UNIVERSITY OF BIRMINGHAM

University of Birmingham

Final year project

Procedural Generation of Computer Game Content

Author:
James Moulang

Supervisor:
Hayo Thielecke

April 10, 2015

Declaration

The material contained within this thesis has not previously been submitted for a degree at the University of Birmingham or any other university. The research reported within this thesis has been conducted by the author unless indicated otherwise.

Signed James Moulang

Acknowledgements

I would like to thank my family for their constant support throughout my university career, everyone who participated in user testing, and my supervisor Hayo.

Abstract

This report outlines the development of an arcade-style computer game that is appropriate to a wide range of individual players with different skill levels. To this end, a metric of the player's skill level is calculated. Game content is generated algorithmically, at a difficulty level that is appropriate to the player skill level metric using dynamic difficulty adjustment. Comparing two versions of the game, one with dynamic difficulty adjustment, and one without, revealed that these techniques resulted in a game that was more appealing to a wide range of players.
Key words: procedural generation, dynamic difficulty adjustment, computer games.

Contents

1 Introduction **6**
- 1.1 Aims . . . 6
- 1.2 Motivation and Context . . . 6
- 1.3 Report Overview . . . 8
- 1.4 Glossary . . . 9

2 Background Research **11**
- 2.1 Procedural Generation in Existing Games . . . 11
- 2.2 Dynamic Difficulty Adjustment in Existing Products . . . 13
  - 2.2.1 Flow . . . 13

3 Analysis **14**
- 3.1 Scenario 1: Timed gameplay . . . 14
- 3.2 Scenario 2: Un-timed gameplay . . . 14
- 3.3 Actors and Use Cases . . . 14
  - 3.3.1 Player . . . 14
  - 3.3.2 Level Generation Module . . . 15
  - 3.3.3 Enemy AI . . . 15
  - 3.3.4 Dynamic Difficulty Module . . . 15

4 Specification **15**
- 4.1 Functional Requirements . . . 15
- 4.2 Non-Functional Requirements . . . 16
- 4.3 Extensions . . . 17
- 4.4 Revisions to Specification . . . 17

5 System requirements **18**
- 5.1 Libraries and Editors . . . 18
- 5.2 Software Languages . . . 18

6 Design & Implementation **19**
- 6.1 Unity3D Class Architecture . . . 19
- 6.2 World Generation Style . . . 19
- 6.3 Path Representation . . . 20
  - 6.3.1 Design . . . 20
  - 6.3.2 Implementation . . . 20
  - 6.3.3 Asteroid Belt Shapes . . . 23
- 6.4 Bézier Splines . . . 24
  - 6.4.1 Smoothness across knots . . . 24
  - 6.4.2 Spline Followers . . . 24
  - 6.4.3 Runtime Spline Generation . . . 25
  - 6.4.4 Getting the Player's Position on the Spline . . . 26

6.5 Artificial Intelligence . . . 26
6.5.1 Design . . . 26
6.5.2 Implementation . . . 27
6.5.3 Limitations and extensions . . . 28
6.6 World Population . . . 28
6.6.1 Sphere Collision Resolution . . . 29
6.7 Packing Circles with Rectangles . . . 30
6.7.1 Identifying Valid Points . . . 32
6.7.2 Corner Representation . . . 33
6.7.3 Updating Rectangle Shape . . . 34
6.7.4 Identifying Invalid Points . . . 34
6.8 Optimisation . . . 34
6.8.1 Spline Object Pools . . . 35
6.9 Dynamic Difficulty Adjustment . . . 35

7 Testing Strategy **37**
7.1 White Box Testing . . . 37
7.2 Black Box Testing . . . 37

8 Project Management **38**
8.1 Software Engineering . . . 38
8.2 Risk Management . . . 38

9 Evaluation **39**
9.1 Strategy . . . 39
9.2 Results . . . 39

10 Discussion **41**

11 Conclusion **42**

References **43**

A Project Schedule **45**

B Questionnaire **46**

C UML Diagrams **47**
C.1 Spline Followers . . . 47
C.2 Spline Object Pools . . . 48

1 Introduction

1.1 Aims

The aim of this project is to create a video game that adapts its content dynamically to individual players. The system will create a model of the player's current skill level. Given this model the system will procedurally generate game content at a difficulty level that is appropriate for the player. This will project will require the creation of the game and game content that these systems will be applied to. The game itself will be an arcade style space action game.

1.2 Motivation and Context

Computer games are an increasingly prevalent entertainment medium. In 2009 computer games were played by 68% of American households (Hendrikx et al., 2013). A report conducted by the U.S. Entertainment Software Association (Siwek, 2014) revealed that from 2009 to 2013 U.S. sales of computer games increased from $10.1 billion to more than $15.4 billion.

What is game content? Game content refers to, aside from the player and the game engine itself, anything in the game that affects gameplay. Game content can include textures, levels, stories, dialogue, and terrain (Peytavie et al., 2009).

What is procedural content generation? 'Procedural content generation' in the context of computer games, refers to the use of algorithms to generate game content, rather than generating it by hand. PCG can be used to generate content on the fly as the game is running, or to augment the creation process during development.

PCG sees frequent use when memory and file size are constricted. PCG can be seen frequently in some of the earliest computer games, which were designed around serious constraints in memory and file size. One such example of successful PCG in an early computer game is the space-exploration game *Elite* (Acornsoft, 1984), which can generate 2^{48} distinct galaxies, each generated by applying the generation algorithm to a distinct seed value. Contemporary examples also exist: the 2004 game .kkrieger (.theprodukkt, 2004) utilises PCG to pack a first-person shooter into just 97,280 bytes of disk space. In contrast, the developers claim that a manually generated version of the game would require 200-300mb. Contemporary advancements in computer hardware have lessened memory and file size constraints, and generating game content procedurally to get around hardware constraints has become less of a necessity.

Despite this, PCG is still relevant for contemporary games. First, as reported by the ESA, smartphone apps are an increasingly substantial area of the computer games industry: Apple (2013) (cited in Hendrikx et al.) reported that 145 of the top 300 paid

apps in the Apple App Store were computer games. Smartphones reintroduce memory and storage constraints (Gao, 2011). Downloading apps on mobile data makes small file size a priority, and even most high-end smartphones have significantly less RAM available than home computers. Furthermore, PCG has applications outside of working within constraints.

PCG can make content creation cheaper. AAA computer games have increased dramatically in cost since their inception (Takatsuki, 2007). For example, *Grand Theft Auto V* (Rockstar, 2014) cost $265 million, and was developed by a team of over 1,000 people over five years. Game development still relies heavily on manual content creation, and any technology that can mitigate this bottleneck will lower the risk of development, and allow developers to experiment with riskier game concepts (Yannakakis and Togelius, 2011). For example, the 2012 title *SSX* utilised PCG to generate three hundred game levels on the same budget that had been applied to just ten levels on past games (Howard and Lemus, 2012). Another example of PCG being used to successfully speed up the content creation process is *Speed Tree*, foliage generation middle-ware that can generate a wide range of realistic, animated trees (Togelius et al., 2012).

PCG can personalise the computer game experience to an individual player. The game playing demographic has expanded dramatically the last twenty years (Taylor, 2006). It is therefore more important than ever for games to adapt their content to the player's individual preferences and skill level (Bateman and Boon, 2005). A game that is able to model the player's experience and feed this data back into dynamic content generation can tailor the game content to each player. It is for this reason that this project aims to implement a dynamic difficulty adjustment system. Pedersen et al. (2009) demonstrated the possibilities of this approach by modifying a *Super Mario Bros* clone to generate levels that are personalised to the player.

Figure 1: Pedersen et al.'s level generation for a human player.

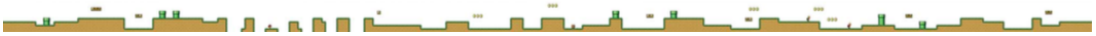


Figure 2: Level generation for the world champion super mario bros AI.

Advancements in computer games have applications in of computer science and other domains. Computer games have applications beyond entertainment for the sake of entertainment. Games see use in military simulation (Stanley et al., 2006), and in training disaster relief workers for disaster situations in a safe manner. The application of game mechanics in non-game domains, known as gamification, has applications in the fields of marketing and education. Any technological development that can make games more accessible can increase the social benefits of existing gamified domains.

1.3 Report Overview

Following on from this section, Section 2 will examine technical details of existing games utilising PCG and dynamic difficulty adjustment. Sections 3 and 4 outline the requirements analysis and specification. Section 5 examines system requirements such as external libraries and software languages. Section 6 describes the design and implementation phase. Rather than separating design and implementation into separate sections, they are considered alongside each other, to better communicate how the design changed throughout the implementation process. Section 7 details the testing strategies utilised throughout development. Section 8 examines project management techniques and software engineering practises. Section 9 examines evaluation techniques and their results. Section 10 looks back on the project's successes and failures. Section 11 is the conclusion, and addresses how well the project addresses the aims outlined in the introduction. In the appendix can be found the project schedule, and UML diagrams for some of the project's classes.

1.4 Glossary

Seed A number used to initialise a pseudorandom number generator.

First-person shooter A computer game centred around gun combat, experienced as if through the eyes of the player's avatar.

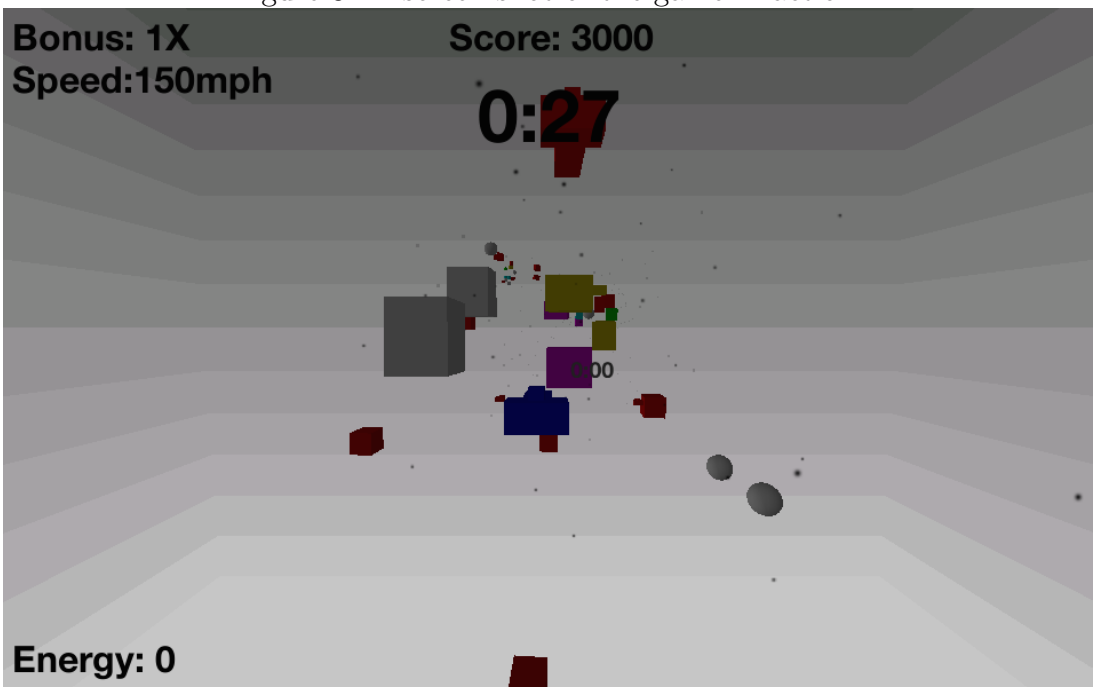
AAA computer game Games classified as AAA or *triple A* are those games with the highest development and advertising budget. AAA games have an expectation of critical acclaim, innovation, and financial success.

