Dynamic Difficulty Adjustment in Procedural Content Generation

Senior Project submitted to
The Division of Science, Mathematics, and
Computing of Bard College

By Charles Calder

Annandale-on-Hudson, New York

December, 2017

Abstract

The world of game development has gone through many chapters since its birth. Many complex techniques, designed to give every player a personal game experience, have been developed by those who love to create. This project explores modern Procedural Content Generation and Dynamic Difficulty Adjustment techniques. The algorithm developed in the course of this project is designed to increase developer and player quality of life. Through adaptive generation, this project will explore how players engage with games and what developers can do to add to the player experience.

Acknowledgements

I would like to thank Robert McGrail for advising me through this project
and Keith O'Hara for sparking my interest in computer science.
I would also like to thank my parents and my brothers
for their constant and reliable love and support.

Dedication

I dedicate this to my best friends, Jasmine Clarke and Elias Posen.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

This paper explores three topics. The first two topics are Dynamic Difficulty Adjustment and Procedural Content Generation. The third topics discussed in this paper is the effect that difficulty adjustment algorithms have in procedurally generated environments and how that adaption affects player engagement and skill. There has been a large amount of research into elements of all three of these fields independently.

As games have become more complex there has been a growing amount of research into the algorithms that control the continuous generation and adaption of game environments. There has not, however, been much research into the combinations of these fields. Analyzing player engagement and skill in virtual environments is a thoroughly explored topic in the world of game research, although it is extremely broad and difficult to define. This paper describes the process of expanding the world of research on these topics.

The topic of Dynamic Difficulty Adjustment using Mihaly Csikszentmihalyi's Flow theory

recently has been a powerful concept in the video game research world [3] [1] [5]. Active Dynamic Difficulty Adjustment provides players with conscious choices to help them directly customize unique and optimal video game experiences. The implementation of difficult adjustment in this form is closer to a game design element than a data analysis element. Instead, this research concentrates on passive difficulty adjustment, which involves developing a data analysis algorithm that focuses on gathering, analyzing, and utilizing in-game player data. This paper will grow the concept of Dynamic Difficult Adjustment algorithms by incorporating them into procedural generation and how this affects player engagement.

Defining Procedural Content Generation as a game design element is also a common research topic [7] [2]. In many research papers, adaptive or player-driven procedural generation is also commonly explored. There has not been, however, much explortion further into games with deterministic adaptive Procedural Content Generation on a second-to-second scale. This is most likely because doing so requires building a game designed around this concept, considering there are no popular examples of this to draw from. Although exploration of Procedural Content Generation algorithms has been done in depth, modern research most often focuses on defining what is and what is not Procedural Content Generation, and thus the effect that Adaptivity in Procedural Content Generation has on player engagement or skill has not been discussed thoroughly. This paper will extend the definition of Procedural Content Generation by introducing new design elements that have not been looked at closely in a research environment.

The research around analyzing player engagement has had more time to develop than the other two, but is a little more difficult to pin down [6] [4] [5]. The concept of forming a fully realized model of player engagement in games is often done by monitoring some number of elements of user interaction with games. Common types of tracking variables are concentration, challenge, skills, control, clear goals, feedback, immersion, etc. The problem with analyzing engagement is that each player interacts with a game in their own way and trying

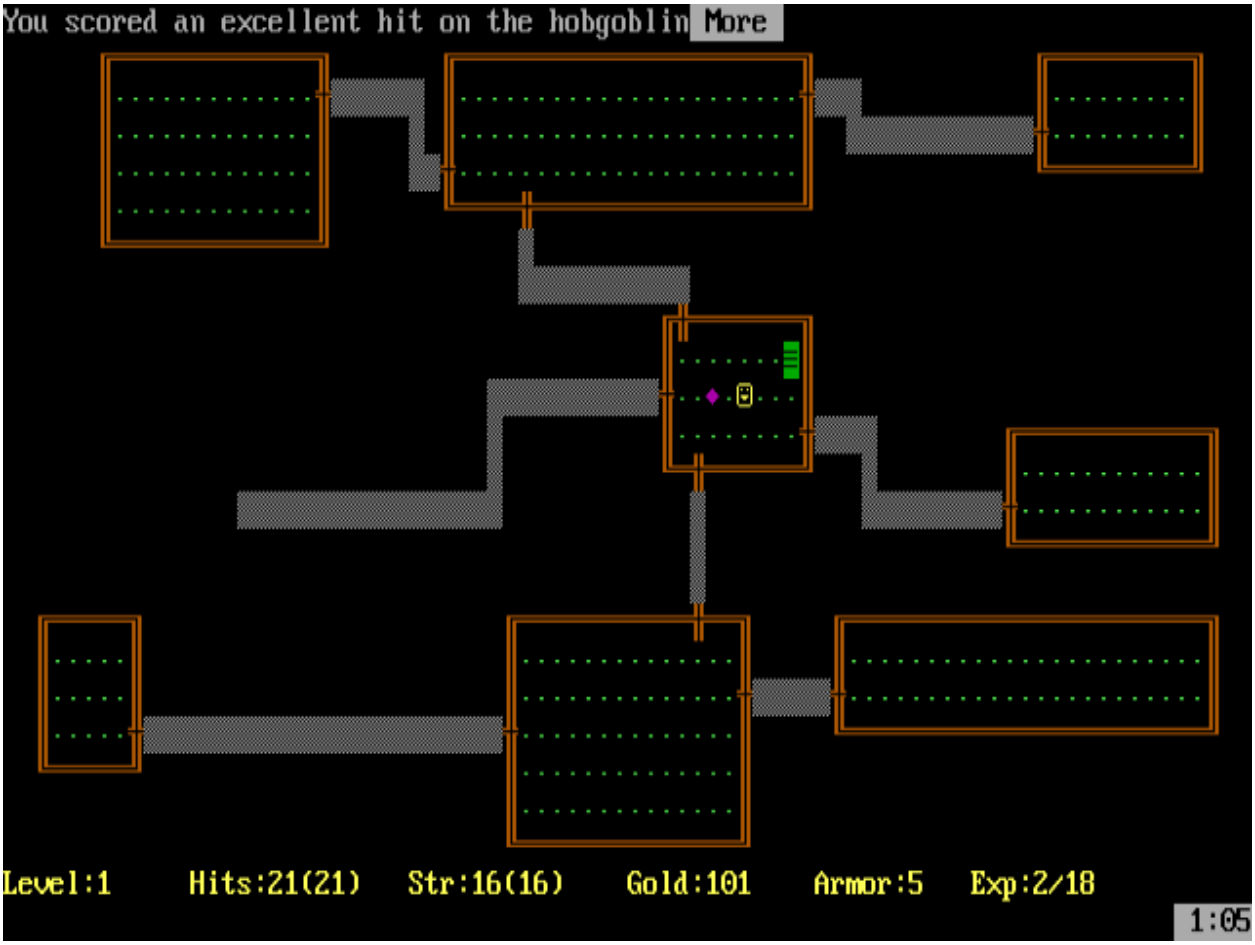
to simplify that into understandable data can be very difficult. Most modern research looks into monitoring engagement very generally, but not into monitoring engagement through the context of specific genre or game style. This paper will help define engagement analysis by building a player model through the context of players engaging only with game Flow in Procedural Content Generation using Dynamic Difficulty Adjustment.

This paper details the process of combining all three of these concepts through a single project. This is done by exploring each of these concepts independently, then bringing them together to build a game. The Procedural Content Generation in this game is designed based on descriptions outlined in "What is Procedural Content Generation? Mario on the Borderline" [7]. The Dynamic Difficulty Adjustment used is designed based on the Data Cycle concept, which is detailed in "Flow in Games" [1]. Finally, the engagement analysis is based around the Sweetser and Wyeth concept model for evaluation engagement and enjoyment in games [4]. The following sub-sections define in detail and discuss the theory behind a few of these topics.

1.2 Procedural Content Generation

Procedural Generation is a generation technique that involves infinite, automatic, and artificial creation of content using simple algorithms. **Procedural Content Generation** (PCG) in games is the process of generating game content using these procedural generation algorithms. The most well-known forms of PCG in games are the cave or dungeon PCG in *Rogue* (AI Design 1980) (Figure 1.1), which inspired generations of games with similar PCG techniques, the PCG technique to distribute weapons, armor, and other items throughout the game area in *PlayerUnknown's Battlegrounds* (Bluehole 2017), and the environment and enemy PCG used in *MineCraft* (Mojang 2011).

Figure 1.1: Screenshot of Rogue Procedural Dungeon Generation



Those who know these games will immediately see that PCG is an extremely broad term and there is no way to define it exactly. There are many differing opinions on multiple aspects of the definition of PCG. People in different fields of Computer Science might not even agree on what "content" is, and would probably agree even less on which generation techniques are considered procedural generation [7]. Content generated can also vary widely in its affect on games, from simple texture generation, that has no direct effect on user-game interaction, to item or map generation, which can control every step of the player's navigation through a game. For the purposes of this project, however, we will now take a moment to specify elements of language that will be used for the rest of this paper. First, the term PCG will refer specifically to PCG in games, second, we will use the definitions laid out in [7] as the standard for what constitutes a correct form of PCG, and third, the term

Interactable Object (InOb) will refer to any game content generated by the procedural generation algorithm developed for this project.

Modern PCG research often attempts to untangle the complex question: What is and what is not PCG? We will start with an easy definition. PCG is not player or designer created content, online or offline. This refers to anything designed by a user or game designer and statically placed in the game environment for players to interact with. This is an obvious but necessary first step in the definition of PCG. Next, the concept of randomness and how it relates to PCG. To say that a game is generated randomly does not mean random in the pure sense that most people imagine, complete and uniform randomness. Instead, random generation in the context of games refers to generation that follows some set of rules defining which types of content can be generated when and where. At each iteration of generation a random, **Prefabricated Item** (Prefab) is chosen from a set of items allowed by pre-determined rules. This is the most commonly accepted form of PCG across fields. The process of PCG within this definition can still vary widely, but is enough to satisfy the definition of PCG for the context of this project.

A very important definition in the context of this paper is that of adaptive or parameterized PCG, specifically deterministic and continuous adaptive PCG. This, as the name implies, is PCG that adapts based on parameters given by the player, passively or actively. Unfortunately, there are no usable examples of PCG with deterministic continuous adaptivity in recent research. This hole in the genre of PCG research is where the this paper comes in. Through the definitions of PCG laid out by Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N. Yannakakis, a common language can be used to look deeper into what goes into adaptivity in PCG. In the following chapters this paper will extend their research into PCG by exploring the possibility of creating a game that brings to life their imagined example of a game with adaptive PCG.

1.3 Dynamic Difficulty Adjustment

Dynamic Difficulty Adjustment (DDA) is defined in multiple researcher papers and is a relatively new concept being explored in games [1] [5]. DDA in games is a rather simple concept to describe abstractly. The use of DDA often is as follows, games should increase or decrease in difficulty to match the player's skill level, eventually finding an equilibrium level of difficulty that fits the player's ability. DDA is built heavily on Mihaly Csikszentmihalyi's theory of Flow. **Flow** describes a general feeling of total immersion in the current activity, brought about by a perfectly balanced sense of challenge and accomplishment. DDA is a design element that game developers and researchers have explored heavily in order to automatically control a game's Flow around the actions of a player.

