<size> - size (in percentages) in which the original graphic is going to be copied; the normal size is **100**.

<flags> - Indicates the mirrors and transparencies with which the original graphic will be copied in the destination; the values are the following ones:

0-Normal graphic
1-Horizontal mirror
2-Vertical mirror
3-Horizontal and vertical (180°) mirror
4-Transparent graphic
5-Horizontal transparent and mirror
6-Vertical transparent and mirror
7-Horizontal and vertical transparent, mirror

This function will modify the indicated graphic, but only its copy that has been loaded in the computer's memory. The original graphic, that is stored in the **FPG** or **MAP** archives of the disk, **will remain unchangeable**. For that reason, if at a specific moment of the game the aim is to recover the original state of the graphic, it will be necessary to unload it from the memory (with the **unload_fpg()** or **unload_map()** functions) and then, load it again.

Notes:
The **map_xput()** function is a little more complex than the **map_put()** function, which is easier to use when it is not required to put rotated, scaled, mirror and transparent graphics.

The **map_block_copy()** function must be used to put a part of a graphic (instead of the full graphic) inside another one.

When the **map_put()** function (or any other similar) is used to modify a graphic that is being used as background of a **scroll** window, it is possible that the graphic you have put does not immediately appear on screen. To solve this problem, you must use the **refresh_scroll()** function.

---

move_scroll(<scroll number>)

Description:
Forces to scroll automatically and immediately. This function is rather advanced and, for that reason, it could be difficult to understand its purpose.

As a parameter, the function requires the <scroll number> from 0 to 9 that was indicated in the **start_scroll()** function as first parameter when the scroll started.

This function is used when a scroll region is automatically controlled, as the **camera** field of the **scroll structure** corresponding to the identifier of a process has been defined.

The purpose is to force the (**x0**, **y0**, **x1** and **y1**) values of that structure to be updated. If this function is not used, these values won't be updated until the following game's frame. That is

to say, when a scroll is automatically controlled and another process needs to know the value of the coordinates of that scroll before the next frame (normally to be located in a position in keeping with the background movement), do as follows:

1 - The scroll starts with **start_scroll()**.

2 - The process that will be used as camera is created and its **identifying code** is put in the **camera** field of the **scroll structure**.

3 - A very high priority must be set for this process, for it to run before the rest of the processes (putting in its **priority** local variable a positive whole value like, for instance, 100).

4 - The **move_scroll()** function will be called just before the **FRAME** statement of the process' loop used as camera.

Thus, you will guarantee the previous execution of this process and, just at the end, the updating of the values (**x0**, **y0**, **x1** and **y1**) of the **scroll structure**, so the rest of the processes may use these variables already updated.

The most widespread use of this function the aim is to have more than two backgrounds in a scroll window. For that, a series of processes simulating a third or fourth plane are created. The position of their coordinates will depend on the exact position of the scroll in every frame.

---

move_text(<text identifier>, <x>, <y>**)**

Description:
Moves a text towards other screen coordinates. The **text identifier** and the (**x**, **y**) screen coordinates towards which the text must be moved are specified as parameters. The identifier of the text is a numeric code returned by the **write()** and **write_int()** functions when they are required to write a text.

The **centering code** specified in the **write()** or **write_int()** functions will remain when this function is used.

The specified coordinates always deal with the screen and may be in or out of it. It is necessary to use the **text_z** global variable to modify the **z coordinate of the texts** (the depth plane in which they appear).

Note:
To delete a text definitively, the **text identifier** is also required, and the **delete_text()** function must be used for that.

---

near_angle(<angle>, <final angle>, <increment>)

Returns:
A new angle closer to the final angle.

Description:
Brings an angle closer to another one at the given increment. The function returns the new angle.

It is used when the aim is that an angle (<angle>) gradually varies until it becomes another angle (<final angle>). For that, the function needs the original **angle**, the **final angle** and the angular **increment** that is going to be added to or subtracted from the original angle.

Keep in mind that all the angles are specified in degree thousandths. The angular **increment** is but a small angle (such as one degree (1000) or five (5000).

---

out_region(<identifying code>, <region number>)

Returns:
True if the process is out of the region or **False** if it is not.

Description:
This function determines whether a process is out of a screen region. For that, the function requires the **identifying code** of the process and a number of region.

The screen regions can be defined with the **define_region()** function and they are simply rectangular zones of screen.

Region number **0** can not be defined, as it will always be equivalent to the entire screen. Therefore, if **0** is specified as a second parameter, this function determines whether a process is out of the screen (if it is not seen).

In case that the process' graphic is out of the specified region, the function returns **True** (an odd number). Otherwise, if the graphic is seen in that region, even partially, the function returns **False** (any even number).

The process whose **identifying code** is indicated must have its graphic correctly defined (normally in its **graph** variable). Otherwise, the system will notice an error, since it is not possible to calculate the dimensions of a graphic if the process lacks it.

---

play_cd(<track number>, <mode>)

Description:
Starts playing a cd-audio track. The track number (from 1 to the number of songs contained in the cd) must be indicated. The way to do it is as follows:

Mode:

0 - To play the song and then stop.
1 - To play this song and then the following ones.

Notes:
To have a song indefinitely playing, a loop must be implemented, using the **is_playing_cd()** function to determine when the song is over.

The cd-audio reproduction volume can be controlled with the **setup** structure and the **set_volume()** function.

---

pow(<expression>, <expression>)

Returns:
The first expression **raised** to the second one.

Description:
Calculates the result when the first **expression** is raised to the second one.

For instance, **pow(3, 2)** will return 9, which is 3 squared, that is to say, 3^2, or **3*3**.

Note:
Take into account that in the language **it is only possible to use integers within the (min_int ... max_int) range**. Therefore, when the result of the function exceeds this range, **incorrect results will be shown**. In this case, the system won't notice any error, so much care must be taken.

---

put(<file>, <graphic>, <x>, <y>)

Description:
Puts a graphic in the screen background. The function requires the **file code** in which the graphic is stored, the **graphic code** inside the same file and the (*x*, *y*) coordinates at which the graphic is intended to be put.

The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the file with the code 0).

If the center graphic was not specified (setting its **control point** number 0 from the painting tool), the coordinates will be referred to the position on the screen in which the graphic **center** will be located.

The graphics displayed on the screen background like this will be within the game's display **under all the processes, scroll regions, texts, etc.**

If the aim is to have a graphic over some others, it must be **created as a new process** and its **z** variable must be established, indicating the priority of its display.

The **clear_screen()** function must be used to clear the screen background.

Notes:
If the graphic that is intended to be put is merely a background screen, it is easier to use the **put_screen()** function, since it does not require the screen coordinates, because it will automatically center the graphic on the screen.

The **xput()** function is a little more complex than the **put()** function, but has more features since, at the same time, it allows us to put rotated, scaled, mirror and transparent graphics.

To put a graphic inside another one (instead of in the screen background), the **map_put()** or **map_xput()** functions must be used.

---

put_pixel(<x>, <y>, <colour>)

Description:
Establishes the **colour** of the pixel located in the background screen's (*x*, *y*) coordinates. That is to say, it sets a pixel of the indicated colour in the indicated coordinates.

The pixels put with this function in the background screen will be displayed in the game **below all the processes, scroll regions, texts, etc.**

If the aim is to see a pixel over other graphics, you must create a new process, assigning the picture of a point (in its **graph** variable) as a graphic and fixing its **z** variable with the priority of its printing.

To clear the background screen, the **clear_screen()** function must be used.

Notes:
To read the colour of a specific background screen colour, the **get_pixel()** function must be used, returning a number between 0 and 255 corresponding to the order of the colour inside the palette.

The **put()** function must be used to set a graphic on the screen, instead of a simple pixel.

It is also possible to set the colour of a pixel in a specific graphic, instead of in the background screen, by using the **map_put_pixel()** function.

---

put_screen(<file>, <graphic>)

Description:
Establishes the background screen. The function requires the **file code** in which the graphic is, and the own **code of the graphic** intended to be displayed in the background screen inside the file.

The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the file with the code 0).

The function does not require any coordinate as a parameter since, if the graphic size (in pixels) is different from that of the screen, the former will simply be displayed centered in the latter.

The **clear_screen()** function must be used to clear the screen background.

Note:
If the aim is to display a graphic on a specific part of the screen or a graphic **that is not centered**, the **put()** may be used. Moreover, the **xput()** function allows us to display rotated, scaled, mirror and/or transparent graphics in any screen region.

---

rand(<minimum value>, <maximum value>)

Returns:
A random numeric value.

Description:
Returns a random number (chosen at random) between the **minimum value** and the **maximum value**, both included.

This function is normally used to set all the parameters intended to be varied in a game when it is restarted. For instance, the coordinates of an enemy may be initialised with random numbers, so it may appear in a different position in every game.

This function has another use. Thus, if we want that an action does not always occur, but that it has a certain probability to occur, we normally use a statement of the following type:

```
IF (rand(0,100)<25)
   // Action ...
END
```

In this case, the **action** will take place, on average, 25 per cent of the times the IF statement would be executed. The reason for that is that, on obtaining a random number between 0 and 100, this number would be less than 25 in a fourth of times, approximately.

Note:
By default, the values returned by the **rand()** function will completely be different in every execution of the program. If we want to have always the same series of numbers, we may use the **rand_seed()** function, specifying a number behind which the series of numbers returned by the **rand()** function will always be predetermined.

---

rand_seed(<numeric value>)

Description:

This function sets a seed for the generator of random numbers (the numbers generated by he **rand()** function).

The seed can be any integer within the range (**min_int ... max_int**). If the seed is set, all the numbers generated by the **rand()** function will be the same in every execution of the program. That is to say, after having been established an origin seed, the **rand()** function will return a series of numbers predetermined for this seed.

---

refresh_scroll(<scroll number>)

Description:
This function is used when a graphic that is being used as a **background of a scroll region** has been modified with the **map_put()**, **map_xput()**, **map_block_copy()** or **map_put_pixel()** functions, in order to update it.

The parameter required by the function is the <scroll number> that was specified when the scroll started with the **start_scroll()** function.

When a graphic that is being used as background of a scroll is modified, it is not automatically updated on the screen. On the contrary, it is necessary to call this function for that purpose.

Once the graphic has been modified, it will remain like this during the rest of the program execution, unless the graphic is unloaded from the memory (with the **unload_fpg()** or **unload_map()** functions) and loaded again. In this case, the original state of the graphic will be restored.

Note:
It can be noticed that, if the displaying graphic is at some coordinates out of the screen, it won't be necessary to call this function, because the parts of the scroll that are gradually appearing on-screen are automatically refreshed.

---

reset_fli()

Description:
This function **rewinds** an **FLI/FLC** animation to the beginning. This animation started with the **start_fli()** function.

After having called this function, the animation will be displayed again from the beginning (to display every frame of the animation, you must call the **frame_fli()** function).

The use of this function deals with the possibility of stopping an animation and repeating it again from the beginning, without unloading it (**end_fli()**) and loading it again.

If the aim is to perform an animation indefinitely, restarting when it is over, then **it is not necessary to use this function**, since it will automatically be done with **frame_fli()**, if you keep on calling once the animation is over.

Only one animation can exist at the same time. Thus, it is not necessary to specify any parameter for this function.

---

reset_sound()

Description:
Advanced function, only for very expert users. Resets the sound system.

This function is used to activate new parameters of the sound hardware.

The following values of the **setup global structure** must be established:

setup.card
setup.port
setup.irq
setup.dma
setup.dma2

This function is normally used inside the sound setup programs.

To activate the rest of the values of the setup structure, those referred to the **mixer** volume, the **set_volume()** function must be called. The values to establish the volume are the following ones:

setup.master
setup.sound_fx
setup.cd_audio

---

roll_palette(<initial colour>, <number of colours>, <increment>**)**

Description:
Rotates a range of palette colours. This function is used to create movement effects in static graphics, like the effect of flowing water.

To use this function, it is first necessary to create graphics that use a range of consecutive colours of the original palette, in a perpetual cycle (for instance, colours ranging from **0** to **15**, painting something with the colours 0, 1, 2, 3, ... , 14, 15, 0, 1, 2, ...).

Then, it is necessary to take care that those colours are not used by other graphics that are going to appear on the screen at the same time, if you do not want to implement the effects on them.

The **increment** (third parameter) is normally **1** to perform the rotation in a direction and **-1** to perform it in the opposite direction, but other values may be used to perform the colours cycle at higher speed.

To perform a cycle of colours from **0** to **15**, it would be necessary to specify **0** as <initial colour> and **16** as <number of colours>.

Notes:
If the aim is to determine the palette with which the cycle of colour must be performed, this palette must be loaded from an archive with the **load_pal()** function.

To perform other palette effects without replacing some colours by other ones in cycles, the **fade()** function must be used. This function allows us to perform many colours fading and saturations at different speeds.

There are two simplified versions of this last function that allow us to carry out a fading to black (**fade_off()**) and to undo it (**fade_on()**).

---

save(<archive name>, <OFFSET datum>, <sizeof(datum)>)

Description:
Saves a data block from the program memory to a file in the disk, to recover it later, when it is required, with the **load()** function.

For that, the function requires the **archive name**, the offset (inside the computer memory) of the variable, table or structure stored on the disk (the datum offset is obtained with **OFFSET <datum name>**) and the number of memory positions that this datum occupies (which may be obtained with **sizeof(<file name>)**).

It is possible to save several data (variables, tables or structures) if they have consecutively been defined inside the same section (**GLOBAL**, **LOCAL** or **PRIVATE**). In this case, the **OFFSET** of the first datum must be indicated as a **second parameter**, and the addition of the **sizeof()** of all the data must be indicated as a **third parameter**.

Note:
It is not necessary to specify a path together with the archive name.

---

set_fps(<n. of frames per second>, <n. of allowed omissions>)

Description:
This function regulates the games' speed; it defines the game's number of frames per second that will be displayed.

By default, the display will be regulated at 18 frames per second, which means that if a process moves a pixel per every (**FRAME**), it will move on-screen at a speed of 18 pixels per second.

This function may establish the number of Frames Per Second (FPS) from a minimum of 4 to a maximum of 200; in general, no more than 24 frames per second are necessary to obtain a fluid and slight movement.

The second parameter, **maximum number of allowed omissions**, is referred to how the program must preferably work when it is executed in a computer fast enough to calculate the required number of frames per second. It works as follows.

Number of allowed omissions.

0 - The game will go at slower speed when it is executed in a computer too slow. That is to say, it will display the frames per second that the computer has had time to calculate.

1 - If the computer can not calculate all the frames, it is allowed to occasionally omit any frame to try to keep the game's relative speed. The game movements will become a little more abrupt, but faster.

2 or more - The game is allowed to omit as many consecutive frames as it is indicated in this parameter to maintain the original relative speed of the game. For instance, if the number of omissions is set at 4 and in the game a process moved one pixel by one, in a very slow computer it could move even in four pixels at a time.

---

set_mode(<new videomode>**)**

Description:
Establishes a new video mode for the game execution. The allowed videomodes that may be specified as a parameter are the following ones:

m320x200 - VGA standard
m320x240 - X Mode
m320x400 - X Mode
m360x240 - X Mode
m360x360 - X Mode
m376x282 - X Mode
m640x400 - SVGA VESA
m640x480 - SVGA VESA
m800x600 - SVGA VESA
m1024x768 - SVGA VESA

When a change of the videomode in the program is made, a fading to black (of the program's colours palette) will automatically be performed and in the following displays, the colours palette will gradually be restored. That is to say, **set_mode()** always performs a **fade_off()** just before changing the videomode and a **fade_on()** just after having changed it.

By default, all the programs start with the 320 by 200 pixel activated mode (**set_mode(m320x200)**).

Note:
By using the **set_mode()** function, all the scroll and mode 7 windows that were activated in the game, as well as all the processes displayed inside them, will be deleted.

---

set_volume()

Description:
Advanced function, only for very expert users. Adjusts the different volume controls managed by the **mixer** of the system sound.

To adjust the volume, the following values of the **setup global structure** must be set:

setup.master - General volume
setup.sound_fx - Sound effects volume
setup.cd_audio - Cd-audio music volume

This function is normally used inside the sound setup programs, or even in the rest of the programs, normally to adjust the CD_Audio music volume.

Note:
To activate the rest of the values of the setup structure (those referred to the sound card's parameters) the **reset_sound()** function must be called with the following defined values of the structure:

setup.card
setup.port
setup.irq
setup.dma
setup.dma2

---

signal(<id>, <signal>)

Description:
Sends a signal to a process (an object of the game). This function is mainly (but not only) used to destroy (to kill) a process from another one, sending it a **s_kill** signal.

Any process may send a signal to another one, provided that the former has the identifying code of the latter.

The **signal types** that may be sent to a process are the following ones:

s_kill - Order to **kill** the process. The process will not appear in the following frames of the game any longer.

s_sleep - Order to **make** the process **dormant**. The process will remain paralysed, without executing its code and without being displayed on screen (nor being detected by the rest of the processes), as if it had been killed. But the process will continue to exist in the computer's memory.

s_freeze - Order to **freeze** the process. The process will remain motionless without running its code. But it will continue being displayed on screen and it will be possible to detect it (in the collisions) by the rest of the processes. The process will continue to exist in the computer's memory, even if its code is not executed.

s_wakeup - Order to **wake up** the process. It returns a **slept** or **frozen** process to its normal state. The process will be executed and displayed again from the moment that it receives this signal normally. A process that has been deleted (killed) can not be returned to its normal state, since it does not exist in the computer's memory any longer.

