MAKING OF

Index:

First of all, who are we?

We are **Nibble Games**. We are a game development studio created on September of 2018 at the University of Alicante by a group of students of Multimedia Engineering, namely:

- Carlos Soler Mujeriego (https://github.com/SolerMultimedia)
- Enrique Vidal Cayuela (https://github.com/EnriqueVid)
- Alejandro Gutiérrez Martínez (https://github.com/agm280)

@NibbleGamesDev

And what is "*Crimson Knight Adventures*"?

Crimson Knight is the game that we developed this year for the CPCRetroDev 2018. It's an adventure/action game where your objective is to infiltrate into an evil enemy castle and defeat all the enemies that you encounter in your way.

Crimson Knight Adventure has been fully developed in assembler code, with the help of the game engine "CPCTelera", made by Fran Gallego. Although CPCTelera could work if we programmed the game in C, this was an opportunity for us to learn how assembler code "works" (assembly code for the Z80) so we took advantage that CPCTelera supported assembler code as well.

The game can be perfectly played in a real Amstrad CPC 464 machine (we tested our game in a real Amstrad CPC multiple times during development).

What technologies were used during the development?

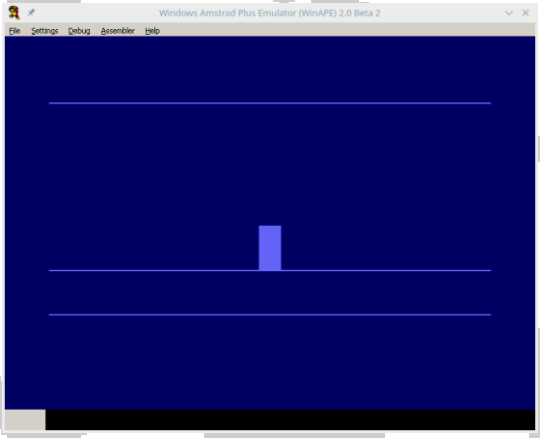
- *CPCtelera* → Game engine for Amstrad used during development.
- *Sublime Text 3* → Text editor used for the code.
- *Arkos Tracker* → Musical software used to produce the music for the game.
- *Gimp* → Image editor used to create sprites, game art…
- *WinAPE* → Amstrad CPC emulator used to test the game.
- *GitHub* → Development platform used to manage the project.
- *Tiled* → Map editor used to create the game maps/levels.

Early stages of development (Weeks 1-2)

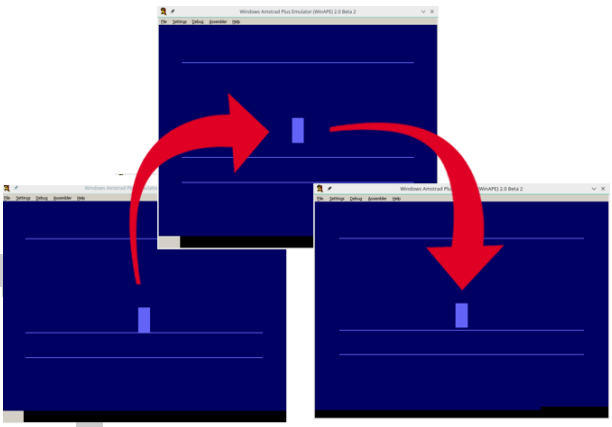
A picture of an Amstrad Keyboard that we would later use to test our game later in development

The first weeks of development were instead about **learning how the Z80 assembler works** and what possibilities we had with it. Our initial project idea was very, very different than what we ended up doing in the end. After doing some research and testing the Z80 possibilities, we finally started developing what ended up being our final product: Crimson Knight.

We wanted to do an action game, where players had to finish their levels quickly, but enemies would try to stop them. The player would have total control of the main character and the scenery changed when he walked through the level or when he completed it. Before we could develop the character and the enemies, we tested the keyboard inputs, so we could know if a player pressed a key.



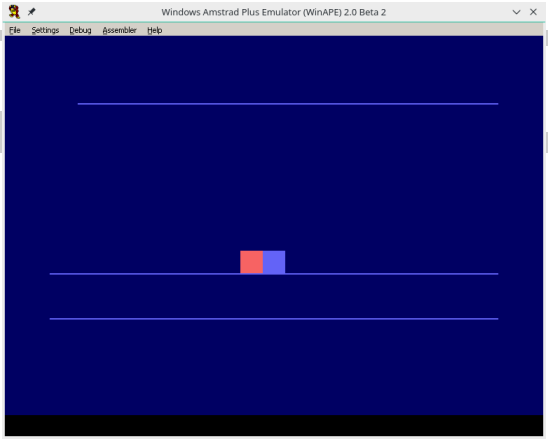
First visual representation of the character during development stages



Representation of how jumping works in the game.

