Space Escape

Technical Design Document

Developer: Abigail Sutton

Date: 16/5/25

Version: 2.0

Table of Contents

[Introduction 3](#_Toc196824028)

[Technical Overview 3](#_Toc196824029)

[Logic and Technical Algorithms: 3](#_Toc196824030)

[Technical Design Diagrams: 6](#_Toc196824031)

[UML Class Diagram: 9](#_Toc196824032)

[Debug Features: 9](#_Toc196824033)

[Coding Standards 10](#_Toc196824034)

[Relevant File Formats 12](#_Toc196824035)

[Acceptance Test Plan 12](#_Toc196824036)

[Bibliography: 14](#_Toc196824037)

## Introduction

This Technical Design Document (TDD) outlines the technical implementation for the retro styled, pixel art, adventure game, “Space Escape”. The document will serve as a guide for working on the project, detailing the algorithms, systems, architecture, and technical specifications that will be required to implement the game as described in the Game Design Document.

## Technical Overview

Space Escape will be developed using C++ on Visual Studio with additional libraries including FMOD, SDL2, Glew, OpenGL, and ImGui.

Core systems include:

* Game state management
* Procedural planet generation
* Combat system
* Item and inventory management
* Player progression tracking
* Environmental effects system
* User interface
* Debug system

## Logic and Technical Algorithms:

Enemy spawning algorithm:

This algorithm is to spawn and keep track of the types of enemies spawned currently. It takes a base enemy spawn count and adjusts that based on the planet level. The planet level is also used to determine the difficulty of enemies spawned.

Pseudo code:

*baseEnemyCount = 5*

*additionalEnemies = planetLevel \* 2*

*totalEnemies = baseEnemyCount + additionalEnemies*

*if planetLevel < 3*

*enemyTypes = “Basic”*

*else if planetLevel < 5*

*enemyTypes = “Advanced”*

*else*

*enemyTypes = "Elite”*

*for int = 0 to totalEnemies*

*x = random(0, width)*

*y = random(0, height)*

*spawnEnemy(x, y)*

Boss system:

This algorithm calculates the chance of a special weapon being dropped by a boss. Each type of boss has its own percentage for chance and all weapons dropped are random.

Pseudo code:

*weaponDropChance = 0.05%*

*if bossType == “mini”*

*weaponDropChance \*= 0.5*

*else if bossType == “final”*

*weaponDropChance = 1.0*

*if random(0, 1) < weaponDropChance*

*availableWeapons = getAvailableWeapons(player)*

*dropRandomWeapon(availableWeapons)*

Planet effect system:

This algorithm is to assign a random planet effect to the level the player is currently on. There are several different effects, and one is chosen at random for the level, except for the first one.

Pseudo code:

*effects = {*

*“reverseControls”, inverts the players movement controls*

*“slowTime”, reduces the game speeds by 30%*

*“freezeTime”, freezes time randomly for a few seconds*

*“darkness”, reduced visibility*

*“toxic”, health slowly drains*

*“antiGun” melee weapon only*

*}*

*planetEffect = randomEffect(effects)*

Life tracking system:

This algorithm tracks the number of lives the player currently has, and restarts the game based on whether all lives lost.

Pseudo code:

*playerLives -= 1*

*if playerLives > 0*

*resetHealth()*

*else*

*displayGameOverScreen()*

*resetGameProgress()*

Rift vial collection logic:

This algorithm checks if the vial can be used based on the number of enemies left and ship parts collected.

Pseudo code:

*currentPlanet.vialCollected = true*

*player.inventory.riftVials += 1*

*if currentPlanet.enemiesRemaining == 0 and currentPlanet.shipPartCollected*

*unlockNextPlanet = true*

Player Spin algorithm:

This algorithm calculates points on a Fibonacci spiral to animated the player in the black hole.

GenerateSpiral(points, centerX, centerY)

fibonacciPoints = empty list

dx = playerPositionX – centerX

dy = playerPositionY – centerY

startDistance = square root of (dx^2 + dy^2)

For i from 0 to points

Progress = i / points

Angle = progress \*10pi

Radius = startDistance \* (1 – 0.9 \* progress)

pointX = centerX + radius \* cos(angle)

pointY = centerY + radius \* sin(angle)

if i == 0 then

playerPosition = point

fibonacciPoints.add(point)

fibonacciPoints.last = centerX, centerY

AnimateSwirl()