A process can also send these signals to itself, taking into account that the identifying code of a process is always **ID** (word reserved in the language to this purpose). The statement would be as follows:

signal(id,<signal>)

Self-deleting a process in this way, sending a **s_kill** signal to itself, will not instantaneously destroy the process, but in the following (**FRAME**) display. The **RETURN** statement can be used to immediately delete a process. **All the signals sent to processes will be implemented just before the next display of the game,** that is to say, in the next **FRAME** of the game (not instantaneously).

Together with these four signals, there are other four signals that directly correspond to the previous ones. They are: **s_kill_tree, s_sleep_tree, s_freeze_tree** and **s_wakeup_tree.**

These signals are sent not only to the indicated process, but also **to all the processes that it has created.** That is to say, if a **s_kill_tree** signal is sent to a process, the latter and all its descendants (sons, grandsons, ...) will be deleted as well as all the processes created by it and the processes created by the latter.

An exception to these last four signals is when there is an **orphan process,** that is to say, a process whose father (the process that called it) is already dead. The orphan processes will not receive the signal when it is sent to a process from which they are descended as, on their father having disappeared, it won't be able to send the signal to the processes it created.

Note:
When a process is created, the system defines the **son** variable of the father with the identifying code of the son, and the **father** variable of the son with the identifying code of the father.

---

signal(TYPE <process name>, <signal>)

Description:
This second meaning of the **signal** function is similar to the previous one, with the exception that, instead of sending a signal to a process from its identifying code, it allows us to send a signal **to all the processes of a specific type** or to them and their descendants, when the used signals are of the type **s_kill_tree.**

For instance, if several processes of the **enemy** type exist or may exist in a game, and the aim is to freeze these processes (without freezing their descendants), the following statement will be used:

signal(TYPE enemy, s_freeze);

As it can be noticed, it is necessary to have the identifying code of a specific process in order to send a signal to it. To delete a group of processes, it is necessary either that they are of the same kind, that this group is made up of a process and its descendants, or that all their identifiers are known (in order to send them the signal one by one).

It is possible to send a signal to a type of process, even if no process of this type is being executed in the game. But if a signal is sent to a process that has already been killed, with its identifying code (first meaning of the **signal** statement), there is a risk that the identifying code is now used by another process, which is going to receive the signal. This happens, for instance, when the aim is to kill a process that has already been killed, as it is possible that another different one is being killed.

Note:
If the aim is to delete all the processes except the current one, the **let_me_alone()** function may be used. This function sends a **s_kill** signal to all the processes, except the one that executed this function.

---

sound(<sound code>, <volume>, <frequency>)

Returns:
The **channel number** through which the sound is played.

Description:
Plays the effect whose **sound code** is specified as first parameter. At first, the sound must have been loaded from a PCM archive with the **load_pcm()** function. This function returns the **sound code** corresponding to this effect.

As a second parameter, it is necessary to specify the **volume** at which the sound is intended to be reproduced, taking into account that **0** is the minimum volume, and **256** the maximum volume.

As the third parameter, you must specify the **frequency** (speed) at which the sound is intended to be reproduced, being **256** the standard frequency that will reproduce the original sound. With lesser values, the sound will be reproduced with more accentuated bass. On the contrary, with higher frequency values, it will be reproduced with more accentuated treble.

The function returns the **channel number** that can be used by the **stop_sound()** function to stop the sound and by the **change_sound()** function to modify its volume or frequency.

There are 16 sound channels. Thus, up to 16 sounds may be simultaneously played.

---

sqrt(<expression>)

Returns:
The entire square root of the expression.

Description:
Calculates the square root of the expression passed as a parameter, **truncated to an Integer.**

For instance, as a result, **sqrt(10)** will return **3** and not **3.1623**, which is the real value (approximately) of the square root of ten.

---

start_fli(<archive name>, <x>, <y>)

Returns:
The animation's number of frames.

Description:
Starts a **FLI/FLC** animation contained in the specified **archive**, in the coordinates (*x*, *y*) (the upper left coordinate of the display window must be specified).

The path can be specified in the <archive name>. The path is not necessary if the archive is in the DIV Games Studio directory or in a subdirectory whose name coincides with archive extension (for instance, "fli\anima.fli").

The screen must hold the whole animation. That is to say, if the animation occupies the whole screen, the videomode must be fixed at first with the **set_mode()** function, starting then the animation at the (0, 0) coordinate with the **start_fli()** function.

For your information, the function returns the number of frames that the whole animation comprises.

The system will automatically activate the colour palettes that the **FLI/FLC** animation could have. This can cause problems dealing with the representation of other graphics or fonts of the program, if they had been drawn with a different palette.

If the aim is to combine other graphics with animation on-screen, the latter must have just one colour palette (which is normally called **"palette low FLI/FLC"**) and the graphics must have been drawn with that same palette.

Once the animation has started, its frames will gradually be shown with respective calls to **frame_fli()**.

It is possible to have but one active animation at every time. Therefore, after having started an animation with **start_fli()** and having been displayed with **frame_fli()**, this animation must finish with the **end_fli()** function before starting another different animation.

Note:
The **reset_fli()** function allows us **to rewind** the animation, so that the **frame_fli()** function continues to execute it from the beginning.

---

start_mode7(<number of m7>, <file>, <graphic>, <external graphic>, <number of region>, <height of horizon>)

Description:
This is an advanced function which requires a skillful user.

Creates a mode-7 display window. That is to say, it displays a **three-dimensional** graphic in a folded plane. In order to obtain this effect, this function will be called with the following parameters:

<m7 number> - Up to 10 mode-7 windows can be created on-screen, numbered from 0 to 9. If the aim is to create but one, the best thing is to define window number 0. This number will be necessary later to modify the window parameters, as the system will need to know which one of the possible 10 mode-7 windows is intended to modify.

<file> - The graphics intended to be folded in the window must be in a file whose **file code** must be specified here, as a second parameter of the function. The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the file with the code 0).

<graphic> - The third parameter must be the **code of the main graphic** that is going to be folded in the window and it must belong to the file previously indicated.

<external graphic> - Here, it is possible to indicate either a 0, if the aim is not to see any graphic beyond the graphic folded in the perspective, or a **graphic code** of the same file that will be shown in the perspective beyond the **main graphic**, until it gets the horizon. The height and width of this graphic must be powers of two, not higher than 8192 (these powers of two are: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 and 8192). For instance, it can be a 64 pixel width by 32 pixel height graphic. This graphic will also be shown folded.

<region number> - Here, the rectangular screen region in which the mode-7 is going to be shown, will be indicated. If 0 is indicated as a region number, this region will be shown on the whole screen. The rest of regions must previously be defined with the **define_region()** function (a **region** is but a rectangular zone of the screen).

<Height of the horizon> - The last parameter to indicate will be the distance, in pixels, from the upper part of the window, where the horizon line is intended to be put. If the camera is placed above the folded plane, then nothing will be displayed above the horizon line (this space is normally filled with another scroll or mode-7 window). Otherwise, if the camera is placed below the plane, then nothing will be shown below this horizon line.

Besides the call to the function, some values of the m7 global structure must be initialised for the window to work correctly. This is a structure of 10 records (one for every possible mode-7 window) and every record has the following fields:

Camera - Identifying code of the camera
Height - Height of the camera
Distance - Distance of the camera
Horizon - Height of the horizon
Focus - Focus of vision
Z - Depth plane
Colour - Exterior colour

Important:
In order to activate the mode 7 window it is **essential** to start the **camera** field as, without this field, the window can not determine from where the folded plane must **be seen.**

The camera will be placed in the folded plane, at the indicated **distance** of the process whose **identifying code** has been set in **camera**, orientated at its angle itself (the one indicated by its **angle** local variable). The height at which the camera is located with respect to the bottom will be that indicated in the **height** field.

See the help about the **m7 structure** for further information about these issues, or about how to access them.

How to visualise process graphics in mode 7.
To create a process whose graphic is displayed in the mode 7, its **ctype** local variable must be defined as **c_m7 (type of coordinate as mode 7 coordinate)**, which will be done with the following statement:

```
ctype=c_m7;
```

After this, the process will be displayed in the mode 7 with its graphic (**graph**) scaled depending on the distance at which it is. The process must only modify its ***x*** and ***y*** variables to move through the folded plane.

When a process belongs to the mode-7 (that is to say, the value c_m7 has been assigned to its local variable ctype):

- Its ***x*** and ***y*** variables will be referred to the folded main graphic's point above which the process graphic will be placed.
- Its ***z*** variable will lose its effect, as the graphics will appear in strict order of depth. This variable will only be useful to indicate display priorities in graphics exactly placed in the same depth plane.
- The process will automatically be deleted when the mode 7 window, to which the process belongs, is deleted with the **stop_mode7()** function, or when the videomode is changed with the **set_mode()** function as, by doing so, the mode 7 windows will also be deleted.

If there were several **mode 7** windows, the process would be displayed in all of them by default. If the process had to be displayed just in one of them, its **cnumber** local variable should be then defined. For instance, if there were 6 mode 7 windows (from number 0 to number 5) and the aim was to display a process only in windows 0 and 2, the following statement should be included in it:

```
cnumber=c_0+c_2;
```

For a process to have several graphics (several views), depending on the angle from which it is observed, its graphic must be defined with the **xgraph** local variable (instead of the **graph** variable). To define this variable, it is necessary, at first, to create a table (of any name), first indicating the graphic's **number of views** and then the **graphics codes** for these views, starting with **angle 0** and in an counterclockwise direction. For instance:

```
GLOBAL
  views_car[ ]=4,100,101,102,103;
```

The table **views_car** would define 4 views: graphic 100 for angles near 0 degrees, graphic 101 for angles near 90 degrees, graphic 102 for angles near 180 degrees, etc.

And then, the **xgraph** variable must be initialised in the process code with the following statement:

```
xgraph=OFFSET views_car;
```

To get an example about what we have just seen, examine some of the DIV Games Studio's sample games that use this technique. Thus, read the comments about these programs (for instance, see the program **Speed for dummies**).

---

start_scroll(<scroll number>, <file>, <graphic>, <background graphic>, <number of region>, <locking indicator>)

Description:
This function has a certain complexity, requiring a skillful user.

Creates a scroll window, in which it will perform a view against a background graphic (the décor of the game). That is to say, by using a graphic bigger than the display window as a game background, a part of this graphic can be showed and shifted in any direction.

To obtain this effect, this function will be called with the following parameters:

<scroll number> - Up to 10 scroll windows can be created on screen, numbered from **0** to **9**. If the aim is to create only one, the best thing is to define window number 0. This name will later be necessary to modify the parameters of the window, as the system will need to know which one of the **10** possible scroll windows is intended to change.

<file> - The graphics that are intended to be shown as a background or décor in that window must be in a file whose **file code** must be specified here, as a second parameter of the function. The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the file with the code 0).

<graphic> - The third parameter must be **the code of the main graphic** that is going to be displayed as a background in the window and that must belong to the file previously indicated. This graphic is normally the main décor of the game on which the action will be developed. It is a graphic bigger than the display window, that will be shifted in one or several directions and on which the graphics of the game will be placed.

The scroll window will be initially placed with the **control point** number **0** of this graphic in the upper left corner, when this point has been defined in the **graphic editor**.

<background graphic> - Here, **0** will be indicated if the aim is to obtain a single scroll plane (a single background graphic), or another **graphic code** if it is intended that it appears as scroll background (deeper), behind the foreground. In order to see this background plane, it is essential that the **main graphic** (foreground) has parts painted in colour number 0 of the palette, as these transparent zones will allow us to see the **background graphic** through them.

<region number> - The rectangular screen region in which the scroll is going to be shown will be here indicated. If **0** is indicated as a region number, it will be shown on full screen. The rest of regions must previously be defined with the **define_region()** function (a **region** is but a rectangular zone of the screen).

<locking indicator> - A value defining whether each of the two scroll planes is horizontally and vertically cyclical will be here indicated. For instance, a plane is horizontally cyclical when, on leaving the picture on the right, the picture appears on the left. To obtain this value, the following quantities must be added:

+ 1 - If the foreground is horizontally cyclical
+ 2 - If the foreground is vertically cyclical
+ 4 - If the background is horizontally cyclical
+ 8 - If the background is vertically cyclical

That is to say, **0** if none of the two planes must be cyclical, **15 (1+2+4+8)** if both planes must be cyclical in both axes, **12 (4+8)** if only the background must be cyclical, etc.

When a (foreground or background) graphic is smaller than the display window, the system will force it to have a cyclical scroll plane. Otherwise, the scroll window could not be completely filled, without **cyclically** repeating the graphic (tile).

Besides the call to the function, some values of the scroll global structure must be initialised for the correct working of the window .This is a structure of 10 records (one for each possible scroll window) and every record has the following fields:

x0, y0 - Foreground coordinates
x1, y1 - Background coordinates
z - Depth plane
camera - Identifying code of the camera
ratio - Background's relative speed
speed - Foreground's maximum speed
region1 - First screen region
region2 - Second screen region

There are two ways to program the movement of the scroll windows:

- Manually, modifying in each frame of the game the fields **x0**, **y0**, **x1** and **y1** of this structure (the scroll planes' coordinates).

- Automatically, for what the **identifying code** of a process is needed in the field **camera** of this structure. From then, the system will be in charge of following the graphic of this process in the scroll window.

See the **scroll structure** either for further information about these fields, or to know how to access them.

How to display processes' graphics in the scroll.
In order to create a process whose graphic is displayed in the scroll window, it is necessary to define its ctype local variable as **c_scroll (type of coordinate as scroll coordinate)**, which will be done with the following statement:

ctype=c_scroll;

After that, the process will be displayed in the scroll with its graphic (defined in the **graph** local variable). The process must modified only its ***x*** and ***y*** variables to scroll.

When a process belongs to the scroll (assigning the value c_scroll to its local variable **ctype**):

- Its **x** and **y** variables will be referred to the point of the foreground's graphic on which the graphic of the process will be placed.

- Its **z** variable will be now referred to the **variables z** of the processes that also belong to the same scroll window. That is to say, each time that a scroll window is displayed, all the graphics that belong to it (ranged by their **z**) will be displayed just after it. Then, the processes that **don't belong to that scroll window** will continue to be displayed.

- The process will be automatically eliminated when the scroll window to which the process belongs is eliminated with the **stop_scroll()** function. Or when the videomode is changed with the **set_mode()** function as, on doing so, the scroll windows will be also eliminated.

If there were several **scroll** windows, the process would be displayed by default in all of them. If it had to be displayed only in some of them, its **cnumber** local variable should be defined. For instance, if there were 6 scroll windows (numbered from 0 to 5) and the aim was to display a process only in windows 0 and 2, the following statement should be included in it:

cnumber=c_0+c_2;

In order to observe an example of what it has been said, the best thing is to examine some of the sample games of DIV Games Studio that use this technique. Thus, the reader is directly referred to the comments of these programs (for instance, see the example **Helioball**).

---

stop_cd()

Description:
Turns the CD-Audio off, stopping the song that was playing. The songs are reproduced with the **play_cd()** function.

Note:
The cd-audio reproduction volume may be controlled with the **setup** structure and the **set_volume()** function.

---

stop_mode7(<number of m7>**)**

Description:
Deletes the mode 7 window whose number (from 0 to 9) is passed as a parameter. This <m7 number> was indicated as first parameter in the **start_mode7()** function, and it is necessary as there can be up to 10 different mode 7 windows, and the system needs to know which one is finishing.

On deleting a mode 7 window, all the processes that exclusively belong to that window will be killed automatically. These processes have their **ctype** variable with the **c_m7** value and they are not displayed in any other mode 7 window.

Important:
On changing the videomode with the **set_mode()** function, all the mode 7 windows (and their processes) will also be deleted. In this case, it is not necessary to use this function **(stop_mode7()).**

Note:
The creation of a mode 7 window is a somewhat advanced process and it requires to start several parameters, like that of the camera, some of them required by the **start_mode7()** function and some others contained in the **m7** global structure (like the **m7.camera** variable).

---

stop_scroll(<scroll number>**)**

Description:
Deletes the scroll window whose number (from 0 to 9) is passed as a parameter. This <scroll number> was indicated as first parameter in the **start_scroll()** function and it is necessary since there can be up to 10 different scroll windows, and the system needs to know which one is finishing.

On deleting a scroll window, all the processes that exclusively belong to that window will automatically disappear. These processes have their **ctype** variable with the **c_scroll** value and they are not displayed in any other scroll window.

Important:
On changing the videomode with the **set_mode()** function, all the scroll windows (and their processes) will also be deleted. In this case, it is not necessary to use this function (**stop_scroll()**).

Note:
The creation of a scroll window is a somewhat advanced process and it requires the start of several parameters, some of them required by the **start_scroll()** function and some others contained in the **scroll global structure** (like the **scroll.x0** variable).

---

stop_sound(<channel number>**)**

Description:
Stops the sound that is being played through the channel, passed as a parameter.

The required <channel number> is the value returned by the **sound()** function when the reproduction of a sound effect starts.

There are 16 sound channels. Thus, up to 16 sounds may simultaneously be played.

