Star Caller

Calvin Michele, Ethan Sahlstrom, Sheng Her

University of Alaska, Anchorage

CSCE A201 501

Kamran Siddique

July 28, 2024

Table of Contents

Contributions:

Sheng Her worked on player class, ship.cpp, star.h, shooting star animation, and main screen visuals. She also wrote the abstract, introduction, and github sections in this report.

Calvin Michele was the sole contributor to main.cpp, particles.cpp, solarsystem.cpp, and the resources folder, and contributed to the ship.cpp, marketupgrade.cpp, mission.cpp, star.cpp, and the star.h. He helped revise and write the README.md and wrote the GUI, discarded ideas and regrets, and references sections in this report.

Ethan Sahlstrom created marketupgrade.cpp and mission.cpp, and contributed to star.cpp, and the star.h. He worked with Calvin to plan and structure the project initially, as well as envision and communicate much of

the non-graphical code with a top down approach, which went into revising and the README.md and wrote the Game Loop section of this report.

Abstract

For the summer 2024 CSCE A201 end-of-the-semester project, our group decided to create a spacefaring theme game called Star Caller. Star Caller is an incremental game that lets you build up resources in order to level up to get more resources. The objective of the game is to pay off your debt to the company, Mort Corp. This game is not time-consuming and aims to be a simple play. We used the programming language C++ to write the code and also incorporated the lightweight programming library Raylib for the GUI. Our group collaborated through GitHub where we stored our project and were able to see and make changes to it.

Introduction

Our game, Star Caller, is reminiscent of the space-theme games we used to play as kids. Star Caller being an incremental game makes it easier to keep track of the items the player has by using simple arithmetics. Although we were a bit ambitious with this project, we didn't want anything too complicated in terms of resources and coding because of time constrictions. Therefore, the goals and missions of the game are simple and do not require many skills, making this game suitable for all ages. The objective of the game is to clear the player's debt owed to the company Mort Corp. To do this, the player must go on missions to earn resources and money.

These resources and money can then be used to level up the player to gain even more resources and money, or the money can be used to pay off the debt. Having a time limit allows for a definitive end, so if the player does not pay off their debt in time, they lose.

Process

Procedure

Our process started when we decided on what game we wanted to create and what resources we wanted to use. After that, we roughly followed the following steps:

1. Write out and explain our ideas to each other. This allowed us to be on the same page.
2. Write code for the idea. We used the programming language C++, and the programming library Raylib.
3. Implement the code into our main file. This was done by creating classes and header files.
4. Run it to see if it works. Since we had different devices and editors we had to make sure it worked for all of us.
5. Seek advice from team members. We often asked each other for help and input.
6. Fix the code if needed. We often had changes in plan and so we had to incorporate them.
7. Repeat.

GitHub

Our group used GitHub as our main source of collaboration, in which we stored all files related to our project there. GitHub made it easy for us to see what each other was working on

and allowed us to easily share files. It was a place where we could share our ideas and plans and keep track of where we were in the project. Although it was a bit confusing at first, because of its many features and processes, we slowly got the hang of it after working with it for a while. Overall, GitHub was a great resource for this project.

Game Loop

The game loop of Starcaller is more ambitious than anything we've tried before. It was primarily conceptualized by E. Sahlstrom and C. Michele, but the loop underwent iteration as we all contributed. What we landed on was a structure of three header files, six implementation files, and a main file. On top of that are resources, notes, and a Makefile. Header files "star.h", "raylib.h", and "raymath.h" supply the definitions for our game. The game loop begins, of course, with "main.cpp", our main file.

Main

The first thing that main does is initialize the user's class, as well as many graphical elements needed to present the game. The graphical features will be described in greater detail in a later section. The main function here begins the process of the game loop. The screen opens, the title card, stars, and custom font greet the user to Star Caller, and the game begins when the user opts to start a new game. Main progresses through these screens until we get to the main game screen, where a solar system is rendered with planets, buttons, and a hub. The implementations for solar system functions used by main are in "solarsystem.cpp". More detail

on the solar system visual feature (randomness, orbits, etc.) will be discussed in the graphics section. Each celestial body can be selected with a mouse click, and main handles the selection. The player's ship is represented on this screen by a small triangle, and each object clicked on is assigned as the ship's destination by main. Next, the lose condition is established. In Starcaller, the player will lose when "Time Til Repo" drops to zero. This condition results in a "Game Over" screen. The timer is displayed to the player and decreases according to the animation timer. Main also handles the win condition when a player is able to pay off his debt and escape repossession. In either case, credits are displayed, and the game will either restart or end according to the player's choice.

Solar System

Solar system classes and functions are implemented within "solarsystem.cpp". This file implements classes Sun, Planet, and HubPort, as well as certain standalone functions. The methods here allow the user to select planets or the HubPort. If they select a planet, the user receives a set of missions for each one. If HubPort is selected as a destination the player is presented with a set of upgrades available for purchase. Separate classes and implementations are employed for missions and market upgrades. A mission handling function draws buttons for the missions at each planet, which are fleshed out in "mission.cpp". HubPort, which utilizes the framework of a planet class, contains market buttons rather than mission buttons. HubPort's market handler initializes sets of purchasable upgrades, which are initialized in vectors of their class type. The Player class, which is implemented in "star.cpp", is referenced to determine which tier of upgrade should be drawn from the vectors.

