

Create your own Spectrum game Workshop using *The Churrera maker*

Chapter 1

Introduction

What can you do with this

How do we start

Let's Start

Chapter 2

Tileset

Collision

Interactable

Making a 16 tile tileset

Automatic shading

Examples

Making a 48 tile tileset

We already have the painted tileset. Now what?

The tileset for Mappy

The tileset to import

A bit of manual work

Chapter 3

Defining our map

Creating a project in Mappy

Exporting our map

Converting our map to code

Chapter 4

What are sprites?

Sprites in the Churrera Maker

Building our sprite-set

Side View Sprite-sets

Top View Sprite-sets

Drawing our sprite-set

Converting our sprite-set

Chapter 5

Extra Sprites

Editors Notes – Extra Sprites additional tutorial

Changing the explosion

Changing the shot

Fixed screens

Title screen

Editors Notes – Using ZX Paintbrush

The framed screen game

Combined title and frame screen

The final screen

Converting screens to Spectrum format

Compressing screens

Chapter 6

Basics: Enemies and Hot-spots

Enemies

Hot-spots

Preparing the necessary materials

Setting up our project

Basic program management

Putting enemies and platforms

Placing hot-spots

Generating the code

Chapter 7

The configuration file

General configuration

Map Size

Start position

End position

Number of objects

Initial life and recharge value

Multi-level games

Engine Type

Collision Box Size

General Directives

Types of extra enemies

Shooting engine

Scripting

Directives related to the top view

Directives related to side view

Display Settings

Graphic effects

Setting the main character's movement

Vertical axis in lateral view games

Horizontal axis in lateral view / general behavior in top view

Behavior of tiles

Preparing our compilation script

Compiling

Chapter 7B

What's New?

Destructible Tiles

Combination tile types

Running out of Bullets

Improvements to type 7 enemies

Scripting engine stuff

Corrections and optimizations

How do I upgrade?

NOTICE

Chapter 7C

Timers

Scripting

Checks

Commands

Control of pushable blocks

Check if we get off the map

Type of enemy "custom" gift

Keyboard / joystick configuration for two buttons

Masked Bullets

Chapter 8

But it's programmed

Saving values: flags

How do I activate scripting?

My first clauses

Chapter 9

Basic Scripting

Let's refresh a little

Go for it!

Counting dead monks

Logic of the boxes

Interesting Improvement

I'm a little lost

Example: Sgt. Helmet Training Day

Version Changes

Version 3.99.2

Timers

Scripting

Checks

Commands

Control of push blocks

Check if we exit the map

Type of enemy "custom" gift

Keyboard / joystick configuration for two buttons

Shooting up and diagonally for side view

Masked bullets

Version 3.99.2mod

Animated Tiles

Version 3.99.3

Animated Tiles

128K Mode

Type 3 Hotspots

Pause / Abort

Message catching objects

Version 3.99.3b

Version 3.99.3c

Item Engine

Shooting / stepping enemies disable

Create your own Spectrum game Workshop(Chapter 1)

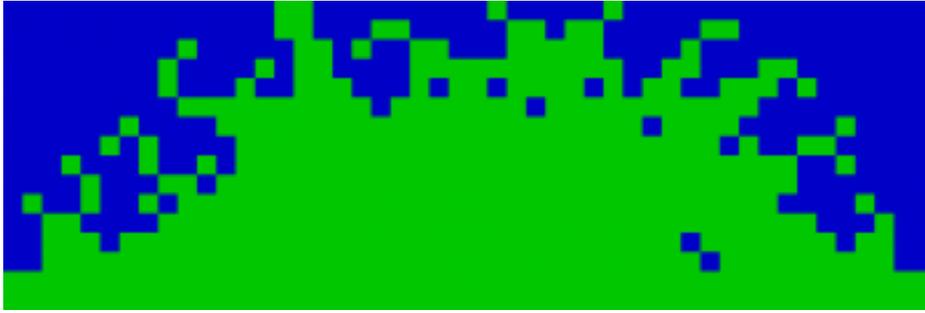


One of the most remembered features of our beloved Micro-hobby was one that allowed many to take the first steps in programming and hardware. Among other things, and almost always in the form of short workshops, we could “chop” lines and lines of code to make our own games. Then we could modify them and even learn by self-teaching. It was the closest thing to that eternal promise of “buy the Spectrum that will help me study.” Well, relatively speaking, we have always wanted in The Spectrum World a technical section and today, thanks to the Mojon Twins, starts a new section workshop where you can learn to make your own game for the ZX Spectrum. It is never too late and that is good. It is now the time that you took off the thorn stuck 20 or 30 years ago and finally you can be the author of your own ZX Spectrum masterpiece game.

Divided into chapters, the Mojon Twins tell, with their traditional surrealistic style and humor, the method to use their Churrera Maker v3.99b. You will see how easy it is when it is explained. If you have any questions you can ask what is through the comments for which a teacher Mojono, monkey or person, answer you, I'll sell you a used bike.

Without further ado, here's the first chapter of The Churrera maker v3.99b – Tutorial and other nonsense – Copyleft 2013 the Mojon Twins.

Chapter 1: Introduction



What a hedge?

That I say, but what a hedge? Phew... There is so much to say and so little space. You could throw me hours chatting and saying crap, but I'll try not to. I have been told that I have to be clear and concise, and, if it costs me, I will try to be.

Let's start at the beginning. Actually a little later, there are many arguments between creationists and evolutionists. Let's get to 2010... (This sounds typical music put images of the past, that makes taaaa, taaaa taaaa, tada...)

Earlier this year, we had a brilliant idea. We were basically tired of copying and pasting when making games. Because see, everything is not a matter of copying and pasting, but it is true that many things always do the same, changing several parameters. Let's see, if you think that we do, we write the same routine of painting the screen with tiles each time we make a game... no no. We were also tired of the hard manual work. We hand make sprites to the format of the splib2, as we hand sort the tiles so SevenuP takes them in the correct order, then we pass the map, and place enemies with a sheet of squares... There were a thousand tasks when making games that were tedious and boring. And who wants to be bored while doing something that supposedly likes? We do not. And neither do you either, I suppose.

I already know that. That designing made on graph paper is very 80's and this is seriously a pain. One may be a geek, but not masochistic.

It occurred to us that what we needed was a framework, that is what this is. A framework that allows us to have the code modules that are used easily, and that would make us smoothly around the subject of the conversion and integration of data (graphs, maps and positioning of enemies and objects...). We started timidly writing conversion utilities, to go then lifting, using pieces from here and there, an engine which will serve as a base for a game.



We had lots of ideas for the engine. We could have developed gradually and then make the final game, but did not work well. As we were going paranoid, we were frustrated every time we were getting into a new feature into the engine. Thus, as was ready "operational minimum", which premiered with *Lala the Magical*, *Cheril of the Bosque*, *Sir Ababol y Viaje al Centro de la Napia*.

As developers within the retro-scene, was that we did games “as Churreras” (but what Churreras!), we decided to call the system “**The Churrera Maker**”. And so it began...

But then what exactly is **The Churrera Maker**?

The Churrera Maker is a framework that consists of several very cool things:

1. The **engine**, or "engine", the heart of the Churrera Maker. It is a beastly mess of code that is "governed" by a main file called “config.h” in which we say which parts of the engine we will use in our game and how they will behave.
2. The **conversion utilities** that allow us to design our game in our favorite editors and so colorless, odorless and tasteless, put all that data into our game.
3. A lot of monkeys, which are essential for anything you want to do in terms.



The Churrera Maker had many versions over the last three years. Internally, we have reached version 4.7, but the mess of code rolled us so much that is not at all, presentable. It has too many hacks and very messy. So when we came to do a tutorial we decided to go back a little, to a point in the past in which the subject was still manageable: version 3.1 (*Office Space, Zombie Skull, Prologue*). But do not think that we have limited ourselves to give ourselves the "old version". No, nothing like that.

We have for a couple of months dedicated to taking version 3.1, and correcting all the things that were in a mess, change half of the components to make them faster and more compact, and add a lot of features. So when we build the 3.99b version, which is what we offer you, is more advanced, faster, and does more things than version 4.7. In fact, it has gone so well that we continue to develop from this 3.99b version, including improvements "branch 4" (*Ramiro the Vampire*) we consider interesting.

The 3.99b version is optimized so that if we recompile the old games with it, we get a binary between 2 and 5Kb smaller, with movements faster and more fluid. And here it is at your disposal. In addition, recorded on a diskette gives the necessary consistency to the magnetic material so that, launched in ninja plan, clearly encroach the heads of enemies.

What can you do with this?

Well, a lot of things. To us, they have already happened a lot. While it is true that there are common elements and certain limitations, you can often go paranoid combining different elements you have at your disposal. Do you want examples? Well, for that we have it launched the Mojon Twins Cover-tape No. 2. If you do not already have, Get it. NOW.

For Mojon Twins Cover-tape No. 2 what we did was hire a tribe of feet-dirty Indians (natives of the *Die Hard Badajoz*). For each, we write a feature of the Churrera Maker on the back and another in the chest, and encourage them to go down the hills making a meal. When they came down and did a photo of combinations. With these combinations, we made a game. Then we called Alberto, Monkey Tuerto, Just a story invented to justify that with a convincing argument. And it works, really.

The trick is to combine these things, throw some imagination, cheat a little with graphics, and, ultimately, be a little creative. For example, if we put a weak gravity (which will make the main character falls very slowly) and we enable the ability to fly with very little acceleration, we put a blue background and your character is shaped like a diver can do as if we were underwater. You can also try extreme things, such as putting the values of vertical acceleration and gravity in the negative so that the character would be pushed up and would force to sink... Which, incidentally, have never tried and that just gave me an idea... As you talk with Alberto and an argument we invent new games.

You see? That's how it works!

How do we start?

With imagination. To me, it's not worth taking a game that is already made and just change some things. No. So you will not get anywhere. People think yes, but NO. Invent something new, start from scratch, and build gradually.

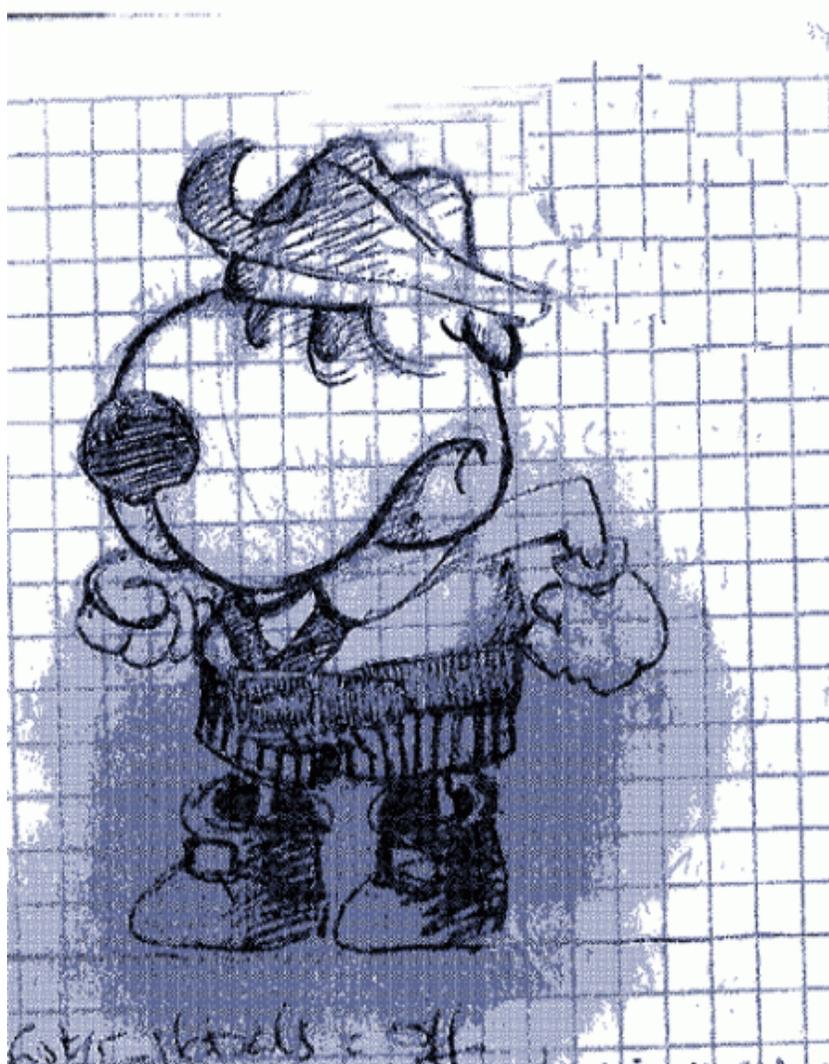
If you are not good at drawing get yourself, a friend who knows how to draw. Seriously, there is always someone. If you do not find anybody, nothing happens: You can use any graphics that have made by us. In the packages, all the source code is all .png graphics and such. Learn to cut and paste into a graphics editor.

Anyway, for beginners, we suggest that you build the *Dogmole Tupowski* game little by little. Why do this? Because it uses a lot of things, including scripting. But don't want you to go, download the Cover-tape #2 Font Pack, and limit yourself to follow the tutorial looking at files and such. What are you starting with the package of the empty engine which we are going to offer more down, and that, for each chapter, go to getting different resources and performing actions, as if really were creating the game from scratch.

Why the drama? Because when you assemble your first game, it will not be the first time and believe me, is a real advantage. And because the theatrics is cool.

Let's Start

The first thing is to make up a story that points of the game-play. We will not even write a story for the game (because now we do not need) - that will come shortly. First, we will decide what to do. Let's make a game with *Dogmole Tupowski*, a character we invented some time ago and it looks like this:



First, we will make a side perspective platform game where the character jumps. He also will jump, say that you can cover a distance of about four or five tiles horizontally and two vertically. This we must decide at this point because we have to design the map and be taken to make sure that the player can reach the sites you decide that you can get.

We will do that in the game has to carry out two missions to finish it. This can be achieved through scripting, it will be something that we will leave until the end of development. The two missions are simple and employ automatic engine characteristics so it is not too complicated to make a script:

1. There will be some kind of enemies you have to eliminate. Once eliminated, you will have access to the second mission because a stone block will have to be removed from the screen that gives access to another part of the map.
2. We will have to take objects, one by one to the part of the map that is unlocked with the first mission.

To justify this, we explain that the enemies to be eliminated are sorcerers or monks or something magical that do a power that remains closed the part of the scenario where you have to carry objects.

The story we will write one!

Therefore, we know we have to build a side view platform game with jumping. You can step on certain types of enemies and kill them. You will need to carry over an object. We will need scripting to paint the stone at the entrance of the site where objects must take if we have not killed the enemies. In addition, the fact taking things one by one to leave on a site will need a little scripting as well.

As we have a seed, we invented the story, which, if you read in your day details Cover-tape # 2, and you know:

Dogmole Tuppowski was the skipper of an old iron boat used to do some smuggling – mainly rare objects and obscure artifacts I deliver to the Certain department in the University of Miskatonic (province of Badajoz). Sadly, one night, the sea went grumpy and launched the boat to a barrier reef. In the crash, all the boxes I HAD to deliver got spread around the beach and the caves underneath the University.

Miss Meemaid from Miskatonic, was combing the hair WHO of her dolls in the full moon light sitting in her room at the top of the tower on the cliffs, witnessed the crash. Knowing of the valuable contents of the probably pit, she Decided to get them for her. As her truck was broken and would not get repaired Until the day after, in September she her minions to guard the boxes, and Besides, just in case, she ordered her twenty sorcerers to launch to haunt Which would leave the University closed by a boulder.

Dogmole's mission is twofold: First, you have to find and eliminate all 20 Sorcerers (by Means of jumping over them). That would open the University. Secondly, I have to collect 10 boxes and carry them to the University one by one. Boxes are delivered Where it reads "BOXES" by pressing "A"

With this, we can start designing our game. Actually the issue usually goes well. In our traps and sometimes we play with an advantage: many of our games have arisen because we have added a new capability to Churrera Maker and had to try it, as happened with *Bootee*, *Balowwn*, *Zombie Skull* or *Cheril the Goddess*. The creative process is unfathomable and requires that you have some inventive and imagination and that, unfortunately, is not something you can teach.

Oh, I forgot. We also did a drawing of the Meemaid. Here it is:



Let's start putting things together. First, you'll need **z88dk**, which is a C compiler and **splib2**, which is the graphics library we use. As we do not feel that you complicate installing things (especially since the splib2 is very old and it is difficult to compile it using a modern z88dk), we have prepared, for Windows users, this package must decompress directly on the root of C: and containing 1.10 and splib2 z88dk. If it works out well, you should see a folder C: z88dk10 with stuff inside.

<http://www.mojontwins.com/churrera/mt-z88dk10.zip>

Linux users and other systems should have no problem installing the latest version of z88dk in their systems and copy the file splib2.lib and splib2t.lib where clibs and spritepack.h are includes. I have left these two files for them here:

copy splib2.lib and splib2t.lib to **C:\z88dk10\lib\clibs**
copy spritepack.h to **C:\z88dk10\include**

<http://www.mojontwins.com/churrera/mt-splib2.zip>

We will also need a text editor. If you are a programmer, you will already have one that you like the host. If you are not, please do not use the Windows Notebook. Get **Crimson Editor**, for example. Actually, anyone is better than the Windows notebook. If you have Linux you will already have at least seventeen text editors installed that are better than the Windows notebook: there you have an advantage with Linux.

We will need the editor **Mappy** for maps of the game. You can download the official version, although it is better to use the Mojon version which is modified with the things we need and a pair of custom features that come in handy and we'll see.

<http://www.mojontwins.com/churrera/mt-mappy.zip>

Another thing you need to have is **SevenuP** to convert graphics. Download it from their official website, here:

<http://metalbrain.speccy.org/>

When we get to the part of the sound of talk and **BeepFX Beepola** utilities. Look for them and download them now if you want, but we will not use them until the end.

Another thing you will need is a good graphics editor to paint the monkeys and small pieces of scenery. You'd better save in **PNG** format. I reiterate that if you cannot draw and do not have a friend who knows graphics you can capture the graphics from Mojon Twins. Also, you'll need a graphic editor to cut and paste our graphics in yours. You can use anything. I use a super old version of Photoshop because it's what I'm used to. Many people use **Gimp**. There is a lot, choose the one you like. But remember to save in **.png**.

Once it is installed and such, you will need the **Churrera modules**. The following address is the full package version 3.99b:

<http://www.mojontwins.com/churrera/mt-churrera-3.99b.zip>

To get started, we extract the package Churrera Maker and we customize for our game by following these steps:

1. We changed the name to the home directory Churrera Maker3.99b by our game. For example "*Dogmole Tuppowski*".
2. We changed the name to the main C module for our game. This module is in **/dev/** and is called **churromain.c** . We will change the name to **dogmole.c**.
3. Make.bat edit the file on **/dev/** with your text editor to suit our game. First you have to replace where it says% 1 and put the name that you put in **churromain.c** . In our case, *Dogmole*. It should look like:

```
@echo off
rem cd ..\script
rem msc dogmole.spt msc.h 24
rem copy *.h ..\dev
rem cd ..\dev
cd ..\map
..\utils\mapcnv mapa.map 8 3 15 10 15 packed
copy mapa.h ..\dev
cd ..\dev
zcc +zx -vn dogmole.c -o dogmole.bin -lndos -lsplib2 -zorg=25000
..\utils\bas2tap -a10 -sLOADER loader.bas loader.tap
..\utils\bin2tap -o screen.tap -a 16384 loading.bin
..\utils\bin2tap -o main.tap -a 25000 dogmole.bin
copy /b loader.tap + screen.tap + main.tap dogmole.tap
del loader.tap
del screen.tap
del main.tap
del dogmole.bin
echo DONE
```

Every time we make a new game we will have to unpack a new copy of the Churreras and also customize it.

We're ready to go... But you have to wait for the next chapter.

Create your own Spectrum game Workshop (Chapter 2)

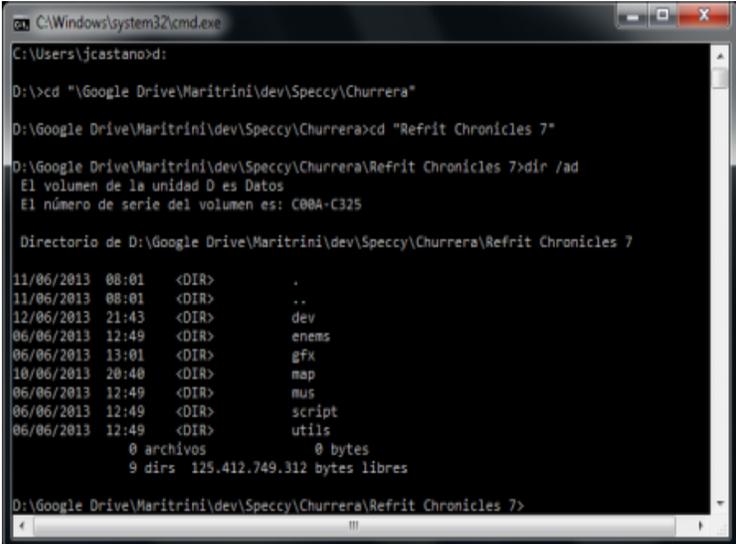
Continue your own Spectrum game creation workshop with the framework of the Mojon Twins. In this second chapter, we will get into trouble and start to see the process of creating the game. Do not be scared, it seems complicated, but it is no problem if you dedicate some time and attention. In addition, it is explained with humor and in a pleasant way, it is always more bearable.

Let's not waste any more time. Let's go there.

Chapter 2: Tile set

Before we start.

In this chapter and in practically all the others we will have to open a window of command line to execute scripts and programs, to launch the compilation of the game and things like that. What I mean is that you should have some basic notion of these maneuvers. If you do not know what this is that I put down here, it is better to consult some basic tutorial on the command line window (or console) operating system you use.



```
C:\Windows\system32\cmd.exe
C:\Users\jcastano>
D:\>cd "%Google Drive\Maritrini\dev\Speccy\Churrera"
D:\Google Drive\Maritrini\dev\Speccy\Churrera>cd "Refrit Chronicles 7"
D:\Google Drive\Maritrini\dev\Speccy\Churrera\Refrit Chronicles 7>dir /ad
El volumen de la unidad D es Datos
El número de serie del volumen es: C08A-C325

Directorio de D:\Google Drive\Maritrini\dev\Speccy\Churrera\Refrit Chronicles 7

11/06/2013  08:01  <DIR>      .
11/06/2013  08:01  <DIR>      ..
12/06/2013  21:43  <DIR>      dev
06/06/2013  12:49  <DIR>      enems
06/06/2013  13:01  <DIR>      gfx
10/06/2013  20:40  <DIR>      map
06/06/2013  12:49  <DIR>      mus
06/06/2013  12:49  <DIR>      script
06/06/2013  12:49  <DIR>      utils
                0 archivos          0 bytes
                9 dirs 125.412.749.312 bytes libres

D:\Google Drive\Maritrini\dev\Speccy\Churrera\Refrit Chronicles 7>
```

Material

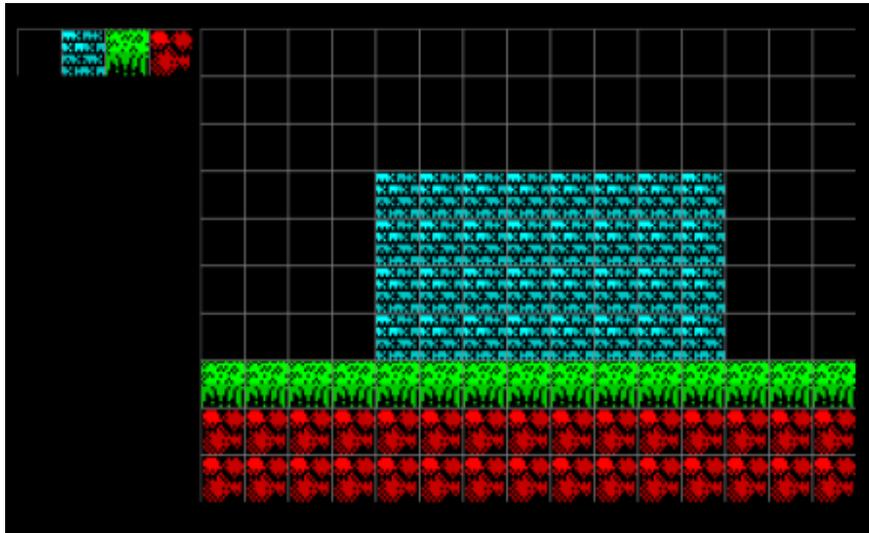
The material needed to follow this chapter I have left here:

<http://www.mojontwins.com/churrera/churreratut-capitulo2.zip>

Download it and put it in a temporary folder, which we'll now put things in our project as we are going to need them. Inside there are beautiful things.

Tileset ... What are we talking about?

As for tiles, what is a tile? Well, to put it simply is not more than a bit of graphic that is the same size and shape as other small pieces of graphics. So you can see it, looking for the translation: tile means “tile”. Now think about the wall of your bathroom, and imagine that on each tile there is a little graphic. We have the tile with a little brick, the tile with a little grass, and the black tile and the tile with a little soil. With several of each, we can order them so that we make a drawing that looks like a country house. A bathroom like this would irritate the host, by the way.



This is what The Churrera Maker uses to paint the background graphics. How to save a full graphic screen occupies an egg, what I do is to keep a certain number of small pieces and then a list of what parts occupy each screen. The collection of small pieces of a screen is what is known as “tileset”. In this chapter, we will explain how Churrera maker tilesets are, how they are created, how they are converted, how they are imported and how they are used. But first, we need to understand several concepts. Go prepare a drink.

Collision

The Churrera maker also uses the tiles for something else: for the collision. Collision is a very cool name referring to something very silly: the protagonist of the game can walk by the display or not depending on the type of tile you are going to step on. That is, that each tile is associated with behavior. For example, to the black of the example above tile, we could put “traversable” behavior so that the player could move freely through the space occupied by these tiles. On the other hand, the grass tile should be “obstacle”, meaning that it must prevent that the protagonist moves through space that they occupy. A game of platforms, for example, the engine will fall the protagonist whenever there is a tile “obstacles” under their feet.

The Churrera maker games we have the following types of tiles, or, rather, the following behaviors for the tiles. Each one also has a code that we will need to know. Now, no, but later, when we already have everything and we are riding the game. For now, it is enough with the list:

Type “0”, Traversable. In platform games this can be the sky, some bricks for the background, the picture of Uncle Narcissus, an ugly vase or some mountains. In top view games, we will use them for the floor where we can walk. In other words, objects that do not stop the main character sprite.

Type “1”, Traversable and dangerous: may be transferred, but if you touch life is subtracted to the protagonist. For example, some spikes, a pit of lava, radioactive squid, broken glass, or the mushrooms in the forest of Cheril (do you remember? It could not be touched!).

Type “2”, Traversable but hidden. They are the *Zombie Skulls*. If the character is standing behind these tiles, it is supposed to be “hidden”. The effects of being hidden are very interesting because they only affect bats in *Zombie Skull*, but hey, there it is, and we mentioned.

Type “4” platform. They only make sense in the platform games, obviously. These tiles only stop the protagonist from above, that is, if you are down you can jump through them, but if you fall from above you will pose on top. I do not know how to explain it to you... As in Sonic and that. For example, if you paint a column that occupies three tiles (head, body, and foot), you can put the body of type “0” and the head of type “4”, and so you can upload one of the columns. It is also good to use this type for thin platforms that do not stick to being obstacles at all, like the typical metal platforms that come out in many of our games.

Type “8”, obstacle. This stops the character from all sides. The walls. The rocks. Soil. All that is a type 8. Do not let the character pass.

Type “10”, intractable. It is an obstacle, but that is of type “10” makes the engine know that it is special. Of this type are, for now, the locks and blocks that can be pushed. We'll talk about them shortly.

You might think, there are missing numbers! And the more that were missing before. This is done because it simplifies the calculations a lot and allows putting more types in the future. For example, notice how anything greater than or equal to 4 will stop the protagonist from above, or that anything less than 7 will let pass the character horizontally. You see? Programming cheats.

In the future, however, it can be easily expanded, as we have said. For example, the code could be added to the Churrera Maker so that the tiles of type “5” and “6” were like conveyors to the left and to the right, respectively. It could be added. Might. Maybe, in the end, we do a chapter of changes and put new code in the Churrera Maker... Why not?

Interactable

We have mentioned the intractable tiles. In the current version of the Churrera maker are two: locks and pushable. You have to decide if your game needs these features.

Lock you will need them if you put the keys in the game. If the character collides with a lock and holds a key, then open it. The lock will disappear and you can continue.

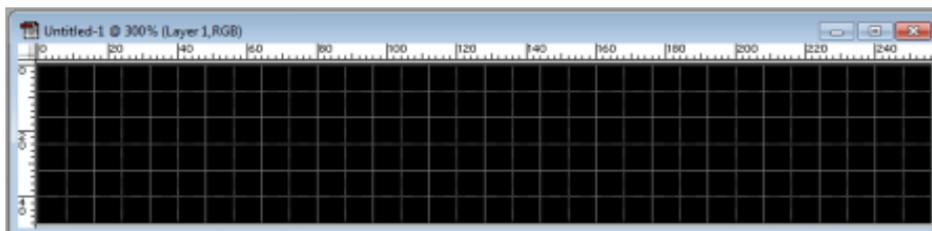
The pushable tiles are a few tiles that push them with the main character, will change its position if there is room. In the platforms games you can only can push laterally; in the top view, they can be pushed in any direction. As we shall see in due course, we can define a few things related to these tiles held, e.g. If we want enemies, not to transfer them (and so you can use them to confine them and “remove them on the way” as in *Cheril of the forest* or *Monono*, for example).

We were going to the mess already, isn't it?

We are going to draw our tileset, or scavenge it, or ask a friend who knows to draw. Yes, man, that you seek one, there are many graphic designers out there that love to draw. The first thing we have to decide is if we are going to use a tileset of 16 different tiles or 48, which are two sizes of tilesets withstanding the Churrera maker. What nonsense, you're thinking, of 48! They are more! Of course, there is more, my dear Einstein, but one thing: 16 different tiles can be represented by a 4-bit number. That means that in one byte, which is 8 bits, we can store two tiles. Where I want to go? Well, you guessed it! Maps occupy exactly half of memory if we use tilesets of 16 tiles instead of 48 tilesets.

I know that 16 may seem few tiles, but think that the majority of our games are made as well, and they are not ugly. With a little ingenuity can make screens very cool with few tiles. In addition, as we will see later in this chapter, use tilesets of 16 tiles will allow us to activate the automatic shadow effect, which will make it appear that we have quite a more than 16 tiles. But, patience, that we have not yet gotten there.

Open your favorite graphics editing program and create a new file of 256×48 pixels. Sure that your graphic editing program has an option to turn on a grid (or grids). Place to make boxes of 16×16 pixels and, if possible, having 2 subsections, so that we can see where each character begins. This will help us to make graphics following the restrictions of the Spectrum, or know where each tile begins and ends when cutting them or drawing them. I use an old version of Photoshop and when I create a new tileset I set it up like this:



Making a 16 tile tileset

If you have decided to save memory (for example, if you plan on ending the game engine being moderately complex, with scripting and many cool things, or if you prefer that your map is very big) and use tilesets of 16 tiles, you need to create something like this:



The tileset is divided into two sections: the first, formed by the first sixteen tiles, is the section of map. It is that we use to make our map. The second, made up of the following four, is the **special section** that will be used to paint special things.

Let's start looking at the **map section**:

By Convention, 0, that is, the first tile of the tileset (real developers begin counting at 0), will be the tile's main tile, which will occupy most of the background in most screens. This is not a requirement, but it will provide us with life when we do the map for obvious reasons: to create an empty map already will be all the tiles to 0.

The tiles of 1 to 13 can be whatever you want: tiles in the background, obstacle, platforms, treachery...

Tile 14 (the next to the last one), if you have decided that you are going to activate the tiles held, will be the **pushable tile**. It has to be 14, and nothing else.

The **tile 15** (the last), if you have decided that you are going to activate the keys and locks, will be the **tile lock**. It has to be 15, and nothing else.

If you're not going to use held or keys/locks you can use the tiles 14 and 15 freely, of course.

As for the **special section**, it consists of four tiles that are, from left to right:

Tile 16 is the **recharging life**. It will appear on the map and, if the user catches it, recharge a bit of life.

The **tile 17** represents **objects**. They are the ones the player will have to pick up during the game, if we decide to activate them.

The **tile 18** represents the **keys**. If we have decided to include keys and locks, the key in this tile paint.

The **tile 19** is the **alternative background**. To give a variation to the screens, randomly, this tile will be painted from time to time instead of tile 0. For example, if your tile 0 is the sky, you can put a stencil on this tile. Or, if you are doing a top perspective game, you can put a variation of the ground.

Are you understanding? Basically you have to draw 20 tiles: 16 to make the map, and 4 to represent objects and subtract monotony from the backgrounds. Needless to say, if, for example, you are not going to use keys and locks in your game, you can save yourself painting the key on tile 18.

Here's a chart for your use

Tile 0 Leave Black	Tile 1	Tile 2	Tile 3	Tile 4	Tile 5	Tile 6	Tile 7	Tile 8	Tile 9	Tile 10	Tile 11	Tile 12	Tile 13	Tile 14 Push Tile	Tile 15 Lock Tile
Tile 16 Life Tile	Tile 17 Object Tile	Tile 18 Key Tile	Alt 19 Back ground	Tile 20	Tile 21	Tile 22	Tile 23	Tile 24	Tile 25	Tile 26	Tile 27	Tile 28	Tile 29	Tile 30	Tile 31
Tile 32	Tile 33	Tile 34	Tile 35	Tile 36	Tile 37	Tile 38	Tile 39	Tile 40	Tile 41	Tile 42	Tile 43	Tile 44	Tile 45	Tile 46	Tile 47

In the *Dogmole*, for example, there are no pushable tiles. That is why our 14 tile is a Mussel from the Cantabrian Sea which, as we all know, can not be pushed.