Note:
To stop a sound gradually, turning its volume down little by little, several calls to the **change_sound()** function must be made to slightly reduce the channel volume until it reaches level 0. Then, the **stop_sound()** can be called to definitively stop the sound.

---

system(<"external command">**)**

Description:
Executes the operating system's command passed as a parameter.

Notes:
One of the utilities of this command is, for instance, to delete a temporary archive that has been created in the program, calling the command of the system **DEL** <archive name>.

The system can be blocked depending of the executed commands. In these cases you must reset the computer. There is no guarantee dealing with this function running, due to the multiple incompatibilities that can appear between the external commands and the manager of internal processes of DIV Games Studio.

---

unload_fnt(<font code>**)**

Description:
Unloads from the memory the **font (the type of letter** or the **set of graphic characters)** whose code is passed as a parameter.

This <font code> is the value returned by the **load_fnt()** function by loading a new letter font stored in a **archive FNT** in the computer's memory.

After having unloaded a font, **much care must be taken** not to go on using this font in the program. Otherwise, the program can become blocked.

It is not necessary to unload the font before finishing the program, as the system will do it automatically.

So, a font must be unloaded from the memory only when it is not going to be used for a specific time and when the aim is to free up the occupied space in the computer's memory to load other resources (other graphics files, sounds, fonts, etc.).

Note:
Font number 0, (the system font having 0 as font code), **can not be unloaded.**

---

unload_fpg(<file code>)

Description:
Unloads the graphics file whose code is passed as parameter from the memory. This <file code> is the value returned by the **load_fpg()** function when a new graphics file is loaded in the memory.

After having unloaded a graphics' file, **much care must be taken** not to go on using in the program any graphic that was in that file. Otherwise, the program could become blocked.

It is not necessary to unload the file from the memory before finishing the program, as the system will do it automatically.

Therefore, a file must be unloaded from the memory only when it is not going to be used for a while and when the aim is to free up space occupied in the computer's memory in order to load other resources (other graphics files, sounds, fonts, etc.).

Note:
The graphics individually loaded with the **load_map()** function will not be unloaded when file number 0 (with code 0) is loaded , even if they were used as if they belonged to it. These graphics will have to be unloaded by using the **unload_map()** function.

---

unload_map(<graphic code>)

Description:
Unloads the graphic whose code is passed as a parameter from the memory. This <graphic code> is the value returned by the **load_map()** function by loading a new graphic stored in a **archive MAP** in the computer's memory.

After having unloaded a graphic, **much care must be taken** not to go on using this graphic in the program. Otherwise, the program can become blocked.

It is not necessary to unload the graphic before finishing the program, as the system will do it automatically.

So, a graphic must be unloaded from the memory only when it is not going to be used for a specific time and when the aim is to free up the occupied space in the computer's memory to load other resources (other graphics files, sounds, fonts, etc.), which will make sense only with graphics of a certain size, that is to say, big enough so as to be worth freeing up the space they occupy.

Note:
The graphics individually loaded with the **load_map()** function will not be unloaded when file number 0 (with code 0) is unloaded with the **unload_fpg()** function, even if these graphics are used as if they belonged to it.

---

unload_pcm(<sound code>)

Description:
Unloads the sound whose code is passed as a parameter from the memory. This <sound code> is the value returned by the **load_pcm()** function when a new sound effect is loaded in the memory.

After having unloaded a sound effect, **much care must be taken** not to go on using in the program this effect (its code) for the **sound()** or **unload_pcm()** functions. Otherwise, the program could become blocked.

It is not necessary to unload the sound from the memory before finishing the program, as the system will do it automatically.

Therefore, a sound must be unloaded from the memory only when it is not going to be used for a while and when the aim is to free up space occupied in the computer's memory to load other resources (other graphics files, sounds, fonts, etc.), which will be logical just with sound effects of a certain length, that is to say, big enough so as to be worth freeing up the space they occupy.

Note:
The **stop_sound()** function must be used to stop a sound effect, but keeping it in the memory in order to be played again when desired.

---

write(<font>, <x>, <y>, <centering code>, <text>)

Returns:
The identifying code of the text that has been written.

Description:
This function is used to show an alphanumeric text on-screen. For that, it requires the following parameters:

<font> - The **font code** or **type of letter** that is going to be used. Here, you must put either 0 when the aim is to use the system's font (white, small font, 6 by 8 pixels), or the **font code** returned by the **load_fnt()** function when a new font is loaded in the program.

<x>, **<y>** - The coordinates referred to the screen in which the text is going to be displayed, first in the horizontal axis and then in the vertical one.

<centering code> - This code determines the position of the text specified by the previous coordinates. Its values are:

0-Up left	**1**-Up	**2**-Up right
3-Left	**4**-Center	**5**-Right
6-Down left	**7**-Down	**8**-Down right

For example, if a text is written at the 160, 0 coordinates and with the centering code 1 (Up), then the text will center in the column 160 and it will be displayed from line 0 downwards. Or, if the aim is to have a text in the upper left corner, it must be displayed at the 0, 0 coordinates and with centering code 0 (Up left).

<text> - The text to be written as a literal (that is to say, a text in inverted commas) will be specified as last parameter.

The displayed text will remain on-screen until it is deleted with the **delete_text()** function, that requires as parameter the **identifying code** returned by **write()**.

The **write_int()** function must be used to display the numeric value of a variable (such as the score of the player).

The texts will remain unchangeable on screen even if graphics are displayed on it or processes' graphics pass before or behind them.

Notes:
The depth plane in which the written texts appear is controlled through the **text_z** global variable, that is useful to regulate which graphics must be seen above the texts and which ones must be seen below them.

Then, it will be possible to move the texts towards another position if necessary, by using the **move_text()** function, which also requires the **identifying code** returned by **write()** as parameter.

When fonts loaded from **archives FNT** are used, the colours' palette used to generate these fonts must be activated (see **load_pal()**). Otherwise, the colours may appear changed, being the text incorrectly displayed.

---

write_int(<font>, <x>, <y>, <centering code>, <OFFSET variable>)

Returns:
The identifying code of the text that has been written.

Description:
This function is used to show the numeric value of a variable. For that, it requires the following parameters:

<font> - The **font code** or **type of letter** that is going to be used. Here, it is necessary to put either 0 when the aim is to use the system's font (white, small font, 6 by 8 pixels), or the **font code** returned by the **load_fnt()** function when a new font is loaded in the program.

<x>, <y> - The coordinates referred to the screen in which the numeric value is going to be displayed, first in the horizontal axis and then in the vertical one.

<centering code> - This code determines the position of the numeric value specified by the previous coordinates. Its values are:

0-Up left	1-Up	2-Up right
3-Left	4-Center	5-Right
6-Down left	7-Down	8-Down right

For example, if a numeric value is written at the 160, 0 coordinates and with the centering code 1 (Up), then the numeric value will be centered in the column 160 and it will be displayed from line 0 downwards. Or, if the aim is to have a numeric value in the upper left corner, it must be displayed at the 0, 0 coordinates and with centering code 0 (Up left).

<OFFSET variable> - The offset inside the computer's memory of the variable whose value is intended to be displayed, must be specified as last parameter (the offset of the data is obtained with the **OFFSET** operator).

The displayed numeric value will remain on-screen until it is deleted with the **delete_text()** function, that requires as parameter the **identifying code** returned by **write_int()**.

Important:
During the time that the value of the variable appears on screen, this value will automatically be updated every time the variable is modified. For that, new calls to **write_int()** are not necessary.

The **write()** function must be used to display any kind of alphanumeric text (a fixed text).

The texts will remain unchangeable on screen even if graphics are displayed on it or processes graphics pass before or behind them.

Notes:
The depth plane in which the written texts appear is controlled through the **text_z** global variable, that is useful to regulate which graphics must be seen above the texts and which ones must be seen below them.

Then, it will be possible to move the texts towards another position if necessary, by using the **move_text()** function, which also requires the **identifying code** returned by **write()** as parameter.

When fonts loaded from **archives FNT** are used, the colour palette used to generate these fonts must be activated (see **load_pal()**). Otherwise, the colours may appear changed, being the text incorrectly displayed.

Warning:
It is not possible to display an expression, as it is shown below:

```
write_int(0,0,0,0,OFFSET variable + 1);
```

To display the value of the variable plus 1. That is to say, if the aim was to display this value, it would be necessary either to add 1 to the variable or to create another variable, assigning it the value of the original variable plus 1, for instance:

```
variable2 = variable + 1;
write_int(0,0,0,0,OFFSET variable2);
```

In this case, you should take into account that you had to update the value of the **variable2** at least once every **FRAME** of the game, as by changing **variable** the value of **variable2** will not automatically be updated unless the **variable2 = variable + 1** statement is again executed.

---

xput(<file>, <graphic>, <x>, <y>, <angle>, <size>, <flags>, <region>)

Description:
Advanced version of the **put()** function to put a graphic on the screen background. This function requires the following parameters, in order:

<file> - file code with the graphics library that contains both graphics. The graphics loaded with the **load_map()** function will be used as if they belonged to the first file (the file with the code 0).

<graphic> - code of the graphic inside the file that is going to be displayed on screen.

<x>, <y> - coordinates dealing with the screen where the graphic is intended to be put. These coordinates reveal the position in which the graphic center (or the **control point** number 0, if it is defined) will be placed.

<angle> - angle (in degree thousandths) in which the graphic is going to be displayed; the normal angle is **0**.

<size> - size (in percentage) in which the graphic is going to be displayed; the normal size is **100**.

<flags> - Indicates the mirrors and transparencies with which the graphic will be displayed; the possible values are the following ones:

0-Normal graphic
1-Horizontal mirror
2-Vertical mirror
3-Horizontal and vertical (180°) mirror
4-Transparent graphic
5-Horizontal transparent and mirror
6-Vertical transparent and mirror
7-Horizontal and vertical transparent, mirror

<region> - Number of region (window inside the screen) in which the graphic must be displayed. This value will normally equal **0** to display the graphic at any position of the screen. The **define_region()** function must be used to define a region.

The graphics displayed in this way on the background screen will be in the game display **below all the processes, scroll regions, texts, etc.**

If the aim is that a graphic is above others, it is necessary to **create it as a new process** and fix its **z** variable with the priority of its display.

The **clear_screen()** function must be used to clear the screen background.

Notes:
The **put()** function is a simplified version of the **xput()** function, and it is useful when you do not want to rotate, scale, mirror or display the graphic with transparencies.

The **map_put()** or **map_xput()** functions must be used to put a graphic inside another one (instead of the screen background).

If the graphic intended to be put is just a screen background, it is easier to use the **put_screen()** function, as it does not require the screen coordinates because it will automatically center the graphic on screen.

APPENDIX C

C

Appendix C: Data Predifined In The Language

C1 – Predifined GLOBAL Structures

GLOBAL STRUCT joy

This global structure is used to control the **joystick**. It contains a series of logical fields related to the programming of this device: the state of the buttons (whether they are pressed or not) and the state of the main four control directions.

To access these fields, the name of the field must be preceded by the word **joy** and the symbol **.** (period). For instance, to access the **left** field (which indicates whether the left control is pressed), it is necessary to use **joy.left**.

---

left - This field will be at **1** when the **joystick** is orientated to the **left**, and at **0** in the opposite case.

right - This field will be at **1** when the **joystick** is orientated to the **right**, and at **0** in the opposite case.

up - This field will be at **1** when the **joystick** is orientated **up**, and at **0** in the opposite case.

down - This field will be at **1** when the **joystick** is orientated **down**, and at **0** in the opposite case.

For instance, to perform an action in a program **when the joystick is moved to the right (joy.right)**, a statement like the following one must be included in the code:

```
IF (joy.right)
    // Action to perform (statements)
END
```

For diagonal positions, the two fields comprising this diagonal must be verified. For instance, to perform an action when the **joystick** is in the upper right diagonal, the following statement will be used:

```
IF (joy.up AND joy.right)
    // Action to perform (statements)
END
```

---

button1, **button2**, **button3** and **button4** - These fields indicate the state of up to four joystick's buttons, being at **1** when the respective button is pressed, and at **0**, when it is not.

Some joysticks only have 2 buttons. In this case, they will be buttons number 0 and 1. In computers with two connected joysticks, the second joystick will have the buttons number 2 and 3.

Note: When an analogical reading of the **joystick** is required (to know the exact coordinates at which the joystick is located), it will be necessary to use the **get_joy_position()** function. Obviously, this function will only be useful in an analogical **joystick**, and it won't work in the digital ones.

GLOBAL STRUCT m7

This 10 record structure contains certain fields dealing with changeable parameters of the **mode 7 window**. The ten records have the same field names, but each of them modifies the parameters of a different mode 7 window (as up to 10 windows of this type may be activated).

A **mode 7 window** could be defined as a screen region that shows a graphic plane three-dimensionally folded (for instance, like a sheet of paper with a picture horizontally positioned, displayed on screen with a virtual bottom (or top).

For a record (numbered from 0 to 9) of the **m7 structure** to make sense, that **mode 7 window** (from 0 to 9) must first be activated with the **start_mode7()** function (see this function for further information about the mode 7 windows).

It is understood that the fields of this structure are complementary to the call parameters of this function. In order to observe a practical example of a mode 7, it is possible to access the help about the **start_mode7()** function.

How to use the m7 structure:
To access these fields, the field name must be preceded by the word **m7**, the number of record in square brackets and the symbol . (period).

For instance, if two mode-7 windows, number 0 and number 1 were initialised, the **camera** variable of both windows could be accessed as **m7[0].camera** and **m7[1].camera**, respectively. When the mode-7 window number 0 is accessed, it is also possible to omit the number of windows in square brackets. That is to say, the **m7.camera** variable and the **m7[0].camera** variable are, to all ends, the same for the language.

---

camera - **Identifying code of the process** followed by the camera. To move the camera that controls the mode-7 view, only a mode-7 process must be created, a process having its local variable **ctype=c_m7**, and its **identifying code** must be put in the **camera** variable of this structure. After so, only the **x**, **y** and **angle** local variables of this process must be modified and, for instance, the **advance()** function must be used to move the camera forward.

For the mode 7 window to be activated, it is **essential** to initialise the **camera** field. Without this field, the window can not determine from where the folded plane **must be seen.**

---

height - Height of the camera. This variable of the structure manages the distance to which the camera is placed from the bottom. By default, its value equals **32**. Any positive number will make the camera be placed upper as the number is greater. If a negative number (less than zero) is put in the **height** field of this structure, then the camera will be placed below the folded plane, showing a "**top**" instead of a "**bottom**".

Two mode-7 images can be created within the same region: one as top and the other as bottom (one with positive **height** and the other with negative **height**). In this case, it is important to establish the **z** variable of the **m7 structure** of both, thus to determine the depth plane in which each one must be painted.

---

distance - Distance from the camera to the followed process. The perspective of the camera will always be positioned slightly behind the process whose identifier has been put in the **camera** field of the structure. This is done for the graphic of the process used as a camera to be seen, just in case this process has defined it (in its **graph** or **xgraph** local variable).

By default, the camera will be positioned at **64** points behind the process. "Behind" means a point placed at the indicated distance from the graphic in the angle opposite to that one to which the process is orientated. For instance, if the process is facing right, **64** points to its left.

---

horizon - Horizon's height. This is the same value as that indicated as last parameter of the **start_mode7()** function. Its initial value will equal to the one indicated in the call to this function. The utility of this variable is to make the horizon go up or down in every frame of the game, depending on the needs of the latter.

On changing the **horizon's height**, the "**facing up**" and "**facing down**" effects will be obtained in the **mode 7 window**.

---

focus - Focus for the camera. This variable controls the perspective of the camera. By default, its value equals **256**, but any value ranging from **0** and **512** may be put, obtaining different distortion effects of the three-dimensional plane.

That is to say, this field controls the angle got by the camera focus. The greater this value is, the closer all the objects (processes) placed in the folded plane will be seen.

---

z - Mode-7 display priority. To indicate the depth plane in which this window must be painted, with respect to the rest of processes. By default, this variable will equal **256**, which means that, as the processes have their **local z** variable at **0** by default, the mode-7 window will be painted in a greater (deeper) depth plane, being the graphics of the processes painted above the window. This situation may change by modifying the **z** variable of the window (for instance, putting it at **-1**) or the **z** variable of the processes (for instance, putting it at **257**).

colour - Colour for the mode-7 exterior. When, in the call to the **start_mode7()** function, any **external graphic** is not specified (the fourth call parameter is put at **0**), this variable will control the colour in which it is aim to paint the exterior. In other words, the colour that the screen must be painted in beyond the graphic that is being folded (beyond its limits).

By default, this field is initialised at 0, which is normally the black colour in the colour palette, Therefore, if this field is not assigned another value (and an external graphic is not defined) the screen will be seen in black beyond the foreground.

GLOBAL STRUCT mouse

This global structure is used to control the mouse. It contains a series of fields related to the programming of this driver, such as the screen position, the pointer graphic, the state of the buttons, etc.

In order to access these fields, the name of the field must be preceded by the word **mouse** and by the symbol . (period). For instance, in order to access the field ***x*** (horizontal coordinate of the mouse pointer), it is necessary to use **mouse.x**.

---

x, y - Horizontal and vertical coordinates of the mouse. It will be necessary to read only these two fields (**mouse.x** and **mouse.y**) to know the position of the mouse cursor on screen. To **position the mouse at other coordinates** (to force its position), assign the new coordinates to these two fields.

