Technical Design Document – Outline

1 Title Page

Berzerk - A Reproduction of Classic Game

2 Document History

Version	Date	Author(s)	Changes
0.1	2024-03-08	Aurelio Rodrigues	
0.2	April 12, 2024	Aurelio Rodrigues	

3 Table of Contents

Table of Contents

4 Game Summary

This is my version of the classic arcade game Berzerk, launched for Atari in the early 80's. The game brings a fun twist to the classics. Here, the player is a soldier with a special dragon spear to fight enemies. These bad guys also have dragon spears and will chase the player if they get too close.

The player can also pick up items that make them move faster. The goal is to get through mazes, beat the enemies, and grab items to go to the next level.

5 Development Environment

5.1 Development Hardware

PC running Windows 10 or 11.

5.2 Programming Languages

C++23 with SFML 2.6.1.

5.3 Development Tools

Visual Studio.

5.4 External Code

SFML (https://www.sfml-dev.org)

5.5 Game Engine

GEX game engine developed in prog1266. This is an Entity-Component-Systems based game engine.

6 Architectural Analysis

6.1 Classes

Describe the classes that will have to be implemented. For each class, provide:

- Its responsibilities
- How it collaborates with other classes

Class	Responsibilities	Collaborations
Animation.h	• Manage and control frame-by-frame animation for sprites. • Handle the timing and sequence of frames within an animation	sf::Sprite sf::Texture sf::Time

	cycle. • Control the playback of the animation (start, stop, and determine if it has ended). • Provide access to the associated sprite and its bounding box dimensions.	
Assets.h	• Manage and centralize access to game assets such as fonts, textures, sound effects, animations, and sprite definitions. • Load assets from files based on configuration or specific paths.	sf::Font sf::Texture sf::SoundBuffer Animation sf::Sprite
Command.h	• Encapsulate action details within the game, providing a uniform approach to handle user inputs or game actions. • Store identifiers such as the name and type of the command, which can be used to determine the action to be executed.	GameEngine or Controller Classes Event Handling System
Components.h	The Component system in the game architecture utilizes a classic entity-component design pattern to promote modularity and flexibility in game object behavior and appearance. Each component represents a distinct aspect of functionality that can be attached to game entities.	CAnimation CSprite CTransform CBoundingBox CState CInput CPowerUps CScript
Entity.h	• Represent individual objects within the game world, each possessing various behaviors and properties as defined by their components. • Manage lifecycle events of the entity such as creation, activation,	EntityManager Components (CSprite, CAnimation, etc.)

	deactivation, and destruction.	
EntityManager.h	• Manage the lifecycle of all Entity instances within the game. • Organize entities by tags for efficient categorization and retrieval.	Entity EntityVec and EntityMap
GameEngine.h	• Centralize and manage the core game loop, including initialization, rendering, and updates. • Handle scene transitions, maintaining a map of all available game scenes and managing the active scene.	Scene Assets sf::RenderWindow sf::Text and sf::Time
MusicPlayer.h	• Manage background music for the game. • Control playback of music tracks, including play, stop, and pause functionalities. • Adjust music volume and manage different music tracks loaded from files.	sf::Music
Physics.h	• Provide core physics calculations related to entity interactions within the game, specifically focusing on collision detection. • Determine the overlapping area between two entities, which is crucial for resolving collisions and interactions in gameplay.	Entity sf::Vector2f
Scene.h	• Serve as the base class for different scenes in the game, such as menus, gameplay levels, and game over screens. • Manage entities within the scene using	GameEngine EntityManager Command

	EntityManager.	
Scene_Berzerk.h	• Manage gameplay mechanics specific to the "Berzerk" game mode, including player and enemy behaviors, scene dynamics, and game rules. • Control movement, animations, collisions, and entity spawning within the game.	EntityManager GameEngine Entity Scene
Scene_Credits.h	• Manage the display of credits for the game, including scrolling text and background images. • Handle transitions between different background images to enhance the visual appeal of the credits scene.	GameEngine Scene sf::Text, sf::Font, sf::Sprite, sf::Texture
Scene_Menu.h	• Manage the main menu interface of the game, allowing players to navigate through options such as starting the game, viewing credits, or exiting the game. • Handle the graphical presentation of the menu, including background animations and text display.	GameEngine Scene sf::Text, sf::Font, sf::Sprite, sf::Texture
SoundPlayer.h	• Manage playback of sound effects within the game, including starting, stopping, and positioning sounds based on game events and interactions. • Maintain a pool of sound objects to ensure efficient sound management and playback.	sf::Sound sf::SoundBuffer sf::Vector2f
Utilities.h	• Provide utility functions related to mathematical	SFML Classes (sf::Vector2f, sf::Rect, etc.)

	operations, vector manipulations, and graphical object manipulations in SFML. • Normalize vectors, calculate bearings, and convert between radians and degrees.	

6.2 Game Loop

1. **Process Input**

Handle Events: The game processes user inputs or other event triggers. This includes keyboard presses, mouse clicks, or other forms of user interaction. In the context of your setup, this would be handled by the sUserInput() function in the GameEngine, which checks for actions mapped in Scene_Berzerk and other scenes.

2. **Update Game State**

Update Scene: Each active scene updates its internal state. This involves moving characters, running AI routines, checking for collisions, and possibly triggering events based on interactions or timers.

Movement and Physics Calculations: The sMovement() function in Scene_Berzerk would handle the movement of entities based on input and game logic.

Handle Collisions: The sCollisions() function would be called to handle interactions between entities, such as collisions between the player and enemies or obstacles.

Custom Updates: Other custom updates per scene, such as spawning or removing entities, adjusting scores, etc., are also handled during this phase.

3. **Render**

Draw Graphics: The current state of the game is drawn to the screen. This usually involves clearing the previous frame's graphics and then drawing all game elements in their current state.

Render Scene: The sRender() function in each Scene subclass, such as Scene_Berzerk, would take responsibility for drawing all visual elements to the window managed by GameEngine.

4. **Post-Rendering Operations**

Post-Update Clean-Up: After rendering, the game might clean up or reset certain states, remove spent entities, or handle memory management tasks.

Remove Stopped Sounds: Sound effects that have finished playing might be cleaned up here as part of managing resources efficiently.

5. **Synchronize and Sleep**

Timing Adjustments: The game engine might delay the next loop iteration to keep the game running at a consistent framerate.

Frame Rate Management: The game adjusts the timing to ensure that each loop iteration aligns with the desired frame rate, preventing the game from running too fast on high-performance hardware.

6. **Check for Exit Conditions**

Loop Termination: The loop checks if there are any conditions that require the game to exit, such as the player closing the game window or pressing an exit key. This can be handled by monitoring a flag within GameEngine that indicates whether the game should continue running.

7 Technical Risks

Risk	Severity	Mitigation (what is to be done to eliminate or minimize this risk)
Performance issues due to high entity count	High	Implement and optimize entity management strategies. Use profiling tools to identify bottlenecks and refactor to use more efficient data structures and algorithms. Consider using an Entity Component System (ECS) for better performance.
Delays in content creation (art, sound)	Medium	Establish clear workflows for content creation. Use placeholders to keep development moving. Consider contracting freelancers if internal capacity is exceeded.
Difficulty in balancing game mechanics	Medium	Use data-driven design to allow easy adjustments of game parameters. Implement logging and collect feedback to identify balance issues. Schedule regular playtesting sessions throughout development.