Automatic shading

Automatic shading can make our screens look much cooler. If activated, the Churrera maker will make obstacle tiles cast shadows over others. To achieve this, we need to define an alternate version of the map section with tiles that are not obstacle shaded, which we paste in the bottom row of our tiles file as follows:



We will have complete control, therefore, of how the shadows are projected. The result obtained you can see in many of our games. For example, in the *Cheril Perils*, which belongs to the tileset above. Notice how the tiles leave a shade on the background. Also seen in the *Lala Lah*, which is the other screen:



In *Dogmole* we will not use this because we need the space that would occupy the automatic shadows for another thing we'll see at the time.

Examples

To see it, let's take a look at some tilesets of our games, so you can see how they are designed.



Here is the tileset for *Lala lah*. As we see, the first tile is the blue background that is seen on most screens. It follows a piece of platform that is also a tile of background, and then the ridge that is a tile type "platform" (type 4). If you play the game you will see how this tile behaves, to finish understanding it. The yellow pebble that follows is an obstacle (type 8). Then there are two psychedelic

colors to decorate the background (type 0). Then another pebble (8), a brickwork background (0), a variation of the squares (0), a kite kickball (type 1), a star box (8), two tiles to make non-transferable strips and therefore of type 8), a platform (type 4), and to finish tile # 15 will be type 10, because we use locks and the locks have to be interlocking obstacles. Then we have the recharge, the object and the key, the alternative tile for the background, and the strip below that used in automatic shading. Let's see another:



This is the tileset of *D'Veel'Ng* a top perspective game. This starts with two floor tiles (type 0), followed by four obstacles (type 8) - the bones, the skull, the canine and the stone, two matador and malignant tiles (type 1), which are those ugly red skulls, Another obstacle in the form of yellow bricks (type 8), another tiled floor (type 0), another tile that kills you as a malignant mushroom (type 1), white bricks (type 8), more floor And another obstacle skull (type 8). This game has tiles that are pushed, so tile 14 is a red box of type 10. We also have keys, so tile 15 is a lock, also type 10. Then we have the typical life recharge, the Object and key, and the alternate tile for the background that is painted randomly. In the row below, we have a shaded version of the background tiles again. Notice how the tiles that kill them have left the same on the strip "shaded": this is so they always look well highlighted. Some more:



Now it's the game *Monono*. This is very simple to see: we start with the main background tile, empty at all (type 0). We continue with six obstacles (type 8). Then we have two more tiles background, to decorate the bottoms: the window to peer and the shield. Then there are three more obstacle tiles (type 8), a pinch of poison (type 1), our typical metal platform copyright Mojon Twins signature special (type 4), a box that can be pushed (tile 14, type 10) and a lock (Tile 15, type 10). Then the usual: recharge, object, key, alternative. It has no automatic shading.

Making a 48 tile tileset

As we have noticed, the maps made with 48 tiles tilesets occupy double than those made with 16 tilesets. If you still decide to go this route, here is an explanation of how to do so.

First, there is no explicit differentiation between the map and the special section: is all together. In addition, it is not possible to add automatic shading (we have no site for the alternative versions of the tiles). Especially, note that this of the tilesets of 48 tiles was an additive for *Zombie skull* have not returned to use and it is not refined. But hey, it can be used.

In the tilesets of 48 tiles, you can use the tiles from 0 to 47 to make screens except for those corresponding to the special characteristics: the tile 14 for the pushable tiles, lock 15, 16 for refills, for objects 17 and 18 for keys, if it is that you will be using. If you're not going to use, you can put your own tiles. The tilesets of 48 tiles 19 as alternative background tile, nor is used so you can put whatever you want in that space.

As the only example, we have the tileset of *Zombie skull*. If you look, in *Zombie skull* keys (or no locks), so that only are occupied as 'specials' for reloads 16 and 17 for objects. There is also no tiles held. All others are used to paint the stage:



We already have the painted tileset. Now what?

The first thing is to record it as a master copy inside the /gfx folder by calling it work.png. Now we have to prepare it for use. We will make two versions: the import in the game and which will use Mappy to make the map and the enemies and objects stand for... well, guess.

If you are following the tutorial with the *Dogmole* without own experiments, you can find the **work.png** of the *dogmole* in the bundle of files in this chapter.

The tileset for Mappy

Mappy is messed up. You need tile 0 to be empty, i.e., be all black. If loads it a tileset that does not have the first tile all black, it puts one at the beginning and moves all the other tiles. That is a huge problem because we would be doing a map with all the indexes. No, not more. As we cannot change this behavior of Mappy, we will modify our tileset. If you have used a graphic in tile 0 (for example, making the bottom of your game blue, or painting dots to simulate a lawn in top view) we will have to modify the tileset to mappy so that this tile completely black. We are going to cut the 16 tiles (if our tileset 16 tiles) and are going to leave the first one black, thus (in our case have not had to do anything because our tile 0 was eliminated, but in games, such as, *Cheril of the forest*, it was necessary to do so):



This tileset recorded it in the folder /gfx as **mappy.bmp**, more than anything else because Mappy and the stand are better understood with that format.

Remember, that is important: (read this with mother's voice, so you listen more) If your tile 0 is not completely black, you have to leave it completely black. See the examples above. In *Zombre skull*, *Lala lah* or *D'Veel' Ng* had to do this.

The tileset to import

Splib2, the library on which operates the Churrera maker, manages a charset of 256 characters for painting the assets, in addition to the sprites that go over. Therefore, the next step is to create a charset for import into the game using our tileset and font. The Churrera Maker uses 64 characters that correspond to ASCII codes 32 to 95, that is, these letters:



Using a font editor (there are many free) that paint up this graphic (which you will find inside the package in the file **source-base.png**). Whatever what you finish doing it, burn it to /gfx as fuente.png. It is very important that the letters within the chart positions are respected because if not the text and numbers come out wrong. This is the fuente.png that we use in *Dogmole*:



The first thing we'll do is reorder our tileset. What we will do will be “split” each tile into four 8 x 8 pieces. Each of these pieces corresponds to what the Spectrum fans know as **UDG** (User Defined Graphic) and, in addition, will lead an associated color attribute. It's something to get to be able to convert it to code C using **SevenuP**:



Years ago, we did this by hand and it was a real pain. In fact, the first application that we did for the Churrera maker was the order of tilesets in UDGs. The application is in the folder /util of the Churrera maker and is called **reordenator.exe**. To use it, open a command line window, we get into the folder /gfx of our project, and we write something like this:

```
.. /utils/reordenator.exe work.png udgs.png
```

****Editors note:**

*I would recommend that you save your tileset image as a **PNG** before running reordenator. Name the file as **work.png**. I found it easier just to move the tileset image to the /utils directory and run **reordenator**. Afterwards you may move the file back to the /gfx folder.*

*With the tileset in the /utils directory (in the command prompt) run
reordenator.exe work.png udgs.png
now move the file **udgs.png** back to the /gfx folder*

This will make **reordenator.exe** work, generating a new file: **udgs.png**. This will be exactly the same as that you have seen directly above.

****Editors note:**

When you run reordenator, it will chop up the tiles into 8x8 blocks which will look like 8x16 blocks.

Now that we have our source in **fuentes.png** and the udg in **udgs.png**, we return to our graphics editor, create a new file of 256×64 pixels, and paste the udg under the source, as well:

***Editors note:*

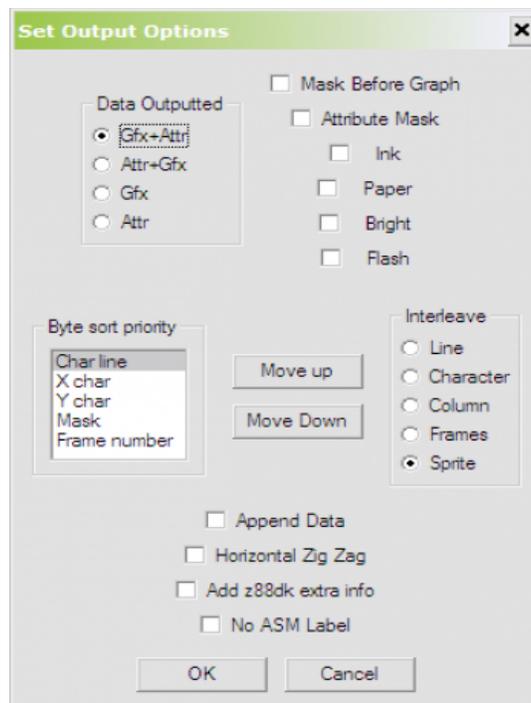
Create a new blank image of 256x64. Load the udgs.png in another tab and load the fuentes.png and yet another tab. Copy the font characters to the top of the newly created blank image and then copy your newly split tileset to the bottom of the image.



That is what we are going to import in our game. We recorded it in **/gfx** as **tileset.png**. This is very important, since **SevenuP** will generate the code from our image and the data structure called it, exactly, “tileset”, which is what the engine needs.

Perfect. With all this done, open, finally, **SevenuP**. Once opened, click “I” to “Import”, which will open a selection dialog of files where you can navigate to **/gfx** of our project to the folder and select **tileset.png**. SevenuP will import it and convert it to the Spectrum format.

Now you need to configure the export of data from the program so that it will remove the charset in the order we need. To do this we go to the menu **File -> Output Options**, which opens a dialog box with many options. There will be no touch anything except the box that puts **Byte Sort Priority**, which we will have to leave at exactly that (click on “**Char line**” and then press the button “**Move up**”).



That done, click **OK**. Now we will export the data: we give to “export data” to “**D**” and other file-selection dialog will be opened to us. We now have to navigate to your folder `/dev` and save the file as **tileset.h** (must be selected “**C**” in the type of file, because I believe that by default is ASM).

In this way, SevenuP will generate a file `/dev/tileset.h` with the code necessary to have our charset in the game. In particular, write 8 bytes for each of the 256 characters, more 256 bytes with the same attributes.

Learn that top of memory. No, seriously, you want to generate **tileset.h** for yourself. That’s why I have not put it in the package.

A bit of manual work

Do not worry, it’s not much.

Hopefully at this point you know how the Spectrum works with colour clash or the mixture of attributes. With this known, it is easy to understand the concept that, in the making of your game, sprites do not have own color, they take the background color.

The engine will paint the screen and then put the sprites over. If you've been careful, and all the tiles marked as 'Background' are yellow, the game will look quite uniform and the mixture of colors may not be noticed as much. The problem is the characters that are integers of the same color: SevenuP is pretty smart when selecting the two colors that has each character, but if there is only one color in the character it does not have too much to do about it. What ends up making doesn't alert us, above all in the case which concerns us, in the game *Dogmole*, in which the background tile 0 is all black: encodes the attributes such as **PAPER 0** and **INK 0**. With this as well, the sprites will not be, obviously. And this should fixed it by hand.

```
279 0, 0, 0, 0, 70, 6, 66, 2,  
280 66, 2, 2, 67, 66, 2, 2, 3,  
281 2, 67, 67, 3, 67, 3, 3, 65,  
282 5, 65, 65, 1, 15, 69, 5, 5,  
283 69, 6, 69, 6, 69, 6, 69, 6,  
284 67, 67, 3, 3, 70, 71, 71, 7,  
285 76, 12, 76, 12, 13, 13, 13, 13,  
286 11, 11, 11, 11, 71, 69, 69, 5,  
287 66, 66, 66, 66, 7, 70, 70, 6,  
288 69, 70, 70, 6, 71, 0, 0, 0,
```

Open `/dev/tileset.h` in a text editor. You should go to the line 279. If your text editor does not mark the line number is that you should change in text editor, by the way. When you have a text editor in condition, going to the line 279. This is the first line where defines the attributes of the characters that form our tileset. As you can see, SevenuP formats the data in such a way that there are 8 bytes per line. That means that each line of attributes are the colors of two tiles. If you look, (and if you are following the tutorial with the data of the *Dogmole*), SevenuP will have generated a line such as:

```
0, 0, 0, 0, 70, 6, 66, 2,
```

Do you see what I told you? Those four zeros are the four attributes of the first tile. Well, because we are going to change them. We will put the ink to white, or INK 7. If you remember, the attribute values are calculated with $INK + 8 * PAPER + 64 * BRIGHT$, so to put 0 PAPER, INK 7 and BRIGHT 0 need to change those zeros by Sevens.

7, 7, 7, 7, 70, 6, 66, 2,

In fact, we should aim for us if from there to the end goes another "0" that we don't want, and replace it by a the same value. If not you realize now, believe me that you will notice when play the game and see how sprites pieces disappear when passing certain sites.

Is everyone Okay at this point?

Not bad. If you have arrived here, you have passed the first test of fire. You must use the command line to run things with parameters. It should edit graphics and cut and paste. You must use conversion applications. Let's change things by hand! I know that many will be thinking "Hey, this is not as we thought it would be". Well, for very easy that they put it, to make a game it is necessary to run it. Everything has its work from behind. The first requirement that we need to make a game is to have the desire to do so, perseverance, and the purpose of closure a little.

And with the tutorial, you stay on this same channel for the next chapter, where we will build the map.

Small clarification

There have been a couple of questions on the subject of the normal tileset and Mappy tileset and such. Let's give some examples to illustrate the topic.

First, let's look at our *Dogmole Tuppowski*. To simply get the tileset of Mappy, we cut the first 16 tiles (top Strip) and will record it as **mappy.bmp**. No, to do nothing more, since the first tile is already completely black.



Here we have another example. Here we see the tileset of Cheril of the forest. To get the tileset of Mappy cut, as always, the first 16 tiles, and also will have to leave the first completely black, originally has a texture of minimalist cool grass. After that, we recorded it as mappy.bmp.



I think you already caught, but we put one more example. Here's the tileset of journey to the center of the proboscis. Here 0 tile is pink. We cut the 16 tiles and let the pink tile in black. And we recorded it as **mappy.bmp**.



Create your own Spectrum game Workshop (Chapter 3)

Chapter 3: Maps

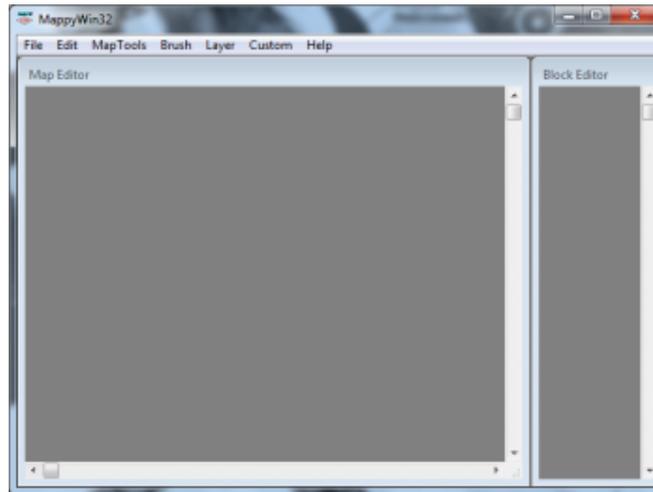
First of all, download the package of materials corresponding to this chapter by clicking on this link:

<http://www.mojontwins.com/churrera/churreratut-capitulo3.zip>

The map!

In this chapter we are going to make the map. The theme is to build each screen of the game using the tiles we drew in the previous chapter. Each screen is a kind of grid of squares, where each square has a tile. Specifically, the screens are formed with **15×10** tiles.

To build the map will use **Mappy**. Mappy is quite messy but the truth is that it works well and what is best, allowed to meddle. Mappy version you downloaded from Mojonía has a couple of additions and modifications to work as we want.



Before we start getting our hands dirty and maybe we'll explain a little about how Mappy works. The program manages a complex format to describe maps that allows for several layers, animated tiles and a lot of paranoia. This format, the native of Mappy, is called **FMP** and we'll use it to store the working copy of the map. It is this that you change in Mappy every time you want to change something. However, this format is too complex to be imported into the game, since it has a lot of things that we do not even need.

We only want a string of bytes to tell us what tile is in each box on the map. That Mappy has another format, the **map** format, which is a simple and customizable format. We have left it in the smallest expression: a string of bytes that tell us what tile is in each box on the map, precisely. It is the format in which you must save the map *when we want to generate the copy which then will process and incorporate the Game.*

In other words, the **MAP** format is used to input to the churrera, while the **FMP** format is for Mappy. Load and save to Mappy in **FMP** and export to **MAP** for the churrera.

We must be *very careful with this* as it may update the map, when we record the map, and we forget to save the **FMP**, so the next time you want to go to Mappy to make any changes will have lost the last changes. We have to Colacao, Monkey Coscao, that tells us when we forget to record the **FMP**, but we understand that you do not have with you any coscao monkey to look after you, so be sure to save the **map** and **FMP** every time that you make a change!

How is this? You'll see, but the issue is as simple as **mapa.map** and **mapa.fmp** grab or when recording the map with **File** → **Save**. Simple, but effective.

Defining our map

The first thing we have to do is decide what the size of our map will be. The map is nothing more than a **rectangle of N by M screens**. Of course, the more screens our map has in total, the more RAM it will take. Therefore, the maximum size that can have the map will depend on which characteristics we have activated in the engine for our game. If it is a simple engine game, we will fit many screens. If we have a more complex game, with many features, scripting and such, it will fit less. Each screen of the map is **15×10** tiles.

It is very difficult to give an estimate of how many screens will fit us, at most. In games of single engine with few features, such as *Sir Ababol* (which, being built using version 1.0 of the Churrera maker, only incorporates the basic features of a platform game), we fit in 45 screens and still had left about 3 Kb Of RAM, so it could have had many more screens. However, in more complex games like *Zombie*, *Cheril the Goddess* or *Skull*, we occupy almost all the RAM with much smaller maps (24 and 25 screens respectively).

Depending on how your game will work, maybe a small map is enough. For example, in the aforementioned *Cheril the Goddess* is a fair walk from side to side and cross it several times to bringing objects to the altars, so the game is not anything short.

As **Mappy** is possible to resize a map once done, it may be good idea to start with a map of moderate size and, in the end, if we see plenty of RAM and we want more screens, zoom.

In the previous chapter we talked about tilesets of 16 and 48 tiles, and we said that using one of the first screens took up half. This is because the map format used is the one we know as packed, which stores two tiles in each byte. Therefore, a map of these characteristics occupies $15 \times 10 / 2 = 75$ bytes per screen. 30 displays the Magical Lala, for example, occupy bytes 2250 (30×75). 48 tile maps can not be packaged in the same way, so each tile occupies one byte. Therefore, each screen occupies 150 bytes. The 25 screens occupy *Zombie Skull* therefore 3750 bytes. Do you see how to use less tiles?

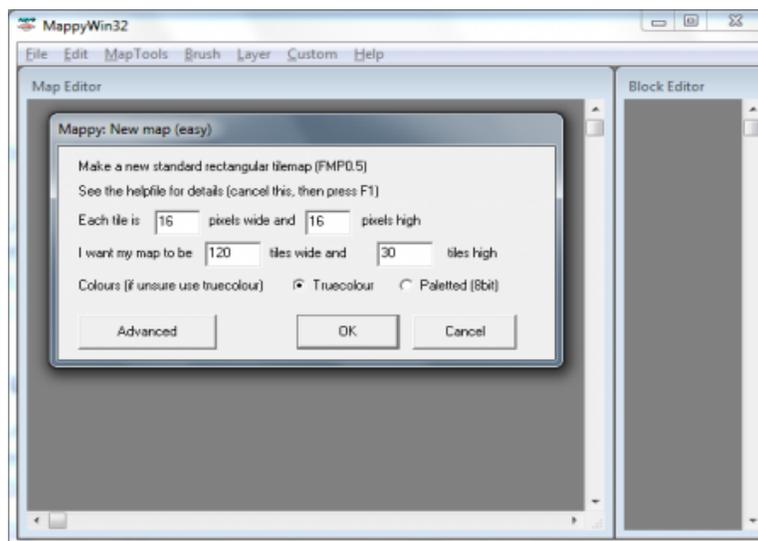
I am aware that with all this information that I have released I have clarified very little, but it is that little I can clarify to you. This is quite objective. I could have begun to study each feature occupies as we activate the Churreras, but the truth is that I never desired to do. Yes, I can tell you that the fire engine, for example, takes up a lot, especially in side view (as in *Mega Meghan*) because it needs many routines that add to the binary. Flying enemies chasing you (*Zombie Skull*) also occupy a lot. The issue of push blocks, keys, objects, tiles that kill you, bouncing off the walls, or the different types of movement (to fly (*Jet Paco*, *Cheril the Goddess*), automatic jump (*Bootee*), accumulated jump (*Monono*), Occupy less. Scripting can also take up a lot of memory if we use many different commands and checks.

You look at a number that runs between 25 and 40 screens and will fit. And if not, it is trimmed and ready. You can also wait for us to incorporate some kind of RLE compression to the maps, which is something we have in mind almost from the beginning but that, is something we have never done.

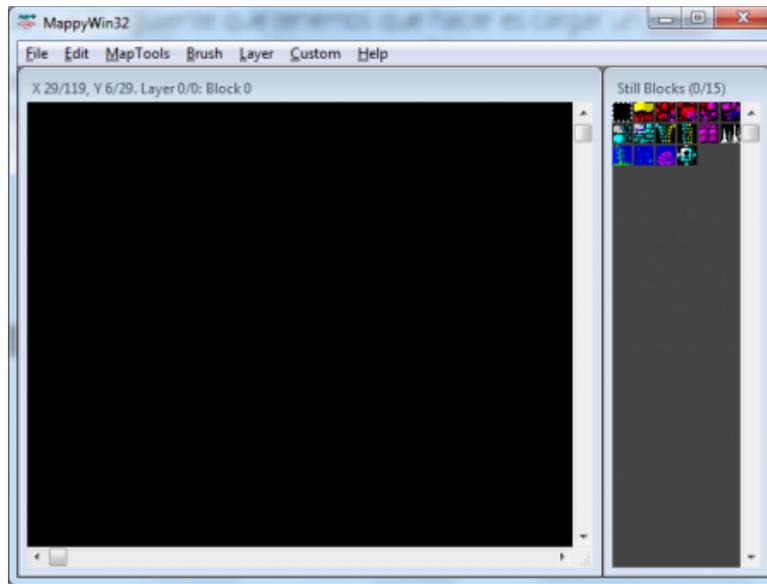
Creating a project in Mappy

The first thing to do is open Mappy, go to **File** → **New Map**, and fill the box where we will define the important values of our map: the **size of the tiles (16×16)**, and the **map size in tiles**.

In our *Dogmole Tupowski*, the map is 8×3 screens, Maria del Mar, the Mona knows Multiplying, points out to us are a total of 24 screens. As each screen, we said, measures 15×10 tiles, this means that our map will measure 8×15 = 120 tiles wide and 3×10 = 30 tiles high. That is, 120×30 tiles of 16×16. And that's what we have to fill in the box important values (called **REDEVAIM**):



When we select OK, Mappy, which is helpful, give us a message reminding us that the next thing you have to do is **load a tileset**. And that is precisely what we are going to do. We go to **File** → **Import**, which will open a dialog file selection. Navigate to the Gfx folder of your project, and select our **mappy.bmp**. If all goes well, we'll see Mappy is obedient and loads our tileset properly: we will see in the palette on the right, which is the palette of tiles:

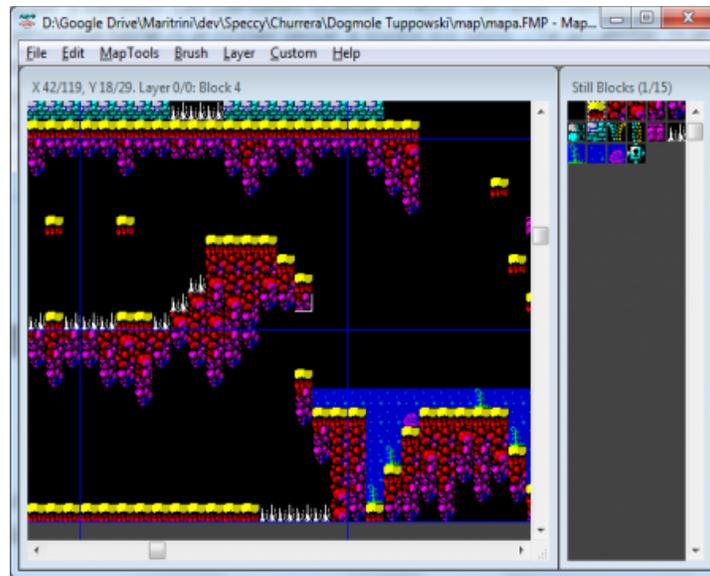


Now there is only one more thing to do before we begin: we need a help to know where each screen starts and ends, since the edges of each screen have to be consistent with those of the adjacent screens: if there is an obstacle to the right edge of A screen, there must be another obstacle to the left edge of the screen to the right. Sometimes we forget, as will be thinking that we should follow normally: the most traditional of the Mojon Twins bugs have to with the map. Do not take us as an example and look carefully at the edges.

Let's put those guides. Select **MapTools** → **Dividers** to open the dialog box guides. Check the **Enable Dividers** box and fill **Pixel gap x** and **pixel gap y** and with the values 240 and 160, respectively, which are the dimensions in pixels of our screens (that we know again thanks to Maria del Mar, which has estimated that $15 \times 16 = 240$ and $10 \times 16 = 160$). Press OK and you will see how the work-space is divided into rectangles with blue guides. Yes, you guessed it: every rectangle is a game screen. We are ready to start working!

This is where we make the map. Let's click on the tiles palette on the right to select which tile to draw, and then put the tile in the area on the left making the screens. It is laborious work, slow, and sometimes a little painful. We recommend that you do not fall into monotony and make boring screens with large areas of repeated tiles. Try to make your map organic, irregular and varied. The screens look better. You also have to keep in mind that our character has to be able to reach all the sites. Do you remember that at the beginning we decided that we would make the character jump around two high tiles and four or five wide? You have to design the map with that in mind. Another thing to be respected is that there should be no "no turning back" sites. That is very cheap and very playful 80's. Do not fall for it. Do not confuse difficulty with bad design.

So, little by little, we build our map, we will being so (you can load the package **mapa.fmp** materials in this chapter to view):

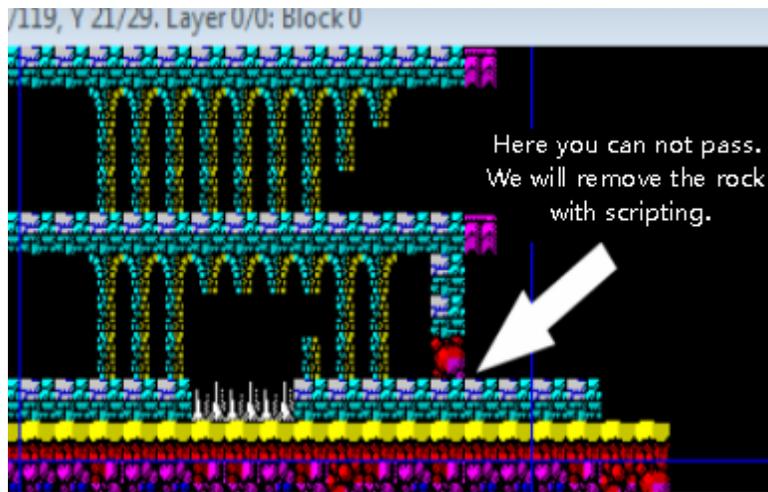


Remember to save from time to time in our directory map map as **mapa.fmp** (**File** → **Save As**). It is important to put always mapa.fmp hand or you click on mapa.fmp in the list of files to be recorded in the **fmp** format so not lose anything.

If your game has locks (tile 15) or tiles (locks 14) place them where you want them to come out when the game starts. For example, on this screen we have placed a lock. Do not worry about the keys: we will place them later, when we put the enemies and the objects on the map.



If you are, like our monkey Colacao, you've seen that there is an area of the map (in particular the upper left) that can not be accessed. This area corresponds to Miskatonic University. If you remember, when we invented the crazy idea of a game we said that the University would be blocked until we eliminated to all the sorcerers or monks, that Meemaid had put there to put a mental spell that closes it so that it does not We could carry the boxes. When we get to scripting we will see how to add code to eliminate that obstacle during the game, when it detects that the sorcerers or monks are all dead (this is something we can do with scripting: modify the map on the fly). For now, we simply put a pebble there and forget:



Another thing with which you have to be careful is with the spikes: there must be a way out of any pit with spikes. When the main character bounces with the spikes it jumps a little, but not much, so do not put them in deep pits. Notice how they are placed on our games.

Exporting our map

When we are done with our map (or not, nothing happens if we set up the game with a small piece of the map around, to go testing), it's time to export it as a type **map** and turn it to include it in our game.

We go to **File** → **Save As** and recorded as **mapa.map** on our map directory. Yes, you have to write mapa.map. When you're done, you go back to File → Save As and record it as mapa.fmp again. Writing letter to letter map.fmp. Pay attention to me. Do it, seriously. You may lose things because you forgot to record the **FMP** after a quick change is a pain. That we have more than studied this. That has happened to us a thousand times. Really. Pay attention to this monkey-faced bug:



Converting our map to code

```
..\utils mapcnav mapa.map 8 3 15 10 15 packed
```

For this we will use other utilities of the Churrera maker. In fact, the second we did: the great **mapcnav**. This utility caught Mappy map files and divides them into screens (so that the engine can build more easily, saving time and space). Also, if we are using tilesets of 16 tiles, pack tiles as explained (2 for each byte). So, once we have our **mapa.map** exported file in the directory **map**, we use a command line window, and execute **mapcnav** (which is in the directory **/utils**) with these parameters:

***Editors note:*

*I found it easier just to move the *.map file to the /utils directory and run mapcnav. Afterwards you may move the file back to the /dev folder.*

*With the *.map file in the /utils directory (in the command prompt) run*

mapcnav.exe 8 3 15 10 15 packed

now move the file back to the /dev folder

```
..\utils mapa.map mapWidth mapHeight widthTiles heightTiles lockTile packed
```

Explain them one by one:

mapa.map is the input file with the newly exported map made with Mappy.

mapWidth is the width of map screens. In our case, 8.

mapHeight is the height of the map screens. In our case, 3.

widthTiles is the width of each screen in tiles. For the Churrera maker, it is always 15.

heightTiles is the height of each screen in tiles. For the Churrera maker, it is always 10.

lockTile is the number of tile making a lock. For Churrera maker always be the tile number 15. If your game does not use locks, put a value out of range as 99. For example, we do not use locks in *Zombie Skull* here, so we put here by 99 to convert the map. We do have locks on *Dogmole*, so it will be 15.

packed sets, as is, is our tileset of 16 tiles. If we use a 48-tile tile, we simply do not put anything.

Therefore, to convert our map, we will have to execute **mapcnav** like this:

```
..\utils mapa.map 8 3 15 10 15 packed
```

With this, after a mysterious and magical process, we get a **mapa.h** file you have to move the **dev** directory of our project.

**** Editors note**

For 48 tilesets you would run

mapcnav.exe 8 3 15 10 15

```
C:\Windows\system32\cmd.exe
D:\Google Drive\Maritrini\dev\Speccy\Churrera\Dogmole Tuppowski\map>..\utils\map
cnv.exe mapa.h 8 3 15 10 15 packed
** WARNING **
MapCnv convierte un archivo raw de mappy (mapa.map, por ejemplo)
a un array directamente usable por los juegos de la churrera.
Si metes mal los parámetros ocurrirán cosas divertidas.
packed
Se escribió mapa.h con 24 pantallas empaquetadas (1800 bytes).
Se encontraron 0 cerrojos.

D:\Google Drive\Maritrini\dev\Speccy\Churrera\Dogmole Tuppowski\map>
```

If you open this **mapa.h** with the text editor, you will see a lot of numbers in an array of C: That's our map. Just below, the locks are defined in another structure. As you will see, there will be as many defined locks as we have put on the map. If this is not so, you have done something wrong. Go over all the steps!

```
33
34 #define MAX_CERROJOS 2
35
36 typedef struct {
37     unsigned char np, x, y, st;
38 } CERROJOS;
39
40 CERROJOS cerrojos [MAX_CERROJOS] = {
41     {6, 5, 1, 0},
42     {9, 7, 8, 0}
43 };
44
45
```

Perfect, all cool, all right.

Very good. We're done for today. In the next chapter we will paint sprites of the game: the main character, bad guys, platforms... We will see how to make a turn sprite-set and how to use it in our game.

In the meantime, you should practice. Something we recommend you do, if it has not yet occurred to you, is to download the source code packs from our game and take a look at the maps. **Fmp** opens with **Mappy** files and see how things are made.

Until next time!

Workshop creates your own game of Spectrum (Chapter 4)

Here comes the fourth part of the workshop by the Mojon Twins learning how to use the Churrera maker. In this chapter, you will learn to create our sprites. If you have arrived here you already have the necessary elements to advance considerably in your future game of Spectrum. What sounds good, huh? Well, let's go there, we know that you want to continue with the Workshop.

Chapter 4: Sprites

First of all, download the package of materials relevant to this chapter by clicking on this link:

<http://www.mojontwins.com/churrera/churreratut-capitulo4.zip>

What are sprites?

At this point in the story, if you don't know what is a sprite, our bad... And no, I will not do the typical joke of carbonated drink. As this chapter is about sprites, however, we will have to start explaining that they are. If you don't know, it's one of the laws of the tutorials from video games. No matter what level you are or which system you deal with: is required to explain what is a sprite is.

Let's see: the concept of sprite, really, has absolutely no meaning in systems like the Spectrum, in which the graphic output is limited to an array of pixels. Absolutely no sense. However, it is used by analogy. Let us elaborate: Traditionally, a graphics chip designed to work on a games machine handled two entities, mainly: the background and a certain (limited) number of sprites. The sprites were just handfuls of pixels, usually square or rectangular, that the graphic processor was responsible for composing on the background when sending the image to the monitor or the TV. That it: they were objects completely alien to the background and that, therefore, you could move without affecting it. The CPU of the system had only to tell the graphic processor where each sprite was and to forget since that graphic processor was the one that was in charge of constructing the image sending to the monitor sprite pixels instead of background pixels when it was necessary. It is as if the pixels were not really there, hence their name: the word "sprite" means "fairies" or "pixies."



