Aeroscape

Team Java Project
CS 321-01
Group 2

Members
Tristan McGinnis
Jacob Neel
Nick Davis
Gavin Brady

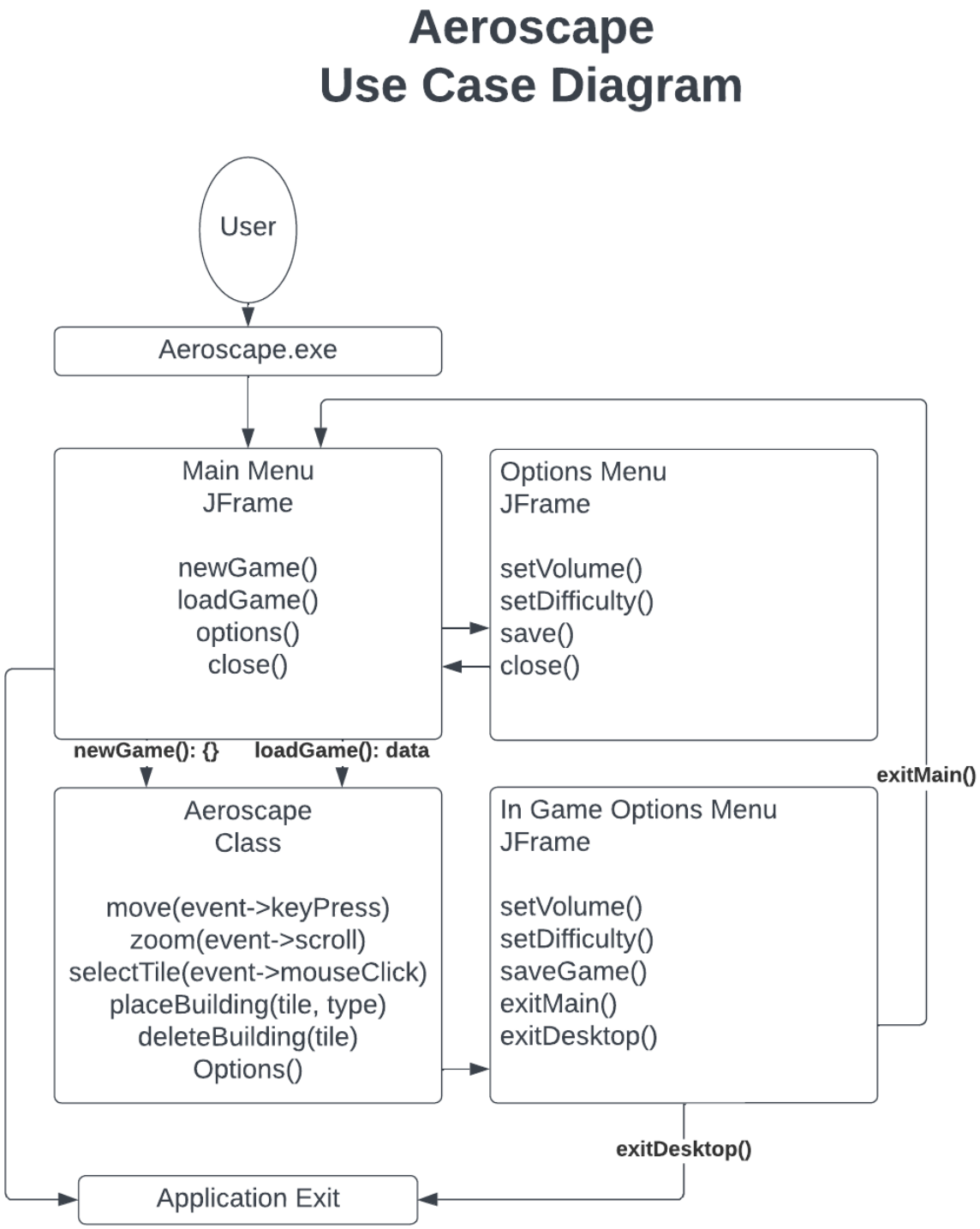
Description

In Aeroscape, players explore a grid-based world inspired by *Factorio* to gather materials to build structures and weapons to defend against enemies and work towards an escape. Each grid acts as a container to an object, making strategic placement of structures and defenses crucial for survival. Players harvest materials from the procedurally generated map upon creation by placing mining machines on resource spots and using smelters to make raw resources usable. Once the player reaches a critical amount of resources, escape is possible by constructing a rocket escape vehicle.