If fibonacciPoints is empty then return

EasedProgress = progress^2 \* (3 – 2 \* progress)

pointIndex = easedProgress \* (total points – 1)

pointA = floor(segment)

if (pointA + 1 < fibonacciPoints count – 1 then

pointB = pointA + 1

else

pointB = fibonacciPoints count – 1

t = segment – pointA

positionX = linear(fibonacciPoints[pointA]X, fibonacciPoints[pointB]X ,t)

positionY = linear(fibonacciPoints[pointA]Y, fibonacciPoints[pointB]Y, t)

playerPosition = position;

if pointB < fibonacciPoints count – 1 then

direction = fibonacciPoints[pointB] – fibonacciPoints[pointA\

angle = arctan2(directionY, directionX)

player.setRotation(angle)

scale – 5.0 \* (1 – swirlProgrss)

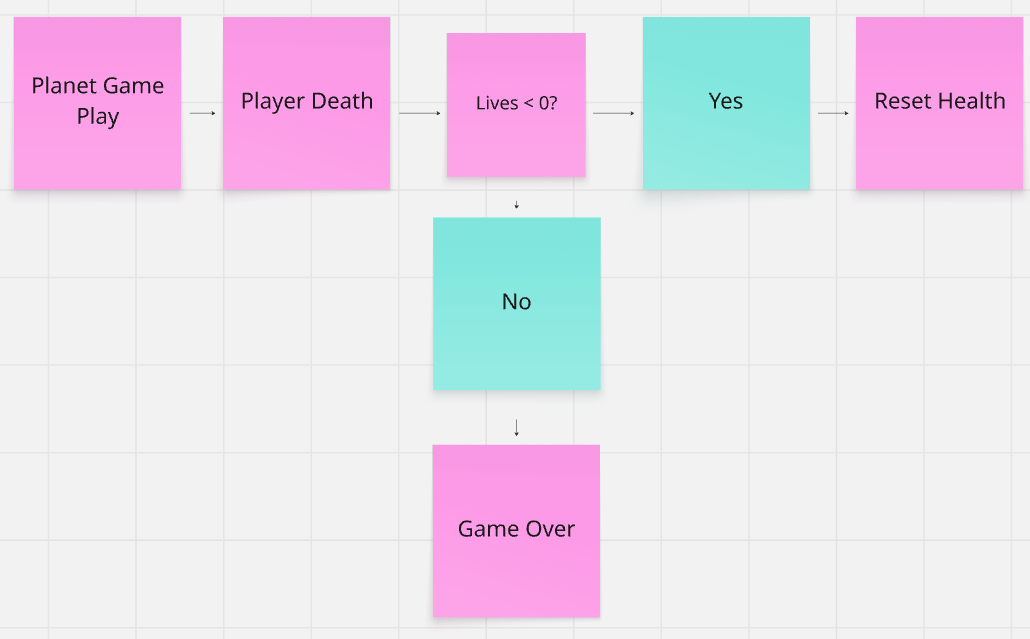
player.setScale(scale)

## Technical Design Diagrams:

Game state flow diagram:

A diagram of a diagram

AI-generated content may be incorrect.

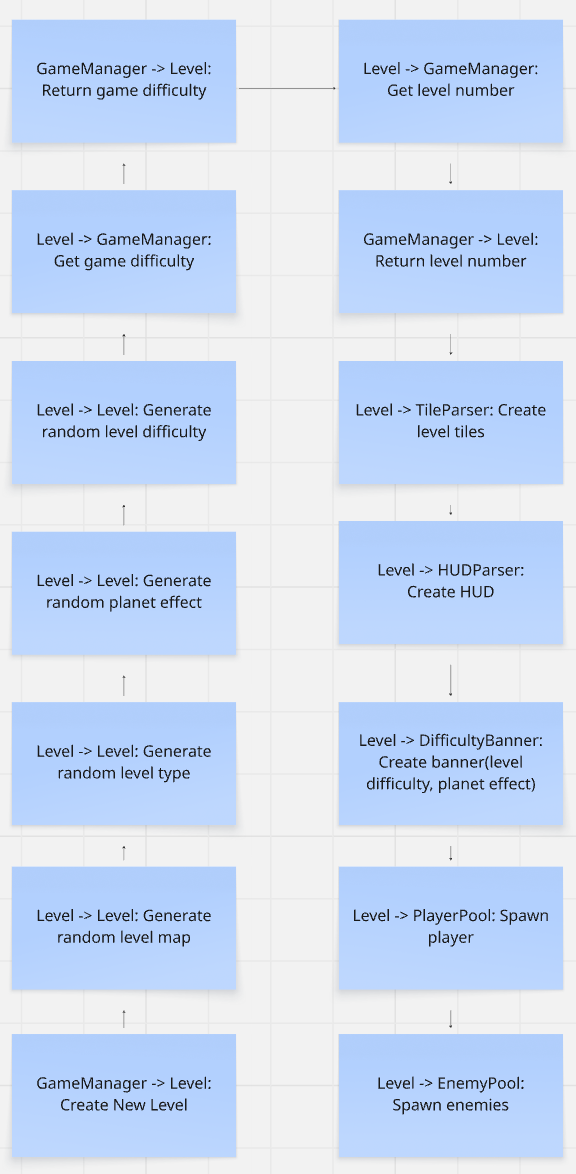


Combat system activity diagram:

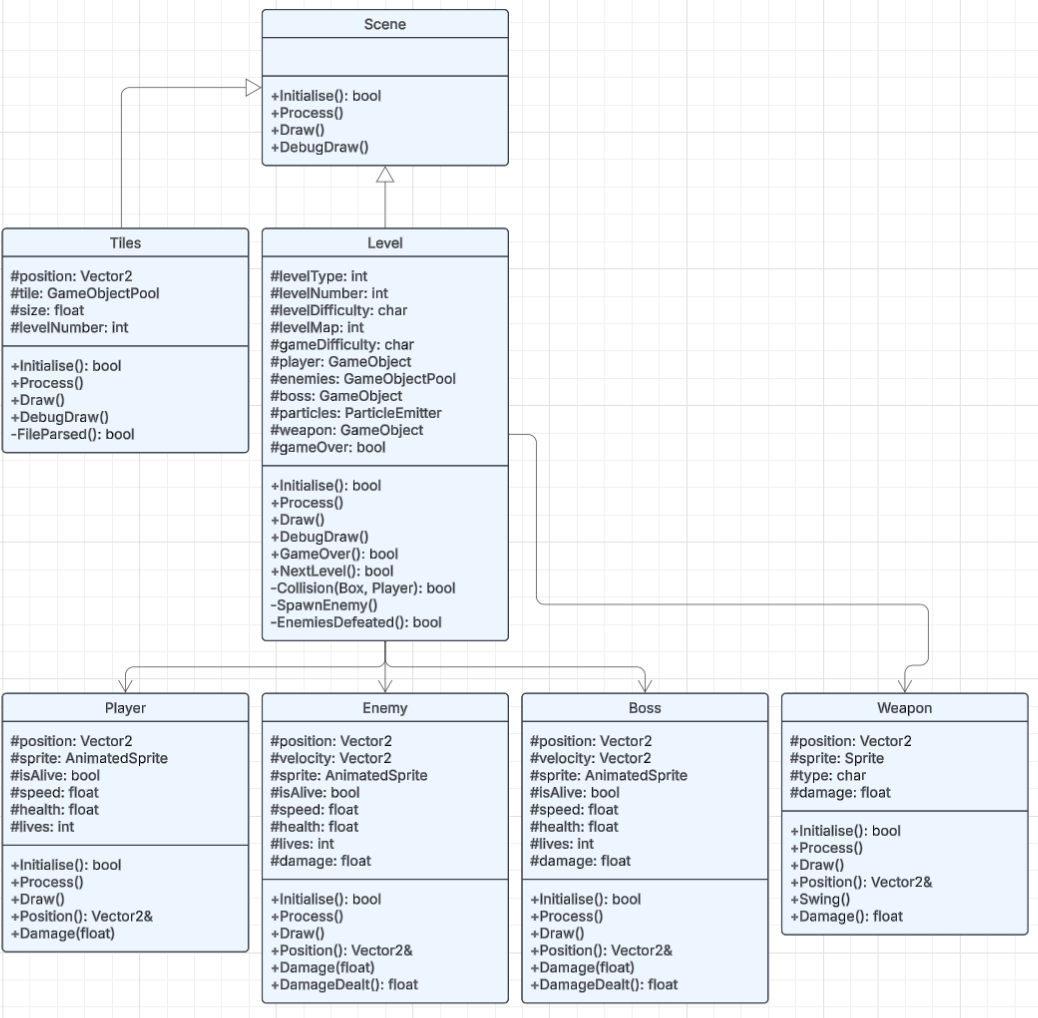
A screenshot of a diagram

AI-generated content may be incorrect.

Planet Generation sequence diagram:



## UML Class Diagram:



## Debug Features:

Space Escape will implement a debug system using ImGui to aid development and testing. The debug features will be accessible through a developer mode activated by a special key – insert.