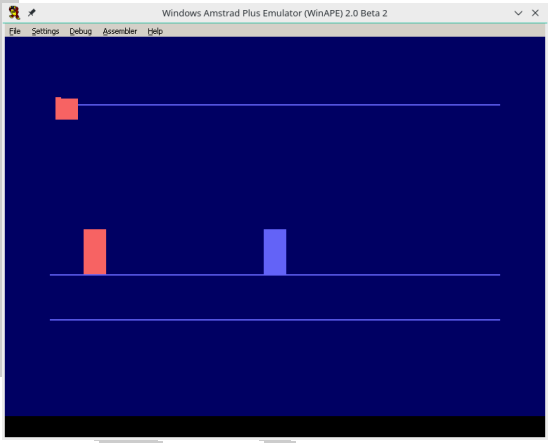
The first thing we had to do: Making sure the character moved and behaved correctly. We programmed the **first version of the character** during the second week of development, and it pretty much stayed the same during all development if we don't count the fact that we did not have sprites on the game yet. With the press of a key, the character could **jump, crouch, and fall** with an apparent gravity (using a "*jump table*", where the character would decrease a different height each iteration) very fluidly.

The red box near the character represented the "hitbox" at early stages of development



In this picture, the character is performing a crouch, that's why his blue box is smaller. You can attack even if you are jumping or crouching, that's why the red "hitbox" was also visible.

But... we also needed him to **perform attacks**! To do so, we created what we called a "*hitbox*". This box appeared in the position where the character was looking at: it had the same width as the character. When the character jumps, the hitbox also has the same height of the player's body. The player can also attack while he is crouched. It would be very unfair if the player could keep attacking forever, so we limited a bit the time the hitbox appeared on the screen and how often can be used.



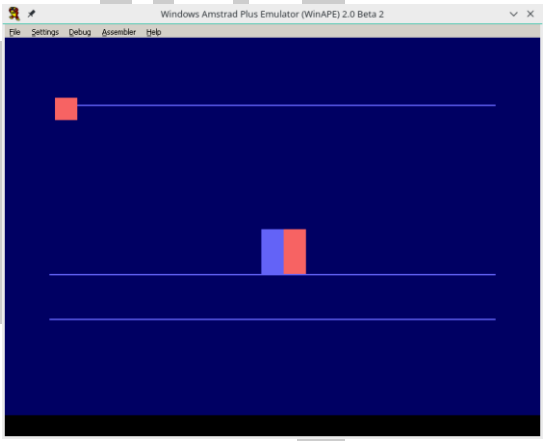
The standard enemies have the same width and height as the player. In the first weeks of development, the enemies simply moved and passed through the player, not performing any damage or doing anything special.

We also started working on the **enemies and projectiles**, but during the first week they were not doing much. We prepared them until the collision system was ready to test them out.

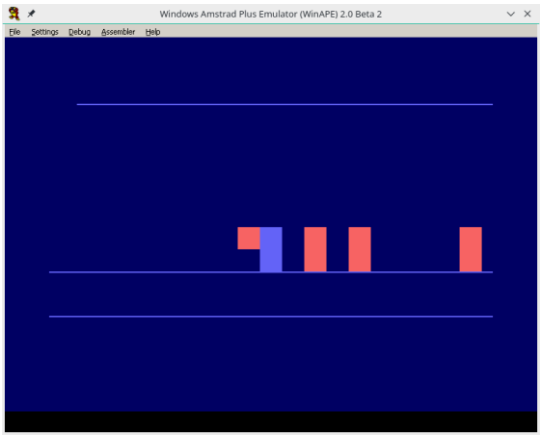
Around this time we also implemented the **"interruption handler"** as well: it allowed us to check the user input of the keyboard (to check if some key or button has been pressed) more frequently without performance issues and would let us play music files without slowing down the game in the future. To do so, we used CPCTelera's *cpct_setInterruptHandler_asm* method, that allows us to take advantage of Z80 interruptions so we could reproduce whatever lines of code we wanted during an interruption (an interruption allows you to program asynchronous events. When the Z80 receives an interruption, it runs the interruption handler code, that you can change with CPCTelera. After a few clock cycles, an interruption is automatically sent to the Z80)

Middle stages of development (Weeks 3-4)

The first thing that we developed in this stage was the **collision system** that we had to implement in the game: the enemies, projectiles and player would run into other entities, so we had to handle every collision that could potentially happen in both X and Y axis. So, we created some methods that could inform us if a collision was made between to objects if we provided it with how large and tall those objects were. With a series of simple calculations taking into account the previous parameters, it was easy to determine if an object was exactly near other one or inside it.



The enemy, represented with the red box, is colliding with the player and won't move past him: the game detects the collision and responds not allowing the enemy to move and making damage to the player.



Multiple enemies being generated from the right side of the screen. This was before the random generation direction was implemented.

The enemies were not quite over yet as we wanted them to be generated from each side of the screen. In order to do that, we used a **random number generator**: but we had a problem, the Z80 can't generate random numbers by itself, it needs "something" that changed the seed of the random number within the game code, so we did the following: the random number seed would increase each time the player DIDN'T press a button from the title screen. Doing it this way, the random number would always be different because each time the player takes a different time to press a button to start the game. With this new random number generator created, we simply used the first byte of the random number to check if the next enemy should be generated from the left, or from the right. If the byte was between 00 to 79, then it's from one side, and if it is from 80 to FE, the other side. The random number method that we used was heavily based of one of the CPCTelera's repository examples about how to create a pseudo-random number generator in Z80.