Table of Contents

Analysis

Use Case



Project Items

Essential Items

Discuss completed and uncompleted essential items
E-1: Rendering of the grid and playing field graphics
    Completed
E-2: Data(text) entry by the user
    Complete
E-3: Menu interfaces for interacting with the program
    Complete
E-4: Persistence of data across restarting of the application
    Completed (not fully implemented)

Enhancement (Additional) Items

Discuss completed and uncompleted non-essential items
A-1: Enemy forces to attack the player
    Incomplete
A-2: Item shop system
    Incomplete
A-3: ‘Improved Graphics’ (Enemy, Player, and Machine Icons)
    Complete
A-4: Defense system against enemies
    Incomplete
A-5: Sound effects/Music
    Close to completion
A-6: Working escape system
    Incomplete
A-7: Working machine harvest/Refine System
    Complete

Design

Basic Parts

The game is composed of these primary and secondary elements:

- ➔ Aeroscape (The primary game)
- ➔ Menus
  - ◆ MainMenu
  - ◆ OptionsFrame
  - ◆ GameSettings
  - ◆ LoadGameFrame
- ➔ Grid
- ➔ Grid Renderer
- ➔ Tile
  - ◆ Buildings
    - Miner
    - Smelter
    - SAM_PLATFORM
- ➔ Missile
  - ◆ BMissile
  - ◆ SAM
- ➔ Player
- ➔ Camera
- ➔ Inventory
- ➔ LevelData
- ➔ TextureEngine
- ➔ AudioEngine
- ➔ MouseInputManager
- ➔ KeyboardInputManager

The Player, Camera, Grid, Menus, and Rendering components are responsible for all GUIs, Visuals, Interfaces, and Interactions between the player and the game. The Tile, Buildings, and Inventory are responsible for overseeing the actions made by the player while the game runs such as using resources and placing machines within the grid. LevelData is responsible for providing the persistence for game data across different instances of the game. The Texture Engine and Audio Engines are responsible for loading in texture and playing audio data. The Mouse Input and Keyboard Input Managers are responsible for handling player input

The Domain Model

- Game World
    - Grid-based world inspired by factorio
    - Procedurally generated
    - Resource spots for material gathering
    - Strategic placement of structures and defenses
    - Contains: Player, Structures (Buildings)
- Player
    - Explore the game world
    - Harvest materials from the map
    - Build structures and defenses
    - Work towards constructing a rocket escape vehicle
    - Contains: Inventory
- Materials
    - Gathered from resource spots on the map
    - Require mining machines to collect
    - Raw resources collected require smelting for usability
- Structures
    - Placed on the grid and occupy a tile
    - Examples include miners, smelters, and SAM platforms
- Weapons and Defenses
    - Placed on the grid for strategic defense
    - Require materials for construction
- Rocket Escape Vehicle
    - Constructed with a critically large amount of resources
    - Allows the player to escape the game world and win the game

Design elements for technical items

Graphical User Interface

Element E-3 utilizes the MainMenu class. This class extends the JFrame class and implements the ActionListener interface to create an interactive main menu, allowing players to create new games, load games, access options, and quit the game. The ActionListener interface is used to define the behavior of these interactive elements. Although the current GUI is not finished, it is a solid foundation for future enhancement.

Text Formatting and Processing

Element E-2 uses the MainMenu class to handle text formatting and processing. The JTextField objects, such as ‘nameRequest’ and ‘nameEnter’, allow players to input

their names when starting a new game or loading a saved one. Additionally, the ActionListener helps manage user inputs and process them accordingly. When a player enters their name and clicks the 'nameButton' the text is then captured and utilized to initiate either a new game session or to load a saved game.