Almost every game consoles 8 and 16 bits (the **Nintendo** and **SEGA** at least the **Atari** systems were rare noses), the MSX and Commodore 64, among other systems, have a graphics processor that really does Backgrounds and sprites.

In computers like the **Spectrum** or **Amstrad CPC**, or any PC, the concept of sprite makes no sense. Why? Because the hardware only handles an array of pixels, which would be equivalent to just handling the background. In these cases, the sprites have to paint with the CPU: you have to take care of replacing certain pixels of the background with pixels of the character. In addition, the programmer must be careful about some method to move the group of pixels: you must be able to restore the screen to the way it was before painting the sprite and redraw it in the new position. However, by analogy, these groups of pixels are called sprites.

We call them sprites, and that sometimes we are purists of the host.

Sprites in the Churrera Maker

The Churrera Maker maker handles four sprites of 16×16 pixels for the enemy (including the character who manages the player) and generally up to three sprites for projectiles (in game kill). enemy sprites are drawn using graphics that can define what is known as the **sprite-set**. Said sprite-set contains 16 graphs, of which 8 are used to encourage the main character and 8 to encourage 4 types of enemies (2 for each, if you're good mathematics). A sprite-set looks like this (this is the sprite-set the *Dogmole*):



As you will see, there is always a graphic of a puppet and next to it there is a strange thing just to you're right. Let's explain what that weird thing is before we go on. In the first place, it is not called a rare thing. It's called a mask. Yes, I know it does not look like a mask, but it's called that. Neither computer mice look like mice nor the Horace looks like a kid and I have not seen anyone complain yet. The **mask** is used to tell the graphics library (in this case, splib2), which is in charge of moving the **sprites** (remember that "this is Spectrum: no graphic chip to do these things), Which pixels in the chart are "transparent", that is, which pixels in the chart should NOT replace the background pixels.

If there were such a **mask**, all the sprites will be square or rectangular, and that is pretty ugly.

We have ordered our **sprite-set** so that each graph has a corresponding mask looks just right. If you look, the masks have black pixels in the areas where the corresponding graphic should be seen, and color pixels in the areas where the background should be seen. It is as if we define the silhouette.

Why are masks necessary? If you are a bit insightful as you may have guessed: Spectrum graphics are 1-bit depth, which means that each pixel is represented using one bit. That means that we only have two possible values for the pixels: on or off (1 or 0). In order to specify which pixels are transparent, we would need a third value, and that is not possible (that it is what the binary is!). That's why we need a separate structure to store this information, the mask!

Building our sprite-set

Before building the **sprite-set** is very important to know what kind of view will have our game: side view or top view. I suppose that, at this point, it is something that we have already decided (go, if we have already made the map). The order of graphics on the sprite-set depends on the type of view of our game.

Side View Sprite-sets

For side view games, graphics 16×16 (accompanied by their masks) that make up the sprite-set must have this order:

	Spriteset for Top View Games
0	Main character, walking right, frame 1
1	Main character, walking right, frame 2, used for standing facing right.
2	Main character, walking right, frame 3
3	Main character, walking right, jumping
4	Main character, walking left, frame 1
5	Main character, walking left, frame 2, used for standing, facing left.
6	Main character, walking left, frame 3
7	Main character, walking left, jumping
8	Type 1 enemy, frame 1
9	Type 1 enemy, frame 2
10	Type 2 enemy, frame 1
11	Type 2 enemy, frame 2
12	Type 3 enemy, frame 1
13	Type 3 enemy, frame 2
14	Moving platform, frame 1
15	Moving platform, frame 2

As we see, the first eight graphics serve to animate the main character: four for when he looks to the right, and four for when he looks to the left.

We have three basic animations for the character: standing, walking, and jumping / falling:

Standing: The first is when the character is standing (as its name suggests). Stationary means that it is not moving by itself (if it is moved by an external entity it is still "stopped"). When the character is stopped, the engine draws it using frame 2 (chart number 1 if you look to the right or 5 if you look to the left).

Walking: This is when the character moves laterally above a platform. In this case, a four – step animation is performed using frames 1, 2, 3, 2,. In that order (graphics 0, 1, 2, 1... if we look to the right or 4, 5, 6, 5... if we look to the left). When drawing, the character must have both feet on the ground for frame 2, and legs extended (with the left or right front) in frames 1 and 3. That is why we use frame 2 in the animation " stopped".

Jumping / Falling: It is when the character jumps or falls. Then the engine draws the frame "jumping" (graphic number 3 if you look to the right or number 7 if you look to the left).

The following six graphs are used to represent **enemies**. Enemies can be of three types, and each has two animation frames.

Finally, the last two graphs are used for **moving platforms**, which also have two frames of animation. Moving platforms are precisely, and as its name implies, platforms that move. The main character can climb on them to move. To draw the graphics we have to take care that the surface on which the main character must stand must touch the top edge of the graphic.

To make it clear, let's look at some examples:



The sprite-set above corresponds to the *Cheril Perils*. As we can see, the first eight graphics are the Cheril, first looking to the right and then looking to the left. Then we have three enemies that we see in the game and in the end the moving platform. Study on the subject of animation of walking, imagine move from frame 1 to 2, 2 to 3, 3 to 2, and 2-1. Look at the graph and picture it in your head. do you see it? you see how it moves the character? Ping, pong, ping pong... Also, see how the frame 2 is the best for when the puppet stands.



This another sprite-set corresponds to *Monono*. In the same way, we have 8 graphics to *Eleuterio*, three types of enemies, and in the end the moving platform. See how the top surface of the moving platform always touches the top edge of the frame of the sprite. Also look at as it is made animation walk of *Eleuterio* and as the head is lower in the frames that extend the legs.

Top View Sprite-sets

For the top view games, the 16 graphics of the **sprite-set** have to have this order:

	Spriteset for Top View Games
0	Main character, walking right, frame 1
1	Main character, walking right, frame 2
2	Main character, walking left, frame 1
3	Main character, walking left, frame 2
4	Main character, walking up, frame 1
5	Main character, walking up, frame 2
6	Main character, walking down, frame 1
7	Main character, walking down, frame 2
8	Type 1 enemy, frame 1
9	Type 1 enemy, frame 2
10	Type 2 enemy, frame 1
11	Type 2 enemy, frame 2
12	Type 3 enemy, frame 1
13	Type 3 enemy, frame 2
14	Type 4 enemy, frame 1
15	Type 4 enemy, frame 2

Again, the first eight graphics serve to animate the main character, but this time the subject is simpler: since we have four directions in which the main character can move, we only have two frames for each direction.

The remaining eight graphics correspond to four types of enemies since in a top view game there is no platform.

Again, to make it clear, let's look at some examples:



This is the sprite-set of our great conversion of the *Hobbit* (one of our great honors: be the last in a competition is almost better than being the first.). Notice how the theme changes: above we have 8 graphics with 2 frames for each direction, and below we have four types of enemies instead of three more platforms.



This one is the Mega sprite-set, Meghan. Here again, we have the main character in all four directions with two frames of animation for each strip on top and four women (who make enemy) in the bottom strip.

Notice how, in this case, our top view we have designed with some perspective: when the puppets go up they turn and when they go down look straight.

Drawing our sprite-set

Nothing is easier than creating a new 256×32 pixel file, activate the grid to not get out, and draw the graphics. Here we will have to respect a very simple rule (or two, as you will see). To paint graphics and masks we will use two colors:

In the graphics, **pure black** is the color of the paper, and the other color (we want, it is a good idea to use white) is the color INK. Remember that the sprites will take the color of tiles that have a background as they move around the screen.

In masks, **black** is the part of the graph must remain solid, and another color is used to define the transparent parts.

In our examples, you will see that we use a different color (besides black) in graphics and masks, but it is mostly for clarity. You can use the colors you want. I personally use different colors because I usually paint mask and sprite in the same box 16×16, then select only the pixels of the "mask color" and move to the next box. It's a trick you can apply if you're crafty with your graphics editor.

Once we have all of our graphics and masks on file sprite-set, we keep it as **sprites.png** in **/gfx** and are ready to turn them into C code directly usable by the Churrera maker.

By the way, make sure **again that you used black is pure black**. All black pixels must be RGB = (0, 0, 0).

Converting our sprite-set

Again we have a little bit of command line window work in our hands. We will use another utility The Churrera Maker which, as you could guess, was the third we did: **sprncv.exe**, sprite-sets converter. This utility is really silly and simple and only takes two parameters. We're going to **/gfx** and run:

```
..\utils\sprncv.exe sprites.png.. \dev\sprites.h
```

****Editors note:**

I found it easier just to move the sprites.png file to the /utils directory and run sprncv.exe. Afterwards you may move the file back to the /gfx folder.

*With the sprites.png file in the /utils directory (in the command prompt) run
sprncv.exe sprites.png sprites.h
now move the file back to the /gfx folder*

Running this, and through a magical and mysterious process, we get a sprites.h file on our dev folder. And we're done.

Your sprites.h file should look like this...

```
61 defb 0, 255
62 defb 0, 255
63
64 ._sprite_1_a
65 defb 0, 255
66 defb 9, 246
67 defb 77, 178
68 defb 63, 192
69 defb 63, 192
70 defb 25, 224
71 defb 58, 192
72 defb 249, 0
73
74 defb 123, 128
75 defb 63, 192
76 defb 63, 192
77 defb 127, 128
78 defb 111, 144
79 defb 140, 115
80 defb 8, 247
81 defb 16, 239
82 defb 0, 255
83 defb 0, 255
84 defb 0, 255
85 defb 0, 255
86 defb 0, 255
87 defb 0, 255
88 defb 0, 255
89
90 ._sprite_1_b
91 defb 192, 63
92 defb 196, 59
93 defb 220, 35
94 defb 244, 3
95 defb 238, 1
96 defb 254, 1
97 defb 220, 3
98 defb 168, 7
99 defb 143, 0
100 defb 174, 1
101 defb 248, 7
102 defb 252, 3
103 defb 236, 19
104 defb 194, 61
105 defb 90, 159
106 defb 16, 239
107 defb 0, 255
108 defb 0, 255
109 defb 0, 255
110 defb 0, 255
111 defb 0, 255
112 defb 0, 255
113 defb 0, 255
114 defb 0, 255
115
116 ._sprite_1_c
117 defb 0, 255
118 defb 0, 255
119 defb 0, 255
120 defb 0, 255
121 defb 0, 255
122 defb 0, 255
123 defb 0, 255
124 defb 0, 255
125 defb 0, 255
126 defb 0, 255
127 defb 0, 255
128 defb 0, 255
129 defb 0, 255
130 defb 0, 255
131 defb 0, 255
132 defb 0, 255
133 defb 0, 255
134 defb 0, 255
135 defb 0, 255
136 defb 0, 255
137 defb 0, 255
138 defb 0, 255
139 defb 0, 255
140 defb 0, 255
141
142 ._sprite_2_a
143 defb 0, 0
144 defb 0, 0
145 defb 0, 0
```

In the next chapter, we will explain the theme of the fixed screens: the title, the frame of the game, and the screen of the end. Until then, to make sprites!

Workshop creates your own game of Spectrum (Chapter 5)

We already at our 5th chapter of the "Create Your Own Spectrum Game" Workshop. Today we end of sharpen the graphics and add the screens of starting and ending of the game among other things. We hope as always that you like it and take advantage of it.

Chapter 5: Ending graphics (for now)

First of all, download the package of materials corresponding to this chapter by clicking on this link:

<http://www.mojontwins.com/churrera/churreratut-capitulo5.zip>

What are we missing?

There is little left to draw. In this chapter we are going to explain two things: on the one hand, how to change the extra sprites (explosion and bullet). On the other hand, we will see how to make, convert, compress and include the frame of the game, the title screen, and the screen of the end.

Extra Sprites

At first, the issue of extra sprites change was not planned for this course / tutorial for a very simple reason: since the explosion to draw *Cheril Perils* and *Zombie Skull* bullet for those graphics have not changed to any game. Yes, we have had the task of remove all other games having explosions or bullets without changing those graphics. As it seems that there is interest in customization, let's use a few paragraphs explaining how to change them.

Needless to say, if your game does not have foes or bullets, or if they are worth the ones that are, you can skip this section.

All right. To change these graphics we have to do is replace those that are defined in the file **Churrera\dev\extrasprites.h**. Let's start by opening the file in our non-crap text editor and taking a look at it. In this file, the numbers are used to define three graphs for sprites (and their masks): the numbers 17, 18 and 19. The 17 is the one that corresponds to the explosion. The 18 is an empty sprite, which uses the engine for missing one of the three maximum enemies that fit on a screen and other things I now do not remember. That, obviously, we are not going to change it. The 19 is the sprite of the bullet and is smaller (you will see that there are fewer numbers).

Changing the explosion



The explosion is a 16×16 sprite. [Pest Alert] How splib2 is built, to define a 16×16 sprite, you have to do it in the upper left corner of a 24×24 frame. This is because splib2 is character-oriented, and a 24×24 character-aligned box can always contain a full 16×16 sprite. Do not worry if this sounds Chinese to you, you do not have to understand it (although if your curiosity is piqued, you can always ask us). We only mention it so that you locate and know what we are going to change. In the file, you will see three sprites 24 rows of pairs numbers, each under **a_sprite_17_X** label, where **X** is **a**, **b** or **c**. These are the three columns of 24 rows of 8 pixels that make up our sprite and are the data that we will have to replace. Place them well in the file.

Once done, save it as `\gfx\extra.png` and passed it to code using `sprncv` as we explained in the last chapter, but this time specifying `extra.h` as output file:

```
..\utils\sprncv.exe extra.png extra.h
```

Which will generate us a file `extra.h` in `\gfx`. We are going to open this file in a text editor.

This file will contain definitions for 16 sprites. We only need the first of these sprites for cut and paste it into `extrasprites.h`, replacing the 17. Let's do this: select the three columns of the sprite 1 (marked by labels. `_sprite_1_X`, where **X** is **a**, **b** or **c**) and copy to the Clipboard.

Now go to `extrasprites.h`, removes the three columns `_sprite_17_X`, paste three columns `_sprite_1_X` Clipboard, and now, carefully change each label `_sprite_1_X` by `_sprite_17_X`.

Perfect! Please beware of this. Make sure that you change the correct data and that modified the tags correctly.

Changing the shot

The original shot is a ball centered on a box of 8×8. We put a ball because we use the same graphic for shooting in all directions, so a form "oriented" not worth it to us.

In the same way that a 16×16 graphic is to define it in one greater than 24×24 picture by the issue of the positioning on the screen, there is one 8×8 to define it in a box of 16×16, or what is the same, in two columns of 8×16 pixels. In the file `extrasprites.h` are the hooligans who under the labels. `_sprite_19_a` and `_sprite_19_b`.

For reasons of speed, bullets have no mask. This means that the graph is defined by the first value of the first eight lines. As you can see, these 8 values are **0, 0, 24, 60, 60, 24, 0, 0**. If you go to binary values and the put one on top of another, you will see how they form a ball:

```
00000000
00000000
00011000
00111100
00111100
00011000
00000000
00000000
```

We are going to change it, for example, by a star ninja (friends, a little imagination):

```
00000000
00010000
00011000
00100110
01100100
00011000
00001000
00000000
```

We could pass the values to decimal and replace them, but it is not necessary, because the z88dk assembler is a competent guy (as Vincent, the clever monkey) and swallowed the binaries directly if we stick them at the beginning one @, so what we will do is replace the first numbers of the first 8 pairs of values that exist under `._sprite_19_a` by something:

```
125 #INJECT_PLAYER_CAV_FIRE
126 #asm
127     ._sprite_19_a
128     defb @00000000, 0
129     defb @00010000, 0
130     defb @00011000, 0
131     defb @00100110, 0
132     defb @01100100, 0
133     defb @00011000, 0
134     defb @00001000, 0
135     defb @00000000, 0
136     defb 0, 0
137     defb 0, 0
138     defb 0, 0
```

Damn, what did you say?

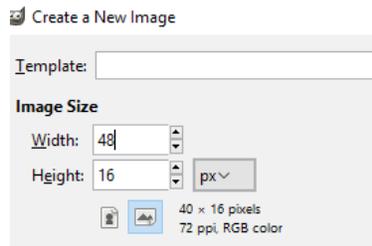
Yes, try not to charge anything. And you're annoying, for asking. It is likely that right now you are thinking now that the explosion and bullet that come included by default are not so bad either.

Editors Notes – Extra Sprites additional tutorial

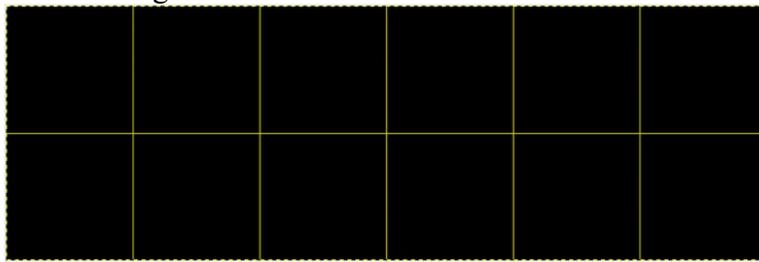
Here's my personal tutorial on extrasprites.h

Open up your favorite image editor (in this case we are using GIMP).

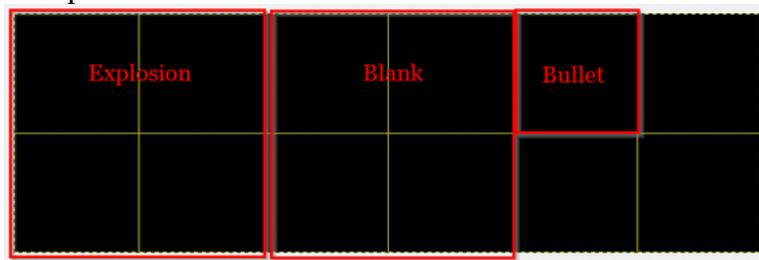
Create a new image 48 pixels wide and 16 pixels high.



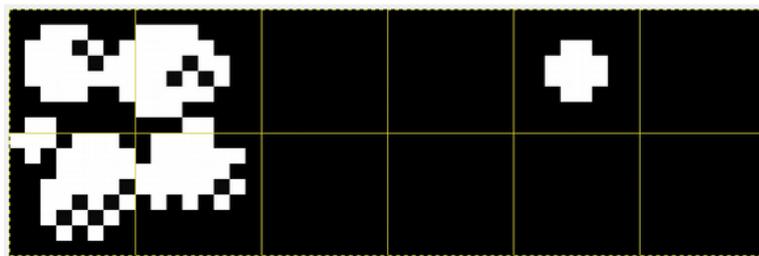
Zoom in the image and show the grid



The layout for your extra sprites are as follows



So a layout might look like this



Paint your extra sprites and save as **extrasprites.png**.

Move the image to the /utils directory

use the Sprite conversion utility: `sprcnv.exe extrasprite.png extra.h`

Your sprite should now be converted.

Open the extra.h file in your favorite text editor.

You should see something similar to this

```
// Sprites.h
// Generado por SprCnv de la Churrera
// Copyleft 2010 The Mojon Twins

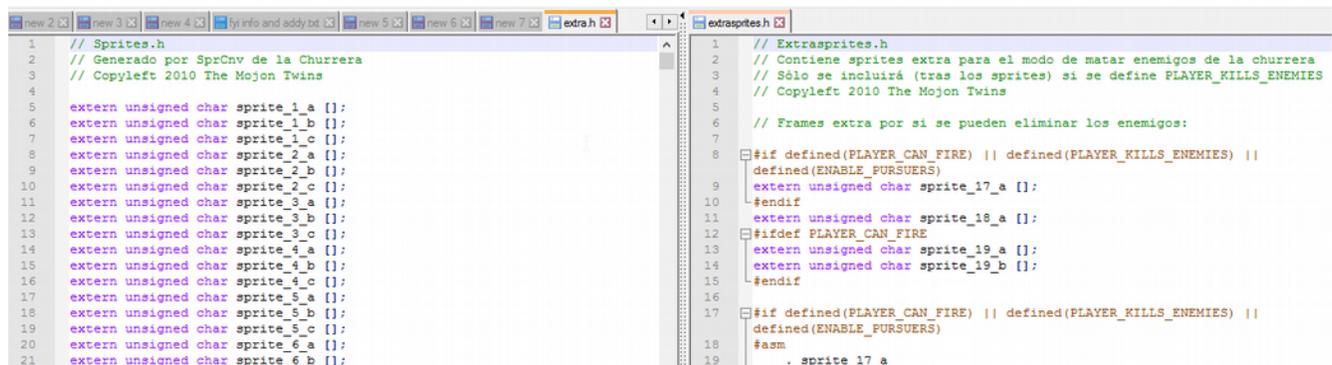
extern unsigned char sprite_1_a [];
extern unsigned char sprite_1_b [];
extern unsigned char sprite_1_c [];
extern unsigned char sprite_2_a [];
extern unsigned char sprite_2_b [];
extern unsigned char sprite_2_c [];
extern unsigned char sprite_3_a [];
extern unsigned char sprite_3_b [];
extern unsigned char sprite_3_c [];
extern unsigned char sprite_4_a [];
extern unsigned char sprite_4_b [];
extern unsigned char sprite_4_c [];
extern unsigned char sprite_5_a [];
extern unsigned char sprite_5_b [];
```

This is not quite yet usable, our data is in there, but we have to make some modifications.

I'm going to promote my favorite text editor Notepad ++, it has two noticeable features that I am going to take advantage of in this part of the tutorial: tabbed editing and split screen editing.

Browse to the /dev folder in the Dogmole directory and open up extrasprites.h.

in Notepad ++ go to view - Move/Clone current Document – move to other view.



Tabbed and split views in Notepad++

You will notice that the extrasprites.h file consist of several different sections

```
// Extrasprites.h
// Contiene sprites extra para el modo de matar enemigos de la churrera
// Sólo se incluirá (tras los sprites) si se define PLAYER_KILLS_ENEMIES
// Copyleft 2010 The Mojon Twins

// Frames extra por si se pueden eliminar los enemigos:

#if defined(PLAYER_CAN_FIRE) || defined(PLAYER_KILLS_ENEMIES) || defined(ENABLE_PURSUERS)
extern unsigned char sprite_17_a [];
#endif
extern unsigned char sprite_18_a [];
#ifdef PLAYER_CAN_FIRE
extern unsigned char sprite_19_a [];
extern unsigned char sprite_19_b [];
#endif
```

Declarations

```
#if defined(PLAYER_CAN_FIRE) || defined(PLAYER_KILLS_ENEMIES) || defined(ENABLE_PURSUERS)
```

```

#asm
_sprite_17_a
defb 0, 128 //00000000
defb 56, 0 //00111000
defb 117, 0 //01110101
defb 123, 0 //01111011
defb 127, 0 //01111111
defb 57, 0 //00111001
defb 0, 0 //00000000
defb 96, 0 //01100000
defb 238, 0 //11101110
defb 95, 0 //01011111
defb 31, 0 //00011111
defb 62, 0 //00111110
defb 53, 128 //00110101
defb 42, 128 //00101010
defb 20, 128 //00010100
defb 0, 192 //00000000

defb 0, 255

```

```

_sprite_17_b
defb 0, 3 //00000000
defb 240, 1 //11110000
defb 248, 0 //11111000
defb 236, 0 //11101100
defb 212, 0 //11010100
defb 248, 0 //11111000
defb 224, 1 //11100000
defb 24, 0 //00011000
defb 124, 0 //01111100
defb 120, 0 //01111000
defb 244, 0 //11110100
defb 168, 0 //10101000
defb 0, 1 //00000000
defb 0, 3 //00000000
defb 0, 63 //00000000
defb 0, 127 //00000000

defb 0, 255

```

```

_sprite_17_c
defb 0, 255

defb 0, 255
defb 0, 255
defb 0, 255
defb 0, 255
defb 0, 255
defb 0, 255
defb 0, 255

defb 0, 255
defb 0, 255
defb 0, 255
defb 0, 255
defb 0, 25
defb 0, 255
defb 0, 255
defb 0, 255

```

```

#endasm
#endif

```

This equates to



In the Notepad ++ tab that has your new sprite definition (extra.h) find `_sprite_1_a` (should be at line 64). Lines 65 – 79 is the sprite definition for the first 16 lines, left side of your sprite. Copy these lines (65 – 79) and paste them into your `._sprite_17_a` data, lines 20-35.

We are going to repeat this for `._sprite_1_b` (lines 91 to 106) to `._sprite_17_b` (lines 47 to 62).

For the projectile sprite we are going to use `._sprite_2_a`

Copy the first 8 lines (lines 143 to 150) and copy them to `._sprite_19_a` (lines 132-139). Save your `extrasprites.h`



```
9      defb 0, 255
10     defb 0, 255
11
12     ._sprite_2_a
13     defb 0, 255
14     defb 0, 255
15     defb 24, 255
16     defb 60, 255
17     defb 60, 255
18     defb 24, 255
19     defb 0, 255
20     defb 0, 255
21
129     #ifndef PLAYER_CAN_FIRE
130     #asm //ball shape
131     ._sprite_19_a Extrasprites.h
132     defb 0, 0
133     defb 0, 0
134     defb 24, 0//0001 1000
135     defb 60, 0//0011 1100
136     defb 60, 0//0011 1100
137     defb 24, 0//0001 1000
138     defb 0, 0
139     defb 0, 0
140     defb 0, 0
```

Fixed screens

Well, let's go with the theme of the fixed screens. Basically, Churrera Maker games have three fixed screens: **the title screen**, which is what also shows the menu (to select the type of control and start playing), the **framed screen game**, where will be located markers lives, objects, and so on, and the **end screen**, it will be shown that once the player has accomplished the objective assigned.

To save memory also is the possibility that the title screen and the screen with the frame of the game are the same, thus saving enough memory, and that will be the option we will take to our *Dogmole*. But do not worry we will explain everything.

Another thing I do not want to fail to mention is that the screens are stored in the game in a compressed format. The type of compression used (like almost all compression) works the better the simpler the images. That is the more repetition and / or fewer things on the screens, the less they will occupy the end. Keep this in mind. If you look at it from memory, one way to save that works very well is to make your fixed screens less complex.

Title screen

As we have already mentioned, it is the screen that is displayed with the title of the game and control options. The selection of control is fixed: if the player presses 1 will select the control by keyboard. If you press 2, you will choose Kempston, and if you press 3 it is because you want to play with a joystick of the Sinclair standard (Interface 2, port 1). What does this mean? Well, we will have to draw a screen that, besides the title of the game, shows these three options. For example, something like this:



When you make yours, save it as title.png in the \gfx directory.

Editors Notes – Using ZX Paintbrush

Another tool I would like to promote in this document is ZX Paintbrush available at <http://www.zx-modules.de/> .

This tool can be used by itself or with the collection of modules that the author has written. It's a handy utility that allows one to capture graphics from anywhere on the web or your image program.

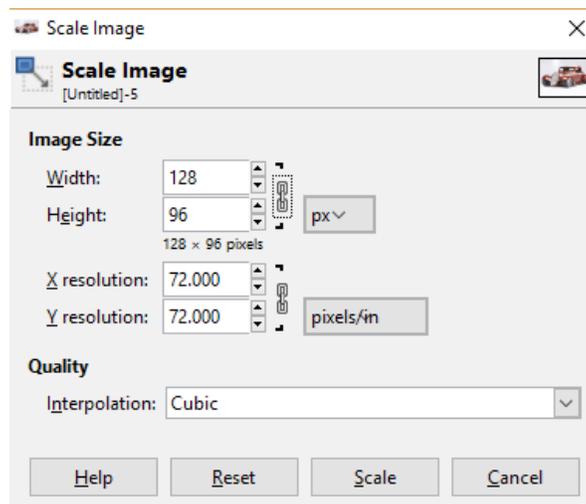
Let's say that I want to import an image of a hot-rod. I would first find a nice image on the web somewhere.



I am going to capture this image and paste it into GIMP.

I want to resize the image so that it fits into the area that I want it to. In this case let's make it 128x96.

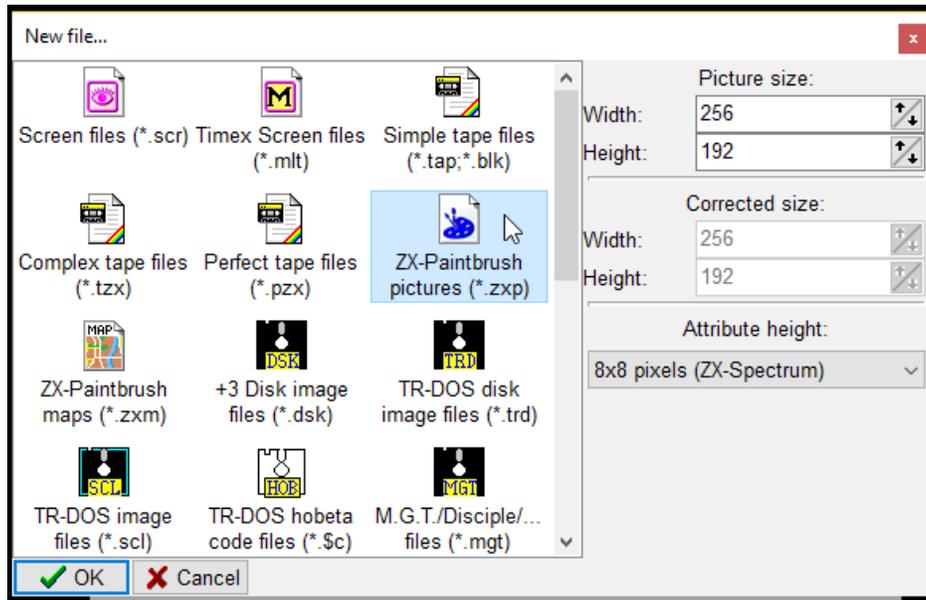
In GIMP, I would drill to Image – Scale Image.



Select scale when finished.

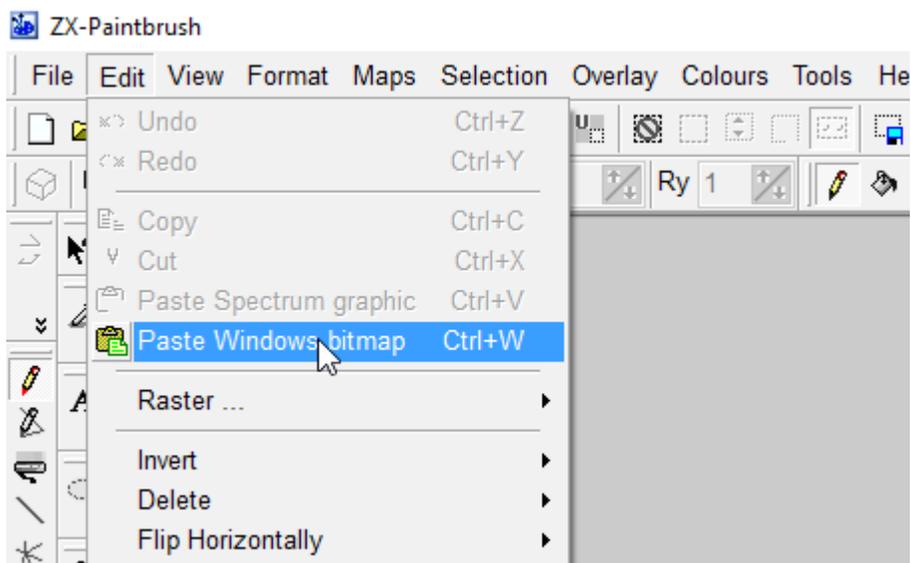
You can now either modify your image within GIMP, or if you are happy with the image, simply select the entire image and copy.

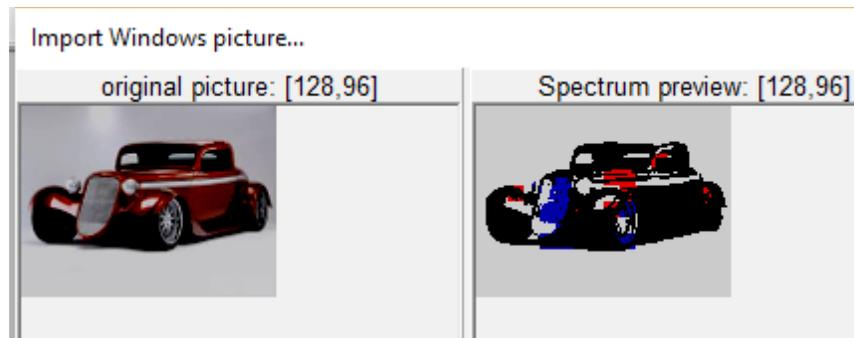
Now, let's open ZX Paintbrush.



Drill to File – New. Select ZX-Paintbrush picture.

Drill to Edit – Paste Windows bitmap.