---

graph - **Graphic code** assigned as a mouse pointer. By default **the mouse won't be visible.** To make it visible, it is necessary to create the graphic that is going to be used as a pointer in the **graphic editor**, to load it in the program (with the **load_fpg()** or **load_map()** functions, depending on whether this graphic has been stored in a **file FPG** or in an **archive MAP**) and finally, to assign its **graphic code** to this variable (**mouse.graph**). Then, the mouse pointer will be seen on screen.

The center of the graphic will appear at the **mouse.x**, **mouse.y** coordinates, unless its **control point** number **0** has been defined in the **graphic editor**. If this point (usually called **hot spot**) is defined, then it will appear at the coordinates indicated in the fields **mouse.x** and **mouse.y**.

For instance, if an arrow is created to depict the mouse pointer (as it happens dealing with the mouse pointer of DIV Games Studio), the hot spot (**control point** number **0**) will be defined in the upper left corner of the graphic, as it is the **active point** inside the graphic. Then, when the mouse was located at the (0, 0) coordinates, for instance, the **"tip of this arrow"** would precisely be located at those coordinates.

---

file - **File code** containing the graphic. The **file code** containing the graphic of the mouse pointer is defined in this field. **It is not necessary** to indicate a value here if the graphic was loaded from an **archive MAP**, or if it is stored in the first **archive FPG** loaded in the program. Otherwise, **mouse.file** will have to be assigned the **file code** that returned the **load_fpg()** function on loading the file that contains the graphic of the mouse pointer.

---

z - Priority of the graphic display. Indicates the depth plane in which the graphic of the mouse pointer must be displayed. **By default** this field will be equal to **-512**, which implies that **the pointer will be seen above the rest of graphics and texts**. The bigger this field is, the deeper the mouse pointer will be located.

If the aim was to make a graphic of a process appear above the mouse pointer, suffice would be to assign an integer less than **-512** (for instance, **-600**) to the **local z** variable of that process.

---

angle - angle with which the graphic of the mouse pointer will be seen. The value of **mouse.angle** by default is **0**, which implies that this graphic won't be seen rotated, unless a new angle is assigned to this field.

Keep in mind that the angles must be specified in degree thousandths. For instance, the **mouse.angle=90000;** statement will make the pointer appear rotated **90 degrees**.

---

size - Size of the graphic in percentage. By default, this field will be equal to **100** (the graphic will be seen 100%). Then, it is not necessary to indicate another value here, unless the aim is to scale the graphic (to display it expanded or reduced).

If, for instance, the aim was to double the original size of the graphic (being displayed at 200%), the **mouse.size=200;** statement should be used.

---

flags - In this field, different values will be indicated when the aim is to mirror the graphic of the mouse (that is to say, horizontally or vertically inverted), or to display it as a (semi) transparent graphic. The possible values that can be assigned to the **mouse.flags** are the following ones:

0-Normal graphic (value by default)
1-Horizontal mirror
2-Vertical mirror
3-Horizontal and vertical mirror (180°)
4-Transparent graphic
5-Transparent and horizontal mirror
6-Transparent and vertical mirror
7-Transparent, horizontal and vertical mirror

---

region - Graphic's clipping region. A value must be assigned to this field **just when the aim is to make the mouse pointer visible only inside a region** (a rectangular zone of the screen). In order to achieve it, it is necessary first to define this region with the **define_region()** function and then, to assign the number of the region that has been defined to this field (**mouse.region**).

By default, this value will be equal to **0**, that is a number of region referred to the entire screen. Therefore, the graphic will be seen on the whole screen.

---

left, middle and right - These three fields store logical values (**0** or **1**) depending on whether the mouse buttons are pressed or not (they correspond with the left, central and

right mouse buttons). Normally, only two buttons of the mouse (**left** and **right**) are activated, being ignored the state of the central button. This depends on the mouse **driver** installed in the computer.

For instance, to perform an action in a program **when the mouse left button is pressed (mouse.left)**, it is necessary to include the following statement in the code:

```
IF (mouse.left)
  // Action to perform (statements)
END
```

GLOBAL STRUCT scroll

This **10** record structure contains certain fields related to changeable parameters of the **scroll windows**. These ten records have the same field names, but each of them modifies the parameters of a different scroll window (as up to 10 windows of this type can be activated).

A **scroll window** could be defined as a screen region that only shows a part of a graphic bigger than that window (this graphic is normally the **décor** or **background** of the game). The **scroll** is the movement of that window through the graphic in any direction, being displayed the entire graphic little by little, section by section.

For a record (from **0** to **9**) of the **scroll structure** to make sense, that **scroll window** (from **0** to **9**) must first be activated with the **start_scroll()** function (for further information about the scroll windows, see this function).

It is understood that the fields of this structure are complementary to those of the call parameters of this last function.

How to use the scroll structure:

To access these fields, the field name must be preceded by the word **scroll**, the record's number in square brackets and the symbol **.** (period).

For instance, if two scroll windows, number 0 and number 1, are initialised, it can be possible to access the **camera** field of both windows as **scroll[0].camera** and **scroll[1].camera**, respectively. Moreover, when the scroll window number 0 is accessed, it is possible to omit the window's number in square brackets. That is to say, the **scroll.camera** and the **scroll[0].camera** variables are, to all intents and purposes, the same for the language.

---

x0, y0 - Coordinates of the scroll's foreground, when the scroll **ISN'T** automatic (the **camera** field has not been defined). **These are the fields that will have to be modified in order to move the scroll window's foreground.**

These two fields store the horizontal and vertical coordinates of the upper left corner of the scroll window (the point of the foreground's graphic that will be seen in the window's upper left corner).

When the **camera** field of this structure has been defined, the movement of the scroll window will be automatic; thus, they are read-only fields. In order to check where the scroll is at every moment (see the **move_scroll()** function).

---

x1, y1 - Background's coordinates, when a graphic for the background has been defined. When the scroll **ISN'T** automatic (the **camera** field has not been defined), **these are the fields to modify in order to move the background of the scroll window.**

When the **camera** field of this structure has been defined, the movement of the scroll window will be automatic; thus, they will be read-only fields, and the definition of the background's movement speed will depend on the **ratio** field of the same structure.

---

z - Scroll display priority, to indicate the depth plane in which this window must be painted, with respect to the rest of processes. By default, this variable will equal **512**, which implies that, as the processes have their **local z** variable at **0** by default, the scroll window will be painted in a greater (deeper) depth plane, being the graphics of the processes displayed above the window. In order to vary this situation, it is possible to modify either the **z** window's variable (for instance, putting it at **-1**) or the **z** processes' variable (for instance, putting it at **600**).

---

camera - It is not necessary to initialise this field, as it will be initialised when the aim is that the scroll is **automatic**, that is to say, that the system deals with it to follow a process (a game's graphic) always. For that, it is necessary to put the process' **identifying code** in this field. Thus, the shift of the scroll window will pass to be controlled automatically by the system, always trying to center the graphic of this process in the window. This process must have the **ctype** local variable with the value **c_scroll**.

By default, this field will equal **0**, which implies that the scroll won't follow any process, unless the **identifying code** of a process is assigned to **camera**. When it is done, this process will be known as the scroll's **camera process**.

Note: A series of fields are now shown **only for automatic scroll windows**. It means that for those fields to make sense (and, therefore, effect), the **camera** field of this structure has to be defined previously with the **identifying code** of the process that is going to be centered in the scroll. These values will affect the way in which the process called **scroll camera** is going to be followed.

---

ratio - **Automatic scroll windows.** When two scroll planes have been defined in the call to the **start_scroll()** function, in this field it is possible to define the **movement speed of the background** with respect to that of the foreground. By default, this value will equal **200**, which implies that the background will move half the speed of the foreground; if it is defined as **400**, it will move at the fourth part (four times slower), **100** at the same speed, **50** at double speed of the foreground, etc.

---

speed - **Automatic scroll windows.** Maximum speed of the scroll foreground, which will equal **0** by default. It means that no speed limit is imposed. If a limit is imposed, specifying the maximum number of points that the foreground can be shifted for every game's frame, the **camera process** will be uncentered in the scroll window when it is moved at a higher speed.

---

region1 - **Automatic scroll windows.** Scroll lock region, whose value by default equals **-1**, which means that there is no lock region. If this field is defined with a number of region (a rectangular zone of the screen previously defined with the **define_region()** function), then the system won't scroll as long as the **camera process** remained inside it.

---

region2 - **Automatic scroll windows.** External region of the scroll. By default, its value is equal to **-1**, which means that there is no external region. If this field is defined with a region's number and a **maximum speed** has been defined in the **speed** field, then the system will ignore that speed limit when the **camera process** is going to exceed from this region (it is done in order to continue to see the process (for its graphic to be visible always within the scroll window).

Note: If the two regions (**region1** and **region2**) are defined, region 1 is normally lesser than region 2 (the first one is contained in the second one). It will imply that:

- The background won't shift (the scroll won't be performed) while the camera process' graphic is inside region 1.
- If the maximum **speed** has been defined, then a scroll will be performed to try to restore the graphic of the camera process to region 1, but without exceeding the imposed speed limit.
- If the graphic of the camera process tried to exceed from region 2, the imposed speed limit would be ignored in order not to allow it.

GLOBAL STRUCT setup

This is a very advanced data structure, which is not at all necessary to create a game, no matter how difficult it is, as DIV Games Studio's process manager will normally take charge of the sound hardware automatically.

All the fields referred to the sound hardware are automatically updated by the program if you have a sound card, provided that the **BLASTER** or **GRAVIS** environment variable is properly initialised.

This one record structure contains a series of fields divided into two groups: the first one, to activate new parameters of the sound hardware installed in the computer, and the second one to adjust the different volume controls managed by the sound system's **mixer**.

The **reset_sound()** function must be called to activate the new parameters of the sound hardware inserted in this structure (in the **card**, **port**, **irq**, **dma** and **dma2** fields).

The **set_volume()** function must be called to activate the new volume levels inserted in the structure (in the **master**, **sound_fx** and **cd_audio** fields).

This structure is normally used inside the sound system setup programs.

Note: To access these fields, the field name must be preceded by the word **setup** and by the symbol **.** (period). For instance, **setup.master** must be used to access the **master** field (which indicates the mixer's general volume level).

---

card - Indicates the type of sound card installed in the computer. The program accepts cards of the **Sound Blaster** (tm) and **Gravis Ultra Sound** (tm) families, as well as all those **100%** compatible with them.

The values that this field can take are the following ones, depending on the sound card type:

Without card or sound = **0**
Sound Blaster 1.5 = **1**
Sound Blaster 2.0 = **2**
Sound Blaster Pro = **3**
Sound Blaster 16 = **4**
Sound Blaster AWE = **5**
Gravis Ultra Sound = **6**
Gravis Ultra Sound MAX = **7**

---

port - Indicates the computer's communications port in which the data of the sound card must be written and read.

The values that this field can take are the following ones, depending on the port assigned to the sound hardware:

0x210 = **0**
0x220 = **1**
0x230 = **2**
0x240 = **3**
0x250 = **4**
0x260 = **5**

---

irq - This field indicates the number of IRQ (Interrupt request) assigned to the active sound card.

The values that this field can take are the following ones, depending on the IRQ used by the card:

IRQ 2 = **0**
IRQ 3 = **1**
IRQ 5 = **2**
IRQ 7 = **3**
IRQ 10 = **4**
IRQ 11 = **5**
IRQ 12 = **6**

IRQ 13 = **7**
IRQ 14 = **8**
IRQ 15 = **9**

---

dma - The direct memory access (DMA) channel's number used by the sound card must be indicated in this field. This field can take values from 0 to 10, directly depending on the channel's number.

---

dma2 - Some sound cards have a second direct memory access channel faster than the previous one, of 16 bits, commonly named HDMA, DMA2 or DMA16. Like in the previous field of this structure, this second channel can take values from 0 to 10 depending on the 16 bit channel's number used by the card.

---

master - This field contains the output general or master volume of the card. A number ranging from 0 (minimum volume) and 15 (maximum volume) must be here indicated. By default, the value equals **15**, the maximum volume. Turning the master volume down will affect the sound effects' volume as well as the CD audio music reproduction's volume.

---

sound_fx - This field controls the volume to which the sound effects executed with the **sound()** functions are reproduced.

This volume is independent from that used with the sound functions. The former is general for all the sound effects. On the contrary, the latter (volume indicated in the functions) is specific for every sound.

The values of this field also range from 0 (minimum volume) and 15 (maximum volume). By default, the value will be equal to the maximum volume.

---

cd_audio - This field controls the volume of the music that will be reproduced from the audio tracks of a CD ROM or Compact Disc.

Similar to the two previous fields, the values of this field can also rage from 0 (minimum volume) and 15 (maximum volume). By default, the value will be equal to the maximum volume.

C2 – Predifined GLOBAL Tables

GLOBAL timer[ ]

This is a 10 position global table, from **timer[0]** to **timer[9]**, and each of these 10 positions is a counter of **second hundredths** that is automatically incremented.

At the beginning of the program, these 10 counters will be put at zero. They are used to time within a program. For that purpose, they can be put at zero at any time.

There are 10 counters so that the user can dedicate each of them to perform a different action inside the game, no matter which ones of the 10 counters are used. Normally, if the program only needs one counter (most of the times), that numbered **0** (**timer[0]**) is used, as the language allows us to omit the zero in square brackets in this case. That is to say, if only one counter is needed, it is possible to use **timer** simply.

For instance, to implement a process that 5 seconds after the beginning of its execution (if it had been called) performed a specific action, it would be constructed in a way similar to the following one (by using, for instance, the counter timer[9]):

Note 1: As timing is performed in hundredths of a second, these counters can be incremented in **1**, **2**, **3**, **4**, etc. in every frame of the game That is to say, the user can not wait for **timer[9]** to equal **500** exactly, as a frame could indicate **497** hundredths passed (since it was put at zero with **timer[9]=0;**) and the following frame **502** hundredths, without having passed through value **500**.

Note 2: It is also important to note that much care must be taken to prevent several processes of the program from using the same counter for different purposes.

If, for instance, a **process_example()** was created, in every frame of the game these processes would never manage to execute the action of the five seconds, as each of them would put the counter **timer[9]** at **0** at the beginning of their execution, thus invalidating the timing of the previous processes.

Bearing in mind that the counter timer[9] is **GLOBAL**, that is to say, it is the same for all the game's processes, if a process puts it at **0**, it will be put at **0** for the rest of the processes.

Note 3: Finally, much care must be taken regarding the conditions similar to **IF (timer[9]>=500) ...** , as these conditions won't only be activated **once every 5 seconds**, but they will be activated **always after the first 5 seconds**.

GLOBAL ascii

This global variable always indicates the ASCII code of the **last pressed key** in the last game's frame.

The **ascii** variable will be at **0** if no key has been pressed in the previous frame of the game.

The ASCII codes are a list of characters (letters, numbers and symbols) numbered from **0** to **255** that have been standardised. The codes less than 32 are called control characters; from 32 to 127 appears the international set of characters; and from number 128, appears the extended set of characters (according to the PC standard).

Therefore, an ASCII code is referred to the **character that has been created with the last keystroke** (or keystroke combinations, in those cases such as letters bearing a stress mark).

Important: There is another predefined global variable, called **scan_code**, which also contains the code of the last pressed key. But, unlike **ascii**, this new variable stores the **scan code** of the key. That is to say, it indicates **which key has been pressed** and not **which character has been generated** by it (like **ascii**).

There are some constants designating these **keys codes** (keyboard scan codes). The **key()** function of the language is normally used in order to verify whether a key is being pressed or not. This function receives one of these **keys codes** as a parameter, and returns **0** if the key is not pressed or **1** if it is pressed.

GLOBAL dump_type

This global variable indicates the frame dump on screen types that must be performed in every frame of the game.

The term **dump** means that the game's frames are sent to the monitor (to the video memory of the graphic card).

There are two applicable types of dump which directly correspond with two constants that can be assigned to the **dump_type** variable.

partial_dump - When indicated with the following statement, **partial dumps** will be performed:

```
dump_type=partial_dump;
```

Only the graphics that are updated, that have changed with respect to the previous frame, will be dumped on screen in this mode. It is advisable to activate this dump **in order to gain speed** when a game (or one section of it) is programmed without a scroll or mode 7 window

occupying the entire screen. That is to say, either when the game shows graphics' movements against a fixed background or when the active scroll or mode 7 windows are smaller than the screen.

complete_dump - When indicated with the following statement, **complete dumps** will be performed:

dump_type=complete_dump;

In this mode, the entire screen will be dumped no matter whether the graphics have changed or not. This mode is slower than the **partial dump**. Nevertheless, it must be used when the game has a scroll or mode 7 window occupying all the screen.

By default, the value of **dump_type** is **complete_dump**. That is to say, if no other value is indicated in this variable, **complete dumps** on the screen will be performed after each game's frame (which is normally **slower** than performing partial dump).

The dump type can be changed during a program's execution as often as necessary, according to the requirements of the stages (or sections) under execution at each moment.

Note: There is another global variable also related to DIV Games Studio's management on screen. This is called **restore_type** and it defines the type of **restoring** that must be performed on screen after every game's frame (which graphics or texts must be deleted).

