Making of *Raven Squad*

Authors

Alejandro Culiáñez Llorca (Acllorca777@gmail.com)

Héctor Mateo Pastor Pérez - (Hmateo09@gmail.com)

Antonio Gómez - (afgm@alu.ua.es)

Twitter: https://twitter.com/_ravengames_

Special thanks to:

Enrique Morales Castelló (for making such awesome music)

Technologies used

- CPCtelera 1.5.0: Fast Amstrad CPC game engine for C and Assembler developers.
- Visual Studio Code: It's a code editing tool that allowed us to write the code of our game.
- Github: distributed source code version-control system.
- Arkos Tracker: Musical tool for Amstrad CPC, Atari ST, ZX Spectrum, MSX, Oric, Apple 2, Vectrex and Sharp MZ-700!
- Winape: Amstrad CPC emulator for PCs running any 32 or 64-bit version of Windows

Easter Egg of the Prince of Persia

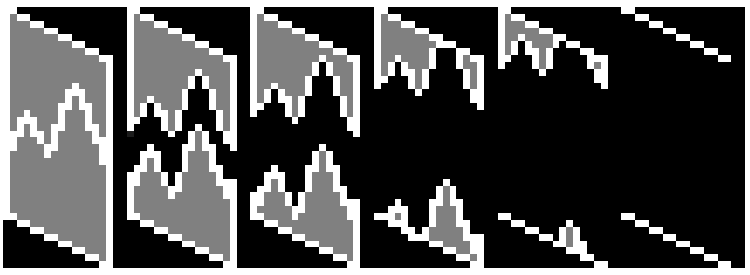
I am writing this section in a personal capacity (Hector).

When the reference to Prince of Persia was mentioned, I was very clear about what I wanted to introduce as an easter egg. I remember being about five years old, and that my father brought in diskettes the MS-DOS version of the game, it was one of my first experiences as a player, I remember writing down in a sheet of paper the commands that seemed to be witchcraft to be able to execute the game. And I remember the frustration caused by the countdown, but there was something else that I felt bad about.

The door knives.

I hated those doors with all my soul, I never got the hang of them, my self of twenty five years ago hated them with all its soul. Even today I still partly hate them when I play the game again in DOSBOX sometimes.

When I exposed the idea to my colleagues, they accepted it, and gave me carte blanche to implement it, and this version of those doors appears for the first and only time in the last level of the game, the nine. It may not be the best version of the door, since drawing is not my thing, but I'm satisfied that I was able to represent that enemy that I hated so much as a child in this game.



Development of the idea

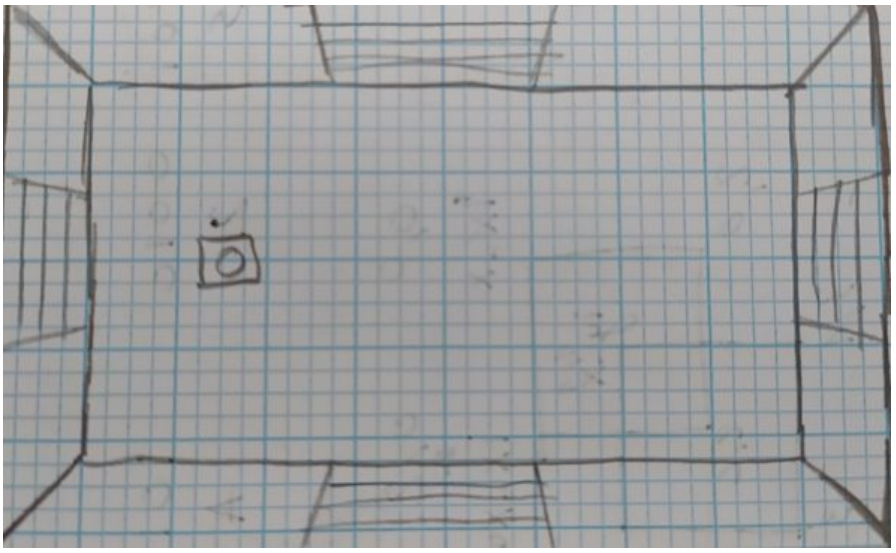
At first, the original idea we had was to make a game based on Smash TV, so during the development of the engine we used for the game, we tried to think about how to develop the idea, the problem was certain design decisions that made it difficult to implement.

1. Original Amstrad Smash TV

We played some games on Smash TV, and we saw serious problems of the game due to the limitations of Amstrad, for example, having to draw the environment (walls and doors), as well as the interface, the playable space was considerably reduced, this, added to the fact that the control we wanted for our game should be "classic", using the button pattern of retro games, was a problem when designing the control system.

In the classic Smash TV game, the shot is made in the direction where the character is "looking", but by having only the four directional keys and the shot key, the problem arises when to "look" in one direction, you have to "advance" first in this. This could be uncomfortable at times in the original game, and given our experience in game development, and even more so in assembly, it could be a challenge to create a character control system that was not frustrating for the user.

Adding this to the problems of setting up the game in 0 mode initially, limiting the available width of the screen, we thought that, although it might be functional, it wouldn't be any fun due to all these factors that would make the game experience worse.



2. Diseño inicial de la pantalla de juego de Smash TV

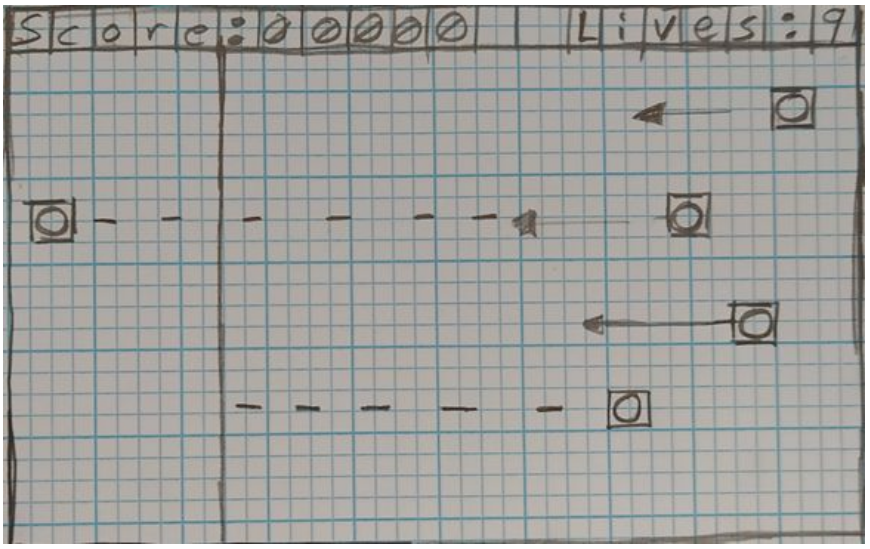
This made us reconsider which game we would use as a reference when creating our own, so we started looking at Commando, another shooter, in which we saw fewer problems, especially when it came to representing on screen everything we wanted to represent.

3. Commando game screen

We set to work to design the concept of what we wanted to see on screen, this time, having more free space, since we were not forced to draw the edges of the screen with those doors and walls, we could already visualize how the game interface would be shown where the points and lives would be displayed.

Since our game design wasn't going to look enough like Commando either at the playable level or at the technical level because of our own limitations, we decided to shift the focus from a vertical advance to a lateral advance, giving more space to dodge enemy bullets, and more time until they reach us.

With all this in mind, we proceeded to realize the first concept of the game screen.



4. Initial concept of our Commando version

The vertical bar you see on the right would be the character's movement boundary, that is, the area he cannot leave, replicating the original Command where when he tries to leave that area of the screen, the map scrolls.

The design process, a step-by-step story

Scroll

The first thing we thought it would be necessary to implement would be the scroll, as it was the most basic part of Commando.

The mechanics of enemy appearance, in the original Commando, was a scroll where the map was already designed and appeared as the character advanced vertically on the screen, without giving the possibility of going back.

In our case, with our knowledge at that time of Z80 assembler and video game development, we thought of adopting a substitute, using a "generator" that would be used as a trigger to make enemies appear as long as the character is "walking" to the right at the edge of the screen. In this way, we could create a "scroll" effect where the map was apparently moving, even though it was not.

Collisions

For the collisions, initially it was proposed to have control over the entities by dedicating memory spaces to what types of entities we were going to create. In this way, only the shots and their collision with the enemies would be checked, or the character colliding with the enemies, optimising the process of checking collisions.

However, after talking to the teacher, we decided that we were "optimizing a priori", instead of making a simple approach first and then refining it if necessary, so finally the collisions are checked in a system of everyone with everyone.