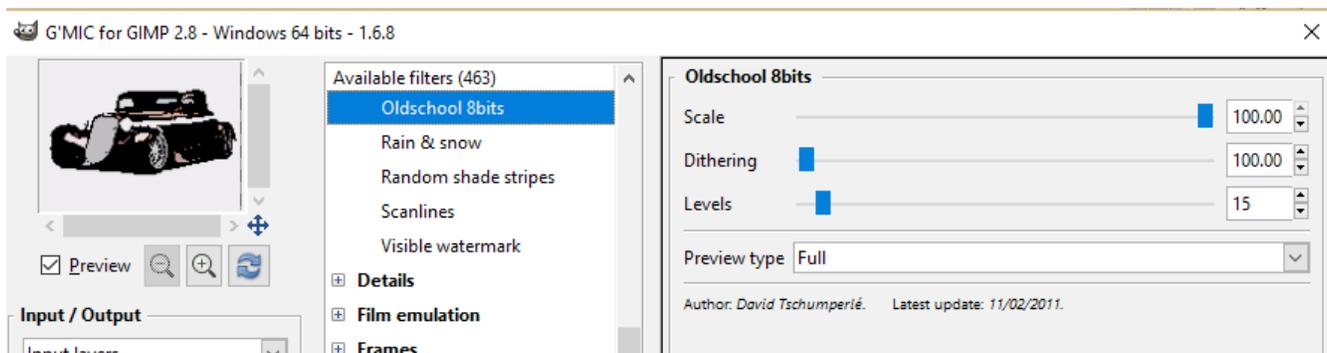




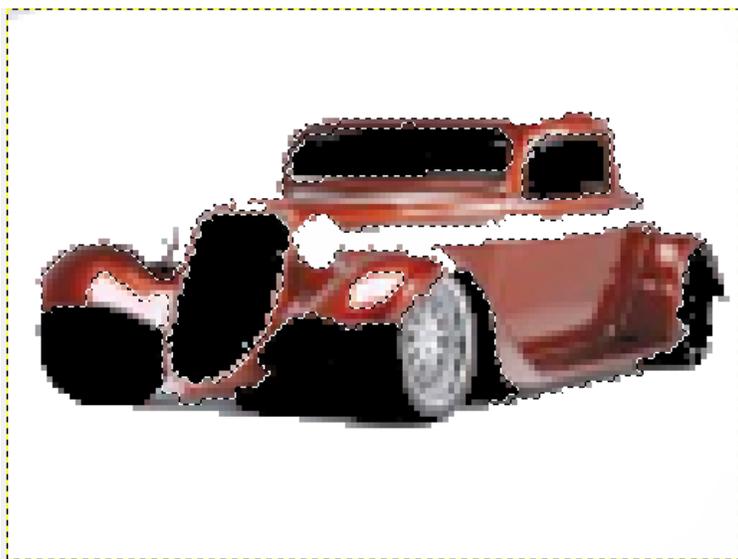
If you are not happy with the results, there are a number of things you can do prior to committing. This is just a preview. Try some of the options available to you in ZX Paintbrush, such as adjusting the brightness and contrast or try dithering.

If you are still not happy, go back to GIMP and try to simplify the color scheme.

Another way to go is by using a filter called G'MIC for GIMP, a very powerful filter. One of the many filters available in G'MIC is "Old School 8 Bit".



Another popular tool is the magic wand, select the areas that you want to be the same color and continually refine the selection process.

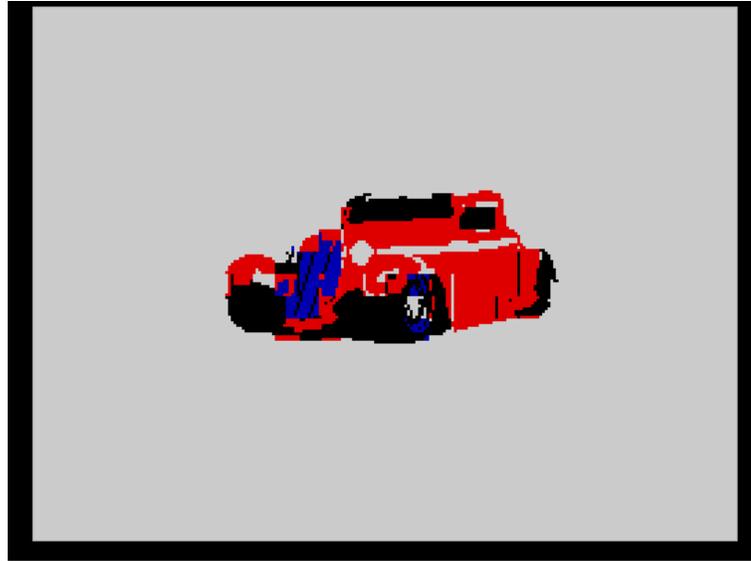


Paint in the selected areas



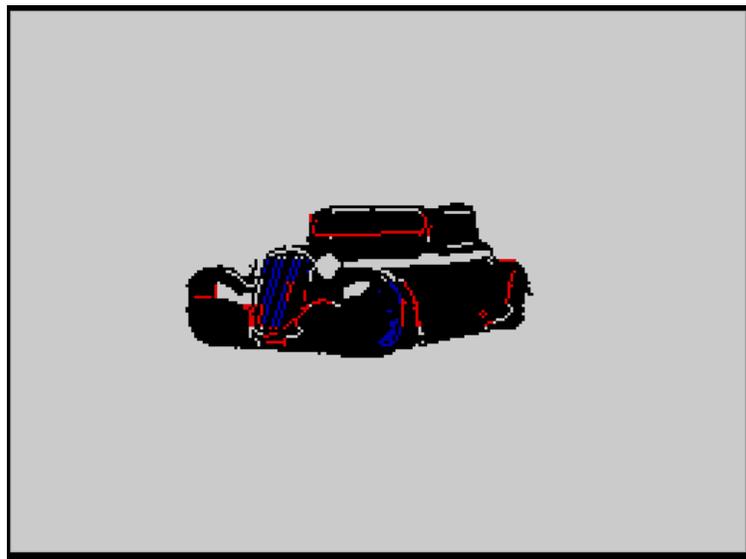
After enough of fighting with the image, you just might come up with something decent.

When happy, paste the Bitmap into ZX Paintbrush.



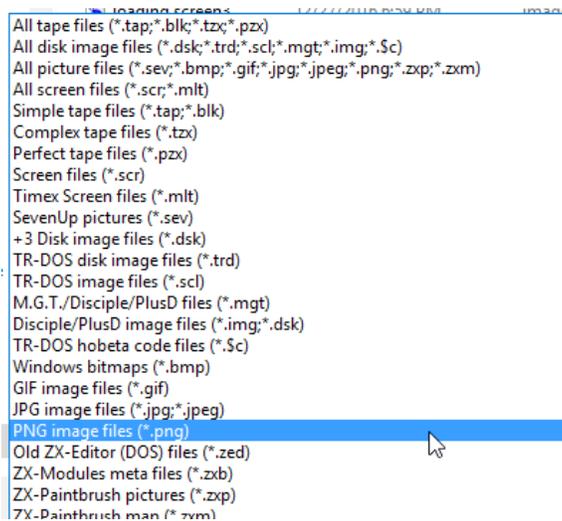
You still may have to fidget around with the image.

Or try something new



It's all an artistic choice.

When Finished drill to File – Save as.



Save as a PNG.

The framed screen game

There is more here. The screen of the game frame must reserve several important areas and then separate them with an ornament and indicators. The areas that we have to reserve are:

The **playing area**, which is where things happen. As you guessed, it should be just as big as each of our screens. If you remember, the screens are 15×10 tiles and, therefore, they occupy 240×160 pixels. You must reserve an area of that size as the main area of play.

The **number of lives / energy / vitality / whatever**: two numeric digits to be drawn with the font you defined when we made the tileset. You should place it somewhere in the frame, adding some graphic that gives it meaning. We usually put a cartoon of the protagonist and an "x", as you will see in the examples.

The **marker keys** (if you use keys on your game). It has the same characteristics as the life marker.

The **marker object** (if you use objects in your game). Ditto for objects.

The **number of killed enemies** (if you can kill enemies and it is important to count). Well, the same.

All these things we can place where we want, as long as they are aligned with character. In addition, we must point out the position of each of these things (in character coordinates, that is, 0 to 31 for the X and 0 to 23 for the Y), because then it will have to be indicated in the configuration of the Churrera Maker (as we will see within a couple of chapters).

Let's look at an example. This is the frame screen *Lala Prologue* game. Let's look at the different elements. In *LaLa* we have keys and in addition, we have to collect objects, so we will have to put those two things, along with the energy marker, in the frame, as we see:



The **playing area**, as we see, begins at coordinates $x = 1$, $y = 2$, that is, (1, 2).

The space we have left for the **energy counter** is at (4, 0).

The **marker objects** displayed such that (11, 0).

Finally, the **marker keys** are (18, 0).

As I say, we have to point all those values because we will have to use them when building the configuration of our game, within a couple of chapters.

When you have yours, it is recorded as **marco.png** in the `\gfx` directory.

Combined title and frame screen

Since very early we started doing this because it helped us save a lot of memory. The theme is simple: to a screen of the game frame, you add the title and menu options within the area reserved for the main area. The truth is that it looks quite good, and, reiterate, you can save enough memory.

This is, in fact, what we are going to use in our *Dogmole*, as we see here:



As we see, on the one hand, we have the title of game and control options, as we explain when we talk about the title screen. On the other hand, we have the score of the game. The game area, as will be shown, will be displayed on the space that occupies the title and control options:
The **play area** will be placed at coordinates (1, 0).

Counter for eliminated enemies (sorcerers) will be drawn in (12, 21).

The **box counter** (ie, objects) will be drawn in (17, 21).

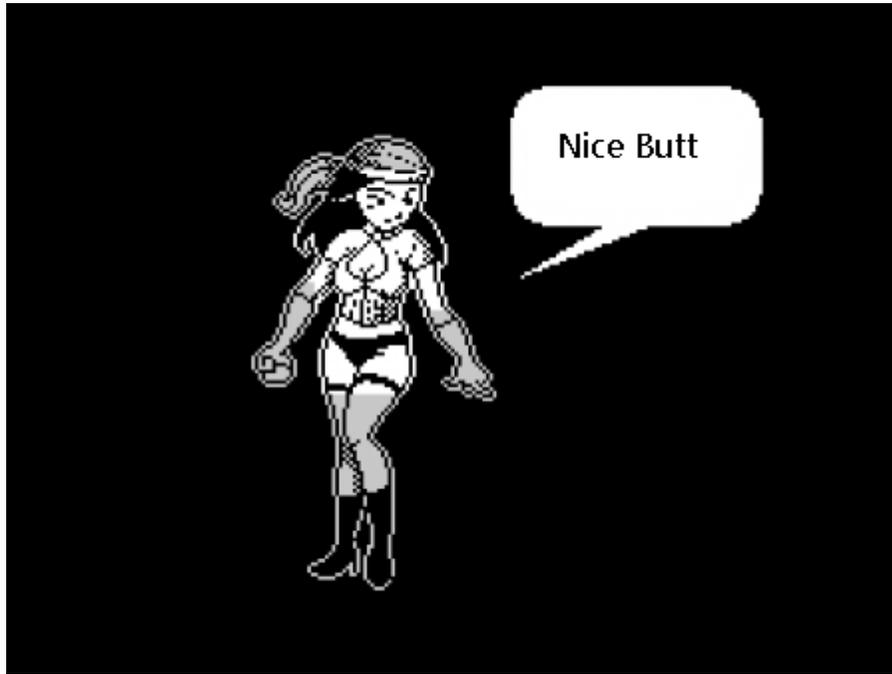
The **life counter** is drawn in (22, 21)

Finally, the **counter keys** we will have in (27, 21).

In this case, the title screen / combined frame is saved as **title.png** in **\gfx**. There will be no marco.png file for games that combine title and frame on the same screen, as in our case with *Dogmole*.

The final screen

It is the screen that is displayed when the player finishes the game successfully. There is no restriction here: you can draw whatever you want. If you have the skill it looks better, but if you like to make serious game maybe not paste. This is the end of *Dogmole* screen:



When we have the record it as ending.png in the gfx directory.

Converting screens to Spectrum format

For this, we will use **SevnuP** again. You can use other programs for these purposes, but as we started, then SevnuP. Once opened, I press Import. In the file selection box that appears seek our **title.png**. SevnuP will load the file and convert it to the Spectrum screen format. Press S to save, and we recorded it in \ **gfx** as **title.scr**.

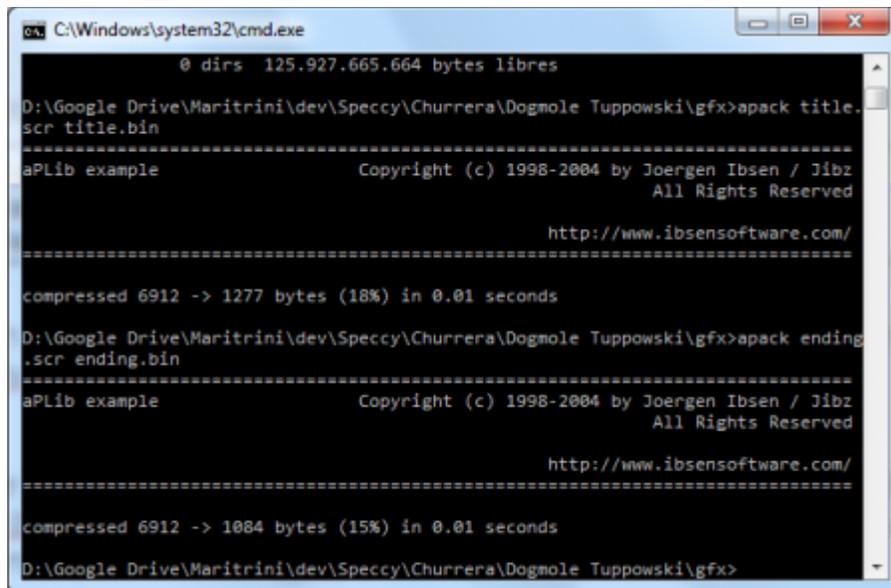
We do the same for **marco.png** (if we will use) and **ending.png**. At the end, we stay with **title.scr**, **marco.scr** (if we will use) and **ending.scr** in \ **gfx**. Do not be goofy and delete **png**, then if you want to change something or reuse the framework for another game.

Compressing screens

As you may have noticed, the 6912 bytes that each screen occupies for three (or two) are large, so it will have to be compressed. For that, we will use the **apack.exe** compressor package **aplib** library. We have included in the **utils** folder. We opened our window command line, we go to the **gfx** directory and compress each scr in a file named bin but at the end with these three orders (or two, if not going to use the separate frame).

```
.. \utils \ apack.exe title.scr title.bin  
.. \utils \ apack.exe frame.scr frame.bin  
.. \utils \ apack.exe ending.scr ending.bin
```

With this have obtained three (or two) that we **copy the .bin files to the \ dev folder**.



```
C:\Windows\system32\cmd.exe  
0 dirs 125.927.665.664 bytes libres  
D:\Google Drive\Maritrini\dev\Speccy\Churrera\Dogmole Tuppowski\gfx>apack title.  
scr title.bin  
=====  
aPLib example Copyright (c) 1998-2004 by Joergen Ibsen / Jibz  
All Rights Reserved  
  
http://www.ibsensoftware.com/  
=====  
compressed 6912 -> 1277 bytes (18%) in 0.01 seconds  
D:\Google Drive\Maritrini\dev\Speccy\Churrera\Dogmole Tuppowski\gfx>apack ending  
.scr ending.bin  
=====  
aPLib example Copyright (c) 1998-2004 by Joergen Ibsen / Jibz  
All Rights Reserved  
  
http://www.ibsensoftware.com/  
=====  
compressed 6912 -> 1084 bytes (15%) in 0.01 seconds  
D:\Google Drive\Maritrini\dev\Speccy\Churrera\Dogmole Tuppowski\gfx>
```

If you look, the size of the files has dropped drastically. For *Dogmole*, title.bin 1277 bytes and occupies 1084 bytes ending.bin. Between the two they add 2361, which is considerably less than the 13824 bytes they would occupy without being compressed.

And we're done

What, now? What a boring chapter. I know. But hey, it was necessary. In the next chapter, we will learn how to place enemies, objects and keys on the game map. And soon, very soon, we can compile for the first time to start seeing everything in motion.

Workshop creates your own game of Spectrum (Chapter 6)

A new installment of the workshop "Create your own game of Spectrum" with the Churrera Maker of the Mojon Twins. On this occasion, we will put the different elements that we have been creating. Without further ado we go there:

Chapter 6: Placing Things

First of all, download the package of materials corresponding to this chapter by clicking on this link:

<http://www.mojontwins.com/churrera/churreratut-capitulo6.zip>

Now, what?

That's when we put things. It is the equivalent of when you arrive from the supermarket loaded with bags and you have to place them throughout the kitchen. It is a tedious but necessary process. You can choose not to, of course, but then to see who finds the sausages. Or, I do not know, the filters on the coffee maker. Does anyone still use coffeepot filters?

Basically what we are going to do is to put on the map the enemies, the objects, and the keys. And no, we are not referring specifically to doing this:



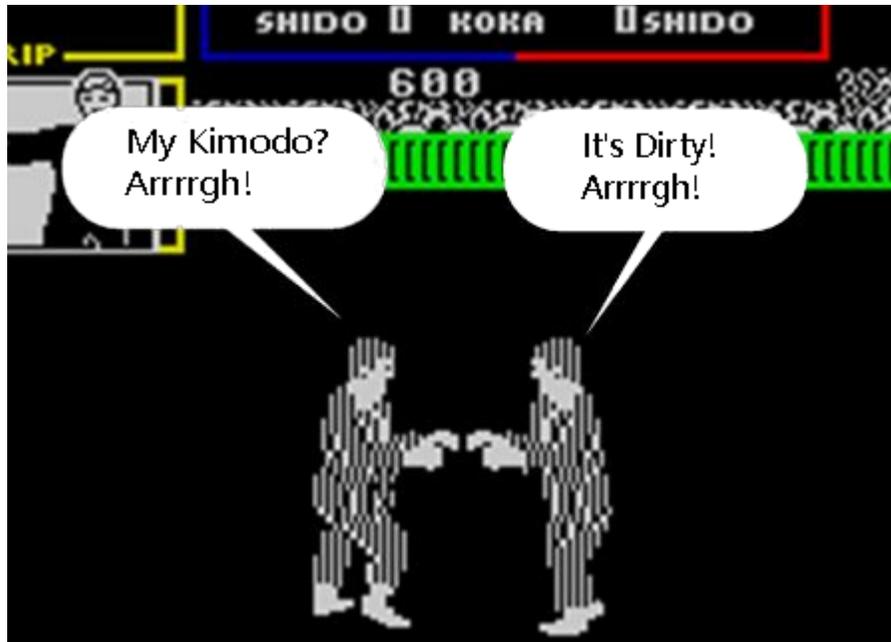
Each item will belong to a particular screen and will have a series of values that indicate its position, its type, its speed in the event that they move, and other things like that. We are actually talking about a huge number table that would be quite a pain to create by hand. That is why we have made an application to the host of grotesque to be able to complete this work in a visual way.

The tool of which I have spoken, we call that very imaginatively **The Underwriter** (nothing to do with illegal substances), is not only useful for making games with Churrera maker. Any project that you have with tiles of 16×16 pixels, for any size of screen, could benefit from this utility. Without going any further, we use it to place bugs in games *Uwol 2* of CPC, *Lala Prologue* of MSDOS or some experiment for some 16-bit that console to see if someday we dared to finish and get a blessed time. The program will seem very cumbersome, but the truth is making the workaround.

Incidentally, **the Underwriter** is a program for Windows platforms. However, being compiled with GCC and using Allegro as a graphical library, it should be very easily portable to Linux. If you are willing to do so and provide an additional tool for the users of this platform, contact us. The program is completely contained in a .c file so generating a Linux executable should be the simplest task on the globe.

Basics: Enemies and Hot-spots

Let's start by explaining some basic concepts before we start the application, more than anything because the terminology we use tends to be less intuitive than Uchi-Mata.



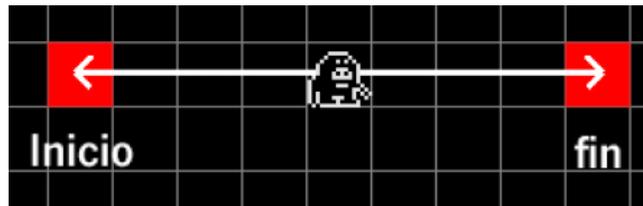
Enemies

First we have the concept of **enemy**. An enemy, for Churreras, are those things that move on the screen and they kill and also moving platforms. Yes, for The Churrera Maker moving platforms they are enemies. So when we talk about placing enemies, we also talk about placing moving platforms. And when we say that the screen can have a **maximum of 3 enemies**, you must also count the moving platforms.

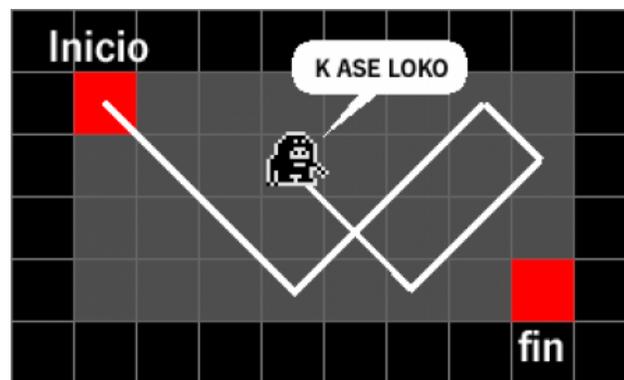
Enemies have associated different values and something very important: the type. The type of enemy defines its behavior and, in addition, the graph with which it is painted. Attention because this is somewhat confusing, especially since it was not designed as a priority and is somewhat patched (read on the subject of moving platforms):

The enemies of **types 1, 2 or 3** (and **type 4** in the side view game) describe linear paths and are drawn with the first, second or third (or fourth) tileset sprite enemies. When we speak of **linear paths** we refer to two possible cases:

Trajectories warp, vertical or horizontal paths: the starting point and end point are defined describing a straight vertical or horizontal line. The enemy follows that imaginary line coming and going continuously.



Diagonal paths: they came out of a "feature" of the engine (side effect not intended due to a crappy algorithm), but leave them because they are cool. We have used them much. If the starting point and the end are not in the same row or column, the puppet moves inside the rectangle describing both points, bouncing off the walls and moving diagonally.



The enemies of **type 4**, on side view games, are moving platforms. They behave exactly like linear enemies, but with a limitation: although we can define a diagonal path, we do not guarantee that the operation is correct in all cases. Therefore, only vertical or horizontal trajectories can be used.

The enemies of **type 5** were added when we did the *Zombie Skull*. It's the type of bats. They are painted with the graph of the third enemy of the tileset and have this special behavior: they are created off-screen, and if the protagonist is not hidden, they are chasing him. If the protagonist hides, they move away until they leave the screen again. We have not tried this type of enemies in this revised version of Churreras, so we can not assure that work well. Do you dare to try? By default, the code to move and manage type 5 enemies is not included in the engine, but must be explicitly activated.

The enemies of **type 6** are not implemented in this version of Churreras for one simple reason: they have always been totally and completely custom. *Cheril the Goddess* in myopic bats were only chasing you if you approached a distance by walking away and returning to your site. In the Gheg Ramiro Vampire bats are the crypts that appear when you activate the trap and pursue you until you catch all the crosses. In *Uwol 2* for CPC, they are the Fanty of the lower floors. For each game in which we have used type 6 we have programmed a different behavior (reusing things, yes). That is why we have left this "hole", in case you need a custom behavior for some game we have where to add it. Just like the engine, if you create a type 6 enemy, it will not even appear on the screen.

The enemies of **type 7** are what we call **EIP**, i.e. **Enemies Incredibly persistent**. These are linear pursuers. These enemies appear at the point where you create them and are dedicated to chasing the player. They can not cross the stage. If the player can shoot and kill them, they will appear again in the same place from where they first appeared. The graph with which they are drawn is chosen randomly from all available ones. They work best in top view game and you can see them in action in Mega Meghan. Like type 5 enemies, the code to manage them is not included in the default engine and must be explicitly specified.



At *Dogmole* will use only the enemies of types 1 to 3 and type 4 moving platforms.

For the final chapters of this tutorial, which will deal with additions and modifications, we will see how to do two things with the engine: add a type 6 with a custom behavior and how to modify the entire enemy engine to be able to use more than 4 basic linear types of Two ways: using 8 different types without animation, or adding more frames to have 8 different types of animation (spending a lot of memory in the process). But for this, there is still a lot.

Hot-spots

Hot-spots are, quite simply, a position within the screen where there may be an object, a key, or a recharge. In each display a single **hot-spot** is defined. Each hot-spot has an associated value. If it is given a value "0" this hot-spot will be disabled and nothing will ever appear. If a value of "1" is given, an object will appear in this position. If it is given a value of "2", a key will appear. Hot-spots with values "1" or "2" are considered "active". Once we have taken the key object or an active hot-spot, the engine may make appear a refill of life there next time we go on that screen.

When we talk about **objects** we refer to the item that is drawn with the tile tileset number 17 and will be automatically counted by the game engine without having to do us nothing but tell the engine that will use objects. It is in *Lala Prologue* potions or crosses *Zombie Skull*, pencils in *Journey to the Center Napia* or diskettes in *Working Trash*. We are going to use objects to represent the boxes that we have to pick up and take to Miskatonic University in our game. Then, through scripting and configuration, we will make the behavior of the objects different (we will not count the object until, after taking it, we have deposited it in a certain point of the university, much like how floppy disks work working *Trash*), but basically objects should be placed on the map using hot-spots object type (type 1).

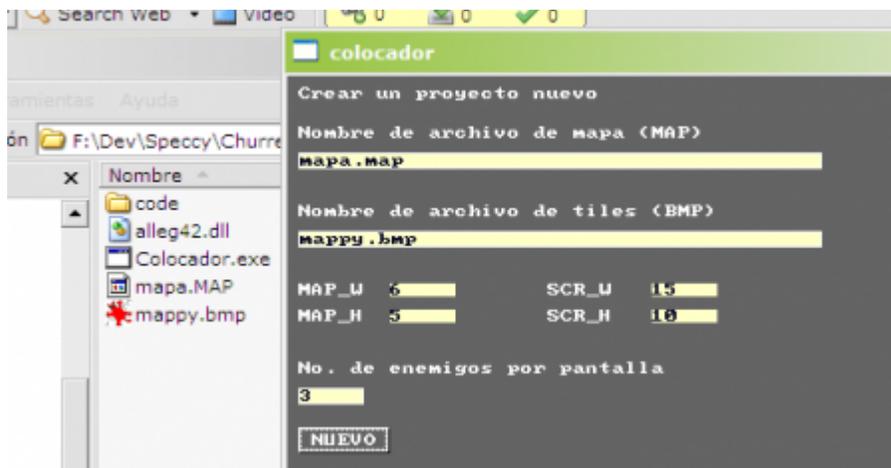
As I know you are going to ask, **you can not assign more than one hot-spot on the screen** as it is programmed the engine.

Preparing the necessary materials

All right. Let's get down to business. The first thing we will have to do is copy the necessary materials to the directory where the Colocador is, which is the `\enems` directory. We need two things: the map of game in `.map` format, and the tileset we prepared for Mappy in `.bmp` format. Therefore, copy `\map \map.map` and `\gfx \mappy.bmp` to `\enems`. We are ready for the march!

Setting up our project

When you run the **Underwriter** (for example, by double clicking on `colocador.exe`) the main screen that appears, or charge an existing project, or configure a new one. As we do not have an existing project, we will create a new one.



As we see, in the first two boxes will have to enter the name of our map and our tileset (**mapa.map** and **mappy.bmp**, which should already be in the `\ENEMS` directory). Then there are four boxes that define the size of the map and the screens. **MAP_W** and **MAP_H** should contain the width and height of your map measured on screens. **SCR_W** and **SCR_H** are to specify the width and height of the screens in tiles. Paras The Churrera Maker the Ghieg of these values are, as should already know plenty, **15** and **10**.

The last box to fill is the number of enemies that will appear on each screen. In the Churreras are 3. No, if you put more they will not go more in the Game. You'll just make it all go wrong. Put 3. I assure you that 3 are enough.

When all the files are clicked on **NEW** and then we can start working.

Basic program management

The handling is very simple. More than anything because the program is very simple. If you look, there is a grid with the current screen. If the grid does not come out, or what comes out on the grid is not the first screen of your map, ill come on. Review the previous steps, especially those referring to the dimensions of the map and the screens.

To navigate the map (so that another screen appears in the grid) we will use the keys of the cursors. Test it. You should be able to do a nice tour of your map.

If you press **ESC** the program, something tremendously useful when we are finished will close and we want to go. But be careful: it comes out no more. Careful, careful, careful. **Do not press ESC without saving earlier.** It does not warn you. The program closes.

Precisely to have the **S** key to record. In fact, we will use it right now, even if we have not placed any enemies. By convention, we use .ene as an extension for the Colocador files. Press **S** and in the dialog that comes out, writes enems.ene and click on the OK button. We have already recorded our placements. Do this very often. Hear us.

Let's see this in action. Press **ESC** to exit. Note that there is now a **enems.ene** in \ **ENEMS** file. **Colocador.exe** reruns. This time, instead of filling in values, writes enems.ene in the box is labeled Open an existing project and click on the button that says **Load**. If all goes well, you should exit the first screen on the map again.



This file **enems.ene** is not used to use directly in the game. In order to have our enemies in the game will need the stand to export code to C. This is done by pressing the E. When we do, a dialogue similar to that of the save will appear. There must write **enems.h**, press **OK**, and copy this file to \ **dev**. This will be what we will do when we have finished placing all the enemies and we want to integrate them in the game.

Putting enemies and platforms

The simplest is to place linear enemies (horizontal, vertical, or those rare diagonals we saw before). What is done is to define a trajectory, a type, and a speed. Let's do it.

The first thing to do is place the mouse over the **box beginning of the path and click**. This position will be the initial one, where the enemy will appear, and it will also serve as one of the limits of their trajectory (look at the pictures above, when we talked about enemy types). When we do this, a dialog where we enter the **type of enemy** will appear. Remember that in the case of linear enemies will be a value of 1 to 4, 4 for platforms in side view games. Put the little number and press OK.

Now what the program hopes is to tell you where the trajectory ends. We go to the **square where the path must end** and we click again. We will see how the path is shown graphically and a new dialog box appears in which we are asked about speed.

The value entered will be the number of pixels that will advance the enemy or platform for each game box. These values, so that there are no problems, should be powers of two. That is, 1, 2, 4, 8... Actually, the values that are worth 1, 2 or 4. Something beyond is already too fast and would give problems of all kinds. Values that are not powers of two can also give problems. If you want you can try to put a 3 or something to see what happens, but I tell you since you are likely to end up with the enemy going fishing off the screen or worse. Once we have put the numerical, press OK and we already have our first enemy placed.



For enemies of other types (5 or 7, for now) the subject changes a little. For example, type 5 enemies no matter where you put: always come from outside of the screen (see bats *Zombie Skull* to know what I mean). You can set the beginning and end of the path where you feel like it. The speed you put is also the least. With the enemies of type 7 (Mega Meghan) only take into account the starting position. The speed and position of the end will be ignored equally.

When we put, in the future, to program in the engine a type of enemy custom for the type 6, we will do taking into account that we have the positions of start and end and a value of speed to play. For example, if we are going to make a cannonball that fires, we can use the end position to define the direction in which it will fire. Well, let's not go ahead. In addition, *Dogmole* does not use non – linear enemies.

You can put a maximum of three per screen (the program will not let you push more). There does not have to be three on each screen, you can have screens with one, two, or none.

To **delete or edit** the values of an enemy we have already placed, just click on the box start of the path (where the little number that indicates the type appears). Then we will see a dialog box where we can change its type or speed or eliminate it completely.

And so, little by little, we will place our enemies and platforms on the map, with a maximum of 3 per screen, being careful not to put a value out of range as an enemy, and not forgetting that the speed must be 1, 2 or 4.

Placing hot-spots

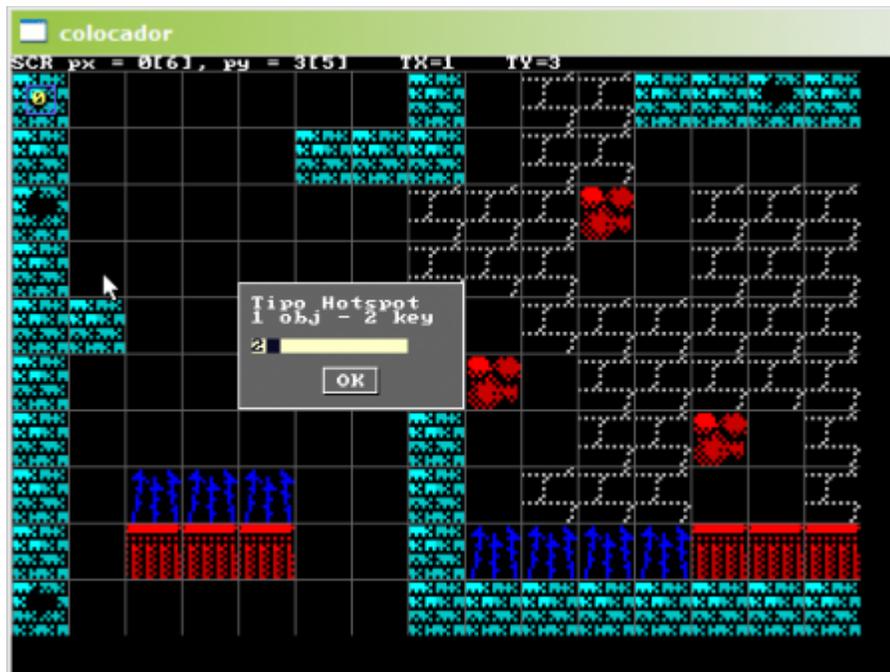
As we said earlier, a **hot-spot** is the box where a key or an object appear during the Game, and where a recharge may appear if we caught after said key or said display object.

Remember, we mentioned that each hot-spot has a type. For the Churrera Maker, this type can be 0 (deactivated), 1 (contains an object) or 2 (contains a key). If you get cool then you can expand the engine for more, but as it is the Churrera Maker supports these three types.

Recall that each screen supports a single hot-spot. That means that the total number of keys and objects needed to finish the settling can not exceed the number of screens. For example, we have 30 screens *Lala Prologue*. In order to finish the game successfully it is necessary to collect 25 objects, and in addition there are four locks to open, reason why we need 4 keys. This means that in 25 of the 30 screens there will be an object, and in 4 there will be a key. Perfect: we need 29 screens of 30. It's important to plan this beforehand. If you get excited by placing a bunch of locks then you will not be able to fit all the keys and all the objects that need to be picked up.

You should also make sure that you **put enough keys to open all the locks** and that there is always a way to finish the game without getting locked in. Be careful with this, it is possible to build incorrect key combinations and locks if you are branching and the player may spend the keys on a route that you did not plan and then can not move forward.

To place the hot-spot of the current screen, simply **we click the right mouse button in the box where we want the key or object to appear**, which will show a small dialog box where we enter the type. In this capture I am placing one of the keys *Lala Prologue*.



With patience we will screen at placing the hot-spot, indicating the value 1 in the places where should leave an object (the boxes in *Dogmole*) and the value 2 in the sites where it should leave a key.