There are many examples of games that use DDA, some of the most popular examples are *Crash Bandicoot* (Sony Computer Entertainment 1996), a platformer that used passive DDA as a design element to aid weaker players without changing the game for more advanced players, *flOw* (Sony Computer Entertainment 2006) who's application of active DDA popularized the concept of incorporating mental immersion into gameplay, and *Fallout: New Vegas* (Bethesda Softworks 2010) which used a combination of active and passive DDA to generate tougher variants of enemies or enemies with higher statistics and better weapons as the player performed better in the game.

DDA can be used is a very wide variety of applications. To describe DDA in a more tangible sense, there are two basic forms that all DDA implementations can be categorized into. **Active DDA** usually falls along these lines: Over some iteration (level-to-level, room-to-room, etc.) the player is given choices with *clear* difficulty distinctions. Most often these different choices are paired with different rewards relative to the level of difficulty. This style of DDA is always built on a discrete scale rather than a continuous one, simply by virtue of the fact that players must make decisions that directly affect the current game's difficulty.

Through active DDA, designers are able to give the player direct control over their own Flow. Perfect active DDA gives the player the chance to make choices to affect the games rate of difficulty, and gives the player these choices very frequently. Through discrete, active DDA, designers often produce concepts that allow games to mimic continuous, passive DDA without involving background algorithms. This implementation allows the player to feel a sense of control over the product and the designer to have full control over the game design without having otherwise necessary knowledge of more complex algorithms.

The second category of DDA is **Passive DDA**, which usually falls into one type of pattern. In this paper, this pattern will be referred to as the **Data Cycle**. In games, the Data Cycle is a method of data analysis in which raw data is gathered on a player while they play, the data is processed in the background, the processed data is used to affect something in the game, and the cycle repeats. This style of DDA is chosen such that the players are never aware of the game adjusting itself, and thus are allowed to become fully immersed in a game without having to guide the Flow of the game. In the context of this project, the Data Cycle will be used as a tool to bridge between PCG and DDA. Through the Data Cycle methodology, DDA will gather in-game data and use it to adjust content generation.

This project will also use the **Rubber Band AI** [5] methodology to structure the effect of the Data Cycle adjustment. Rubber Band AI is an Artificially Intelligent difficulty adjustment technique which, in concept, virtually holds the player and the player's AI opponents together by a rubber band as the player interacts with the game. If the player pulls in any direction, the 'rubber band' makes sure that the opponents are pulled in the same direction. Rubber Band AI is most often associated with racing games or sports games, in which there are AI opponents that the player is facing off against. In these games, the 'rubber banding' is implemented by dynamically updating the ability of the AI so that the player constantly feels as though they are only just beating the AI. In this project, the DDA method is designed to use Rubber Band AI as the platform for the Data Cycle. The player will 'pull' the rubber

band until they find an equilibrium that satisfies their style of play. This way, there are no discrete adjustments according to pre-defined categories of player types.

In many games that use passive DDA there is often an alpha phase of the game development cycle, in which the game is play-tested and the difficulty is manually adjusted to be just right. From this, the game holds a baseline player style to which all players of the game are then compared to. Although this does require an extra round or two of play-testing and fiddling with the game design, it is a simple and effective solution to the concept of simultaneous player data analysis and game adjustment. Through this initial modeling, the Data Cycle and Rubber Band AI methods have a way to compare the current player's game data. This method of finding an initial non-preferential model was used in this project in order to expedite the development process.

The form of DDA that will be used in this project is influenced by [5], [3], and [7]. The DDA created for this project is a passive DDA built on the Data Cycle and Rubber Band AI concepts with an initial round of play-testing to create a model of pre-established data to which players can be compared. This project will build on the current research of DDA by incorporating all of these concepts to influence PCG.

1.4 Unity

A **Game Engine** is an Integrated Development Environment built for game developers and designers that allows them to create games and release them to many different platforms. There are many free game engines that exist for beginner to advanced game designers. It is also possible for game developers to build their own game engine as they build their game. This process, however, usually takes a much longer time than was available to build a game and run tests on it for this project. It would have also been over-kill to design an entire

game engine around a game as simple as the one created for this project.

Unity is a cross-platform game engine developed by Unity Technologies and initially released in June of 2005. Unity is primarily used to develop video games and simulations for computers, consoles and mobile devices. It supports both 2-dimensional and 3-dimensional game development and is written in C#. Unity is known for its ease of use and quick development cycle, but there are many big titles that have come out of Unity, including, Cuphead (StudioMDHR 2017) (Figure 1.2), Hollow Knight (Team Cherry 2017), and Kerbal Space Program (Squad 2015).

Figure 1.2: Cuphead Artwork

The game developed for this project was built in Unity. All assets and scripts were created or written by the researchers for the purpose of exploring the concepts laid out in this chapter with the exception of a single object model. The game was developed for and built in the macOS environment and not for any other platform.

1.5 Previous Research

In [8], Ricardo Lopes, Elmar Eisemann, and Rafael Bidarra write on topics that are extremely relevant to this project. Their research was published in March of 2017 and focuses on adaptivity in procedurally generated environments. This project is mainly inspired by the algorithms created for the player modeling and generation adaption in [8] because they were able to demonstrate that PCG could be used in a more player-centric manner. There are a couple of different issues with their paper that this project attempts to reconcile. First, the authors of that project decided to categorize players into only two categories. This project attempts to show that, even though the accuracy of the model is inversely proportional to the number of categories, using only two categories over simplifies the data and can cause underfitting. Second, the authors built the game to use non-dynamic player modeling, meaning that modeling does not occur while the player interacts with the game, only in between play-sessions. This project attempts to show that through dynamic modeling and adaption in PCG, player engagement is much more likely to stay at a high, constant level.

Chapter 2

Methods

2.1 Design of game

Overview

This section details the overall design of the game. The user experience, the user interface, and the general game play of the project are inspired by the infinite-runner and stunt-driver genres. Most of the following game elements designed independently of the test, which will be detailed in the next chapter. The following sub-sections describe how the PCG and DDA were implemented into Unity.

Player Controls

The player controls a car. While the car has all four wheels on the ground, the player can accelerate forwards or backwards, turn left or right, hand-brake (for tight turns), and jump. While the car is in the air, the player can manipulate the car's rotation. The player can

control car's pitch, causing the car to flip forward or backward, and control the car's yaw, causing the car to spin. These in-air controls are how the player performs tricks. A full rotation around either the car's Z or X axis gives extra points to the player. These are all of the actions available to the player to advance through the game.

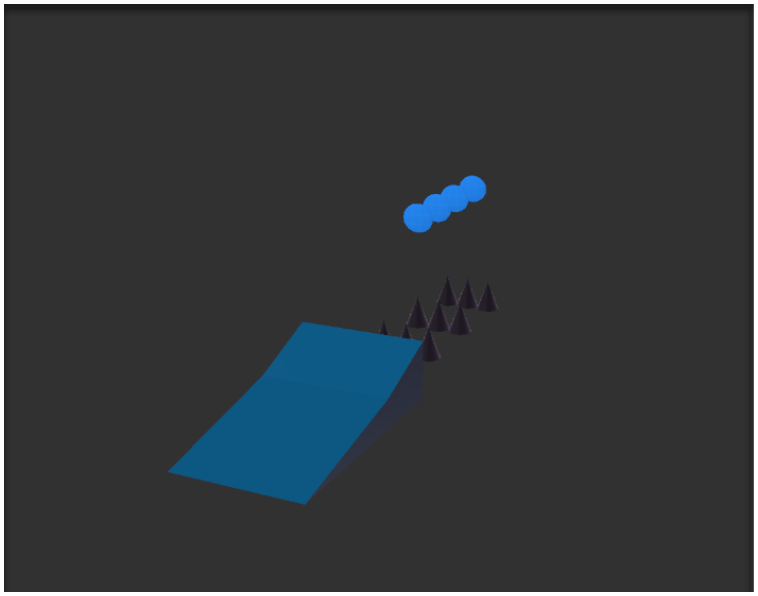
This control setup is very simple for two reasons. First, using a simple control layout means that there is less time spent by testers learning the game and more time playing at a competent level. This way more data that reliably reflects the growth of the player's skill at the game can be extracted in a shorter period of time. Second, simple controls means that there are fewer dimensions of data that need to be tracked. The more dimensions of data that need to be watched, the more difficult it is to evaluate the data, and the more difficult it is to implement the DDA Data Cycle. Thus, in order to hit the Data Cycle's qualifiers of low dimensionality and high reliability in the data extracted from play sessions, the game is designed with simple, easy to learn controls that still allow for the player's skill and personal style to be represented through the gameplay.

The concept that the controls are simple and easy to learn was reflected by the testers of the game. Testers were asked in questionnaires after playing the game to rate on a scale from zero to ten how easily they learned the controls. The mean of the answers is 7.78 and the standard deviation is 2.44 meaning that many more testers found the controls easy than difficult.

Interactables

There are twelve InObs, and thirteen unique objects overall that can be generated by the game. For an object to be an InOb means that it is designed specifically for the player to interact with and gain points from it. There are two ways of gaining points from object interactions, performing tricks and collecting **Orbs**, which are small spherical objects placed

Figure 2.1: Ramp Interactable (Level 1 Example)

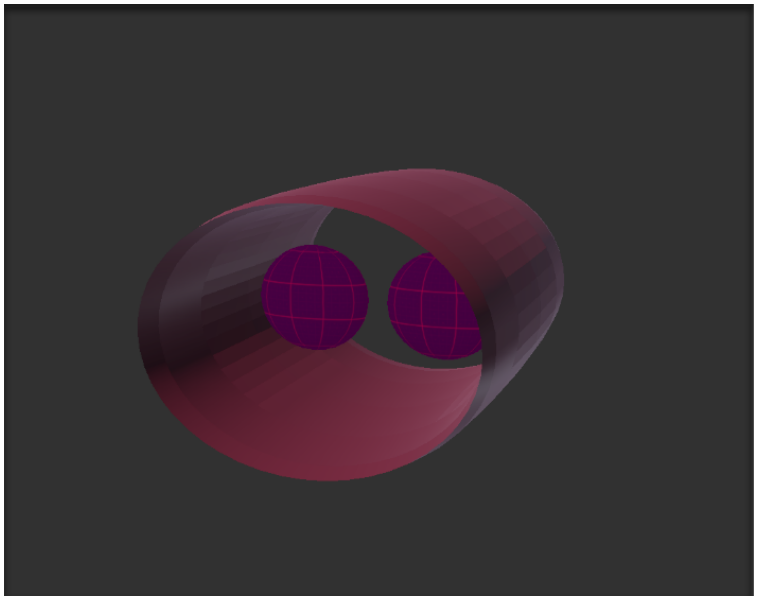


in the Prefabs of each InOb. The non-InOb that can be generated is a spike strip and is only designed to 'kill' the player, by removing all InObs from the currently generated game area and reseting the players position back to (0, 0, 0).