As it is not a Cartesian product (I don't need to know if B collides with A if I have already checked if A collides with B), the process finally adopted was to check for each entity if it collides with the ones following it, starting from the first one to the penultimate one.

This left us some room for improvement that would not be complex to implement later on, such as incorporating by means of some attribute a control system to know what each entity had to collide with, and omitting the steps of collision checking if entity A did not have to collide with B, such as the character's bullets with himself, since he should not kill himself if he collides with his own bullet.

Shooting

Basically, the shots are treated like other entities in the system, with the ability to move according to a vertical and horizontal (physical) speed, with an associated sprite to be drawn on the screen, with an auto-delete system and with the ability to collide with other entities. In fact, the initial collision system was made to detect the collision of the player's shots with the enemies.

Another system that was developed at the beginning was to control the number of shots that both the player and the enemies could make, so that if any AI or the player decided to make a shot, the number of available shots would allow it or not. This was done for performance reasons, to avoid too many simultaneous shots, and also to control the offensive capacity of the player and the enemies, so that depending on the level and other characteristics of the game, the number of shots could change to more or less capacity.

The system was also expanded to indicate the range of the shot, so that a shot didn't have to reach the horizon but could reach a previous point and disappear. This extended the possibilities of defining different types of weapons and their characteristics.

Another improvement that was added later, was to be able to indicate the generation position of the shot, which depended on the enemy or player sprite, so a property was added to the entities capable of shooting to be able to indicate the relative initial position of the shot.

Later a system was added to set the amount of damage an entity type caused (not necessarily just shots) and also the strength of an entity or its ability to withstand damage. This could control the level of damage caused by different types of weapons and the ability of an enemy to take one or more hits before dying, although in general this system would work for any pair of colliding entities.

One aspect that was improved later on in the shootings was the rendering, and this was done by designing a specific rendering system for the bullets. Instead of using sprites, we switched to drawing and direct pixel level erasing. This was a major improvement depending on whether the number of "bullets" on the scene was significant.

A system was also added to show an animation after the impact of a shot on a target, at the point of impact.

Direction and cadence of the player's shot

An important aspect related to the player's shooting was how to control the direction of the shot and its rate, and with which keys that direction and frequency would be controlled.

With respect to the direction, the aim was not to use specific keys to control the direction of the shot other than the keys that controlled the player's direction.

Several alternatives were tested and in the end the following decisions were made:

- The player could not shoot vertically from the side, either up or down, or from behind
- To compensate the enemies could not shoot laterally or from behind the player.
- Depending on whether or not the player was moving vertically in one direction or the other at the time of the shot, the bullets will have a speed component in and, i.e. the bullets will move up or down with a preset constant speed.
- The player's bullets always move to the right, i.e. they always have a positive value for velocity component on the x-axis
- Similarly, enemy fire always has a negative X-velocity component.

With regard to the **shooting cadence**, this should be controlled by the player with the space key, but limited by a maximum number of bullets (as explained in the previous section).

In the end it was decided that the best way to control the shot was to complete the cycle of pressing the space key and releasing it. Initially this was done so that the shot would occur when the key was released, but was eventually changed to occur when the key was pressed, with no further shots being taken until the key was released and pressed again.

Generation of enemies

For the test shots, we had predefined enemies on screen that appeared static, but this wasn't going to be enough for the game, we needed to be able to generate enemies at runtime.

The initial idea of generating enemies could be "set" by default so that the same ones would always appear, establishing a seed and using pseudo-random numbers so that they would always be generated in the same positions sequentially, always depending on how much our character had moved. But this was not the case, so we came to the conclusion that we did not have enough knowledge about how pseudo-randoms worked, and they did not always appear in the same position. However, for the time being, it was useful for us to test the game.

As we already had limited the movement of the character to its playable box, we should only in the same step that we are establishing its position to that limit and assigning a zero to its horizontal speed, call our enemy generator to increase a counter or some similar system and when that counter exceeded certain values, invoke some enemies in a random position prefixed by that seed that we would have chosen.

After weighing up the random factor, and seeing that this could lead to mediocre results, besides the fact that we didn't feel any kind of intentional "challenge", we couldn't control all we wanted the difficulty of the game, and adding enemies of different types ended up being very unbalanced depending on what we touched, it was decided to move to a new system, the enemies would still appear by means of a counter, but this one would be prefixed, along with their coordinates of appearance and their type of enemy.

But this would take a while to appear, first we had to fix certain bugs caused by not having a good physics system capable of telling us when things were going off the screen or destroying them.

Refined physics system with self-cleaning of entities

The physics system was initially dedicated only to updating the X and Y positions of the on-screen entities according to the vertical and horizontal velocity values of the objects, but two additional responsibilities were soon added.

Firstly, the responsibility was added to calculate whether, when updating the position of the entities, they were left off the screen, which had a default behaviour associated with it, which was to mark that entity for destruction. In other words, for simplicity's sake, it was decided that everything that went off screen was automatically deleted from memory. This does not affect the player who has his own control system that prevents him from leaving the limits of the screen when the user controls it.

On the other hand, the responsibility of controlling the screen scroll was added to the system, which basically moves the X position of all entities on the screen horizontally to the left. This responsibility was moved here after trying to directly update the X of all the entities when the player reached the limit edge that triggered the scroll, but it was observed that it caused unwanted and somewhat random effects that were solved once this control was transferred to the physics system.

Rendering system

The initial rendering system was based on erasing the sprites associated with the entities by drawing a box with the background colour in the last sprite position according to the width and height of the sprite, and then drawing the sprites in the new position.

Soon it was seen that this system caused some sprites to blink, so it was thought of designing a configurable system, where for each entity it could be said whether the previous position was deleted with the box system using the background colour, or the edges of the sprite itself were used to auto-delete as it changed position, which also resulted in a performance improvement by saving the drawing of the deletion box.

Auto-erase was not applied to all sprites mainly because of the bullets, which were very small elements, but could have a lot of variability in their direction and speed, which together with the horizontal scroll would have required an edge that was too large to be viable, which would also have affected the impact area of the bullets and the overall performance of the sprites.

Level generation (Enemy System 2.0, Part 1)

At this point, we knew that it was time to look for a way to be able to perform pre-set levels to establish a minimum quality that we could control, as well as to be able to have control over what happens at each level.

We had the enemy generator, which generated an enemy type entity, reading a generic enemy template every time a counter reached zero, so we only had to modify a minimum part of the code, the place where the counter appeared, as well as the enemy template to generate and its Y position.

In this way, the first structure of the level manager started, a source destined to have only one function that consulted the A register, where the current level would be indicated, and returned the pointer to where the level data starts.

The first version of this, was very simple, it used relative positions, a level was a set of trios of bytes, being the first one the relative distance with respect to the previous generation of enemy, the second one the position Y where we wanted the enemy to be generated, and the

last byte would be the type of enemy we wanted to create. This required another additional expansion, the enemy manager.

Enemy Manager

The Enemy Manager was born from a simple concept, as indicated above, using a byte as an indicator of the type of enemy that the level generation system had to generate. At first we used this byte to have a flag bit that indicated the type of enemy, although this would end up being problematic, this will be discussed in the section Enemy Manager 2.0.

As we initially had only two types of enemies, a generic enemy with simple behaviour that moved up and down the screen and an enemy that tried to "lock" the passage of the player seeking to position itself in front of him, as well as running towards him once he was in this position, this system was more than enough.

The implementation was very simple, in the header we had defined the types of enemies with the bit that should be each one, and the manager had a get function that returned in the HL register a pointer to the template of each type of enemy.

To know which enemy to return, it consulted the A register. With this we could start the level generation system.

Level generation (Enemy system 2.0, part 2)

Once we had a way for the generator to consult what kind of enemy to generate, when to do it, and where to do it. However, we lacked something, to indicate when the level would end.

That's when the concept of distance was born. Once the player had travelled a certain distance, the level would end, giving way to the next level.

For this, we started to use the Info_Screen module, which showed us information on the screen, both the level we were going to play and the game over screen. We implemented a very basic version of the distance system, using the ASCII codes of the numbers from zero to nine, putting in the register IX the position of the lowest number, that is, the units. We then checked if it was zero, as we were decreasing that number, and if the condition was met, we assigned that number to nine, and called the function again by reducing the position stored in register IX, so that we could aim for the tens. This was used for each digit of the distance.

Seeing that it was more or less functional, we proceeded to create a basic system of points, performing the inverse operation, if the distance system decreased the ASCII codes, the points system was dedicated to increase this ASCII.

This system, although rustic, was functional, and the score has been maintained in this way until the final version.