Voxel Volumetric pixel - the name given to an individual unit on a three-dimensional grid.

Open World In a computer game, an open world refers to a game or game level that allows the player to choose their approach and the order in which they explore the world. The opposite to this would be a linear game.

Linear game A computer game that presents the player with challenges to be performed in a specific order.

Figure 3: A screen shot of the game in action.

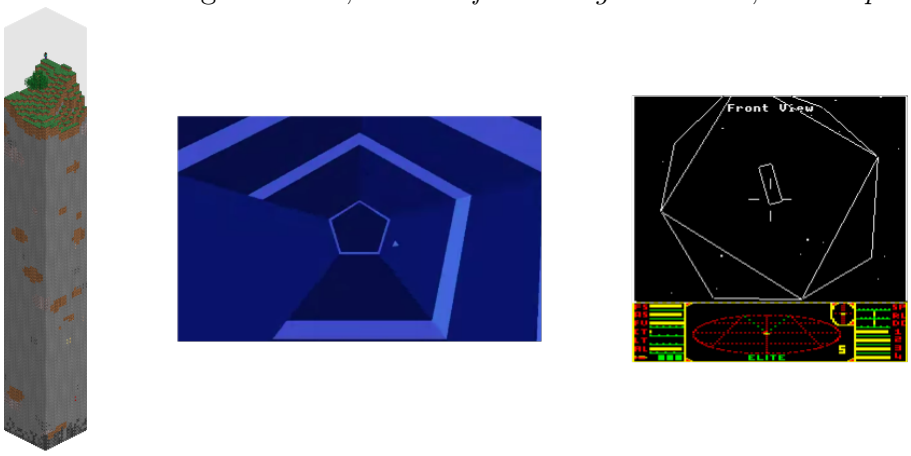


2 Background Research

At the onset of the project, a number of existing games that illustrate successful use of procedural generation and dynamic difficulty adjustment were investigated.

2.1 Procedural Generation in Existing Games

Figure 4: From left to right: *Elite*[1], *Minecraft chunk generation*[2], and *Super Hexagon*[3]



Minecraft With over 19 million players, *Minecraft* may be the most commercially successful example of procedural generation in games. Using procedural generation, *Minecraft* creates an open world, composed of hundreds of thousands of voxels. The use of a seed value, based on the player's position, allows *Minecraft* to generate a persistent world. The world generation algorithm (Persson, 2011), given the same seed value, will always produce the same terrain. Terrain is the product of several layers of Perlin Noise (Perlin, 2002), a pseudo-random noise generation algorithm favoured for nature-inspired procedural generation. The world is generated in 16x16x256 *chunks* as the player explores the world.

Super Hexagon *Super Hexagon* (Cavanagh, 2013) is an example of a linear, arcade-style computer game, which uses a more simple form of procedural generation to create game content. Pre-made chunks of game content called 'gauntlets' are arranged randomly. Procedural generation in this case strikes a balance between the control over game design that comes with handmade content, and the unpredictability that comes with PCG.

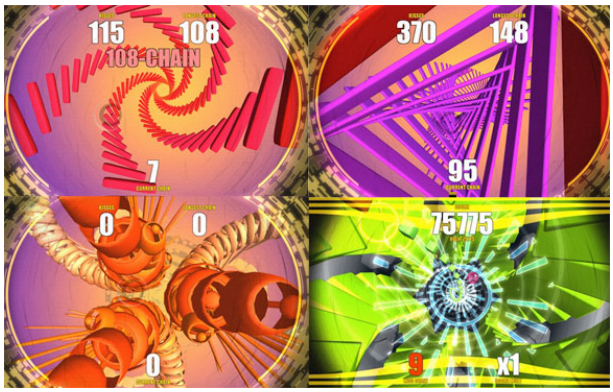
[3]Image source: http://wikipedia.org
[3]Image source: http://minecraft.gamepedia.com
[3]Image source: http://superhexagon.com

The *Aaaaa!* series of games utilises three distinct modules to implement PCG: *sequencers*, *selectors*, and *mutators* (Lambe, 2012). Sequencers create groups of objects, for example, a tunnel composed of cubes, or a collection of power-ups placed along a sine wave. Selectors return a subset of a sequencer, for example, all objects on the positive side of a plane. Mutators are used to apply a change to an object. The pipeline for creating a chunk of the game world is to first sequence a group of objects. Then, apply a mutator to the resulting set of a selector applied to the sequence. For example:

1. Sequence a column of blocks along the player's axis of movement.
2. Select every even numbered block.
3. Apply a mutation to this selection: orient each block sinusoidally over vertical distance.

Small changes in mutator parameters can lead to significant changes in the game content generated.

Figure 5: Different parameters applied to the same sequencer-selector-mutator combination can produce distinct results.[5]



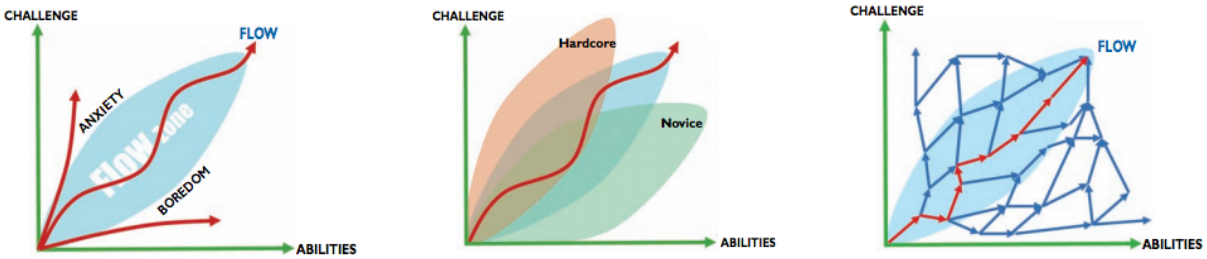
[5]Image source: Lambe (2012)

2.2 Dynamic Difficulty Adjustment in Existing Products

2.2.1 Flow

Csikszentmihalyi (2014) describes flow as a state of complete immersion in an activity, resulting in a highly fulfilling sense of enjoyment. In order to enter flow, a person must strike the ideal balance between challenge and their own skill level. A too high ratio of challenge to abilities leads to anxiety, too low, and boredom occurs. The 'flow zone' lies in the middle of the two. Gamers are drawn to games that induce flow (Holt and Mitterer, 2000), which makes any technique that increase the likelihood of flow for a wide range of players a worthwhile enterprise. Chen (2007) claims that, in order to induce flow in as wide a range of players as possible, games must attempt to settle on the individual player's flow zone. Chen suggests dynamic difficulty adjustment, combined with giving the player the ability to affect their game's difficulty level, as a means to this end.

Figure 6: From left to right: flow zone factors; different flow zones for different players; gameplay choices leading to an adaptation of the flow experience.[7]



Another way to adapt a game to the skill level of the player, and thus induce flow, is to use machine learning techniques to adapt game parameters. Jennings-Teats et al. (2010) propose the use of Ranking SVM to rank game content by difficulty, combined with a dynamic model of the player's current performance to determine game difficulty.

[7]Image source: Jenova Chen

3 Analysis

Problem analysis consisted of describing usage scenarios for the game, and identifying the actors and their use cases in each scenario. As future evaluation would compare two versions of the game to be used in A/B testing (Section 9), two usage scenarios were identified. A description of each usage scenario follows below, followed by the use cases that were constructed as a result of these scenarios.

3.1 Scenario 1: Timed gameplay

This scenario is the way the final product is intended to be played. A human player will control a virtual spaceship with the keyboard. The ship will fly through a procedurally generated environment, gaining more points the further they travel and for shooting enemies, at a speed they can control by spending energy points on boosts. The game ends when a timer reaches zero seconds. The player's aim is to gain as many points as possible in that time limit. The player will store points on the ship, and can choose at any time to 'cash in' these points for more play time. The player will gain more points the longer they can fly without hitting any obstacles. If the player does hit an obstacle, any points that have not been cashed in will be lost permanently.

3.2 Scenario 2: Un-timed gameplay

This scenario plays similarly to the first scenario, but with the omission of any timer-based gameplay. The difficulty will simply increase at a linear rate, and the player will get an amount of points that is directly proportional to the amount of time they stayed alive. The reason for the existence of this much simple mode is to measure the comparative success of the dynamic difficulty adjustment through A/B testing.

3.3 Actors and Use Cases

In the case of a computer game, there exist several entities that are not users, but are still considered to be actors. These entities monitor the state of, and interact with the system. If a use case features in both usage scenarios it will be referred to as a *common use case*, otherwise the specific scenario will be mentioned.

3.3.1 Player

Common use cases are as follows: change the position of the ship using the arrow keys; lock on to and fire bullets at enemy ships using the x key; take damage in the event of a collision with an obstacle. Scenario 1 specific use cases: spend energy points on speed boosts; turn points into extra time.

3.3.2 Level Generation Module

Generate a segment of the world; delete obsolete world segments based on the player's position; populate world segments with obstacles; tweak world segment population to fit current difficulty level. There are no use cases that are specific to either usage scenario.

3.3.3 Enemy AI

Fly ahead of the player; shoot at the player if ahead; change position to avoid obstacles.

3.3.4 Dynamic Difficulty Module

Calculate parameters to be used by the level generation module, given the current difficulty level; in scenario 1, update the difficulty level based on the player's actions.

4 Specification

The following specifications were derived from the use cases and scenarios.

4.1 Functional Requirements

- Path Generation: Given a direction, a distance, and a gradient, the system must be able to generate a segment of the world, represented as a flight path that the player follows.
- Path Concatenation: The system must be able to place new paths at the end of other paths. The system must be able to create a new world segment and delete obsolete ones as the player moves along, to ensure the player never runs out of content.
- Path Content: The system must be able to populate a fixed radius around the path with static obstacles, power-ups, and enemy ships.
  - Obstacles: The system must be able to instantiate obstacles at a given density (obstacles per volume in game units).
  - Power-ups: Power-ups must be placed at regular intervals along the path. When a player collides with a power-up the player's energy level should be incremented.
  - Enemy Ships: Given a regular interval and number of enemies to instantiate, enemy ships should be instantiated along the path at this interval.
- Path Movement: Objects must be able to move along the path at a constant rate of game units per second.

- Player Behaviour: The player must be able to change their relative position to the path while moving along it using the keyboard. The player must follow the path at a constant speed. The player must be able to fire bullets.
- Bullets: Bullets fired by the player must lock on to the nearest enemy ship. Upon firing, the bullets must travel along the path faster than the player and hit the enemy.
- Scoring Systems:
  - Timed gameplay: The game must end when the timer runs to zero. The player must be able to store points locally and convert them into permanent points and more game time. The amount of points gained and the difficulty must increase with the player's distance travelled along the curve. The GUI must display the time remaining, temporary score, final score, difficulty modifier, player speed, and player energy level. If the player hits an obstacle the difficulty must drop down a level.
  - Un-timed gameplay: The score and difficulty level must increase with the player's distance travelled along the curve. The GUI must display the current score and difficulty level.
- Enemy Ships: Must be able to fly ahead of the player. Once in front of the player enemy ships should fire bullets towards the player. Enemy ships must dodge obstacles as well as possible while keeping their movement speed within a certain range.

