GET OUT OF MY STORE!

MAKING-OF

HOW THE GAME WAS CONCEIVED

The first idea we got to make this game was far away from the final product we have right now. At first, we wanted to make a game whose main mechanic would be dashing! When we in the moment thought that our concept was nearly impossible to implement in such a tiny deadline, we decided to group up and lay out some ideas.

The next idea we had was a top-down shooting game with a bow and arrow, in which you'll have to shoot some bullseye with your bow and complete some puzzles. But that idea didn't convince us at all since it didn't pose a challenge to the player regarding movement and mechanics. We wanted something more fast paced, which required practice and skill, not just knowing the answer to the puzzles.

Then, an idea just came into our minds: What kind of game makes us play an infinite amount of hours? And then, we all thought the same: survival round-based games.

Our main influences are:

- Smash TV, an Amstrad CPC classic.
- Boxhead, a very popular flash game.
- Call of Duty Zombies, the most popular side mode of every Call of Duty iteration.

TECHNOLOGIES USED

The software we used to make this game is:

- Visual Studio Code for programming.
- CPCTelera as the framework.
- Arkos Tracker for making music and SFX
- GIMP for making sprites and scenarios.

DESIGN PROBLEMS AND SOLUTIONS

During the development of Get Out Of My Store! We faced many problems and challenges.

The first problem that we faced was the enemies. How many of them should we include? And how difficult should it be? For this problem, we decided to make easy enemies, each time increasingly difficult to both implement and play against. We started with the zombie, which is the simplest enemy of the game because its pattern is just to go to the

food. After that, we made the ghost, which is quite a simple enemy whose pattern is to chase you, with almost the same code used for making the zombie.

With only these two types of enemies, we noticed that the game lacked substance. We needed something that made the player take decisions and constantly change his game strategy. So we decided to make more enemies. We made enemies that had simple movement patterns, like UFOs and bats.

When these enemies were implemented, we made more difficult enemies, like the pumpkin, whose pattern is to seek you and then go to your position at a high speed, the spider whose pattern is to move between the walls, and the skeleton, whose pattern is erratic but predictable.

The second big problem we had is the playground. We wanted a game with only one scenario and we wanted to use this scenario as much as we could. For this issue, we made a door system, in which, on some levels, some doors will open and some will close. This includes changes of strategy because it's not the same having to deal with enemies with 2 doors than dealing with them with 5 doors laid out differently. These changes made the game more entertaining and less difficult in the first stages of the run. While making it more interesting and demanding in the later stages.

TECHNOLOGIC PROBLEMS

Technology-wise, we had many difficulties with assembly language, since we were just learning it for the first time.

Ironically, our main obstacle was not breaking the game while building it.

In many instances, we accidentally overwrote chunks of code, which lead to various (and sometimes scary) crashes. Other times we mixed up the tags we were using in some parts of the game, which led to what's broadly known as 'spaghetti code'. Nonetheless, we continued forward, learned from our mistakes, and during the last weeks of development, it flowed as smooth as butter.

Having learned how tags work, how the CPU writes and reads from the RAM, and most importantly, how to not break chunks of memory by accidentally drawing a box outside of video memory, by the last stages of development, we felt much more comfortable with

assembly language, and even took the liberties of adding features to the game we would had never thought of us being capable of doing in so little time.

Our little journey to solve flickering

As you may know, our game has tons of enemies, and is very fast paced, so, inherently, we would have problems with the render. We had flickering and framerate issues at the start. We had two main options to solve this: Double buffer, or using Interruptions to time the rendering process.

At first we tried making the double buffer, but it had many downsides, including having to lower the screen resolution, and making many changes to what we already had, without even mentioning the cost in RAM it would imply. After a couple of tests, we decided it wasn't the way to go.

We finally went for interruptions. But it had a tiny problem; Writing any kind of string of letters would mess up the timings of the interruptions, and to make the render solution we needed, we required perfectly timed interruptions. The way we came about this was implementing a custom font, with our own method of drawing strings that wouldn't touch the interruptions.

Once we had the custom fonts, another problem came up. The memory of the game was broken. Every once in a while the whole game would crash, and have all sorts of scary glitches because we were randomly wrecking parts of the RAM.

After many hours of debugging, we found out what was breaking our game: A badly initialized rendering value.

All entities on screen save a pointer to the last position they had on video memory so they can erase the sprite that was there before drawing the next frame. What we didn't take into account was that the pointer in question was initialized to 0x0000, which is a pointer to the start of the RAM. Therefore, in the first frame of the game, the first entity would first try to draw a box on its previous position (which was 0x0000), which wrote random bytes inside the code of our game, and if you were unlucky enough, you would encounter one of these random strings of bytes where code was supposed to be, and the game breaks.

We solved this by initializing all entities' previous pointer to video memory to 0xC000, which is the start of video memory, so we wouldn't break the code of the game.

With that out of the way, we could FINALLY fix the rendering issues.

We made a variable initialized to 0. It was the should_render flag. Long story short, if this variable was 0, the game loop would wait until it is not 0, so it can start rendering the next frame. We set the variable up with interruptions to always become 1 when the raster had covered about 3/4ths of the screen. So we would start rendering the next screen in advance, without affecting the game logic.

But, however, this implied another problem. Now we had permanent flickering on the bottom part of the screen. Because it would overwrite the current screen with the next one in that bottom part.

The fix we finally found for this was to change a bit the way we rendered the game. We had only one function to render all entities, but we split it up to two. One that would render the upper part of the screen, and another that would render the lower part of the screen. The only difference between these two is just a comparison in the Y axis before rendering, to know if that entity was in the upper part or the lower part. We made another variable that would indicate when to render the lower part of the screen.

So, we would start rendering the upper part of the screen first, and when it ends, it would wait for the second variable to become 1. By interruptions, we made this second variable to become 1 in the 6th interruption around the Vsync area, so it would render the lower part of the screen when we knew for a fact the raster had already gone over it.

With this done, we solved our flickering issues. We went from being able to draw 3 flickering enemies on screen, to being able to draw 4 perfectly clean enemies plus the main player plus a bullet without the game loop throttling or the sprites flickering.

Right now, the only flickering that can be perceived is when you have the maximum number of enemies on screen and you shoot with the fastest weapon in the game from one side of the room to the other. The sprites only briefly blink, but decreasing the fire rate or the enemy count would negatively affect gameplay, so we decided to keep it this way.

SCREENSHOTS AND VIDEOS

You can see all the development progress of Get Out Of My Store! in our **twitter account**.
https://twitter.com/PyroBombastic