GLOBAL fading

This global variable indicates if a screen fading (a gradual change of the game's palette colours) is being performed at a specific moment. Its value will be:

false (0) - If a fading **isn't** being performed.

true (1) - If a fading is being performed.

The purpose of this variable is to be able to determine the end of a screen fading started with the **fade()** or **fade_on()** functions.

On using these functions, a fading of the palette's colours will start, gradually coming closer to the definitive colours in the next frames of the game. That is to say, in every **FRAME** statement a part of the fading will be performed.

When a fade is started, the **fading** variable will automatically become equal to **true (1)** and when it is finished, it will recover its original value, **false (0)**.

Note 1: Generally, this variable is used to control the **fade()** function, and verify whether the fading has already been executed (performed). For instance, to stop the program's execution until the fading is finished, which can be done with a statement as follows (just after the call to the **fade()** function):

```
WHILE (fading)
      FRAME;
END
```

Literally this statement defines: "**while the fading continues to be performed, a new frame must be displayed**".

Note 2: All the programs perform a fade (**fade_on()**) at the beginning of their execution (automatically). Therefore, this variable will be put at **true** (1) at the beginning of all the programs until this initial fading doesn't finish (while the screen "**fading on**" is being performed).

GLOBAL joy_filter

This global variable is used to define the filter applied to the read joystick's coordinates.

It is defined as a percentage from **0 %** to **99 %**. By default, **joy_filter** will equal **10** (a 10% filter will be applied).

The purpose of applying this filter to the joystick's coordinates is to make its movements gentler and to avoid possible "**irregularities**" in the coordinates' reading. Those joystick's coordinates must be obtained with the **get_joy_position()** function. The **joy_filter** variable will only be useful when the latter function is being used.

The bigger the filter applied to the joystick is, the gentler the movements of the latter will be. But, at the same time, its answer will take longer.

Note: It can be noticed how, for small values of **joy_filter**, many "**irregularities**" appear in the reading, and for very big values (like 95%) the coordinates' reading is much gentler and regular, but slightly slower.

It is essential to have a joystick (or gamepad) connected to the computer for this variable to be useful. If the joystick is connected during the program's execution, the system won't detect it (it must be connected from the beginning).

GLOBAL joy_status

The state of the joystick (or gamepad) connected to the computer is indicated in this global variable. These are the values that this variable takes by default:

0 - If the joystick reading system is disabled. This value means that a joystick connected to the computer either has not been found at the beginning of the program's execution, or has been disconnected.

1 - If the joystick reading system is active. This is the initial value by default, but if the joystick is disconnected (or there is no joystick

connected), the reading system will be disabled (indicating **0** in the joy_status variable).

If the system is disabled, it can be reactivated by simply assigning **1** to **joy_status** (with the **joy_status=1;** statement). But if, after a limited time, no joystick is detected, the system will be disabled again.

There is a **special mode** in which the joystick reading system **won't be ever disabled.** This mode is simply defined by **assigning 2 to joy_status.**

```
joy_status=2; // Activates special mode
```

Nevertheless, much care must be taken as, if the joystick reading system is activated in this way, and there is no joystick connected to the computer, **the game's execution may be slowed down.**

Note: To read the joystick in the programs, the **global joy structure** is normally accessed. This structure always indicates its offset and the state of its buttons (whether they are pressed or not).

GLOBAL max_process_time

Programs are provided with an **anti-blocking** system that will make the manager of processes of DIV Games Studio interrupts its execution when a process exceeds the **maximum execution time** in a game's frame.

This maximum time is indicated in the **max_process_time** global variable in **hundredths of second.** By default, its value is **500 hundredths (5 seconds).**

That is to say, when a process takes longer than the indicated time in executing a **FRAME** statement (which indicates that the process is ready for the following frame of the game), an execution error will arise.

Note: The utility of the possibility of changing this variable, assigning a new value to it, is to avoid this error in the programs in which there is a process that must be doing calculations for a long time.

The following statement must be used to order the process' manager, for instance, not to interrupt a process, unless its execution in a frame is longer that 30 seconds:

```
max_process_time=3000;
```

As 30 seconds are 3000 second hundredths.

Note: Keep in mind that the time used by every computer to do the program's calculations is different. Therefore, this value must be defined with a certain margin, in order to avoid to exceed the **maximum execution time** when the game is executed in slower computers.

GLOBAL restore_type

This global variable indicates the restoring type that must be performed after each frame on screen.

The term **background restoring** means to recover the screen zones in which graphics have been painted or texts have been written in the previous frame. That is to say, **"unpaint"** the graphics and **"unwrite"** the texts (delete them).

There are three applicable restoring types which directly correspond to three constants that can be assigned to the **restore_type** variable.

no_restore - The fastest one, the background is not restored (-1)
partial_restore - Average, partial restoring (0)
complete_restore - The slowest one, complete restoring (1)

By default, the value of **restore_type** equals **complete_restore**. That is to say, if a different value is not indicated in this variable, a complete screen restoring will take place after each frame of the game.

This restoring mode (complete) is the slowest one out of these three modes. Thus, it will surely be possible to gain speed in the game's execution (for it to be faster in slow computers), if a different value is assigned to this variable. For instance, the following statement must be used to indicate a partial restoring:

```
restore_type=partial_restore;
```

This statements orders the process' manager of DIV Games Studio to partially restore the screen background (only those screen zones where graphics or texts have been put) after the following frames of the game.

The **no_restore** type (not restoring the screen background) is the fastest mode. However, it is only applicable when the game develops inside a scroll or mode 7 window occupying the entire screen. Otherwise, the graphics will leave signs (of the previous frames) on moving through the screen.

The restoring mode can be changed under a program's execution as often as necessary, according to the requirements of the stages (or sections) under execution at each moment.

Note: There is another global variable also related to DIV Games Studio's management on screen. This is called **dump_type** and it defines the type of frames **dump** that must be performed (what information must be sent to the monitor after every frame of the game).

GLOBAL scan_code

This global variable always indicates the scan code of the **last pressed key** in the last frame of the game.

The **scan_code** variable will be at **0** if no key has been pressed in the previous frame of the game.

This variable is often used to wait in a program for the user to press any key with a statement similar to the following one:

```
WHILE (scan_code == 0)
      FRAME;
END
```

This statement indicates that, while no key has been pressed in the previous frame (while **scan_code** equals 0), the frames of the game must continue to be displayed.

The **scan codes** are simply a numeric list of the PC's keys. These codes can slightly vary (in any key) regarding different keyboards, as there are keyboards of different languages, with a varied number of keys (101,102..), etc.

However, almost all the codes of the main keys remain constant. There is a predefined list of constants (synonymous for these codes) in the language that can be seen by accessing the help about **keys codes** (or keyboard scan codes). These numeric values will precisely be assigned to the **scan_code** variable when the respective keys are pressed in the program.

Important: There is another predefined global variable, called **ascii**, which also contains the code of the last pressed key. But, unlike **scan_code**, this new variable stores the **ASCII code** (character) generated by the key. That is to say, it indicates **which character has been generated by the last pressed key** and not **which key has been pressed** (like **scan_code**).

The **key()** function of the language is normally used in order to verify whether a key is being pressed or not. This function receives one of these **keys codes** as a parameter, and returns **0** if the key is not pressed or **1** if it is pressed.

GLOBAL shift_status

The state of different special keys, such as **ALT**, **CONTROL**, etc. is indicated in this predefined global variable.

Each of these keys have the following code assigned:

Right SHIFT key = **1**
Left SHIFT key = **2**
CONTROL keys = **4**
ALT and/or ALT GR keys = **8**
SCROLL LOCK key = **16**
NUM LOCK key = **32**
CAPS LOCK key = **64**
INSERT key = **128**

The **shift_status** variable will contain the **addition of all the codes of the pressed or activated keys**.

For instance, if the **ALT** key was pressed and the **CAPS LOCK** was activated, the **shift_status** variable's value would equal **72** (8+64).

In order to verify whether a key like **ALT** is pressed, it is not possible to check that **shift_status** is equal to 8, as it would imply that **ALT** is the **only** pressed or activated special key.

A correct verification would be carried out as follows:

```
IF (shift_status AND 8 == 8)
    // The [ALT] key is pressed ...
END
```

Note: The **key()** function is normally used to verify whether a key is pressed. But it is not possible to determine with this same function whether keys such as **CAPS LOCK** are activated, but only if they are pressed or not.

There are two variables containing the code of the last pressed key; **scan_code** (**scan** code of the last pressed key) and **ascii** (**ascii** code of the last pressed key).

GLOBAL text_z

The depth plane in which the texts must appear on screen is indicated in this global variable. That is to say, it indicates what must appear above the texts and what below them.

The depth planes can be any integer within the range (**min_int ... max_int**) and, the greater the number is, the deeper the text or graphic will be placed.

By default, the processes' graphics have their **local z** variable at **0** , the texts **text_z** at **-256** and the mouse pointer has **mouse.z** at **-512** by default.

That means that, by default, if these values are not modified, the texts will appear above the processes' graphics and the mouse pointer above the texts.

If, for instance, the aim was that the texts appeared above the mouse pointer (opposite to which has been established by default), two things could have been done:

a) To place the pointer's plane lower than the texts' plane (a greater number), such as, for instance: **mouse.z=-200;** (as **-200** is a number bigger than **-256**).

b) To place the texts' plane upper that the pointer's plane such as, for instance, **text_z=-600;** as **-600** is a number lesser than **-512** and, thus, a lesser depth plane (less deep).

Note 1: The **text_z** variable is **GLOBAL** for all the texts. That is to say, it is not possible to define texts in different depth planes.
Note 2: The texts can only be displayed with the **write()** (alphanumeric texts) function or with the **write_int()** (variables' numeric values) function.

LOCAL STRUCT reserved

In this structure, different **variables of internal use** (used by the **manager of processes of DIV Games Studio**) are stored.

They are local variables reserved for the system. It is not necessary to know these variables, as most of them are not useful to create programs.

Important: The modification of the values of these variables will probably provoke the **blocking** of the computer, an incorrect working of the **manager of processes** or problems on using many of the internal functions. Therefore, no responsibility is assumed for the hypothetical problems derived from an incorrect use of the **reserved** structure.

---

process_id - Identifying code of the process. This value is normally obtained with the reserved word **ID** and the value of this field must not be modified.

---

id_scan - It is internally used on detecting collisions in order to save the identifying code of the last process that has collided with the current process.

---

process_type - Type of the current process, normally obtained with the operator **TYPE**, later indicating the process name.

---

type_scan - It is internally used to detect collisions or obtain identifying codes of processes of a specific type.

---

status - Present state of the process. The values that this field can adopt are the following ones:

0 - Non-existent process.
1 - Process that has received a signal (s_kill).
2 - Alive or awake process (s_wakeup).
3 - Asleep process (s_sleep).
4 - Frozen process (s_freeze).

---

param_offset - Offset of the computer's memory in which the parameters received by the process are located.

---

program_index - Program's counter. Offset of the computer's memory in which the first statement that must execute the process in the next frame is located.

---

is_executed - It indicates whether this process has already been executed in the current frame.

---

is_painted - It indicates whether the graphic of the process has already been painted in the current frame of the game.

---

distance_1 - Vertical distance of the process (reserved for processes displayed in a mode 7 window).

---

distance_2 - Horizontal distance of the process (reserved for processes displayed in a mode 7 window).

---

frame_percent - Percentage of the following frame completed by the process. This value will be useful when the **FRAME** statement is used indicating a percentage. Otherwise, it will simply be equal to 0 (0%) when the process has not been executed and 100 (100%) when it has already been executed.

---

box_x0, box_y0 - Upper left coordinate of the graphic in the previous frame of the game (where the graphic was placed at screen coordinates).

---

box_x1, box_y1 - Lower right coordinate of the graphic in the previous frame of the game.

LOCAL angle

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **angle** variable.

The **angle** local variable defines the angle in which the graphic of the process must be seen, indicating an angle with regard to the original graphic in **degree thousandths**.

By default, the value of this variable will be equal to **0** (0 degrees) for all the processes, but when the graphic is modified, it will **rotate** to adjust to the new angle.

The angle may be defined as any integer within the range (**min_int ... max_int**).

Some examples of the angles that define certain values in the **angle** local variable are now shown (keep in mind that the angles are expressed in degree **thousandths**):

```
...
-180000  - Angle to the left
-90000   - Angle downwards
-45000   - Angle of the diagonal down/right
0        - Angle to the right
+45000   - Angle of the diagonal right/up
+90000   - Angle upwards
+180000  - Angle to the left
+270000  - Angle downwards
...
```

Important: When the aim is to rotate the graphic of a process, **it is advisable to paint it orientated to the right**, as it will be displayed like this by default (with the **angle** local variable equal to **0**).

Thus, when another angle is specified, the graphic will appear exactly orientated towards it.

For instance, a graphic that has been drawn to the right can be seen orientated upwards (to the angle of 90 degrees) by indicating the following statement:

```
angle=90000; // 90 degree thousandths (90 degrees).
```

That is to say, if a graphic was painted orientated towards another angle, (for instance, downwards), it would become orientated downwards by default, **in the angle 0**, which can provoke confusions when it comes to orientating the graphic towards another angle.

Note: To make the graphic of a process advance its coordinates **x**, **y** towards its angle (the one specified **angle** in the local variable of the process) a specific distance, the **advance()** function can be used.

The graphic of a process must be indicated assigning a **graphic code** to the **graph** local variable.

LOCAL bigbro

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **bigbro** variable.

This variable always contains the **identifying code** of the process created by the father just before creating the current process after it. That is to say, when the process that called the current one had created another one before, this variable will indicate which one is it.

Inside the language, **elder brother** is the name given to this process. For further information, see the **hierarchies of processes** in the language.

This variable will be equal to 0 if the **father** process (the one that called the current one) has not created any other process before. If it has created one, or more than one, **bigbro** will indicate the **identifying code** of the last one.

Note: The **identifying code** of the **younger brother** is indicated in the predefined **smallbro** local variable.

LOCAL cnumber

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **cnumber** variable.

The local **cnumber** variable is **exclusively** used when, in a game, several **scroll windows** or several **mode 7 windows** simultaneously appear on screen.

- For further information about the **scroll windows**, see the help about the **start_scroll()** function, which is used to activate them in the program.
- For further information about the **mode 7 windows**, see the help about the **start_mode7()** function, which is used to activate them in the program.

The **cnumber** utility lies on indicating **in which of these windows the graphic of the process must be seen**. Obviously, this variable must be defined only in processes visible inside the **scroll windows** or the **mode 7 windows**. This variable is useless for the rest of the processes (screen processes or processes with no graphics).

If the process must be seen in all the windows, then it won't be necessary to modify this variable, as the value of **cnumber (0)** by default precisely indicates so.

Up to 10 windows of both types may be activated, numbered from 0 to 9. There are ten predefined constants used to define the value of **cnumber**. These are **c_0, c_1, c_2, ...,** c_9 and directly correspond with the 10 possible windows of these types.

cnumber must be assigned **the addition of the constants** corresponding with the **windows in which the process must be visible**.

For instance, if there are **4 scroll windows** numbered **0**, **1**, **2** and **3** in a program, and the aim is to define that a specific process must be only visible in windows **0** and **2**, the following statement must be used:

```
cnumber=c_0+c_2;
```

The value of **cnumber** can be changed during the process execution if necessary.

Note: Keep in mind that for the graphic of the process to be seen in all the windows, it is not necessary to do anything, as it is the option by default.

LOCAL ctype

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **ctype** variable.

The **system of coordinates** used by the process is indicated in this variable. That is to say, it shows how the process' coordinates (contained in the **x** and **y** local variables) must be interpreted.

It is possible to use three different systems of coordinates, directly corresponding with three constants that can be assigned to the **ctype** variable.

c_screen - Screen coordinates
c_scroll - Scroll coordinates
c_m7 - Mode 7 coordinates

By default, the ctype value is c_screen, used for the process' graphic coordinates to be interpreted as referred to the screen, where the upper left corner is (0, 0).

With the following statement, **c_scroll** will be assigned to **ctype**:

```
ctype=c_scroll;
```

For the process' graphic coordinates to be interpreted as referred to a scroll window, with coordinates located above the foreground's graphic.

With the following statement, **c_m7** will be assigned to **ctype**:

```
ctype=c_m7;
```

For the process graphic coordinates to be interpreted as referred to a mode 7 window, with coordinates located above the main graphic, three-dimensionally folded in that window.

Note: There is another local variable that also affects the way in which the process coordinates must be interpreted. This variable is **resolution**, which establishes the resolution (scale) in which the coordinates are defined.

LOCAL father

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **father** variable.

This variable always contains the **identifying code** of the process that created (called) the current process (the one that has this variable). That is to say, it indicates which process called it.

Inside the language, **father** process is the name given to the process that calls another one. The process that has been called receives the name of **son** process. For further information, see the **hierarchies of processes** in the language.

The DIV's **manager of processes** is the process named **div_main**. Its function is to create the main process of the program (**PROGRAM**) at the beginning of the game's execution. Therefore, it will be the **father** of the main program, as well as the **father** of all the processes that become orphaned (processes whose father has been killed or finished before them).

Note: The **identifying code** of the son process is indicated in the predefined **son** local variable.

LOCAL file

This is a predefined **LOCAL** variable, which means that every process will have its own value in its **file** variable.