4.2 Non-Functional Requirements

Non-functional requirements are qualities that the final system must fulfil, referring to the results of functions and desired state of the system rather than desired functions.

- Path Results: The gradient of the path must remain smooth at all times. There can be no sudden changes in the direction of the curve that would be jarring to the player.
- Obstacles: It must be ensured that there is always a valid path the player can take through the obstacles. Even at the highest difficulty level the game must not be impossible. Obstacles must not overlap other obstacles, or power-ups.
- Path Movement: Objects that reach the end of a path must move seamlessly onto the next path - if an object moves x units over the end of a path, it must start at x units along from the start of the new path.
- Performance: The frame rate of the game must remain as consistent as possible. The frame rate must not jump noticeably during gameplay, and must stay above 60 fps at all times on a mid-range system.

4.3 Extensions

These functional and non-functional requirements are not necessary to consider the project complete, but could make the final product more desirable to the player.

- Multiple Path Population Algorithms: To avoid repetitive gameplay, construct multiple algorithms for populating path segments, and alternate these algorithms.
- Procedural Colour Scheme Generation: Generate a unique colour scheme from RGB values for each world segment.
- Moving obstacles: Procedurally generate obstacles that move around the game world in a way that is challenging to the player.

4.4 Revisions to Specification

Realisations during the development process prompted changes to the specification.

- Path Segments: Originally, the system was intended to generate a portion of the game world given a position in the world as a seed value. However, when the game changed from an open static world to a path-locked linear game 6.2 it made more sense to generate it a segment at a time, as the order would be known and it was less important for the world to be persistent.
- Runtime World Generation: Originally the ability to generate the world as the player moves through it was a possible extension, rather than its current position as a functional requirement. This change was also prompted by the focus of the game changing from open world to linear. A static world could be generated at the start of the game but for performance reasons that will be stated later it became necessary to generate the world incrementally at runtime.

5 System requirements

5.1 Libraries and Editors

The use of middle-ware to handle game rendering and physics could allow for more focus on the procedural generation techniques themselves. An entirely distinct project could have been devoted to constructing from scratch a 3D game engine, and while challenging, does not represent any new directions in the field of Computer Science and would detract from the main focus of the project. However, the advantages of middle-ware come at a cost: if software is reliant on the functionality of middle-ware and the functionality is lacking, there is no option other than to abandon the middle-ware and code from scratch. The Unity3D engine was chosen for this product as it is widely used in the games industry - 600 million gamers worldwide have the plug-in installed[8] and if not installed is a small download, so it can reasonably be expected that most of the audience will be able to play the game with relative ease. Unity avoids constricting the user, as any function of the engine can be re-written from scratch if necessary. Furthermore, the author's familiarity with the Unity engine it this a desirable choice. Unity also facilitates compilation to multiple platforms: Windows, Mac OSX, Linux, and Web. This will allow the final product to reach a wide variety of users in the evaluation phase 9.

5.2 Software Languages

Once Unity3D was decided on as the engine to be used in the game, it was necessary to decide on a programming language. Unity supports three programming languages natively: C#; UnityScript, an ECMAScript-inspired custom language; Boo, a .NET language with a Python-like syntax. C# was chosen due to its syntactical similarity with Java, a language the author was already familiar with.

[8]Source https://unity3d.com/public-relations

6 Design & Implementation

Design choices were made at the beginning of the project, but agile methodology left these choices open to change. New choices were made throughout the process of implementation. Broadly speaking these changes did not significantly alter the design from its original state, but some previously designed and some implemented features were replaced with more appropriate features. The main change during implementation was the shift from a static world to an infinitely generating linear path. Taking this into consideration, in this section I will explain design choices alongside their implementation, justifying initial design choices with reference to the system specification and changes encountered during the implementation.

6.1 Unity3D Class Architecture

One feature of Unity3D is a component architecture paradigm. In this paradigm, every object that is present in the game world is an object of *GameObject*. For example, the player will be a game object, as will the bullets they fire, the asteroids they dodge, or enemy ships. Game objects store a *Transform*, an object which contains information pertaining to world position, scale, and orientation, and a list of *Components*, classes that derive from *MonoBehaviour*. Every time the game updates, every component in every game object is updated. Components tend to contain data and functions that address a specific concern of a game object, for example, a *HitPoints* component could derive from MonoBehavior, and contains an int *healthLevel* and a method *CheckHealthLevel()* that kills a game object if its health runs below zero. Using a component architecture paradigm means that the player, the enemies, the bullets are not distinct classes - they all derive *GameObject* and the way they behave is defined from their list of components. A component-based approach can cut down on repeating code, and increase code clarity - each component is focused on one behaviour, so functionality is naturally separated across classes.

6.2 World Generation Style

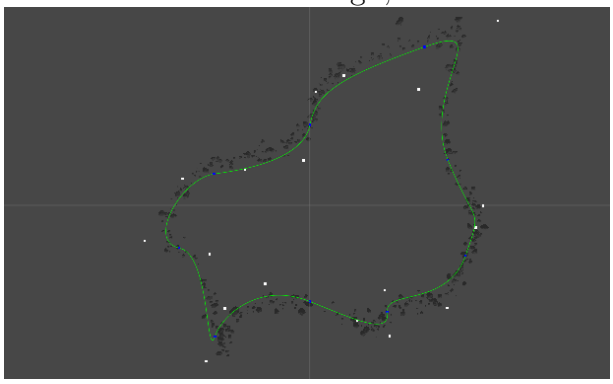
It was necessary to identify the desired results of the world generation. One possibility for the world generation style was a static, unchanging world, generated at the start of the game. Another option would be to lock the player to a path and generate the game on the fly as the player moves through it. Originally, a static world design was chosen, as a persistent world could be more appealing to the player by offering more exploration options. However, it was discovered during implementation that procedurally generating smaller amounts of the world as the player moves through the world removed long initial loading times, and was better suited to the dynamic difficulty adjustment, as segments of the world could be generated with parameters adjusted by difficulty modifiers. It was harder to tweak a static world post-generation than to generate at an appropriate difficulty level as the player progresses.

6.3 Path Representation

6.3.1 Design

Regardless of whether the game world was to be a static asteroid belt or an infinitely generating linear level, it was necessary to have a means of representing a path in the game world. Around this path, obstacles can be generated - this allows the world generator the ability to choose a general shape, and fill in the details automatically. A static asteroid belt would be represented as a looping set of paths, and an infinite level could be a a series of connected paths.

Figure 7: The initial world design, a static asteroid belt.



One simple way to represent the path would be a series of points connected by straight lines. This choice was ruled out, as in the non-functional requirements (section 4.2) a requirement was that the gradient of the path must remain smooth at all times, with no sudden changes in direction. The gradient of of the path would change at connecting points. A parametric representation of a curve, such as a Bézier curve, could fulfil this requirement. Through manipulation of the end points of curves, gradient continuity could be maintained. Equally, parametric representations of curves have performance advantages: Bézier curves were chosen as three-dimensional curves of arbitrary lengths can be generated from a set of three-dimensional vectors and a value from zero to one, where zero is the start of the curve and one is the end. Points on the curve can be generated to as high a precision as the input parameter is capable of. It was necessary to decide what degree Bézier curve would be used to represent the path. Cubic Bézier curves were chosen as they had the best balance of performance and smoothness. This design decision is explained in more detail in section 6.3.2.

6.3.2 Implementation

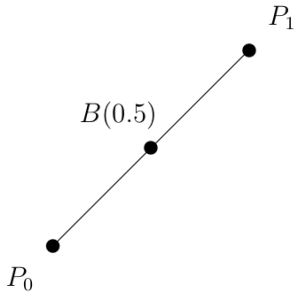
Cubic Bézier curves were chosen to represent the flight path. A Bézier curve of n degrees can be defined recursively as a linear interpolation between points in two Bézier curves of $n - 1$ degrees: for example, a point on a cubic Bézier curve is given by a linear interpolation between two points on two quadratic Bézier curves, which are turn given by a linear interpolation between two points on on two linear curves. A linear Bézier curve

is represented by two points, P_0 and P_1, and a point on the curve is simply generated by linearly interpolating between P_0 and P_1. The 'curve' is given by

$$B(t) = P_0 + t(P_1 - P_0)$$

and should result in a straight line starting at P_0 and ending at P_1.

Figure 8: Linear Bézier curve with the point at $t = 0.5$.



A quadratic Bézier curve is represented by three vectors: start and end points P_0 and P_2, and an intermediate control point P_1. To calculate a point on a quadratic Bézier curve at t, first calculate the points at t on two linear Bézier curves: C_0 with start point P_0 and end point P_1, and C_1 with start point P_1 and end point P_2. Then linearly interpolate between these two points. The curve for a quadratic Bézier curve is given by

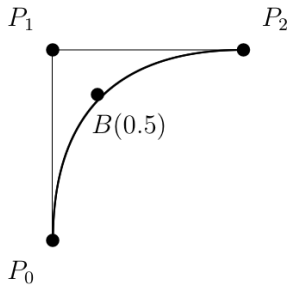
$$B(t) = (1-t)((1-t)P_0 + tP_1) + t((1-t)P_1 + tP_2)$$

or rewritten into a more complex form

$$B(t) = (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2$$

and should look like a line being stretched towards P_2.

Figure 9: Quadratic Bézier curve with the point at $t = 0.5$ marked and straight lines indicating C_0 and C_1.

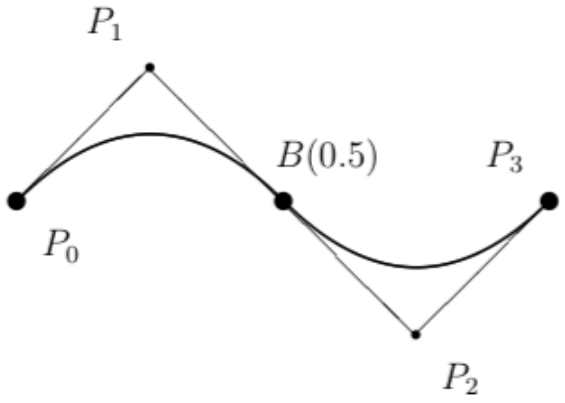


Cubic Bézier curves are simply the next level of interpolation: curves are represented by start and end points P_0 and P_3 with two control points P_1 and P_2. To calculate a point on a cubic Bézier curve calculate the points at t on two quadratic Bézier curves: curve C_0 with points P_0, P_1, and P_2, and curve C_2 with points P_1, P_2, and P_3. The curve for a cubic Bézier curve is given by

$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)^2 P_2 + t^3 P_3$$

and should look like a line being stretched towards P_1 and P_2.

Figure 10: Cubic Bézier curve with the point at $t = 0.5$ marked and straight lines indicating C_0 and C_1.