There are four classes of InObs called **Types**. Each Type allows for a different kind of interaction with the player. The Ramp Types (Figure 2.1) are self-explanatory, they are ramps up which the player can drive to collect Orbs and, if their car is going fast enough, get enough time in the air to perform tricks. The Speed Boost Types (Figure 2.2) can be described as gates through which the player can drive. When a player drives through a Speed Boost, they collect Orbs and their car is given a big boost of forward speed. The Destructible Wall Types (Figure 2.3) are large walls that the player can drive through, knocking them into pieces and collecting Orbs in return. The Spike Pit Types (Figure 2.4) are similar the the spike strip except that they are smaller and have Orbs floating above them. The player can jump over the Spike Pit Types to collect the Orbs and, if they have time, perform a trick.

Each Type contains three subclasses, called **Levels**. Each Level is more difficult and re-

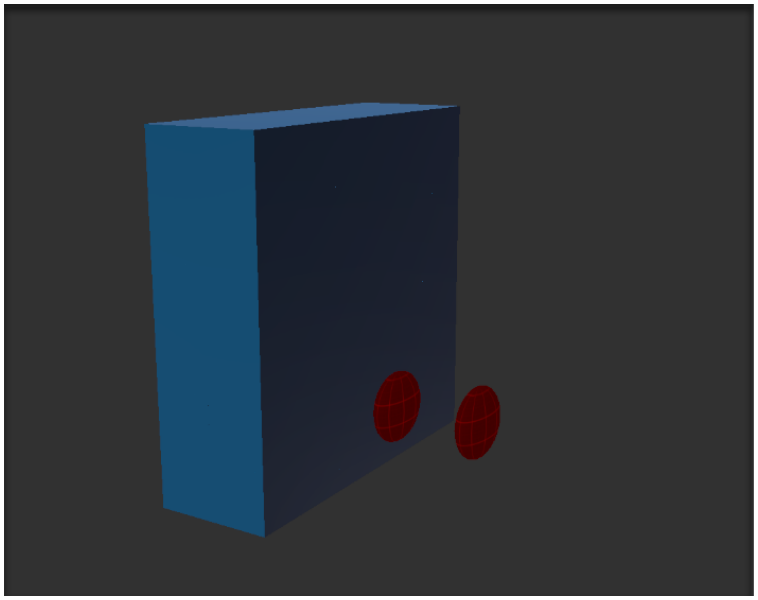
Figure 2.2: Speed Interactable (Level 2 Example)



warding than the previous. The different Levels are used in combination with the Rubber Band design of the DDA to control game Flow. The Type and Level categorization of these different InObs is designed to aid exploration of PCG and DDA.

In all modern games in which players must use quick reactions and pattern memorizing in order to progress, gain points, and avoid death/damage, there must be classes of object or enemies that are generated as obstacles for the player to overcome. This is essential to game Flow because the balance of keeping a player engaged involves always simultaneously giving the player new, challenging obstacles while also allowing them to learn, understand, and overcome previous challenges that they have seen multiple times before. By creating these classes and subclasses of InObs to which the player must learn how to react, the influence that controlling the game's Flow has on a player's ability to learn these classes and subclasses can be isolated and captured. In order to procedurally generate anything, there must be a Prefab version waiting in the game assets to be called upon. In order to fully explore modern PCG techniques this project was tested using multiple generation styles until the finding best style for the game's variation of Prefab assets. When using DDA, there must

Figure 2.3: Wall Interactable (Level 1 Example)

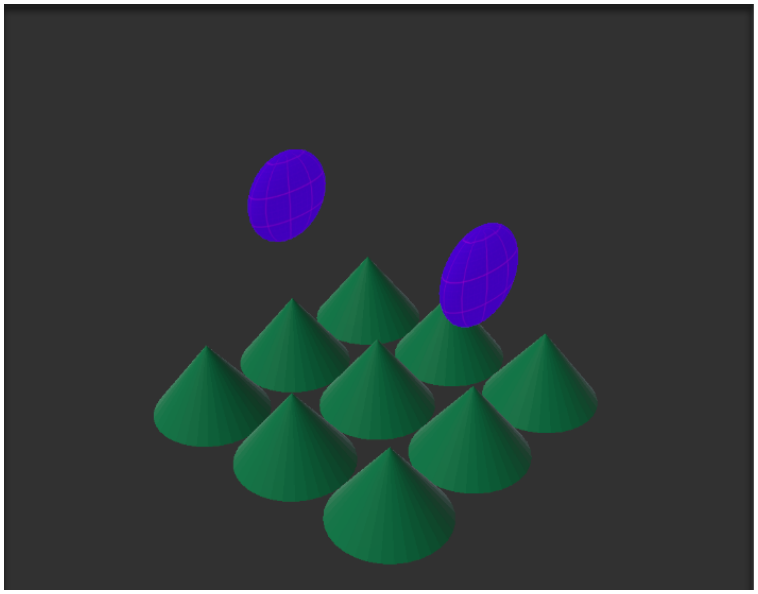


be predetermined classes that the generation can guide to control game Flow, or else there is no order added to the randomness.

Procedural Content Generation

The main interaction between the game and the player takes place on an infinitely generating plane. This infinite plane is actually an **Object Pool** of thirty, two-dimensional Unity game objects. An Object Pool is space saving game development technique for controlling the states of duplicate objects. To save space, the Z position of these objects is simply updated based on the position of the player, a more detailed description of how this is done will be given in the next subsection. As the player drives forward, road tiles are continuously and infinitely generated in front of them. The player can also drive the opposite direction, but they will be shown a large 'WRONG WAY' sign if they do and no InObs will be generated for them. There are also invisible walls that prevent the player from driving too far to the left or right. With each road tile that gets generated as the player drives forwards, there is a chance that a game object will be generated as well.

Figure 2.4: Spike Interactable (Level 0 Example)



The game is designed to use DDA techniques to influence the different Type's rate of generation. The game is also designed to use PCG techniques to create these objects and control their placement on their respective tile. The objects are always randomly generated some distance from the center of their tile. The PCG is controlled by a seed value, which is always the epoch time at the beginning of the game. This way, no two play throughs are the same and players are able to drive forward infinitely.

This style of generation means that the player must make decisions about which objects to interact with as they drive by steering themselves towards the chosen object. Thus, the player is explicitly and implicitly taught to only drive forward and must purposefully steer towards specific InObs to gain points. Through this, modern PCG techniques can be explored in a controlled system. The specific areas of code that relate to PCG will be explained in detail later.

Engagement, Skill, and Preference Values

Each time a player interacts with a game object, there are four main statistics that are recorded. There are thirty-five variables that are also tracked by scripts in the background during play. The four main statistics recorded for each interaction are raw values that relate to how well the player interacted with the object. For each interaction these values track the number of frames spent in the air, the number of total points gained, the speed upon exiting the interaction, and the number of tricks performed while in the air. The total points gained and the number of tricks performed together also implicitly record a fifth raw statistic that is the number of points gained from Orb collection. Recording the raw number of Orbs collected does not tell us much considering there are different classes of Orbs that give different amounts of points. The raw data that comes from every interaction allows the Data Cycle to create a model of the player. This is done by processing the data and comparing the player's performance between different Types and Levels to generate **Preference Values**, which represent the player's ability or preference towards specific Types and Levels. Using Preference Values the rates of generation can be influenced, creating the DDA Data Cycle.

Dynamic Difficulty Adjustment

The DDA used in this project is designed around the Data Cycle concept. In order to understand the Preference Values of the current player, the Data Cycle uses pre-determined values that represent a non-preferential style of play. This allows the Data Cycle to influence the rate of Type and Level generation towards the preferences of the player. For example, If the player shows that they are capable of interacting with higher Level InObs, the DDA will guide generation towards InObs of higher Levels. The Data Cycle outputs generation rates at a speed of once per interaction, which is the highest possible iteration frequency available based on the frequency of raw data input. This means that every time the player interacts

with an object, the raw data is used to update the Preference Values, which are stored and compared to influence rates of generation, and then the cycle repeats.

There are design elements in the game that are used intentionally to aid in the implementation, execution, and evaluation of the Data Cycle in this project. Each Type is designed to create a different kind of interaction with the player. These interactions, however, are not all built to be equally enjoyable. The Spike Type, for example, is designed to have a high ratio of danger to point gain built into the kind of interaction that the player can have with it. This is done simply by not giving the player the opportunity to gain more points through tricks as the Levels increase. Instead, only the difficulty increases, thereby implicitly guiding the preference of the player towards other Types. Thus, if testers express post-play that they did not enjoy interacting with the Spike Type and this same attitude is expressed through the Preference Values, then the player model and the Data Cycle are correctly evaluating the preferences of players based on how they interact with the game. A simple PCG technique was chosen to aid in the implementation and execution of my DDA Data Cycle. Having only four classes of object and three subclasses allowed the system to both have a manageable amount of data to process with each interaction and a solid number of different categories to move the player around in. With each class or subclass added, the amount of data processed through the Data Cycle grows proportionally, and the less precise the Preference Values become. Having too few classes, however, forces players into categories that are not necessarily representative.