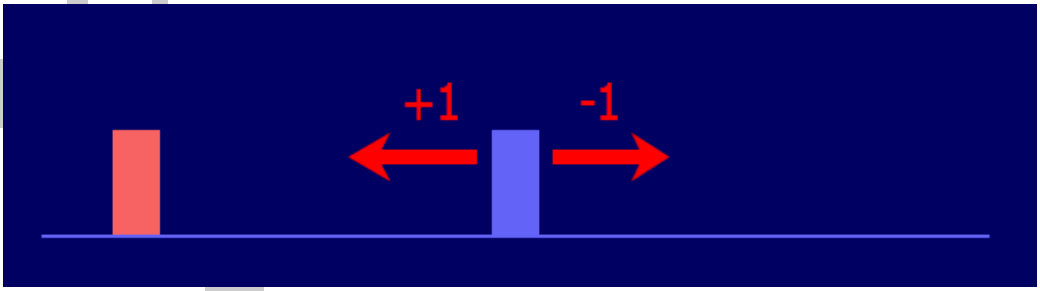
Also, the enemies now didn't attack each iteration, each enemy has a time before it can deal damage to the player again, to make the game more balanced.

The projectiles were also defined around this time as well, they were basically the same as enemies but with some of the properties changed: they didn't move exactly the same and wouldn't stop the player in his way. They would deal more damage though and could come from the top (so the player has to crouch) or from the bottom (so the player has to jump).

Another problem that we had to start working on was how the player and world movement were going to affect the enemies: the enemies movement should be influenced if we were running away from them and the map where we were playing should as well move if we were moving along with it (this is known as “scrolling”).

However we want to make clear that this was the **early concept of our camera system,** that **is completely different to what we ended up programming** in the final weeks of development, but this one helped us understand how sometimes something that you think could work in your mind doesn’t always translate well to the game.

While implementing this first **camera prototype**, the **problem** we had was “how can we tell the program that everything has to change when we move”. Our initial solution was to create a pair of variables (values that can change over time): one that tells the program **if the screen can be moved** and another one that indicates **where is the screen moving and how much**.



The number was set each time the player used the keyboard to “move” his character to the left or the right. If he moved to the right the number was “-1”, it meant: “The player moved to the right, the positive direction, so the rest of the things on the map should move to the left, the negative direction, to compensate the new screen position”. This was the idea for the early camera system, but it was scrapped during development because it didn’t look fluid or realistic and the enemies behaved incorrectly.

Initially, this *“can-you-move?”* variable was always changed to 0, 0 indicating that “we *can* move!”, but this variable will have a different value if, for example, an enemy is attacking us (we then are trapped until we destroy that enemy). If we pressed A or D to move to left or right, then the second variable *“what-direction-are-you-moving?”* would have a value of 1 or -1, depending on the direction where the WORLD had to move (this is explained in the picture above).

In short, we were playing with the illusion that we were moving our hero, but we were moving the entities around him. Later on, when we implemented the map and were testing this method, we noticed how the enemies moved extremely fast, and it was very, very difficult for the player to destroy the enemies with that amount of speed: this is caused because our map moved 8 pixels each time we moved it (due to the map being tile-based, each tile being 8 pixels long, this will be explained later on), and if we moved the enemies with 8-pixels speed every time the player wouldn’t stand a chance against them.

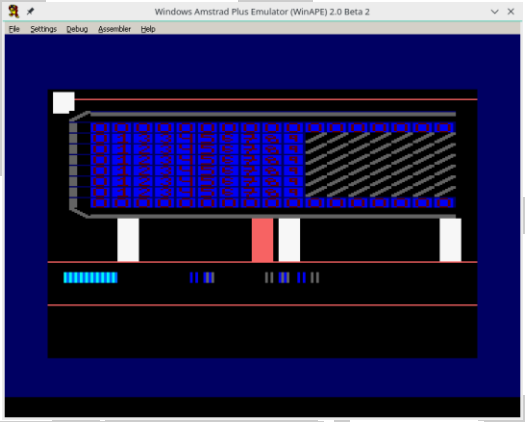
To try to solve this issue we applied this “solution”: the enemies will have their standard speed, and their running-away speed. The first one will apply normally, when the enemy is chasing us. The other one applies when we were running from the opposite direction of the enemy, to reproduce the feeling that the enemies were becoming slower because we were running away from them. Again, this method was **scrapped** at the later stages of development due to it being very inefficient and not-realistic.

Early test of the title, game over and stage clear screen.

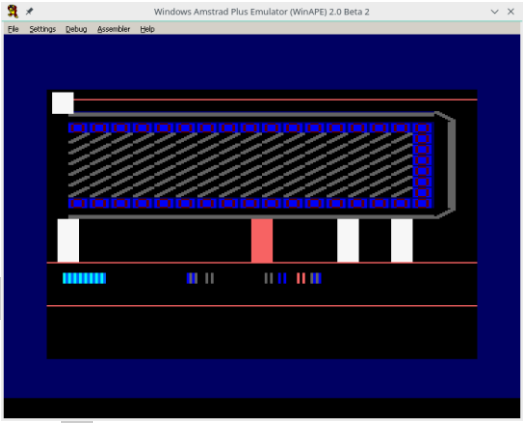
First prototype of the HUD, with every value represented with boxes, including the numbers of the Score and Timer.

In parallel with that, we developed our first "**message screen**" that would later on be used as the base for the title, game over and end level screen. At this point of development, the screen only displayed a box. When a button was pressed, the screen was erased and the game would then proceed to the main game loop (to check if a button was pressed we used *CPCTelera's* cpct_isAnyKeyPressed_asm method).