In the case that several graphics' **files FPG** have been loaded in a program, the **file** local variable indicates which file contains the graphic that the process is using.

The graphic of a process must be indicated by assigning a **graphic's code** to the **graph** local variable.

If just one file has been loaded in the program, it won't be necessary to assign any value to **file**, as the **code of the first loaded file** will equal **0** and this is the value of the variable by default.

If the graphic has been loaded with the **load_map()** function, it won't be necessary to assign any value to **file** either, as the graphics loaded with this function are used as if they belonged to file number **0** (to the first one that is loaded in the program).

When more than a file is loaded, it is necessary to indicate in each process in which one its graphic is stored. It is done by assigning the **file code** returned by the **load_fpg()** function (on loading this **file FPG**) to the **file** local variable.

Note: Normally, if several files are sequentially loaded in a program, the first one will have code **0**, the second, code **1**, the third, code **2** and so on.

In general, if several files are used, it is a good practice to have the same number of global variables (named, for instance, **file1**, **file2**, ...) containing the code of each of the files, to use them in the processes when it comes to defining its **file** variable (the **file FPG** that must be used).

The variables would be defined inside the section **GLOBAL** in the following way:

```
GLOBAL
    file1; // First file's code
    file2; // Second file's code
    ...
```

Next, these variables would be assigned the **file codes** on loading them with the **load_fpg()** function in the following way (supposing that the names of the files is **name1.fpg**, **name2.fpg**, etc.):

```
file1=load_fpg("name1.fpg"); // Files loading
file2=load_fpg("name2.fpg");
 ...
```

These files are generally loaded at the beginning of the program. Later, the used file would only have to be defined inside each process with the following statement (supposing that the process uses graphics stored in the file **name1.fpg**):

```
file=file1 // The first file is used
```

Note: Keep in mind that defining the **file** local variable is futile, unless a **graphic's code** is assigned to the **graph** local variable.

LOCAL flags

This is a predefined **LOCAL** variable, which means that every process will have its own value in its **flags** variable.

The **flags** local variable indicates the mirrors and transparencies of the displayed graphic in the processes. The possible values are the following ones:

0-Normal graphic.
1-Horizontal mirror.
2-Vertical mirror.
3-Horizontal and vertical mirror (180°).
4-Transparent graphic.
5-Transparent and horizontal mirror.
6-Transparent and vertical mirror.
7-Transparent, horizontal and vertical mirror.

By default, the value of the **flags** variable is **0**. That is to say, if it is not modified, the graphic will be displayed opaque (not transparent or mirror).

The terms **mirror** and **transparency** are now defined:

- **Horizontal mirror**, the graphic will be horizontally flipped. That is to say, if it was facing left, it will face now right and vice versa.
- **Vertical mirror**, the graphic will be vertically flipped. That is to say, if it was facing up, it will face now down and vice versa.
- **Transparency** (or **ghost-layering**), the graphic will be displayed semitransparent. That is to say, it will be possible to see what is placed behind the graphic, as if it was a coloured window, unlike the opaque graphics normally displayed.

For instance, the following statement must be used to display a transparent graphic of a process:

```
flags=4;
```

Note: The graphic of a process must be indicated assigning a **graphic code** to the **graph** local variable.

LOCAL graph

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **graph** variable.

Normally, most of the processes correspond with a graphic object displayed on screen that will be placed at the indicated coordinates in the **x** and **y** local variables. It is necessary to define which graphic corresponds with this process by assigning a **graphic code** to the **graph** local variable.

By default, this variable will be equal to **0**, which implies that no graphic will be displayed for this process.

The graphics must first be created in **the graphic editor** of DIV Games Studio (with the option **"New..."** of the maps menu) and then, they can be saved in an **archive MAP** (containing this graphic), or in a **file FPG** together with other graphics (it is possible to create a new file with the option **"New..."** of the files menu).

That is to say, the graphics used in a program may come from an **archive MAP** (that contain just one graphic) or from a **file FPG** (that may contain many graphics).

The same graphic may be used in a program by many processes at the same time.

Archives MAP

In order to use a graphic from an **archive MAP** in the program, it must be loaded by calling the **load_map()** function, which will return the **graphic code** that must be assigned to the **graph** variable.

The **graphic codes** returned by this function are simply integers from **1000**.

A **GLOBAL** variable is normally used to save this **graphic code** and then, it is assigned to the **graph** variable.

Files FPG

In order to include a **graphic** that has been done in the **graphic editor** in a **file FPG**, it is necessary to **drag the graphic window to the file window** (click on the graphic, move to the file and release). Then, the program will ask for the **graphic code**, so an integer ranging from **1** and **999** must be included here.

Thus, to use the graphic in a program, the **file FPG** that contains it must first be loaded with the **load_fpg()** function, assigning then the **graphic code** to the **graph** variable.

It won't be necessary if only one file is loaded, as the **file** variable equals **0** by default in all the processes and **0** will always be **the first file's code** loaded in the program.

Note: There are more local variables related to the graphic of a process. The most important ones are mentioned below:

Graph - Graphic code.
File - File code.
X, Y - Graphic coordinates.
Z - Depth plane.
Angle - Graphic angle.
Size - Graphic size.
Flags - Mirrors and transparencies.
Region - Display window.

LOCAL height

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **height** variable.

The local **height** variable is **exclusively** used in the processes that belong to **mode 7 windows**. That is to say, processes that have their coordinates' system inside a three-dimensional window (its local variable **ctype=c_m7**).

It is used to define the height at which the graphics of the processes must be placed above the three-dimensional plane. The **local z** variable is not used for this purpose, as it is used to define the depth plane of the graphics (even if it is now useful only for processes placed at the same coordinates).

The height of the process can be defined as any integer within the **(min_int ... max_int)** range, even if positive numbers are normally used, as the **height** of the bottom is **0** and processes are placed above it.

By default, the value of the **height** variable is **0** for all the processes, which means that if another value is not specified, the graphics of the processes will appear just above the bottom of the **mode 7** (above the plane three-dimensionally folded).

The **graphic's base** will first be placed in the indicated **height** of the process, unless **control point** number 0 is defined. In this case, this point will be placed in that height.

Note: For further information about the **mode 7 windows** and how to place graphics inside these windows, see the help about the **start_mode7()** function, which is used to activate them in the program.

This variable can be used for any other purpose in the non **mode 7** processes, as the system will completely ignore it.

LOCAL priority

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **priority** variable.

In the preparation of each frame, all the processes will be executed in the priority order established by the **priority** local variable.

The higher the value of **priority** in a process is, the sooner it will be processed in each frame. The priority value may be established as any integer within the **(min_int ... max_int)** range. For instance, to establish the priority level of a process at **10**, the following statement must be used:

```
priority=10;
```

All the processes active in the program having **the same level of priority** will be executed in a **undetermined order** that, moreover, may vary from some executions of the game to others.

By default, the **priority** local variable will be initialised at **0** in all the processes created in the program. Thus, it will be possible to execute them in any order, if the value of this variable is not defined.

If the **priority** of a single process is fixed at a positive number, such as **1**, it will be executed before the rest of the of the processes. On the other hand, if it is fixed at a negative number, such as **-1**, then it will be executed after the rest (supposing that the **priority** variable of the rest has not been modified, so its value is still equal to **0**).

When the processes priority must be established?
When a process needs to use data of another process for its calculations, it is normally advisable to execute it after the latter, defining its lowest priority for the data of the second process to be updated when they are read.

For instance, if process B must place its graphic 8 pixels lower than the graphic of process A, the priority of A must be greater than that of B, for the latter to be executed first.

Thus, when process B obtains its y coordinate by adding 8 to the one of process A, this calculation is done with the y coordinate of process A already updated for the following frame (to ensure that in each frame, the y coordinate of process A first, and then that of process B will be established).

For that purpose, suffice would be to define either the priority of A as 1 or the priority of B as -1, since by default both priorities are at 0.

Note: The priority level of the process has nothing to do with the depth plane in which its graphic appears on screen, as this plane is indicated in the **local z** variable. That is to say, the fact that a process is processed before does not mean that the graphic is painted before.

LOCAL region

This is a predefined **LOCAL** variable, which means that each process will have it own value in its **region** variable.

The **region** local variable defines the zone of the screen in which the graphic of the process must be visible, indicating the **number of region**.

A region is a rectangular zone of the screen, such as a window, associated to a number.

By default, this variable will be equal to **0** in all the processes, making reference to **region number 0** that is the **entire screen**.

That is to say, by default the graphics of the process will be visible in the whole screen (at any point of the screen in which they are placed).

At the beginning, only **region number 0** is defined. To define new screen regions, it is necessary to use the **define_region()** function.

For instance, for the graphic of a process to be visible only inside a 100 by 100 pixel box placed in the upper left corner of the screen (at the coordinates 0, 0), first the new region should be defined in the following way, supposing that region number 1 is defined:

```
define_region(1,0,0,100,100);
```

and then, the number of region (1) should be assigned to the **region** local variable of the process with the following statement:

```
region=1;
```

The regions may be redefined at any moment inside a program. That is to say, they can change their position or size if necessary.

Note: The graphic of a process must be indicated assigning a **graphic code** to the **graph** local variable.

LOCAL resolution

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **resolution** variable.

Normally, the coordinates of a process (indicated in the **x** and **y** local variables) are defined in screen pixels.

The **resolution** local variable must be used when the aim is to define the coordinates in **units smaller** than the pixel.

That is to say, this variable indicates the precision of the process' coordinates.

By default, the variable will equal **0** and the coordinates will be specified in pixels.

The greater the value of **resolution** is, the smaller (and more accurate) the unit in which the coordinates are interpreted will be. Some examples are show below:

resolution=1; - The coordinates are specified in pixels (similar to resolution=0, which is the value by default).
resolution=10; - They are specified in tenths of pixel.
resolution=100; - They are specified in hundredths of pixels.
resolution=2; - They are specified in half pixel.
...

For instance, a process located at **160, 100** with **resolution** equal to **0** (or **1**), will be in the same position as a process located at **1600, 1000** and with **resolution** equal to **10**.

The value of **resolution** is normally defined as a **positive integer multiple of 10** (10, 100, 1000, ...).

In short, when the value of **resolution** is defined, the processes' manager of DIV Games Studio **will divide** the coordinates of the processes between **resolution** when it comes to painting their graphics on screen.

Important: Much care must be taken when, in a program, there are several processes with different resolutions of coordinates, as some functions, such as **get_dist()** (used to obtain the distance between two processes), will return **incorrect results** when two processes using different resolution of coordinates are accessed.

It is normally advisable that all the processes active in the game, at least all that interact (that are detected, modified or that can be collide) use the same resolution.

LOCAL size

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **size** variable.

The **size** local variable defines the size in which the graphic of the process must be seen. This size is a percentage related to its original size.

By default, the value of this variable will be equal to **100** (100%) for all the processes, and when the graphic is modified, it will **scale** (reducing or expanding its size) to adjust to the new size.

That is to say, to **double** the size of the graphic displayed, it will be necessary to specify **200%**. The following statement will be used for this purpose:

```
size=200;
```

Therefore, if this value is lesser than **100**, the graphic will be seen smaller; otherwise, it will be seen bigger.

At first, there is no limit for the graphic size, but if the **size** local variable is put at **0** (0%), then the graphic of the process won't be seen.

Note: The graphic of a process must be indicated assigning a **graphic code** to the **graph** local variable.

LOCAL smallbro

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **smallbro** variable.

This variable always contains the **identifying code** of the following process created by the father of the current process after it. That is to say, when the process that called the current one calls later another one, this variable will indicate which one is called now.

Inside the language, **younger brother** is the name given to this process. For further information, see the **hierarchies of processes** in the language.

By default, this variable will be equal to **0** until the **father** process makes a call to another process. At this moment, the new process (the younger brother of this one) will be created, indicating its **identifying code** in **smallbro**.

Note: The **identifying code** of the **elder brother** is indicated in the predefined **bigbro** local variable.

LOCAL son

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **son** variable.

This variable always contains the **identifying code** of the **last** process created (called) by the current process. That is to say, it indicates which is the **last** process called.

Inside the language, **father** process is the name given to the process that calls another one. On the other hand, **son** process is the name given to the process that has been called. For further information, see the **hierarchies of processes** in the language.

By default, this variable will be equal to **0** until the process makes a call to another process. At this moment, the new process will be created indicating its **identifying code** in **son**.

Note: The **identifying code** of the **father** process is indicated in the predefined **father** local variable.

LOCAL xgraph

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **xgraph** variable.

This is an advanced level variable. Thus, its use requires certain experience.

The **xgraph** local variable (**extended graphic**) allows us to use **multiple graphics**. To define the graphic of a process as a **graphics set** among which it is necessary to see **the most appropriate with the angle's process** (specified in the **angle** local variable).

That is to say, if the **xgraph** variable is defined, the **graph local variable** which normally defines the graphic of the process **will be ignored** and one graphic or another will be used depending on the **angle** variable.

Therefore, on changing the process' angle **the graphic of the process won't appear rotated**, but it will use this angle to select the process' graphic (inside the defined set).

By default the **xgraph** variable will equal **0** in all the processes, which indicates that they are not going to use **multiple graphics**.

The utility of the multiple graphics lies on the possibility of creating games in **perspective**, where the change of an angle in the process doesn't implied a rotation of its graphic, but the replacement of the graphic by another one painted in a different **perspective** (painted with another angle inside that perspective).

How to use the multiple graphics.

1 - First, the different pictures that are going to represent the process' graphic have to be painted with different angles in perspective. The latter will be a finite number of graphic's views, such as **4**, **8**, **12**, etc., (or any other integer bigger than **1**).

Take into account that if 4 views are defined, a different view will be defined every 90 degrees, if 8 views are defined, every 45 degrees, etc.

2 - It is necessary to put these graphics in order according to their angles. First, the graphic corresponding with angle 0 (towards the right) and then, the rest in a clockwise direction.

3 - A table, generally **GLOBAL**, must be created and initialised with the following values:

```
Number of graphic's views,
Graphic's code for angle 0 (first view),
Code of the following angle (second view),
 ...
```

The name given to this table makes no difference. For instance, if a multiple graphic is defined with 4 views, which must be the graphics with the codes 10, 11, 12, and 13, the definition of the table could be as follows:

```
GLOBAL
      table_graphic1[ ]=4,10,11,12,13;
      ....
```

4 - Finally, the offset of this table must be assigned inside the computer's memory to the **xgraph** local variable of the process, which is done with the following statement (inside the process in question):

```
xgraph=OFFSET table_graphic1;
```

The **OFFSET** operator is used to obtain the offset of a program's datum in the memory.

Once the **multiple graphic** has been defined, in each frame of the game the system will use the graphic corresponding with the angle **closest to the process' angle** (the one indicated in its **angle** variable).

The **xgraph** variable must be put at **0** again in order to disable the **multiple graphic** system in a process.

Important: If any graphic's code is put with a **negative sign** inside the table that defines the set of graphics, then this graphic will appear **horizontally flipped**. That is to say, if the graphic was facing right, it will appear facing left, and vice versa.

Note: The multiple graphic system is normally used in **mode 7 windows**, as in the folded three-dimensional plane the graphics must be seen in a different way, according to the angle from which they are observed.

For further information about this technique, see the **start_mode7()** function used to activate a **mode 7 window** in the program.

LOCAL x, LOCAL y

These are predefined local variables, which means that every process will have its own value in its **x** and **y** variables.

These local variables of the processes define where their graphic (defined in the **graph** local variable) must be placed.

The **x** local variable defines the process' **horizontal coordinate**, which may be defined as an integer within the range (**min_int** ... **max_int**), putting the positive coordinates to the right and the negative ones, to the left.

The **y** local variable defines the process' **vertical coordinate**, which may be defined as an integer within the range (**min_int** ... **max_int**), placing the positive coordinates downwards and the negative ones, upwards.

By default, these coordinates will be specified in **pixels**, referred to screen coordinates, where the upper left corner is the point placed at (**0, 0**).

Type of coordinates.

There are several systems of coordinates that may be used by the processes and that are defined with the **ctype** local variable. The coordinates related to the screen are the system by default.

Resolution of the coordinates.

The **resolution** local variable indicates the precision of the process coordinates. By default, this variable will be equal to **0** and the (**x, y**) coordinates will be specified in pixels.

The higher the value of **resolution** is, the smaller (and more precise) the unit in which the coordinates are interpreted will be. Some examples are now shown:

resolution=1; - The coordinates are specified in pixels.
resolution=10; - They are specified in tens of pixels.
resolution=100; - They are specified in hundreds of pixels.
resolution=2; - They are specified in half pixel.
...

Note: A different type and resolution of coordinates may be either defined for each process or changed while executing if necessary.

Important: When a graphic is placed at some specific coordinates, it is the graphic **center** that will normally be placed at these coordinates.

This can be changed by defining in the **graphic editor control point** number **0** of the graphic of the process (whose **graphic code** is indicated in the **graph** variable).

If the control point has been defined, it will be placed at the specified coordinates.

For instance, if **control point number 0** is placed in the upper left corner of the graphic, and then, the graphic is put at the (100, 100) coordinates, the upper left corner of the graphic will be placed at these coordinates.

LOCAL z

This is a predefined **LOCAL** variable, which means that each process will have its own value in its **z** variable.

