LEGEND OF STEEL

MAKING OF

TOD STUDIOS

Content table

1 Motivation

We have always liked all kinds of video games and, moreover, despite the fact that nowadays technology allows us to create hyperrealistic graphics and increasingly sophisticated and complex games, we still like those that have made us have such good times since time immemorial, and which are the classics. For this reason, this project has been presented to us perhaps as the one we have enjoyed the most in the career.

We are three students in the 4th year of the degree in Computer Science at the University of Alicante studying the specialization itinerary in advanced computing. This itinerary has several subjects, including a so-called Automatic Reasoning taught by Dr. Francisco José Gallego Durán.

Francisco proposed to us as practice of the same one, to realize a video game in language assembler Z80, with the idea of competing in the CPCRetrodev of this year 2018. As good engineers, we were delighted with the challenge for two simple reasons: We love to play and develop video games, but especially because there was a month and a half left for the competition, we like challenges and this was not going to be trivial ;)

2 Origin of the idea

As we already know, Amstrad CPC is a platform for which there are many and varied games from its golden age back in the 80s and early 90s. But, in addition, lately it is increasing again its catalogue thanks to the new creations made by enthusiasts and professionals of the indie/retro scene. This makes it increasingly difficult to come up with a type of game that is no longer available on CPC.

However, something that perhaps has been missed has always been a good dungeon crawler with top-down view of Zelda style or similar. So, inspired by similar action mechanics, and taking into account the technological limitations, we have made a game design that will certainly give hours of fun to the player.

As for the theme is inspired rather in the movies of barbarians, with a touch of fantasy, where we can find everything from orcs to powerful sorcerers, through immortal beings and strange creatures that will do everything possible to prevent us from achieving our goal.

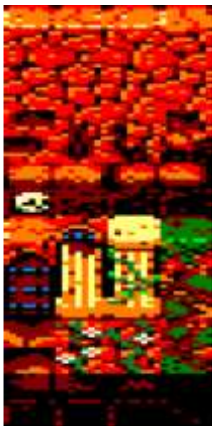
3 Graphic art

For the development of graphic art, with the restrictions of Mode 0: 160x200 in 16 colors we have opted for sizes 8 x 16 for the characters and 8 x 8 for the items, except special cases such as when our hero attacks that happens to require a size of 16 x 16.

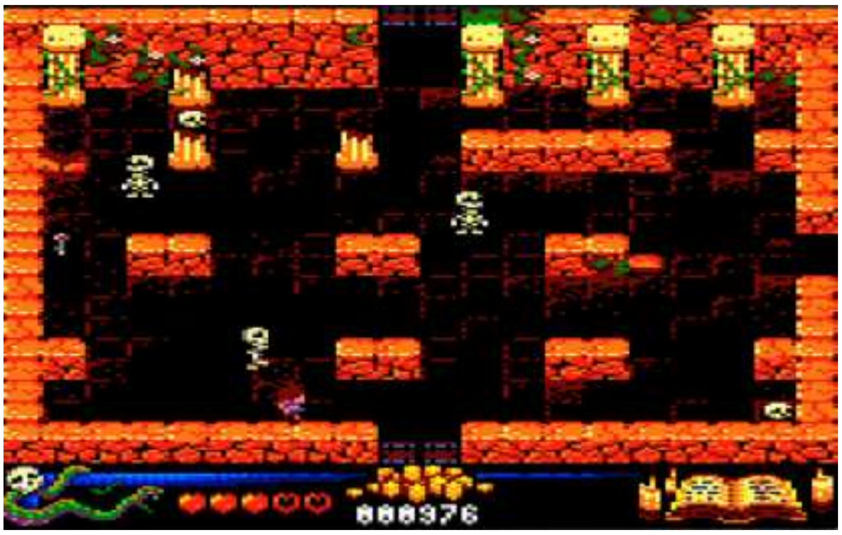
We have designed 3 worlds with 3 different tilesets: Dungeon, cavern and palace developed in several maps of 22 x 20 tiles each. As you can see, the tilesets corresponding to the caverns and the palace are half the size of the dungeon, in the end 64Kb are few and you have to trim a little of all places.

Dungeon:

Used tileset



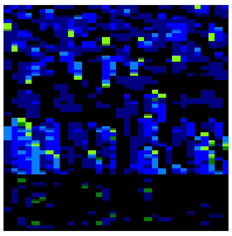
Result of one of the generated maps

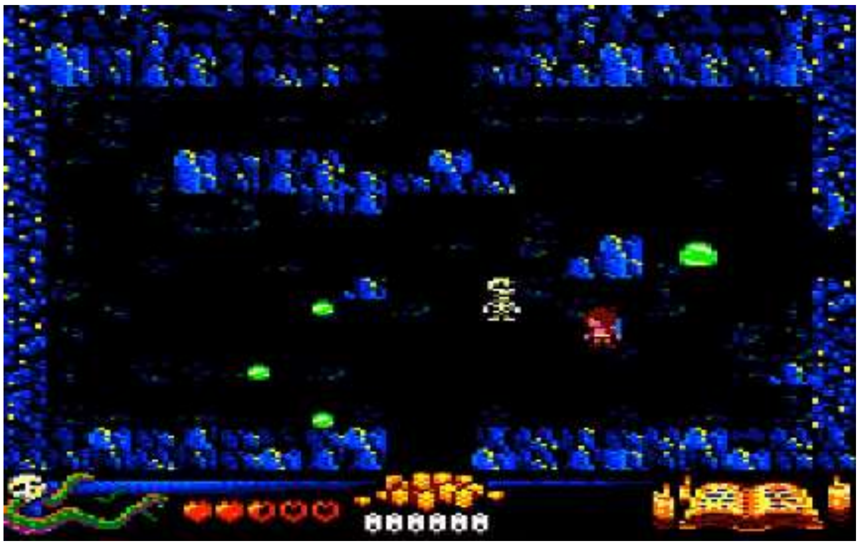


Something that is worth mentioning about this tileset is that, contrary to what happens in other similar 8-bit games, the tiles are designed so that the repetitions of these are not noticed, so that, for example, although there are repeated stones in the image, this cannot be appreciated at first sight.

Cavern:

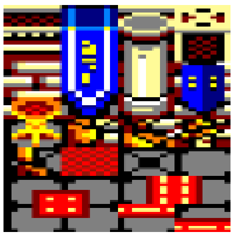
Used tileset *Result of one of the generated maps*

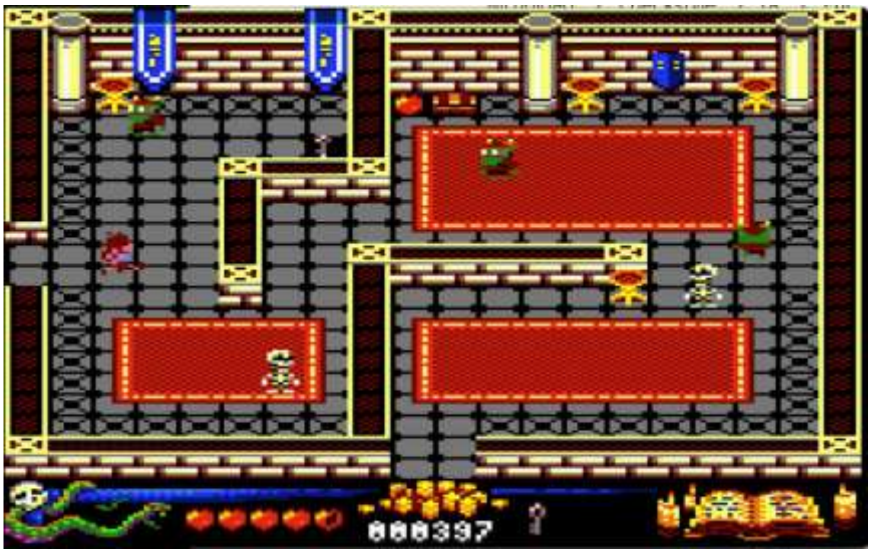




Palace:

Used tileset *Result of one of the generated maps*



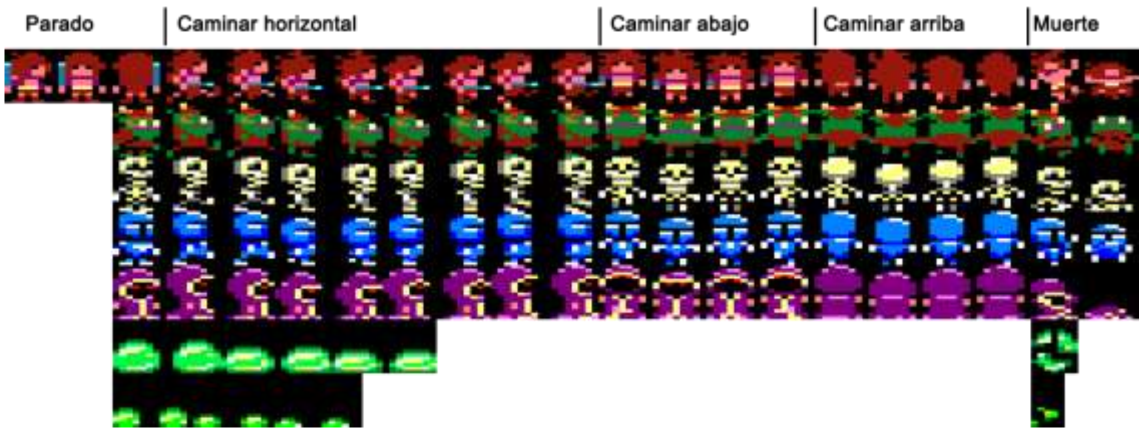


3.1 Animations

Most animations, consist of 4 frames that, in the case of the animation to run left or right, finally had to be done in 8 to get a smoother movement (see section "Video Limitations").

Animation tilesets for all characters

(By order: Main hero, orc, skeleton, knight, magician, big slimer, and small slimer)



Tilesets for the ítems

Heart, key, chicken, coins, chest, potion.

4 Music and sound

To develop music and sound effects we have used Arkos Tracker. For convenience we have chosen to use Arkos Tracker 1 for the music because for this version, CPCtelera provides an automatic converter in the compilation toolchain itself. For the effects we have used Arkos tracker 2 that, although it has forced us to adapt the files by hand, it allows us to separate player of effects from that of the music. The need for this will be seen later in the section "working with scarce memory".

We have used a Solton MS60 keyboard to shape the melodies before writing them into the program. We have also designed most of the instruments that compose it, as well as the instruments for in-game sound effects.

Challenges within this process include the difficulty we have encountered in shaping an instrument by trying to make it sound as close to what we have in mind with the adjustments the program offers. Although it is well documented, it is a process that requires experience and we were noticing this in the last ones we designed. In the first cases we have based ourselves on example instruments to which we have made modifications and later we have managed to create some of our own from scratch.

As for challenges within the melody, they are infinite. In the end, composing is a world that requires very specific knowledge and qualities. Within our possibilities we have tried to carry out an interesting and catchy melody making the most of the 3 channels offered by the AY-3-8912 which has the Amstrad CPC 464.

Finally we have developed 10 different effects for the in-game experience:

- Hero hits an enemy.
- Hit from an enemy to the hero.
- Death of the hero.
- Death of an enemy.
- Hero ends the game.
- Sword strike.
- Collect coins.
- Collect chicken/heart/potion.
- Collect key.
- Wizard throws fireball.

5 Programming

As we said at the beginning, the game is developed 100% in Z80 assembler. This has been the main factor responsible for the arduous and pronounced initial learning curve. However, as development progressed and thanks to the enormous amount of online help, we were able to pick up a pace and that allowed us to finish everything we had set out to do on time.

As for information to develop for CPC, for those of you who are just starting out, we fully recommend the videos of the "Curso de ensamblador desde cero" that Fran Gallego has on youtube:

https://www.youtube.com/watch?v=smwXc3vShZw&list=PLmxqg54iaXrijQi4-J9IkAWDEguKRX9Dh

Once we have finished at least the first two levels, practically everything we need to make a game like ours is in the videos corresponding to the courses of "Razonamiento Automático" and "Videojuegos I" of the University of Alicante, also taught by Fran Gallego and available on YouTube too:

https://www.youtube.com/watch?v=Ojac4Y4sxF0&list=PLmxqg54iaXrjhvQy_GbzYdvURV4sKs0CT

https://www.youtube.com/watch?v=13JGNTWcNLA&list=PLmxqg54iaXrjA0wzZmc_HDeZxs-vDwDFS

https://www.youtube.com/watch?v=k5Z_qLLM6hw&list=PLmxqg54iaXrjtcWWbRjv1JEzjFAZqRywi

5.1 The CPCtelera engine

It would not have been possible to finish any of this in a month and a half without this powerful Amstrad library.

CPCtelera allows to develop a game for CPC in C language or, if we prefer it, in assembler to have more control at low level and to be able to optimize to the maximum.

It has all kinds of high performance routines to draw sprites, tiles maps, text, play sound and music, receive input (keyboard and joystick), various functions of memory management, compression, tape loading, and so on, as well as a toolchain that allows us to compile our entire game with a single order, automatically importing all the assets we use (images, sounds, tiles…).

In our case we have used version 1.5 which is still under development, but which offers new features and substantially improves the performance of the previous version.

5.2 Special features

5.2.1 Double buffer

The graphics are sent to the screen via the CRTC chip of the Amstrad. This chip does this every 1/50th of a second by sweeping the entire screen from top to bottom.

To move a character around the screen, in each frame the first step is to erase the previous graphic and then draw it in the new position. If we just do this, it can happen that the CRTC passes at the wrong time. For example, we can erase before the chip sweep arrives but when we go to draw the CRTC has already passed and even if we paint again, the result will not be visible until the next frame.

This is what causes the famous flicker produced by working directly on the video memory. The first solution for this is not to paint at any time, but when the CRTC chip sends us the vertical synchronization signal, which indicates that it is currently reading at the top of the screen.

This solves much of the problem but if we have many graphics we can pass another effect even worse and is that at the top the graphics are never seen, as the delayed always gains speed to the processor.

The solution for this is to use double buffer. This technique consists of having two video buffers. One will be the front buffer (or currently visible buffer) and the other the back buffer (or hidden buffer).

The front buffer will be what we see on screen but we will always draw in the back buffer and when we have finished the graphics tasks we will tell the CRTC chip to exchange the pointers of these two buffers to show what we just modified. At this moment what was the front buffer becomes backbuffer, and it will be in this one where we will prepare the next frame while the current one is shown. This will be repeated in a loop and, as a result, the flickers and sprites that disappear will never be a problem again, at the expense of a large amount of RAM used.

5.2.2 Hardware scroll

As we have seen in the previous section, the CRTC chip allows us to modify the pointer that indicates the memory position that we want to use as video memory. This in Amstrad can be used to make a hardware scroll.

If we increase the position of the pointer in two bytes (due to hardware limitations we can only move two by two), the effect will be that the whole screen will move 4 pixels (being in 0 mode) to the left. And if we reduce its position in two bytes, we will do the same effect but to the right.

Vertical movement is achieved by varying the pointer this time +80 bytes to move up or -80 bytes to move down.

Of course, it's not enough to just move the pointer. Besides this we must paint the new row or column of tiles to make the scroll effect.

5.2.3 Vertical rupture

Besides changing the video pointer, the CRTC chip allows us to do many more things. In our case we have used it to apply a mythical effect in Amstrad, known as vertical rupture.

The objective of this effect is to divide the screen with an imaginary horizontal line where we can simultaneously show two video buffers, one in the upper half and another in the lower half. In fact, you could make more divisions and even have it painted on the edge of the screen making more area visible. The problem is that this would also increase the memory needed for the video.

For our game we simply needed to make two zones: a big one at the top (176 pixels high) for the game scene and a smaller one at the bottom (24 pixels high) for the panel.

The main problem with this effect is that, to achieve it, we must perform the buffer change operations at the exact moment and without skipping the process at any time since we started it for the first time, permanently invoking these routines in a perfectly synchronized way in each frame throughout the game.

This may seem too complicated (and in fact it is not simple at first), but in Amstrad we have the advantage of having a system of interruptions that, preparing it conveniently allows us to assign a routine that will be executed at regular intervals and at predictable moments. Specifically, the CPC launches 6 interrupts at every 1/50th of a second. We can take advantage of this to make the buffer changes we are talking about since each one of these 6 interruptions will correspond to a moment in which the CRTC sweep is in a specific position on the screen. It is already our decision which one to wait for to make the division.

5.3 AI of enemies

5.3.1 Sensors for proximity detection with walls

Enemies walk all over the mapping, but when they must follow a certain route and have a wall in front of them, they don't advance until they collide with it, but they implement an intelligence that makes them keep a certain margin with the walls. This is what anyone in the real world would do... and that is that in reality we don't bump into the walls when we walk... or at least not intentionally!

Para ello, todos los enemigos tienen un sensor de proximidad. En este caso, se activan los estados correspondientes para modificar la dirección del enemigo en función de su destino.

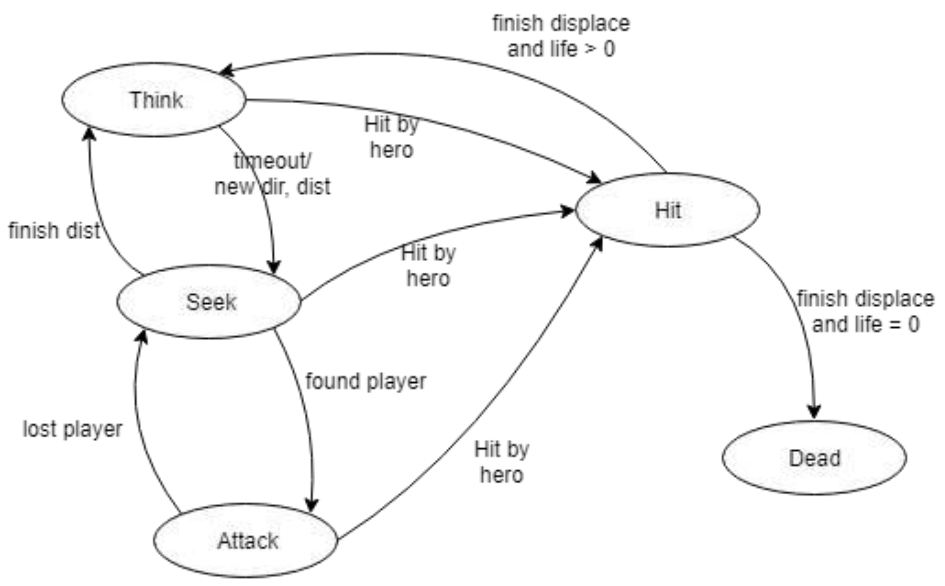
5.3.2 Pathfinding

The basic routine of an enemy will be to walk randomly through the scenario until he sees the player. An enemy can't see through the walls, so we won't need a complex path search algorithm here because his target will be a visible point from his position.

Once the enemy has located the player he will go after him as explained in the next section.

5.3.3 State machines

As for the intelligence of the enemies, it has been designed as a base a common behavior to all of them from which the necessary variations are added depending on the type of enemy. For this basic behaviour, a state machine has been created as shown in the following image:



As you can see, there are 5 base states, **Think, Seek, Attack, Hit** y **Dead**.

- The **Think** state chooses a random direction taking into account that it can move in that direction, using a sensor that detects the proximity to the walls and that will be explained later. In addition to the direction also takes a random distance to walk trough.

- The **Seek** state moves the enemy using the direction and distance chosen in the **Think** state. If during the movement it encounters a wall with the sensor, it returns to the **Think** state to choose another direction. This prevents enemies from continually colliding with the walls. On the other hand, if you detect the hero at less than 8 squares away, the Manhattan distance changes to **Attack** state. It is important to note that if the hero is less than 8 squares away, but there is a wall in front of him, he ignores the hero and maintains **Seek** state. Finally, once you have completed the distance chosen in the **Think** state, return to that state to choose another new path.

- The **Attack** state follows the hero along the Manhattan road and, if he ever loses sight of the hero, he returns to the **Think** state. Also, check to see if he has collided with the hero. In that case, you put the hero's state in **Hit** and switch to **Think** state.

- The **Hit** state displaces 3 tiles and decreases the life of the enemy. In case of being without life it would pass to the **Dead** state, otherwise it would pass to the **Think** state.

- The **Dead** state only performs the death animation and remains permanently in that state.

All enemies inherit the base behavior described above, but each one has some variations to get a different behavior.

- **The skeleton** refines the **Dead** state by adding a 4-second counter that, when finished, returns to the **Think** state, meaning that it "resurrects" because it is a skeleton and cannot die.
- **The knight** has no refinements in terms of states, but to get to reduce the speed we update his logic of states every 2 times, this generates a halving of speed compared to the rest of enemies.

- **The wizard** has a refinement of the **Attack** state which consists of a new machine of states within that state. The states of that new machine are **Escape, Align** and **Throw**.
    - The **escape** state is activated if the hero is less than 4 squares away and makes the magician escape from the hero by the axis that is less aligned from him.
    - The **align** state is activated if the hero is between 4 and 8 squares away and what it does is move the magician in the axis that is more aligned to align it with the hero either vertically or horizontally. This is to look for a shooting position towards the hero (as this enemy throws fireballs).
    - The throw state is activated if the hero is between 4 and 8 squares away and is aligned on one of the axes and what it does is throw a fireball in the direction of the hero.
    - Finally, if it is less than 2 squares away from the hero, the **Attack** base state is activated, as explained above.
- The **Slimer** has the following changes:
    - In **Dead** state, instead of dying, create 3 smaller slugs, which in turn when you move to **Dead** state definitely die.
    - On the other hand, it has a **Sticked** state. This state is activated if you collide with the hero by reducing the speed (following the same technique as in the knight) and has a counter in which every second takes life of the hero. To exit the **Sticked** state the player must move left and right quickly.

5.4 Working with scarce memory

The Amstrad CPC464 for which this game has been designed has only 64Kb of RAM distributed by default as follows:

This is a real problem if you want to make a game of these features with well varied graphics and multiple enemy types with differentiated logics. In our case, as we have already seen, we have 3 big tilesets from which a good number of maps is generated, the game has 3 phases, 5 types of enemies + the main character, and 6 types of items.

To all this, we need to add a rich graphics menu, including a colourful pre-game tutorial and a main theme of a minute and a half duration.

In addition, we must take into account the code that moves all this and the necessary functions of the CPCtelera library that makes it possible. A priori it may seem unfeasible that all this will fit in 64Kb, and is that, given the level of detail of the graphics, not even the compression routines managed to reduce the space occupied in a sensitive way.

But if all this still seems like a small problem, we still have to take into account that the game uses double buffer for the fluid and flicker-free rendering of the graphics.

For those who don't know, the video memory of the Amstrad is part of the main RAM. To keep the graphics on screen, we must have 16Kb reserved from the 64Kb available, so we would have 48Kb, but using double buffer we need 32Kb for video and surprise ... We only have 32Kb left for the game!

If we talk about real numbers, our game actually occupies more than 45Kb, so a priori does not fit whole.

5.4.1 First solution

The solution has been to load the menu in the double buffer area and, during this part of the program, only the video memory will be drawn directly (without double buffer). Once the game starts, the code and menu graphics will be overwritten with the back buffer. We are not going to go back to the main menu once the game has started, because in case we want to restart the game or continue from a saved point, there is an option to do so at the end of the game.

This has given us a lot more space available but, even so, it hasn't been enough, because in the end, only the block of the game just surpasses the 32Kb we had.

5.4.2 Reclaim even more space (and without trimming content)

The maps of tiles that are shown are compressed and what is done at every moment is to decompress in a reserved zone the one that is going to be shown.

Well, to save the space we lacked to enter the 32Kb what has been done is to use this area for the routines and functions of CPCtelera that were only used during the initialization of the system and the game (all those that are only used once). So, when the game starts, it reuses that area to decompress the maps, so this is implemented without memory cost.

Finally the game has fit, but we only have 14 bytes left! ... so that the memory has been more than well used ;-)

5.5 Working with limited process capacity

The CPC has a Z80 processor that works at approximately 4Mhz this limits what can be done but, even so, as we already know many things can still be done.

Our game manages to work at 50 fps with up to 9 entities moving on screen. This framerate is also not reduced when scrolling, since it has been implemented by hardware (as explained above).

Sometimes, the number of entities in the scene can cause us to skip a frame. In this case, being synchronized with the monitor's retrazing, the result is that we go down directly to 25fps. However, this does not cause the movements to become slow in the game, but this situation is detected and increases in positions are automatically doubled to adjust the speed of the game to the current fps.

This makes the speed of the game independent of the load it has to execute.

5.6 Video system limitations

The video mode chosen for the graphics has been mode 0 because, although it has less resolution gives us more colors (in total 16). In this mode, each byte of the video memory represents two pixels. This is a problem, because when we move a sprite one position horizontally, does not move one pixel but two.

We wanted to polish this detail by getting it to move pixel by pixel at the cost of consuming more memory.

To solve it, what we do is, for the animations of walking in horizontal of the characters, instead of using 4 frames like the others, we use 8. The 4 extra frames are exact copies of the previous ones, only moved one pixel to the right. In this way, when we move a character, we alternate these duplicate frames so that, although the graphic moves two by two in memory, we perceive it as if the movement were one by one.

5.7 Expansion of CPCTelera functionality

5.7.1 Drawing 4x8 maps

Version 1.5 of CPCtelera has for the first time a routine that draws tilemaps with 4x8 byte tiles (the previous one was limited to 2x4). With this function you can paint the mapping much faster as it requires much fewer calculations.

However, it has one limitation and that is that, during the painting of each row, it disables the interruptions. This is a major problem for us, since our game uses a vertical rupture (explained above) that requires constant updating to maintain it, or else, the break will be desynchronized and the graphics will be seen spinning on the screen.

To solve this, we have implemented our own routine to paint 4x8 tiles that, although it is 25% slower than CPCtelera does not interfere with interruptions.

5.7.2 Drawing partial tiles

Our scroll advances horizontally at a speed of 4 in 4 pixels. However, the tiles have a width of 8.

When moving a scroll step (4 pixels), we need to paint a new column of tiles on the side of the screen, but being 8, they stand out from the screen and appear on the opposite side, which is an unwanted effect.

The solution has been to create two new routines, which, from a tile of 8x8 pixels, one paints the left half and another in the right half. In this way, in each step of the scroll, we alternate calls to these two functions to form the tilemap in order which will appear of 4 in 4 pixels.

5.7.3 Loading from disk in assembler

CPCtelera has a function to load tape data from assembler, but this is not suitable for disk.

Due to the memory layout explained above, we need to load the menu block by code and this is only possible once the program has started as it is necessary to disable the firmware beforehand. Therefore, we have also implemented a function to load from disk and thus, our game is compatible with the whole range of CPC

5.7.4 Drawing text with sprites

The text drawing function of CPCtelera uses the original typography of the ROM. However, in 0 mode these letters are too big for our game and we also wanted a different typography with the possibility of having several simultaneous colors.

For this reason we have also made a routine that, from a text string and a tileset with ASCII characters, writes on screen the corresponding text. The following image corresponds to the text tileset used in the game:

!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`ABCDEFGHIJKLMNOPQRSTUVWXYZ{|}~

5.7.5 Playing back sound effects with Arkos Tracker 2

Due to memory restrictions, the game includes the main theme of the music in the main menu, but could not enter an in-game music. In addition, only the player occupies 2Kb.

This player can not be separated from the sound effects in the version of Arkos Tracker 1, so even though we were not going to use it, would still occupy 2Kb only for effects.

However, Arkos Tracker 2 does allow to separate these two blocks, being only 500bytes what occupies the sound effects player.

For this reason we have made a function that, following the same syntax of CPCtelera plays sounds using the Arkos Tracker 2 player. And in this way, we have music and effects during the menu, and only effects during the game with 2Kb of memory savings.

However, the game does have short melodies made by editing the instruments, varying the pitch to create the notes. This has allowed us to include the melody of game over, and others that appear in certain situations.

6 Tools used

- CPCtelera (Game engine)
- Gimp (Graphics desing)
- Arkos Traker 1 (music)
- Arkos Traker 2 (sound effects)
- Ubuntu 18.10
- Tiled (Map design)
- Trello (work organization)
- Github (version control)

7 Conclusion

Once the work is finished and reflecting on the result, the satisfaction is very great. It is worth clarifying that when we talk about results we do not refer exclusively to what can be seen.

For us the challenge was a great challenge, it was an ambitious project with a very tight time frame. We had to take hours from anywhere to get "finished" on time. Between quotation marks the word finish since we had to leave many ideas in the drawer as the project progressed, but we had to be realistic. We had a very close date when we needed a product ready for production and we couldn't afford not to. It's also true that none of the features we couldn't implement were considered indispensable, and they're 100% complete.

Apart from the experience at the level of project development and teamwork, everything learned at the technical level has been most interesting. Working on such limited hardware has led us to optimize our byte code and invent all kinds of "tricks" in order to scratch a few bits or clock cycles in the weight routines.

We don't know what destiny will give us, but it would be ugly to deny that experience has made us think about future projects for CPCRetrodev'19 and I think in short this says it all :)