The first derivative of the curve is given by

$$B'(t) = 3(1-t)^2(P_1 - P_0) + 6(1-t)t(P_2 - P_1) + 3t^2(P_3 - P_2)$$

and equals the gradient of the tangent to the curve. This comes in useful when objects are travelling along the curve, as they need to face the direction of travel. Cubic Bézier curves were chosen over quadratic curves because they are $C(2)$ continuous. Generally, a curve is considered $C(n)$ continuous if all its derivatives up to n match across curve segments. The application of this is that the rate of change of change of gradient in a $C(2)$ continuous curve is continuous. Continuousness over $C(2)$ was not needed and would only increase the number of interpolations needed, so $C(2)$ was picked as a balance between smoothness and performance.

Figure 11: $C(1)$ and $C(2)$ continuity across two curve segments[9]

A static class was created to store functions, that given $0 <= t <= 1$ and four points P_0, P_1, P_2, P_3, could return a position on a curve, the length of a curve, and the first derivative at that point. The length of a Bézier curve can be approximated by getting the combined distances between a set of points along the curve.

[9]Image Source: http://research.cs.wisc.edu/graphics/Courses/559-f2004/docs/cs559-splines.pdf

Figure 12: The path that approximate curve length is calculated from.

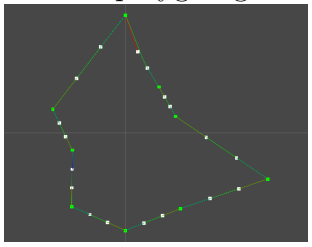
6.3.3 Asteroid Belt Shapes

To generate a randomly-shaped unpopulated asteroid belt, generate a random polygon by rotating around an origin and placing points at random distances, then linking the points up and make each edge a Bézier curve. To make the curve continuous across the loop it is necessary to implement Bézier splines.

Data: Minimum and maximum distance from centre
Origin (three dimensional vector)
Number of edges: n
Result: A random polygon
Initial bearing is 0°;
while *bearing* $< 360°$ **do**
  Add a point along the bearing from the origin, at a random distance between minimum and maximum distance;
  Bearing $+= 360/n$;
end
forall the *Points* **do**
  if *This is the last point in the list* **then**
    Make a new Bézier curve: P_0 = this point, P_3 = first point in list;
  else
    Make a new Bézier curve: P_0 = this point, P_3 = next point in list;
  end
end

Algorithm 1: Random polygon generation

Figure 13: Random polygon generation results



6.4 Bézier Splines

A spline is a set of multiple curve functions. Curve n's start point should be the same as curve $(n-1)$'s end point - this shared point is known as a knot.

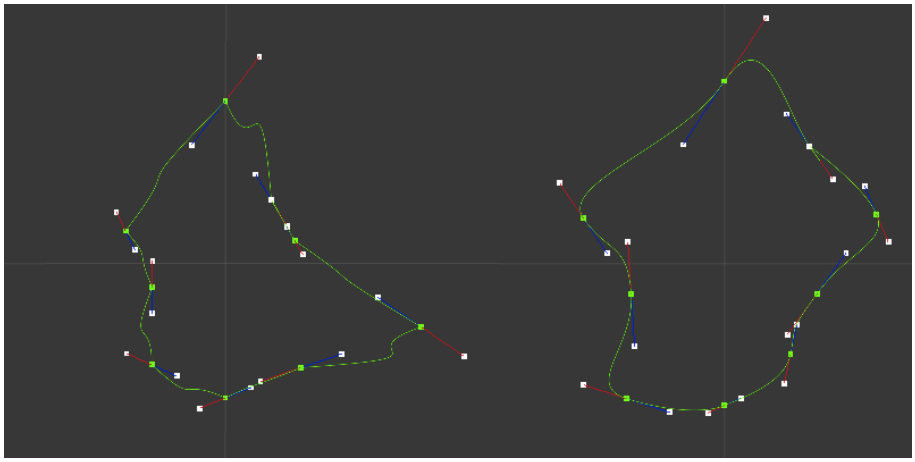
6.4.1 Smoothness across knots

Smoothness across knots is an essential quality of splines. If Bézier curve C_1 is followed by curve C_2, the following conditions must be true to ensure smoothness across the knot:

1. The start and end points are equal to the knot: C_1's $P_3 = C_2$'s P_0
2. The knot lies on the line connecting C_1's P_2 and C_2's P_1

Condition one should be satisfied, as the start point of the new curve is always set to the end of the first curve. To enforce condition 2, get the vector V as C_1's $P_3 - C_1.P_2$. Place C_2's P_3 at C_1's $P_3 + V$, mirroring the points over the knot.

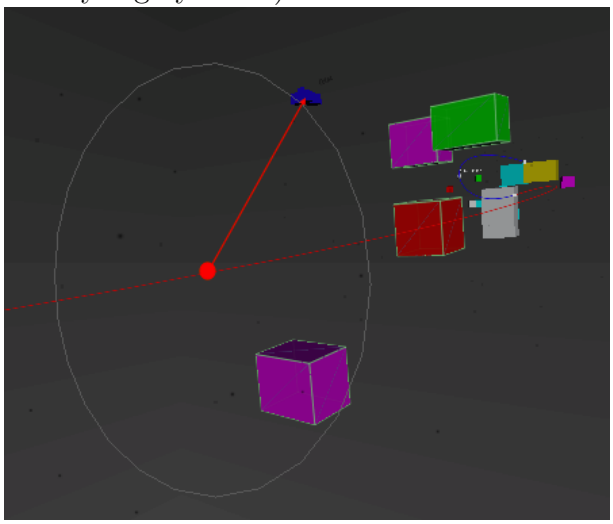
Figure 14: Random polygons with smoothness conditions unenforced (left) and enforced (right).



6.4.2 Spline Followers

The player, enemies, and the player's bullets all need to be able to travel along the spline. One possibility was to give entities that follow the spline free movement, and if they stray too far from the spline, rotate them back towards it. The further away from the spline the entity is, the more forceful the rotation should be. However during playtesting this turned out to be frustrating - users did not like having control taken away from them. Instead, entities moved directly along the spline by incrementing their t value. The entity's position on the spline was subsequently calculated as $B(t)$, with the P values being the points of the occupied curve. Spline followers also have a two-dimensional vector called an offset (Fig. 15), which determine their position within a plane with normal $B'(t)$.

Figure 15: A spline follower's position is $B(t)+$ offset (red arrow) along the plane with normal $B'(t)$ (represented by a grey circle)



Objects that need to move along the spline (enemies, the player, player bullets, the camera) are given a component that derives from *SplineFollower*. Every frame, spline followers move along the curve by $\delta t = s/l_1$, where s = game distance units travelled per second and l_1 is the length of the occupied curve. Spline followers need to move at a constant speed, even across curve segments, but curve segments do not necessarily all have the same length as each other. Therefore, if the incremented t is over one, the spline follower has left the current curve segment and t must be adjusted taking into account the next curve segment's length. First, the ratio r of how far over the curve the new t value is calculated as $(t-1)/\delta t$. The distance d the spline follower has to move along the next curve segment with length l_2 is given by $r * s$. Thus, the new t value is corrected to d/l_2. UML diagrams for spline followers can be found in the appendix.

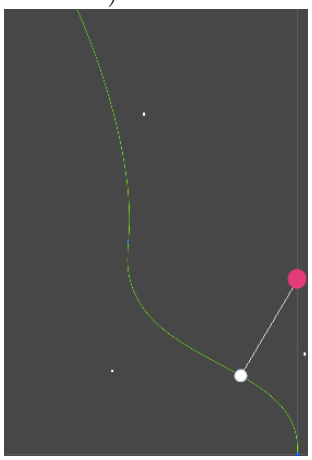
6.4.3 Runtime Spline Generation

The spline that the player follows consists of three curves: C_1, the curve the player is currently on, C_2, the next curve in the spline, and C_0, the curve before the current curve. When the player reaches the end of C_1, the curves are 'shifted down' by one: C_1's curve data (the set of four three-dimensional vectors) moves into C_0, C_2 to C_1, and at C_2 a new curve is generated with C_1's end point as its start point. Smoothness conditions are enforced (section 6.4.1). All spline followers other than the player have their curve index decremented by one to keep them in the same position. The class *InfiniteSpline* holds the three curves and updates the curve when the player passes off the edge of the current curve. When the curves update *InfiniteSpline* also updates the curve index of every spline follower.

6.4.4 Getting the Player's Position on the Spline

While the player still had free movement, a way of detecting what their t value on the current curve was needed, as when that value goes over 1, a new curve segment should be generated. When the player's position on the spline was set directly, there was no point in calculating this, but as this was not an insignificant problem to solve it will be described here. Binary search was performed on the curve - first, the curve was bisected with a plane, with a point $B(0.5)$ and normal $B'(0.5)$. Depending on whether the player's position is on the positive or negative side of the plane, the bisection runs again on the curve from $t = 0.5$ to $t = 1$ or $t = 0$ to $t = 0.5$, and runs a number of times given as a parameter to the function. Experimentation revealed that 16 was the point of diminishing returns for curves of length 2000, the length picked to be used in game. This was a relatively cheap operation, but one optimisation technique would be to take into account past results. As spline followers all travel in the positive direction, if a spline follower is on the positive side of a plane, it can be assumed to always be on the positive side and this does not need to be recalculated. This method was implemented and added to the Bézier static class.

Figure 16: Player's (pink dot) t value on the curve (white circle).



6.5 Artificial Intelligence

6.5.1 Design

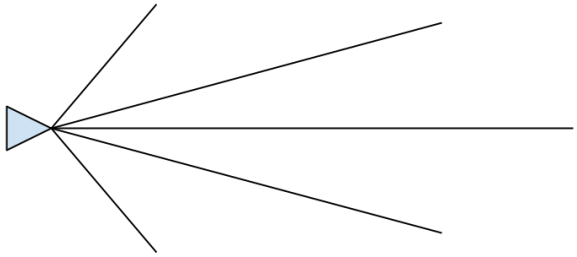
Enemies need a way to avoid obstacles while traversing the spline. One option is to use a search algorithm to construct a path through the obstacles, follow it, and calculate a new path to follow when the end of that path is reached. One pathfinding algorithm considered was A* search, which has the advantage of guaranteeing a valid path through the obstacles. One disadvantage however is that A* search would have to be performed as many times are there are current instances of enemies. It would be possible to calculate one or a fixed number of paths that all the enemies could follow, but if enemies were to overlap they would collide with each other and not move in a way that is satisfying to the

player. Equally, it is hard to reconcile A* search with a moving game world (a possible extension in section 4.3) as search would have to be recalculated whenever an obstacle intersected the path. It became necessary to abandon path finding and adopt a method that only looks at immediate obstacles. Enemies will look ahead of their movement direction using virtual depth sensors, and adjust their position based on whether they are about to run into an obstacle. Another softer advantage of this approach is that enemies will behave much like an actual player-controlled ship, as they are reacting to what they can see, and do not have the omniscience that comes with pathfinding algorithms like A* search. In this case, non-perfect pathfinding is actually an advantage.