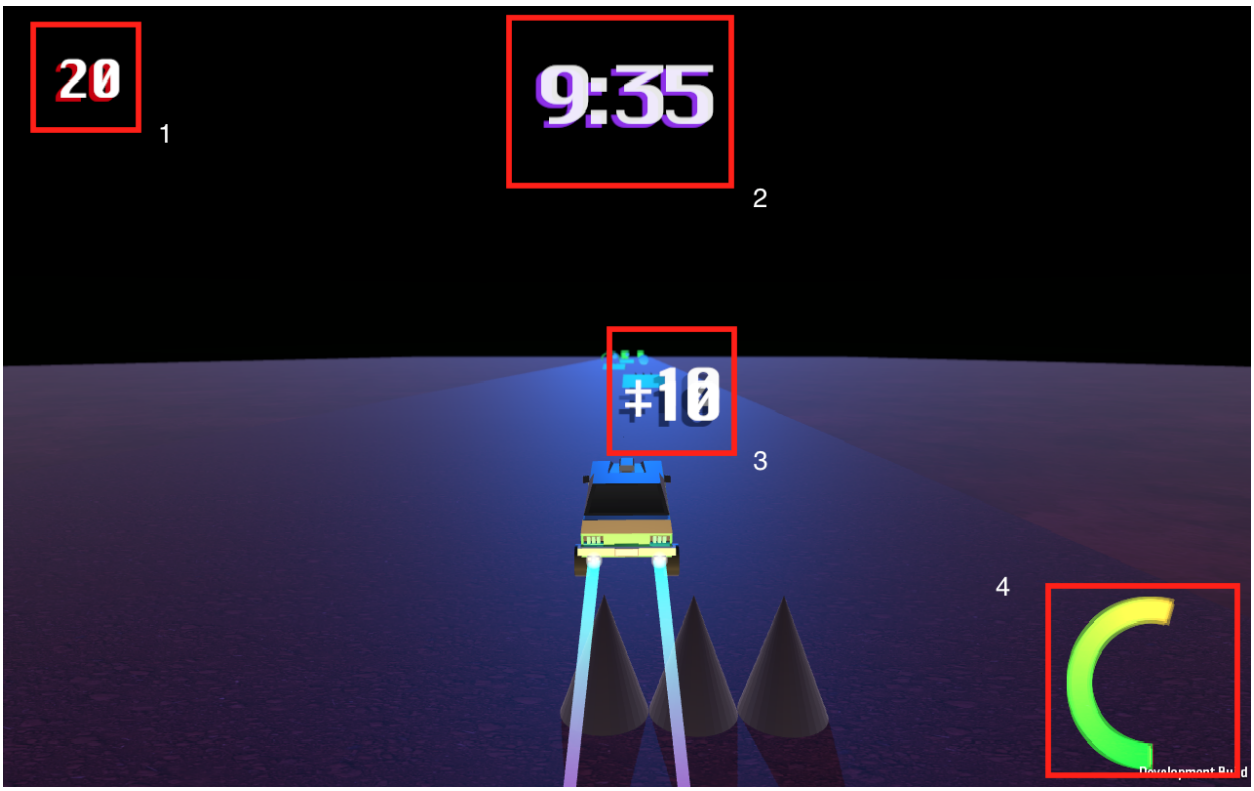
Player Goal

There is an inherent assumption made in the implementation of the PCG and the DDA that the main goal of the player is to gain points at the highest rate possible. This assumption is a valid one to make because of two reasons, the in-game **User Interface** (UI) and the

available player inputs.

The UI is designed to implicitly guide the player towards this goal of points and speed. The

Figure 2.5: User Interface



UI is composed of four text objects (Figure 2.5), not including the 'WRONG WAY' sign that is displayed when the player drives in the incorrect direction. Each UI object is used to control the player goal. Arguably the most important UI component, is the points monitor (Figure 2.5.1). The points monitor updates and displays the player's score in large, colorful numbers in the top left corner of the screen. There is also an animation (Figure 2.5.3), which occurs each time a player gains points, displaying a number showing exactly how many points the player gained for the action they performed (collecting Orbs or performing tricks). The score monitor and the score animation are both used to tell the player that the most important part of the game is gaining points. There is a UI element displayed at the top-center section of the screen that displays how much time is left (Figure 2.5.2). This timer starts at 10 minutes and counts down to zero. This simple design tells the player that

they must try to do what they are doing as quickly as possible before the time runs out. The last UI element is a speedometer (Figure 2.5.4). The speedometer is simply a ring that both grows radially and changes color according to the speed of the player. This tells the player that it is very important to the game to drive fast. Thus the design of the in-game UI is geared specifically to teach the player that they must gain points as quickly as possible while driving as fast as possible and they only have a limited amount of time to do so.

Although the background design concepts of the game, PCG and DDA, are often linked to genres like roguelikes, roguelites, and RPGs, the actual player-game interaction is closer to a platformer or an infinite-runner. In roguelikes, for example, the player is often faced with numerous, multifaceted decisions at each turn of the game. In infinite runners, however, the decisions that the player must make at each moment are simple and instinctual. The inputs in this game are limited to driving, jumping, and flipping in the air. The game is fast-paced and reaction-based. There is no available game play for anything other than driving quickly and trying to collecting points.

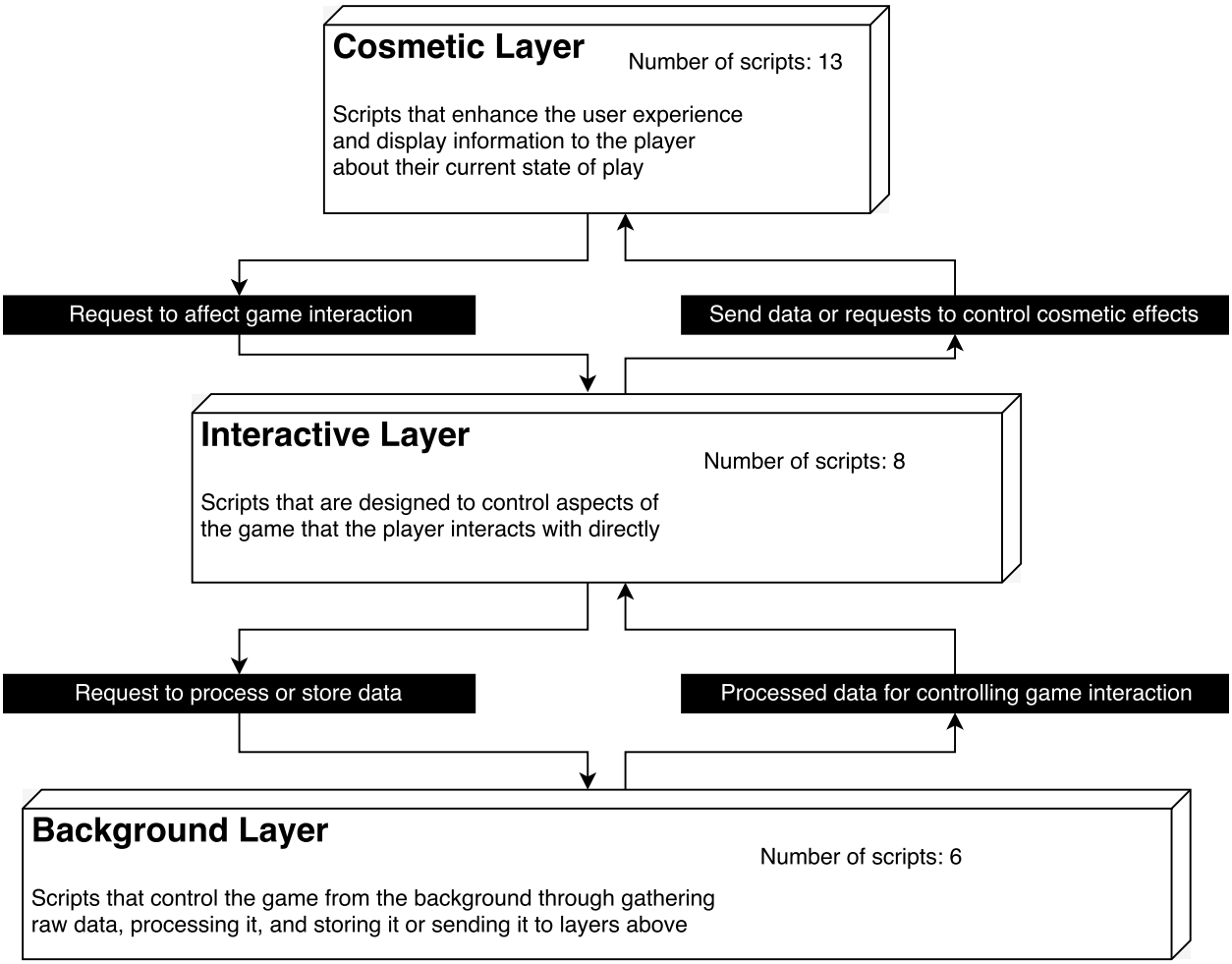
Through the design of the in-game UI and the available user actions, the player is implicitly controlled to interact with the game in only one kind of way. Because of this, the Preference Values used in the Data Cycle, and therefore, the DDA in general, can make the assumption that higher points, air time, numbers of tricks, etc. relate directly to a player's higher preference for the Type and Level of the object with which they interacted.

2.2 Code Overview

Layer Architecture

There were twenty-seven C# scripts written and used for this project. These scripts can be sorted into one of three categories. These categories are designed in a framework composed of layers (Figure 2.6). Higher layers are more responsible for collecting data and displaying information back to the player. Lower layers are more responsible for data processing. Each script in these layers relies on scripts in the layers above and below in order to function.

Figure 2.6: Layer Architecture



Thirteen of the scripts can be categorized as **Cosmetic**. The Cosmetic category is the top layer. These scripts were mainly written to enhance the user experience and display information to the user about their current state of play. Some of the scripts in the Cosmetic layer also contain small, essential design elements for the game.

Eight of the scripts can be categorized as **Interactive**. The Interactive category is the middle layer. These scripts are designed to control aspects of the game that the player interacts with directly, and transfer the essential data to the Cosmetic layer in order to be displayed. Examples of this include the Car Controller script or the UI Controller script. While these types of scripts are vital to the user experience, they still must be built on top of a lower, more essential layer in order to be fully functional.

This last category is called the **Background** category and it is the lowest level layer. There are six scripts in this layer, all of these scripts control the game from the background. Examples of scripts in this category are the PCG controller, the DDA controller, and the Car Stabilizer scripts. The six scripts in this category also send information to the Interactive layer in order to control interactive elements of the game or to be sent further up to the Cosmetic layer for displaying information directly to the player. All three layers have their required jobs, and each works in tandem with the others to complete the jobs.

Descriptions of Essential Scripts

The most essential script is *MasterController.cs*. This script is in the Background layer because there is no direct interaction between the player and this script. The goal of this script is to continuously update and organize all valuable and trackable information about the player's interaction with InObs. This script is also responsible for calculating Preference Values for all of the Types and Levels, which is the most important step in the DDA Data Cycle. An even more in depth evaluation of the Preference Value calculation can be found

in the next subsection.

The *GenerateInfiniteFull.cs* script is responsible for generating the entire environment that the player can interact with. This script is in the Background category because, although it takes into account the player's Preference Values, there is no direct player-script interaction. There are two integral things that this script is in charge of. While tracking the motion of the player, this script constantly recycles the road tile Object Pool to keep the player centered in the game area. The script is also responsible for choosing and generating the InObs for the player so that the game can be played. InOb generation is decided in a top-down manner. The Preference Values for each Type and Level are stacked, then two random numbers are generated and wherever these random values fall decides which object is generated next. A more detailed description of the choosing process is described in the next subsection.

The script that is used to log and store values related to testing is called *StatJSON.cs*. This script is also in the Background category because the player does not interact with it directly. *StatJSON.cs* is used to gather all of the recorded data taken from the tests and log them for later data analysis. This script creates four files at the end of each play session. The ID file contains unique identifiers for the test including a timestamp, the generation seed, the date, and an indicator of whether the test used adaptive generation or not. The Timed Data file contains snapshots of the game, which are taken every 30 seconds. These snapshots contain data that changes over time, like Preference Values, total points, and total tricks performed. The Final Data file is a final evaluation of player statistics when the test is over. This file contains information about the statistics that the player tallied up over the course of play. The last file it creates is the Interactions file. This file is a log of each time the player touched an InOb and the four main statistics of the interaction that were recorded.