The debug system will use ImGui to provide and immediate mode graphical user interface for debugging. It will be integrated as follows:

Performance monitoring:

* FPS counter
* Memory usage
* Entity count

Entity debugger:

* List of all active entities with hierarchal view
* Entity spawning and removal tools

Planet generation controls:

* Force specific planet types and layouts
* Adjust enemy count
* Preview and test planet effects

Cheat options:

* Invincibility toggle
* All weapons unlock and selection
* Skip to specific planets
* Add lives or resources

Debug console:

* Log output

## Coding Standards

Naming conventions:

Classes and structs:

* PascalCase
* Nouns or noun phrases

Methods and functions:

* PascalCase
* Verb or verb phrases that describe the action
* Boolean functions should ask a question, e.g. IsAlive()

Variables:

* camelCase
* private member variables should start with m\_
* static variables start with s\_

Constants and enums:

* UPPER\_SNAKE\_CASE
* Enum values should use PascalCase

File naming:

* Match class name
* One class per file

Code structure:

Indentation and Spacing:

* Line length up to 100 characters
* One empty line between methods
* Curly brackets on new lines for class or namespace declarations, same line for methods or loops

Comments:

* Class and public method declarations must have comments
* Use // for single-line comments
* Use /\* \*/ for multi-line comments
* Comment complex algorithms and non-obvious behaviour

File Organization:

* Header files (.h):
  + #ifndef \_NAME\_H\_\_
  + #define \_NAME\_H\_\_
  + #endif // !\_NAME\_H\_\_

Programming Practices:

* Log errors with appropriate detail and severity
* Explicitly delete copy constructors/assignment operators for non-copyable types
* Use object pooling for frequently created/destroyed objects

## Relevant File Formats

Asset formats:

* Images:
  + Sprites: PNG with transparency
  + Naming conventions: name\_name.png
* Audio:
  + Naming conventions: name\_name.WAV
* Fonts:
  + TrueType Font - .ttf

Data formats:

* .ini for game settings
* .txt for level and hud layouts

## Acceptance Test Plan

The following test plan contains ten questions that will validate the completion of Space Escape:

1. Player Movement and Controls:
   1. Does the player respond correctly to all input methods (keyboard, controller) with appropriate movement speed and collision detection?
   2. Acceptance: Player can move in all directions, collides with environment properly, and the control scheme matches the specified layout in the design document.
2. Combat System:
   1. Do both melee and ranged weapons function correctly with appropriate attack animations, damage calculation, and enemy response?
   2. Acceptance: Different weapon types have distinct behaviours, damage is calculated correctly based on weapon type, and hit detection works properly.
3. Enemy AI and spawning:
   1. Do enemies spawn according to the planet-based difficulty progressions and exhibit appropriate AI behaviours?
   2. Acceptance: Each planet has the correct number and types of enemies, enemies think, pursue, and attack the player when in range, and difficulty increases with planet level.
4. Planet Effects System:
   1. Are the unique planetary effects properly applied, and do they affect gameplay as intended?
   2. Acceptance: Each plant correctly displays its effect on entry, the effect modifies gameplay as specified, and effects are removed when leaving the planet.
5. Item Collection and Progression:
   1. Can the player collect rift vials and ship parts with proper inventory updating and progression tracking?
   2. Acceptance: Items can be collected by player interaction, inventory updates visually, and collecting all required items allows progression to the next planet.
6. Life System and Game Over:
   1. Does the life system correctly track player deaths and reset the game appropriately when all lives are lost?
   2. Acceptance: Player starts with three lives, losing a life results in the health resetting, and losing all lives results in complete game reset.
7. Boss Encounters and Rewards:
   1. Do boss encounters function correctly with phase changes, do bosses spawn at the specified rate, do the bosses drop upgraded weapons?
   2. Acceptance: Bosses attack with large enough damage, health tracking works correctly, and item drops occur at the expected frequency.
8. Weapon Upgrade System:
   1. Can players collect weapon upgrades from bosses and does this stay during the current run?
   2. Acceptance: Weapons can be collected from boss drops, upgraded weapons have improved stats, and upgrades stay until the game is completely reset.
9. UI Functionality and Information Display:
   1. Does the UI correctly display player health, collected items, weapon information, and lives remaining?
   2. Acceptance: All UI elements update in real-time, information is clearly visible during gameplay, and planet effect notifications are displayed on entry.
10. Performance and Stability:
    1. Does the game maintain a stable framerate throughout all gameplay scenarios and handle resource loading/unloading properly?
    2. Acceptance: Frame rate remains above 60 FPS on hardware, no memory leaks occur during extended gameplay, and transitions between planets occur smoothly.

## Bibliography: