Report: Stardew of Isaac

efrei PARIS

Introduction

The free project is, in comparison with projects seen in first year, more complex, especially because we have to define our functional specifications, where to stop, and how to do it. The only two requirements were to use trees or linked lists, and to create the code using the C language. This project also has a bigger scale, and requires an important involvement of the students

We decided to build our own game, from scratch, applying knowledge about game mechanics we have found from our own experience and research.

Presentation of the project

Summary

Stardew of Isaac is an adventure where you incarn Isaac, a young farmer that inherited a farm from their grandfather. But the farm is not like any other one: it is cursed! The only way to harvest your crops is to go inside of them, and clean them from the monsters, including a terrifying boss.

We designed the game as a mix between a farming simulator and a rogue-like. Indeed, we were huge fans of Stardew Valley (a life/farming simulator) and Binding of Isaac (an exigeant adventure and action game where every run is different), and wanted to pay tribute to them, while improving our competences.

Mechanics

The player's core game loop is to plant seeds, wait for them to grow, and finally go inside them (the dungeon!). Inside the dungeon, the goal is to reach the item room, protected by several monsters and a guardian. This room allows the player to finally harvest the vegetable.

However, the game does not resume itself to this loop over and over again without any changing elements. During their progression inside dungeons, the player is collecting loot (loot quality depends on the difficulty of the level). This loot can be used to earn money at the shop, and buy new seeds to keep going in the game.

We also wanted to forcelong-term investment. Firstly, the player can temporary boost its statistics (speed, damages, speed of the weapons) using potions. They are able to provide an upgrade until the player moves to the next day (action done by saving). Secondly, by completing a dungeon, the player collect the vegetable. It can be sold for a non-negligible amount of money, or be eaten. This action is providing a permanent upgrade of different statistics, rewarding the player.

The reason these long term effects are important is that the difficulty of the dungeons rises as you move to different vegetable. The difficulty level affects the health and damage of enemies as well as the number of enemies that will spawn.

But, because it wasn't enough mechanics, we implemented two different weapons, with different damage, speed and hitbox.

Dungeons are a huge part of the game (this is where the player spend most of the time). First of all, the structure of the level is completely random (but rooms in themself are not, to avoid generation problems and ensure quality). They are separated into different zones, requiring keys to access them. They also are inhabited with three types of monsters, with different patterns. Finally, a boss is present at the end, with several attacks.

Details

Having raw gameplay isn't enough to have a reliable software. For this purpose, we added several extra features and polished everything, to propose an improved experience.

We created a whole ambiance, by implementing interfaces, but also having many alternative graphics (for dungeons), and adding sounds and music. This doesn't seem important, but it adds a lot.

Always with this idea in mind, we developed the system of save, and added options, to let you personalize your session of gaming. It is possible to change the framerate, the sound volume and keys.

Structure of the code

Due to the ambition of the project, we had to quickly figure out how to separate our work. To properly do it, an engine was built, allowing us to easily split code.

Engine

The engine is shared into several files. Firstly, it initialises functions,data and the main structures needed. Secondly, it is in charge of the game loop. Indeed, we decided to divide the continuously looped section of the code into three sections.

The first one is the event part, mainly dedicated to the properly receiving user input. The second section is dealing with the logic of the game, processing the player's output and accordingly change some of the game's data, as well as run the ongoing processes. Finally, the third part is accurately displaying the current state of the game on the user's window.

These three loop parts, plus one, run only once for the initialisation of the data used inside, represent what we call a scene. In other words, a scene is an environment (overlay, menu, playable, etc) with everything it needs to properly work (user interaction, effects of them, visual feedback).

It may seems simple, but in order to have something clear, we decided to use pointers of function, an odd functionality of the C language, so that we could efficiently chose which scene to run from one core loop.

To avoid leaks and problems, we centralized the loading of elements from the hard drive (sounds, musics, images, fonts). Effectively each scene has a list of sound/image/… elements it may need need, and each time a new scene is loaded, the old elements are freed from the hard drive as the new data is loaded onto memory, to quickly use it when necessary. The system of scene, images, fonts, and sounds is accessible through the Engine structure. It is also used to keep in memory the configuration of the user (FPS, keys, volume levels).

Moreover, we needed an access to general data in memory, without having a mess of variables. That is why we have the Data structure, only used to keep in memory data at a fixed point. It is storing the different variables of each scene. However, because each scene has its own needs, they all have different variables. We also have inside this structure the list of active mobs, a reference table for the items, a variable to stop the game…

The main code is at the root of *src* and of *src/engine*. The code of scene is located in *src/scenes*. Resources (fonts, images, etc) are inside the folder *src/resources*.