An extremely important script for player engagement is the *CarController.cs* script. This script is in the Interactive category. This is because, although it is also involved in essential data collection, it takes direct inputs from the player and translates them to car movements

in the game. This script is what allows the player to accelerate, reverse, handbrake, jump, and rotate in the air. It is also responsible for gathering data on the state of the player in each frame and sending that data to *MasterController.cs*. Data taken from this script is used to evaluate Preference Values, engagement, and skill progression throughout the game.

Lastly there are two scripts that work together to control a large part of the game play. *InteractController.cs* and *OrbController.cs* are both responsible for tracking player-object interaction and notifying the *MasterController.cs* script of points gained by the player. These scripts are in the Interactive category. *InteractController.cs* is attached to each InOb and *OrbController.cs* is attached to each Orb in the game. Both scripts keep track of the Type and Level of the Prefab they are attached to. These scripts send this unique identifying information to *MasterController.cs* for interaction evaluation when the player's collider enters their collider.

2.3 DDA Data Cycle

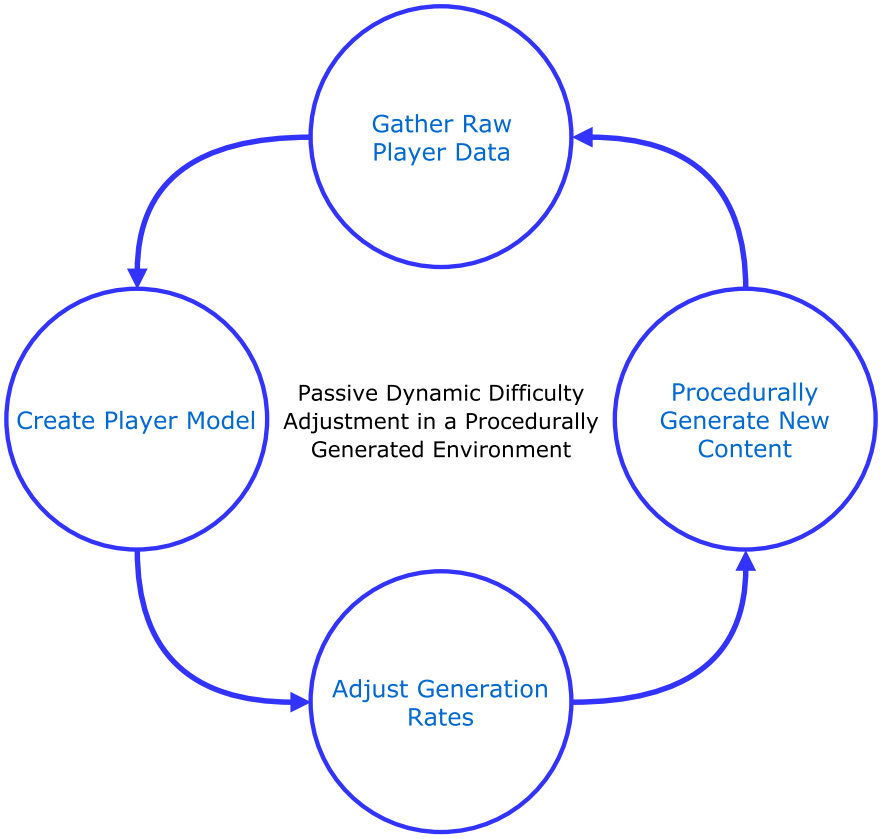
Overview

As described in a previous subsections, Preference Value calculation is an essential step in the DDA Data Cycle. These values represent the game's current evaluation of the user's style of play and are used to control the rates of generation for each InOb. The following is a description of the Data Cycle process that occurs after each interaction.

Types And Levels

As stated earlier, there are four Type categories: Ramp, Speed Boost, Destructible Wall, and Spike Pit. The *MasterController.cs* script contains classes that are abstract references

Figure 2.7: DDA Data Cycle



to each of these Types. These four classes hold the information about every player-object interaction with that specific Type category.

The InOb Type categories each have three sub-categories called Levels. The Level categories are L0, L1, and L2. *MasterController.cs* also contains an abstract class referencing each of these Levels as well, where all four Type classes in the script each have three Level subclasses that store information about every InOb interaction for that specific **Type-Level**. Type-Levels is a keyword that will reference one of the twelve different combinations of Type and Level categories. As an example, if the player interacts with an object of Type-Level Destructible Wall L0, the L0 subclass of the Destructible Wall class records, processes, and stores the information about that interaction. In this example, the Destructible Wall class also processes the gathered information for it's own Preference Value calculations.

ExpectedValues

ExpectedValues is a three dimensional matrix (four by three by four). The first dimension relates to the four different Type classes. The second dimension relates to the three Level classes for each Type. The third dimension relates to the four different kinds of statistics that are recorded for each Type-Level. The values for the ExpectedValues matrix were calculated in the initial rounds of the game development. The ExpectedValues were filled in by recording multiple test play throughs, then analyzing the resulting values to find the statistics that most directly represented a simple, low-risk, and, most importantly, non-preferential style of play. From this analysis the ExpectedValues could now be used to inform the player modeling step of the Data Cycle in the DDA.

The ExpectedValue matrix is static, meaning that all players will be compared to the same values. For Preference Value calculation the most important factor in analyzing the player's statistics is that the expected raw statistics are programmed to represent the same type of play across all Type-Levels. Because of this, the skill level (magnitude of points gained, tricks performed, etc.) represented by the ExpectedValues does not matter, only the relationship between the values for different Type-Levels matters. Preference will be shown through the difference in the magnitude of recorded statistics between one Type-Level and another.

As a final note, because the ExpectedValue matrix is designed to represent non-preferential play, setting the values uniformly would not be effective. This is because the expected raw value of frames spent in the air after going off of a Ramp L2 should inherently be different than that of frames in the air after going through an Destructible Wall L2.

In summary, the different physical elements of the InObs produce different player-object interactions and thus the only way to evaluate the player's preferences through these interactions is to code the ExpectedValues to be non-preferential. Thus, the only way to correctly understand the play style of a player at any given time is by comparing their performance

to a non-preferential player model.

StatSum

The **StatSum** matrix is the intermediate step between the raw statistical values recorded from InOb interactions and the final Preference Values. The StatSum matrix is a four by three nested array in which the first dimension correlates to the four Types and the second dimension correlates to the three Levels for each Type. This matrix is designed to store the processed information about the statistics for each Type-Level. The calculation of each entry of the StatSum matrix is the first player modeling step in the DDA Data Cycle.

The entries are calculated by comparing raw data recorded through play to the raw values in the ExpectedValue matrix. Equation 2.1 shows the process for calculating StatSum values. This equation is used at the start of the game and after each interaction. If the number of interactions is zero and the average mean value of the statistics is equal to that of the ExpectedValue for each Type-Level, the resulting StatSum value is going to be equal to the sum of the weights for each kind of statistic. This means that at the beginning of the game, since there have not been any interactions, the player is believed to have a completely non-preferential play style. Because of this, each StatSum value will be completely uniform until the first interaction.

$$
\begin{aligned}
n &= numberofstats - 1 \\
CT &= currentType \\
CL &= currentType\text{-}Level \\
value &= \sum_{i=0}^{n} \frac{CL.averageStats[i]}{ExpectedValue[CL.ID, CT.ID, i]} \times Weights[i] + \sqrt{Cl.numInteractions}
\end{aligned}
\tag{2.1}
$$

Below are the weights for each type of statistic and their total sum.

Frames in the air = 0.4

Number of tricks = 0.5

Speed upon exiting = 0.4

Number of points gained = 0.8f

Thus based on the value of the weights plugged into the above equation, each spot in the StatSum matrix before any interactions have taken place is equal to 2.1.

$$(1 \times 0.4 + 0) + (1 \times 0.5 + 0) + (1 \times 0.4 + 0) + (1 \times 0.8 + 0) = 2.1$$

There are two arrays and one floating point value that store valuable elements of information related to the StatSum matrix. The calculations for each of these is shown in equation 2.2. **TotalSum** is the sum of all StatSum values. This number is used to compare Preference Values between different Types. The **TypeSum** array is an array of size four in which each index relates to one of the Types. The value of each index in the TypeSum array is the sum of the StatSum values of the Levels for each index's related Type (the sum of each column in the StatSum matrix). These are used to compare Preference Values between Levels of the same Type. Third, the **LevelSum** array is an array of size three in which each index relates to one of the three Levels. The value of each index is equal to the sum of the StatSum values of all the Types for each each index's related Level (the sum of each row in the StatSum matrix). The values in this array are used to compare Preference Values Types-Levels of the same Level.

$n = numberoftypes - 1$

$m = numberoftypes - 1$

$CL = currentType\text{-}Level$

$CT = currentType$

$value =$ value calculated from equation (1)

At the start of the game:

$$TotalSum = \sum_{level=0}^{n} \sum_{type=0}^{m} StatSum[type, level] \tag{2.2}$$

$$LevelSum[level] = \sum_{level=0}^{n} StatSum[type, level]$$

$$TypeSum[type] = \sum_{type=0}^{m} StatSum[type, level]$$

After each interaction:

$$TotalSum = TotalSum + (value - StatSum[CL.ID, CT.ID])$$

$$TypeSum[CT.ID] = TypeSum[CT.ID] + (value - StatSum[CL.ID, CT.ID])$$

$$LevelSum[CL.ID] = LevelSum[CL.ID] + (value - StatSum[CL.ID, CT.ID])$$

Preference Value

Preference Values are essential to the DDA Data Cycle because they are used to control generation. The calculation of the Preference Values are based on the relative magnitude of the StatSum values. Each Type class in the *MasterController.cs* script has its own preference variable that keeps track of the player's preference for that Type. Each Type-Level in the *MasterController.cs* script also contains its own preference variable. This results in sixteen different Preference Values that are re-calculated after each interaction. The creation of the Preference Values is the final player modeling step in the DDA Data Cycle.

The Preference Values for Types and Type-Levels are created in different ways. For Types, they are calculated by comparing the magnitude of the TypeSum for that Type to the TotalSum variable. This way, all Types are evaluated by how well the player performed with that Type in comparison to the rest of the Types (each column is compared to the whole matrix). The Type-Level preferences are calculated by comparing their relative StatSum values to the LevelSum value for their Level (each square is compared to the whole row in which it lies).

The pseudo-top-down fashion of this design for calculating Preference Values is chosen for a very specific reason. A straight forward top-down fashion would pick the most preferred Type, then the most preferred Level in that Type. The problem with this design is that it would always lean towards the higher Level items. Through analysis of game play it was found that as a player becomes more comfortable with the controls and they begin to take more risks, not only will get higher points than the ExpectedValues, but this difference in value grows exponentially as the Level of objects goes up, resulting in StatSum values that always lean toward higher Levels, even if that is not reflected in their gameplay. By comparing Type-Levels to other Type-Levels across Levels instead of through Types, the algorithm is able to remove this bias in the StatSum matrix. The methods for the calculation of Preference Values are shown in equation 2.3.

$$
\begin{aligned}
&Types = \text{List of the four Type classes in } \textit{MasterController.cs} \\
&Levels = \text{List in each Type class of its three Level subclasses in MasterController.cs} \\
&Types[type].Preference = 100 \times \frac{TypeSum[type]}{TotalSum} \\
&Types[type].Levels[level].Preference = 100 \times \frac{StatSum[type, level]}{LevelSums[level]}
\end{aligned}
\tag{2.3}
$$

Procedural Content Generation

The generation of the terrain and the InObs are both handled by the *GenerateInfiniteFull.cs* script. The first check in the main update loop of the script is done on the position of the player. If the player is found to have traveled further than the size of a single road tile, then the code for recycling the terrain and generating a new InObs is called. The *GenerateInfiniteFull.cs* script is responsible for the third and fourth steps of the DDA Data Cycle.