Generating the code

Once we have finished placing everything, as we said before, we will have to generate the code that the Churrera Maker needs. Do you remember the super table of values we mentioned at the beginning of the chapter? Well, that. When you're done, then press E and records the code as **enems.h**. Copy this file to \dev. If you're curious, you can open it in the text editor and see how many numbers you've saved to write.

And we're done!

In addition, with more emotion: in the next issue, we will compile for the first time our game, even without scripting, to see how it is going.

Workshop creates your own game of Spectrum (Chapter 7)

We arrived at the seventh part of the Workshop "Create your own game of Spectrum" with the Churrera Maker of the Mojon Twins. Finally, we can compile our game but before we have to put the batteries because this chapter requires maximum attention. Ready? Let's go there.

Chapter 7: First Assembly

First of all, download the package of materials corresponding to this chapter by clicking on this link:

<http://www.mojontwins.com/churrera/churreratut-capitulo7.zip>

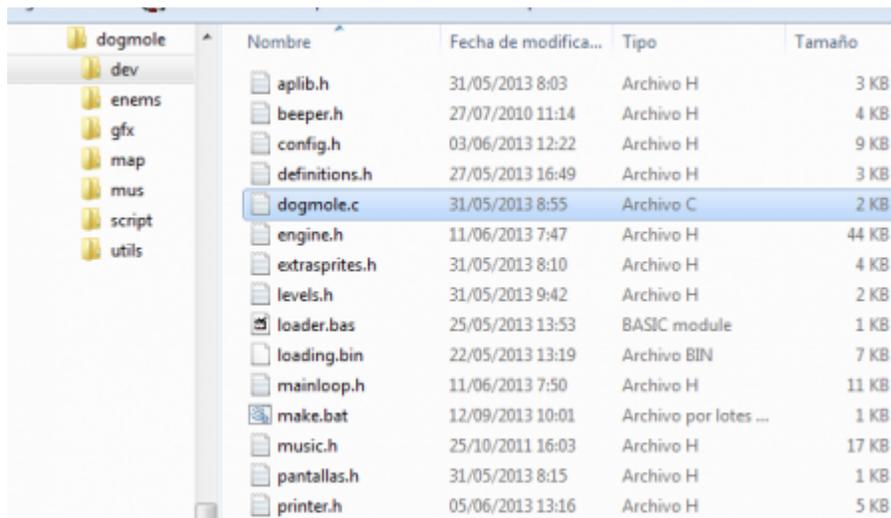
Come now. I want to see things moving!



That's where we are. This chapter is going to be deeper. Get ready to absorb what tons of information. To follow this chapter we recommend that you eat a good Tico-Tico watermelon, which strengthens the brain.

Let's start off easy, let's first make a kind of recap to see that we have everything we need.

First of all, the first thing we should have done is to unzip the package from the Churrera Maker and custom the files as explained at the beginning of this tutorial. With this, we should have something like this:

A screenshot of a Windows file explorer window showing a directory structure. The left pane shows a tree view with folders: dogmole, dev, enems, gfx, map, mus, script, and utils. The right pane shows a list of files with columns for Nombre, Fecha de modifica..., Tipo, and Tamaño. The file 'dogmole.c' is selected and highlighted in blue.

Nombre	Fecha de modifica...	Tipo	Tamaño
aplib.h	31/05/2013 8:03	Archivo H	3 KB
beeper.h	27/07/2010 11:14	Archivo H	4 KB
config.h	03/06/2013 12:22	Archivo H	9 KB
definitions.h	27/05/2013 16:49	Archivo H	3 KB
dogmole.c	31/05/2013 8:55	Archivo C	2 KB
engine.h	11/06/2013 7:47	Archivo H	44 KB
extrasprites.h	31/05/2013 8:10	Archivo H	4 KB
levels.h	31/05/2013 9:42	Archivo H	2 KB
loader.bas	25/05/2013 13:53	BASIC module	1 KB
loading.bin	22/05/2013 13:19	Archivo BIN	7 KB
mainloop.h	11/06/2013 7:50	Archivo H	11 KB
make.bat	12/09/2013 10:01	Archivo por lotes ...	1 KB
music.h	25/10/2011 16:03	Archivo H	17 KB
pantallas.h	31/05/2013 8:15	Archivo H	1 KB
printer.h	05/06/2013 13:16	Archivo H	5 KB

On this we should have copied all the files that we have been elaborating, converting and generating, namely:

```
The tileset, in the tileset.h file
The sprite-set, in the file sprites.h
The map, in the map file.h
```

The fixed screens, **title.bin**, **frame.bin** (if we are going to use it) and **ending.bin**

```
Enemies and hot-spots, in enems.h
```

Do we have everything? Insurance? All right. Then we can start to configure the engine to make our first compilation. Hold on, curves come.

The configuration file

If you look at our development folder **/dev**, in which you should already have all the resources prepared and ready to use, there is a file called **config.h**. Stick with your face: it is the most important file of all since it is the one that decides which pieces of the engine are assembled to form the game, and what parameter values are used in said pieces. You should have done it already, but just in case: open **/dev/config.h** in your favorite text editor. And now I show you how many things there are. Let's go by parts, explaining each section, what are values, how to interpret them, and filling them for our *Dogmole* example. Let's take some air and start.

General configuration

This section configures the general values of game: the size of the map, the number of objects (if applicable), and the like:

Map Size

```
#define MAP_W 8 //
#define MAP_H 3 // Map dimensions in screens
```

These two directives define the size of our map. If we remember, for our *Dogmole* the map measures 8×3 screens. We, therefore, fill in 8 and 3 (**W of Width, width, and H of Height, height**).

```
#define SCR_INITIO 16 // Initial screen
#define PLAYER_INI_X 1 //
#define PLAYER_INI_Y 7 // Initial tile coordinates
```

Start position

Here we define the position of the main character when starting a new game. **SCR_INICIO** defines which screen, and **PLAYER_INI_X** and **Y** define the coordinate of the tile where the character will appear. We started on the first screen of the third row, which (if we count or calculate) is the number 16 screen. We placed *Dogmole* on the ground and next to the rock wall in the coordinate box (1, 7) (Remember! Real developers start counting on 0).

End position

```
#define SCR_FIN 99 // Last screen. 99 = deactivated.  
#define PLAYER_FIN_X 99 //  
#define PLAYER_FIN_Y 99 // Player tile coordinates to finish game
```

Here we define the final position that we must reach to finish the game. It may interest us to make a game in which we simply have to get to a particular site to finish it. In that case, we would fill these values. As in the game that occupies us (and in none that we have done we have not used this never!), We are not going to use it, we put values out of range. Usually, the 99 is out of range (our map does not have as many screens, for example), and is a number that spring.

Number of objects

```
#define PLAYER_NUM_OBJETS 99 // Objects to get to finish game
```

Pay attention to this: this parameter defines the number of objects that we have to gather to finish the game. In simple gestures like *Lala Prologue*, counting objects and checking that we have all is automatic and uses this value: as soon as the player has that number of objects will be displayed the screen of the end. In our case, not: we're going to use scripting to handle the objects and the checks that we've done everything we had to do to win the game, so we're not going to need this at all. Therefore, we will put a value out of range, our beloved, so that the engine ignores the automatic object count. If you are doing a game on your own in which you simply have to collect all the objects, as in so many that we have launched, places here the maximum number of necessary objects.

Initial life and recharge value

```
#define PLAYER_LIFE 15 // Max and starting life gauge.  
#define PLAYER_REFILL 1 // Life recharge
```

Here we define what the character's initial life value will be and how much it will increase when we take an extra life. In *Dogmole*, we will start with 15 lives and recharge 1 in 1 to find hearts. As we are going to configure the engine so that a collision with an enemy causes us to blink, lives will be lost little by little. In shots like *Lala Prologue*, where collisions produce rebounds and the enemy can hit us many times, "lives" are considered "energy" and higher values such as 99 and more generous recharges are used, 10 or 25.

Multi-level games

```
// # define COMPRESSED_LEVELS // use levels.h instead of map.h and enems.h  
// # define MAX_LEVELS 2 // # of compressed levels  
// # define REFILL_ME // If defined, refill player on each level
```

The Churrera Maker is able to handle several compressed levels, in principle in low RAM, but with the possibility of admitting a slight modification to use extra pages of RAM in the 128K models. This is an advanced feature that we will explain in future chapters.

Engine Type

Now we start with the good: the configuration of the parts of the code that will be assembled as a multiple transformers to form our spawn. This is where all the fun is. It's fun because we can experiment using strange combinations (we've done it on Covertape # 2) and testing the effects. As we have already mentioned, it is impossible to activate everything, not only because there are things that cancel each other out, but because, simply, it would not fit.

Here we have two types of parameters: those that take a number, such as we have already seen, and those that can be active or inactive. To deactivate a capacity, simply comment. In C the comments are put with two bars //. When you see a line with two bars at the beginning, it is commented out and, therefore, the characteristic that describes is deactivated.

Let's go in parts, as Victor Frankenstein said...

Collision Box Size

```
#define BOUNDING_BOX_8_BOTTOM // 8x8 aligned to bottom center in 16x16
// # define BOUNDING_BOX_8_CENTERED // 8x8 aligned to center in 16x16
// # define SMALL_COLLISION // 8x8 centered collision instead of 12x12
```

The collision box refers to the square that "really" occupies our sprite. To understand us, our sprite will hit the stage. This collision is calculated with an imaginary square that can have two sizes: 16×16 or 8×8. The engine simply checks that square does not get into an obstacle block.

For you to understand the difference, see how Lala interacts with the stage in *Lala Prologue* (which has a collision box of 16×16 that occupies the whole sprite) and in *Lala Lah* (in which we use an 8×8 collision box, smaller than The sprite). The 8×8 collision has been one of the additions to this special version 3.99b of the Churrera Maker, and, in our opinion, makes the control more natural. Anyway, we have let the developer (i.e, you), choose if you prefer the old collision of 16×16, which may be more appropriate for depends on what situations.

If we choose an 8×8 collision with the scenario, we have two options: that the box is centered in the sprite or that is in the lower part:



The first option (centered box) is designed for top view games, like *Balowwwn* or *D'Veel'Ng*. The second works well with side-view games or top-looking "with a bit of perspective" games, like *Mega Meghan*.

Only one of the two directives can be active (because they are exclusive): if we want centered 8×8 collision we activate **BOUNDING_BOX_8_CENTERED** and deactivate the other one. If we want 8×8 collision in the lower part we activate **BOUNDING_BOX_8_BOTTOM** and deactivate the other one. If we want 16×16 collision we deactivate both.

The third directive refers to collisions against enemies. If we activate **SMALL_COLLISION**, the sprites will have to touch us much more to give us. With **SMALL_COLLISION**, enemies are easier to dodge. It works well in fast-moving games such as *Bootee*. We are going to leave it disabled for *Dogmole*.

General Directives

```
#define PLAYER_AUTO_CHANGE_SCREEN // Player changes screen automatically
```

If we define this, the player will change screen when he / she leaves the edge. If it is not defined, you must be pressing the specific address for this to happen. Normally it is left activated, so that if the player moves only by inertia also change of screen. We can not think of a situation where this is not desirable, but anyway there is the option to disable it.

```
// # define PLAYER_PUSH_BOXES // If defined, tile # 14 is pushable  
// # define FIRE_TO_PUSH
```

These two directives activate and configure the pushable blocks. We are not going to use pushpins in *Dogmole*, so we disable them. Activating the first one (**PLAYER_PUSH_BOXES**) activates the blocks, so tiles # 14 (with behavior type 10, remember) can be pushed.

The second directive, **FIRE_TO_PUSH**, is to define whether the player must press fire in addition to the direction in which he pushes or not. In *Cheril Perils*, for example, you do not have to press fire: just touching the block while we move will move. In *D'Veel'Ng*, it is necessary to press fire to push a block. If you are going to use pushbutton blocks, you have to decide which option you like best (and best suits the type of gameplay you want to get).

```
#define DIRECT_TO_PLAY // If defined, title screen = game frame.
```

This directive is activated to achieve what we said when we were doing the fixed screens: that the title of the game also serves as a frame. If you have a separate **title.bin** and a **frame.bin**, you should disable it. In our case, where we only have a **title.bin** that also includes frame, we leave it activated.

```
// # define DEACTIVATE_KEYS // If defined, keys are not present.  
// # define DEACTIVATE_OBJECTS // If defined, objects are not present.
```

These two directives are used to disable keys or objects. If your game does not use keys and locks, you must activate **DEACTIVATE_KEYS**. If you are not going to use objects, we activate **DEACTIVATE_OBJECTS**.

Surely some of you will be thinking why we do not activate **DEACTIVATE_OBJECTS** in *Dogmole*, if we said that the objects we will control by scripting? Good question! It is simple: what we will control by scripting is the count of objects and the final condition, but we need the engine to manage the collection and placement of objects.

What a mess, right? Patience.

```
#define ONLY_ONE_OBJECT // If defined, only one object can be carried  
#define OBJECT_COUNT 1 // Defines FLAG to be used to store object #
```

We continue with two directives with a very specific application: if we activate the first, **ONLY_ONE_OBJECT**, we can only carry an object. Once you pick up an object, the collection of objects is blocked and you can not catch any more. To re-activate the collection of objects we will have to use scripting. With this we get the effect we need to take the boxes one by one: we set the engine to only allow us to carry an object (a box), and then, when we do the script, we will do that when we take the box to Site where you have to go depositing (a specific site of the University) to activate the collection of objects so that we can go to the next box.

The second directive, **OBJECT_COUNT**, is used to display the value of one of the flags of the scripting system in the object marker instead of the internal account of collected objects. We'll see it in the future, when we explain the scripting engine, but the scripts have up to 32 variables or flags that we can use to store values and perform checks. Each variable has a number. If we define this directive, the engine will display the value of the flag indicated in the marker counter. From the script we will increase this value each time the player arrives at the University and deposits an object.

Briefly, we only need to define **OBJECT_COUNT** if we are the ones that are going to take the account, by hand, from the script. If we are not going to use scripting, or we will not need to manually control the number of objects collected, we will have to comment on this directive so that it is not taken into account.

```
// #define DEACTIVATE_EVIL_TILE // If defined, no killing tiles are detected.
```

Uncomment this directive if you want to disable the tiles that kill you (type 1). If you do not use type 1 tiles in your game, uncomment this line and save space, as this way the detection of matting tiles will not be included in the code.

```
// # define PLAYER_BOUNCES // If defined, collisions make player bounce
// # define FULL_BOUNCE
// # define SLOW_DRAIN // Works with bounces. Drain is 4 times slower
#define PLAYER_FLICKERS // If defined, collisions make player flicker
```

These two directives control rebounds. If you activate **PLAYER_BOUNCES**, the player will bounce against enemies by touching them. The strength of this rebound is controlled by **FULL_BOUNCE**: if activated, the rebounds will be much more beasts because the speed will be used with which the player originally advanced, but in the opposite direction. Disabling **FULL_BOUNCE** the rebound is half the speed.

If we define, in addition, **SLOW_DRAIN**, the speed at which we lose energy if we get stuck in the path of the enemy (remember *Lala Prologue*, *Sir Ababol* or the original version of *Journey to the Center of the Napia*) will be four times smaller. This is used in *Bootee*, where it is easy to get trapped in the path of an enemy and complicated to leave it. This makes the game more affordable.

As you can imagine, **FULL_BOUNCE** and **SLOW_DRAIN** depend on **PLAYER_BOUNCES**. If **PLAYER_BOUNCES** is disabled, the other two directives are ignored.

Activating **PLAYER_FLICKERS** allows the character to blink if he collides with an enemy (being invulnerable for a short period of time). Usually we will choose between **PLAYER_BOUNCES** and **PLAYER_FLICKERS**, but they can work together. We in *Dogmole* want the protagonist to only blink when he collides with an enemy, so we deactivate **PLAYER_BOUNCES** and activate **PLAYER_FLICKERS**.

```
// # define MAP_BOTTOM_KILLS // If defined, exiting the map bottom kills.
```

Disable this directive if you want to, in case the character is going to leave the map below, the engine will bounce and subtract life, as in *Zombie Skull*. If your map is closed from below, turn it off to gain a few bytes.

```
// # define WALLS_STOP_ENEMIES // If defined ... erm ...
```

This directive is related to pushable tiles. If you define pushing blocks, you may want enemies to react to them and alter their trajectories, as in, for example, Monono or Cheril of the Forest (among many others). If you do not use pushy tiles or you do not want them to go through them, turn it off to gain a lot of space.

Types of extra enemies

Let's now see a set of directives that will be used to activate enemies of types 5, 6 or 7. Remember what we mentioned about this type of enemies: 5 are the bats *Zombie Skull*, 7 are the enemies who are after you Of Mega Meghan, and the 6 is reserved for us to implement a new type of enemies.

In *Dogmole* we do not use any kind of extra enemy. For your games, you can experiment with these bugs, or you can wait for the end of the tutorial when explaining how to implement a type of custom enemy for type 6.

```
// # define ENABLE_RANDOM_RESPAWN // If defined, flying enemies
// # define FANTY_MAX_V 256 // Flying enemies max speed.
// # define FANTY_A 12 // Flying enemies acceleration.
// # define FANTIES_LIFE_GAUGE 10 // Amount of shots needed to kill.
```

Activating **ENABLE_RANDOM_RESPAWN** activates type 5 enemies. These enemies appear off the screen if there is any type 5 enemy or a dead enemy, and they chase the player unless he is touching a **type 2 tile** (is hidden). The following directives are used to configure their behavior:

FANTY_MAX_V sets the maximum speed. To give you an idea, divide the value between 64 and the result is the number of pixels that will advance for each frame as a maximum. If we set 256, the flying enemy can accelerate to 4 pixels per frame.

FANTY_A is the acceleration value. Each game box, the speed will increase by the value indicated in direction towards the player, if it is not hidden. The lower this value, the longer the enemy will react to a change of player's direction.

FANTIES_LIFE_GAUGE defines how many shots the character must receive to die, if we have activated the shots (see below).

```
// # define ENABLE_PURSUERS // If defined, type 7 enemies are active
// # define DEATH_COUNT_EXPRESSION 8+(rand () & 15)
```

Activating **ENABLE_PURSUERS** activates type 7 enemies. These enemies appear where we have placed them (remember the chapter of the enemies) and chase the player. They stop with stage obstacles, unlike Type 5 enemies.

If we have the firing engine activated, when we kill an enemy of type 7 it will take some time to re-exit. This time, expressed in number of frames, is calculated using the expression defined in **DEATH_COUNT_EXPRESSION**. The one that is seen in the example is the one that is used in Mega Meghan: 8 plus a random number between 0 and 15 (that is, between 8 and 23 frames).

Shooting engine

The firing engine is quite expensive in terms of memory. Activate it includes quite a few pieces of code, since you have to check more collisions and take into account many things besides the extra sprites needed.

```
// # define PLAYER_CAN_FIRE // If defined, shooting engine is enabled.
```

If we activate this directive, we are including the shots engine. The following directives are used to configure their behavior.

```
// # define PLAYER_BULLET_SPEED 8 // Pixels / frame.  
// # define MAX_BULLETS 3 // Max number of bullets on screen.
```

The **PLAYER_BULLET_SPEED** directive controls the speed of the bullets. 8 pixels per frame is a good value and is the one we have used in all games. A higher value can cause collisions to be lost, since everything on the screen is discrete (not continuous) and if an enemy moves quickly in the opposite direction of a bullet that moves too fast, it is possible that from frame to frame they cross without colliding. If you think about it a bit and imagine the game in slow motion as a succession of frames you will see.

The **MAX_BULLETS** value controls the maximum number of bullets that can be displayed. Be careful with this, I am already seeing the intentions to upload it: in theory we could have more than three bullets, but to do that we would have to modify the part of the engine that reserves the memory that the sprites system needs. Each moving sprite spends a lot of memory (14 bytes per block, and a 8x8 sprite uses four blocks), so the reserve is just. The truth is that everything is tight to the max and can not even the hair of a shrimp, but by the end of the tutorial explain how to modify the number of blocks that are reserved for the sprites so that we can have more bullets.

Why let us define it, then? Well to be able to put less. Being able to fire only one bullet can come in handy for some games, for example.

```
// # define PLAYER_BULLET_Y_OFFSET 6 // vertical offset from the player's top.  
// # define PLAYER_BULLET_X_OFFSET 0 // horz offset from the player's left / right.
```

These two directives define where bullets appear when fired. The behavior of these values is relatively complex (good, not so much) and changes according to the view:

If the game is in side view, we can only shoot left or right. In that case, **PLAYER_BULLET_Y_OFFSET** defines the height, in pixels, at which the bullet will appear counting from the top of the character sprite. This serves to adjust so that they leave the gun or wherever we want. **PLAYER_BULLET_X_OFFSET** is ignored completely.

If the game is in top view, the behavior is the same as the one described if we look left or right, but if we look up or down the bullet will appear shifted laterally **PLAYER_BULLET_X_OFFSET** pixels from the left if we look down or from the Right if we look up. This means that our character is right-handed. Look at Mega Meghan's sprites. To make left-handed players, you have to change two lines of the engine. If you are super interested and your life depends on it, write to us and we will tell you, friend Flanders.

```
// # define ENEMIES_LIFE_GAUGE 4 // Amount of shots needed to kill enemies.  
// # define RESPAWN_ON_ENTER // Enemies respawn when entering screen
```

These are used to control what happens when bullets hit normal enemies (type 1, 2 or 3). First, **ENEMIES_LIFE_GAUGE** defines the number of shots that must be taken to die. If we activate **RESPAWN_ON_ENTER**, the enemies will be resurrected if we leave the screen and re-enter. As in classical games, *Illo*.

By the way, the killed enemies are counted and this value can be controlled from the script.

Scripting

The following directives are used to activate the scripting engine and define a couple of things related to it. For now we will leave not activated, so we can compile and test the game without having to make a script. Because we're already in the mood, right? Calm down, we'll get back to them when we start getting hardcore.

```
// # define ACTIVATE_SCRIPTING // Activates msc scripting and flags.  
#define SCRIPTING_DOWN // Use DOWN as the action key.  
// # define SCRIPTING_KEY_M // Use M as the action key instead.
```

The first is simple: if we enable **ACTIVATE_SCRIPTING**, the entire scripting system will be included in the engine. The other two define what will be the action key: down or fire. Only one of the two can be activated. We will use below to perform the actions, so we activate **SCRIPTING_DOWN**.

As I told you, for now we leave **ACTIVATE_SCRIPTING** disabled. We will activate it when we begin to make our script.

Directives related to the top view

```
// # define PLAYER_MOGGY_STYLE // Enable top view.  
// # define TOP_OVER_SIDE // UP / DOWN has priority over LEFT / RIGHT
```

If we activate **PLAYER_MOGGY_STYLE**, the game will be of top view. If not activated, the game will be side view. For *Dogmole* we left it disabled, therefore.

The next, **TOP_OVER_SIDE**, defines the behavior of the diagonals. This is useful especially if your game also has shots. If you define **TOP_OVER_SIDE**, when moving diagonally the puppet will look up or down and will therefore shoot in that direction. If it is not defined, the puppet will look and shoot left or right. Depending on the type of game or the configuration of the map you will be more interested in either option. No, you can not shoot diagonally.

Directives related to side view

Here are a lot of things:

```
#define PLAYER_HAS_JUMP // If defined, player is able to jump.
```

If this is defined, the player can jump. If the shots have not been activated, fire will cause the player to jump. If activated, the "up" key will be activated.

```
// # define PLAYER_HAS_JETPAC // If defined, player can thrust a jetpac
```

If we define **PLAYER_HAS_JETPAC**, the "up" key will activate a jetpac. It is compatible with being able to jump. However, if you activate the jump and the jetpac at the same time you will not be able to use the shots ... although we have not tried it, maybe something strange happens. Do not do it. Or, I do not know, do it. If you do not know what this is, play *Cheril the Goddess* or *Jet Paco*.

```
#define PLAYER_KILLS_ENEMIES // If defined, stepping on enemies kills them  
#define PLAYER_MIN_KILLABLE 3 // Only kill id >= PLAYER_MIN_KILLABLE
```

They activate the trample engine. With **PLAYER_KILLS_ENEMIES** activated, the player can jump over enemies to kill them. **PLAYER_MIN_KILLABLE** helps us not to kill all enemies. In *Dogmole*, we can only kill the sorcerers, who are type 3. Eye with this: if we put a 1 we can kill all, if we put a 2, the enemies type 2 and 3, and if we put a 3 only to the Of type 3. In other words, it will be possible to kill the enemies whose type is greater than or equal to the value that is configured.

```
// # define PLAYER_BOOTEE // Always jumping engine.
```

This directive activates the continuous jump (see *Bootee*). It does not support **PLAYER_HAS_JUMP** or **PLAYER_HAS_JETPAC**. If you activate **PLAYER_BOOTEE**, you have to disable the other two. If not, crawl.

```
// # define PLAYER_BOUNCE_WITH_WALLS // Bounce when hitting a wall.
```

The player bounces against the walls, as in *Balowwwn*. This also works in top view (in fact it was programmed specifically for *Balowwwn* which is in top view), I do not know why we have put it in this section. Oh, I do not know.

```
// # define PLAYER_CUMULATIVE_JUMP // Keep pressing JUMP to JUMP higher
```

This works in conjunction with **PLAYER_HAS_JUMP**. If it is defined, when you press the jump key we will start jumping a little and every time we go bouncing we will gain more and more altitude, as in *Monono*.

Display Settings

In this section we place all the elements on the screen. Do you remember when we were designing the frame? For this is where we are going to put all the values that we pointed out, namely:

```
#define VIEWPORT_X 1 //  
#define VIEWPORT_Y 0 // Viewport character coordinates
```

They define the position (always in character coordinates) of the playing area. Our play area will start at (0, 1), and those are the values we give to **VIEWPORT_X** and **VIEWPORT_Y**.

```
#define LIFE_X 22 //  
#define LIFE_Y 21 // Life gauge counter character coordinates
```

They define the position of the life marker (from the numerical, go).

```
#define OBJECTS_X 17 //  
#define OBJECTS_Y 21 // Objects counter character coordinates
```

They define the position of the object counter, if we use objects (the numerical position).

```
#define OBJECTS_ICON_X 15 // Objects icon character coordinates  
#define OBJECTS_ICON_Y 21 // (use with ONLY_ONE_OBJECT)
```

These two are used with **ONLY_ONE_OBJECT**: When we "carry" the object above, the engine will indicate us by blinking the icon of the object in the marker. In **OBJECTS_ICON_X** and **Y** we indicate where this icon appears (the tile with the box's drawing). As you will see, this forces us to be using the icon on the marker, not a text or something else.

Yes, it is a limitation.

Graphic effects

In this section we define several graphic effects (very basic) that we can activate and that will control the way in which the game is shown. Basically we can configure the Churrera Maker to paint shadows or not when building the stage, and define a couple of things related to sprites and such.

```
// # define USE_AUTO_SHADOWS // Automatic shadows made of darker attributes  
// # define USE_AUTO_TILE_SHADOWS // Automatic shadows using tiles 32-47.
```

These two take care of the shadows of the tiles, and we can activate only one of them, or none. If you remember, in the chapter of the tileset we speak of automatic shadows. If we enable **USE_AUTO_SHADOWS**, the obstacle tiles draw shadows on the trashable tiles using only attributes (sometimes results, but not always).

USE_AUTO_TILE_SHADOWS uses shaded versions of the background tiles to make the shadows, as explained. Disabling both will not draw shadows.

In *Dogmole* we will not use shadows of any kind, because the attributes we do not like and the tiles we can not afford because we will use these tiles to embellish some screens by printing graphics through the script.

```
/ # define UNPACKED_MAP // Full, uncompressed maps.
```

If we define **UNPACKED_MAP** we will be telling the engine that our map is 48 tiles.

```
// #define NO_MASKS // Sprites are rendered using OR
// # define PLAYER_ALTERNATE_ANIMATION // If defined, animation is 1,2,3,1,2,3 ...
```

These two directives were designed specifically for *Zombie Skull*. The first one (NO_MASKS) causes the sprites not to be masked, which saves memory. As *Zombie Skull* did not need masks, we were able to get a bunch of bytes for other things. However, the sprites converter that is currently in the Churrera Maker is not able to export sprite-sets without masks, so for now this directive will not serve you much. We will consider telling Amador, the Programmer Monkey, to update the sprites converter to be able to remove sprite-sets without masks if there is enough demand.

PLAYER_ALTERNATE_ANIMATION may suit you, as it changes the order in which the character is animated when he walks. Normally, the animation is 2, 1, 2, 3, 2, 1, 2, 3... But if you activate this directive it will be 1, 2, 3, 1, 2, 3,...

Setting the main character's movement

In this section we will configure how our player will move. It is very likely that the values you put here have to be adjusted by the trial and error method. If you have some basics of physics, you'll be fine to know how each parameter affects.

Basically, we will have what is known as Uniformly Accelerated Rectilinear Motion (MRUA) on each axis: the horizontal and the vertical. Basically we will have a position (say X) that will be affected by a velocity (say VX), which in turn will be affected by an acceleration (i.e., AX).

The following parameters are used to specify various values related to the movement on each axis in lateral view games. In the top view, the values of the horizontal axis will also be taken for the vertical.

To get the smoothness of movements without using decimal values (which are very expensive to handle), we use fixed-point arithmetic. Basically the values are expressed in 1/64 pixel. This means that the value used is divided by 64 when moving the actual sprites to the screen. That gives us an accuracy of 1/64 pixels on both axes, which translates into greater smoothness of movement.

We are aware that this section is deeper, so do not worry too much if, from the outset, you get lost a bit. Experiment with the values until you find the ideal combination for your game.

Vertical axis in lateral view games

Vertical movement is affected by gravity. The vertical velocity will be increased by gravity until the character lands on a platform or obstacle. In addition, when jumping, there will be an initial impulse and an acceleration of the jump, which we will also define here. These are the values for *Dogmole*; We will then detail each of them:

```

#define PLAYER_MAX_VY_CAYENDO 512 // Max falling speed
#define PLAYER_G 48 // Gravity acceleration
#define PLAYER_VY_INICIAL_SALTO 96 // Initial jump velocity
#define PLAYER_MAX_VY_SALTANDO 312 // Max jump velocity
#define PLAYER_INCR_SALTO 48 // acceleration while JUMP is pressed
#define PLAYER_INCR_JETPAC 32 // Vertical jetpac gauge
#define PLAYER_MAX_VY_JETPAC 256 // Max vertical jetpac speed

```

Free Fall: The player's speed, measured in pixels / frame will be incremented by **PLAYER_G** / 64 pixels / frame \wedge 2 until it reaches the maximum specified by **PLAYER_MAX_CAYENDO** / 64. With the values we have chosen for *Dogmole*, the vertical free fall rate will be increased by $48/64 = 0.75$ pixels / frame until it reaches a value of $512/64 = 8$ pixels / frame. That is, *Dogmole* will fall faster and faster until it reaches the maximum speed of 8 pixels per frame. Increasing **PLAYER_G** will reach the maximum speed much sooner. These values affect the jump: the more gravity, the less we jump and the less the initial impulse lasts. By modifying **PLAYER_MAX_CAYENDO** we can get the maximum speed, which will be reached before or after depending on **PLAYER_G**, be greater or less. Using small values we can simulate low-gravity environments such as outer space, the surface of the moon, or the sea floor. The value of 512 (equivalent to 8 pixels per frame) can be considered the maximum since higher values and very long falls could result in glitches and rare things.

Jump: The jumps are controlled using three parameters. The first, **PLAYER_VY_INICIAL_SALTO**, will be the value of the initial impulse: when the player presses the jump key, the vertical velocity will automatically take the specified value upwards. While the jump is being pressed, and for about eight frames, the speed will continue to increase by the value specified by **PLAYER_INCR_SALTO** until we reach the value **PLAYER_MAX_VY_SALTANDO**. This is done so that we can control the force of the jump by pressing the jump key more or less time. The acceleration period, which lasts for 8 frames, is fixed and can not be changed (to change it would have to touch the engine), but we can get sharper jumps by raising the value of **PLAYER_INCR_SALTO** and **PLAYER_MAX_VY_SALTANDO**.

Usually finding the ideal values requires a bit of trial and error. Keep in mind that jumping horizontally from one platform to another also comes into play the values of horizontal movement, so if you have decided that in your game the game should be able to overcome distances of X tiles you will have to find the optimal combination by playing With all parameters.

The remaining two values are not used in *Dogmole* because they have to do with the *Jet Paco*. The first is the acceleration that occurs while pressing the up key and the second the maximum value that can be reached. If your game does not use jetpac these values are not used at all.

Horizontal axis in lateral view / general behavior in top view

The following set of parameters describes the behavior of the movement on the horizontal axis if your game is in lateral view, or those of both axes if your game is in genital view. These parameters are much simpler:

```

#define PLAYER_MAX_VX 256 // Max velocity
#define PLAYER_AX 48 // Acceleration
#define PLAYER_RX 64 // Friction

```

The first, **PLAYER_MAX_VX**, indicates the maximum speed at which the character will move horizontally (or in any direction if the game is in genital view). The larger this number, the more it will run, and the farther it will come when it jumps (horizontally). A value of 256 means that the character will run to a maximum of 4 pixels per frame.

PLAYER_AX controls the acceleration of the character while the player presses an arrow key. The higher the value, the faster the maximum speed will be reached. Small values make it "hard to boot". A value of 48 means it will take approximately 6 frames (256/48) to reach the maximum speed.

PLAYER_RX is the value of friction or friction. When the player stops pressing the movement key, this acceleration is applied in the direction opposite to the movement. The higher the value, the character will stop beforehand. A value of 64 means it will take 4 frames to stop if it was at full speed.

Using small values of **PLAYER_AX** and **PLAYER_RX** will make the character appear to slip. It is what happens in games like, for example, *Journey to the Center of the Napia*. Except for mysterious exceptions, it almost always "burns better" if the value of **PLAYER_AX** is greater than that of **PLAYER_RX**.

Behavior of tiles

Do you remember that we explained how each tile has a type of associated behavior? Tiles that kill are obstacles, platforms, or intractable. In this section (the last, finally) of **config.h** we define the behavior of each of the 48 tiles of the complete tileset.

