Advanced programming project report

This year's project involves the design and implementation of an interactive game inspired by “Meat Boy” using the C++ programming language. The purpose of this report is to document the structure and design of the project.

All features and technical requirements from the assignment were implemented. The libraries used for this project include SFML 2.5.1, which is dynamically linked, and nlohmann-json, which is statically linked and included as a git submodule.

Project structure

The game consists of 3 libraries, engine, game and renderer. Engine and game correspond to “core” / “logic” in the assignment and renderer corresponds to “Game” / “game representation” in the assignment. This allows for a clear separation between the main parts of the project and makes it easy to reuse or extend it.

The engine library contains generic and reusable elements that the game library builds upon. It includes the Engine class, an interface for the Game class, and an API for the renderer library. The library also includes an entity system and corresponding components, abstract factory interfaces for creating components, a state design pattern interface, a singleton Stopwatch class for managing the framerate, a Camera class for projecting coordinates, a Vector2f class for representing 2D points/vectors, and math utilities.

The game library uses the engine library to implement the actual game logic. It includes game states, menu and world states, as well as concrete game entities and their states. World states contain the entities and gameplay logic.

The renderer library uses the SFML graphics library to manage the window, create and render the view of the game logic, and handle user events and input. It contains an instance of the game and uses its API to simulate, update, process input, and sleep in the main game loop. The renderer also implements the concrete components and component creators from the engine library and passes them to the game instance.

Entity system

The game uses an entity component system to represent game world objects. This design pattern uses composition over inheritance, allowing entities to span multiple domains without coupling them together. Entities contain a transform (position, scale, and rotation) and a list of components, which are updated by the entity during an update cycle. This enables the easy creation of new entities with behavior defined by their components.

The game includes physics, view, and audio components. The view and audio components are interfaces, and the corresponding concrete components are created in the renderer using an abstract factory design pattern. The observer pattern is used to update the concrete components when the interfaces are updated, with inheritance rather than observer pointers for added control.

The UIEntity class inherits from Entity and uses the composite design pattern to implement uniform UI entities. These entities contain a parent pointer and a list of child pointers, allowing them to manage the state of their children recursively and place UI elements relative to their parent.

Resource loading

During startup, the game loads game resources into memory using the ResourceManager class. This class is an interface that is implemented in the renderer library. Game resources are loaded from resource configuration files that specify the paths to the game assets. After that, the game can create components with the appropriate component creator (abstract factory), which then creates the concrete component with a pointer to a game resource.

Input handling system

The user input is handled on three different layers, hardware, engine and game events. The hardware inputs are processed through SFML in the renderer library and mapped to engine inputs. Those engine inputs are passed to the game, which are then mapped, depending on the key bindings in the config, to game input events. Those input events are passed to game state which then handles them or passes them to their entities.

The input handling system in this project is divided into three layers: hardware, engine, and game events. The SFML library is used to handle inputs from the keyboard and mouse at the hardware level, which are then mapped to engine inputs using the InputMapper class in the renderer library.

Those engine inputs are passed to the Game class and mapped to specific game events depending on the key binding settings in the config. Finally, the Game state classes handle the game input events and control the game logic based on them. This system allows for a flexible and intuitive response to user input.

Game state

The game transitions between the menus and the world states are modelled using a state machine (pushdown automata, which allows for overlay menu's and paused states) and they transition between their states depending on user input and button states.

The Player and Button classes are also modelled using a state machine and they transition between their states depending on the user input and internal conditions/state.

The state design pattern is generically implemented using a templated class.

The game's state machine is implemented using a pushdown automaton, which allows for overlay menus and paused states. The game transitions between menus and world states based on user input and button states.

The Player and Button classes are also modelled using a finite state machine, where they transition between states depending on user input and internal conditions/state.

The state design pattern is implemented using a templated class, which allows for more reusability and ease of use.

Gameplay

The game world is represented by a World State class, which contains all the entities and handles game control and collisions between entities.

During startup, the Game class loads level data from a configuration file and creates an intermediate level data structure. The Level State class inherits from World State, and it creates the entities

specified in the level data structure. The game progresses to the next available level when the player collides with the finish object.

The layout of the game levels is loaded from a custom file format. In the “meta” header the metadata for the level gets loaded with origin, level boundary and camera data. In the “entities” header the entities are parsed with their entity type (player, finish, wall and tile), a position, size and extra entity metadata.

The levels were designed with a map editor called Tiled and then converted to the custom file format using a script written in Python. This allows for more efficient and convenient level design, by easily editing the maps with a visual editor, and then converting it to the format that the game can read.

References

Assets used:

- Character: https://rvros.itch.io/animated-pixel-hero
- Terrain and background: https://oisougabo.itch.io/free-platformer-16x16
- Finish object: https://opengameart.org/content/spinning-glassy-shapes

Tiled map editor: https://www.mapeditor.org/

Other references, algorithms, design patterns, etc.:

- Physics integration basics: https://gafferongames.com/post/integration_basics/
- Decoupled physics time step integration: https://gafferongames.com/post/fix_your_timestep/
- Accurate sleep function on windows: https://blog.bearcats.nl/accurate-sleep-function/
- Entity component system: https://gameprogrammingpatterns.com/component.html
- Input handling: https://www.gamedev.net/blogs/entry/2250186-designing-a-robust-input-handling-system-for-games/
- Line segment intersection: https://stackoverflow.com/a/565282/12557703
- State design pattern: https://gameprogrammingpatterns.com/state.html
- Templated state pattern: https://codereview.stackexchange.com/questions/40686/state-pattern-c-template