6.5.2 Implementation

The raycasting feature of Unity3D's phsyics library was used for this functionality. Given an origin, a direction, and a distance, a raycast returns true if it collides with any game object that has collisions enabled and optionally a list of all said game objects. This list was not needed, as the enemies had to dodge anything they could collide with. According to their *SplineFollower* component enemies will traverse the spline at the same speed as the player, unless they are behind the player in which case they will move slightly faster in order to catch up. It is within the *OffsetController* that obstacle dodging occurs. Chenell (2014) outlines a two-dimensional version of this obstacle avoidance using five rays, and upgrading to three-dimensions was a matter of adding rays in the z direction. Each enemy has nine rays which are cast every frame: one going straight ahead, and four pairs of directional raycasts (up, down, left, and right). For each directional raycast there is a long-range version as well as a shorter-range version, to prevent enemies from moving sideways into obstacles they cannot see.

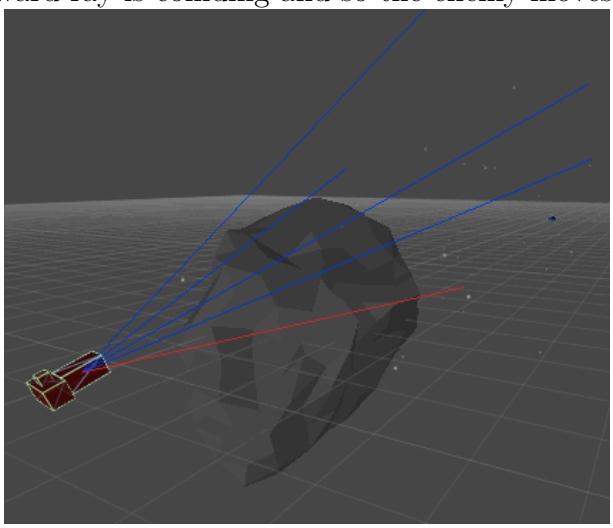
Figure 17: Top down diagram of enemy raycasting.



If a ray cast to the left returns true, the enemy is heading towards an obstacle and adjusts its offset to the right. A right ray returning true sends the enemy to the left, up to down, and down to up. If just straight ahead, both left and right, or both up and down are true, the enemy has to decide which direction to move, as all directions are equally valid. Enemies are assigned a random desired offset when they are instantiated, which is an offset they attempt to return to if they are not due to collide with any obstacles. In the case the enemies have to make a decision which way to dodge, they attempt to move

in the direction of that desired offset. This desired offset can also be set to be the offset of the player to make enemies fly in front of the player. Enemies also begin dodging at a low speed, and the longer they are due to collide with an obstacle, the faster their dodging becomes. This means enemies will only move as fast as they have to to avoid obstacles and will not twitch violently.

Figure 18: Enemy AI raycasts. Rays that hit colliders are rendered red in debug mode. In this case the downward ray is colliding and so the enemy moves upwards.



6.5.3 Limitations and extensions

One issue encountered was that an enemy that was constantly dodging in the direction of their desired offset would reach the desired offset and then not be able to move off it. A boolean *stillColliding* was used to keep track of whether the enemy had been dodging last frame, and if it was, it would maintain its dodge direction. This way enemies would move past their desired offset and keep going until they were safe, not just charge head-on into the obstacles. Another issue is that in areas with a high density of obstacles, enemies are constantly dodging. If the enemies were attempting to head towards the player it was unlikely this would ever happen. Fortunately, the player is forced to move close to the enemies in order to shoot them anyway. One possible extension to enemy behaviour would have been to set the enemies' desired offset points in such a way that makes them fly in a formation.

6.6 World Population

The world population design changed more during implementation than any other part of the design. Once a path representation is established, it is necessary to populate the path with game content. As stated in the non-functional requirements (section 4.2) obstacles must not overlap other obstacles. The original design choice was to naively drop

obstacles in the game world, and afterwards, resolve any overlapping obstacles. At first, it was decided to use Unity's in-built physics engine to move the obstacles apart from each other, but it was discovered that Unity's physics engine does not necessarily result in a stable placement of the obstacles. One simple solution would be to let Unity's collision resolution run for a number of frames and then freeze the results. In practise, however, Unity's physics engine is linked to frame progression, so the player would see a number of extremely slow frames before the game began. A custom sphere collision resolution system was designed and prototyped.

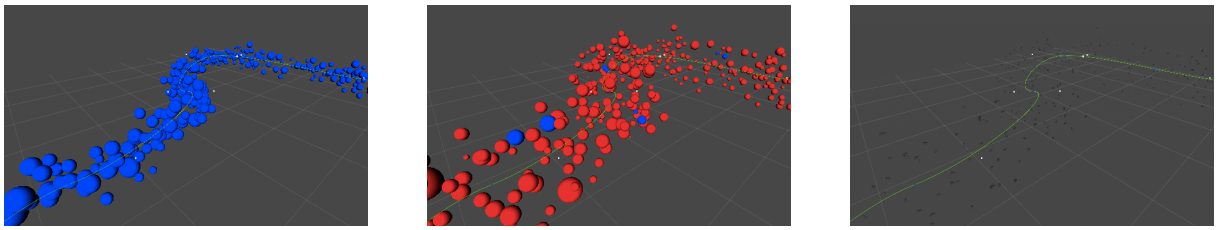
6.6.1 Sphere Collision Resolution

Asteroids were placed at random points around the spline, without checking if they overlapped. An object of class *Sphere* was created at the midpoint of every asteroid, with a radius that encompassed the extents of the asteroid model. Once all spheres were in place, collision resolution was performed. Two spheres with radii R_1 and R_2 and centers P_1 and P_2 are overlapping if:

$$||\overrightarrow{AB}|| > R_1 + R_2$$

If two spheres are colliding, the sphere that is further away from the spline is moved to the point at which $||\overrightarrow{AB}|| = R_1 + R_2$. Once the collisions are resolved the asteroid models themselves are placed in the world at the resting point of the sphere, unless the sphere does not resolve, in which case it is abandoned.

Figure 19: Initial sphere placement, spheres after collision, and final asteroid placement. Blue spheres are colliding with other spheres, red spheres have completed collision resolution and are stable.

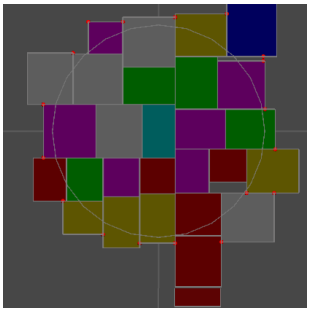


A significant delay during the implementation of sphere collision resolution occurred as the result of isolated vertices in the open source asteroid models used as placeholder art. An isolated vertex is a vertex that is not connected to an edge, and can be the result of an error during the modelling process. The size of the model was used to calculate the size of its sphere collision object, but, as the radius of the model was calculated as the distance between the centre of the model and the furthest vertex from the centre, a stray vertex could significantly alter the estimated model size. This was a difficult error to identify as isolated vertices are invisible.

In practise during implementation, however, sphere collision resolution at this scale was discovered to be prohibitively slow. If collision detection is run on every pair of

spheres it has a complexity of $O(n^2)$. Even limiting the collision detection to the closest n spheres was ineffective - an n value large enough to examine all collisions was too slow to be viable. With hundreds of asteroids being placed along the spline, it became necessary to develop a world population system that would not result in any collisions that needed to be resolved. To this end 'slices' of the path with no colliding obstacles were generated and the obstacles within these slices were spread out along the curve. This design lent itself well to dynamic difficulty adjustment, as slices could be spread out less to increase the density and hence the difficulty. This design is analogous to *Aaaaa!*'s sequencer-selector-mutator pipeline. First, the slice is sequenced by placing obstacles in a non-overlapping layout. Second, when the slice is placed on the spline, a proportion of the obstacles are selected to toggle on - the higher the difficulty, the more obstacles are toggled on, increasing the obstacle density. Thirdly, mutation occurs - obstacles are moved along the curve in the positive or negative direction, spreading them out along the curve. The higher the difficulty, the less the obstacles are spread, again increasing the obstacle density.

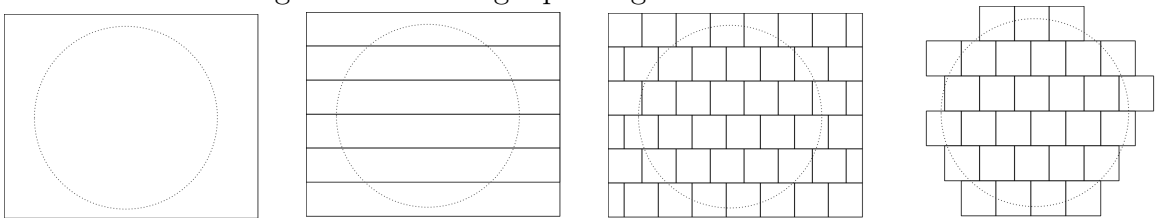
Figure 20: A slice of the path consists of a rough circle, tightly packed with non-overlapping obstacles.



6.7 Packing Circles with Rectangles

The desired results of the slice generation is a collection of non-overlapping rectangles tiled in a circle area with a minimum amount of gaps. One way of arriving at this result would be to start with a square, divide it horizontally into strips and divide these strips vertically into rectangles, and then remove any resulting rectangles with a high enough percentage of their area outside the circle.

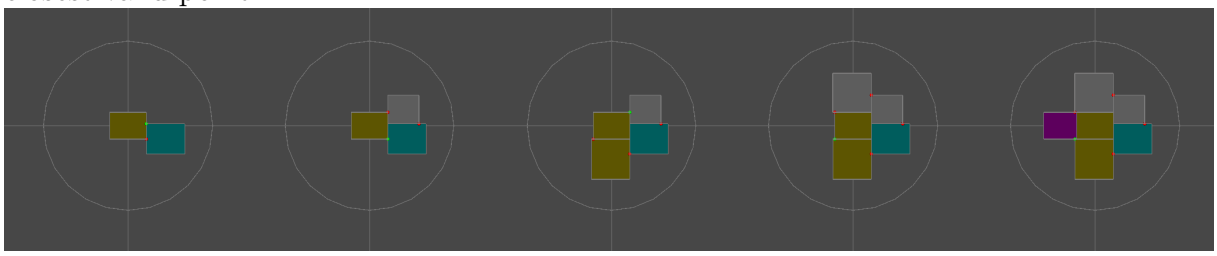
Figure 21: Rectangle packing as a result of bisection.



This is a simple way of approximating the desired results, and guarantees there will be no overlapping rectangles or gaps between rectangles. The downside of this method is the results are too predictable, as squares are aligned within their strips. A method was conceived wherein the circle would be built one rectangle at a time. To minimise gaps between rectangles, rectangles should be placed at points where the corner of one rectangle meets another rectangle's edge. An advantage of this method is that rectangles of a variety of sizes can be placed in the circle. A brief overview of this algorithm is to start with two touching rectangles and follow this loop:

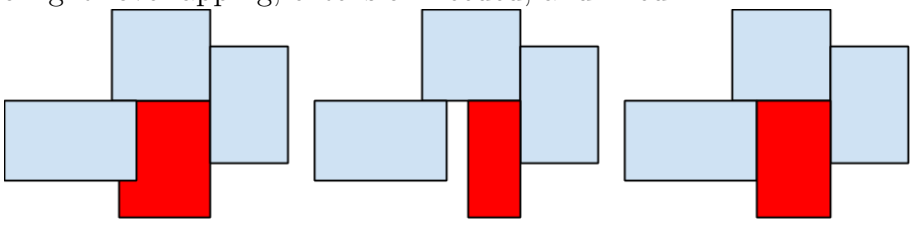
1. Identify valid places to place a rectangle.
2. If there are no valid places, exit the loop.
3. Place a rectangle at the valid point that is closest to the center.
4. Set the rectangle's width and height to a random values between a pre-determined minimum and maximum distance.