Graphics (Sampled & Constructive)

Element E-1 utilizes the GridRenderer class to manage the game's visual aspects, including rendering the game grid, player texture, and other game components. GridRenderer extends JPanel and collaborates with the Camera object to implement translation transforms on the grid and other game elements based on the camera's position.

Our graphics approach is characterized as both sampled and constructive. We employ sampled elements in the form of textures and images for various game components for components such as the player, water, and resources. On the other hand, the constructive aspect is evident in the dynamic creation and rendering of the grid and additional components.

The GridRenderer class takes several objects as input, including Camera, Player, LevelData, and Inventory, along with the dimensions of the game grid. This class is responsible for rendering the grid, player, and game UI in their respective positions, which adhere to the camera translations. The GridRenderer class is also equipped with multiple methods that handle the placement of different game objects like miners, smelters, and SAM platforms. The TextureEngine is used for managing the various images and textures needed for rendering different game components.

The use of a combination of sampled and constructive graphics allows for flexibility in design and the ability to modify and expand the game's visual elements. The GridRenderer class serves as a clear illustration of how the game's graphics are rendered, and makes debugging visual issues effortless.

Persistence of Information

Element E-4 uses only the class *LevelData,* responsible for taking in, converting, and logging, then parsing, converting, and returning the game's save data.

The route of persistence in our case was JSON. While difficult in requiring type conversion for special types and arrays, the JSON format allows data to be manipulated outside of the program directly within the save file due to the specified JSON format. Additionally, the SimpleJSON library made it much easier to view *how* data was being converted and saved through the provided SimpleJSON methods since the JSON format is standard. Data types and other information can be determined just by looking at the printed '.JSON' file which is helpful in debugging issues with type conversions to-and-from JSON objects.

Implementation

API Overview

Rendering of the grid and playing field graphics: The GridRenderer class, which extends javax.swing.JPanel, is responsible for rendering the grid and playing field graphics.

Data (text) entry by the user: The MainMenu class which extends javax.swing.JFrame, handles user text input.

Menu interfaces for interacting with the program: The menu interfaces are implemented using the LoadGameFrame, MainMenu, and OptionsFrame classes, which extend javax.swing.JFrame.

Persistence of data across restarting of the application: The GameSettings and LevelData classes, both are responsible for managing and persisting game data across application restarts.

Public Elements

Aeroscape: Initializes the game and its components, and manages and updates the game loop. This in part satisfies the responsibilities of initializing the game and managing the overall game loop to facilitate the rest of the program.

AudioEngine: Loads and plays audio effects. This in part satisfies the responsibility of managing audio resources to provide feedback to the user.

BallisticMissile (extends Missile): An enemy missile that spawns to target the player's structures. Helps satisfy the need for enemies that the player needs to defend against.

Building (extends Tile): Facilitates the placement of structures onto tiles for harvesting and refining resources. Helps satisfy the need to allow for construction of buildings by the player and harvesting of resources.

Camera: Follows the player character around the map to facilitate exploration. Helps satisfy the responsibility of letting the player traverse the game world and look for resources.

EnemyHandler: Manages the spawning of enemies within the game. Helps satisfy the need for enemies that oppose the player and must be strategically countered.

FireControl: Manages movement of SAMs fired by player buildings. Helps satisfy the need for strategically placed defenses.

GameSettings: Stores and loads setting preferences for the game. Helps satisfy the need for modifiable settings.

Grid: creates a grid to store and manage all tiles. Helps satisfy the responsibility of generating a grid and managing its tiles.

GridRenderer: Renders the grid and its textures in a way that can be viewed by the user and manages object placement on tiles. Helps satisfy the responsibilities of rendering the game's textures and placing objects onto its tiles.

Inventory: Stores the resources and ingots the player has gathered. This helps support the harvesting of materials in order to work towards a target amount.

KeyboardInputManager (extends KeyAdapter): Receives and interprets keyboard input for the player, allowing them to interact with the game and move throughout the grid. Helps satisfy the need for exploring the game world.