As stated earlier, the infinite road is a simple Object Pool. When the player moves forward the amount of distance equal to the size of one tile, the list of road tiles is treated as a stack. The tile at the top of the list is popped, the position of the tile is updated, and the tile is added back to the end of the list. This way, the list of road tiles mimics the real-space line up of the road tiles and the player is kept on the tile in the center of the stack throughout gameplay. If the player is moving backwards through the game, the list of road tiles is treated as a queue. The same recycling process is undergone, only with LIFO recycling instead of FIFO. This method allows the player to drive forward forever and is common practice in infinitely generated games because of its low computational cost.

The generation of the InObs is more complicated. The *GenerateInfiniteFull.cs* creates objects at the same frequency that terrain tiles are recycled, but since the InObs are all created from different prefab models, they cannot be treated as an Object Pool and cannot be recycled in the same way. The list of InObs is treated as a FIFO queue. Whenever a new object is created, the object at the top of the queue is destroyed and the new object is placed at the end of the queue.

GenerateInfiniteFull.cs uses the Preference Values calculated by the *MasterController.cs* script to pick the Type and Level of the next InOb. *GenerateInfiniteFull.cs* holds a cumulative list of Type preferences and the Type-Level preferences, which are updated after each

interaction. The processes of updating these two cumulative Preference Value lists is the third step of the DDA Data Cycle. The process of picking the next object is the fourth step in the DDA Data Cycle.

The fourth step in the DDA Data Cycle begins with generating two random numbers. Both of these numbers are generated between zero and the total cumulative value of the Type and the Type-Level preferences respectively. These random numbers define which object will be created. The randomly generated numbers are compared to their respective lists of cumulative values. The first value that is larger than or equal to the random number tells us which Type or Type-Level to generate. This method of deciding Types and Type-Levels is done in a top-down manner. This means that the Type is picked first, then the Level is picked out of that Type. This is how the game uses random number generation to control the generation of content.

At the beginning of the game, since each Type has a uniform Preference Value of twenty five, a random number would be generated between zero and the cumulative value of each Type Preference, one-hundred. In this example let us assume the random number generated is thirty-seven. This would result in the game picking the second value in the cumulative list of Preference Values, fifty. The Type picked will be the Type with the static ID that relates to the index of that Preference Value, which is one. Thus, the InOb generated would have a Type of Speed Boost.

This method of InOb selection is also done in order to allow some noise into the rate of object generation. Generating objects simply by picking the Type and Level with the largest Preference Value would create a positive feedback loop where the same InOb would constantly be generated and the game would not have the data to evaluate any other type of player-object interaction. By adding some randomness into the generation, the user's gameplay style can organize the random noise into a coherent pattern based on Type and Level preferences, which can be understood by the game and later recreated through recorded data.

Chapter 3

Experiment

3.1 Design of Experiment

In order to test the ability of the DDA in PCG, experiments were run on the finished product of the game that was detailed in the previous chapter. Participants were recruited to run these experiments, under the guidance and approval of the IRB. In order to isolate relevant data, participants were split into two groups, a control group and an experimental group. The control group was given a version of the game that uses complete random generation, while the experimental group received the full experience of the DDA.

The experiment was designed as follows: Each participant is given a packet with three items, the consent form, the instructions and controls sheet, and the questionnaire. Regardless of group, each participant plays for a maximum of 40 minutes. Each tester must first play a tutorial, which is a maximum of ten minutes long. In the tutorial, players are shown each of the twelve InObs, one-by-one. Players are given the chance to learn the controls by driving through, on, and over these InObs, starting with all of the L0 InObs and finishing with all of the L2 InObs. After this, players are placed in a game-area that contains two copies of

each InOb to interact with. They are allowed to spend the rest of the tutorial time playing in this area or they can move on to the next section if they feel ready to do so.

The next section of the experiment is where the two groups differ. The participants in group A, the experimental group, are first given two rounds of the adaptive version of the game, then a final round of the non-adaptive version. Group B participants, the control group, simply receive three rounds of non-adaptive gameplay. Each round of gameplay is ten minutes long. The game only generates L0 InObs until the player has gained two-hundred points overall, in order to give the game time to evaluate the player's Preference Values. Finally, after the participant goes through all three rounds of the experimental phase of the test, she is then asked to fill out the questionnaire form.

During each of the three rounds that both groups play, there are many different variables that are tracking the player's Preference Values, skill level, and engagement level. These variables are recorded and logged to text files for later extraction. During the non-adaptive versions of the game, the game moves through each step of the Data Cycle except for the generation adjustment phase. This way data is calculated and saved the same way no matter which group the participant is in.

The test is designed to isolate the data that is be most relevant to this paper. These experiments are attempting to extract data that relates to how adaptivity can affect a player's skill progression through a game. The participants were each given the chance to learn the game in separate environments, then were placed in the same environment for one final round. The data analyzed in this experiment is taken only from the final play through. If this paper's hypothesis is correct, participants in the experimental group should show signs of having learned the game better than those in the control group through higher point gain and more positive interactions with InObs. Through capturing snapshots of the participant's progression through the game, the way each participant interacted with the game can be recreated and compared based on the group in which the participant was placed.

3.2 Experiment Results

Skill Progression and Engagement

Overall fourteen people participated in the experiment, seven in the experimental group and seven in the control group. Because there is only a limited amount of data that can be extracted from this number of participants, a statistical technique called **Bootstrapping** was used to artificially expand the data set. Bootstrapping is a technique used on small sample sizes in statistical analysis. The idea behind Bootstrapping is that in order to increase the number of values for analysis, new values can be created by finding the difference between two values randomly selected from each group. By repeating this process many times over, the data available for analysis can be expanded. The results of this technique can be used to show bias in the data. For example, if the experimental group all gained much higher points on average, then the histogram of the Bootstrapped data would clearly show a mean larger than zero and a variance that kept a majority of the data above zero. Analysis of participant's skill progression and engagement both relied heavily on this technique.

In this project, all Bootstrapped data is created by randomly selecting values from the experimental group and subtracting them by random values from the control group. This means that if, for example, the mean value of some Bootstrapped data is positive, the experimental group had higher values on average for that data type.

The first data type analyzed was the statistic that most directly relates to skill and engagement, the total number of points gained. This variable is intertwined with both engagement and skill. Players with low skill level but high engagement can produce similar statistics to players with high skill level but low engagement. Because both skill and engagement are related to this variable, analyzing it can show, very generally, how players in the different groups performed. To analyze this statistic, a histogram of the Bootstrapped data was

created (Figure 3.1).

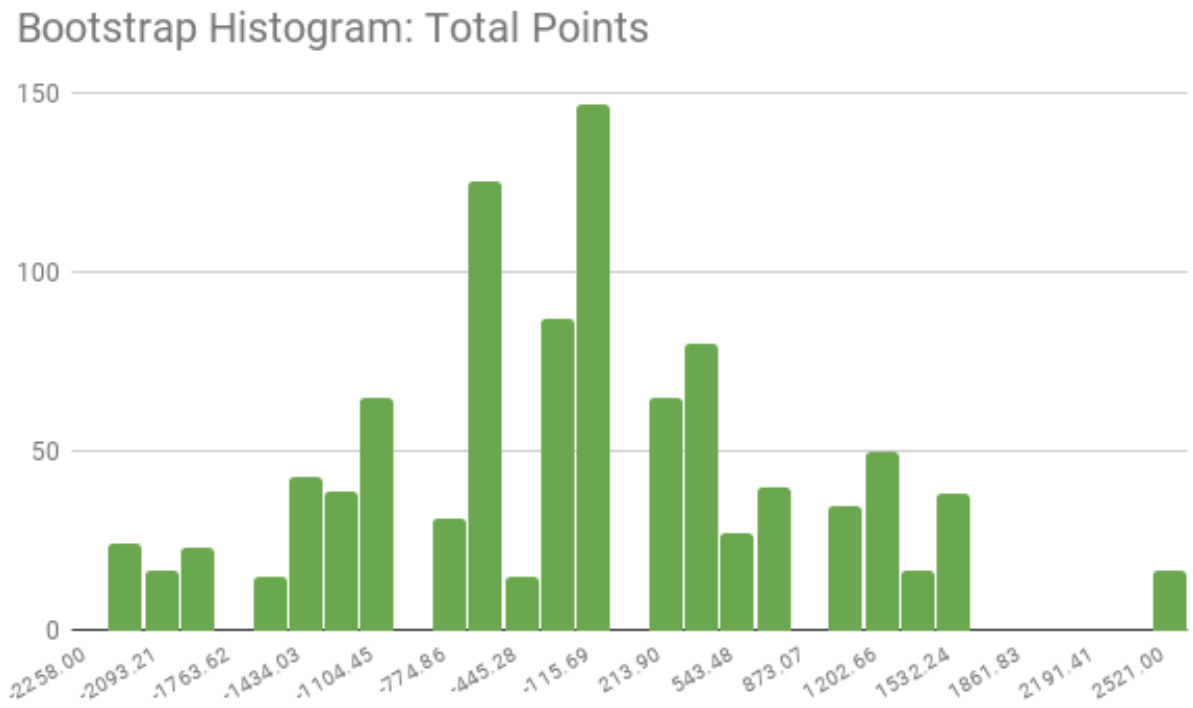


Figure 3.1: Total Points Histogram

The mean of this data is -104.209, while the standard deviation is 987.389. As can be seen by this information and through Figure 3.1, the values span almost exactly over zero. This tells us that this game's implementation of the DDA Data Cycle did not make any perceivable difference in the growth in skill or the engagement level of the participants.

Total points represents a combination of skill and engagement level and thus only gives a surface level glance into the meaning behind the data extracted from this experiment. In order to confirm what the above data shows, it is important to analyze variables that can compare individually either the skill or engagement level of participants.

To do this, the next type of data analyzed is the number of deaths. The number of deaths overall is a variable that can tell us specifically the skill level of the player. This statistic is updated each time the player runs into a spike object with the body of their car. Most often

this occurs when the player does not correctly time a jump over the Spike Type or when the player goes off a ramp and lands on a spike further down the play area. Players with higher skill levels will be able to anticipate these two dangers and find techniques to die as infrequently as possible. Players of lower skill will die at a higher rate according to their skill level regardless of how engaged they are in the game. The Bootstrapping technique was also used on the number of deaths, and the histogram of the results are displayed in Figure 3.2.