These two systems eventually moved to a separate module, called GUI, and an additional system would be created which would then be retouched, the lives system, using the same logic as the points but limited to no more than nine lives.

With this done, we lacked a way to communicate the distance of the level to the interface, and to the game itself, to know when the level was finished, this was done through readings to the level data, we decided that the first three bytes of each level would be the ASCII codes of its distance values, for example, to symbolize that a level had three hundred of distance, the first three bytes of the level would have the values 0x33, 0x30, and 0x30.

Finally, there was only one way to communicate that the level was over. We opted for the input system itself to take care of this, a questionable solution, if we think about the design as such, but from the point of view of efficiency and with our little experience in the field, it was a small way of optimizing the game, since the input controls through the remaining level distance whether we have to stop the scrolling effect and free the character from his area in order to move freely or not.

In this way, just as the character was released from his zone, we proceeded to see if the X of our character ever passed the value sixty (approximately three quarters of the screen). With this we knew that the level was over.

Enemy behaviour

Having already two very basic and simple enemies, we could start creating enemies and test them easily, since we only had to create one level and put them with relative distance 0 to make them look directly on the screen.

Also, thanks to our enemy manager, we could specify the new ones in a very quick and simple way, and the generator could interpret them and generate them when they should, combining the level and enemy system.

The logical step then, as mentioned, was to start designing new enemies. The first thing we thought about was the prototype of a "tank", a bigger enemy, which would remain static and besiege the player, so that we would have some kind of enemy that we could position on the screen without having to worry about its movement, just shooting.

We then designed a more hostile enemy, a jeep loaded with a machine gun that would shoot and appear on the screen moving in a straight line to the left side of the screen, where the player is, but this posed certain problems, the jeep was too wide, and our generator generated the enemies in the position marked as the maximum width of the screen minus the width of the sprite, so the jeep would "pope" almost three quarters of the screen directly.

To fix this, we tried to implement a multi sprite system, where the render itself could, interpreting the components of each entity, detect that it was a multi sprite and draw it in "pieces", as it appeared on the screen.

MultiSprite Render System

We already had entities with useful attributes such as components, and with this we could start specifying to our entities what types of behaviour they had to have, so we thought of implementing an alternative render, which would take care exclusively of the entities that were too wide to be rendered normally on the screen, and allow them to appear little by little on the right side.

The idea and implementation was simple, the operation would be identical to that of the render, but the very wide entities would not have a single sprite, using the same system as that of the animations, we would cut the sprite into pieces that would be drawn as possible due to their position and the width of the screen.

After a whole day of trial and error, we arrived at a functional version of this, if our jeep was 20 pixels wide in 0 mode, we would cut it out in five-pixel pieces, then by looping its X and increasing it with respect to its width, we could know when to draw. It was similar to the following pseudo-code:

```
Loop:

        If X_Current < maximum_size_screen - piece_width, skip to piece_drawing

        jump to end_loop

        draw_piece:

        Draw current piece

        X_Current = X_Current + piece_width

        If there are pieces left to draw, jump to Loop

end_loop
```

Although functional, it was very basic, and to delete the entities many things had to be taken into consideration, so in spite of everything and after losing a whole day in this system, it was decided to discard it and adopt the design criteria of enemies that these were narrow by default, to avoid the effect of "popping" on the screen.

Despite the fact that this system was discarded, it helped to gain fluency with the Z80 programming, as well as to get to know the machine better due to the difficulties involved in adapting this, so that we do not see it as a waste of time, but as another experience that has contributed to our learning.

Enemy behaviour, part two

After adopting our new design patterns when thinking about enemies for our game, we thought again about bullets, and how they work.

Until this point, the bullets were random, we had a maximum number of possible bullets on screen for our enemies, and each enemy used a pseudo-random number to decide when to fire.

This could be strange, we had enemies that always appeared in the same place, with their own behaviour, but even though the experience of playing a level was always the same in the generation of enemies, the shots were totally random, turning two games in which the player did the same into totally different experiences, adding once again that unpredictability factor that we didn't want them to have.

With this in mind, we thought about how to solve this issue using things that we already had in the game, to avoid having to waste a lot of time, and we found a solution, using the AI counter that the bullets used and that by default had all our entities as a countdown to shoot, this way, in addition, we could have different rates of shooting according to the enemy.

Once we had changed our enemy's IAs so that they would fire in this way, we could implement some idea like that of an enemy in a fixed machine gun that would fire at a much higher rate than that of ordinary soldiers, but that's where the problems came from, as performance was impoverished to unsuspected levels.

Performance problems, optimizing the game

This was another of the headaches we had, and it took about two days of thinking about a solution.

The problem was simple, we had our logic and rendering in the game cycle and at the end we were expecting two VSYNC, at this moment we started to measure the cycles using Winape to see where the biggest load of the game was at that moment.

The answer was as expected, collisions and rendering.

The rendering became a more specific system at this time, a specific way of rendering bullets was created to avoid additional calculations, checking before rendering an entity what type it was, and in case it was a bullet, simply copying a black pixel to the video memory address where it had to be represented according to its position.

This, although it improved approximately 5% the rendering time, it was still insufficient, so we moved on to improve the collisions.

The collision system made many checks, more than necessary, for example by checking things that did not have to collide with each other. We then remembered the system that we had initially thought of, and which we left for later, adding an attribute for each entity that told it what it should collide with.

This system is the one we have left today, to check what each entity should collide with, we consult a byte that is used as a flag, since the types of entities that we have are eight, and we can use each bit to make comparisons about whether entity B is one of the types indicated in the collides_with attribute of entity A. In addition, when using the bits as flags, the collisions can have multiple things to collide with, for example the player must check that it does not collide with the enemy bullets, and at the same time with the enemies or obstacles.

With this implemented, collisions dropped approximately 30% in cycle consumption, and the game was back to acceptable performance, or so it seemed.

Since we didn't want to waste a lot of time creating a system of collisions with walls, in order not to add complexity, we decided to create static enemies that wouldn't behave, in this case, we talk about the enemy barbed wire, an almost totally vertical entity that serves to limit the player's movements without having to resort to implementing "braking" collisions.

The horror came next, the performance of the game was again terrible, as the vertical sprites consumed a lot, due to the calculation of their vertical position and how the video memory of the Amstrad is structured.

At that point we were left blank, we didn't know how to tackle the problem, which led us to investigate the operation of the machine even more.

The first idea was to delay the execution of certain functions in case the performance was optimal, thus worsening the time between frames, and to eliminate one of the VSYNCs, but after hours of investigation, we did not find any way to do this, at least none that did not consist of welding some clock to the machine's board.

We tried then to optimize everything, but no matter how much it will be optimized, the render still took a lot of time, so when we waited for the two VSYNCs, sometimes one had already passed, having then in some frames three VSYNCs, the one that had passed by the time the render took to finish, and the two that we always had.

We didn't know how to deal with this, we had to find some way that a VSYNC wouldn't be done in case the render had taken too long, we were running out of options. Because we didn't know what we were doing.

After giving it a couple more laps, we fell into the little that we understood what we were doing, why were we waiting for two VSYNCS at the end of each cycle? Could we wait for another time that was not at the end?

And so it was, like moving one of the two VSYNCS at the end of the logic part, just before the render, and leaving the other one after the render, all the performance problems went away.

It was quite frustrating to see that the simplest solution was the optimal one, and to see how easy it would have been from the beginning if we really understood at that moment what was happening, but on the other hand, it was a very didactic experience, which made us learn quite a lot from the machine, especially at the electronic level when we saw the diagrams where it was indicated how to solder a clock.

And that's how all the development started going well, or so we thought, until our greatest enemy to date, the phantom bullet, arrived.

The Phantom Bullet, also known as The Bug

Normally, we assumed that we were being killed by bullets that we hadn't seen coming, but as we increased the rate of fire from some enemies like the fixed machine guns, we realized something, there were bullets that weren't coming from the machine gun, they were appearing out of nowhere.

This led us to raise many hypotheses, the most possible was that the render system was not destroying the bullets in time, making them reappear in the middle of the screen when coming out from the left side, or that some enemy coming out from the left side, shot before dying, causing the bullet to be in this position when appearing out of coordinates.

We tried many things, preventing the enemies from shooting once they passed a certain position on the screen, which was only partially successful, because although less, it kept happening, which confirmed a great fear we had, the bug was the result of something else, but it was caused by many factors.

We also noticed the totally unpredictable behaviour, sometimes it was practically impossible to play without several bullets appearing out of nowhere in the same mission, in others, it never happened and you could play several levels without it happening.

Desperate to see where this bug was coming from, and having lost a whole day trying to figure out what was going on, we discovered that Winape allowed us to record the sessions, so we waited a long time for it to happen, starting the game over and over again and pressing record.

It may seem simple, but at this point it seems that the bug realised that we wanted to correct it and it became more elusive, we spent about an hour starting the emulator, playing, recording, and it didn't happen.

By the time we got it right, it was on the first level, practically at the end, we had to wait about forty seconds to find the behaviour we wanted to inspect.

At this point, we discovered yet another feature of Winape, we could make recordings of the recordings themselves, while they were running, so we started the playback, and when there were a couple of seconds left for it to happen, we pressed the record button again. With this we had a three second recording where the bug appeared.

After inspecting this carefully, we finally discovered the cause, the destruction of entities was not working as we had hoped.

Once again, we faced the problem from the wrong perspective, the problem was never the destruction of the bullets, but the enemies, and after losing several hours in this, we realised. The behaviour was very complex, and it was the following:

1. An enemy was dying on the screen.
2. That enemy was our **last entity** in the array of entities.
3. By being killed from collisions, and checking the collisions before executing the AI, there was a possibility that an enemy **would shoot** being dead.
4. At that point, we had a bullet being created, and a dead enemy at the end of the array.
5. When the bullet entity was created, the **entity manager** would look for where to write this entity, and this position would coincide with the enemy that would fire this same bullet.
6. Since bullets use the position of the person who fires them, let's say his "father" entity to define his position, but at this moment the enemy **was dead**, when the bullet was created he over-wrote his own father.
7. By overwriting his father, the bullet had the **default position** that we established for him in his template, and this was none other than the very centre of the screen.

All this behaviour was passing through something very simple, our entity manager was not marking well the destroyed entities, until now, it worked with an entity counter, which was insufficient for such complex and unpredictable behaviours as these, so we decided to completely rewrite the code of entity destruction.

We did without the counter completely, which also forced us to change the control inversion, but since all systems depend on this inversion, just by changing that function we had already fixed all possible failures. The system we chose was to put an invalid entity at the end of our array of entities, so that when we resort to the inversion of levels, it ends up finding an entity with this specific type.

With this, the destruction was already well done, and we didn't have any more problems, even so, we also decided to move the collision checks to the end of the game cycle, just before the rendering, as collisions are the biggest cause of death of our entities, there

wouldn't be any possibility that anyone would execute behaviours after being marked to be destroyed.

This was the last big challenge we faced, since with this, the system was solid enough to be able to incorporate behaviours, enemies and levels, in a very simple way.

All this happened during week five of development, at the end of this week, exactly **eleven days** from the delivery of the game to the competition, which put some pressure on us, we had a level system, enemies, collisions, rendering, the game was going well, but we had no content, we had to quickly design something to be able to continue testing and iterating on the game.

We decided to make one more enemy in order to have variety, and not to design more until we had a few levels to show the game even as a demo.

Testing the final engine, first advances in content

As mentioned earlier, we had eleven days to add content to the game, and make it fun, so we decided to create a new enemy, something more challenging and "intelligent" than we had before, an enemy that could get into position to shoot the player.

The idea was that when the player was "on top" of this enemy, the enemy would get in this position to cut him off and shoot him, following the same logic when the player was below. The first versions of this enemy had a lower speed than the player, but it was very easy to ignore it.

After changing the speed to be the same as the player's, it now seemed that the enemy's Y position was fixed to the player's, which gave a strange feeling.

Also, bullets inherited the Y speed from the entity firing them, a rudimentary way for enemies to "take aim". It was at this point that it was decided to dispense with this inheritance of speed, and that enemy bullets would always be in a straight line.

With these changes, it could feel a little unfair, it was difficult to get close, and the feeling that the Y was fixed on the player made it feel very strange, so we were about to erase it, but we discovered that by having this effect and that the player's bullets inherited their speed Y, it was very easy to eliminate as long as the player avoided it the first time by moving upwards, avoiding his shot, and then moving downwards by shooting.

For this reason, we decided to keep this enemy, because once the mechanics were understood, it was satisfying to be able to eliminate him easily, even though the first encounters with him were challenging.

All this was done in parallel while we were creating levels, where we had four levels that met a minimum quality standard, and finally we could see how our idea became a playable demo.

To finish this demo, we noticed that the player's shots felt strange, there was no feedback to the player about whether his shots were right or wrong, so enemies who were hit near the right side of the screen could give the impression that they were not being hit. This, considering that there were enemies that were alive, i.e. that were able to withstand more than one hit, was odd.

That's when we decided to implement, using our animation system, an "explosion" animation for the bullet, which would transmit that the shots were right, this was done by creating temporary entities.

These entities had AI that made them self-destruct when their animation was over, and they had no collision component, allowing them to be harmless. The improvement of feedback for the player and the visual experience was notorious.

We also decided to put on some music, and asked a fellow robotics student with musical knowledge if he could do something simple for us to try to make a music system and later add some definitive music to the game.

One of the disadvantages of this, was that when passing us the music in midi format, we could import it to arkos 2, but CPCtelera only worked with the binaries generated from arkos 1, which was included.

It didn't take much thought, after looking for documentation on the difference in formats and seeing that it was non-existent, and based on our previous experience of keeping everything as simple as possible, we moved on to another approach, importing the music into Arkos 2, and handing over the score from one program to another, along with the instruments, speed, and all the attributes.

Perhaps it would have been faster to use the raw data generated by Arkos 2 directly, but since we did not know how that would work with the CPCtelera functions, and seeing that the information generated by the songs in binary format in Arkos 1 when translated by CPCtelera did not have half the information of the raw data in Arkos 2, for fear of wasting more time trying to make this data functional, we decided to do it this way.

This finally worked out, and we were able to test the music system to check that we had done the process well.

First review of the game

After the effort involved in providing content to the game in those four days before the review, from what we saw as the first time the game really reflected what we wanted to do, we received feedback from the teacher, with whom we agreed on everything.

The first thing he told us, which we hadn't thought of, since it was natural for us to eliminate all the enemies on screen, was that you could go underneath practically all the enemies

while staying stuck at the bottom of the screen, which was true, it was a funny feeling to see someone play the game for the first time and see the first "exploits" appear.

The second thing was the interface, we had the lives, the score, and the distance in text, which was a bit ugly, which we had to agree with, we had paid so much attention to the bugs, and then to the content, that we had forgotten about this section completely since its first implementation, which was before we even had the first version of the functional level generation finished.

The third and last one was to add a button to mute the music, as it was very short and repetitive, and could end up tiring the player.

Once the meeting was over, we got down to work, and decided to use a cut-out of the player's sprite head as a life counter, which would be located in the upper left part of the screen, and the score would dispense with the text informing that they were score, as it was easily intuitive.

For the distance, we decided to implement a bar with a flag, symbolizing the goal, or end of level, and an arrow indicating where you were.

To facilitate this task, we decided to change the data structure that our levels had, we changed the three bytes that indicated the ASCII of the distance by two bytes that would have the information of the distance, and the third byte became the partial distance. This partial distance represented ten percent of the total distance of the level, and with this we could know when to move the arrow where the player's position was marked.

After a few proofs of concept, the distance was already represented correctly, it increased by eight pixels each time the player covered the amount equivalent to ten percent of the total distance of the level, and stopped when he reached the goal.

This meant that we were able to dispense with many strings, and check both lives and distance with the ASCII codes, which significantly improved not only the visual quality of the game interface, but the code itself. With this done, it was rare to see the new interface elements floating around, so we decided to put a couple of blue squares at the limits of movement of the game's characters.

These blue squares, are rendered at the moment we initialize our interface at the beginning of each level, so they are drawn only once, like the whole interface in general, the only thing that is redrawn is the arrow when it changes position, as well as the score when it changes, this way, we save time in rendering in constantly redrawing these elements.

Enemy behaviour, part three

With this we had solved one of the three points, we were missing two. Since adding a button to mute the music was the lowest priority, we started designing enemies.

At that time we had the following:

- Normal soldier, vertically patrolling the screen and shooting.
- Dog, which first tracked the player's Y position, and once aligned with it, ran towards it in a straight line.
- Fixed machine gun, which was a static enemy that fired more quickly than the rest.
- Tank, which currently had no noticeable difference from the machine gun, apart from being a tank.
- Commander, who sought to place himself in the same Y position as the player and shoot him.
- Barbed wire, designed to be an obstacle that the player could not cross, killing him if he tried.