Figure 22: Step by step rectangle packing. Red dots are valid points, the green dot is the closest valid point.



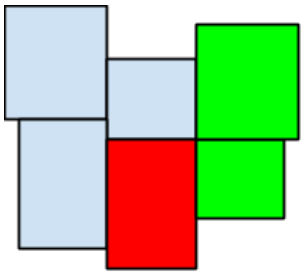
This design appeared to be relatively simple but during implementation a number of specific cases were identified through white box testing (section 7.1). This necessitated some changes to the design. In some cases rectangles have to be resized: if a rectangle is placed, but leaves a gap that is too small for another rectangle of minimum size to be placed, it should be extended to fill that gap; if a rectangle is placed, and overlaps another rectangle, it should be shrunk to the correct size.

Figure 23: Cases in which the size of the rectangle being added (red) needs to be changed. From left to right: overlapping, extension needed, and fixed.



Another special case is the case in which the corner of one rectangle has the same position of another rectangle's corner. In this case, it is possible to place two rectangles at this point.

Figure 24: Two corners touching results in two valid rectangle placement points (marked in green)



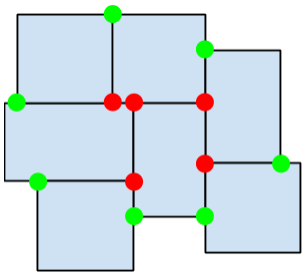
The final implemented design of the algorithm is as follows:

1. Place a rectangle, and a second rectangle with a corner on one of the first rectangle's edges.
2. Identify the initial valid points to place rectangles (section 6.7.1).
3. Perform this loop until there are no valid points to place rectangles:
   (a) From the list of valid points (section 6.7.1, get the point that is closest to the centre of the circle.
   (b) Place a new rectangle at this point.
   (c) Extend or shrink the new rectangle appropriately (section 6.7.3).
   (d) Identify any new valid points.
   (e) Remove any invalid points (section 6.7.4) from the new valid points.
   (f) Add new valid points to the list of known valid points.
4. For every rectangle, create a three-dimensional obstacle that fills the rectangle.

6.7.1 Identifying Valid Points

Points at which the corner of one rectangle intersects the edge of another rectangle are potentially valid placement points. Intersection points can only be valid placement points if they are on the outside of the existing group of rectangles. One way to identify which placement points are valid and which are invalid would be to generate all potential points, and cull the ones that are touching three or more rectangles. Another way would be to examine the newly added rectangle, and add to a persistent list of valid placement points. The advantage of this method is that it is not necessary to recalculate the placement point list every time a new rectangle is added.

Figure 25: Potentially valid placement points at one step of the algorithm. Valid points are marked in green, invalid in red.

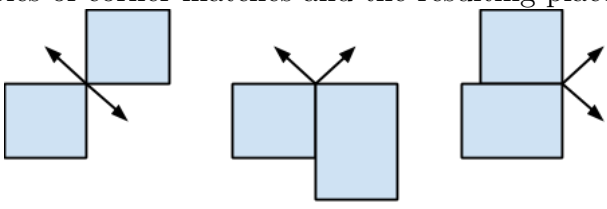


6.7.2 Corner Representation

In order to determine where to place a rectangle, placement points have to have an associated corner type. This corner type refers to which corner of the rectangle will be placed at the placement point. The corner type is determined by the two intersecting rectangles: for example, a corner of type *topleft* touching the right edge of another rectangle will result in a placement point with corner type *bottomleft*.

First, the algorithm examines the new rectangle's corners: which corners are touching the edges of an existing rectangle? Exact corner matches are checked for first. For each of the old rectangles' corners, and for each of the new cube's corners, if the positions are equal, the corners match. As positions were represented by floats, equality checking had to be changed to checking if the distance between the two points was less than 0.1. This value was small enough not to give false positives and large enough to catch all 'equalities'. If it is the case that the two corners have the same position, the algorithm will place two rectangle placement points. The type of placement point is determined by the types of the two corners that are touching.

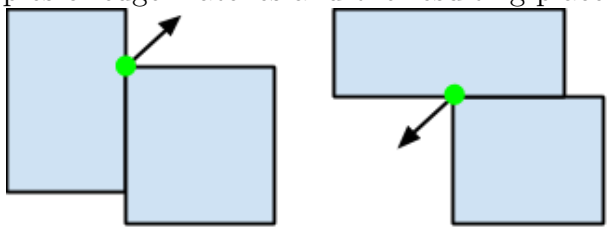
Figure 26: Examples of corner matches and the resulting placement point types.



If a corner does not exactly match another corner, the algorithm checks whether it is touching the edge of another rectangle. If it is, a potential placement point can be added at the intersection. Again, the type of placement point depends on the corner type that is touching the edge and the type of the edge it is touching.

The algorithm repeats, examining all the corners of the set of existing rectangles: which corners are touching the edges of the new rectangle? In this case, the algorithm does not check for exact corner matches, to avoid duplicate placement points.

Figure 27: Examples of edge matches and the resulting placement point types.



6.7.3 Updating Rectangle Shape

Rectangle's shapes need to be updated if they can expand or have to shrink to fill a gap. To update the rectangle's shape, check for each edge of the rectangle R_1 with bottom left corner (l_1, b_1) and top right corner (r_1, t_1) : is the edge overlapping another rectangle R_2 with bottom left corner (l_2, b_2) and top right corner (r_2, t_2)? To resolve the right edge, first check that the edge lies within R_2's top and bottom edges

$$(b_1 > b_2 \wedge b_1 < t_2)||(t_1 > b_2 \wedge t_1 < t_2)$$

then the right edge is overlapping if

$$l_1 < l_2 \wedge r_1 > l_1$$

and the right edge needs to be extended if

$$|r_1 - l_2| < w \wedge r_1 < l_2$$

where w is the minimum valid width for a rectangle. The potential new edge values are stored in a list, and only the value that is closest to the centre is used as the final edge value.

6.7.4 Identifying Invalid Points

During rectangle placement, if an invalid point is detected for any reason, it can be added to a list of known invalid points. These invalid points are culled from the list of new valid points before it is added to the list of known valid points. If changing a rectangle's shape causes it to obscure a previously valid corner, the obscured corner is added to the invalid points list. Also, the point that the new rectangle is created at is added to this list, so rectangles are not placed on the same point more than once.

6.8 Optimisation

The large amounts of objects the game would be creating at runtime necessitated some optimisation. One method considered was distance culling - simply not rendering objects that are too far away from the player to be considered important. However, this creates a 'popping in' effect where obstacles seem to appear out of thin

air, which visually is not desirable. Equally it does not actually stop objects from being created, just hides them from the player's view. Instead, a system of object pooling was decided upon - the game world will consist of a small amount of objects (a pool), created when the game loads, and being activated when needed and deactivated when obsolete. The disadvantage of this method is the increased initial loading time, but should ensure a more consistent frame rate. Object pooling also lends itself well to the world population design choice (section 6.2) as slices can be pre-generated and given custom configurations when they are placed on the curve, rather than generating a new slice every time it is needed.

For every object that needed to be pooled, a class deriving from *ObjectPool* was created. Any object that was going to be created and destroyed frequently was pooled: enemies, slices, bullets, power-ups and the player's bullets. At construction, *ObjectPool*s create a number of game objects, given as a parameter, and add them to a list. Classes that derive from *ObjectPool* override *CreateGameObject()* with the specific code required to create their game object. When any other class wants to create an instance of a game object it calls *GetPooledObject()*, which searches through the list for an inactive game object and returns it if found, else, returns null. If an *ObjectPool* has its *willGrow* value as true, and an inactive game object is not found, at this point it will expand the list and create a new game object and return that.

6.8.1 Spline Object Pools

For any object that needed to be placed on the spline, an object of class *SplineObjectPool* was created. *SplineObjectPool*s are given a separation value s, specifying the distance along the spline in between objects, and an *ObjectPool* to fetch these objects from. Given the player's t value t_p on the current curve, the *SplineObjectPool* looks ahead on the curve at $t_s = t_p + d/l$ where d is the distance to look ahead, and l is the length of the current curve. If t goes over one, the remainder is adjusted in the same way spline followers move over curve segments (Section 6.4.2). *SplineObjectPool*s contain a field t_l to keep track of the last t value at which an object was placed. If $t_s > t_l + s$ objects need to be separated by, an object is placed and the last t value is updated. Objects behind the player by a certain amount are deactivated, freeing them up to be pooled again. Also, in the timed mode of the game, when the player hits an obstacle, t_l is reset to zero after the difficulty value is updated, forcing the spline to regenerate at an appropriate difficulty level immediately. UML diagrams for Spline Object Pools can be found in the appendix.

6.9 Dynamic Difficulty Adjustment

There exist already several machine-learning techniques for tweaking difficulty parameters in computer games, which have been shown to effectively settle on suitable parameters. These algorithms are mostly designed as an automation technique for playtesting

before a game is released, and the specifications stated that the game should adjust parameters while the game is running. Therefore, a similar technique to the one employed in Jenova Chen's *flOw* was adopted, as it allows for a relatively quick settling on the appropriate difficulty level through player choice. The difficulty manager contains a method to generate an appropriate value for each difficulty dependent parameter of the procedural generation. For example, for obstacle density at difficulty t, with a minimum and maxmimum density d_1 and d_2, the difficulty manager returns

$$d_1 + d(d_2 - d_1)$$

DifficultyManager also returns text relating to the difficulty for the GUI to represent.

Figure 28: Level generation for a player of the lowest skill level.

Figure 29: Level generation for a player of the highest difficulty level features higher obstacle density and enemy placement.

An abstract class *ScoreManager* was used, and different classes can derive from *ScoreManager* to create different scoring systems. One of these scoring systems updated the difficulty relative to the distance along the curve the player travelled, and decremented by 0.1 if the player failed (hit an asteroid or an enemy bullet). The other simply increased the difficulty linearly over time. The ability to use multiple scoring systems came in useful during evaluation, as different scoring systems could be compared to evaluate their relative success. *ScoreManagers* also return text representing the score that the GUI displays.

Three different difficulty progression functions were implemented. A static class *DifficultyProgression* was used to hold these functions. By switching out the difficulty progression function the difficulty could progress linearly, exponentially, or following a square root graph, and the game was playtested with a variety of functions to identify the most appropriate. Linear progression emerged as the most appropriate of the three.

7 Testing Strategy

The project testing strategy was twofold: white box testing throughout development, and black box testing primarily at the end of development.

7.1 White Box Testing

White box testing examines specific aspects of the internal structure and behaviour of software. The primary white box testing strategy used was smoke testing.

Smoke testing, or sanity testing, aims to confirm that high-level functions of software work. These functions are derived from the specification. To confirm functionality, a static class *DebugUtil* was created with a method *Assert*. Test methods to check if requirements were being fulfilled were designed, and their results passed into *Assert*. If the test method returned false, *Assert* would throw an exception and a suitable error message. Examples of tests include a test to confirm no obstacles were overlapping each other, or a test to ensure the gradient of the Bézier spline was consistent across curve segments. When the project failed to pass smoke tests, static testing and, in some cases, visual debugging, was used to identify and fix the problem. Static testing tests software without running it and primarily consisted of manual code reviews.