The **HUD (interface) prototype** was developed around the same time, too. It consisted of just 3 elements: the health bar, the time remaining and the score. Because we didn't have sprites at that moment, everything was being represented with boxes. The time and the score consisted only of 4 decimal numbers that would show the player the numbers on the screen to have feedback about his situation in the game. Because Z80 wouldn't obviously assign a box depending on the number that our variables had on them, we had to assign them by ourselves. The health bar just consisted of a simple box that would decrease his size each time the health of the player decreased.

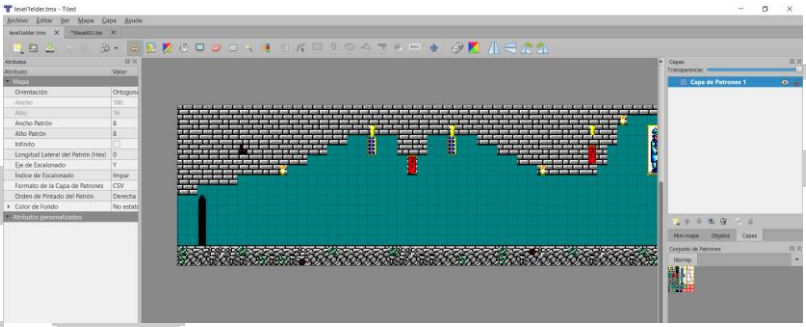


The map was represented above the characters to test how the map looked and responded to the player movement.



The map responds when the character moves to a direction, revealing more of the map content.

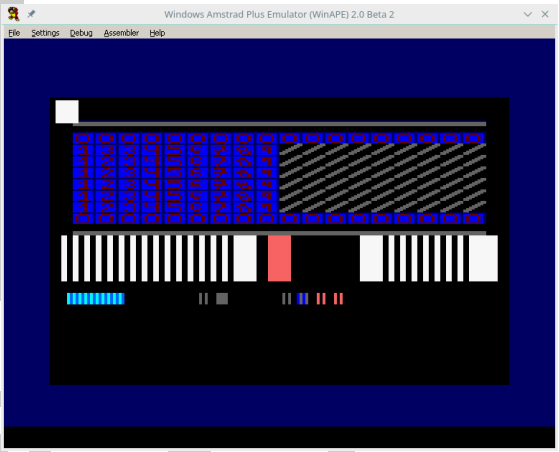
We reached a critical point in development, because the final product wouldn't look nearly as good if we didn't have a map to represent the enemies and the players. Since the CPCTelera engine provides a great method of **representing tilemaps** (maps that are compositions of little images called tiles) with tiles of 8x8 pixels and an accessible way to implement it, we decided to use it for the game. We had to create the map using the program *Tiled* that allows us to create a map using our own tileset, defining the width and height of the tiles and which tile is going to be used in all the positions.



Tiled's screen when creating a tilemap. You can see in the corner right the tileset, and on the center of the screen the tilemap that is being currently created using the tileset's tiles.

All of the **game's maps** were made with *Tiled*, a free software that allows users create their own maps made out of tiles, creating .tmx files that can be added later to the game very easily using CPCTelera. In order to create a map, you need a picture consisting of "Tiles" or rectangular sprites that you will later have to import to Tiled. Once you do that, you can use those tiles in a new Tiled file to create a map.

Using *CPCTelera's* tilemap_conversion, we had full access to the map's data and we were able to represent it on the screen. However, we ran into another **problem:** the tileset along with the tilemap consumed way too much memory, so we also used *CPCTelera's* methods to compress files. That way, only one decompressed map would occupy memory at a time, allowing us to store more compressed maps in the future. At that time of development, we only had 1 map fully working. We linked this map representation with the screen_movement method that we developed before and the map changed its representation each time the player pressed the A or D keys (also restricting the borders of the map where the player could see how the tilemap ended).

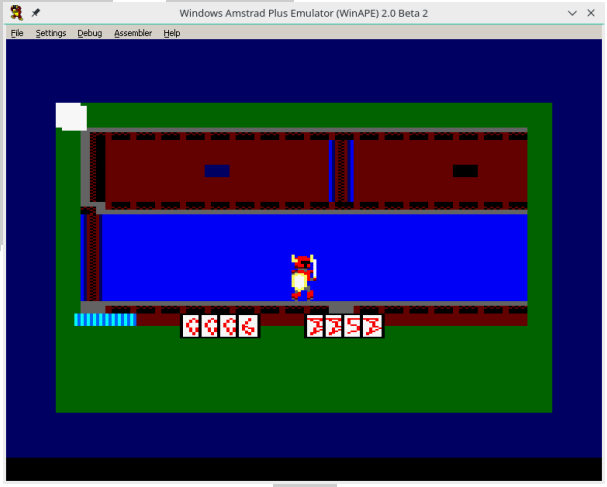


First tests of the double-buffer implementation. The enemies were not being erased after each step, and his old positions could still be seen in the form of a trail.

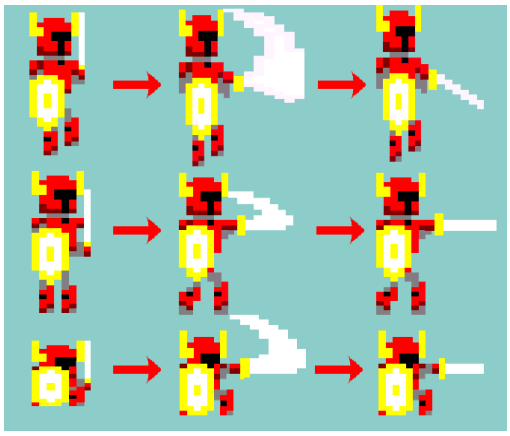
Something that was worrying us from the beginning of the developing (meaning, another **problem** raised) was how **slow** our game was moving each time we had a high amount of elements on the screen: it's no secret that the more entities there are on the game doing their stuff, the slower the machine would go. The problem this time around was that erasing the enemies and the character from the screen was taking too long, and the game speed was seriously affected. Also, a very constant flickering would happen at the first lines of the screen, so that was being a prominent problem too. However, we knew that if we implement a **"double buffer"**, the game speed would be seriously recovered and everything would go much more fluidly, and so we did.

To implement the double buffer, we switched between two memory spaces where the video memory was going to be located (0x8000 & 0XC000), switching them after each iteration. This process was extremely fast, much more than erasing the enemies in a single video memory buffer. However, it had a price: the memory that the Z80 needed for video memory was doubled, leaving us with only barely 8K of memory to store all the code, sprites, compressed maps and uncompressed free space.

Double buffer aside, everything was still made out of boxes and the mere process of representing sprites was something that would also require to be programmed, so we included the **sprite support** around this time. We had everything that we need, because the boxes represented the exact size of everything we had to represent, we just needed to place sprites there instead of boxes.



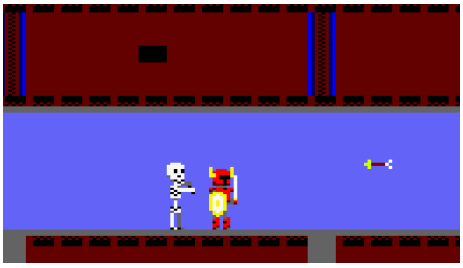
At first, we didn't have much of the sprites finished, but we could at least represent them in the game as placeholders until we developed them more.



Reference sheet used for the character and hitbox sprites when he attacks.

In our renderer class, we included sprite support to draw the sprites of all entities. We started developing the main character sprites and animations (walking and attacking). Because sprites consumed a lot of memory, we used the sprite flipping method from *CPCTelera* (cpct_hflipSpriteM0_asm) to flip the sprites, that way we had to create 2 times less sprites than before. To do the animations, we implemented a method that changed the sprite being represented every X iterations: 1 iteration for sword frame (the beam effect of the sword only lasts 1 frame) and 3 iterations for each step of the walking animation (the frames change every 3 iterations).

Our Knight had his sprites and animations implemented, but the enemies on the other hand were still missing their graphics. **We implemented 2 simple sprites for the enemies** (that would change every 2 iterations) that represented the actions of the enemies. Each time 2 iterations pass the sprites are updated creating a movement sensation. We also added sprites for the projectiles.

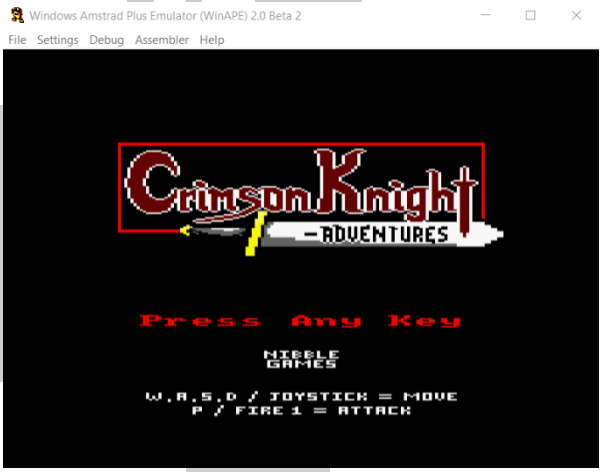


First version of the basic enemy and arrow sprites.

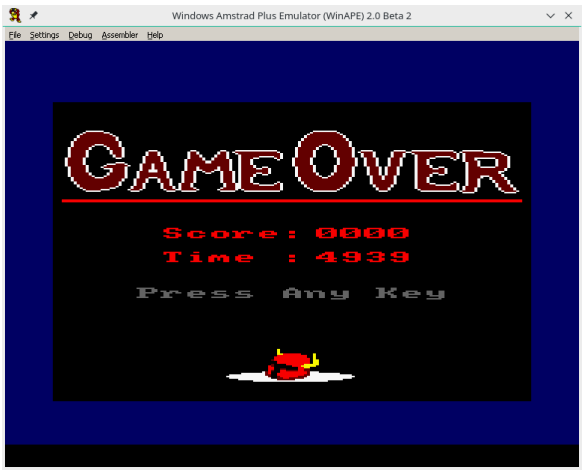
One **issue** that we encountered was that we couldn't use the sprite flipping method the same way we used the last one: the flipping method that we used changed the memory values of the sprites, so any other entity that access that sprite would also represent himself with the sprite flipped, so with enemies and the projectiles, we couldn't do this because there could be multiple enemies on the screen facing different directions.

Late stages of development (Weeks 5-6)

When we tested our game in a real Amstrad CPC, we noticed how the joystick and the buttons didn't work, we had to assign a different key input to them. We did some research and noticed how the joystick and fire buttons had different values than the keyboard keys, so we tested and implemented them on the same day. That means that **players can move using the Amstrad Keyboard or the joystick with the fire buttons**.



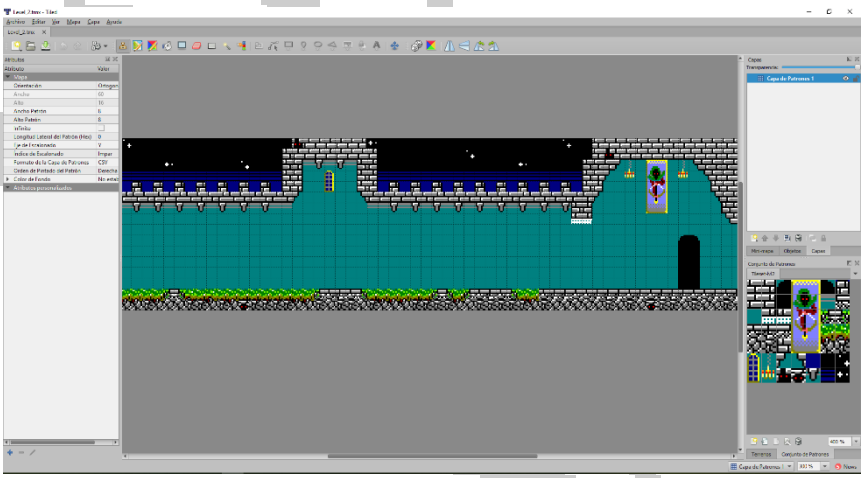
Final version of the title screen of the game.



Game over screen of the game.

Adding sprites was consuming a lot of memory, and we knew that adding a sprite for each letter was not going to be a very good solution to represent text, that's why we used *CPCTelera's* method to **represent text** (cpct_drawStringM0_asm). We were able to represent text everywhere we wanted. Also, a function that makes text appear and disappear was implemented to recreate that retro feel of the typical "press any button to play" texts. To save even more memory, we **directly printed** an image into these message screens, so we wouldn't have to import more sprites only used once (using *CPCTelera's* image_conversion.mk config file with the "screen" properties).

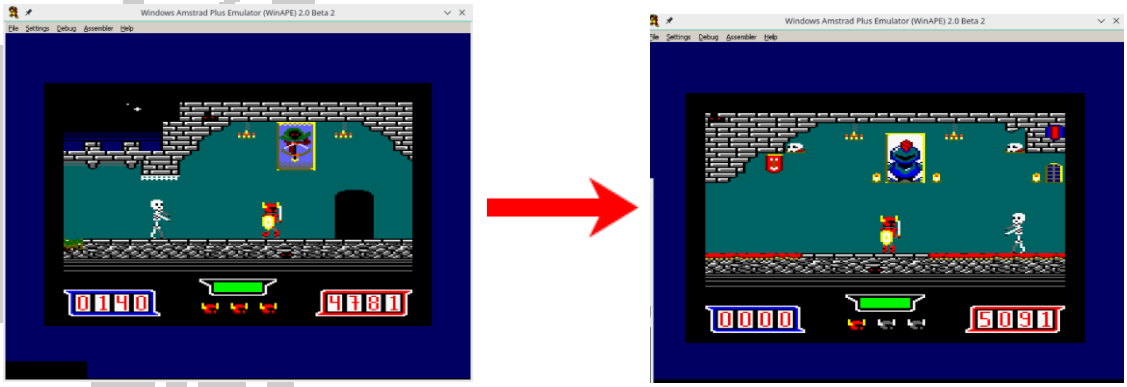
The message screens still looked very simplistic, we needed something more dynamic. We added **text and sprites to the title screen**, the game over screen and the rest of the screens that could appear in the game to make it much more detailed and easy to understand to the player.



One of the definitive levels being designed in Tiled.

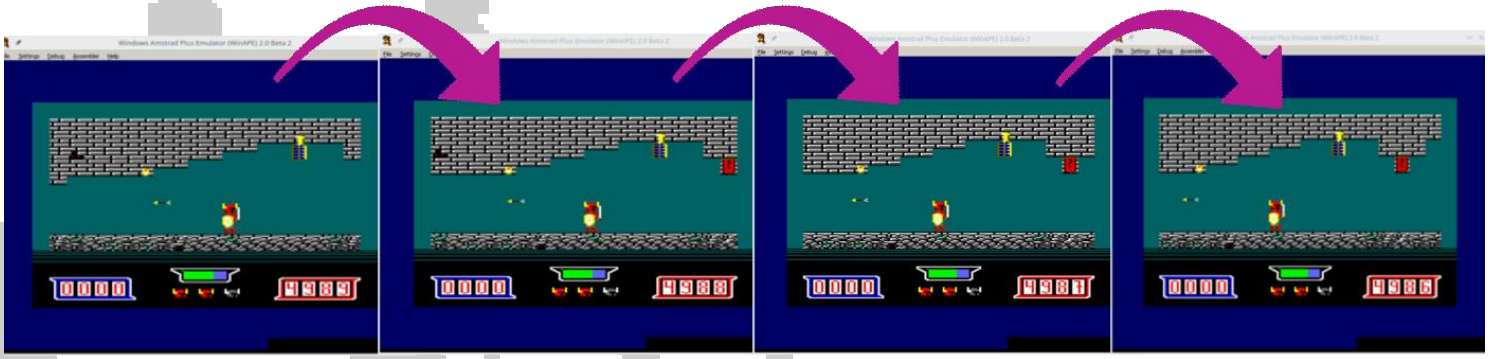
Even though we created one level, that was obviously not enough. **We created more levels**: the tiled files, used the tilemap converter, compressed them in memory and created a method that would switch one uncompressed map with the other. We had to create a method that would restart the game from the first level, and so we did (simply repositioning the player, map and entities). With all this, it was finally possible to add as many levels as we wanted.

Those levels were designed using *Tiled*, like we did for our test maps: creating more professional tilesets (the image that contains the tiles) highly improved the maps. In the picture of the previous page can be seen how one of the levels looked in Tiled.



All of the game levels are connected through gates. If the character map position is the same as the door map position, then you go to the next level of the game.

The camera system that we had been working with was a bit off, as we stated earlier. The enemies felt like they were sliding through the floor, moving weirdly on the screen but not repositioning themselves correctly when the “camera” was moved.

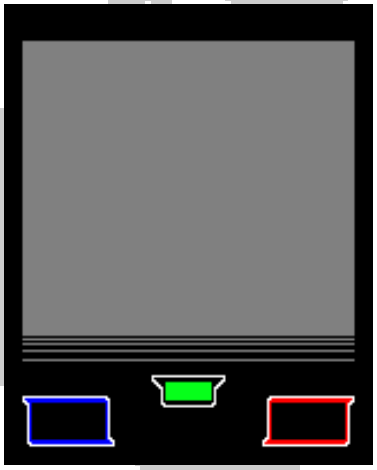


Visual representation of how our definitive camera works in the game. Player and camera are different entities now.

We were always assuming that we wanted to make the camera move along the player , but **we decided that, instead, the camera should be its own unique being** that followed the player when the player is too far away. So, we allowed the player to *actually* move around the screen and re-estrutured the coordenate system for the player and enemies, making it relative to the correct screen position according to the amount of screen re-positionings that happened. In other words, when the camera moves, it updates the player, the enemies and all the obstacles an amount of space equal to the amount of distance the camera moved to the left or the right. Also, we checked that the game wouldn’t move the camera to the left at the beginning of each level.

There was something basic that wasn't programmed yet: **there couldn't be the same amount of enemies in each level and not all the types of enemies would be displayed on the first couple of levels.**

To control this, we established **an enemy generator** that would make enemies appear with different amounts of time: sometimes random and sometimes predetermined. For example, in the first couple of levels, the obstacles are generated always at the exact same time, but a couple levels next, the enemies are randomly generated (with an interval, because harder levels have less waiting time between enemies)

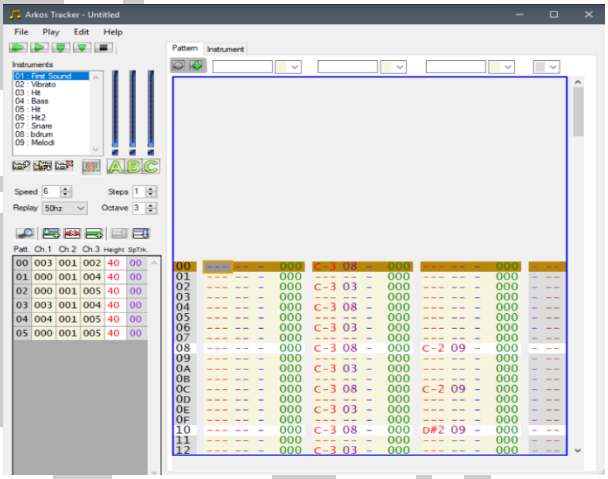


HUD basic reference design.

This is how the HUD actually looks like in-game. You can see the life system displayed beneath the health bar.

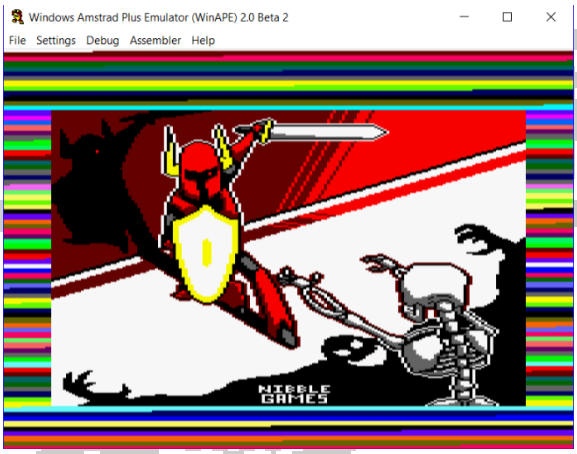
A visual representation of the player HP, score and time left () were very important for our project, because it's important to show to the player how they are progressing through the game and if they should be more careful in order to not lose health.

During the development of the HUD, we realized that the game would be less frustrating if we had a **remaining lives system**, so we added a total of 4 "retries" in the game. If the character life drops to 0 after the 4th retry, then the game is over. Those retries are represented as helmets on the game HUD.