This was insufficient to create levels that were not repetitive in the short term, and we also had the problem that the player could skip certain sections of the levels, not to say whole sections, and go through the bottom. We thought about adding enemies that could nullify these game patterns, the first thing we thought about was something simple, replicating the normal enemies, but being able to assign them reduced spaces to patrol through, that is, just as the ones we had now went from the top of the screen to the bottom and vice versa, having enemies that never moved from the bottom of the screen, only reaching the middle and returning downwards.

With this we got two new types of enemies, and we could already by designing levels prevent the player from being able to avoid the combat, it was a first idea that later was expanded by using the barbed wire to draw "circuits" that limited where the player could move.

With this a new problem arose, we could no longer use the generation of enemies as before, since we had reached eight, and we could no longer use a byte to have a flag indicating the type of enemy we wanted, however, we reached a quick solution. Now the byte would be a number, instead of using it to store a bit flag, and the generator would use CP to check if the enemy we asked for was the right one by checking if the result of this instruction was zero. As the systems were well separated from each other, we were able to make this change quickly, and no outside system was affected, leaving the level generation intact.

We then decided to implement another type of immobile enemy, the trench, to protect strategic points such as the end of the final levels, but with this we already had three immobile enemies that did not differ from each other, the tank, the trench and the fixed machine gun.

This led us to reconsider the behaviour of these enemies, and to think about whether we could do something other than just shooting normal bullets, and so our explosive missile system was born.

First we decided to test it with the tank, the idea was simple, as the bullets had an AI that had stopped being used several iterations ago because it didn't feel good for them to disappear into thin air, but we hadn't erased that code just in case, we could reuse it, adding at the end of its "life", that is, when the AI counter reached zero and the bullet was destroyed, the creation of a square entity that would cause damage.

This led us to redesign certain types of entities, in order to have better control over what each one did, because if the bullet was an enemy, we could shoot it, and the bullets would collide with it. After these iterations, it was no longer necessary to use an "explosion" type to manage the animation of the bullets, and at this point our byte that we used as a bit flag already had eight entity types, so we dispensed with this one to create a new entity type, enemy_background, objects from the environment that would harm the player but would not collide with the bullets.

This also took us to change the barbed wire so that it was of this type, since sometimes the bullets could collide with it being of enemy type, causing a bad experience, reason why we solved two problems in one.

Now we could create an exploding entity, which would be in the same behaviour as the bullets exploding, but the player could collide with it and suffer damage, killing him. With this we could already start to differentiate our enemies better by the projectiles they were shooting.

To finish this, we made the AI of the explosive shot so that it would wait in the B register the distance that the bullet had to travel before detonating, with this, we designed an enemy mortar, which would have a lot of range and explosive bullets, the tank would have a medium distance with the same type of projectile, and the trenches would go to do this too but at a lower range to give the impression that they were throwing grenades.

By putting different firing times for each type of "explosive" enemy, we got a better differentiation, and more tools to design more varied levels, but we thought it was strange that the mortar would only fire in a straight line, so we decided to give another twist to the idea, and that's how, remembering the times when bullets inherited their speed AND from the person who was firing them, we decided that this group of enemies would "aim" at the player.

This was done in a very simple way, using CP to check the distance of the player on the Y axis from the enemy, and so we could get five shooting angles: straight, a little bit down, up, very down and very up. This is currently done by combining this system to "detect" the player's position, along with modifying the speed AND the bullets at the moment they are created according to what suits us to achieve this effect.

To complete our template of enemies, we decided to implement one last normal enemy to populate our levels, a soldier mounted on a gatling machine gun, which was very simple to implement. This enemy would use an AI like the one in the turret but with a slower firing speed, although it would fire more bullets. Thanks to the tests of modifying the speed Y of the bullets to create the aiming system, we thought about this enemy, which would shoot in three different directions. This was achieved simply by calling our function to fire three times, as this function stored in register IX the pointer where the newly created bullet entity had

saved its data, of the three shots we modified in two of them the speed Y, to give that multi-shot effect.

We then turned our attention to our enemy dog, we had already discussed it, thinking that perhaps the idea of shooting an animal could be violent for the player. The concept was born from looking at games from the past like Wolfenstein 3D, where they appeared as basic melee enemies, and it was true that they did seem a bit out of place. We finally decided to visually redesign this character, changing it from a dog to a soldier armed with a knife, although internally it was still called a dog, to recall this anecdote.

While all this design of new enemies was happening, more levels were also created or the previous ones were completely redesigned. It was at this point that we realised that we needed a rebalancing of the player's own power, because, while the enemies were gaining power as the game progressed, presenting a new enemy at each level, the player always maintained the same one. We had to make sure that as the enemies gained power, so did we.

It was at this point that we decided that the player's bullets would go from four by default, to two, but in return, the player would gain one bullet for each level where he rescued an ally, functioning as a reward system that rewarded for completing levels. Reusing the logic of the lives shown in the GUI, the same was done but instead of appearing from right to left, the bullets would appear from left to right, and instead of a bullet being erased when shooting, it would be drawn in grey, to symbolize that the bullet was not available.

We liked this change quite a lot, while at the beginning you felt more vulnerable, as the game progressed you became more powerful and could afford to play more aggressively. By having this element in the GUI as well, it looked more complete and less wasted, and it worked very well visually to indicate in a simple way the bullets that you had available.

At this point, we wanted to add variety, creating a game mode that would serve as an interlude to our levels, allowing us to separate them by blocks, so we started thinking about what mode we could implement that wouldn't be strange or shocking, and would be easy to play with what the player had tried so far. But first we thought again about turning off the music to close our suggestion list, and since we were going to look for a way to turn it off, it was a good time to start thinking about adding shooting sounds to our game.

Sound effect system

At first it was thought to create a generic system to produce different sound effects as a kind of library and thus be able to configure what sound to use with a number of parameters in the templates of the entities.

We reviewed CPCtelera's documentation on audio and Arkos Tracker's documentation, starting this way to make tests to use the music files we already had so that sound effects could be generated from them.

After several tests we discovered that for the sound effects to work the background music should be playing, since it is that same function that makes the mix of the sound effects that have been defined sound, and that one note was not enough to create a perceptible sound effect, at least not with the instrument that we had defined in our music file.

We had to call up the function of generating the effects at different times with different notes in order to create at the end an effect that was not a flat note.

Another added difficulty was to design a sound effect starting from a single instrument and playing only with the notes, volumes and beats basically.

In the end, after several tests, an effect was achieved that could pass for a shot that was the main objective of adding the sound effects.

At this point, another problem arose, which appeared when a new shot was produced when the sound effect of the previous shot had not yet finished. After several tests it was found that it was enough to restart the sound effect without being strange that the sound effect of the previous shot was cut off.

However, once the effect was achieved, we found that since the music had not been designed to leave one of the three channels available for use by the sound effects, the music was affected and the result was of poor quality.

As a palliative measure it was decided to be able to activate or not the sound effects by means of a key, leaving them deactivated by default.

With this we returned to the initial point that led us to raise this point, that of being able to deactivate the music, now that we had the shots, and as a colleague from Robotic Engineering was passing us new songs, we were able to finish this section and start thinking about how to deactivate it at will.

Finally the way was very simple, we used two bytes in our game class, one to store a one or a zero in case the music was active or not, that would be checked in each iteration of the game loop, and in case it was activated, it would execute the function cpct_akp_musicPlay_asm, and another one for the sound effects that would be checked inside the function sound_effects_play, that would not do anything in case we had our sound byte to zero.

Creating a new game mode

After that, we could focus on designing a new game mode, to better symbolise that we were making a transition between blocks, we decided that it would be something that would symbolise that we were travelling, we decided then that the player in these levels would stop controlling an armed soldier to control a vehicle.

Since our character was always the first entity we created by default at the beginning of each level, it was easy to make this change. The game itself would know by a somewhat crude

system, storing in a byte if the current level was a car level or not, and if it was, it would change the sprite and the dimensions of the player to be able to adjust to the new gameplay.

We then designed a few exclusive enemies in this way, first the stones, which would be the enemy of the first of our two levels, being a way of telling that at this moment there was danger, but not so much, that the main problem of the player was the environment through which he was going to move. We also added our enemy mounted on a fixed machine gun, so that we could also represent that the enemies had the passage covered, and that not everything consists simply of not having an accident on the mountain.

For the second level we wanted to symbolize that the danger was increasing, since this level would be the interlude between the middle part of the game and the late game, so we added mines that would replace the stones in this second level. To further enhance this hostile feeling, the fixed machine gun gave way to gatling guns, as well as mortars.

On a playable level, the biggest change this mode presented was that we took away from the player the possibility of controlling his horizontal speed, which was always the same. This could be considered a tribute to another game, which will be described later in the corresponding section on easter eggs. We also took away the possibility of shooting, as it didn't make much sense for the player to shoot at rocks or mines. In this way, we also increased the sensation of danger when we saw one of the soldiers shooting at our vehicle between the rocks or mines, knowing that the only option was to avoid it.

Internally, these changes were simple, we only had to modify the input system, which until now expected the player to press the keys to move forward or backward, and if he moved forward and was in the X position twenty or more, it activated the scrolling effect. We established in the update this way that if the game was in car mode, the player's X speed would always be one, adding this to the previously mentioned that if the player passes from a certain X position the scroll is activated, the game mode was finished.

After a few level designs trying out what worked and what didn't in this mode, we were satisfied with the result and we gave up on the system and left it definitively implemented.

We also noticed that now the end of level animations should not be reproduced, since these levels did not symbolize the rescue of any member of the squad, so we added an additional in-game check to indicate that in case the level we just completed was a car mode, it would not be necessary to reproduce it.

Tuning the collision system

After all these iterations, we started to realize thanks to the car mode that collisions were sometimes activated from too far away between the entities, this was caused by the "margin" we had left to solve the problem of eliminating the trace of our drawings on the screen.

We decided to look for a way to fine-tune the system, our idea was to add four more attributes to our entities, so that we could tell the engine where the collision of each entity started, as well as where it ended.

Since our collision system worked by means of bounding boxes, and we were already making a sum with the height and width of the entities, we only had to replace in this part the width and height with our values of end of X and end of Y, with this we managed to tune up quite a lot the collisions coming from the right or from below, although there were still the collisions coming from the left or from above.

To achieve this, we added to the corresponding X or Y value the start values of X and Y, the system became much more refined, and although it was still far from perfect, the feeling had improved quite a lot.

By this time, we were already making the designs of the final level, we came to the conclusion that the end should be an interior space, where we would change the colour palette we used to make the background colour different, adding more differentiation of this level with the rest. We opted for the colour grey (although Z80 insists that it is white), as our end of level structures, that is, where our allies were to be rescued, had been this colour throughout the game.

Final level, starting with the last changes

At this point, we were close to finishing, while the final level was being designed, some small changes were occurring as well, the new music that our partner was composing arrived, allowing us to implement it in the game. Also, to add variety, new allies were drawn to rescue, so that in each level there was a clear differentiation.

Although we had the feeling that something was missing, we already started to prepare the game for its release in the competition, researching the issue of licenses, adding them to the game, distributing our music through the levels so that it wouldn't always sound the same when faced with the concern that the player would deactivate the music because they thought there was only one song, minor design changes to improve the experience.

With all these changes the game finally felt like a finished product, perhaps with its flaws, but it was a product that was really playable, offered variety to the player, and had a moderate duration. For the time we had, we were satisfied, especially when we thought that because of a confusion of ours we started the development a week later than planned.

However, even though we liked the last level, a level full of sharp steel doors that separated the level into "blocks" full of enemies, it didn't seem like a good ending to the game, there was still a feeling that something was missing.

It was at this point that we decided to implement a final boss, it was thought that it would be the last ally to be rescued, a traitor, and to give the background that the last level was a trap, the player would enter the building where in theory would be his last friend and instead he would find this level, and at the end of this, his friend waiting to fight him.

Final boss, designing the last fight

Unfortunately, due to time problems, we had to avoid this idea with some animation or video scene, preventing the player to understand this detail of the plot, however, we believe that it works because it is something so "classic" to find a final boss, which would not be strange.

After some concept ideas, there were several ways to approach this enemy, the first one was that it was a normal enemy with a lot of life and an improved AI, but since it was going to be the final part of the level, it felt weird to have free movement around the environment and fight against an enemy like that. Adding this to the fact that our character could only shoot in one direction, this combat seemed out of place.

It was then planned to introduce a second easter egg to Prince of Persia, and that the final enemy would replicate your movements in reverse, being a reference to the shadow born from the mirror of this game, but then it didn't feel like a challenge, you just had to know what to do and the combat ended very quickly.

Finally we decided to fall a bit into the "cliché" of making an enemy formed by multiple parts, this enemy would be a sum of all the enemies at the same time, omitting the enemies that tracked the player, it would have a targeting system with which it would shoot a kind of missiles that would cause explosions in one arm, the other arm would be similar to a gatling, having multiple bullets, and the main part, the head, would be a stationary enemy that would shoot with a moderate cadence to prevent the player from moving freely from one side of the screen to the other.

As this would be strange if the enemy was a person, we decided that it would be a kind of robot, but then we realized that it was very strange because the way we had designed the enemy, each part of it was independent from the rest, needing only to break the head to finish the game, and the player could eliminate the arms to make the task of finishing with the head much easier. This strange feeling came from the fact that there was no union between the pieces of the boss, so it was decided to add a box in the colour of the boss on the left side of the screen, giving the impression that all the parts were part of one body.

Positioning was important in this match, as the player could position himself on the boss's face and kill him without being hurt, which was not what we wanted. The movement of the arms was refined, giving him a longer route to increase the "hot" areas where shots could be fired, but even so, it was possible to get very close to the boss and damage it by dodging bullets easily.

It was then that we decided to put a sort of mid-screen, which would prevent the player from getting too close, killing him the moment he touched him. This was one of the two possible solutions we thought of, the other being to increase the points from which the boss could shoot, as this eliminated the most "absolute" form of victory, which was to eliminate the two arms and place oneself in the boss's face by shooting all our magazines without having to move.

All this was a way to win, the boss by how it is designed has several strategies to defeat him, it is also possible to eliminate him without defeating any arm, dodging all the bullets he throws at us, as well as destroying only one of the arms and shooting him in the head from the hot spots created by this eliminated arm to finish him off without pressure.

Last changes, ending the game

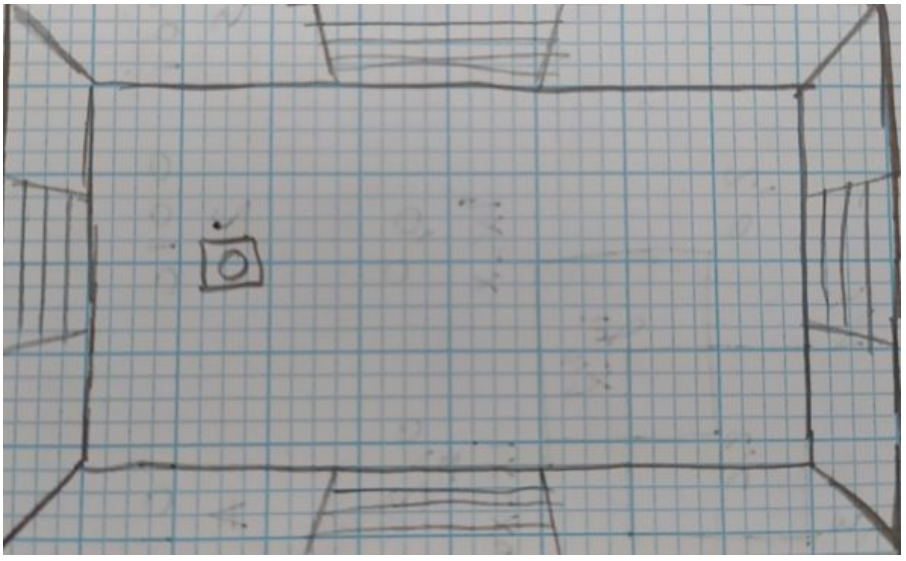
Once we were satisfied with this final boss, we started to make some final touches, we proceeded to balance the life of each enemy, increasing the life of some and reducing the life of others, we changed the distances of the shots of our enemies with explosive ammunition, and a small final scene was made informing us that the forces of evil had been eliminated and that both the world and our friends were safe.

A credits screen was also implemented, we tuned the level-skip key so that if we used it in the last level it would show us the final animation, some sprites were refined, and we added our own music to this last level.