Market Upgrades

All purchasable items are part of the market system, which is implemented in "marketupgrade.cpp". MarketUpgrade buttons are drawn using methods here, with stringstream implemented to format text according to a player's current upgrade tier. Buttons display information about the upgrade they reference and will change depending on whether the tier is maxed or if the player can afford that upgrade. Code for almost everything market related is implemented here, so that it's simple to initialize working buttons with upgrades in the HubPort methods. The code here works by tracking and modifying several player data types, allowing them to apply earnings modifiers, unlock a new mission type, and more. The player's ship is also referenced, and its speed is increased as upgrades are purchased. This is meant to give the player a satisfying and tangible experience of progression in Starcaller.

Player Class

The player class, which resides in "star.cpp" is at the heart of this game's progression. It maintains the player's debt, time remaining until game over, and upgrade tiers. Counters in this class are referenced to track which tier of upgrade the market should offer next. They also serve the dual purpose of designating the upgrade modifiers that increase the player's earnings and decrease their time cost. This file also handles several vital GUI features.

Missions

In order to earn money and win in Starcaller, it's important to take on missions. Missions have their own class, stored in "mission.cpp". Each one is initialized with a name, a reward, a time cost, cooldown time, and a button. Similarly to MarketUpgrade, mission's graphical and interactive features are built into the class to allow for easy initialization at planets. Missions present on the right side of the player's screen while at a planet, and can be selected while not on cooldown. When selected, the mission's methods awards the player with money and charges their timer the time cost. It cannot be selected again until the cooldown is over. One aspect of Missions that proved more difficult was the cooldown, which updates with a timer.

GUI

The primary catalyst for what kind of game we would be working on was the library we would be utilizing as a means of graphical implementation. Michele initially opted to create a small text adventure game that would rely on the ncurses library for C++. Ncurses was a simple terminal takeover that made terminal manipulation an easy endeavor but it wasn't quite sufficient to the scale that Sahlstrom and Michele were hoping to tackle. After much research and deliberation Michele found Raylib.

Raylib

Raylib seemed almost too good to be true, it was lightweight, easy to implement, easy to learn, and highly cross-compatible. Properly installing raylib was a difficult task for our amateur

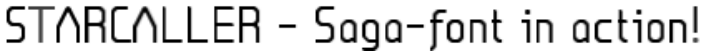
development team but it proved to be a fruitful endeavor. The initial plan was to learn and use QT but Michele was adamant about utilizing a ground-up library that didn't do half of the coding for him. Raylib opened the door for us to create a project that we could proudly call our own.

Initial Ideas

Michele's first ideas for the graphics of a spacefaring, idle-clicking, easy-to-finish-by-July-28 game involved multiple screens for market, mission, and status. With all of that in mind began writing the title card. At first the title card was just going to be a text across the screen with the designated title but with raylib incorporating images was as easy as incorporating text. Michele drew up some prototypes in Aseprite (a steam software for pixel drawing) and incorporated them into the project to be updated later.

Main Menu

Font is an overlooked art that can bring your ideas and aesthetic together in a subtle way. Michele, using the website Fontstruct, began working on a font that we would use as the primary font throughout the game. Saga-font is the result of this effort:



With the font created Michele set out to create a sleek looking title image that would give the player an idea of the projects theme. The soft glow of a galaxy pulses behind the words

STARCALLER as the player decides there course of action - to start a new game or to exit the game. While the options are limited Michele was still excited to share the fact the work as intended and change colors when you hover over them.

The Game

After some work the screens could be cycled through from a hub screen that the player would choose their actions from. While waiting for the mission, player, and ship classes to come together Michele idly created a particle effect for the starts in the background. These starts are ever present and occupy random spaces on the screen with a max limit that can be adjusted. The stars each have their own lifecycle and fade in and out, have randomized colors base on three carefully selected colors, and maximum alphas to represent distance. The stars cycle through three ASCII characters to represent a glow and are ever present from start to finish.

To capitalize on the nice atmosphere that the stars created Michele began to incorporate a Sun and planets. The finished product of the solar system has interactive orbits, orbital positions that change over time (based on some equations using time remaining, distance from sun, and planet mass), unique mission objects for each planet. The planets themselves are randomized to make every playthrough unique though at the turn-in stage of this game, this does not serve much in way of experience. The map was such a nice feature that Michele discarded the idea of the screens and added sub menus on either side of the map that serve different purposes. Michele's implementation of the map as a primary feature was sparked by Her's ship draw feature that allowed for a change in ship speed. Michele had an idea for shooting stars to add to compliment the stars and Her did a wonderful job of bringing this idea to life.

Discarded Ideas and Regrets

If we had more time there is a lot that we would like to have added to the game. NPC interactions, randomized mission boards for each planet that requires more thought into which mission you take, and interactive missions based on the mission type. These ideas were far too grand in scale and would have required several more months for a complete product though as we are still inexperienced developers just trying to pass our class. The next system that we would have wanted to implement first would have been save and load files but again, time was of the essence by the time we really got the ball rolling.

Michele had attempted to implement a retry feature but the code was not structured well enough that this could be done with ease. His use of global variables is his greatest regret with the project as they do not allow for a lot of flexibility. The functions within the classes were also a bit too rigid, so resetting all of their variables was a lot of work that he was not able to do in the last leg of this project's development. A lot was learned from this project and structure and file organization were crucial skills that this project enticed us to learn.

Credits

Raylib Quick Reference Card, 2024, R. Santamaria, https://www.raylib.com/cheatsheet/cheatsheet.html

Raylib Examples, 2024, https://github.com/raysan5/raylib/tree/master/examples

Raylib Games, 2024, https://github.com/raysan5/raylib-games

Special thanks to Ramon Santamaria for the raylib library and all of its examples to learn from.