The **z** local variable defines the depth plane in which the process graphic must be placed on screen (the graphic is defined in the **graph** local variable). That is to say, it defines what must appear above the process' graphic and what under it.

Any integer within the range (**min_int** ... **max_int**) may be used as a depth plane. The greater the number is, the deeper the graphic will be placed.

By default, the depth planes are arranged in the following way:

(+) Greater depth

+512 - Scroll windows (see **scroll[ ].z**)
+256 - Mode 7 windows (see **m7[ ].z**)
0 - Graphics of the processes (local **z**)
-256 - Texts (see **text_z**)
-512 - Mouse pointer (see **mouse.z**)

(-) Less depth

That is to say, the **z** local variable that defines the depth plane of the processes' graphics will be initialised at **0**. The processes' graphics will be placed below the mouse pointer and texts, and above the scroll and mode 7 windows (if the values are not modified by default).

All the objects (texts, graphics, windows, ...) placed in the same depth plane will appear on screen (on being superposed) in an **undetermined order**, that may vary from some program's executions to some others.

If the aim was, for instance, that the graphic of a process appeared above all the objects of the program, a depth plane could be fixed for it above the rest (as -1000), with the following statement:

```
z=-1000;
```

At the beginning, all the processes have their **z** variable at **0**, then the graphic of the processes will appear in any order if the plane in which each of them must be placed is not defined.

The depth plane of a process may be modified (by assigning a new value to its **z** variable) as often as necessary inside a program.

The depth planes of the rest of objects (windows, texts and mouse pointer) may also be modified at any stage of the program.

Note 1: The processes that belong to a **scroll window** (having its variable **ctype=c_scroll**) will be painted in the **depth plane of the scroll window**. Nevertheless, inside that window, all the graphics of the processes will appear **in order, according to their depth plane**.

That is to say, the process' depth plane (indicated as usual in the **z** variable) will be **referred to the scroll window** in which the process appears (see **start_scroll()**).

Note 2: The processes that belong to a **mode 7 window** (having its variable **ctype=c_m7**) will appear in that window **in order, according to the depth order in the three-dimensional plane** ignoring the value of their **z** local variable.

The only sense of the **z** local variable in **mode 7** processes is to define the order in which the processes **exactly** placed at the same coordinates of the folded plane must be superposed. That is to say, if two processes are placed in the three-dimensional plane at the same coordinates, then it will be possible to define, through the **z** variable, which one must appear above the other (see **start_mode7()**).

Keyboard codes

These constants are normally used as a parameter of the **key()** function, to indicate which key is the one that the user wishes to know whether it is pressed.

It can also be used to compare the **scan_code** global variable, that contains the code of the last key that has been pressed, with these values.

The character _ (underlining) followed by the name of the key, is normally used to designate each constant. For instance, for the **A** key, the constant referred to its code will be **_a**.

The whole list of these constants, with their respective values, is as follows (according to the standard arrangement of the keyboard):

_esc	= 1 [ESC] or escape
_f1	= 59 [F1] or function 1
_f2	= 60 [F2] or function 2
_f3	= 61 [F3] or function 3
_f4	= 62 [F4] or function 4
_f5	= 63 [F5] or function 5
_f6	= 64 [F6] or function 6
_f7	= 65 [F7] or function 7
_f8	= 66 [F8] or function 8
_f9	= 67 [F9] or function 9
_f10	= 68 [F10] or function 10
_f11	= 87 [F11] or function 11
_f12	= 88 [F12] or function 12 (DEBUGGER)
_prn_scr	= 55 [PRINT SCREEN]
_scroll_lock	= 70 [SCROLL LOCK]
_wave	= 41 [º] or [ª] key
_1	= 2 Number "1" key
_2	= 3 Number "2" key
_3	= 4 Number "3" key
_4	= 5 Number "4" key
_5	= 6 Number "5" key
_6	= 7 Number "6" key
_7	= 8 Number "7" key
_8	= 9 Number "8" key
_9	= 10 Number "9" key
_0	= 11 Number "0" key
_minus	= 12 Symbol "-" key
_plus	= 13 Symbol "+" key
_backspace	= 14 Delete (<-) key
_tab	= 15 Tabulator [TAB] key
_q	= 16 Letter "Q" key
_w	= 17 Letter "W" key
_e	= 18 Letter "E" key

_r	= 19	Letter "R" key
_t	= 20	Letter "T" key
_y	= 21	Letter "Y" key
_u	= 22	Letter "U" key
_i	= 23	Letter "I" key
_o	= 24	Letter "O" key
_p	= 25	Letter "P" key
_l_brachet	= 26	[^] or [`] key
_r_brachet	= 27	[*] or [+] key
_enter	= 28	[ENTER] (Enter or Return)
_caps_lock	= 58	[CAPS LOCK] or capitals lock
_a	= 30	Letter "A" key
_s	= 31	Letter "S" key
_d	= 32	Letter "D" key
_f	= 33	Letter "F" key
_g	= 34	Letter "G" key
_h	= 35	Letter "H" key
_j	= 36	Letter "J" key
_k	= 37	Letter "K" key
_l	= 38	Letter "L" key
_semicolon	= 39	Letter "Ñ" key
_apostrophe	= 40	[{] key
_backslash	= 43	[}] key
_l_shift	= 42	[SHIFT] or left capitals
_z	= 44	Letter "Z" key
_x	= 45	Letter "X" key
_c	= 46	Letter "C" key
_v	= 47	Letter "V" key
_b	= 48	Letter "B" key
_n	= 49	Letter "N" key
_m	= 50	Letter "M" key
_comma	= 51	[;] or [,] key
_point	= 51	[:] or [.] key
_slash	= 51	[_] or [-] key
_r_shift	= 54	[SHIFT] or right capitals
_control	= 29	[CONTROL] keys
_alt	= 56	[ALT] or [ALT GR] key
_space	= 57	[SPACE] or spacebar
_ins	= 82	[INSERT]
_home	= 71	[HOME]
_pgup	= 73	[PGUP] or page up
_del	= 83	[DEL] or delete
_end	= 79	[END]
_pgdn	= 81	[PGDN] or page down
_up	= 72	Up cursor
_down	= 80	Down cursor
_left	= 75	Left cursor
_right	= 77	Right cursor

_num_lock	= 69	[NUM LOCK] or numeric lock
_c_backslash	= 53	Symbol [/] of the numeric keyboard
_c_asterisk	= 55	Symbol [*] of the numeric keyboard
_c_minus	= 74	Symbol [-] of the numeric keyboard
_c_home	= 71	[HOME] of the numeric keyboard
_c_up	= 72	Up cursor of the numeric keyboard
_c_pgup	= 73	[PGUP] of the numeric keyboard
_c_left	= 75	Left cursor of the numeric keyboard
_c_center	= 76	[5] key of the numeric keyboard
_c_right	= 77	Right cursor of the numeric keyboard
_c_end	= 79	[END] of the numeric keyboard
_c_down	= 80	Down cursor of the numeric keyboard
_c_pgdn	= 81	[PGDN] of the numeric keyboard
_c_ins	= 82	[INS] of the numeric keyboard
_c_del	= 83	[DELETE] of the numeric keyboard
_c_plus	= 78	Symbol [+] of the numeric keyboard
_c_enter	= 28	[ENTER] of the numeric keyboard

It is indifferent to use these constants or the numeric values that they represent. That is to say, it is possible to call the **key()** function, to verify whether the **A** key is pressed, such as **key(_a)** or **key(30)** (in the previous list, it is possible to verify that **30** is the numeric value of the constant **_a**).

Videomodes - Constants: **m320x200** ... **m1024x768**

These constants are used to indicate the videomode in the **set_mode()** function.

Each constant indicates the videomode in the following way: first, the letter **m** and then, the horizontal and vertical resolution of the mode, separated by an **x**. The values defined for these constants are the following ones:

m320x200	= 320200
m320x240	= 320240
m320x400	= 320400
m360x240	= 360240
m360x360	= 360360
m376x282	= 376282
m640x400	= 640400
m640x480	= 640480
m800x600	= 800600
m1024x768	= 1024768

Numbers of window - Constants: c_0 ... c_9

These constants are used to be assigned to the predefined **cnumber** local variable that is used to define the scroll or mode 7 windows in which the graphic of a process must appear.

This will only be necessary when several scroll or mode 7 windows have been activated, and it ISN'T aimed to display the graphic of the process in all of them.

Up to 10 windows of these types may be defined, numbered from **0** to **9**, and that directly correspond with the constants **c_0**, **c_1**, **c_2** ... **c_9**.

For the graphic of a process to appear only in one of these windows, the corresponding constant must be assigned to its **cnumber** local variable. For instance, if the aim was for the graphic of a process to appear only in (scroll or mode 7) window number 3, the following statement would be included in its code:

```
cnumber=c_3;
```

If the aim for the graphic of a process is to appear in several of these windows, then the constants must be added. For instance, for a process to appear in the windows 0, 4, and 5, the following assignment will be performed:

```
cnumber=c_0+c_4+c_5;
```

For the graphic to appear in all the windows, suffice will be to assign **0** to the **cnumber** variable. It won't be necessary if this variable has not been modified, as it is its value by default.

The values equivalent to these constants correspond with the following powers of 2:

c_0	= 1 scroll / mode-7 number 0
c_1	= 2 scroll / mode-7 number 1
c_2	= 4 scroll / mode-7 number 2
c_3	= 8 scroll / mode-7 number 3
c_4	= 16 scroll / mode-7 number 4
c_5	= 32 scroll / mode-7 number 5
c_6	= 64 scroll / mode-7 number 6
c_7	= 128 scroll / mode-7 number 7
c_8	= 256 scroll / mode-7 number 8
c_9	= 512 scroll / mode-7 number 9

---

true

This constant is used to indicate **true** values, to initialise logical variables or to define logical parameters. That is to say, it must be evaluated as a condition.

Its value is **1** and, as in the language all the **odd** numbers are interpreted as **true**, this constant will be evaluated as a condition that is always complied (**true**).

---

false

This constant is used to indicate **false** values, to initialise logical variables or to define logical parameters. That is to say, it must be evaluated as a condition.

Its value is **0** and, as in the language all the **even** numbers are interpreted as **false**, this constant will be evaluated as a condition that is never complied (**false**).

---

s_kill

This constant is used as a parameter of the **signal()** function (to send signals to the processes). Its value is **0**.

This signal transmits the imperative order **kill** to the processes. It is used to eliminate processes in the program (to make certain objects of the game disappear).

That is to say, on sending a signal **s_kill** to a process, the latter will be eliminated and will not appear any longer in the following frames of the game.

The constant **s_kill_tree** is directly linked to this constant, with the proviso that, on sending this signal, the former will eliminate the indicated process and its **sons**, which are the processes created by it.

The whole list of the constants used as signals that can be sent to the different processes of a program is the following one:

---

s_wakeup

This constant is used as a parameter of the **signal()** function (to send signals to the processes). Its value is **1**.

This signal transmits the imperative order **wakeup** to the processes. It is used to restore the processes that have been made dormant (with the signal **s_sleep**), or frozen (with the signal **s_freeze**) to their normal state.

That is to say, on sending a signal **s_wakeup** to a process, the latter will be reactivated in the following frames of the game (it will be seen and processed again).

The constant **s_wakeup_tree** is directly linked to this constant, with the proviso that, on sending this signal, the former will wake up the indicated process and its **sons**, which are the processes created by it.

---

s_sleep

This constant is used as a parameter of the **signal()** function (to send signals to the processes). Its value is **2**.

This signal transmits the imperative order **sleep** to the processes. It is used to make a process dormant. An asleep process will not appear in the following frames of the game, but it won't be eliminated, as it happens with the signal **s_kill**. Indeed, this kind of process may **wake up** at any moment with a signal **s_wakeup**.

That is to say, on sending a signal **s_sleep** to a process, the latter will not appear in the following frames of the game (until it is awaken or eliminated).

The constant **s_sleep_tree** is directly linked to this constant, with the proviso that, on sending this signal, the former will make dormant the indicated process and its **sons**, which are the processes created by it.

---

s_freeze

This constant is used as a parameter of the **signal()** function (to send signals to the processes). Its value is 3.

This signal transmits the imperative order **freeze** to the processes. It is used to freeze (immobilise) a process. A frozen process will continue to appear in the following frames of the game, but it won't be processed, so it will remain immobile. This process can be **reactivated** at any moment if a signal **s_wakeup** is sent to it.

That is to say, on sending a signal **s_freeze** to a process, the latter will stop processing (stop interpreting its statements) in the following frames of the game (until it is activated or eliminated with **s_kill**).

The constant **s_freeze_tree** is directly linked to this constant, with the proviso that, on sending this signal, the indicated process as well as its **sons** (which are the processes created by it) will be frozen.

---

s_kill_tree

This constant is used as a parameter of the **signal()** function (to send signals to the processes). Its value is **100**.

This signal is used to **eliminate** a process and all the process created by it, by sending the imperative order **kill** to them. This is a version of the signal **s_kill**, which eliminates a process, but not the processes that it had created.

That is to say, the signal **s_kill_tree** will **eliminate** the process and all its descendants. Thus, none of them will appear any longer in the following frames of the game.

---

s_wakeup_tree

This constant is used as a parameter of the **signal()** function (to send signals to the processes). Its value is **101**.

This signal is used to **wake up** a process and all the processes created by it, by sending the imperative order **wakeup** to them. This is a version of the signal **s_wakeup**, which wakes a process up, but not the processes that it had created.

That is to say, the signal **s_wakeup_tree** will **wake up** the process and all its descendants. Thus, all these processes will return to their normal state in the following frames of the game.

Processes that have been **made dormant** with the signal **s_sleep_tree** or **frozen** with the signal **s_freeze_tree** can be woken up (reactivated).

---

s_sleep_tree

This constant is used as a parameter of the **signal()** function (to send signals to the processes). Its value is **102**.

This signal is used to **make** a process and all the processes created by it **dormant**, by sending the imperative order **sleep** to them. This is a version of the signal **s_sleep**, which makes a process dormant, but not the processes that it had created.

That is to say, the signal **s_sleep_tree** will make the process and all its descendants dormant. Thus, all these processes will disappear in the following frames of the game (but they won't be eliminated).

These **asleep** processes can be woken up (reactivated) with the signal **s_wakeup_tree**.

---

s_freeze_tree

This constant is used as a parameter of the **signal()** function (to send signals to the processes). Its value is **103**.

This signal is used to **freeze** (immobilise) a process and all the processes created by it, by sending the imperative order **freeze** to them. This is a version of the signal **s_freeze**, which freezes a process, but not the processes that it had created.

That is to say, the signal **s_freeze_tree** will **freeze** the process and all its descendants. Thus, all these processes will stop processing in the following frames of the game (they will remain immobile, as they won't execute their statements).

These **frozen** processes can be unfrozen (reactivated) with the signal **s_wakeup_tree**.

all_text

This constant is used as a parameter of the **delete_text()** function, to delete all the texts displayed in the program with the **write()** and **write_int()** functions.

That is to say, the following statement must be executed in order to make disappear all the texts displayed on screen:

```
delete_text(all_text);
```

The value assigned to this constant is **0**.

---

all_sound

This constant is used as parameter of the **stop_sound()** function, to stop **all** the sound effects previously activated with the **sound()** function.

That is to say, the following statement must be executed in order to stop all the sound channels, active at a specific moment:

```
stop_sound(all_sound);
```

The value assigned to this constant is **-1**.

---

g_wide

This constant is used as a parameter of the **graphic_info()** function, to ask for information about the **width** (in pixels) of a specific graphic. Its value is **0**.

---

g_height

This constant is used as a parameter of the **graphic_info()** function, to ask for information about the **height** (in pixels) of a specific graphic. Its value is **1**.

---

g_x_center

This constant is used as a parameter of the **graphic_info()** function, to ask for information about the **horizontal center** of a specific graphic. Its value is **2**.

The **horizontal center** of a graphic will be half the width (in pixels), if **control point** number **0** (graphic center) has not been defined in the painting tool.

g_y_center

This constant is used as a parameter of the **graphic_info()** function, to ask for information about the **vertical center** of a specific graphic. Its value is **3**.

The **vertical center** of a graphic will be half the height (in pixels), if the **control point** number **0** (graphic center) has not been defined in the painting tool.

---

c_screen

This constant is used to be assigned to the predefined **ctype** local variable used to define the type of coordinates that a process will have. Its value is **0**.

This is the value by default of ctype, used for the coordinates of the graphic of the process to be interpreted as if they were referred to the screen. The (0,0) coordinate is the upper left corner.

---

c_scroll

This constant is used to be assigned to the predefined **ctype** local variable used to define the type of coordinates that a process will have. Its value is **1**.

This is the value assigned to **ctype**, used for the coordinates of the graphic of the process to be interpreted as if they were referred to a scroll window, to coordinates with respect to the foreground's graphic.

For further information about the **scroll windows**, it is possible to access the **start_scroll()** function used to activate them.

---

c_m7

This constant is used to be assigned to the predefined **ctype** local variable used to define the type of coordinates that a process will have. Its value is **2**.

This is the value assigned to **ctype**, used for the coordinates of the graphic of the process to be interpreted as if they were referred to a mode 7 window, three-dimensionally folded in that window.

For further information about the **mode 7 windows**, it is possible to access the **start_mode7()** function used to activate them.