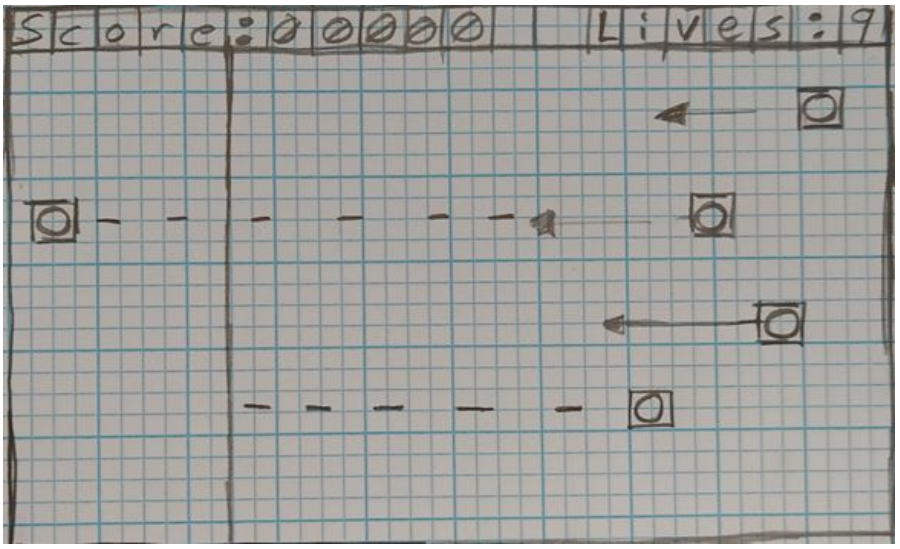
It was also decided to separate the fight of the boss and the final level in two separate levels that would not have transition between them, that is, the text would not appear informing us that it was a new mission, to try to reflect that the boss was an extension of the last level, and not a new level to which we had moved.

With this, we concluded the development.

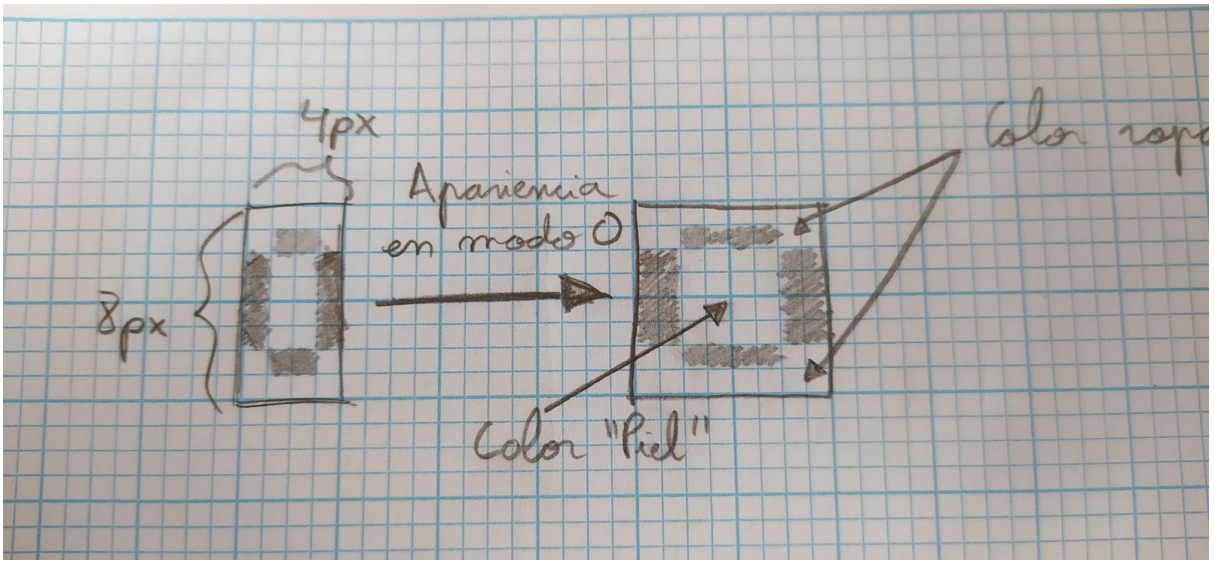
Sketches of the game



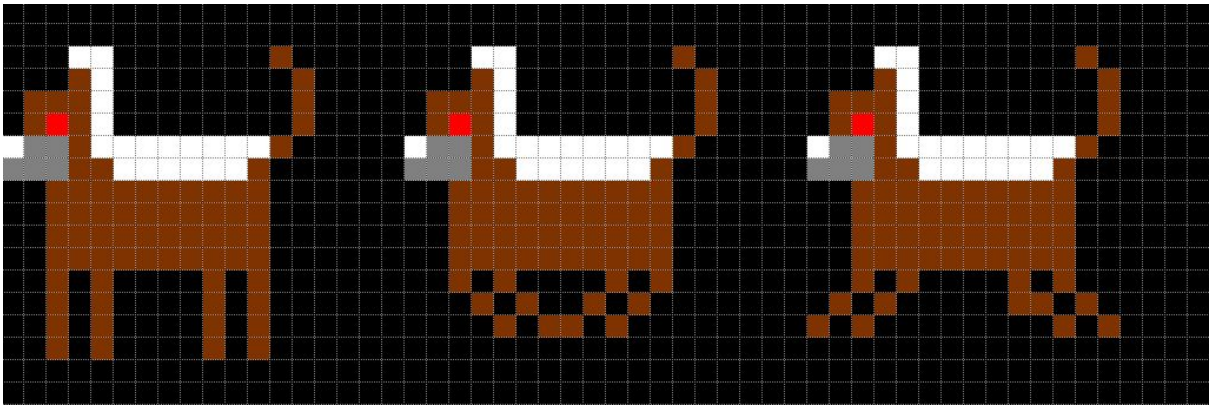
5. Initial concept of game screen based on Smash TV



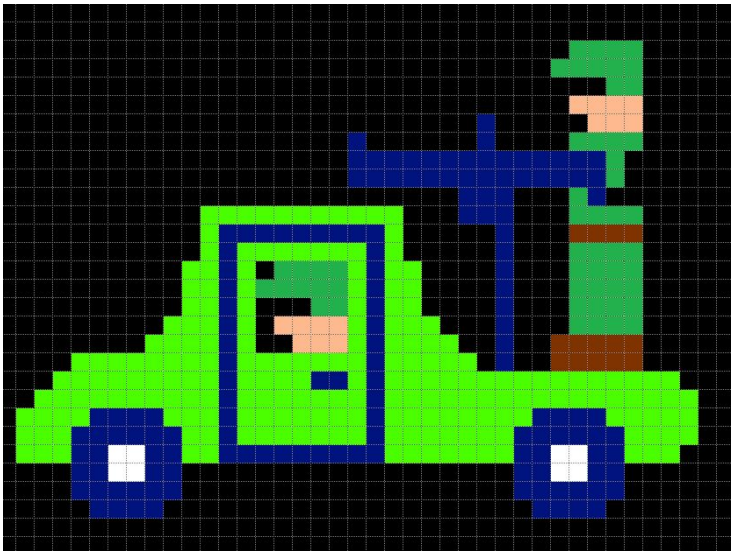
6. Initial concept of game screen based on Horizontal Command



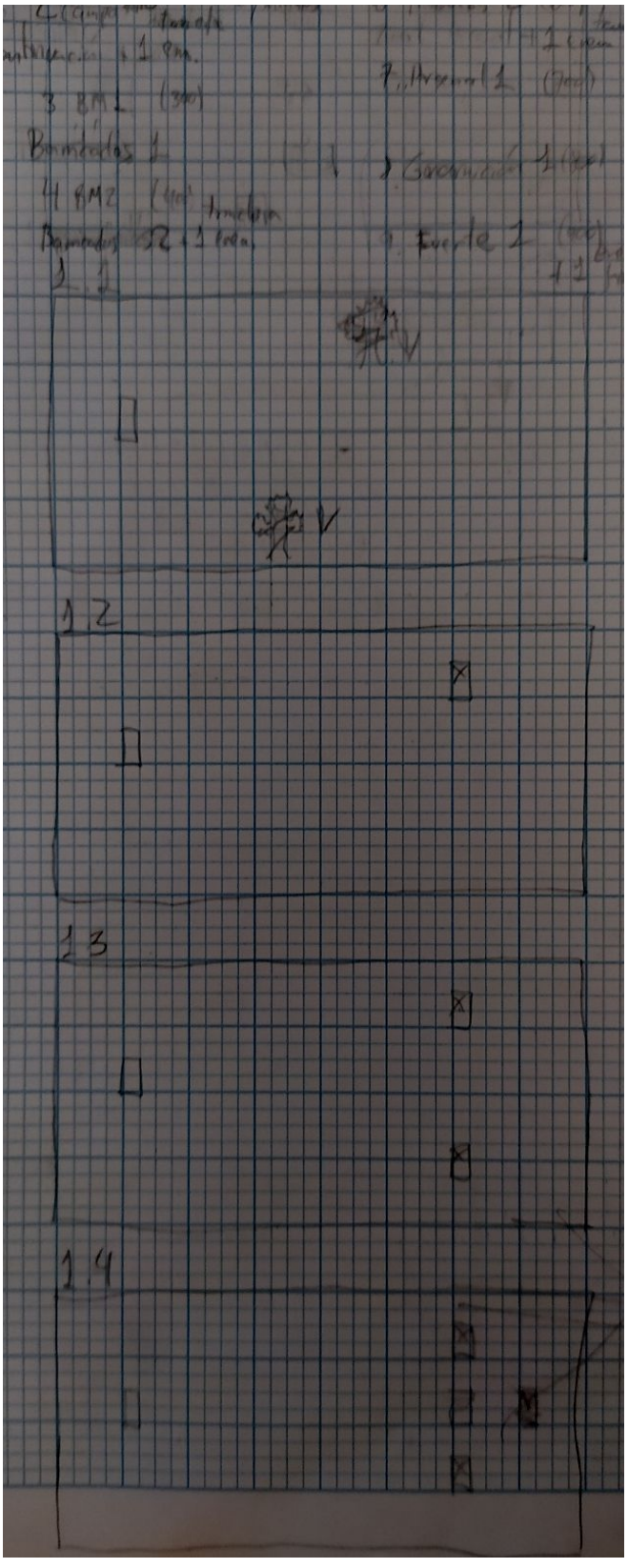
7. Initial design of the player's sprite for testing



