Dragon Attack

Technical Design Document

GAM150S18

Spring 2018

Team D-Sync

|  |  |  |
| --- | --- | --- |
| **Student Name** | **Official Role** | **Coding Responsibilities** |
| Jacob Lim | Producer | Graphics  A.I Logic  Animation |
| Javon Lee | Technical Director | Framework  Collision  Level Editor |
| Andrew Chong | Lead Designer | Stage Designs  Interface Design |
| William Yoong | Product Manager | A.I Logic  GameState Manager  Input handler |

**Table of Contents**

Contents

[1. Project Architecture ( Overview ) 5](#_Toc506585919)

[1.1 Game Summary 5](#_Toc506585920)

[1.2 Overview 5](#_Toc506585921)

[2.Graphics Implementation 6](#_Toc506585922)

[2.1 Camera 6](#_Toc506585923)

[2.2 Create\_Object 6](#_Toc506585924)

[2.3 Render 6](#_Toc506585925)

[2. 4 Sprite 6](#_Toc506585926)

[2.5 Transform 6](#_Toc506585927)

[3. Collision Implementation 7](#_Toc506585928)

[3.1 Collision component 7](#_Toc506585929)

[3.2 Binary Collision Map 7](#_Toc506585930)

[4. A.I ( Characters) Implementation 8](#_Toc506585931)

[4.1 Characters 8](#_Toc506585932)

[4.1.1 Dragon 8](#_Toc506585933)

[4.1.2 Lancelot 8](#_Toc506585934)

[4.1.3 Merlin 8](#_Toc506585935)

[4.1.4 King Arthur 8](#_Toc506585936)

[5. Coding Methods 9](#_Toc506585937)

[5.1 Global variables: 9](#_Toc506585938)

[5.2 User-Defined Types: 9](#_Toc506585939)

[5.3 Naming Conventions: 9](#_Toc506585940)

[5.4 Formatting: 9](#_Toc506585941)

[5.5 Documentation: 9](#_Toc506585942)

[5.6 Source Control System: 10](#_Toc506585943)

[5.7 Rules: 10](#_Toc506585944)

[6. Debugging 11](#_Toc506585945)

[6.1 In-game console debugger 11](#_Toc506585946)

[6.2 Visual Studios debugger 11](#_Toc506585947)

[7. Tools 12](#_Toc506585948)

[7.1 Visual Studios 2015 12](#_Toc506585949)

[7.2 Tortoise SVN 12](#_Toc506585950)

[7.3 GitHub 12](#_Toc506585951)

[7.4 Photoshop 12](#_Toc506585952)

[7.5 Inkscape 12](#_Toc506585953)

[8. Scripting Languages 12](#_Toc506585954)

[8.1 Level Editor 12](#_Toc506585955)

[9. Technical Risks 13](#_Toc506585956)

[9.1 Boss A.I Behaviour 13](#_Toc506585957)

[9.2 Multiple Stages 13](#_Toc506585958)

[Appendix A: Interface Flow 14](#_Toc506585959)

[Interface Flowchart 14](#_Toc506585960)

[Mockup Menu 15](#_Toc506585961)

[Mockup In-game Level 16](#_Toc506585962)

[Appendix B: Art Requirements 17](#_Toc506585964)

[Appendix C: Audio Requirements 17](#_Toc506585965)

# **1. Project Architecture ( Overview )**

## 1.1 Game Summary

Dragon Attack is a **2D action platformer** where the player plays as a young dragon attacking a castle to gain its treasures. The objective of the game is to defeat the King of the castle and gain his treasures.

## 1.2 Overview

Our project is programmed in C++ as it is a form of Object-Orientated Programming that utilises inheritance and polymorphism which allows us to implement certain functions in a more efficient manner by using Classes and Templates that are not available in C.

The game engine is a time-based engine that expands on the provided Alpha Engine by including our own collision detection, graphics loader and logic implementation for our A.I behaviour.

The game engine is divided into several global components:

* Graphics
  + Utilises Alpha Engine for rendering of graphics, applying Camera movement and transformation of game objects.
* Characters
  + This contains the logic and behaviour implementation for all the characters in our game, inclusive of the Dragon, the 3 bosses and all the mob characters.
* Collision
  + Responsible for detecting in-game collisions. Dynamic collision is checked using Axis-Aligned Bounding Boxes (AABB) while Static collision will be implemented using a Binary collision 2d array, i.e Grid collision.
* GameObject
  + This is the father class where most classes inherit from and contains the necessary information that every in-game object needs.
* GameStateManager (GSM)
  + The game state manager handles the transition from one game state to another by making use of function pointers.
* Input Handler
  + The input handler is done by using the Alpha Engine and assigning them to simplified variable names in a namespace.
* Projectile
  + This is a class that inherits from GameOject and contains additional member functions that determines a projectile’s logic.

