**Defenders of Ragnar**

**Technical Design Document**

**A Game by Marc-Andre Michaud**

**Student, Gaming Experience Development Sr**

**PROG1189: Comprehensive Development Project**

**New Brunswick Community College**

**School of Information Technology and Natural Resources**

**April 14th, 2022**

**Made using the SFML Game Development game engine**

# Document History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author(s) | Changes |
| 0.1 | Feb 2022 | Marc-Andre Michaud | First Draft |
| 0.2 | Mar 2022 | Marc-Andre Michaud | Second Draft |
| 1.0 | Apr 2022 | Marc-Andre Michaud | Final Revision |

# Table of Contents

Table of Contents

[1 Document History 2](#_Toc96329609)

[2 Table of Contents 3](#_Toc96329610)

[3 Game Summary 4](#_Toc96329611)

[4 Development Environment 4](#_Toc96329612)

[4.1 Development Hardware 4](#_Toc96329613)

[4.2 Development Tools 4](#_Toc96329614)

[4.3 External Code 4](#_Toc96329615)

[4.4 Game Engine 4](#_Toc96329616)

[5 Architectural Analysis 5](#_Toc96329617)

[5.1 Classes 5](#_Toc96329618)

[5.2 Class Diagrams 8](#_Toc96329619)

[5.3 Behavioral Analysis 9](#_Toc96329620)

[5.3.1 Game States 9](#_Toc96329621)

[5.3.2 Lane Controller States 9](#_Toc96329622)

[5.3.3 Tower States 10](#_Toc96329623)

[5.4 Game Loop 10](#_Toc96329624)

[6 Technical Risks 11](#_Toc96329625)

# Game Summary

Defenders of Ragnar is a medieval fantasy themed Tower Defence game, roughly based on UO:Defender.

The game uses a top-down isometric tile-based view with 2D animated sprites for towers and enemies. The mouse is the primary input method, with a few keyboard shortcuts.

The game's main objective is placing "towers" that attack enemies coming down a path to stop them from coming through. Success is achieved by strategically placing towers, and the game is lost by having too many enemies make it across the game area.

# Development Environment

## Development Hardware

This game was developed on MS Windows 10 using SFML 2.5.1 and C++17. Python 3.8 was used for texture and JSON creation. JSON was used as texture metadata.

## Development Tools

* Microsoft Visual Studio and C++ compiler was used as the IDE and compiler.
* Git was used for Version control
* Vim was used for text editing
* Python was used for running scripts
* Gimp was used for drawing graphics
* Texture Packer was used to pack textures

## External Code

* SFML 2.5.1 - used to make the game engine <https://www.sfml-dev.org>
* JSON for Modern C++ 3.10.5 - used to parse JSON <https://github.com/nlohmann/json>
* Pillow 9.0.0 - used for texture manipulation <https://pillow.readthedocs.io/en/stable/>

## Game Engine

The game code is directly integrated with the game engine with no strict demarcation. However, it would be safe to assume that the application, state stack, scene graph, and scene nodes are part of the engine.

The game engine provides and manages a stack of “states” representing different modes of the game. Whereas the scene graph provides a graph of objects that can be updated and drawn in layers.

# Architectural Analysis

## Classes

|  |  |
| --- | --- |
| Class | Responsibilities |
| SceneNode : sf::transformable, sf::drawable | Nodes of the Scene Graph. Nodes may have child nodes and a parent node.  During the game loop, update and draw are called on the top node in the graph. The update and draw calls propagate to every node down the graph.  Child nodes are managed through unique pointers, thus a node and all its children can be deleted by eliminating the unique pointer.  A SceneNode is also a sf::transformable and a sf::drawable so it the ability to be drawn and transformed through SFML. |
| Enemy : SceneNode | Represents enemies in the game. By default they are always moving in a given direction.  Enemies are generated by Lanes and can be targeted through the LaneController.  Enemy health, movement speed, and sprite is determined by EnemyData provided to the constructor.  Enemies use Animation2’s to draw their walk animation and a Direction vector to dictate the route to be taken through the level. |
| Tower : SceneNode | Represents warriors and wizards that act as the “towers” in this tower defence game.  Towers are generated by the World class and are placed on valid tiles on the game board.  Tower damage, attack speed, sprite, range, and special attacks are determined by TowerData provided to the constructor.  Towers have an extra sprite to display tower range and status.  Towers use Animation2’s to draw their attack animation. |