Visual representations of the state of algorithms proved to be a useful testing approach. The visual focus of the project lent itself well to informal visual testing. Visual testing consisted of editing the Unity editor to display information about the state of the program, and moving through code execution one instance of the game loop at a time. For example, when the square-packing algorithm was placing supposedly valid square placement points in invalid locations, displaying all the points in their position in-game made it much easier to identify what cases were causing errors.

7.2 Black Box Testing

Black box testing compares the behaviour of the program as it would be experienced by an end user, rather than examining the internal structure or behaviour. The behaviour of the project was compared directly to the specification (section 4). The author performed actions in game and checked that the system responded in the expected manner, according to the specification. For example, the player flew into obstacles in game. If the bonus value in the GUI decremented and the player lost all their current points, the test was passed.

8 Project Management

8.1 Software Engineering

An agile software development methodology was chosen as the software engineering solution for this project. Agile development was chosen for its ability to adapt to changing requirements during development (Beedle et al., 2001). This methodology is particularly effective for computer game development, as features often have to be implemented before their appropriateness can be evaluated. One example of how the agile methodology proved useful during development is the shift from an open world to a linear, infinitely generating level.

Iterative development. The project naturally split into three major components: level generation, game content, and dynamic difficulty adjustment. The separation of these components lent itself well to iterative development, as it was possible to work on each component independently of the other two. Furthermore, the components lead into each other: first create the game content, then generate it procedurally, then model the player experience and feed that value back into the procedural generation. Due to their relatively large scope, level generation and game content were split into multiple iterations. Iterative development consists of a cycle, over a period of around a month or less, consisting of four phases: planning, which consists of identifying functional requirements and risks; design, the development of solutions to satisfy the requirements; implementation of the design with frequent tests; integrating the implementation with the rest of the code base.

8.2 Risk Management

Time management risks. Reconciling the project workload with other modules could prove to be difficult, especially in the second term, credit split weighted heavily on the second term. In order to mitigate this risk, minimal functionality was aimed to be complete by the start of the second term. In addition to this a project schedule was designed during the project proposal phase (Appendix A) and all attempts were made to stick to this.

Third party risks. Reliance on the Unity engine introduced the risk of potential bugs introduced with newly released updates. To this end, it was decided to use only the known stable Unity 4.6 and only update to Unity 5 once any game-breaking issues were resolved. A backup of the project was created before upgrading to Unity 5.

Project requirement evolution. Changes to the functional and non-functional requirements could result in a significant time delay. In order to mitigate this risk, agile methodology was adopted.

9 Evaluation

9.1 Strategy

User playtesting was used for product evaluation. Areas evaluated were: the success of the dynamic difficulty adjustment; the player's engagement with the game itself; any unidentified bugs that resulted from user input. A playable version of the game was distributed online and advertised on a variety of game and game development communities, as well as social networks, over a period of two weeks. Two versions of the game were supplied: one version featured the dynamic difficulty adjustment; one version simply increased the difficulty linearly with time. Play testers were required to play both versions of the game and then fill out a brief anonymous questionnaire. The questionnaire evaluated, as a value of one to five, the participant's agreement with the following statements:

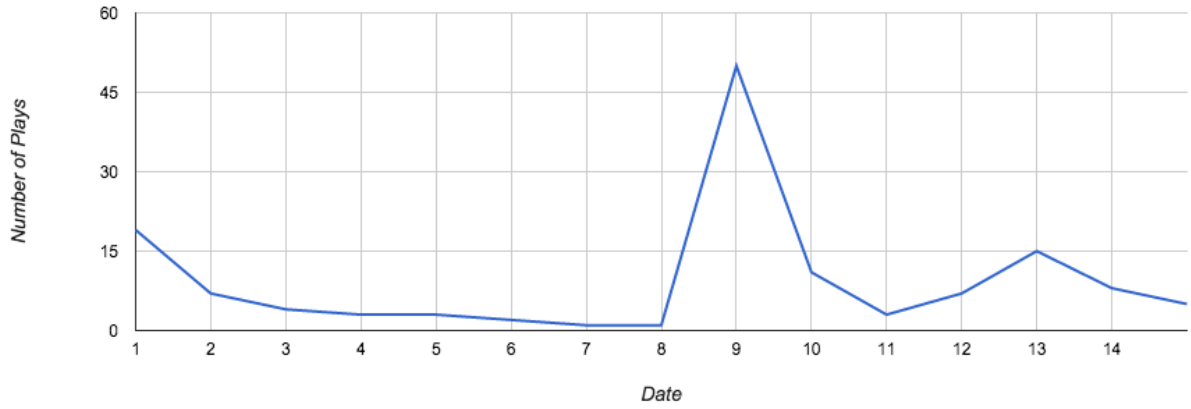
1. The timed version of the game better fit my skill level than the non-timed version of the game.
2. The timed version of the game was more fun than the non-timed version of the game.
3. When I died in the timed version of the game, I felt that it was justified.
4. When I died in the non-timed version of the game, I felt that it was justified.

The dynamic difficulty adjustment version of the game was referred to as the 'timed' version of the game to avoid confusing the playtester. The aim of the dynamic difficulty adjustment was to better fit the difficulty of the game to the skill level of the player, so the player was asked directly if they believed this was the case. However, play testers might not necessarily be aware that this is the aim of the dynamic difficulty adjustment, so a more general quantification of 'fun' was included as well. Players were asked if they felt their deaths were justified, in order to quantify how fair the game felt, and whether the player was just dying as a result of bad controls, in order to evaluate the success of the game content. Testers were also presented with a text box to submit general feedback. The full questionnaire can be found in Appendix B.

9.2 Results

By the end of the testing period, around 150 players had participated in playtesting. Around 25 of those playtesters were referred from the author's *Facebook* page, so fortunately the majority of playtesters were less biased in the game's favour and could give honest feedback. Hosting the game on the popular website *Itch.io* meant players surfing the 'new submissions' page of the site could discover the game on their own, but the largest portion of traffic came from advertising the game on content aggregation website *Reddit*.

Figure 30: Number of players vs. date over the evaluation period.



Questionnaire answers were averaged to gain an agreement metric for each question. Resulting values ranged from one to five. Any value above three was considered to be enough agreement to be successful. It was predicted that playtesters would agree that the timed version of the game would be more fun, and better fit their skill level, and that players would feel the timed version of the game was more fair. The resulting values were:

1. The timed version of the game better fit my skill level than the non-timed version of the game: 3.25
2. The timed version of the game was more fun than the non-timed version of the game: 3.64
3. When I died in the timed version of the game, I felt that it was justified: 3.55
4. When I died in the non-timed version of the game, I felt that it was justified: 3.72

The results confirmed the prediction that players found the dynamic difficulty adjustment more engaging than the control version of the game. It was not expected that testers would find the non-timed version of the game more fair than the timed version. One possible reason for this discrepancy could be communication of the game mechanics to the player. In the general feedback results, 30% of the participants reported that they did not understand how the timed version of the game's score was calculated.

10 Discussion

The project achieved its minimum functional and non-functional requirements. However, none of the extension requirements were met. While this did not make the project unsuccessful, I am not completely satisfied with the amount of variety in the level generation. I would have liked to employ a wider variety of obstacle placement algorithms to create more interesting groups of obstacles such as tunnels, slaloms, or other more visually interesting areas. Obstacles could have been placed around pre-defined player paths within the major flight path, and flight path gradient could have been increased with difficulty to create a more challenging experience.

One of the main problems encountered during development was the lack of a clear desired result for the procedural generation. While this allowed me to adopt an agile methodology and experiment with a variety of level generation techniques, and while most of the work done (spline representation, spline following) during this experimentation eventually found use in the final product, I feel like without this wasted time the final product could have been more substantial.

Working with an existing game rather than developing a bespoke game could have clarified the project. I could have adopted a similar approach to Pedersen et al. and worked with an existing game. Attempting to procedurally generate towards a known result rather than developing the game and generation in parallel would have given me a clearer goal. Furthermore, working with an existing game would have simply given me more time to focus on procedural generation and dynamic difficulty adjustment, the main focuses of this project. While the use of the Unity engine certainly helped reduce the workload of developing the game content, this was not an area of the project that I consider as significant in terms of technical difficulty as PCG or DDA, but still required a significant time investment. If I did the project again, I would decide on a world generation style within the first two weeks of the project's inception, and stick to it, or possibly pick an open source version of a well-established video game and modify that with my own procedural generation. Additionally, I would focus less on game aesthetics as issues with stray vertices were the cause of frustrating delays and of no technical significance.

I was pleased with the results of the project evaluation. Evaluation supported the hypothesis that dynamic difficulty adjustment could make a game more appealing to the player and better fit their skill level. If I had the opportunity to redo the evaluation, I would ask the player how familiar they were with computer games in general. Submitting the game mainly to games development communities was the best way of getting well informed feedback from genuinely interested playtesters, but, it limited the testing audience mainly to players who were already familiar with the computer game medium. It could have been more valuable to test the game with a variety of player skill levels, but, it is easier to convince users who are familiar with games to playtest a game.

Possible extensions The level generation could possibly have been expanded to generate a representation of an audio stream, as it is already possible to place obstacles along the spline at a known time value. If an audio stream could be analysed to identify the time value of beats, these could fairly easily be translated into t values along the curve. I would also have liked to experiment with more difficulty progression functions, possibly developing a difficulty progression editor that could be used by game designers to hand craft custom difficulty progression functions. Given more time, I would like to use machine learning to quicker settle on an appropriate difficulty level.

Possible future applications As established in section 1.2, advancements in the field of computer games can be applied to other domains through the use of gamification. I would especially be interested to see dynamic difficulty adjustment and procedural content generation applied to the field of education. In the same way that game players nowadays have a wide range of skill levels and backgrounds, children in school and all people in education learn at different paces and respond differently to a variety of teaching methods (Christensen et al., 2008). To be able to adapt a child's learning process to their individual learning pace could cause a marked improvement in education quality. Furthermore, the field of education is suited to dynamic difficulty adjustment, given a large set of data in the form of grades, if the data can be collected and analysed correctly an accurate metric of ability could be generated.

11 Conclusion

The project achieved its aims and all the minimum functional and non-functional requirements. The system generated a smooth and consistent flight path and moved the player along it. The system successfully populated the flight path with obstacles in a way that was fun and interesting to the player, and never created impossible challenges. The dynamic difficulty adjustment successfully tailored the level generation to each player's individual skill level. Overall the game was well received during evaluation, and was played by a large number of individuals. I consider the project a success.

References

International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2012, Los Angeles, CA, USA, August 5-9, 2012, Talks Proceedings, 2012. ACM. ISBN 978-1-4503-1683-5. URL http://dl.acm.org/citation.cfm?id=2343045.

Acornsoft. Elite. Software, 1984.

Chris Bateman and Richard Boon. *21st Century Game Design (Charles River Media Game Development)*. Cengage Learning, 2005. ISBN 1584504293.

Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. Web page, 2001. URL http://agilemanifesto.org/.

Terry Cavanagh. Super hexagon. Software, 2013.