LevelData: Stores data related to the player's inventory and the map's layout to allow saving and loading between sessions. Satisfies the need for persistence and saving and loading game data.

LoadGameFrame (extends JFrame): Loads the main game window. Helps satisfy the responsibility of giving the user a means to view the game world to explore it.

MainMenu (extends JFrame): Creates the game's opening menu, from which they can enter the game according to a selected grid size and alter the settings. Helps satisfy the need for procedurally generating and opening the main world.

Miner (extends Building): A type of building that can be placed on a tile containing resources to harvest said resources. Helps satisfy the need for resource harvesting and building placement.

Missile: Facilitates the functionality of enemy and player missiles. Helps satisfy the need for incoming enemies and defenses against them.

MouseInputManager (extends MouseAdapter): Receives and interprets mouse input for the player, allowing them to place buildings on appropriate tiles. Helps satisfy the need for placing structures for harvesting, smelting, and defending.

OptionsFrame (extends JFrame): Creates a graphical interface for altering the settings. Helps satisfy the need for modifiable game preferences.

Player: A player character through which the user sees the game's world. Helps satisfy the need for a player that can explore the game world to interact with it.

SAM (extends Missile): Fired by the SAM platforms as a defense against incoming missiles. Satisfies the need for defenses against enemies.

SAM_Platform (extends Building): A building type that fires SAM missiles as a defense system for nearby buildings. Satisfies the need for the player to construct defenses.

Smelter (extends Building): A type of building that can be placed on tiles not containing resources that refines ores in the player's inventory to convert them into ingots. Helps satisfy the need for building construction and smelting resources for usability.

TextureEngine: Accesses and loads textures in the grid. Satisfies the need for player, enemy, and field graphics, specifically that of enhanced graphics.

Tile: Provides a basis for resource generation and building placement. Helps satisfy the need for generation of the game world and the resources contained within.

Exercise & Testing

Testing

Testing was exercised in a low-level to high-level format for each component of the project. Components would be tested at the individual unit test level (Method/Constructor to a low level class) before working upwards in the hierarchical tree of classes. An example case for testing the *Miner* may go like so:

1. Unit-Test the Miner class via an isolated *Main* or similar method executed within the *Miner.Java* independent from the rest of the program.
2. Scoping outward, test the implementation of *Miner* from within the *Builder.Java* (one step upwards in the hierarchy) via a similar independent method from within the Builder class.
3. A final test of the Miner class can be completed from *Tile* such that it uses *Builder* to create, access, and modify a *Miner* within it.

Walkthrough

When the user runs the game, they initially see the main menu open displaying 4 buttons labeled “New Game”, “Load Game”, “Options”, and “Exit” respectively. From there, a new user may click the button labeled “Options” to open an options menu. Then they can modify their settings as they like, click the “Exit” button, and return to the Main Menu. After changing their settings if they wish, a new player will click the “New Game” button. As a result, a text box will be displayed that asks for the player’s name. Upon entering the name and clicking the “Enter” button, the user will be prompted to select a map size from 4 preset options by clicking one of the corresponding buttons, each labeled with the number of tiles to be generated. After clicking on their preferred size, they will be presented with the main game grid, with the player spawning in the center. The “w”, “a”, “s”, and “d” keys allow the user to move their character up, left, down, and right, respectively, to move around the map. They can left click on a resource tile to place a mining machine on that tile, which will then start generating Ore resources in the player’s Inventory. They can right click on a non-resource tile to place a smelting machine, which will convert Ores in the Inventory to Ingots.

Evaluation

General Evaluation

Overall, the project has made good progress towards its goal of creating an engaging game that allows players to collect resources, build structures, and interact with a variety of features. The completed essential items, such as the rendering of the grid and playing field graphics and the menu interfaces, provide a solid foundation for the game's functionality. Additionally, the completed non-essential items, such as the improved graphics for the enemy, player, and machine icons, and the working machine harvest and refine system, enhance the game's overall experience.