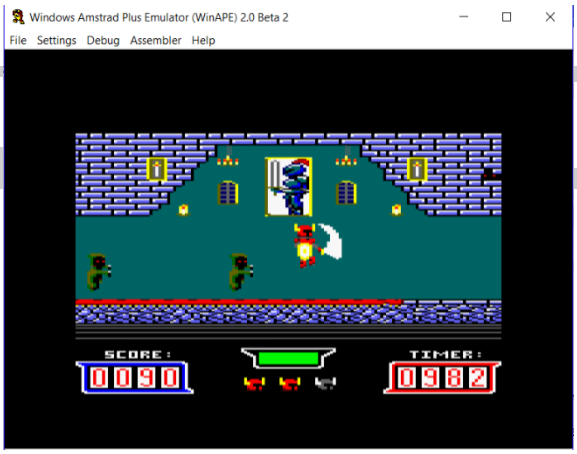


This is how Arkos Tracker 1 interface looks like

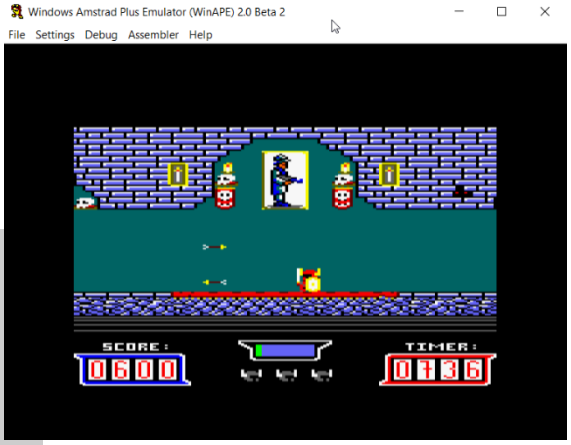
The **music** was created using *Arkos Tracker 1*, a software that can be used to create music in .aks format. The good thing about Arkos Tracker is that the music's pure byte data can be easily extracted using it. Using CPCTelera's music_conversion config file, it was very easy to implement music in our game in real time. The music is about 50 seconds long. We play the music at the interruption handler (see page 4).



CDT/Tape custom loader with the game's cover.



This is how the final game looks like. Notice the updated HUD and the new enemy type.



A reference to Chicago's 30 can be found in the 5th level wall, on one of the portraits the protagonist is displayed.

We improved the graphic quality of the game, to make the experience more enjoyable, all of the enemy sprites, background levels and player animations were improved in order to make the game look much better, we are very proud of the game's aesthetic. The final days of development, a series of final changes and details were made:

- Added a couple of levels reusing some of the tilemaps.
- Improved the HUD, displaying the elements much more clearly.
- The remaining time now is added to the score at the end of the level, along with a little animation.
- The music is not played on the main game.
- Title screen now displays the controls.
- We added a Chicago's 30 easter egg where the player can find a picture of the main character on a portrait on the wall of the Level 5.
- Another type of enemy was added: it had more speed than the average enemy and the players will encounter after the 4th level.
- Added a modified charger for the CDT file made with CPCTelera, which shows the game's cover.
- A total of 7 levels were fully developed, each one offering a different experience to the player, becoming increasingly harder.

These were the more relevant improvements. While all of these changes may not seem very important they really improve the game's potential and expands its content.

Final words and important lessons learned.

During development, we learned a lot of important lessons. Things that we could only experience working on a project like this one:

- **The power of communication**: when working on a team, is very important to take the time to stablish the basics of the project and specify most things of the project to avoid unexpected results later. When everyone knows what the project must be like and what are its characteristics, the work flows better.
- **Don't just learn how to make a program work, know why it's working**: This was constantly reminded to us by our teacher, knowing how to write a code that works is not everything: if you don't know why is the machine behaving like the code says, you won't learn to be a true engineer because you won't have the capability to resolve the true and hard problems that you might encounter in your way.
- **The importance of a correct debugging**: Imagining what the code must be "doing" is not what must be done in order to debug a program. Even if you think you are 100% sure of what some lines of code might be doing, nothing will be more revealing than looking directly at the memory and know what's changing, and why. We think that there are some problems that are impossible to solve in other way than looking directly at how the memory is on a specific moment.
- **Organization is everything**: A correct schedule is the key of success. Knowing what everyone has to do and what tasks are more important (or depends on other ones) is vital for the team.
- **Always keep in mind the limitations**: Amstrad was a very limited environment, and the limitations of the machine where the game should run must be highly noticed when developing.
- **Make iterations over the finished product:** On the last weeks of development, we were constantly reminded that we should always have a finished product before continuing to develop over it to have a playable game at every moment. This could prevent potential critical situations were a the final version of the game might be unfinished.
- **Things doesn't usually go the way you plan it**, when that happens the most important things are: re-evaluate the situation, re-arrange the time you have and **work hard to finish your product**.

Resources used for this document:

File used	License used
https://commons.wikimedia.org/wiki/File:Curved_Arrow.svg	CC 1.0 Universal
https://commons.wikimedia.org/wiki/File:Short_left_arrow_-_red.svg	Public Domain

The rest of the pictures and screenshots were either took or made by Nibble games.

Thank you very much for taking the time to read this document, we hope you enjoy *Crimson Knight Adventures* as much as we did.