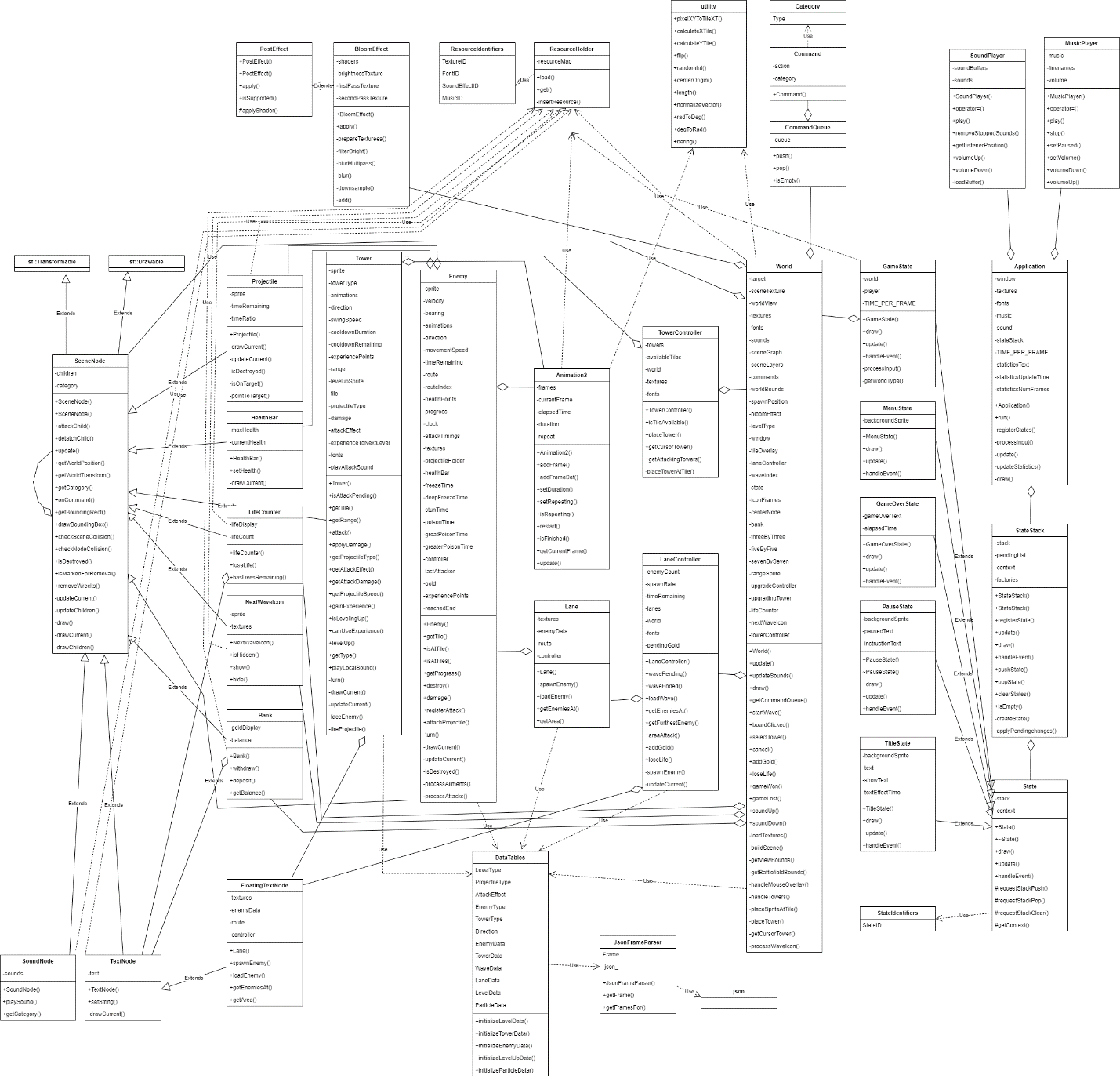
|  |  |
| --- | --- |
| Class | Responsibilities |
| SoundNode : SceneNode | A node in the scene graph that generates sounds. |
| Animation2 | A class that manages textures for animated sprites. Can calculate the appropriate animation frame based on time passed. Returns the sprite’s position on the texture sheet, the x and y offsets, and whether the texture was rotated or not. |
| Projectile : SceneNode | Projectiles are created by enemies once targeted by a projectile attack.  The projectile is attached as a child of the Enemy node, thus it’s 0, 0 coordinate is centered on the enemy. The projectile will point towards and travel to 0, 0 and destroy itself once it reached the target.  The projectile is purely a visual effect. It has no impact on targeting, damage, or any other effect on the game. |
| TextNode : SceneNode | A sprite like object that displays text. |
| FloatingTextNode : TextNode | A text node that floats up and gradually fades away. |
| Command | Command object used to send commands to objects in sceneGraph. |
| CommandQueue | Queue of commands that are sent to all nodes in the sceneGraph during the update cycle. |
| PostEffect | Shader effects. |
| BloomEffect : PostEffect | Bloom effect. |
| ResourceHolder | Template class to load and manage resources like textures, texture atlases, sound effects, fonts, etc. |
| Application | Initialize the game and runs the game loop. |
| Lane | An enemy management class. Each lane has a set defined path for spawned enemies to follow. |
| LaneController | A lane management class. Gets passed LaneData to generate lanes. Later gets passed WaveData to cause lanes to spawn a wave of enemies. |
| State | A game state/screen/scene. |
| TitleState : State | Game Title or splash screen. |
| MenuState : State | Game Menu. |
| PauseState : State | Pause screen . |
| GameState : State | The Game. |
| World | The game world, all initial game objects are created and put in the sceneGraph, collisions are handled, and the sceneGraph is updated and drawn in the game loop. |
| StateStack | Holds and manages game states. |
| Particle | A particle for use in a particle system, details are looked up in the data tables. |
| ParticleNode : SceneNode | A particle system, owns, updates and draws all particles in the particle system. |
| EmitterNode: SceneNode | Emits particles into a particle system . |
| Bank : SceneNode | Displays and keeps track of the player’s gold. |
| HealthBar : SceneNode | Displays and keeps track of an enemy’s health. Gets attached as a child of an Enemy once the Enemy is targeted. |
| Class | Responsibilities |
| LifeCounter : SceneNode | Displays and keeps track of the player’s remaining lives. |
| NextWaveIcon : SceneNode | Displays an image of the Enemy type of the next wave. |
| UpgradeController : SceneNode | Displays and manages the upgrade menu that appears when upgrading a Tower. |
| TowerController : SceneNode | A tower management class. Manages communication between the World class and Towers. |