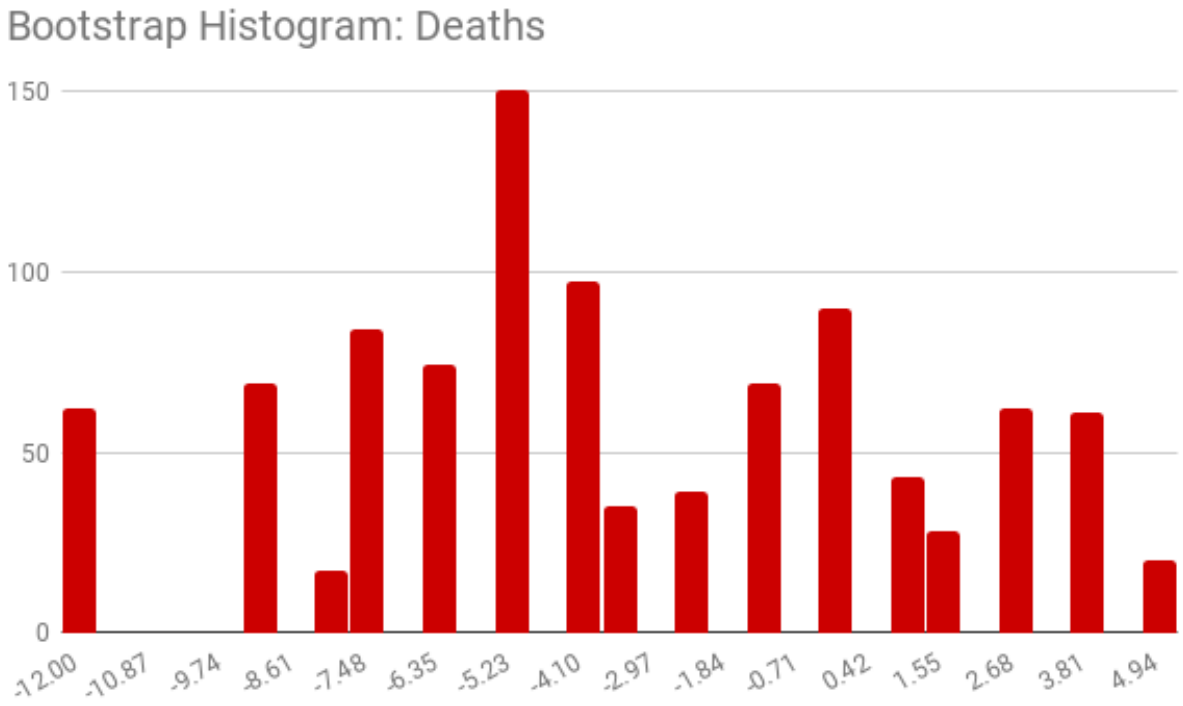


Figure 3.2: Number of Deaths Histogram

The mean of Bootstrapped data is -3.294 and the standard deviation is 4.426. This mean shows that (based on an artificial population created from the experiment's sample data) the control group died more times on average than the experimental group. Unfortunately, the percentage of data that is negative is too small to call this data conclusive in any way. The results of this analysis can only point to the fact that the adaptive version of the game had minimal to no affect on the skill of the player during the third play session.

The third and final piece of data analyzed, is the number of flips performed by the players. This variable directly relates to the participant's level of engagement. A flip is a very simple and easily performed interaction with the game. Performing a flip correctly requires little skill, the player must simply control the pitch of the car while in the air to complete a full rotation. This does require a level of engagement, however, because players must still make a conscious decision to perform a flip each time they interact with an object. Since all tricks are calculated based on the length of time that a player turns the car on a specific axis, the player cannot accidentally perform tricks. Tricks can only occur through active engagement with the controls. Thus, if data shows that player A performed tricks more often than player B, it is fair to assume that player A was more willing to take action to gain higher points with each interaction and thus was more engaged in the game. Again, since there were only

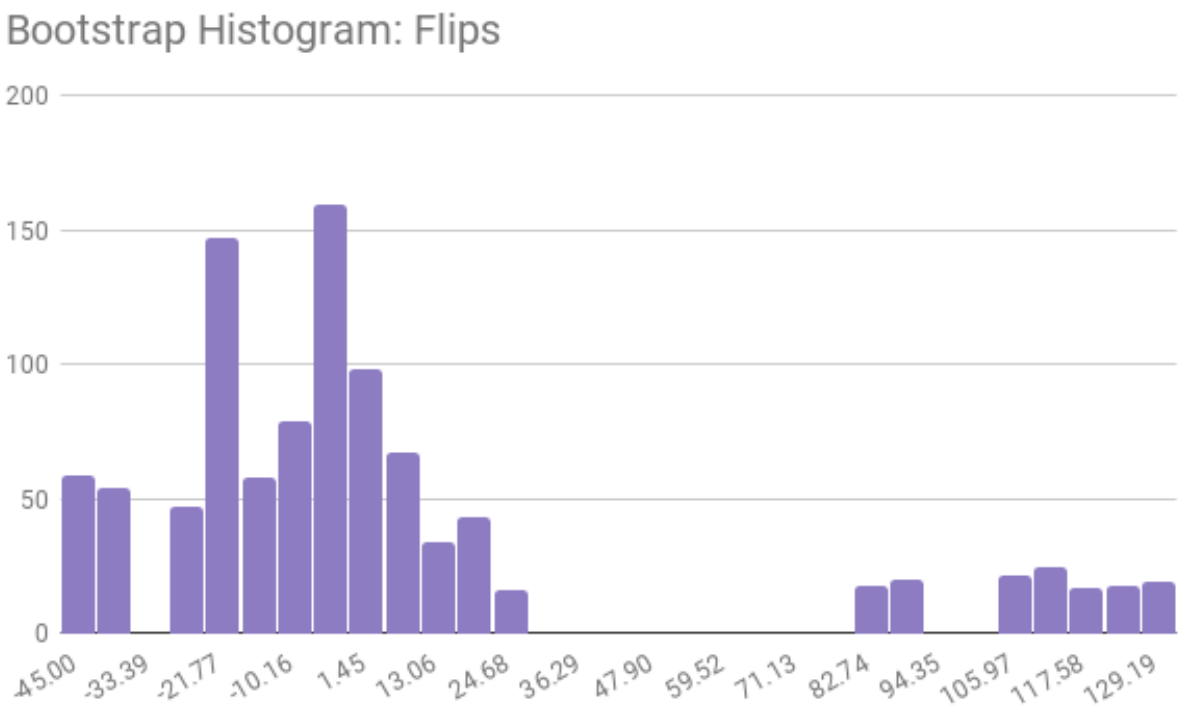


Figure 3.3: Number of Flips Histogram

seven participants in both groups, the Bootstrapping technique was used to perform the analysis of the data that related to flips performed. The histogram that resulted from the

Bootstrapping can be found in Figure 3.3.

The mean of this Bootstrapped data is 8.339 and the standard deviation is 44.586. This data tells the same story that the other two variables told. While the mean is slightly on the positive side, meaning that the experimental group performed more flips than the control group on average, almost all of the data is evenly split into negative and positive. This means the implementation of the DDA had no conclusive effect on the outcome of the player's overall engagement level.

These three data types best represent the skill and engagement levels of participants, and they all tell us that the implementation of this game brought an inconclusive answer to this paper's hypothesis.

Data Cycle Accuracy

The data gathered from the experiments shows no signs of the game having affected player skill or engagement level through the DDA Data Cycle. This section focuses on the steps in the Data Cycle that were and were not responsible for this inconclusive data. The Data Cycle was split into four steps, gather raw data, create current model of player, adjust generation rates, and generate new content.

The accuracy of the first step, data gathering, is difficult to call into question in the context of this project. Since both the creation of the game space and collection of game data are performed simultaneously within the same simulation, the accuracy of the data cannot be inaccurate. Since the data must be accurate, the accuracy of the player model must be analyzed. To do this, an in depth analysis of the Preference Values was performed.

In order to evaluate the accuracy of the Preference Value calculation, each participant was asked in the questionnaire (after completing forty minutes of game play) to rank the four

Types in order of preference. The final Preference Values were also extracted from each participant's third play session and the two data evaluations were compared. Data from the questionnaire is in list format while the in-game Preference Values are integer values. To bring these two different types of data into a comparable format, a simple scoring system was used for the questionnaire answers. The final results of this questionnaire data were: Speed: 47, Ramp: 43, Wall: 35, Spike: 17. The Preference Values from each participant were summed and the results of this evaluation were: Speed: 4063, Ramp: 3834, Wall: 3325, Spike: 2603. The magnitude of each of these values is only relevant for comparisons within the evaluations. Thus, Figure 3.4 shows the *normalized* versions of these two scoring system.

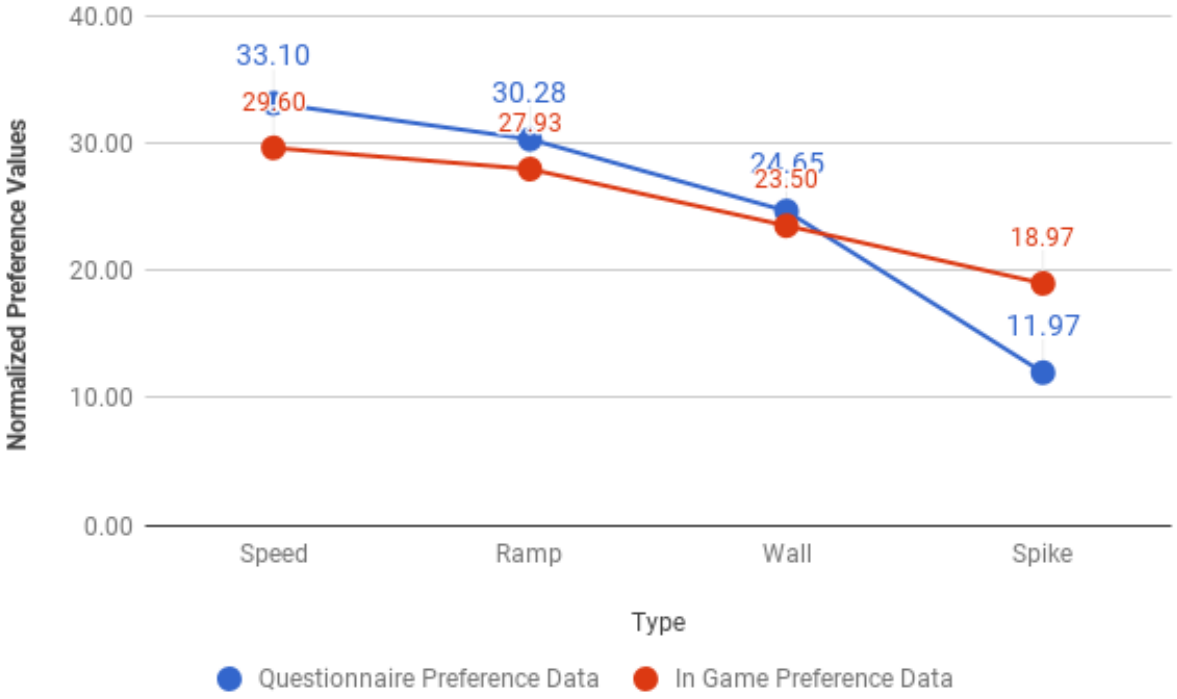


Figure 3.4: Preference Evaluation

The ability of the Data Cycle's Preference Values to match the participant's preferences is very accurate. With only seven participant's data, the normalized in-game preferences scores above are, on average, within 96.50% of the normalized questionnaire preference scores. With

more participants and more ExpectedValue adjustments the data from the two different sources would most likely converge on each other. Because of this evaluation, it can be concluded that the first two steps of the DDA Data Cycle are not responsible for the game's inability to significantly affect player skill level or engagement.