We will have to define the behaviors of the 48 tiles regardless of whether our tile uses a 16 tile. Usually, in all cases, tiles 16 through 47 will have a type "0", but we may need other values if we use The extra tiles to paint graphics and ornaments from the script, as we do. In *Dogmole* we are not going to put any obstacle "outside of tileset", but games like those of the saga of *Ramiro the Vampiro* yes that they have obstacles of this type.

To put the values, simply open the tileset and look, they are in the same order in which the tiles appear in the tileset:



```
Unsigned char behavior_tiles [ ] =
{
0, 8, 8, 8, 8, 8, 0, 8, 4, 0, 8, 1, 0, 0,10,10,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
};
```

As we see, we have the first empty tile (**type 0**), then we have five rock tiles that are obstacles (**type 8**), another bottom tile (**column type 0**), brick tile **type 8**, (Type 1), three bottom tiles (**type 0**), and the lock tile (**type 10**), tiles (type 4), tiles (**type 4**, platform and **type 0**, intractable).

When you modify these values be careful with the commas and that you are not dancing any number.

Arf, arf, arf!

Ah, you still there? I thought I'd already gotten bored and you could not take it anymore. Well, I see that you are a constant type and with much power. Yes, we have already finished setting our game. Now it comes when we compile it. Be patient with this, especially if you're not used to fighting with a compiler in a command line window.

Let's set up our make.bat compilation script to make it very simple. The compilation script will also be in charge of regenerating the map every time, which comes in handy if we are constantly modifying it (for example, during the testing stage) since the process is quite automated.

Preparing our compilation script

Let's open make.bat with our text editor to change some things. Right now, if you listened to me in the first chapter, you should have about a pint similar to the one that appears in the box:

```
@echo off
Rem cd .. \ script
Rem msc dogmole.spt msc.h 24
Rem copy * .h .. \ dev
Rem cd .. \ dev
Cd .. \ map
.. \ utils \ mapcnv map.map 8 3 15 10 15 packed
Copy map.h .. \ dev
Cd .. dev
Zcc + zx -vn dogmole.c -o dogmole.bin -ldos -lsplib2 -zorg
= 25000
.. \ utils \ bas2tap -a10 -sLOADER loader.bas loader.tap
.. \ utils \ bin2tap -o screen.tap -a 16384 loading.bin
.. \ utils \ bin2tap -o main.tap -a 25000 dogmole.bin
Copy / b loader.tap + screen.tap + main.tap dogmole.tap
of loader.tap
of the screen.tap
from main.tap
from dogmole.bin
echo DONE
```

Be sure to put "dogmole. *" (Or the name of your game) on all sites that are in bold and italic. That's the first thing. Also, make sure that you have not forgotten to rename the **churromain.ca dogmole.c** (or the name of your game).

Let's see what each line does because if you are doing your own game you will want to change things.

First, we see that there are four lines that begin with rem. These lines are commented and will not be executed. We will remove the rem when we begin to make our script, so, for now, we will move from them.

Then, the script changes to the directory of the map to re-generate **map.h** and copy it to **/dev**, just in case we have made some changes so that it is reflected automatically. This is one of the lines that you will have to change for your game, indicating the correct parameters of **mapenv** (in particular, the size of screens, or if you do not use locks to put 99 instead of 15, or if you use maps of 48 tiles for Remove packed):

```
Cd .. \ map
.. \ utils \ mapenv map.map 8 3 15 10 15 packed
Copy map.h .. \ dev
Cd .. \ dev
```

The following line compiles the game:

```
Zcc + zx -vn dogmole.c -o dogmole.bin -lndos -lsplib2 -zorg = 25000
```

This is nothing more than a z88dk compiler run which takes **dogmole.c** as its source (and all the .h files it contains), and outputs a binary in machine code **dogmole.bin**. The compilation address is 25000.

The script then creates three tapes in .tap format. The first contains a loader in BASIC (which you can modify if you get very geeky, the source is in **loader.bas**). The following contains the charging screen. Right now we are not going to stop at this, so your game will include the default loading screen of the Churrera Maker (which is in **loading.bin**). The last one contains the **dogmole.bin** we just compiled:

```
.. \ utils \ bas2tap -a10 -sLOADER loader.bas loader.tap
.. \ utils \ bin2tap -o screen.tap -a 16384 loading.bin
.. \ utils \ bin2tap -o main.tap -a 25000 dogmole.bin
```

```
.. \ utils \ bas2tap -a10 -sDOGMOLE loader.bas loader.tap
```

Look at the -sLOADER of the first line, which generates the BASIC charger. That is the name that appears after the Program: loading the tape into the Spectrum, so, if you want, you can change it for something else. Remember that Spectrum tape file names can be up to 10 characters in length. Let's change it so that DOGMOLE comes out:

When we already have the three tapes, each with a block (charger, screen, and game) the only thing left to do is to join them:

```
Copy / b loader.tap + screen.tap + main.tap dogmole.tap
```

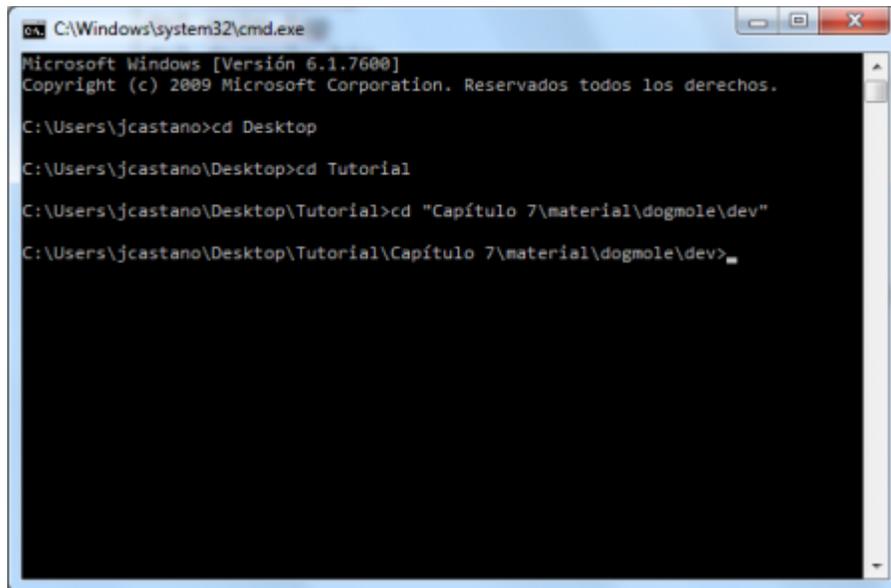
And, to finish, we do a little cleaning, eliminating the intermediate files that no longer serve us at all:

```
of loader.tap
of the screen.tap
from main.tap
from dogmole.bin
```

Write make.bat again with the changes. We are ready. Do we dare?

Compiling

The first thing we have to do is open a command line window and go to the `/dev` folder of our squeeze:

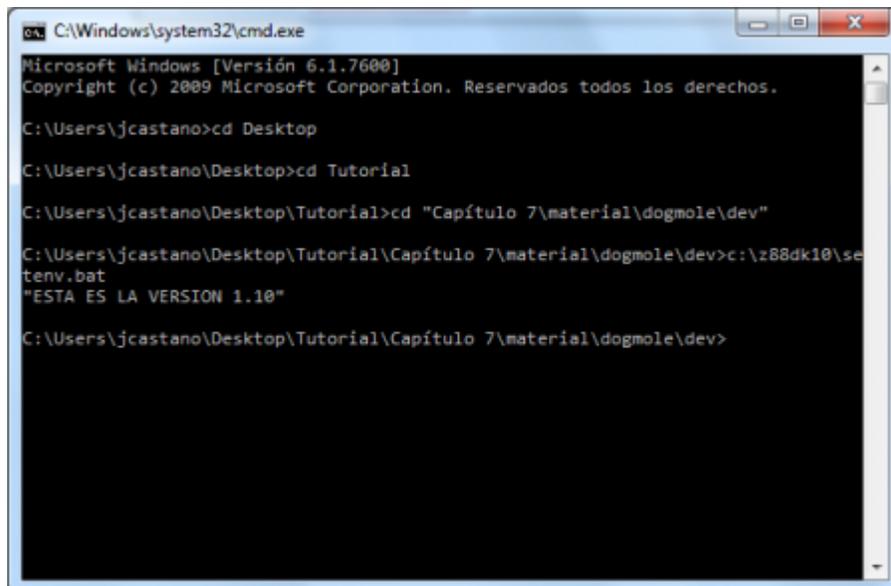


```
C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\jcastano>cd Desktop
C:\Users\jcastano\Desktop>cd Tutorial
C:\Users\jcastano\Desktop\Tutorial>cd "Capítulo 7\material\dogmole\dev"
C:\Users\jcastano\Desktop\Tutorial\Capítulo 7\material\dogmole\dev>
```

Once this is done, let's run a `z88dk` script that sets some environment variables. Remember that we installed it in `C:\z88dk10`. We execute this:

C:\z88dk10\setenv.bat



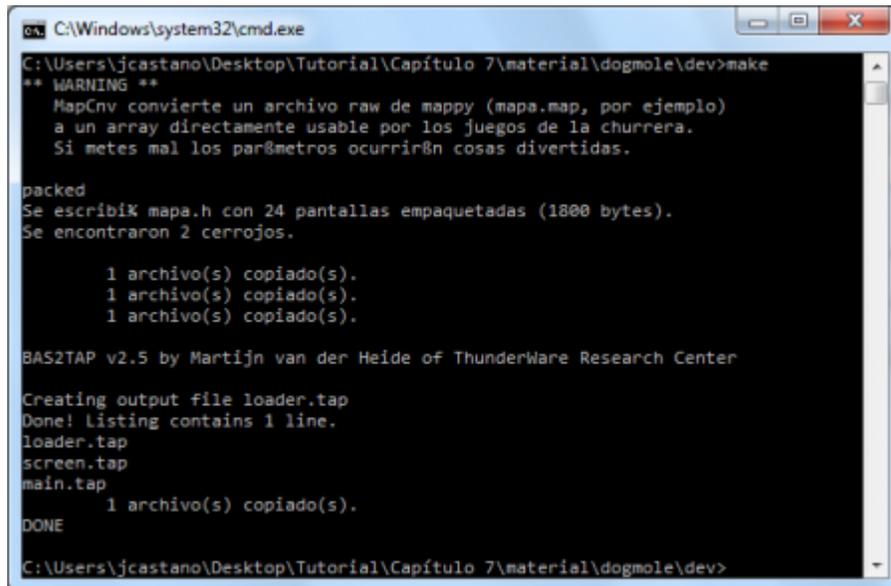
```
C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\jcastano>cd Desktop
C:\Users\jcastano\Desktop>cd Tutorial
C:\Users\jcastano\Desktop\Tutorial>cd "Capítulo 7\material\dogmole\dev"
C:\Users\jcastano\Desktop\Tutorial\Capítulo 7\material\dogmole\dev>c:\z88dk10\setenv.bat
"ESTA ES LA VERSION 1.10"
C:\Users\jcastano\Desktop\Tutorial\Capítulo 7\material\dogmole\dev>
```

Once we have done this (we will have to do it every time we open a command line window to compile a game), we can execute our script **make.bat**. Exit right or wrong, do not close the command line window, you will probably have to recompile a thousand times more (especially if you are adjusting the values of the movement or modifying the map to fix things).

make.bat

If we have done everything well, there should be no problem:



```
C:\Windows\system32\cmd.exe
C:\Users\jcastano\Desktop\Tutorial\Capítulo 7\material\dogmole\dev>make
** WARNING **
MapCnv convierte un archivo raw de mappy (mapa.map, por ejemplo)
a un array directamente usable por los juegos de la churrera.
Si metes mal los parámetros ocurrirán cosas divertidas.

packed
Se escribió mapa.h con 24 pantallas empaquetadas (1800 bytes).
Se encontraron 2 cerrojos.

    1 archivo(s) copiado(s).
    1 archivo(s) copiado(s).
    1 archivo(s) copiado(s).

BAS2TAP v2.5 by Martijn van der Heide of ThunderWare Research Center
Creating output file loader.tap
Done! Listing contains 1 line.
loader.tap
screen.tap
main.tap
    1 archivo(s) copiado(s).
DONE
C:\Users\jcastano\Desktop\Tutorial\Capítulo 7\material\dogmole\dev>
```

It has been DONE! We already have it! Now we have our dogmole.tap with the game ready to be tested. Of course it will not work at all, since we lack the script, but at least we can see that the puppets move, that we have not crap with the map, that the movement is well, that the sorcerers die when they are stepped on, and things like that. If there is something wrong, it is fixed, recompiled (by putting make again), and is tested again.

And since we already have a headache for you and me, we leave it until the next chapter. If you decided to start with something simple like *Lala Prologue* or *Sir Ababol*, you're done. If not, there are still many things to do.

Workshop creates your own game of Spectrum (Chapter 7B)

The Churrera Maker has been updated and the friends of Mojon Twins have been good to do a chapter B of 7 to explain the news that this version brings. From now on we will have more possibilities (if possible) for our own game of Spectrum. To enjoy!

The Churrera Maker 3.99.1

First of all, download the package of the new version of the Churrera Maker by clicking right here below:

<http://www.mojontwins.com/churrera/mt-churrera-3.99.1.zip>

But what hedge?

We interrupt our tutorial briefly before going to the path with the topic of scripting to introduce a new version of the Churrera Maker. We do not want you to be outdated: as we have introduced a lot of very interesting new things and, more importantly, we have corrected some bugs and optimized enough the existing code (so that you get more stuff in the games), we have decided not to We were going to follow the tutorial before you had the current version.

In this tutorial, in addition, we will learn something that will come very well for the future: to update the version of the Churrera Maker with the game started.

Is it mandatory to update? Well, you'll see. If you want to finish following the tutorial with the *Dogmole*, the truth is that you do not need it. But I recommend that you still update yourself.

Are you ready? Come on, let's start by saying what's new.

Click here to read the entire chapter.

What's New?

We are going to go in parts because the novelties spread through several areas. Hopefully, now you have half clear how everything works, and explaining these things will be easier.

Destructable Tiles

Initially (for now) geared to the games that carry the firing engine, we have defined a new type of tile, **type 16**. These tiles will break and disappear (will be replaced by tile number 0) after being hit by a certain number (Configurable) of shots. To include them, in addition to having tiles defined with type 16, we have to configure some things in **config.h**:

```
#define BREAKABLE_WALLS // Breakable walls
#define BREAKABLE_WALLS_LIFE 1 // Amount of hits to break wall
```

The first directive, **BREAKABLE_WALLS** enable this feature and includes all necessary to have destructible tiles (if not active, much as type 16 tiles have nothing will happen) code. The second, **BREAKABLE_WALLS_LIFE**, defines the number of shots that should be useful to break destructible. Obviously, you have to put a number greater than 0. If you put a 3, the tile will break to the third shot it receives.

Combination tile types

This is something we wanted to have put down from the start, but we have been leaving it. Planning to have tiles with combination types (which were both several things at once) is the real reason why the list of tile types has so many holes. Now you'll see why.

Basically, it is about combining tile types, so that a tile is at the same time, for example, a platform that kills you, or an obstacle that can break. To combine types of tiles, just add the types that we want to combine. Hence there are "holes" in the numbering.

For example, to make an obstacle that can be broken, we would have to make that tile had obstacle type (8) and destructible type (16), ie it was type $8 + 16 = 24$. In the last Game we have taken, that of Sgt. Helmet, no electric fences (killing) we can destroy. If you download the fonts (you should, now you're a developer churrero!) And look at the section types of tile (Behaviours) at the end of **config.h** see how the tile of the electrified fence has type 17, ie killing (1) and that is destructible (**16**): $16 + 1 = 17$.

There are many combinations stupid and make no sense, like putting an obstacle kills ($8 + 1 = 9$). In fact, it is one of those stupid and impossible combinations which we have used for special tiles (the blocks are pushed and locks), which are of type 10 ($8 + 2$ i.e, an obstacle that hides you).

The engine continues to support simple types (1, 2, 4 or 8), so no need to change anything in your *Dogmole* or your own Game.

Running out of Bullets

Now you can decide that the bullets run out. In addition, you can place ammo refills on the game. To do this, you must first touch config.h to enable and configure this capability:

```
#define MAX_AMMO 99 // If defined, ammo is not infinite!
#define AMMO_REFILL 50 // type 3 hot-spots refill master, using tile 20
// # define INITIAL_AMMO 0 // If defined, ammo = X when entering game.
```

The first directive, **MAX_AMMO**, if defined, causes the bullets run out and the maximum value is specified. If you want infinite bullets, simply comment on this definition. The next, **AMMO_REFILL** indicates how many bullets we caught when we pick up a refill of ammunition. The third, **INITIAL_AMMO**, if defined, causes the beginning of Game have the specified number of bullets. If not defined, the number of bullets to start will be the maximum, that is, who says in **MAX_AMMO**.

As out there (configuration is Sgt. Helmet), the maximum is 99 bullets in 50 refills we will take time, and start with the maximum, i.e., 99.

To place the recharges we will do it with the setter, and using the hot-spots. So far we had used type 1 hot-spots for objects and type 2 hot-spots for the keys. To place refills, we will need hot-spots type 4. In addition, in our tileset, tile number 20 must be the one corresponding to the recharge. This is the tileset Sgt. Helmet. Notice how tile number 20 is an ammunition refill:



Improvements to type 7 enemies

We put a couple of things for the type 7 enemies. So far, the **type 7** enemies came out of the place that you defined in the setter and were told to chase you. The associated graph was chosen at random among the four sprite-set enemies. Now we allow you to decide what chart you have, and let that be always. In addition, these enemies advanced but were stopped by the obstacle type tiles (**type 8**). Now we can configure that any type of tile that is not transferable (**type 0**) stops them. This was done so that in the Sgt. Helmet, also changed in the electrified fences (not type 8). All this is achieved with two new directives in config.h:

```
#define TYPE_7_FIXED_SPRITE 4 // If defined, type 7 enemies are always #  
#define EVERYTHING_IS_A_WALL // If defined, any tile <> type 0 is a wall.
```

The first is to set a number of enemy graphics for type 7 enemies. The second is for enemies to pursue you only by the tiles of type 0. If you comment the latter, the behavior will be the same as hitherto: Will chase by any tile and will stop only with type 8.

Scripting engine stuff

Yes, we've put a lot of new stuff in here. But we are already waiting for the chapter where we will begin to explain the scripting engine.

Corrections and optimizations

In addition, we have completely rewritten the part of the engine that managed the shots, so that now the collision is more accurate and takes up less space. We have fixed some bugs as well, and we have changed some bits of code optimizing even more. Before the optimizations, the Sgt. Helmet game would not have fit in the memory nor of cina. Spring

How do I upgrade?

It is neither complicated nor painful but, anyway, grab the folder as you have it and make a backup ZIP, as if you do not want the thing, just in case. When you are, follow these simple steps:

1. Rename it to the folder of your Game. I do not know, put "-old" at the end.

2. Prepare the new package as you did with the previous one , ie, unzip, rename it to the folder Game, rename it to **/dev/churromain.c** and script **/churromain.spt** by the name of your Game , and edit% 1 **make.bat** changing the name of your Game. As we have already done.
3. Copy the graphics, maps, enemies and files from your old folder to the new, that is, everything is in **gfx/** in **/map** and **/ENEMS**.
4. Re-edit **config.h** - You have to open the new and the old **/dev/config.h** yours and put the new one as had the old yours. Look carefully at the new features: comment on those you do not want, or try to use them, or whatever you care.
5. Re-copy your own files Game from the **/dev** old to the new. Namely: **tileset.h, spriteset.h, title.bin, marco.bin (if using), ending.bin, mapa.h and enems.h**

It is done. Now you just have to check that you did everything by opening your window command line and running **make.bat** ... If not, check all the steps.

And, for the next, now, yes, the dreaded scripting. In the meantime, playing Sgt. Helmet!

NOTICE

Thanks to Fabio Didone, who is following the tutorial and doing his own game, we have noticed that in the last version of the Churrera Maker we have sneaked a small bug that breaks the behavior of **ONLY_ONE_OBJECT**. We have updated the 3.99.1 package with the bug rectified. You can download it again in the usual way:

<http://www.mojontwins.com/churrera/mt-churrera-3.99.1.zip>

If you started your game, you do not have to reassemble everything. Just suffice that you overwrite **mainloop.h** with the new version included in the package.

Workshop creates your own game of Spectrum (Chapter 7C)

The Churrera Maker 3.99.2

First of all, download the new version package of the Churrera Maker by clicking right here below:

<http://www.mojontwins.com/churrera/mt-churrera-3.99.2.zip>

Again? And the scripting chapter?

Yes, I know we owe you the scripting chapter and twenty thousand more things, but we did not think it was cool to continue expanding the Churrera Maker and that you could not enjoy the new things that it brings. What does this mean? Well, nothing, that maybe you do not know much of some of the characteristics that we are going to explain here, but patience. The scripting chapter is very important and we do not want to do it anyway. We will try to start next week to give the first guidelines. In the meantime, take a look at this... And refer to the previous chapter which explains how to update the source of your game to a new version.

Timers

It adds to the Churrera Maker a timer that we can use automatically or from the script. The timer takes an initial value, counts down, can be recharged, can be set every how many frames is decremented or decide what to do when it runs out. In **config.h**, as always:

```
#define TIMER_ENABLE
```

With **TIMER_ENABLE** necessary to handle the timer code is included. This code will need some other directives that specify how it works:

```
#define TIMER_INITIAL 99  
#define TIMER_REFILL 25  
#define TIMER_LAPSE 32
```

TIMER_INITIAL specifies the initial value of the timer. Refills of time put the underwriter as hot-spots type 5, reload the value specified in **TIMER_REFILL**. The maximum value of the timer for both the initial and recharging is 99. To control the interval time between each decrease of the timer in **TIMER_LAPSE** specify the number of frames that must elapse.

```
#define TIMER_START
```

If **TIMER_START** set, the timer will be active from the beginning.

We also have some guidelines that define what will happen when the timer reaches zero. It is necessary to uncomment those that apply:

```
#define TIMER_SCRIPT_0
```

Defining this, when the timer reaches zero a special section of the script, **ON_TIMER_OFF** run. It is ideal for carrying all control timer scripting, as in Cadàveriön.

```
// # define TIMER_GAMEOVER_0
```

Defining this, the game will end (“GAME OVER”) when the timer reaches zero.

```
// # define TIMER_KILL_0  
// # define TIMER_WARP_TO 0  
// # define TIMER_WARP_TO_X 1  
// # define TIMER_WARP_TO_Y 1
```

If **TIMER_KILL_0** defined, a life is deducted when the timer reaches zero. If further defined **TIMER_WARP_TO** also will be changed to the specific screen, the player appearing in the **TIMER_WARP_TO_X** and **TIMER_WARP_TO_Y** coordinates. Use this if these data will be fixed during the game. In Cadàveriön, for example, they are changing, so everything is handled from the script.

```
// # define TIMER_AUTO_RESET
```

If this option is set, the timer will return to maximum after reaching zero automatically. If you are going to carry out the control by scripting, the better leave it commented.

```
#define SHOW_TIMER_OVER
```

If this is defined, in case we have defined or **TIMER_SCRIPT_0** or **TIMER_KILL_0**, a sign of "TIME'S UP!" Is displayed When the timer reaches zero.

Scripting:

As we have said, the timer can be administered from the script. Interestingly, if we decide to do this, we activate **TIMER_SCRIPT_0** so that when the timer reaches zero the **ON_TIMER_OFF** section of our script is executed and control is total.

In addition, these checks and commands are defined:

Checks:

```
IF TIMER >= x  
IF TIMER <= x
```

That will be fulfilled if the value of the timer is greater than or equal to or less than or equal to the specified value, respectively.

Commands

SET_TIMER a, b – Allows you to set **TIMER_INITIAL** (a) and **TIMER_LAPSE** (b) values from the script.

TIMER_START – It is used to start the timer.

TIMER_STOP – It is used to stop the timer.

Control of pushable blocks

We have improved the engine so that more can be done with the tile 14 of type 10 (pushable tile) that simply push it or that stops the trajectory of the enemies. Now we can tell the engine that launches the **PRESS_FIRE** section of the current screen in the script just after pushing a pushable block. In addition, the number of the tile that is "stepped" and the final coordinates are stored in three flags that we can configure, to be able to use them from the script to make checks.

This is the system used in the script to control Cadàveriön to place statues on pedestals, for instance.

Recall what we had so far:

```
#define PLAYER_PUSH_BOXES  
#define FIRE_TO_PUSH
```

The first one is necessary to activate the pushable tiles. The second forces the player to press FIRE to push and is therefore not mandatory. Let us now look at the new directives:

```
#define ENABLE_PUSHED_SCRIPTING  
#define MOVED_TILE_FLAG 1  
#define MOVED_X_FLAG 2  
#define MOVED_Y_FLAG 3
```

ENABLE_PUSHED_SCRIPTING activating the tile that is pressed and its coordinates are stored in the flags specified by **MOVED_TILE_FLAG**, **MOVED_X_FLAG** and **MOVED_Y_FLAG** respectively directives. In the code shown, the treaded tile will be stored in flag 1, and its coordinates in flags 2 and 3.

```
#define PUSHING_ACTION
```

If we define this further when we push a block **AT ANY PRESS_FIRE** sections and **PRESS_FIRE** of the current screen script will run. In this case, the "JUST_PUSHED" condition is fulfilled and from the scripting can control what happens when you push a block.

We recommend Cadaveriön study the script, which, besides being a good example of using the timer control and pushable block, turns out to be a fairly complex script that uses a lot of advanced techniques. Well, when the tutorial is ready: *)

Check if we get off the map

It is advisable to put limits on your map so that the player can not escape the map and the engine does strange things, but if your map is narrow you may want to take advantage of the whole screen. In that case, you can activate:

```
#define PLAYER_CHECK_MAP_BOUNDARIES
```

It will add checks and will not let the player leave the map. eye! If you can avoid using it, the better: you will save space.

Type of enemy "custom" gift

So far we had left the type 6 enemies without code, but we figured we would not have to put one, for example. It behaves like bats *Cheril the Goddess*. To use them, put them in the enemy setter as type 6 and use these directives

```
#define ENABLE_CUSTOM_TYPE_6  
#define TYPE_6_FIXED_SPRITE 2  
#define SIGHT_DISTANCE 96
```

The first one activates them, the second defines which sprite will use (minus 1, if you want the sprite of enemy 3, put a 2. Sorry for the confusion, but saving bytes). The third one says how many pixels you see from far away. If he sees you, he follows you. If not, return to your site.

This implementation, in addition, uses two directives of the enemies of type 5 to work:

```
#define FANTY_MAX_V 256  
#define FANTY_A 12
```

Define there the acceleration and the maximum speed of your type 6. If you are going to also use type 5 and you want other values, be a man and modify the engine.

Keyboard / joystick configuration for two buttons

There are side view games that are best played with two buttons, one of jump and one of firing. If you enable this policy:

```
#define USE_TWO_BUTTONS
#define FANTY_A 12
```

The keyboard will be the following, instead of the usual one:

```
left
right d
W up
down
N jump
M shot
```

If a joystick is chosen, FIRE and shoot M, and N jumps.

Shooting up and diagonally for side view

Now you can let the player shoot up or diagonally. To do this define this:

```
#define CAN_FIRE_UP
```

This configuration works best with **USE_TWO_BUTTONS**, as this separate "up" button jump. If you do not hit "up", the character will fire to where he is looking. If you press "up" while firing, the character will fire upwards. If in addition, you are pressing an address, the character will fire at the indicated diagonal.

Masked Bullets

For speed, the bullets do not wear masks. This works fine if the background on which they move is dark (few active INK pixels). However, there are situations where this does not happen and looks bad. In that case, we can activate masks for the bullets:

```
#define MASKED_BULLETS
```

And now

With these things you can make many new types of game. Do not scratch yourself too much if half of what is explained here sounds Chinese, because soon you will understand how the scripting works and all the power it puts into your hands.

Workshop creates your own game of Spectrum (Chapter 8)

Almost without time to breathe are already here again the boys of Mojon Twins with his workshop of the Churrera Maker. In this chapter, we will give Scripting. A world of possibilities that will force us to put the batteries. The thing gets tough and we all know it, so you can get the knives out. Put it between your teeth because at last the chapter begins that many were waiting for.

Chapter 8: Beginning with Scripting

First of all, download the package of materials corresponding to this chapter by clicking on this link:

<http://www.mojontwins.com/churrera/churreratut-capitulo8.zip>

Man, at last!

Yes, already. But now you're going to shit. Because this can be as thick and deep as you want. Nah, seriously, it's nothing. Let's do it with Vaseline too. In this first chapter, we will explain how the system is set up so that you understand what it does, what it is for, and how it works, and we will end up seeing super simple examples of level 1, easy easy. The next chapter will end up making the script *Dogmole Tuppowski* and, from there, explore, part by part, what can be done with the script. Because it can be done a lot and varied.

Let's do it, then!

Voucher. What is a The Churrera Maker script? It is nothing more than a set of checks with associated actions, organized into sections, which serve to define the gameplay of your Game. That is, the things that happen, and the things that have to happen to make everything go cool, or for everything to go wrong.

To see, without a script you have a basic gameplay. Take X objects to finish, kill X bugs ... Go beyond that needs checks and related actions: if we are in such a place and we have done such thing, open the castle door. If we enter the screen just talked with which character comes out the text "Hello you" and sounds a little noise. That is what we mean.

The script serves you from to put a nice tile on the screen 4 and a text that put "you're home" up to react to what you do on a screen, check that you have done other things, see that you pushed such tile, And then turn on the timer or change the setting or whatever.

As soon as you know the tools you have for sure you can think of a thousand things to do. Many times we discovered applications that we did not even know were possible when we started designing the system, so you see.

This is really fun!

But it's programmed.

Sure, hell, but it's one thing to have to program a gameplay in C and put it in the engine and another thing is to have a language specifically designed to describe a gameplay and that is so simple to learn and master. Because I am sure that many or it will sound how it is mounted, especially if someday have done an affair with **PAWS** or **GAC** or have I fretted with a game maker. Because to the Mojones we love that of reinventing wheels, and it turns out that the super system that we devise is the most used for these needs of all that exist. You will see.

Okay, tell me how it goes.

Agree. See if you can say in one breath, and then we're going disagree: a **script** consists of **sections**. Each section is nothing more than a set of clauses. Each clause consists of a checklist and a **list of commands**. If all are met and each of the findings of the list, will be executed, in order, each and every one of the commands. The game engine will call the scripting engine on certain occasions, executing one of those sections. **Run a section means going clause by clause making checks your checklist and, if satisfied, in order to execute commands your command list.** That is the important concept that must be understood.

To know which time the game engine calls the scripting engine. We have to understand what the sections are and what types of sections there are:

ENTERING SCREEN n: with n being a screen number, run just as you enter a new screen, once drawn the stage, initialized enemies, and placed hot-spots. You can use them to modify the scenario or initialize variables. For example, associated with screen 3, we can place a script to check if we have killed all the enemies and, if not, to paint an obstacle so that we can not pass.

ENTERING ANY: run for all screens, just before **ENTERING SCREEN n**. That is, when you walk into the screen 3 is executed first section **ENTERING ANY** (if it exists in the script), and just after the section **ENTERING screen 3** run (if it exists in the script).

ENTERING GAME: runs once to start the game. It is the first thing that is executed. You can use it to initialize the value of variables, for example. We'll see about this later.

PRESS_FIRE AT SCREEN n: with n being a screen number, runs on several assumptions being on the screen n: if the player presses the button action to push a block if you have activated the **PUSHING_ACTION** directive, or entering a zone special defined from scripting called "fire zone" (already explain) if we have activated the **ENABLE_FIRE_ZONE** directive. We will usually use these sections to react to the player's actions.

AT ANY PRESS_FIRE: it runs on all the above assumptions, for any screen, just before **AT SCREEN PRESS_FIRE n**. That is, if you press action on the screen 7 clauses **PRESS_FIRE AT ANY** be executed and then those of **PRESS_FIRE AT SCREEN 7**.

ON_TIMER_OFF: runs when the timer reaches zero if we activated the timer and have set to happen with **TIMER_SCRIPT_0** directive.

For version 3.99.2 these are the possible sections, although we will add more in future versions. For example, by eliminating an enemy. But not now.

By the way, it is not mandatory to write all possible sections. The engine will execute a section only if it exists. For example, if there is nothing to do on the screen 8 as there will not write any section of screen 8. If there is no common action to enter all screens, no **ENTERING ANY** section. And so. If there is nothing to execute, the engine does not run anything and anymore.

To see, sum up: for what so much fuss of sections and stuff? Very simple: on the one hand because, in general, the checks and actions will be specific to a screen. This is a drawer. But the most important thing is that we are in an 8 bit micro and we can not afford to be continuously doing all the checks. We do not have frame time, so we have to leave them for isolated moments: no one will be stitched if it takes a few milliseconds more when switching screens or if the action stops briefly when the action key is pressed.

Saving values: flags

Before we can continue, we have to explain another concept: flags, which are just variables where we can store values that we can later query or modify from the script.

Many times we will need to remember that we have done something, or counting things. For this, we will have to store values, and for that, we have the flags. In principle, we have 16 flags, numbered from 0 to 15, although this number can be easily modified by changing a definition of **definitions.h** (search **#define MAX_FLAGS** and change 16 by another number).

Each flag can store a value from 0 to 255, which gives us plenty of things. Most of the time we will only be storing a Boolean value (0 or 1).

In the script, most checks and commands take numerical values. For example, **IF PLAYER_TOUCHES 4, 5** will evaluate to "true" if the player is touching the coordinate box (4, 5). If we put a # to the number, we will be referencing the value of the corresponding flag, so **IF PLAYER_TOUCHES # 4, # 5** will evaluate to "true" if the player is touching the coordinate box stored in flags 4 and 5, whatever This value.

This level of indirection (remember that word to say in the disco: the girls fall struck before the programmers who know this concept) is really useful because this way you can save a lot of code. For example, it is what allows, in *Cadaveriön*, that the control of the number of statues placed or to eliminate the gate that blocks the exit of each screen can be made from a single common section: all the coordinates are stored in flags and we use the operator # To access their values in the checks.