Data processing

In addition to the system of engine, we had to process all the informations from scenes. It includes:

- The items system (inventory, shop)
- The statistics, upgrades, displacement and combat of the player
- The entities (interactions with the player, movements, loots)
- The specificities of each scenes (generation of dungeon, tutorial, actions inside the lobby, configuration, loading save, etc)

The code for the general process is located in *src/engine/game*. Entities ones is under *src/engine/game/entities*. Note that the algorithm for the dungeon generation is so important that it has its own folder: *src/engine/game/dungeon*.

Some values are coming from the hard drive, such as metadata of maps, blank save, etc. Those are inside the folder *src/data*.

Note that most of lists that we are using are linked lists (single or double), because they are really reliable and flexible to process dynamics data.
Some of the hardest part of our code comes from dungeon generation. It is there that we used red black trees and dabbled in graph theory in order to successfully generate dungeons with random maps.

Difficulties encountered

Graphism

We faced many problems with the graphics, because none of us had already experience making graphics for a project of this scale. The first issue was to start from scratch because we weren't really inspired to make a graphical charter. Another problem was the time that we spent making the graphics, because of our lack of skill with digital design at that level and the fact some of us were discovering software, it made the artistic creation incredibly time-consuming. Moreover, the reliability between designing elements in full size and in showing them in real size, blit into another surface, was very complex because of the 3D effect and the textures that do not render the same at all in different size.

SDL and its libraries

A (good) game without a graphical interface is hardly conceivable (that doesn't mean it is impossible!). That is why we decided to use SDL (Simple DirectMedia Layer), and some of its libraries to have a much more reliable tool. Those libraries are SDL_TTF, SDL_Image, and SDL_Mixer. TTF and Image were basically used to easily print text and display image. They were not that hard to master. However, Mixer has been giving us some harsh time. It is used to control sound and music. But, like SDL and other libraries, they are really low level. It is very useful to control everything, but it means that there is no safety.

A lack of knowledge of the documentation and the way the code work made us lose time on a strange bug, that was leading to a full crash, without the possibility to use the Stack, the debugger or Valgrind.

C, a non-optimal language to correctly work on our project

Let's get things straight, C language is very powerful and offers a ton of possibilities. It is also cross platform and very stable.

Nevertheless, it wasn't adapted for a project such as a game in such a time scale, for several reasons. The first obvious one is the non-presence of Object Oriented Programming. It forced us to use some tricks (like function pointers), and was preventing use from securing some variables (a simple example: if we alter the life of the player without using a specific function, bugs will occur).

Nonetheless, we now do understand why it is used to build some project such as the Linux Kernel. Its rigorousness and its lack of conveniences are make it a highly optimized tool.

Collisions and displacement

The way the game handles collision and displacement was complex to set up.

First of all movement. The information of the map (block, wall, etc) are stored inside a 2D array. To have a correct velocity and acceleration, we needed to access to cells were the player would be (basically, the cell with coordinates pos+velocity for x and y). But with an important velocity, it was possible to have a bigger number than the size of the array, crashing the game. It was also hard to figure out when and how to stop movement so the player wouldn't clip inside a wall, or would be slowed down for no reason.

Also, displacement (as well as basically all functions which deal with a moving element is made in function of time (and not frames). This is great as it allows for users to run at different framerates without gameplay being impacted in any meaningful ways. But this forced us to use timers, complexifying the process at basically every step.

Collisions between entities wasn't a happy time. The difficulty was to correctly size up hitbox of weapons, projectile, mobs, and then to check if the box one of them was cutting another one. For a lot of enemies knockback then had to correctly happen.

Managing a team

This problem wasn't induced because of technical issue, but the way we have been working. Even if we had a roadmap and a list of task to complete, we did not carefully distribute the work. Of course, some of us weren't happy with the tasks they had, and others had much more work to do.

Those two problems generated tensions between some members, and in addition to communication issues, it blows up at a moment's notice, jeopardising the project. One of the main way to fix that problem was to start using Slack, a communication tool, and writing everything on it, so that everyone would be on the same page

Furthermore, we decided to rebalance jobs, giving more works to some, and less to others. But this avoided us having other important conflicts inside the team.

Note that we did not really balance the work well. Some person were too dependant of the completion of other members.

Conclusion

We are proud of this project. We put so much effort into it, leading to having something actually playable, that we can only be happy of what we achieved.

This game has been helping us in mastering tools of development: we all moved from CodeBlocks to CLion, we have improved our way of using Git and discovered features of GitLab. Some of us have been developing their knowledge of Gimp and Photoshop. But of course, tools are not important compared to raw coding knowledge. We got better at using the C language and SDL, but also understanding some concept of separation of code and structures.

We also have been aware of some important part of a project, such as the conception, the distribution of charges, and the way we should solve issue.

However, we all think that this project was too great of a task for two months, compared to the goals others have set themselves, which allowed them to reach these goals relatively easily and then spend the remaining time thoroughly polishing every aspect of the project. We could have lower our expectancy, avoiding us a such important implication in the project.