The *GenerateInfiniteFull.cs* script is responsible for both of the last two steps of the DDA Data Cycle, adjusting generation rates and generating new content. The generation adjustment is most likely not the reason behind the failure of the DDA Data Cycle. This project's design of the generation rate adjustment is directly inspired by the Rubber Band AI system. The 'rubber banding' used in this project can be understood by thinking of the player being placed in the center of four pegs (representing the four Types) with a rubber band surrounding all of the pegs. If the player moves in the direction of one of the pegs, the rubber band is stretched in the direction of that Type, and thus the other three Types are concurrently pulled away. This system is a very common style of adjusting game variables and, since it is directly influenced by the player model, which we know to be accurate, it is not because of this step that the Data Cycle failed to affect the player's engagement.

This leaves just the final step of the cycle, the generation of new content. This is most likely where the Data Cycle failed. It is probable that the process of generating random numbers and applying them to the generation rates added too much noise into the generation of objects, which threw off the rest of the Data Cycle. The content itself is also likely to have added to the problem. While the game was able to pick which Types players preferred the most with a high level of accuracy, the Types may not have been different enough to affect player engagement. The difficulty adjustment was also not handled as well as was hoped by the different Levels within each Type. As the player became more comfortable with the controls, the level of difficulty did not rise high enough to meet the player's skill through increasingly difficult InObs.

Chapter 4

Conclusion

4.1 Summary of Thesis Achievements

The purpose of this paper was to explore PCG, DDA, and how their combination could be used to affect player engagement. The data produced by the in-game variable tracking showed that this project's use of adjusting generation rates in PCG had inconclusive effects on player skill and engagement. This project was able to discover, however, the powerful accuracy of the passive DDA Data Cycle when implementing a Rubber Band AI method, as well as the methodology of creating and optimizing a PCG system.

Game Flow is something that must always be considered in game design and development, this project highlighted something that game creators can use in order to increase quality of life for both themselves and their consumers. Passive DDA is possible through modern AI techniques and will most likely be seen more often in the world of game production.

4.2 Applications

The applications of the Data Cycle detailed in this paper could be reapplied to a wide range of different fields. There are many genres of game that could benefit from the implementation of the DDA Data Cycle. Advertising systems could also utilize a version of the player modeling and rate adjustment steps of the Data Cycle to narrow down a user's most likely next purchase by profiling their preferences across multiple different kinds of products. The system is designed to make guesses based on multiple, continuous variable tracking. There are many places in which the application of this system could be beneficial.

4.3 Future Work

With a larger participants pool, more conclusive evidence could have been gathered on the effects of DDA in PCG.

There are also a few parts of the game that could use redesigning. The game Flow is not perfectly handled through the DDA. Removing the driving mechanics could allow the game to control the players speed independently, thus isolating a very important aspect of the game's difficulty that the player currently has full control over. Another game design tweak would be to increase the number of Levels per Type. Designing the game with even higher levels of difficulty and reward would allow very skilled players to be held in Flow while playing.

Further, the player modeling step of the Data Cycle could be optimized. The system could be reworked so that it is not necessary to compare players to the ExpectedValue matrix. This would be ideal in the advancement of the system because as games go through development cycles, the expected play of users is likely to change with the game.

Bibliography

[1] Jenova Chen. "Flow in Games". *ACM* (2007).

[2] Travis Fort. "Controlling Randomness: Using Procedural Generation to Influence Player Uncertainty in Video Games". *HIM* (2015).

[3] Alexander Zook and Mark O. Riedl. "A Temporal Data-Driven Player Model for Dynamic Difficulty Adjustment". *AIIDE* (2012).

[4] Penelope Sweetser and Peta Wyeth. "GameFlow: a model for evaluating player enjoyment in games". *Comput. Entertain.* (2005).

[5] Olana Missura and Thomas Gartner. "Player Modeling for Intelligent Difficulty Adjustment". *Springer-Verlag, Berlin, Heidelberg* (2009).

[6] Georgios N. Yannakakis, Pieter Spronck, Daniele Loiacono, and Elisabeth Andre. "Player Modeling". *Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik* (2013).

[7] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N. Yannakakis. "What is Procedural Content Generation? Mario on the Borderline". *ACM* (2011).

[8] Ricardo Lopes, Elmar Eisemann, Rafael Bidarra. "Gameplay semantics for the adaptive generation of game worlds". *IEEE* (2017).

Appendix

Questionnaire

Researcher's Name: Charles Calder
Project Title: Adaptivity in Procedurally Generated Environments
Advisor's Name: Robert McGrail

If you have any questions, please ask the researcher at any time.

WRITE GAME SEED HERE:_____
Do you enjoy playing video games?

On average, how many hours per week do you spend playing video games?

Of that time, how many hours do you spend playing games that rely on reaction-time based skill? (e.g. bullet-hell, First-Person-Shooter, platformer, etc.)

Do you prefer games designed for keyboard/mouse or controller?

On a scale from 0 - 10, evaluate the following statements. 0 means completely disagree, 10 means completely agree.

The game was frustrating

0 1 2 3 4 5 6 7 8 9 10

The game was difficult

0 1 2 3 4 5 6 7 8 9 10

The game was boring

0 1 2 3 4 5 6 7 8 9 10

The game was repetitive

0 1 2 3 4 5 6 7 8 9 10

I got used to the controls easily

0 1 2 3 4 5 6 7 8 9 10

Below, please list the InObs from the game [Ramp, Speed Boost, Destructible Wall, Spike Pit] in order of preference. First is the most prefered, last is the least prefered.

Game Controls and Instructions Sheet

Researcher's Name: Charles Calder
Project Title: Adaptivity in Procedurally Generated Environments
Advisor's Name: Robert McGrail

Controls:

Action	Keyboard	Xbox Controller
	In Game	
Accelerate	W	Right Trigger
Reverse/Brake	S	Left Trigger
Turn Right	D	Right Analog Stick (Right)
Turn Left	A	Right Analog Stick (Left)
Jump	Space	A Button
Drift	Right Option/Alt	X Button
Reset Position (If Stuck)	Delete	Y Button
Flip Forward In Air	Up Arrow Key	Right Analog Stick (Up)
Flip Backward In Air	Down Arrow Key	Right Analog Stick (Down)
Turn Clockwise In Air	Right Arrow Key	Right Analog Stick (Right)
Turn Counterclockwise In Air	Left Arrow Key	Right Analog Stick (Left)
	UI Controls	
Up, Down, Left, Right	Arrow Keys	Right Analog Stick
Enter	Return	A Button

GROUP A GROUP B

Please read all Instructions before beginning the test. If you have questions at any time, ask whomever is running the test for help.

Instructions:

- Find a computer that no one is using and turn it on.
- Go to the Applications folder and open the Adapt_PCG application.
- In the opening screen you will see four buttons, Tutorial, Start A, Start B, and Quit. Check the top of this page to see which group you are in (A or B).
- Prepare yourself by looking over the controls sheet. When you are ready, press the Tutorial button.
- You will have a maximum of 10 minutes for the tutorial. During this time, you have the opportunity to learn about the mechanics of the game and get used to the controls. When the timer runs out, you will be returned to the main menu. When ready press the button the relates to your group (A or B).
- Part one will be a 10 minute round. When the timer runs out, you will be prompted to start part two. Press Yes to continue.

- Part two will also be a 10 minutes round. When the timer runs out, you will be prompted to start part three. Press Yes to continue.
- Part three is the last part of the test and will also last for 10 minutes. When the timer runs out, a number will be displayed on the screen. Please copy this number onto your questionnaire exactly.
- You may now quit the game and fill out your questionnaire. When you are finished please bring the questionnaire to whomever is running the test.

Consent Form

Researcher's Name: Charles Calder
Project Title: Adaptivity in Procedurally Generated Environments
Advisor's Name: Robert McGrail

I am a student at Bard College and I am conducting research for my Senior Project. I am studying the effect of an adaptive algorithm on a player's overall engagement and skill progression while playing a game designed around PCG.

During this study, you will be asked to play the game I have designed, then answer some questions about the game. The play time and questionnaire are designed to last approximately 35 minutes. The testing will take place in an empty classroom at Bard College.

There are no direct benefits to the participant.

All participants will be entered into a lottery, the winner of which will receive a 50 dollar gift card to Taste Budds in Red Hook.

Potential risks of participation include minor eye or finger strain from playing a videogame for 30 minutes. If at any point you feel uncomfortable, please tell me and we can stop the test and/or you can skip the questionnaire. You will still be entered into the lottery for the gift card.

All the information you provide will be confidential. You will be asked to copy a number from the screen onto your questionnaire in order to have unique identifiers. I will keep my data secure in a password-protected file on my personal computer. Only my faculty adviser and I will have access to this information.

Participant's Agreement

I understand the purpose of this research. My participation in this test is voluntary. If I wish to stop the test for any reason, I may do so without having to give an explanation.

The researcher has reviewed the individual and social benefits and risks of this project with me. I am aware the information will be used in a Senior Project that will be publicly accessible at the Stevenson Library of Bard College in Annandale, New York. I have the right to review, comment on and withdraw information prior to January 1st, 2017.

The information gathered in this study is confidential with respect to my personal identity. I understand that complete confidentiality cannot be guaranteed, since the researcher may be required to surrender data if served with a court order.

The final project resulting from this study will be permanently and publicly available in the Bard College library and online through the DigitalCommons.

If I have questions about this study, I can contact the researcher at cc9431@bard.edu or the faculty adviser at mcgrail@bard.edu. If I have questions about my rights as a research participant, I can contact the chair of Bard's Institutional Review Board at irb@bard.edu.

I have been offered a copy of this consent form to keep for myself.

I am at least 18 years of age and I consent to participate in today's test.

_____

Participant's signature

_____

Date

_____

Participant's printed name

_____

Researcher's signature

If you would like to be included in the lottery for the 50 dollar Taste Budds gift card, please print your email below. I will only use this information to randomly pick the winner and inform them of their winning status. You do not have to fill this out if you do not wish to give away this information. It will not affect the research in any way.

_____

Participant's Email