But do not worry if you do not get this now, that will be clarifying everything.

Enough?

A lot of information? I am aware of it. But as soon as you see it in motion, you will surely catch it. Let's start with the simplest examples of scripting by looking at some of the sections we need for our *Dogmole*, which we'll build little by little. In this, we will dedicate this chapter and the next one. Then we will explain, in thematic form, all possible checks and commands that we can use.

How do I activate scripting? Where is it introduced?

To activate the scripting we will have to do two things: first, activate it and configure it in **config.h**, and then modify our **make.bat** to include it in the project. Let's start with **config.h**. The directives related to the activation and configuration of the scripting are these:

```
#define ACTIVATE_SCRIPTING // Activates msc scripting and flag related stuff.
#define SCRIPTING_DOWN // Use DOWN as the action key.
// # define SCRIPTING_KEY_M // Use M as the action key instead.
// # define SCRIPTING_KEY_FIRE // User FIRE as the action key instead.
// # define ENABLE_EXTERN_CODE // Enables custom code to be run using EXTERN n
// # define ENABLE_FIRE_ZONE // Allows to define a zone which auto-triggers "FIRE"
```

The first directive, **ACTIVATE_SCRIPTING**, is the one that will activate the scripting engine and will add the necessary code to execute the correct section of the script at the right time. It is the one that we have to activate yes or yes.

Of the next three, we have to activate only one, and they serve to configure which key will be the action key, which launches the **PRESS_FIRE AT ANY** and **PRESS_FIRE AT SCREEN n** scripts. The first one, **SCRIPTING_DOWN**, sets the "down" key. This is perfect for side perspective glows, as this key is not used for anything else. The second, **SCRIPTING_KEY_M** enables the "M" key to launch the script. The third, **SCRIPTING_KEY_FIRE**, sets the trigger key (or the joystick button) to do so. Obviously, if your game includes shots, you can not use this setting. Well, if you can, but there you are.

The following directive, **ENABLE_EXTERN_CODE**, will leave it normally disabled unless you are a barbecue teacher. There is a special script command, **EXTERN n**, where n is a number, which is to call a function of C located in the file **extern.h** passing that number. In this function, you can add the code C you want and you need to do fun things. For example, D_Skywalk has used it in Justin and the Lost Abbey to add the code that is painting the pieces of the sword that we have collected in the marker. If you do not need to program your own behaviors in C, leave it disabled and save a few bytes.

Finally, **ENABLE_FIRE_ZONE** is used so that we can define a special rectangle within the playing area of the current screen. Normally, we will use **ENTERING_SCREEN n** to define the rectangle using the **SET_FIRE_ZONE** command x1, y1, x2, and y2. When the player is within this special rectangle, the **PRESS_FIRE AT ANY** and **PRESS_FIRE AT SCREEN n** scripts will be executed from the current screen. This comes in really well in order to execute actions without the player having to press the action key. It is what we use in Sgt. Helmet to put the bombs on the final screen or display the message "I AM SELLING A SEMI-NUDE MOTORCYCLE". If you think you're going to need this, enable this policy. If not, leave it unactivated and save a few bytes. Do not worry we'll explain this more slowly.

The next thing is to configure **make.bat** well, activating compilation and inclusion of the script. If you open **make.bat** and you are fixed, you will see that at the beginning there is a call to **msc**. This is the script compiler, which gets the name of the script file, the output name (which will be **msc.h**) and the total number of screens of your game:

```
Echo ### COMPILING SCRIPT ###
cd ..script
Msc cadaver.spt msc.h 20
Copy * .h ..dev
```

The name of your script is the name of your game with a **.spt** as an extension. The file is located in a **script**. If you enter script you will see a **churromain.spt**. Rename it to match the name of your game (the same as your .c file dev). Do not forget about the number of screens, which is very important. If you do not put it well the code of the script interpreter will be generated badly. For our *Dogmole*, the file will be **devdogmole.spt**, and the **make.bat** have to put:

```
Echo ### COMPILING SCRIPT ###
cd ..script
Msc dogmole.spt msc.h 24
Copy * .h ..dev
```

Because our *Dogmole* has 24 screens.

If now you will script and open the file with the script (which was previously called churromain.spt and have renamed with the name of your Game) you will see that it is empty, or nearly so. Just bring a skeleton of a section. It has this look:

```
Silly title
# Copyleft 201X your group roneón soft.
# The Churrera Maker 3.1

# Flags:
# 1 -

ENTERING GAME
IF TRUE
THEN
SET FLAG 1 = 0
END
END
```

This is where we are going to write our script. Notice how it looks: that is there is the **ENTERING GAME** section, which runs just start the game. Within this section, there is a single clause. This clause simply a check: IF TRUE, it will always be true. Then there is a THEN and right there, and even the END command starts the list. In this case, a single command: SET FLAG 1 = 0, which puts the flag from 1 to 0.

This script is absolutely useless. In addition to doing nothing, it turns out that the system sets all flags to 0 at the beginning, so you do not need to initialize them to zero. Why is it there? I do not mess with it there was something. In fact, the first thing you're going to do is CLEAR IT.

It is interesting to modify that header. The lines starting with # (does not have to be #, you can use, for example, or 'or //, or whatever you want) are comments. Get accustomed to putting comments on your own line. Do not put comments at the end of a check or a command, that the compiler is vague and can interpret what is not. And above all, get used to putting comments. So you can understand what changes you made three days ago, before the drunkenness and that affair with the brunette of reception.

As for now the variables (flags) are identified by a number (I have yet to improve the compiler to define aliases, as well suggested D_Skywalk) is a good idea to make a list of each and every thing. I always do, look:

```
# Cadáverión
# Copyleft 2013 Mojon Twins
# The Churrera Maker 3.99.2

# Flags:
# 1 - Tile trampled by the block being pushed
# 2, 3 - X and Y coordinates
# 4, 5 - X and Y coordinates of the tile "retry"
# 6, 7 - X and Y coordinates of the door tile
# 8 - number of finished screens
# 9 - number of statues to be placed
# 10 - number of statues placed
# 11 - We've already removed the gate
# 12 - Screen that we return to run out of time
# 13, 14 - Coordinates to which we return ... bla
# 15 - Floor
# 0 - stored value of 8
# 16 - I'm selling my new bike.
```

You see? That comes great to know where you have to play.

My first clauses

Since we know where to play, let's start with our script. Let's look at the basic syntax. This is the look of a script:

```
SECTION
CHECKS
THEN
COMMANDS
END
...
END
...
```

As we see, each section begins with the section name and ends with **END**. Enter the name of the section and **END** are the clauses that make it up, which can be one, or may be several. Each clause begins with a checklist, each on a line, the word **THEN**, a list of commands, each on a line, and **END**. Then another clause may or may not come.


```
SET TILE (x, y) = t
```

Where (x, y) is the coordinate (remember, we have 15×10 tiles on the screen, so x can go from 0 to 14 and 0 to 9) and t is the tile number we want to paint. With the map open in front to count squares and see where we have to paint things, we place the pedestal first and then all the decorations:

```
# College Lobby
ENTERING SCREEN 0
# Decoration and pedestal
IF TRUE
THEN
pedestal
SET TILE (3, 7) = 22
SET TILE (4, 7) = 23
# Decor
SET TILE (1, 5) = 29
SET TILE (1, 6) = 20
SET TILE (1, 7) = 21
SET TILE (6, 6) = 20
SET TILE (6, 7) = 21
SET TILE (7, 7) = 28
SET TILE (1, 2) = 27
SET TILE (1, 3) = 28
SET TILE (2, 2) = 29
SET TILE (2, 3) = 27
SET TILE (3, 2) = 32
SET TILE (3, 3) = 33
SET TILE (9, 1) = 30
SET TILE (9, 2) = 30
SET TILE (9, 3) = 31
END
END
```

Okay, you wrote your first clause. It has not been so complicated, right? I guess for satisfaction to be complete you will want to see it. Well: let's add some code that we will remove from the final version and we will use it to go to the screen we want to start the game and try so we are doing everything right.

If you remember, one of the possible sections that we can add to the script is the one that runs just at the beginning of the game: **ENTERING GAME**, which is the one that was empty at the beginning and we deleted because it was useless at all. Well, we are going to make an **ENTERING GAME** that will serve us to go directly to the screen 0 at the beginning and to verify that we have put all the tiles cool in the commands of the clause of the section **ENTERING SCREEN 0**. We add, therefore, this code (You can add it wherever you want, but I usually leave it at the beginning. It does not matter where you put it, but it's always cool to follow an order.)

```
ENTERING GAME
IF TRUE
THEN
WARP_TO 0, 12, 2
END
END
```

What does this do? It will cause the list of clauses to be executed at the beginning of the game, formed by a single clause, which will always be executed (because it has IF TRUE) and what it does is to move to the coordinate (12, 2) of screen 0, Because that is what the **WARP** command does:

```
WARP_TO n, x, y
```

It moves us to the screen x, and makes us appear in the coordinates (x, y).

What will happen? Ooooh, ooh. It's simple: when the player starts the game will run the section **ENTERING GAME**. This section all it does is move the player to the position (2, 2) and switch to the screen 0. Then the screen when entering 0, the **ENTERING SCREEN 0** section, we painted tiles extra run. We're going to try it! Compile the game and run it. If all goes well, we should appear on our screen or decorated:



That's good. Let's do more. Let's paint more tiles to decorate other screens. Exactly in the same way that we have decorated screen 0, we will also decorate screen 1, placing the Miskatonic University sign (tiles 24, 25, and 26) and armor (tiles 32 and 33):

```

# College hall
ENTERING SCREEN 1
# Miskatonic poster, etc.
IF TRUE
THEN
SET TILE (7, 2) = 24
SET TILE (8, 2) = 25
SET TILE (9, 2) = 26
SET TILE (1, 6) = 32
SET TILE (1, 7) = 33
SET TILE (13, 6) = 32
SET TILE (13, 7) = 33
END
END

```

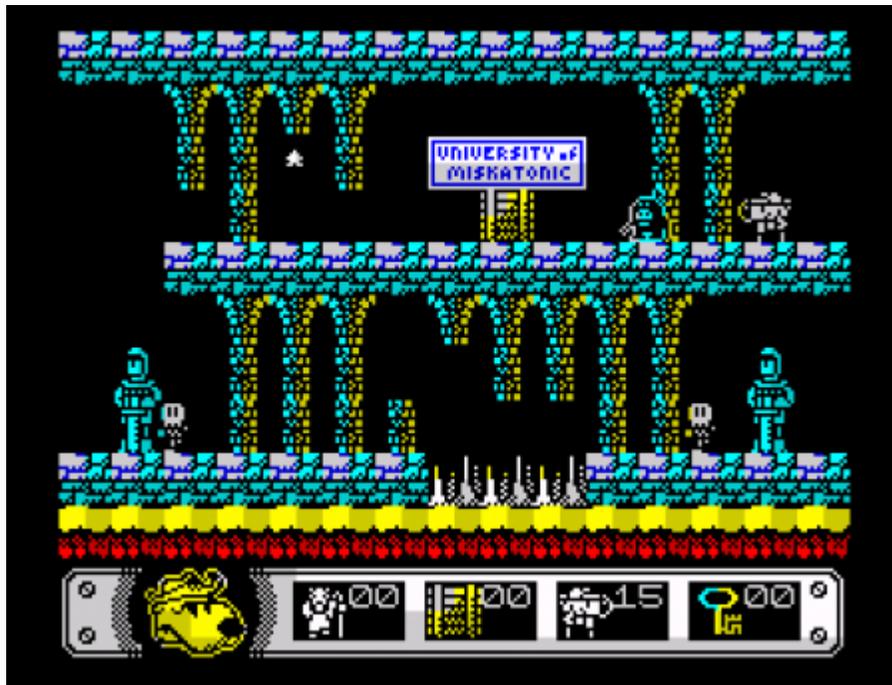
Lets go see it! **ENTERING_GAME** changes the screen to jump to 1 instead of the screen 0:

```

ENTERING GAME
IF TRUE
THEN
WARP_TO 1, 12, 2
END
END

```

Compile, execute ... et voie-la!



In the same way we add code to put more decorations on the map. The truth is that we get bored soon and there is only decoration on screen 6 (a lamp) and screen 18 (an anchor on the beach). Why do not you take advantage and put more? These are the ones that come in the original game:

```
ENTERING SCREEN 6
IF TRUE
THEN
SET TILE (10, 1) = 30
SET TILE (10, 2) = 31
SET TILE (10, 4) = 35
END
END

ENTERING SCREEN 18
IF TRUE
THEN
SET TILE (4, 8) = 34
END
END
```

I think I'm picking it up

Well, it's time to leave. Try to absorb this knowledge well, soak them well. If you have not caught something from here, do not be in a hurry and wait for us to follow, you will surely end up understanding. And, as always, whatever you want to ask, ask it!

In the next chapter, we will introduce the seeds of the gameplay: we will detect that all sorcerers have died to remove the stone from the entrance of the university, and we will program the logic to leave the boxes in the vestibule.

Until then, work with this. In the file with the material in this chapter have the *Dogmole* with the script half done with the things we have seen in this chapter.

Workshop Create your own game Spectrum (Chapter 9)

Here you have at last the new and feared chapter of the Workshop of the Churrera Maker: Basic Scripting. In this ninth chapter, we will see in a fun, fun and detailed way the scripting system that will allow us to fully customize our game with features of all kinds. It opens a whole world in front of your keyboards. Make good use of "force". Ah! And remember that great power carries a great responsibility.

Chapter 9: Basic Scripting



Basic Scripting?

That's it. In this chapter we define the gameplay of *Dogmole Tuppowski* and we will learn some basics of scripting.

The scripting system of Churreras is very simple and seems quite limited, but you can solve problems with some skill and get fairly complex things. In addition, in each version that we take out of the Churrera Maker we expand it and every time it works better, so if you practice you can achieve in your games quite complex gameplay designs. Dogmole rules are simple on purpose to illustrate a simple behavior and learn. Later we will see the script of different games mojonos so that you see how we have managed to do things.

The scripting system has many different commands and checks. Since I do not want to turn this course into a reference to a list of endless things, I refer to **motor_de_clausulas.txt** file in the folder **/script** of the Churrera Maker: there is a list of everything that is able to compile msc. It's no big deal to kick him out.

Let's refresh a little.

Recall that the script is made up of sections, and that each section includes clauses. Each clause is nothing more than a checklist and a list of commands: if all the checks are true, the commands will be executed.

We control which clauses will be executed by placing them in one or another section. Let's remember the sections that exist and when they are executed:

ENTERING GAME: This section will run just start the game, and never again.

ENTERING ANY: This section will run on entering each screen, and also to step on an enemy. Yes, it has no logic, but it is because it comes very well for certain things.

ENTERING n SCREEN: This section will run to enter the screen N.

AT ANY PRESS_FIRE: This section will be executed by pressing action, no matter what screen you are.

N FRESS_FIRE AT SCREEN: This section will run the catch action if we are in the N screen, and also to step on an enemy. Neither has logic, but also comes well.

ON_TIMER_OFF: This section will run if we have a timer, this reaches 0, and we have set in config.h to happen.

Go for it!

Before we start we will recap, because it is important that we know what we are doing. Recall, then, what was the design of our *Dogmole Tuppowsky* gameplay:

In the first place, the door of the university is closed, and to open it all monks must be killed. There are 20 monks placed all over the map, in the bottom (the two lower rows) and you have to load them all. When they are all dead, we will have to remove the stone we place on the map at the entrance to the university.

Then you have to program the logic of the pedestal, inside the university. If we touch the pedestal carrying an object, we lose it and a flag will be incremented by counting the number of objects we have deposited. When this number reaches 10, we will have won the game.

The script of this game will be very simple. The first thing we have to look at is what we will need to store to allocate some flags for it. In our case, as the engine is already in charge of counting the monks we have killed, we will only need to go counting the boxes that we are depositing and also need to remember if we have removed the stone or not. Let's use two flags: 1 and 3. Why these two and not

others? Well, yes. Actually, it does not matter.

Recall that we mentioned in the previous chapter that it was interesting to point out what each flag did at the beginning of our script:

```
# Flags:  
# 1 - general account of objects.  
# 3 - 1 = open university door.
```

Counting dead monks

What a title, huh? But it's cool. The first thing we are going to see is how to count the dead monks to remove the stone from the screen. 2. First of all, we have to see what we are going to remove. Screen 2 is this, and I have marked the stone that we have to remove by scripting:



If we have a little, we realize that the Pedro occupies the coordinates (12, 7). We point them out. We have said that we will use flag 3 to store if we have already killed all the monks or not. If flag 3 is worth 1, it means that we have killed all the monks, and in that case, we would have to modify that screen to erase the stone from the position we have written down. Why not start there? We therefore create a clause for when we enter the screen 2:

```
# University Entrance

ENTERING SCREEN 2

    # Control of the university door.

    IF FLAG 3 = 1

    THEN

        SET TILE (12, 7) = 0

    END

END
```

There is little again in this first clause of gameplay that we have written: it is about checking the value of a flag. Instead of the **IF TRUE** we had used so far, we write **IF FLAG 3 = 1** only evaluate to true if the value of our flag 3 is precisely 1. In this case, the body of the clause will be executed: **SET TILE (12, 7) = 0** write the empty tile on the space occupied by the Piedro, eliminating it. Therefore, when entering this screen with the flag 3 to 1, the stone will be erased and there will be no obstacle. Is the concept caught?

Let's go, then. We have said that the flag 3 to 1 means that we have killed all the enemies, but the flag 3 is not going to put to 1 automatically. We need to create a rule that will actually set it to 1.

As on the screen where the stone appears, there are no monks, there will never be the situation of killing the last monk on the screen of the stone. That is: we will always be on another screen when we kill the last monk. Therefore, a good place to check that we've killed all the monks is to enter any screen, that is, in our section **ENTERING ANY**. And better yet, the particularity that we mentioned before **ENTERING ANY** also runs when we step on an enemy. Each time we enter a screen, we will verify that the number of enemies eliminated is 20 and, if it is the case, we will put the flag 3 to 1:

```

# Open University

ENTERING ANY
  IF ENEMIES_KILLED_EQUALS 20
  IF FLAG 3 = 0
  THEN
    SET FLAG 3 = 1
    SOUND 7
    SOUND 8
    SOUND 9
    SOUND 7
    SOUND 8
    SOUND 9
  END
END

```

With this, we get just what we want. Note that there is new evidence: **IF ENEMIES_KILLED_EQUALS 20** will be true if the number of enemies eliminated (or monks) worth exactly 20. If that is true, immediately afterward check the value of the flag 3 for that is 0. With this we do Is to make sure that this clause will only be executed once, or it would otherwise be executed upon entering each screen.

If everything has been fulfilled, we will put the 3 to 1 flag (which is what we wanted) in addition to releasing a series of pitches. Yes, the **SOUND** command plays sound n n. These are the sounds of the engine. You can look at what corresponds to each number in the file **beeper.h** the end.

With this we will have the first part of our gameplay ready: if all the enemies are dead, we put the flag 3 to 1. In screen 2, if the flag 3 is worth 1, we remove the pebble.

Logic of the boxes

Now we only have to define the second part of the gameplay. If you remember, we set the engine **ONLY_ONE_OBJECT**. That means that the maximum of objects we can pick up is one, that is, we can only carry one box.

The goal of the game is to take 10 boxes to the University counter, so we will have to program in the script the logic that, if we take an object and activate the counter, we subtract that object and increase the counter of objects delivered , Which we have said will be flag 1.

The counter is on display 0, if you remember: we've painted with **SET TILE** from our script in the **ENTERING SCREEN 0** section. The pedestal occupies positions (3, 7) and (4, 7).

Let's now write a piece of script that, if we press the action key on screen 0, it verifies that we are touching the pedestal and that we take an object, to eliminate that object and to increase in one the account.

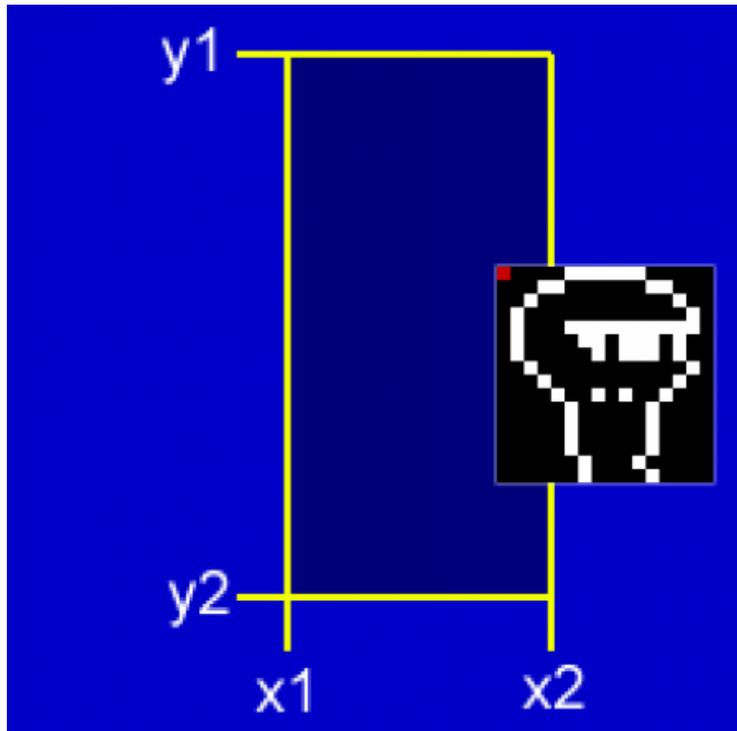
The first thing we have to solve is the detection that we are touching the pedestal. If the pedestal occupied a single tile in (x, y), it would be very simple:

```
IF PLAYER_TOUCHES x, y
```

If any pixel of the player touches the tile (x, y), that condition evaluates to certain. The problem is that our pedestal occupies two tiles. One solution would be to write two identical clauses, one with a **PLAYER_TOUCHES 3, 7** and the other with a **PLAYER_TOUCHES 4, 7**, but that is not necessary because we have other tools.

To check that we are inside an area we have two special checks:

```
IF PLAYER_IN_X x1, x2  
IF PLAYER_IN_Y y1, y2
```



The first one will evaluate to certain if the x-coordinate, in pixels, of the upper left corner of the picture of our character's sprite is between x1 and x2. The second will do so if the y-coordinate in pixels of the upper left corner of our character's sprite frame is between y1 and y2.

Let's look at it with a picture. Here we see an area delimited by x1, x2 y by y1, y2. The player will be "inside" of that area if the pixel marked in red (the one in the upper left corner of the sprite) is "inside" that area.

When we want to check that our character is within the rectangular area that occupies a set of tiles, we must follow the following formula to calculate the values of x1, x2, y1 and y2. If (tx1, ty1) are the coordinates (in tiles) of the upper left tile of the rectangle and (tx2, ty2) are the coordinates (also in tiles) of the lower right tile, i.e.:



With the area defined here, the values of x_1 , x_2 and y_1 , y_2 that we will have to use in the script are those that are obtained with the following formulas:

$$\begin{aligned} X1 &= tx1 * 16 - 15 \\ X2 &= tx2 * 16 + 15 \end{aligned}$$

$$\begin{aligned} Y1 &= ty1 * 16 - 15 \\ Y2 &= ty2 * 16 + 15 \end{aligned}$$

To see it, again, a little picture. Note that I have superimposed a sprite so that you see that to "touch" the tiles must be in the rectangle defined by the coordinates (x_1, y_1) and (x_2, y_2) :



Yes, if you are not accustomed to making numbers programming this is a mess of balls, but in reality, it is not so much when you memorize the formula, or, better if you understand it. It is multiplied by 16 to move from tile coordinates to pixel coordinates because tiles are 16×16 pixels. The addition and subtraction of 15 is to make "collision per box" with the sprite.

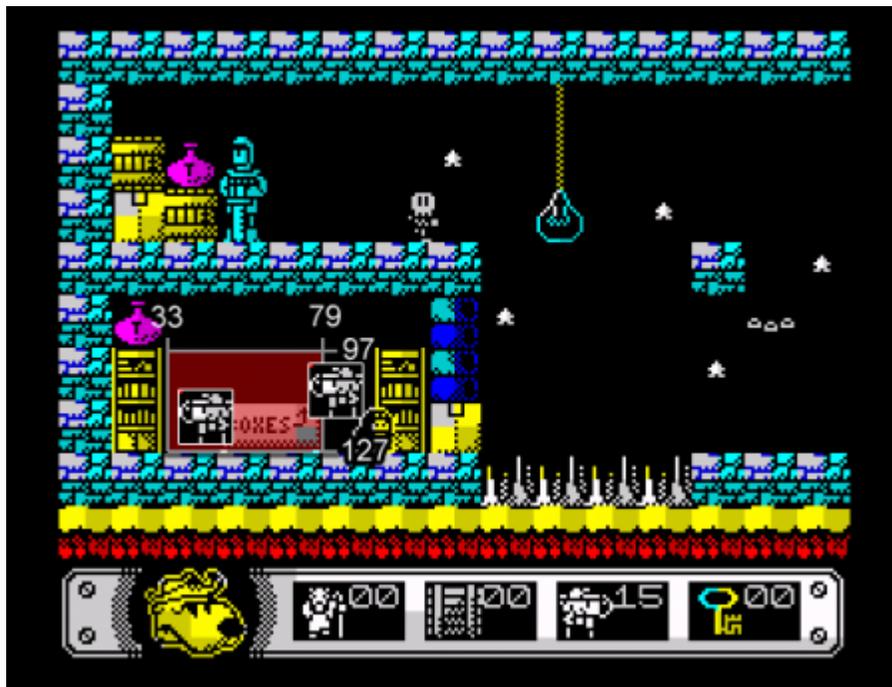
I know we could have designed the scripting to hide some of these tweaks, but so, requiring the programmer to do a couple of math operations on their own, we eliminated a lot of complexity in the code since we are giving the engine the "chewed" data.

To finish seeing it, let us move to our case and make the necessary operations using the values of our game. Here, the rectangle is formed only by two tiles in the coordinates (3, 7) and (4, 7). The tiles of the corners are those two tiles, precisely, so that tx1 equals value 3, ty1 will be 7, tx2 will be 4 and ty2 will also be worth 7. Thus, following the formulas:

$$\begin{aligned} X1 &= 3 * 16 - 15 = 33 \\ X2 &= 4 * 16 + 15 = 79 \end{aligned}$$

$$\begin{aligned} Y1 &= 7 * 16 - 15 = 97 \\ Y2 &= 7 * 16 + 15 = 127 \end{aligned}$$

That is, to touch the counter, the sprite must be between 33 and 79 in the X coordinate and between 97 and 127 in the Y coordinate. Let's see it graphically: notice how the sprite is touching the counter, the Upper left pixel of the square of your sprite (marked in red) must be within the area we have defined:



Also, we will have to check that we carry a box in the inventory. It would look something like this:

```

PRESS_FIRE AT SCREEN 0
# Detect pedestal.
# We detect it by defining a rectangle of pixels.
# We then check if the player has picked up an object.
# If everything is true, decrement the number of objects and increase FLAG 1
IF PLAYER_IN_X 33, 79
IF PLAYER_IN_Y 97, 127
IF PLAYER_HAS_OBJECTS
THEN
    INC. FLAG 1, 1
    DEC OBJECTS 1
    SOUND 7
END
END

```

Here's what we've seen: First, we check Dogmole's position with **IF PLAYER_IN_X** and **IF_PLAYER_IN_Y**. If everything is true, we find that we have an object picked up with **IF PLAYER_HAS_OBJECTS**. If all we met three things: first, we will increase by 1 flag **FLAG INC 1** by 1, 1. Then decrease 1 in a number of collected objects (so it will be 0, and can again pick up another box) with **DEC OBJECTS 1**. Finally, we will play sound number 7.

This done, we only have one thing to do: check that we have taken the 10 boxes. A good place to do it is just after the previous clause. As all clauses in a section are executed in order, just after counting we will place the check that we have already put 10 to finish the game. Therefore, we extended the **PRESS_FIRE AT SCREEN 0** section with the new clause. It would stay like that:

```

PRESS_FIRE AT SCREEN 0
  # Detect pedestal.
  # We detect it by defining a rectangle of pixels.
  # We then check if the player has picked up an object.
  # If everything is true, decrement the number of objects and increase FLAG 1
  IF PLAYER_IN_X 33, 79
  IF PLAYER_IN_Y 97, 127
  IF PLAYER_HAS_OBJECTS
  THEN
    INC. FLAG 1, 1
    DEC OBJECTS 1
    SOUND 7
  END

  # Game over
  # If we have 10 boxes, we have won!
  IF FLAG 1 = 10
  THEN
    WIN GAME
  END IF
END

```

Again, very simple: if we have 10 boxes (ie, if flag1 is 10), we will have won. The **WIN GAME** command causes the game ends successfully and the final screen is displayed.

Do you see that it has not been so much? Okay, the coordinates are a bit messy, but neither is it to cry. Or yes, if you are a sensitive person.

Interesting Improvement

As we have configured our game, the player has to press action to activate the counter and deposit an object. It is not a problem, but it would be more annoying if the player did not have to press anything. Precisely for that, we introduced in the engine what we have called "the zone of fire", or fire zone. This fire zone is nothing more than a rectangle on the screen, specified in pixels. If the player enters the rectangle, the engine behaves as if it has pressed action. The fire zone is automatically deactivated when switching screens, so if we define an **ENTERING SCREEN n**, only active while we are on that screen.

This comes divinely for our purposes: if on entering the screen 0 we define a fire zone around the counter, as soon as the player touches it will execute the logic that we have programmed in the script to leave the object to carry and increase the counter.

The fire zone is defined by the **SET_FIRE_ZONE** command, which receives the coordinates x1, y1, x2, and y2 the rectangle we want to use as a fire zone. If we want to match the fire zone with a rectangle formed by tiles, as in our case, apply the same formulas as explained above. That is, we are going to use exactly the same values.

The first thing we have to do is tell the engine that we are going to use fire zones. To do this, we have to activate the appropriate policy in our **config.h**:

```
#define ENABLE_FIRE_ZONE // Allows to define a zone which auto-triggers "FIRE"
```

That done, we'll just change the **ENTERING SCREEN 0 SET_FIRE_ZONE** adding section x1 y2 at the very end command, y1, x2:

```
# College Lobby
ENTERING SCREEN 0
  # Decoration and pedestal
  IF TRUE
  THEN
    pedestal
    SET TILE (3, 7) = 22
    SET TILE (4, 7) = 23
    # Decor
    SET TILE (1, 5) = 29
    SET TILE (1, 6) = 20
    SET TILE (1, 7) = 21
    SET TILE (6, 6) = 20
    SET TILE (6, 7) = 21
    SET TILE (7, 7) = 28
    SET TILE (1, 2) = 27
    SET TILE (1, 3) = 28
    SET TILE (2, 2) = 29
    SET TILE (2, 3) = 27
    SET TILE (3, 2) = 32
    SET TILE (3, 3) = 33
    SET TILE (9, 1) = 30
    SET TILE (9, 2) = 30
    SET TILE (9, 3) = 31

    # Fire zone (x1, y1, x2, y2):
    SET_FIRE_ZONE 33, 97, 79, 127
  END
END
```

The question you will ask is why #@!! Why did you did not put it in the game? Because this feature, which was originally included in branch 4.0 (in the *Hobbit* game) was reintroduced with version 3.99.1.

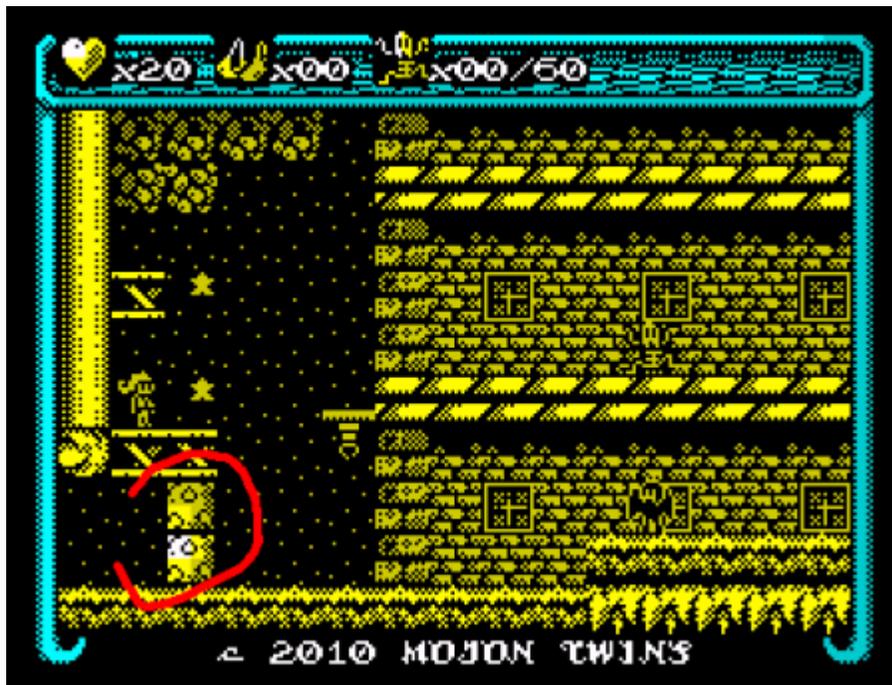
I'm a little lost

I take responsibility. You have to make a little coconut to the operation of the script. I think it is ideal to start with something very simple, even simpler than the *Dogmole* we have seen, and to progress.

I think something great: we were going to finish here the basic scripting chapter, but I think it would be good for us to see the scripts together for some of our games. I will choose a few games with a simple script, and we will explain step by step. It would be interesting that, in the meantime, you were playing the game to see how the different clauses affect.