Jenova Chen. Flow in games (and everything else). *Commun. ACM*, 50(4):31–34, 2007. doi: 10.1145/1232743.1232769. URL http://doi.acm.org/10.1145/1232743.1232769.

Dave Chenell. Obstacle avoidance ai. Web page, 2014. URL http://thewaterbear.com/obstacle-avoidance-a-i/.

Clayton M Christensen, Michael B Horn, and Curtis W Johnson. *Disrupting class: How disruptive innovation will change the way the world learns*, volume 98. McGraw-Hill New York, NY, 2008.

Mihaly Csikszentmihalyi. *Flow*. Springer, 2014.

Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1):1:1–1:22, feb 2013. ISSN 1551-6857. doi: 10.1145/2422956.2422957. URL http://doi.acm.org/10.1145/2422956.2422957.

Robertson Holt and J Mitterer. Examining video game immersion as a flow state. *108th Annual Psychological Association, Washington, DC*, 2000.

Caleb Howard and Carlos Lemus. Asking the impossible on SSX: creating 300 tracks on a ten track budget. In *International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2012, Los Angeles, CA, USA, August 5-9, 2012, Talks Proceedings* DBL (2012), page 33. ISBN 978-1-4503-1683-5. doi: 10.1145/2343045.2343090. URL http://doi.acm.org/10.1145/2343045.2343090.

Martin Jennings-Teats, Gillian Smith, and Noah Wardrip-Fruin. Polymorph: A model for dynamic level generation. In Youngblood and Bulitko (2010). URL http://aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/view/2150.

Ichiro Lambe. Procedural content generation: Thinking with modules. Web page, 2012. URL http://www.gamasutra.com/view/feature/174311/.

Chris Pedersen, Julian Togelius, and Georgios N. Yannakakis. Modeling player experience in super mario bros. In *Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Games, CIG 2009, Milano, Italy, 7-10 September, 2009*, pages 132–139, 2009. doi: 10.1109/CIG.2009.5286482. URL http://dx.doi.org/10.1109/CIG.2009.5286482.

Ken Perlin. Improving noise. *ACM Trans. Graph.*, 21(3):681–682, July 2002. ISSN 0730-0301. doi: 10.1145/566654.566636. URL http://doi.acm.org/10.1145/566654.566636.

Markus Persson. Terrain generation, part 1. Web page, 2011. URL http://notch.tumblr.com/post/3746989361/terrain-generation-part-1.

Adrien Peytavie, Eric Galin, Jérôme Grosjean, and Stéphane Mérillou. Procedural generation of rock piles using aperiodic tiling. *Comput. Graph. Forum*, 28(7):1801–1809, 2009. doi: 10.1111/j.1467-8659.2009.01557.x. URL http://dx.doi.org/10.1111/j.1467-8659.2009.01557.x.

Stephen E. Siwek. Video games in the 21st century: the 2014 report, 2014.

Kenneth O Stanley, Bobby D Bryant, Igor Karpov, and Risto Miikkulainen. Real-time evolution of neural networks in the nero video game. In *AAAI*, volume 6, pages 1671–1674, 2006.

Yo Takatsuki. Cost headache for game developers. Web page, 2007. URL http://news.bbc.co.uk/1/hi/business/7151961.stm.

T. L. Taylor. *Play between worlds: exploring online game culture*. MIT Press, Cambridge, Mass, 2006. ISBN 9780262201636.

.theprodukkt. .kkrieger. Software, 2004.

Julian Togelius, Tróndur Justinussen, and Anders Hartzen. Compositional procedural content generation. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, PCG'12, pages 16:1–16:4, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1447-3. doi: 10.1145/2538528.2538541. URL http://doi.acm.org/10.1145/2538528.2538541.

Georgios N. Yannakakis and Julian Togelius. Experience-driven procedural content generation. *T. Affective Computing*, 2(3):147–161, 2011. doi: 10.1109/T-AFFC.2011.6. URL http://doi.ieeecomputersociety.org/10.1109/T-AFFC.2011.6.

G. Michael Youngblood and Vadim Bulitko, editors. *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2010, October 11-13, 2010, Stanford, California, USA*, 2010. The AAAI Press.

A Project Schedule

Week Starts	Plan and Important Dates
06/10/2014	Read around the subject, examine methods employed by existing videogames, produce some small prototypes to test ideas and possibly adapt the functionality specification. Make any necessary changes to Project Proposal and finish by 11/10/2014.
13/10/2014	**16/10/2014 : Project Proposal Deadline** Read around the subject, examine methods employed by existing videogames, produce some small prototypes to test ideas.
20/10/2014	**23/10/2014 : Ethics Self-Assessment Deadline**
27/10/2014	Begin coding with game content iteration.
03/11/2014	Week 2 of coding.
10/11/2014	Week 3 of coding.
17/11/2014	Week 4 of coding.
24/11/2014	Week 5 of coding. Aim to begin level generation iteration this week.
01/12/2014	Week 6 of coding.
08/12/2014	**Prepare for inspection week:** Produce software-testing for all aspects of the project that are vital to the project inspection. Build a version of the program appropriate for the inspection.
15/12/2014	Week 7 of coding.
22/12/2014	Week 8 of coding.
29/12/2014	**Halfway point.** Week 10 of coding. Level generation should be implemented with Aim to begin dynamic difficulty adjustment iteration this week.
05/01/2015	Week off from coding. Do some user testing.
12/01/2015	**12/01/2014 : Spring Term Begins.** Work on removing any remaining errors and polishing necessary functions to get the project into a presentable state for demonstration week. Start working on a draft of the final report.
19/01/2015	Continue working on necessary functions, and if finished, begin work on optional functions. By the end of this week, should have an instance of the project built as an application that would be appropriate for the demonstration. Optional function to prioritise: on the fly map generation.
26/01/2015	Continue working on optional functions and final report.
02/02/2015	Optional functions, final report. This is a buffer week that has been kept relatively empty to allow for any previous deadlines overrunning.
09/02/2015	Aim to finish a draft of the final report by the end of this week. Start working on the non-application aspects of the demonstration: a presentation and a loose script (should take a few days at most).
16/02/2015	Mainly a buffer week. Should finish optional features and preparations for demonstration.
23/02/2015	**23/02/2015 : Demonstration Week**
02/03/2015	Finish final report.
09/03/2015	**Project Deadline.** Finish final report.

B Questionnaire

Playtesting Questionnaire

Thanks for playtesting my dissertation! Below are a few questions comparing the two versions of the game you (hopefully) played. Please indicate your level of agreement with the following statements,

If you didn't play the second version of the game, please return to http://jamesmoulang.itch.io/infinite-flyer and press N to switch to the mode with no timer, and play that mode at least once.

* Required

The timed version of the game better fit my skill level than the non-timed version of the game. *
Please indicate your level of agreement with this statement.

	1	2	3	4	5	
Disagree completely	○	○	○	○	○	Agree completely

The timed version of the game was more fun than the non-timed version of the game. *
Please indicate your level of agreement with this statement.

	1	2	3	4	5	
Disagree completely	○	○	○	○	○	Agree completely

When I died in the timed version of the game, I felt that it was justified. *
Please indicate your level of agreement with this statement.

1	2	3	4	5
○	○	○	○	○

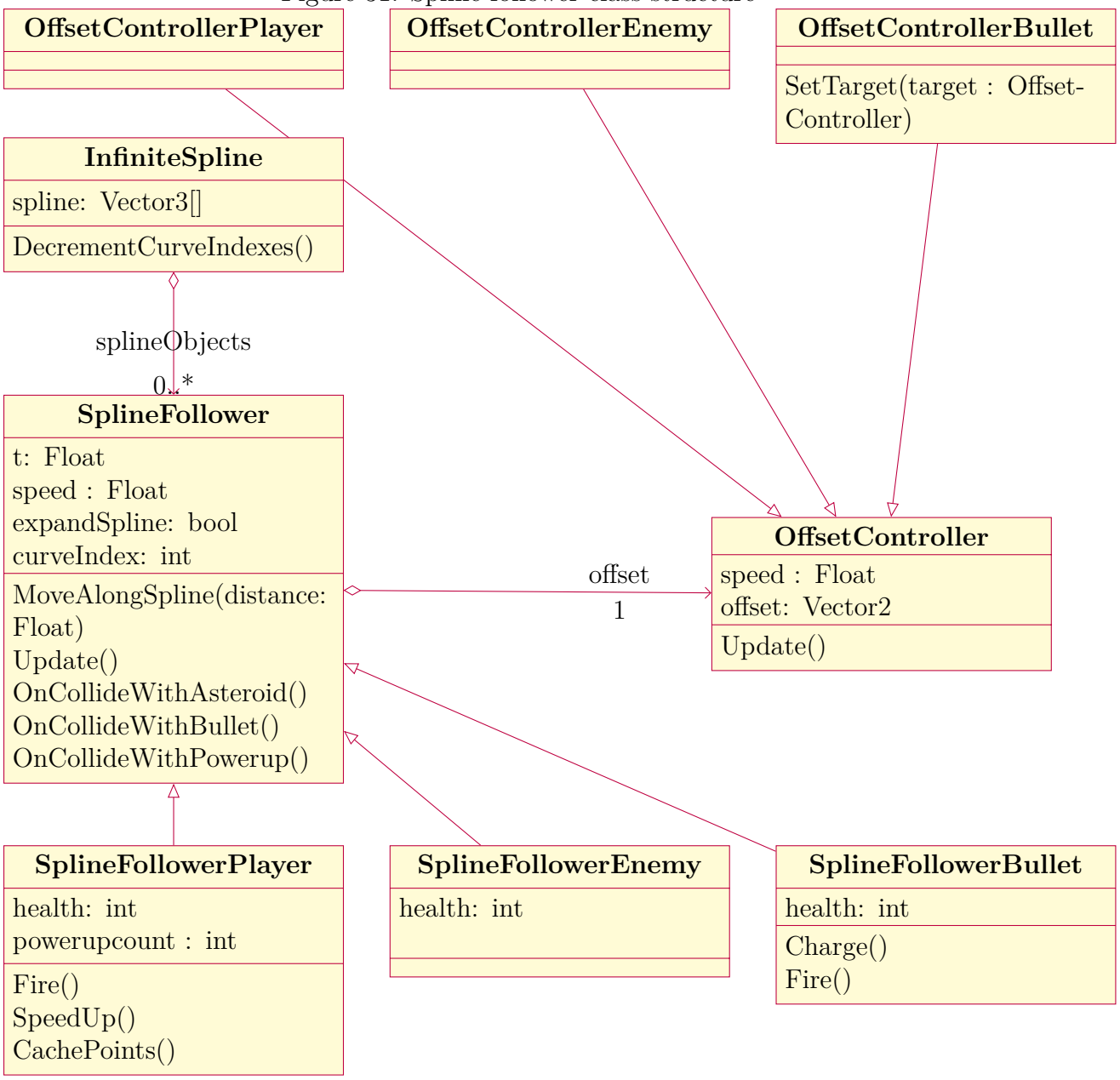
When I died in the non-timed version of the game, I felt that it was justified. *
Please indicate your level of agreement with this statement.

1	2	3	4	5
○	○	○	○	○

C UML Diagrams

C.1 Spline Followers

Figure 31: Spline follower class structure



C.2 Spline Object Pools

Figure 32: Spline object pool class structure