Essential Items (Completed & Not)

Collection - The ability for the player to collect a resource within the game. (Complete)
Building - Enabling the player to construct structures. (Complete)
Enemies - An adversary that the player must overcome. (InComplete?)
Defense - A way for the player to overcome the enemy. (InComplete?)
Rendering and Graphics - A graphical user interface that is appeasing and allows the user to interact with the games functionality. (Complete)
Persistence - A method for saving player progress within a session and reloading it to continue in a future session. (Complete)

Enhancement Items (Completed & Not)

Progression System - The following categories that make the game more engaging and increase the play time longevity. (InComplete)
Advanced Defense Systems - A highly complex configurable interface for the user to design the way the fire control acts and engages enemies. More defensive missile types which have their strengths and weaknesses against threats. (InComplete)
Advanced Enemy's - More complex that is less predictable and requires more planning and strategy to defend against. (InComplete)

Best Work

The implementation of graphics in our game through the GridRenderer class and the TextureEngine demonstrate our team's best work. Our graphics implementation showcases our ability to combine innovative design, technical expertise, and attention to detail. By blending sampled and constructive graphics, we achieved a visually rich and well-running environment that allows for seamless gameplay. The well-structured and organized code in the GridRenderer class highlights our

commitment to adhering to best practices, and facilitating future modifications and expansion to the game's visual elements. This culmination of work and technical prowess exemplifies the standards our team has set for itself.

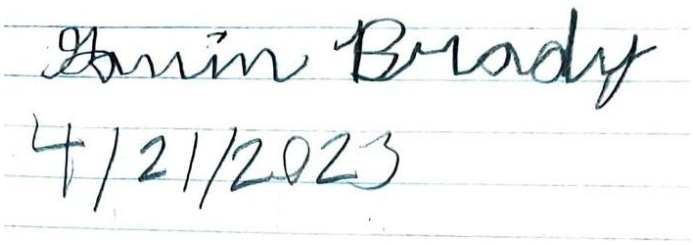
Improving and Expanding

The game currently has a number of essential items that are completed, such as the rendering of the grid and playing field graphics, data entry by the user, and menu interfaces for interacting with the program. However, there are still some essential items that are incomplete, such as the persistence of data across restarting the application.

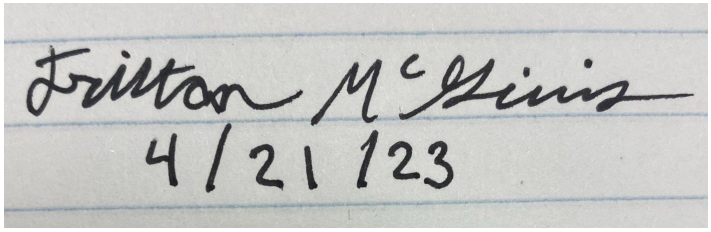
In terms of additional features, there are a number of enhancement items that have been completed, such as improved graphics for the enemy, player, and machine icons, as well as a working machine harvest and refine system. However, there are still several non-essential items that are incomplete, such as adding enemy forces to attack the player, implementing an item shop system, and creating a defense system against enemies.

Other potential enhancements that could be considered include adding more sound effects and music, creating a working escape system, and further improving the graphics of the game. By completing these items and continuously expanding and improving the game, it can become more engaging and enjoyable for players.

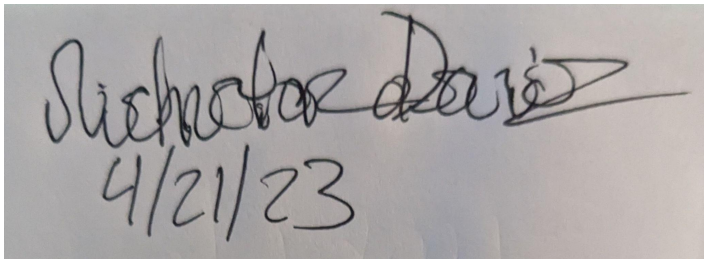
Submission Notice



Gavin Brady



Tristan McGinnis



Nicholas Davis

Jacob Neel - 4/21/2023