Cheril Perils

The game with which we released the scripting engine was *Cheril Perils*. Then everything was in diapers and it was very simple. The Cheryl Perils script is the simplest script of all our script squeezes: here you only do one thing: that we have killed all the enemies, in which case we remove the skewers from the first screen. These spikes:



In principle, it is very much like part of what we have done in *Dogmole*: upon entering the screen, we find that we have killed all the enemies (there are 60 in total). If this is the case, print the empty tile on the spikes:

```

ENTERING SCREEN 20
  IF ENEMIES_KILLED_EQUALS 60
  THEN
    SET TILE (2, 7) = 0
    SET TILE (2, 8) = 0
  END
END

```

But here something happens that did not happen in the *Dogmole*: there are enemies on the screen where you have to remove the skewers. We are not worth detecting this when entering, since if we kill the last bug on this screen (it can happen) we would need to exit and re-enter the screen to make the engine sew. We need more code to detect that we killed the last bug and that it runs when we kill it. Recall that there was a quirk in the engine: when we step on a bug the **PRESS_FIRE AT SCREEN** for the current screen section runs. This comes great: put the same code in this section in **ENTERING SCREEN** solve the problem.

Neither short nor lazy ...

```

PRESS_FIRE AT SCREEN 20
  IF ENEMIES_KILLED_EQUALS 60
  THEN
    SET TILE (2, 7) = 0
    SET TILE (2, 8) = 0
  END
END

```

Now we are going to detect that we "left". If we did not do anything, leaving the screen 20 from the left we fit the screen 19 ... That is also on the other end of the map. In the original game, this was fixed with a pig hack, but with the current version of the Churrera Maker can be done well.

The first is to define a fire zone covering the left side of the screen, so that the section is run **PRESS_FIRE AT SCREEN 20** when we get to it. We add therefore the definition of the fire zone in section **ENTERING SCREEN 2** (we not also forget to activate the functionality inconfig.h). It looks like this:

```

ENTERING SCREEN 20
  IF ENEMIES_KILLED_EQUALS 60
  THEN
    SET TILE (2, 7) = 0
    SET TILE (2, 8) = 0
  END
  IF TRUE
  THEN
    SET_FIRE_ZONE 0, 0, 15, 159
  END
END

```

Is it clear? When entering the screen 20 two things happen: first it is checked if the number of enemies is worth 60, in which case the tiles-rocks that block the exit are eliminated. Then, in any case (**IF TRUE**) a fire zone covering the entire strip attached to the left of 15 pixels wide defined. As the player enters this area (can not do if you have not removed the barrier: it simply can not happen), the **PRESS_FIRE AT SCREEN** section 20 runs. Now we have to add code in the **PRESS_FIRE AT SCREEN** section 20 to detect that the player is trying to get out on the left and, in that case, finish the game successfully. It would stay like that:

```

PRESS_FIRE AT SCREEN 20
  IF ENEMIES_KILLED_EQUALS 60
  THEN
    SET TILE (2, 7) = 0
    SET TILE (2, 8) = 0
  END
  IF PLAYER_IN_X 0, 15
  THEN
    WIN
  END
END

```

Let's recap to make it clear. Let's see what would happen, step by step. Imagine that we got to the 20th screen after having killed all the bad guys. Here is the sequence of events:

1. Upon entering the screen 20, after drawing and such, the section runs **ENTERING SCREEN 20**. In it, we found that 60 killed enemies, which is true, and the barrier is removed. In addition, a fire zone 15 pixels wide on the left side of the playing area is defined
2. The main loop of the game is executed. The player plays and such and such sees the barrier open, and heads to the left.
3. When the player enters the zone of fire, the section runs **PRESS_FIRE AT SCREEN 20**. In it, we found that 60 killed enemies and the barrier is removed. This is redundant and could be avoided with a flag, but we do not care ... not noticeable. What matters is what happens next: check that the X coordinate of the player, in pixels, is between 0 and 15, which is true (as we have entered this section for entering the fire zone, which is defined right in that area), so running WIN and shown us the end of Game.

Is it caught? Do we see another? Do you want to see the flag to eliminate redundancy? Perfect.

We all free flags, so capture 1. The method is simple: we put it to 0 to enter the screen, 1 when we eliminate the barrier and remove the barrier only **PRESS_FIRE AT SCREEN 20** if 0. What I put everything together, you should be able to follow him alone:

```

ENTERING SCREEN
  IF TRUE
  THEN
  SET_FIRE_ZONE 0, 0, 15, 159
  SET FLAG 1 = 0
  END

  IF ENEMIES_KILLED_EQUALS 60
  THEN
  SET TILE (2,7) = 0
  SET TILE (2, 8) = 0
  SET FLAG 1 = 1
  END
END

PRESS_FIRE AT SCREEN 20
IF ENEMIES_KILLED_EQUALS 60
IF FLAG 1 = 0
THEN
SET FLAG 1 = 1
SET TILE (2, 7) = 0
SET TILE (2, 8) = 0
END

IF PLAYER_IN_X 0, 15
THEN
WIN
END
END

```

Now, let's see another.

Example: Sgt. Helmet Training Day

Let's now see a script a little longer, but equally simple. In this game, the mission is to collect the five bombs, bring them to the computer screen (screen 0) to deposit them, and then return to the beginning (screen 24).

There are many ways to do this. The one we use to assemble them is quite simple:

We can count the number of objects we carry from the script, so the pumps will be normal and current engine objects. We place them with the setter as type 1 hot-spot.

When we get to the computer screen, we will make a cool animation by placing the bombs around. We use the discussion this because it helps to know the coordinates of each square (if you put the mouse on a square the coordinates come out at the top of everything) and we point at a paper where we are going to paint them.

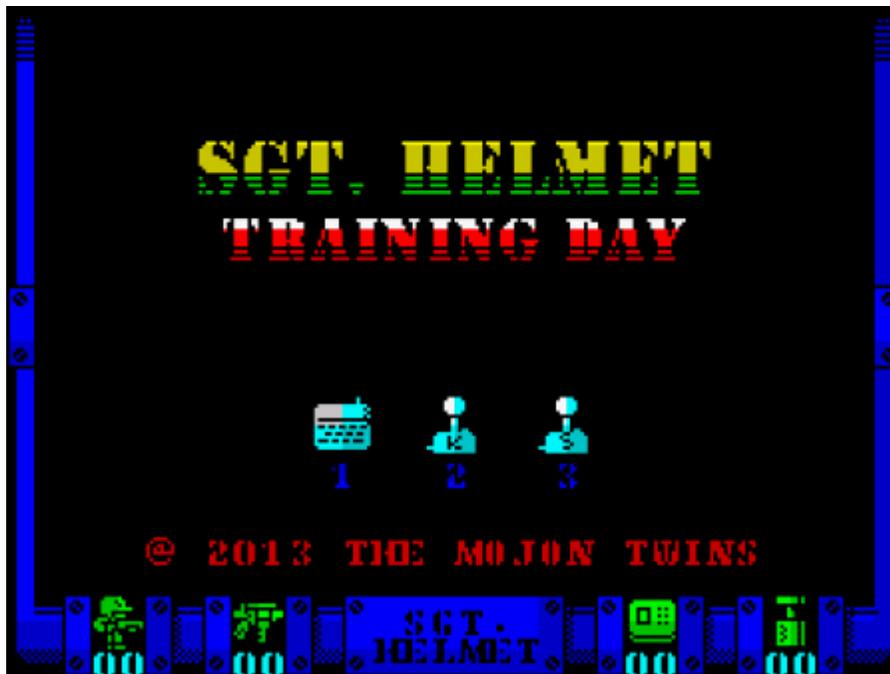
We will use flag 1 to check that we have placed the bombs. At the beginning of the game, it will be worth 0, and we will put it to 1 when we put the bombs.

When we enter screen 24, which is the main screen, we will check the value of flag 1, and if it is 1, the game will end.

In addition, we will be printing texts on the screen with what we are doing. Recall that in **config.h** there were three directives that we mentioned above makes some chapters:

```
#define LINE_OF_TEXT 0 // If defined, scripts can show text @ Y = #  
#define LINE_OF_TEXT_X 1 // X coordinate.  
#define LINE_OF_TEXT_ATTR 71 // Attribute
```

They are used to configure where a line of text comes from which we can write the script with the **TEXT** command. For this, we leave free space in the frame: notice how there is room in the row above since we have configured the line of text in the coordinates $(x, y) = (1, 0)$.



The first thing our script will therefore be to define a pair of messages that appear by default when entering each screen, depending on the value of flag 1. We do this in the section **ENTERING ANY**. This section, runs to enter each screen, just before the section **ENTERING SCREEN n** section. Attention to this: will allow us to define a general text that can easily overwrite if needed for any particular screen, because if we **SCREEN n ENTERING** text which overwrites **ENTERING ANY** positions in the run later.

To print text in the defined text line, we use the **TEXT** command. The following text goes without quotes. We will use the underscore character to represent spaces. It is also convenient to fill with spaces so that if you have to overwrite a long text with a short one, delete it entirely.

The maximum length of the texts will depend on your playing frame and how you have defined your position. In our case we have placed it in $(x, y) = (1, 0)$ because we have border left and right, so the maximum length will be 30 characters.

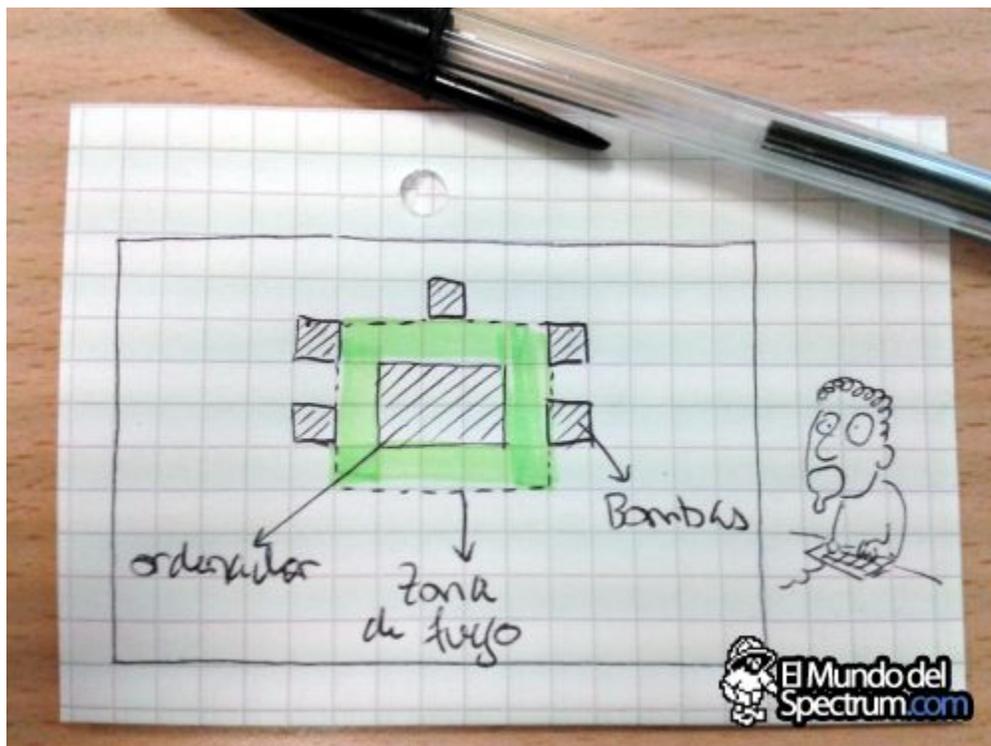
Let's write our **ENTERING ANY** section, then. We have said that we will print one text or another depending on the value of flag 1:

```
ENTERING ANY
  IF FLAG 1 = 0
  THEN
    TEXT BUSCA_5_BOMBAS_Y_EL_ORDENADOR!
  END

  IF FLAG 1 = 1
  THEN
    TEXT MISSION_CUMPLED! _VUELVE_A_BASE
  END
END
```

There is no mystery, right? If the flag 1 is 0, that is, the situation at the beginning of the game (all flags set to 0 at the beginning), entering each screen displays **"Find 5 BOMBS AND COMPUTER"** in the area of the defined framework For the text line. If the flag 1 is 1, which will happen when we place the bombs on the computer, the default text that appears when entering the screens will be **"Mission Accomplished! RETURNS TO BASE"**.

Let's make a sketch of the screen to see where the computer goes and the pumps:



Let's go with the chicha now. The first thing we will do is write the conditions for the computer screen, which is the screen 0. In this screen, we have to do several things. Let's get clear before we begin:

Whenever we enter we will have to paint the computer, which is composed of tiles 32 to 38 of the tileset. We will do it as we have seen, with SET TILE.

In addition, we will have to define a fire area around the computer for the game to automatically detect when we approach it.

If we re-enter the screen after having placed the bombs (can happen), we will have to cope with it and also paint the bombs.

If we for the first time (we have not put bombs) help write a little message that says "**GRAB THE FIVE PUMPS AND RUN**"

If we approach the computer, it will be necessary to make the animation cool to put the bombs, and also place the flag 1 to 1.

Now that we have some experience, we will realize that the first four things are done when entering the screen, and the last one when you press action (or enter the fire zone). Let's go one by one. Let's start with the things to do when entering this screen. I like to start with the things that have to be done always: paint the computer and define the fire zone:

```
ENTERING SCREEN 0
# Always: paint the computer.
IF TRUE
THEN
  SET TILE (6, 3) = 32
  SET TILE (7, 3) = 33
  SET TILE (8, 3) = 34
  SET TILE (6, 4) = 36
  SET TILE (7, 4) = 37
  SET TILE (8, 4) = 38
  SET _FIRE_ZONE 80, 32, 159, 95
END
```

If you look, we have used the formulas explained above to define a wide area around the computer. Specifically, the area is the rectangle formed from the tile $(x, y) = (5, 2)$ to $(9, 5)$. That is a flange of a tile around the six tiles that occupy the computer. Take a role of squares and you lease less. We continue: if we enter after having placed the bombs (something that can happen) we will have to leave and paint the pumps. Nothing simpler:

```
# If we have already put the bombs: paint them.
  IF FLAG 1 = 1
  THEN
    SET TILE (4, 4) = 17
    SET TILE (4, 2) = 17
    SET TILE (7, 1) = 17
    SET TILE (10, 2) = 17
    SET TILE (10, 4) = 17
  END
```

We have looked at the sketch, of course, to know the positions of the bombs. The bomb is on tile 17, which is the tile used to paint objects if you remember.

Now it is only necessary to put a text of help if we have not yet placed the bombs. Note that happens what we said, as this section is executed after **ENTERING ANY**, we print the text here will overwrite the already had. That is why, in addition, we use blank spaces around: center the text and delete the characters from the previous text, which is longer:

```
# If not, message.
  IF FLAG 1 = 0
  THEN
    TEXT _PON_LAS_CINCO_BOMBAS_Y_CORRE_
  END
END
```

Ready. Now only react to the fire zone in **PRESS_FIRE AT SCREEN 0** section. We will make some checks and then we will make the animation:

```
PRESS_FIRE AT SCREEN 0
  IF PLAYER_IN_X 80, 159
  IF PLAYER_IN_Y 32, 95
  IF OBJECT_COUNT = 5
  IF FLAG 1 = 0
  THEN
    SET FLAG 1 = 1
    SET TILE (4, 4) = 17
    SHOW
    SOUND 0
    SET TILE (4, 2) = 17
    SHOW
    SOUND 0
    SET TILE (7, 1) = 17
    SHOW
    SOUND 0
    SET TILE (10, 2) = 17
    SHOW
    SOUND 0
    SET TILE (10, 4) = 17
    SHOW
    SOUND 0
    TEXT ____ AHORA_VUELVE_A_LA_BASE ____
  END
END
```

Let's see it little by little, because there are new things:

The first thing is to check that we are where we have to be (the player can always press the action key instead of entering the fire zone, and do not spring it if the player is anywhere). We do this as we have seen: with **PLAYER_IN_X** and **PLAYER_IN_Y** and the same coordinates of the fire zone.

The next thing is to check that we have the five bombs, or what is the same thing, that we have five objects. This is done with **OBJECT_COUNT**, representing the number of objects that the player wears collected.

Finally, very important, we must check that we have not left the bombs, or fun things could happen. If all these conditions are fulfilled, we will put the flag 1 to 1 (we have already put the bombs) and we make the animation, which consists of painting the bombs one by one and playing a sound. You see there the **SHOW** command, necessary because the changes we make on the screen will not be visible until it is updated, which usually happens when returning to the main loop, but not in the middle of executing a clause. As we want to see each pump just after paint, call **SHOW**. Each sound will also stop the execution for a few moments (we are in 48K mode), which is great. Finally, we will print a help text, again with spaces to the sides to complete the maximum 30 characters and erase what was from the previous text.

And with this, we have finished all that had to be done on screen 0.

If we continue with our script, the next thing we had to do was return to the initial screen, which is 24. What remains to be done is quite simple: it consists of checking, on entering screen 24, that flag 1 is 1

This will only happen if we have previously placed the bombs, so we do not need more ... We just checked that and, if it is true, we finished the game successfully... Nothing simpler than doing this:

```
ENTERING SCREEN 23
  IF FLAG 1 = 1
  THEN
    WIN
  END
END
```

Hey! We already have the game programmed. In the script of Sgt. Helmet there is one more detail: our usual "seeing motorcycle". But I leave that already, there is nothing special: print tiles, define fire zone, detect position, and write a text. All that you know to do already.

I could go on, but we'd better leave it for now. In the next chapter we will see step by step examples, but already with more complex scripts such as, for example, the Cadàveriön. Then we will continue to see interesting things, such as 128K games, change the music and effects, compressed phases... Ugh, we will never end.

Meanwhile, you remember in the **motor_de_clausulas.txt** file on **/script** can view a list of available commands and checks, in case you are curious.

Until another!

Create your own Spectrum game Workshop(Version Changes)

Version 3.99.2

Come on, the churreras are going out like churros. We're breaking it, and We can think of new things every day. We'll get them going as we We can think of games that take them.

These are the new things that are in this version of the churrera:

Timers

Added to the churrera a timer that we can use automatically Or from the script. The timer takes an initial value, counts toward down, can be recharged, can be set every how many frames is decremented Or decide what to do when it runs out.

```
#define TIMER_ENABLE
```

TIMER_ENABLE includes the code required to operate the timer. This code will need some other directives that specify the form of function:

```
#define TIMER_INITIAL 99  
#define TIMER_REFILL 25  
#define TIMER_LAPSE 32
```

TIMER_INITIAL specifies the initial value of the timer. The time, which are set with the setter as type 5 hotspots, will recharge the value specified in TIMER_REFILL. The maximum value of the timer, both for the as recharging, is 99. To control the amount of time elapses between each decrement of the timer, we specify in TIMER_LAPSE the number of frames that must pass.

```
#define TIMER_START
```

If TIMER_START is set, the timer will be active from the beginning.

We also have some directives that define what will happen when the timing to zero. It is necessary to uncomment those that apply:

```
#define TIMER_SCRIPT_0
```

Defining this, when it reaches zero the timer will execute a section script special, ON_TIMER_OFF. It is ideal for carrying all the control of the timer by scripting, as it happens in Cadáverion.

```
// #define TIMER_GAMEOVER_0
```

Defining this, the game will end when the timer reaches zero.

```
// # define TIMER_KILL_0
// # define TIMER_WARP_TO 0
// # define TIMER_WARP_TO_X 1
// # define TIMER_WARP_TO_Y 1
```

If `TIMER_KILL_0` is set, a life will be subtracted when the timer reaches zero. If, in addition, `TIMER_WARP_TO` is defined, it will also be changed to the screen the player appears in the coordinates `TIMER_WARP_TO_X` and `TIMER_WARP_TO_Y`.

```
// # define TIMER_AUTO_RESET
```

If this option is set, the timer will return to maximum after reaching zero automatically. If you are going to perform the control by scripting, better leave it commented

```
#define SHOW_TIMER_OVER
```

If this is defined, in the case that we have defined either `TIMER_SCRIPT_0` or well `TIMER_KILL_0`, a "TIME'S UP!" Poster will be displayed. When the timer reaches zero.

Scripting:

As we have said, the timer can be administered from the script. Is interesting that, if we decided to do this, let's activate `TIMER_SCRIPT_0` so that when the timer reaches zero, the `ON_TIMER_OFF` section of our script and that the control is total.

In addition, these checks and commands are defined:

Checks:

```
IF TIMER >= x
IF TIMER <= x
```

Which will be fulfilled if the value of the timer is greater than or equal to or less or equal than the specified value, respectively.

Commands:

```
SET_TIMER a, b
```

It is used to set the `TIMER_INITIAL` and `TIMER_LAPSE` values from the script.

```
TIMER_START
```

It is used to start the timer.

```
TIMER_STOP
```

It is used to stop the timer.

Control of push blocks

We have improved the engine so that more can be done with the tile 14 of type 10 (pushable tile) that simply push it or stop the trajectory of the enemies. Now we can tell the engine to launch the PRESS_FIRE section of the current screen just after pushing a pushable block. Besides, the number of the tile that is "stepped" and the final coordinates are stored in three flags that we can configure, to be able to use them from the script to do checks.

This is the system that is used in the script of Cadàveriön to control that put the statues on the pedestals, to give an example.

Recall what we had so far:

```
#define PLAYER_PUSH_BOXES
#define FIRE_TO_PUSH
```

The first one is necessary to activate the pushable tiles. The second obliges the player to press FIRE to push and therefore is not mandatory. Let's see now the new directives:

```
#define ENABLE_PUSHED_SCRIPTING
#define MOVED_TILE_FLAG 1
#define MOVED_X_FLAG 2
#define MOVED_Y_FLAG 3
```

Enabling ENABLE_PUSHED_SCRIPTING, the tile to be pressed and its coordinates will store in the flags specified by the MOVED_TILE_FLAG,

MOVED_X_FLAG and MOVED_Y_FLAG. In the code shown, the tile is will store in flag 1, and its coordinates in flags 2 and 3.

```
#define PUSHING_ACTION
```

If we define this, in addition, the PRESS_FIRE AT ANY and PRESS_FIRE of the current screen.

We recommend to study the Cadàveriön script, which, besides being a good example of the use of the timer and the pushbutton control, results be a rather complex script that employs a lot of advanced techniques.

Check if we exit the map

It is advisable to put limits on your map so that the player can not exit, but if your map is narrow you may want to take advantage of the screen. In that case, you can activate:

```
#define PLAYER_CHECK_MAP_BOUNDARIES
```

It will add checks and will not let the player leave the map. eye! If you can avoid using it, the better: you will save space.

Type of enemy "custom" gift

Until now we had left the type 6 enemies without code, but we have thought that it is not difficult for us to put one, for example. It behaves like the bats of Cheril the Goddess. To use them, place them in the Of enemies as type 6 and uses these directives:

```
#define ENABLE_CUSTOM_TYPE_6
#define TYPE_6_FIXED_SPRITE 2
#define SIGHT_DISTANCE 96
```

The first one activates them, the second defines which sprite to use (minus 1, if you want the sprite of enemy 3, put a 2. Sorry for the slut, but saving bytes). The third one says how many pixels you see from far away. if he sees you, he follows you. If not, return to your site (where you've put it With the setter).

This implementation, in addition, uses two directives of the enemies of type 5 to operate:

```
#define FANTY_MAX_V 256
#define FANTY_A 12
```

Define there the acceleration and the maximum speed of your type 6. If you go to also use type 5 and you want other values, be a man and modify the engine.

Keyboard / joystick configuration for two buttons

There are side view games that are best played with two buttons. If you activate this directive:

```
#define USE_TWO_BUTTONS
```

The keyboard will be the following, instead of the usual one:

```
A = left
D = right
W = up
S = down
N = jump
M = shoot
```

If joystick is selected, FIRE and M fire, and N skips.

Shooting up and diagonally for side view

Now you can let the player shoot up or diagonally. To do this define this:

```
#define CAN_FIRE_UP
```

This configuration works best with USE_TWO_BUTTONS, since this separates "Top" of the jump button.

If you do not hit "up", the character will fire to where he is looking. Yes press "up" while shooting, the character will shoot up. Yes, in addition, you are pressing an address, the character will shoot on the diagonal indicated.

masked bullets

For speed, the bullets do not wear masks. This works fine if the background on which they move is dark (few active INK pixels). But nevertheless, there are situations where this does not happen and looks bad. In that case, we can activate masks for bullets:

```
#define MASKED_BULLETS
```

Version 3.99.2mod

This was a special version with a thing that Radastan asked us, the...

Animated Tiles

Everything is based on tilanim.h. This file is included if config.h is defined in ENABLE_TILANIMS directive. In addition, the value of this directive is what defines the number of smaller tile that is considered animated.

In tilanim.h there are, in addition to the definition of data, two functions:

Void add_tilanim (unsigned char x, unsigned char y, unsigned char t) is called from the function that paints the current screen if it detects that the tile that you are going to paint is >= ENABLE_TILANIMS. Add an animated tile to the list tiles.

Void do_tilanim (void) is called from the main loop. Basically select a random animated tile among all the stored ones, change the frame (from 0 to 1, from 1 to 0) and draws it.

To use it, you just have to define the ENABLE_TILANIMS directive in config.h with the smaller animated tile. For example, if your last four tiles (8 in total) are animated, put the value 40. Then, on the map, you have to put the smaller tile of the pair, that is, tile 40 for 40-41, the 42 to 42-43... If you do not do that, funny things will happen. The code is (It has to be) minimal, do not check anything, so take care.

By the way, this has not been proven. If you put it in your games, damages one touch.

Version 3.99.3

Animated Tiles

If it is defined:

```
#define ENABLE_TILANIMS 32 //
```

If defined, animated tiles are enabled. // the value specifies first animated tile pair.

In config.h, tiles >= that specified index are considered animated. in the tileset, they come in pairs. If, for example, "46" is defined, then the only pair of tiles 46 and 47 will be animated. The engine will detect them and each frame will cause one of the tiles 46 to change state.

There can be up to 64 animated tiles on the same screen. If you put more, will choke.

128K Mode

You have to do a lot of manual work with this. I'm sorry, but it's like that. In first you will have to create a make.bat that will build everything you need. For that you can rely on the file spare / make128.bat and adapt it to your project.

The 128K mode is the same as the 48K but use WYZ Player and also supports several levels. You can not have longer levels, but you can have several levels.

To use it, you need to activate three things in config.h:

```
#define MODE_128K // Experiment!  
#define COMPRESSED_LEVELS // use levels.h instead of map.h and enems.h (!)  
#define MAX_LEVELS 4 // # of compressed levels
```

In MAX_LEVELS you have to specify the number of levels you are going to use.

In churromain.c you have to change the position of the pile and place it below of the main binary:

```
#pragma output STACKPTR = 24299
```

Then you have to modify levels128.h, which is where the level structure is defined and is included in 128K mode. There you will see an array levels, with information about the levels. In principle, very little information is included:

```
// Level struct  
LEVEL levels [MAX_LEVELS] = {  
3,2,  
(4,3),  
{5.4},  
{6.5}  
};
```

The first value is the resource number (see below) that contains the level. The second value is the song number in WYZ PLAYER that should ring while is played at level.

To prepare a level, you have to use the new buildlevel.exe utility In / utils. This utility takes the following parameters:

```
$ Buildlevel map.map map_w map_h lock font.png work.png spriteset.png  
Extrasprites.bin enems.ene scr_ini x_ini y_ini max_objs enems_life behs.txt level.bin
```

- Map.map Is mappy mappy
- Map_w, map_h Are the dimensions of the map on screens.
- Lock 15 for autodetect locks, 99 if there are no locks
- Font.png is a 256x16 file with 64 ascii characters 32-95
- Work.png is a 256x48 file with the tileset
- Spriteset.png is a 256x32 file with the spriteset
- Extrasprites.bin you find it in / levels
- Enems.ene the file with the enemies / hotspots of colocador.exe
- Scr_ini, scr_x, scr_y, max_objs, enems_life level values
- Behs.txt a file with tile types, separated by commas
- Level.bin is the output file name.

When we have all levels built, we have to compress them with apack:

```
$ /utils/apack.exe level1.bin level1c.bin...
```

When we have all levels compressed, we will have to create the images binary files that will be loaded into the extra RAM pages. For that we use the utility librarian that is in the folder / bin. In fact, it is a good idea to work Folder / bin for this.

The librarian utility uses a list with the compressed binaries that should be getting into the binary images that will go on the extra pages of RAM. The first thing we will have to put in is the title.bin, marco.bin and ending.bin, in that order. If you do not have .bin you should use a length 0, but you must specify it. Then we will add our levels. For example:

```
title.bin
marco.bin
ending.bin
level1c.bin
level2c.bin
level3c.bin
level4c.bin
```

There we added four compressed levels.

When you run librarian, you will be filling in 16K images destined to go in the Extra RAM. First it will create ram3.bin, then ram4.bin and finally ram6.bin, according to I need more space.

It will also generate the file librarian.h, which we will have to copy to / dev. Here we can see the resource number associated with each binary:

```
RESOURCE resources [] = {
    {3, 49152}, // 0: title.bin
    {3, 50680}, // 1: marco.bin
    {3, 50680}, // 2: ending.bin
    {3, 52449}, // 3: level1c.bin
    {3, 55469}, // 4: level2c.bin
    {3, 58148}, // 5: level3c.bin
    {3, 60842} // 6: level4c.bin
};
```

These resource numbers are the ones we will have to specify in the array levels mentioned above. In particular, resources 3, 4, 5 and 6 are those containing the four levels.

With all this done and prepared, we will have to mount the tape. For this there are which create a suitable loader.bas (you can see an example in /spare/loader.bas) and build a .tap with each block of RAM (again, the example in /spare/make.bat builds the tape with binaries in RAM3 and RAM4).

You will also need RAM1.BIN to build RAM1.TAP, containing the player of WYZ with songs. For this you have to modify /mus/WYZproPlay47aZX.ASM in / mus to include your songs. You have an example in / spare.

As you can see, it's a bit tedious. I recommend that you build mini-projects in 48K as you make the levels, and finally you build a 128K version with all.

In addition, you can use extra space to push more compressed screens, or even code to use passwords to jump directly to levels. You can see examples of all this in Goku Mal 128.

Type 3 Hotspots

We have made this modification, proposed in the forum, fixed to blow of directive. If you define

```
#define USE_HOTSPOTS_TYPE_3 // Alternate logic for recharges.
```

The recharges will appear only and exclusively where you place them, using the type 3 hotspot.

Pause / Abort

If it is defined

```
#define PAUSE_ABORT // Add h = PAUSE, y = ABORT
```

Code is added to enable the "h" key to pause the game and the key "And" to interrupt the game. If you want to change the assignment you will have to Touch the code in mainloop.h

Message catching objects

If it is defined

```
#define GET_X_MORE // Shows "get X more"
```

A message will appear with the items you have left each time you take one.

Version 3.99.3b

Minimal revision. It is arranged to be able to have 128K games with only one level (ie use MODE_128K without COMPRESSED_LEVELS).

Right now there are two examples that can help you if you want to make a 128K game:

- Goku Mal: 128K with compressed levels. See this doc and the sources of the game.
- The new adventures of Dogmole Tuppowksy: 128K with only one level, more info in the forum of mojonía.

Also, in spare I added the file extern-texts.h whose contents you can use in extern.h if you want an easy way to display text on the screen using the EXTERN command n of the script.

Version 3.99.3c

Item Engine

You can use items and you can have the user select them with the "Z" key. To activate them, just have the scripting enabled and define an ITEMSET section at the beginning of the script similar to this:

```
ITEMSET
# Number of holes:
SIZE 6

# Position x, y
LOCATION 1, 21

# Horizontal / vertical, spaced
DISPOSITION HORZ, 3

# Color and characters to paint the selector
SELECTOR 66, 8, 9

# Flag containing which gap is selected
SLOT_FLAG 20

# Flag containing what object is in the selected hole
ITEM_FLAG 21
END
```

Then we will have checks and commands to handle the items:

```
* IF PLAYER_HAS_ITEM t
Description: Evaluate to TRUE if the player has tile item T in their inventory.
Opcode: 01 x

* IF PLAYER_HASN'T_ITEM x
Description: Will evaluate to TRUE if the player does NOT have tile item T in their inventory.
Opcode: 02 x
```

EYE! The two previous ones put enough code of interpreter. It is better not to use them if it can be avoided. There are better ways to manage inventory:

```

* IF SEL_ITEM = t
Description: TRUE if the selected item is T
Opcode: 10 ITEM_FLAG t
The generated code is equivalent to IF FLAG ITEM_FLAG = t

* IF SEL_ITEM <> t
Description: TRUE if the selected item is not T
Opcode: 13 ITEM_FLAG t
The generated code is equivalent to IF FLAG ITEM_FLAG <> t

* IF ITEM n = t
Description: TRUE if slot N is T
Opcode: 04 n t

* IF ITEM n <> t
Description: TRUE if slot N is T
Opcode: 05 n t

* SET ITEM n = t
Description: Assign item t to slot n
Opcode: 00 x n

* REDRAW_ITEMS
Description: Force a redraw of items
Opcode: E7

```

For more info, visit this forum thread:

<http://www.mojontwins.com/mojoniaplus/viewtopic.php?f=9&t=1581>

Shooting / stepping enemies disable

The possibility of enabling the firing / stepping enemies according to the value is added of a flag (to be able to use it, shoot / step enemies must be activated in the engine, of course):

```

#define PLAYER_CAN_KILL_FLAG
#define PLAYER_CAN_FIRE_FLAG 1

```

For example, imagine that you decide that the flag that controls this behavior is the flag 5. First we would define this in config.h:

```

#define PLAYER_CAN_KILL_FLAG 5

```

Then, in the script, we would make sure that flag is 0 at the beginning of the game (Or 1, if we want to kill enemies from the beginning):

```
ENTERING GAME
IF TRUE
THEN
SET FLAG 5 = 0
END
END
```

Imagine that we want the player to start stepping on enemies when catch an object represented by the tile 24 and located on the screen 6, position (7,5). We could do something like this:

```
ENTERING SCREEN 6
# If we have not picked up the object we draw it
IF FLAG 5 = 0
THEN
SET TILE (7, 5) = 24
END
END

PRESS_FIRE AT SCREEN 6
# If we press ACTION on the object...
IF FLAG 5 = 0
IF PLAYER_TOUCHES 7, 5
THEN
# We delete it from the screen
SET TILE (7, 5) = 0
# We activate the ability to kill enemies
SET FLAG 5 = 1
END
END
```