# **2.Graphics Implementation**

# 2.1 Camera

The camera component expands on the AE\_Engine which allows the user to make the camera follow game objects as required by the user or stay static.

# 2.2 Create\_Object

This component contains the functions required to create various meshes for GameObjects by making use of the AE Engine.

# 2.3 Render

Render is a class that is a private member of Sprite. It stores the necessary values for rendering a mesh, namely the RGBA and Transparency values. It also contains the member function to render GameObjects to the screen.

# 2. 4 Sprite

Sprite is a class that is a public member of GameObject. It contains Render as well as certain members that is essential for loading an image on screen. Such members include an object’s mesh, texture, U&V values, width and height.

# 2.5 Transform

Transform is a class that is a public member of GameObject. It privately stores the translation, scale and rotation matrixes that is concatenated to a resultant matrix. The class also contains member functions that can access the degree of an object and it’s determinant in the global scale.

# 2.6 Particle System

Particle System is a class that has public functions that simulates environment effects such as gravity, force, and turbulence. It also stores an emitter class object which holds a vector of particles and the various attributes to apply on the particle such as lifetime, speed, and size.

# **3. Collision Implementation**

# 3.1 Collision component

This is a class that is a public member for all GameObjects and contains the necessary members for collision detection. Such members includes an object’s shape, collision flag, minimum and maximum points if it is a rectangle and midpoint and radius if it is a circle object. The class would also contain member functions that would check for dynamic collision between 2 GameObjects.

# 3.2 Binary Collision Map

This is a 2D run-time array that represents a binary map that is separated into grid cells. Every GameObject would check with the cell’s values to determine it can access that cell or not. If a GameObject is not allowed to access it, it would snap back to the centre of the cell it belonged to. This method is to check for collision between static objects.

# **4. A.I ( Characters) Implementation**

# 4.1 Characters

Characters is a class that inherits from the GameObject class and holds further members that are unique to the characters in the game. Such members include a character’s hp, direction and a Boolean that determines if it is moving or not. This class is further inherited by different individual in-game characters for further behaviour-design implementations.  
Below are the classes that inherits the Characters class.

# 4.1.1 Dragon

Dragon is a class that holds members that a Dragon needs and includes various functions that contains the logic behind a Dragon’s behaviour. Some examples of its members include the current power-up it has and its fireball projectile.   
  
The Dragon behaviour is implemented by making use of the Input Handler while the charging of a mega fireball is implemented by its own logic functions.

# 4.1.2 Lancelot

Lancelot is a class that has its own update/render functions.

The behaviour functions of Lancelot includes:

Idling, moving, Buffs, attacking and different states.

# 4.1.3 Merlin

Merlin is a class that has its own update/render functions. It also includes its own projectiles for his special skills, namely rain arrows and energy ball. These projectiles are stored in a run-time array using std::vector.   
The behaviour functions of Merlin includes:

Idling, moving, blinking, and attacking by calling different functions in different states.

# 4.1.4 King Arthur

King Arthur is a class that has its own update/render functions. It also includes a Boolean variable that would indicate its current phase.  
The logic implementation of King Arthur’s special skills is also done in its member functions as well as its behaviour movement.

Some examples include:

Idling , Moving , avoiding obstacles, attacking and all its special skills.

# **5. Coding Methods**

# 5.1 Global variables:

Global variables are to be avoided as much as possible. If it is necessary to employ such variables, an anonymous namespace is to be used to contain the variables to be within a specific file only.

# 5.2 User-Defined Types:

All member functions of class/structs should be declared in the class/struct declaration in the .h files while the definitions are to be done in the .cpp files. This is to allow code readability and enforce encapsulation of code.

Constructors for classes are to be provided and copy constructors, copy assignment operator and destructors are to be provided if allocation of memory is needed in the construction of a class.

# 5.3 Naming Conventions:

Function names should be intuitive to provide the user information on what the function is for.

Art and music resources are to be named according to the stage or object that they’re being used for, e.g. <dragon>.<png>

# 5.4 Formatting:

Follow standard proper indentation of C++ code

# 5.5 Documentation:

Every .cpp file should be accompanied by a .h file

Code to be commented and documented in doxygen format

Functions descriptions are to be done in the header file to allow members to understand the usage of each other’s functions. This is to support encapsulation of code

# 5.6 Source Control System:

Tortoise SVN is to be used simultaneously with GitHub. This is to ensure that we would have a backup in case of scenarios where either of the server were to experience latency problems, especially nearing project datelines.