DataTables:

Data used by classes in the game are generated in the DataTables functions. This data comes packaged in Data structs that get passed to classes on creation to provide all data required. Some data is read from JSON while some is hardcoded in C++.

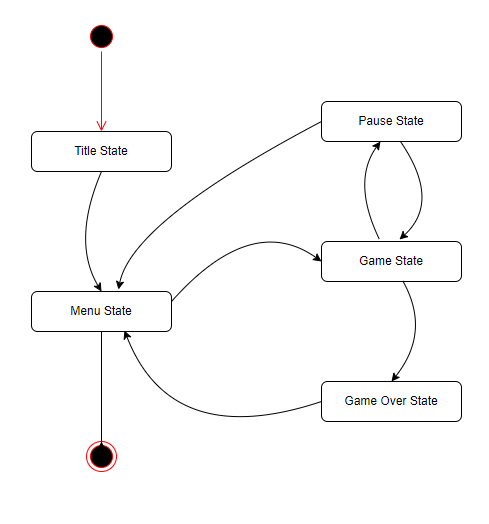
|  |  |
| --- | --- |
| Data | Responsibilities |
| EnemyData | Contains a given enemy’s texture, animations, walk speed, and health |
| WaveData | Contains an EnemyData, enemy count, and spawn rate for a wave of enemies. |
| LaneData | Contains a Lane’s initial position and route. |
| LevelData | Contains the level’s texture, LaneData, and WaveData |
| ParticleData | Particle type and decay rate |
| TowerTileData | A list of buildable tiles for each level |

## Class Diagrams

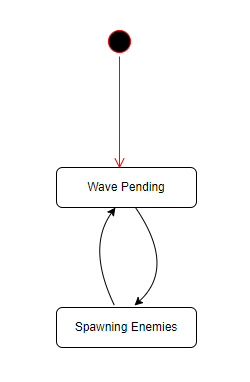


## Behavioral Analysis

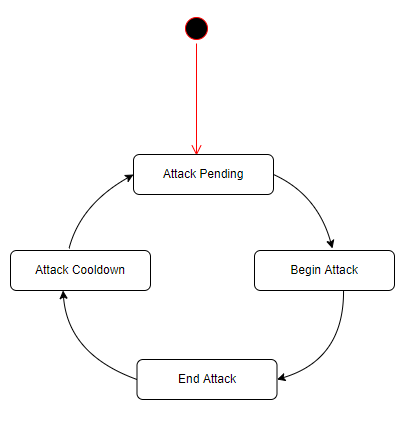
### Game States



### Lane Controller States



### Tower States



## Game Loop

The game loop begins in the Menu State. The state is switched to the Game State when the player clicks on a level icon. The Game State creates a World object with the level type selected by the user. Based on the level type, the World retrieves the appropriate LevelData used to create the Lanes and prepare the WaveData.

During the Game State, the player can pause the game, entering the Pause State. From the Pause State the player can resume the game and return to the Game State, or quit and return to the Menu State.

While playing the game, winning, or losing will send the player to the Game Over State. Which itself will move the player over to the Menu State.  
  
From the Menu State the player can exit the game.

# Technical Risks

* Adapting the engine to effectively use the mouse for input
* Optimizing for the vast amount of animation frames
* Optimizing Tower target acquisition

|  |  |  |
| --- | --- | --- |
| Risk | Severity | Mitigation (what is to be done to eliminate or minimize this risk) |
| Optimizing the game engine for mouse inputs | mild | Using reasonable stop-gap measures. “Premature optimization is the root of all evil” |
| Optimizing for the vast amounts of animation frames | high | This problem will be tackled first. |
| Optimizing Tower target acquisition | medium | Tackle this second. Avoid scanning through all enemies for all towers every frame. |