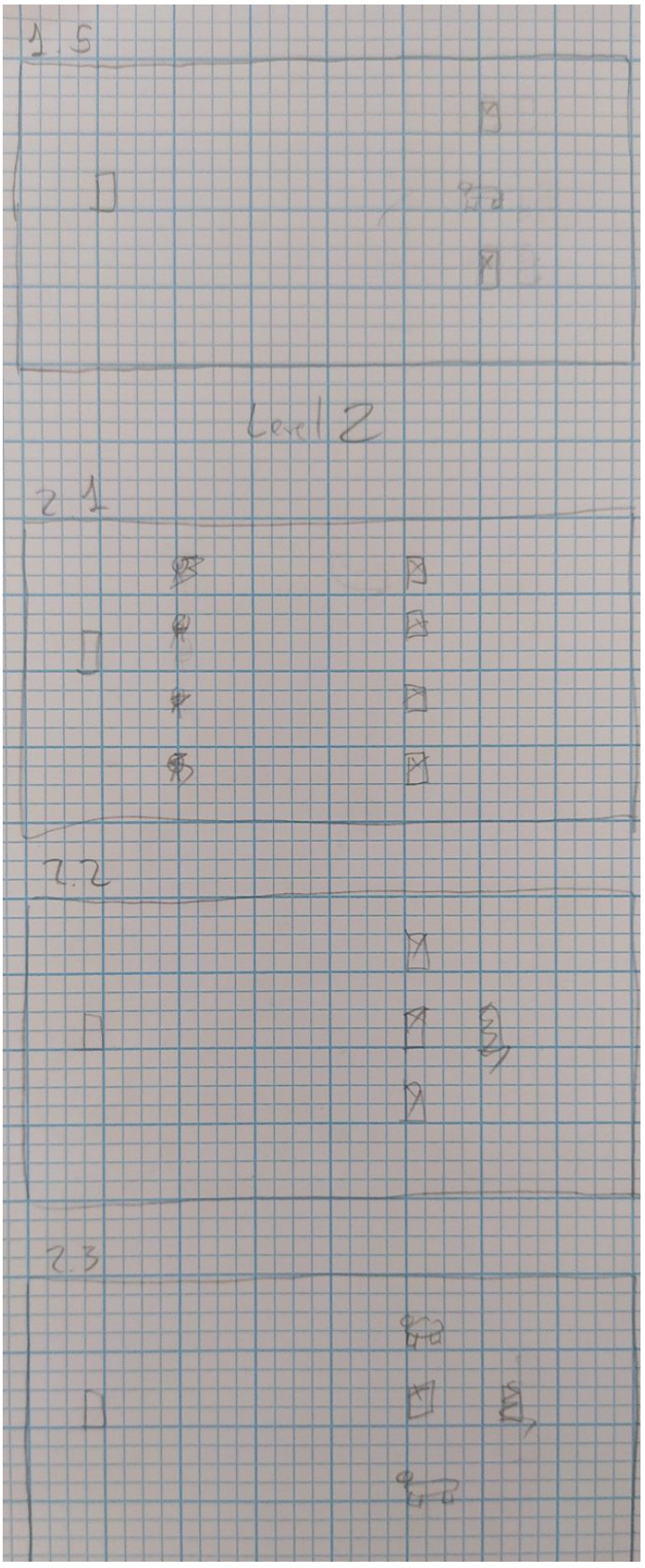
1. Enemy dog, ruled out in the final version



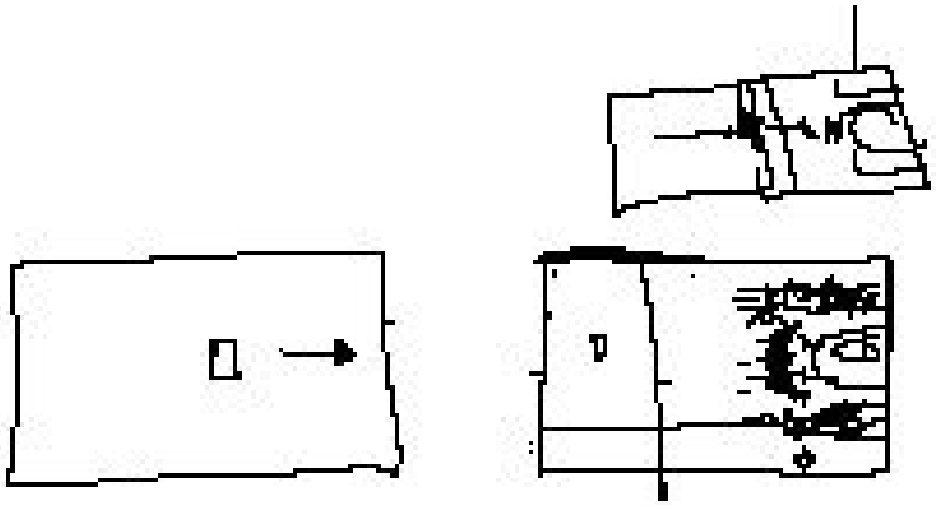
2. Enemy Jeep, discarded in the final version



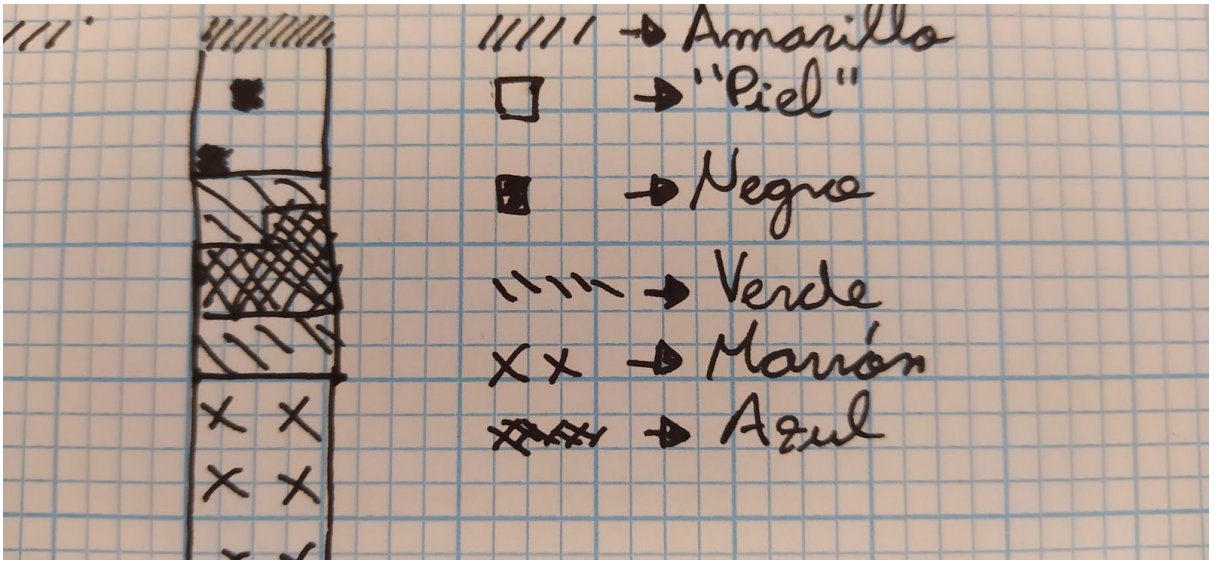
3. Level 1 Sketches



4. Additional level 1 sketches, and part of level 2



5. sketches of the final boss



6. Sketch of the player final model