It is the individual’s responsibility to test their code to be compliable and buildable before uploading

Users are to input a summary of what is being uploaded and a simple description of their code. Further elaborations are to be read from the code comments. This would facilitate any backtracking if necessary.

# 5.7 Rules:

To comment code after it is done before uploading.

Macros/defines to be avoided as much as possible and to be replace using const variables instead.

Test for memory leaks before uploading.

# **6. Debugging**

## 6.1 In-game console debugger

This is to be used for run-time checks, for example for collision checks.

As the interface for hp display may not be numerical, variables like hp should be displayed in the console simultaneously with in-game text objects for debugging and checking.

## 6.2 Visual Studios debugger

Main Debugging tool to be used to check for compilation errors and warnings.

Level 4 warnings are to be set before compilation.

Usage of break points to check for code errors during run-time

Can be used to detect memory leaks

# **7. Tools**

## 7.1 Visual Studios 2015

Main debugging and compilation tool that compiles codes into an executable program.

## 7.2 Tortoise SVN

Main method used for source control system for this project. All new code and project filters are to be updated through committing it via SVN.

## 7.3 GitHub

Secondary method for source control system. Used to check logs in the scenario when code integration faces problems. A summary of the update will be written too for reference when browsing through the logs.

## 7.4 Photoshop

Main software used for graphics editing needed for our game.

## 7.5 Inkscape

Secondary software for graphics editing and can be used to create our pixel art and animations needed in our game

# **8. Scripting Languages**

## 8.1 Level Editor

A level editor will be a form of scripting language that is able to allow us to load the map data from a .txt file and update the map data and binary collision array from the .txt file.

# **9. Technical Risks**

## 9.1 Boss A.I Behaviour

Since a single boss has multiple behaviours in their different phases as well, it might be relatively expensive to include them all in the update function. A large amount of memory is also needed as different bosses utilises their own variables.   
Bosses behaviours might not be behave specifically in the way we want as well as there is no fool proof method for behaviour programming.  
  
Mitigation: A more effective way must be employed to generate the GameObjects to ensure that unnecessary AIs would not be generated when not in use. The number of AIs per stage should be limited too if the memory allocation is too high.   
  
Mitigation: Simpler coding methods can be employed or help should be employed from Teacher Assistants (TAs) if the technical complexity is too high.

## 9.2 Multiple Stages

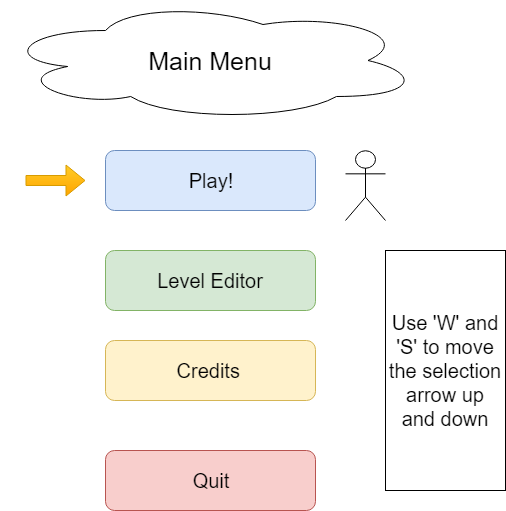
Having multiple stages that is implemented differently can lead to high technical complexity as some stages would require different logics. It would also be difficult to create all the game objects at different positions all the time.

Mitigation: A level editor will be coded first to ensure the implementation of game objects to be easy and intuitive. A stage\_manager would also be used if each stages uses very different functions.

# **Appendix A: Interface Flow**

## Interface Flowchart

## Mockup Menu



This is a simple main menu mock up with intuitive titles for the player to choose his course of actions.

Text Objects will be use to display all these.

## Mockup In-game Level

## 

The Green dragon (in this mock up) represents the player

A dragon symbol will be on the top left-hand corner of the screen to indicate the player’s health

The symbols will ‘disappear’ if a player loses their health

The map of each stage would be bigger than what is visible to the player, therefore there would be a moving camera that follows the dragon as it moves.

# Appendix B: Art Requirements

All images are to be in .png format as it has higher quality, supports transparency values and supports lossless data compression. The naming convention should be followed and be intuitive for what each asset is used for.

All art assets are to be produced by members of the development team.

# Appendix C: Audio Requirements

All audio assets are to be in .ogg format as it is more memory space efficient. Similarly, the audio assets are to follow the naming convention and be intuitive for what each asset is used for.   
Repetition of audio assets shall be done via efficient coding.

All audio assets are to be obtained from the DigiPen Libraries, and produced by members of the development team if needed.

The external library we will be using will be Fmod.