---

partial_dump

This constant is used to be assigned to the predefined **dump_type** global variable used to define the type of dump that will be performed on screen. Its value is **0**.

The following statement is used:

```
dump_type=partial_dump;
```

This statement indicates to the manager of processes of DIV Games that the following dumps must be **partial**.

Dump is the name given to the system of sending the game's frames to the monitor (to the video memory of the graphics card).

There are two types of dumps:

Partial: Only the graphics that are updated and that have varied with regard to the previous frame will be dumped on screen. It is advisable to activate this dump in order **to gain speed** when programming a game (or a section of it) without a scroll or mode 7 window occupying the whole screen. That is to say, either when the game shows graphics movements against a fixed background or when the active scroll or mode 7 windows are smaller that the screen.

Complete: All the screen will be dumped, irrespective of whether the graphics have changed or not. This is the dump by default and it is **slower that the partial dump.** However, the complete dump must be used when the game has a scroll or mode 7 window occupying the whole screen.

---

complete_dump

This constant is used to be assigned to the predefined **dump_type** global variable used to define the type of dump that will be performed on screen. Its value is **1**.

This is the **value by default** of the **dump_type** variable. To establish this value, it is necessary to use the following statement:

```
dump_type=complete_dump;
```

This statement indicates to the manager of processes of DIV Games that the following dumps must be **complete**.

---

no_restore

This constant is used to be assigned to the predefined **restore_type** global variable used to define the type of restoration that must be applied to the screen background after each game frame. Its value is **-1**.

The expression **background restoration** deals with the operation of restoring the screen areas in which graphics have been painted or texts have been written in the previous frame. That is to say, to delete both the painted graphics and the written texts.

The following statement must be used to establish this value:

restore_type=no_restore;

This statement indicates to the manager of processes of DIV Games Studio that, after the following game's frames **it is not necessary to restore the screen background.**

If the background is not restored, **speed will be gained** in the execution of the game (that will go faster in slow computers). But this mode of restoration (**no_restore**) can only be applied in games or in their sections in which **there is a scroll or mode 7 window occupying the whole screen.**

---

partial_restore

This constant is used to be assigned to the predefined **restore_type** global variable used to define the type of restoration that must be applied to the screen background after each game frame. Its value is **0.**

The following statement must be used to establish this value:

restore_type=partial_restore;

This statement indicates to the manager of processes of DIV Games Studio that, after the following game's frames **only the screen areas in which graphics have been painted or texts have been written must be restored.**

This mode of restoration (**partial_restore**) is faster than a complete restoration (option by default), but it must only be applied in games, or in their sections, in which **there ISN'T a scroll or mode 7 window occupying the whole screen.**

---

complete_restore

This constant is used to be assigned to the predefined **restore_type** global variable used to define the type of restoration that must be applied to the screen background after each game frame. Its value is **1.**

This is the value by default of the restore_type variable and, it is the **slowest** mode of the three available restoration modes. The following statement must be used to establish this value:

restore_type=complete_restore;

This statement indicates to the manager of processes of DIV Games Studio that, after the following game's frames **the screen background must completely be restored.**

This mode of restoration (**complete_restore**) is the slowest one (and it is the option by default). Therefore, it can be changed by another one in order **to gain speed** in the execution of the game (so it will go faster in slow computers).

As a matter of fact, this mode of restoration is only interesting for games (or for their sections) that **DON'T have** a scroll or mode 7 window occupying the whole screen, but that have a great number of graphics moving through the screen.

---

min_int

This constant defines the minimum value that any datum can store in this language. This value is **-2147483648**.

All the data are 32 bit **integers** with sign in this language. For that reason, only integers within the range (**-2147483648** ... **+2147483647**) may be used.

When the result of an arithmetic operation exceeds that range, the system won't report any error. In order to avoid this situation, much care must be taken.

---

max_int

This constant defines the maximum value that any datum can store in this language. This value is **2147483647**.

---

pi

This constant defines the equivalence in **degree thousandths** of the mathematical constant pi (approximately **3.14159265** radians).

Its value is **180000** degree thousandths (180 degrees), equivalent to **pi radians**.

It is normally used to define angles. For instance, **180 degrees** could be defined as **pi**, **-90 degrees** as **-pi/2**, **45 degrees** as **pi/4**, etc.

APPENDIX D

D

Appendix D: Summary Of Keyboard Commands

COMMANDS IN THE GRAPHIC ENVIRONMENT

ALT+X - To exit the graphic environment to the operating system.
ESC+Control - To exit the graphic environment to the operating system.

ALT+S - To execute a session of the MS-DOS operating system.

ESC - To cancel a dialog box.
TAB - To choose the selected control of a window or box.
Enter - To activate the selected control.

F1 - To invoke the help window.

F2 - To save the selected program.
F4 - To open a program.
F10 - To save and execute the selected program.
F11 - To compile the selected program.
F12 - To save and debug the selected program.

Control+ALT+P - To save a snapshot of the graphic environment (DIV_*.PCX)

COMMON COMMANDS IN THE GAMES

ESC+Control - To exit the game.
ALT+X - To exit the game.

Control+ALT+P - To save a snapshot of the game (SNAP*.PCX)
F12 - To invoke the program's debugger.
Pause - To stop the game momentaneously.

COMMANDS IN THE PROGRAM'S DEBUGGER

Cursors. - Shift through the listing.
Pg.Up - Previous page.
Pg.Dn. - Following page.

F5 - To see the listing of a process.
F6 - To execute the current process.

F7 - To see or edit data.
F8 - To debug statement.
F9 - To set a breakpoint.
F12 - Invoke the debugger / Advance frames.

F - To execute to the next frame.

TAB - To select button.
Enter - To activate button.
ESC - To exit the debugger.

COMMANDS IN THE PROGRAM'S EDITOR

Generic commands.

F5 - To go to the beginning of a program's process.
Control+Z - To expand the size of the selected program window.

Basic movement and edit commands.

Cursors - Basic movement of the cursor.
Home - To go to the beginning of the line.
End - To go to the end of the line.
Pg.Dn. - Following page.
Pg.Up - Previous page.
Insert - To toggle between insert and overwrite.
Delete - To delete the character under the curser.
Backspace - To delete the character previous to the cursor.
TAB - Insert a tab indent.
Shift+TAB - To remove a tab indent.
Control+Delete,Control+Y - To delete the current line.
Control+Right - Following word.
Control+Left - Previous word.
Control+Pg.Up - To go to the beginning of the program.
Control+Pg.Dn. - To go to the end of the program.
Control+Home - To go to the beginning of the page.
Control+End - To go to the end of the page.

Search and replacement commands.

ALT+F, Control+F - To search for a text.
ALT+N, F3, Control+L - To repeat search.
ALT+R, Control+R - To replace text.

Blocks commands type QEDIT.

ALT+A	- To tag the beginning or end of a permanent block.
ALT+U	- To untag the permanent block.
ALT+C	- To copy the block to the current position.
ALT+M	- To move the block to the current position.
ALT+D, ALT+G	- To delete the block.

Blocks commands type EDIT.

Shift+Movement	- To tag volatile block (Movement keys: **Cursors, Control + Right, Control + Left, Pg.Up, Pg. Dn., Home, End**).
Shift+Insert	- To paste block.
Control+Insert	- To copy block.
Shift+Delete	- To cut block.
Control+X	- To cut block.
Control+C	- To copy block.
Control+V	- To paste block.
Delete	- To delete block.

COMMANDS IN THE GRAPHIC EDITOR

Generic commands.

F1	- To invoke the help window.
ESC	- To exit the graphic editor.
Cursors,OP/QA	- Movement of the cursor.
Spacebar	- Equivalent to click with the left mouse button.
Shift+Movement	- 8 by 8 pixel movement.
Shift+Left button	- To take colour from screen.
W,S	- To choose colour within the current range.
Shift+W,S	- To choose current range.
Control+Cursors	- To choose colour and range.
Backspace	- To undo.
Shift+Delete	- To repeat action (redo).
0	- To select the transparent colour.
B	- To highlight the transparent colour.
C	- Colours window.
M	- Mask window.
Z	- To change the zoom percentage.

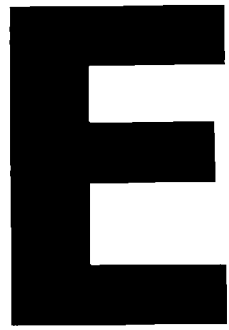
Tool selection commands.

F1	- Dotting bar.
F2	- Pen, for hand drawing.
F3	- Straight lines.
F4	- Multiline, stringed lines.
F5	- Curves bézier.
F6	- Multicurve, stringed curves.
F7	- Rectangles and boxes.
F8	- Circles and circumferences.
F9	- Paint spray.
F10	- Filling of surfaces.
F11	- Blocks edit.
F12	- To undo and redo actions.
Shift+F1	- To write texts.
Shift+F2	- To position control points.
Shift+F3	- Dotting bar

Specific commands.

Control	- To move selection (bar: to select block).
Control	- To level width and height (bars: rectangles and circles).
D	- To smooth (bars: pen, lines, curves and spray).
H	- To hide the cursor (bar: offset block).
+,-	- To vary strength (bar: multicurve).

APPENDIX E



Appendix E: Formats Of Archives

This appendix contains technical information for programmers using other languages. This information is not necessary to develop video games with DIV Games Studio.

PAL Files

Head (+0):

'p','a','l'	3 bytes (ascii).
1A0D0A00	4 bytes (hex).
Version	1 byte (0).

Subtotal: 8 bytes.

Palette (+8)

256 colour components:

Red	1 byte (0..63).
Green	1 byte (0..63).
Blue	1 byte (0..63).

Subtotal: 768 bytes.

Range of colours (+776)

16 definitions of range :

Number of colours	1 byte (8,16 o 32).
Type of range	1 byte (0:direct, 1-2-4-8 editable each "n" colours).
Fixed	1 byte (0:no, 1:yes)
Black colour	1 byte
Colours in the range	32 bytes (according to type).

Subtotal: 576 bytes.

Total : 1352 bytes.

Note : To export other formats, which lack the information about the range of colours, to DIV Games Studio files, the 576 bytes of the ranges must be defined as shown:

```
16,0,0,16 dup (0),
16,0,0,16 dup (16),
16,0,0,16 dup (32),
...
16,0,0,16 dup (240)
```

These 576 bytes cannot be defined as zeros or they would make the file not valid.

Files MAP

Head (+0)

'm','a','p'	3 bytes (ascii).
1A0D0A00	4 bytes (hex).
Version	1 byte (0).
Width	1 word.
Height	1 word.
Code of the graph	1 double word.
Description	32 bytes (asciiz).

Subtotal: 48 bytes.

Palette (+48)

(See file PAL)

Subtotal: 768 bytes.

Ranges of colours (+816)

(See file PAL)

Subtotal: 576 bytes.

Control Points (+1392)

Number of points	1 word.

Description of point x Number of points

Coordinate x	1 word.
Coordinate y	1 word.

Subtotal: 2+(4 x points number) bytes.

Graphic Map (+1394+(4 x points number))

Points of the map	Width x Height bytes.

Subtotal: Width x Height bytes.

Total : 1394+(4 x points number)+(Width x Height)

Files FPG

Head (+0)

'f','p','g'	3 bytes (ascii).
1A0D0A00	4 bytes (hex).
Version	1 byte (0).

Subtotal: 8 bytes.

Palette (+8)

(See PAL file)

Subtotal: 768 bytes.

Ranges of colours (+776)

(See PAL file)

Subtotal: 576 bytes.

Graphic Maps contained in the file (+1352)

To the end of the file, description of a graphic map.

Code of the graphic.	1 double word.
Length of the record in bytes.	1 double word.
Description	32 bytes (asciiz).
Name of file	12 bytes.
Width	1 double word.
Height	1 double word.
Points number	1 double word.

Description of point x Points number.

Coordinate x	1 word.
Coordinate y	1 word.

Graphic Map

Points of the map	Width x Height bytes.

Subtotal (per each map): 64+(4 x points number)+(Width x Height)

Total : 1352 + Records of the maps.

Files PRG

PRG Files are in the standard format of text ASCII MS-DOS.

Files FNT

Head (+0)

'f','n','t'	3 bytes (ascii).
1A0D0A00	4 bytes (hex).
Version	1 byte (0).

Subtotal: 8 bytes.

Palette (+8)

(See PAL file)

Subtotal: 768 bytes.

Ranges of colours (+776)

(See PAL file)

Subtotal: 576 bytes.

Information of the font (+1352)

Groups of characters included in the font 1 double word.

- +1 Numbers.
- +2 Capital letters.
- +4 Small letters.
- +8 Symbols.
- +16 Extended.

Subtotal: 4 bytes.

Font Table (+1356)

256 Structures of the characters, as they are described:

Width of the character	1 double word.
Height of the character	1 double word.
Vertical Slide	1 double word.
Offset of the graphic in the file	1 double word.

Subtotal: 4096 bytes.

Graphic Maps of the characters

Each character comes in the offset indicated in the file.

Map of the character Width x Height bytes.

Total : 5452 + Addition of the 256 (Width x Height)

APPENDIX F

F

Appendix F: DIV Games CD-ROM Contents

GENERAL CONTENTS

The CD-ROM is divided into three main directories: **DATA**, **INSTALL** and **GAMES.**

The **DATA** directory contains all the main archives of DIV Games Studio which will be detailed later in this appendix.

The directories **INSTALL** and **GAMES** contain the sample games of DIV separately. They allow you to play with them even if they are not installed as a tool. Both directories contain in turn 15 sub-directories more, one for each of the games.

ALIEN - Alien Suprimer
BILLIARD - Total Billiards
BLASTUP - Blast'em up
COINS - World Bottle Caps Championship
STEROID - Steroid
SPEED - Speed for Dummies
SOCCER - Soccer
CHECKOUT- Checkout
FOSTIATO- Fostiator
GALAX - Galax
PUZZLE - Puzzle 'o' matic
NOID - Noid
MALVADO- The castle of Dr. Malvado
HELIOBAL- Helioball

In the game versions included in **INSTALL** there is a program called **INSTALL.EXE** within the subdirectory of each game. This is the program you have to execute in order **to install the game independently in the computer** (without need of installing DIV Games Studio too).

However, the directory **GAMES** contains several versions of these games **already installed which can be run directly from the CD-ROM.** The games which have been executed from the CD-ROM will work all right but they won't include the sound effects. To hear these you'll have to install these games in the computer hard disk.

CONTENTS OF DATA

This is the main directory and it is divided into the following sub-directories.

DAT - Archives of data of DIV sample games.
DLL - Dynamic link libraries programmed in C for DIV.
FLI - Animations of some of the sample games.
FNT - Archives of fonts of the sample games.
FPG - Files of graphics of the sample games.
HELP - Archives of the electronic help of DIV.
IFS - Types of basic letters for the font generator.
INSTALL - Archives required to create installations.
MAP - Sample games graphics and libraries.
PAL - Several archives with generic colours palettes.
PCM - Sound of the sample games and tutorials.
SETUP - Archives required for the sound set-up.
SYSTEM - Generic archives of DIV Games Studio.

The **CD-ROM** directories you may wish to access after the program has been installed in the computer are: **IFS,** to access the letter fonts; **MAP,** to access the graphic library; and **PCM** to access the sound library.

A summary of the contents of the directories **IFS** and **MAP** are shown below so that you can localise the archives of these directories quickly.

CD-ROM GRAPHIC LIBRARIES

In the directory of **DATA\MAP\LIBRARY** in the **CD-ROM** are many maps with graphics which can be freely used in new videogames created with DIV Games Studio. This directory is divided into 12 subdirectories which correspond with the different map categories. These are:

TEXTURES - Textures for tiles, fillings, etc.
3DMAN - Human animations in different perspectives.
SPACECRAFT - Several maps with spatial graphics.
EXPLOSIO - Several types of explosion
BLOCKS - Building blocks.
FACES - Several types of faces.
CARS - Car graphics and other related to them.
DÈCOR - Dècor graphics for games.
BACKGROUNDS- Several decorating backgrounds.
GAMES - DIV Sample games graphics.
COUNTRIES - Maps of several countries.
MISCELLANEOUS- Collection of assorted graphics.

INTERNET

DIV Games Studio On The Web

DIV Arena has been set up on the Internet to support the growing community of DIV Games Studio Users.

Pay it a visit, and be part of the community: **WWW.DIV-ARENA.COM**

WHAT WILL YOU FIND THERE?

- Advice
- New Demos
- Latest News
- Forum
- Extra Textures, graphics, and sounds
- Additional code
- Tutorials
- And above all else... a thriving community!

NOTE

We kindly ask you to let us know anything you consider interesting for us, such as unofficial DIV Websites (in order to include their links on DIV ARENA), new DLL libraries for DIV (see README.TXT file of the DLL\SOURCE directory), auxiliary tools, etc.

We believe it is very important for everyone interested in this product to have the opportunity to contact each other so that good development teams can be created: one can specialise in graphics, another in programming, another in design and so on. Just an idea, but more heads are better than one.

FASTTRAK SOFTWARE PUBLISHING, 20 GREENHILL CRESCEN
WATFORD BUSINESS PARK, WATFORD, HERTS, WD1 8XU.
TEL: +44 (0)1923 495496 FAX: +44 (0)1923 228796