GypPO

GypPO: A DSL AND CODE GENERATOR FOR PLATFORMER GAMES

BY
PAVANJOT GILL, B.Eng.

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF ENGINEERING

Master of Engineering (2018)
(Computing & Software)

McMaster University
Hamilton, Ontario, Canada

TITLE:	GypPO: A DSL and Code Generator for Platformer Games
AUTHOR:	Pavanjot Gill B.Eng. (Software Engineering & Game Design), McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Jacques Carette
NUMBER OF PAGES:	vii, 47

Executive Summary

Forming genres for video games is a means of grouping their similarities. This grouping allows for certain expectations regarding the elements or mechanics that would be found in any game of a certain genre. Specifically for 2D platformers, prior to even playing the game, players would expect that the levels will contain platforms and likely a jump ability. The goal of the GypPO (a unique acronym generated for Generative Platformers) project is to determine if there is enough in common amongst the 2D platformer family to create a domain-specific language (DSL) to define a complete platform game and a generator to create them.

By observing 2D platformers throughout the years, from the pioneers of the genre to modern games, it becomes clear which components have stuck and become the norm for development today. It is these types of commonalities that are used to design the GypPO language.

The developed GypPO language is derived from the analysis to ensure all commonalities found in platformers are covered. A user of the GypPO system first needs to provide the specifications for the enemies, upgrades, and weapons that would be found in their platformer, written in the designed DSL. To design the platformer levels, the user specifies the win condition of a level as well as the placement of game objects. This data includes platform placement and the positioning of the elements predefined by users.

The generator stores this specification data into data structures representing the GypPO DSL. With another set of data structures representing the JavaScript language, the GypPO system translates the data from the GypPO DSL to JavaScript. The output of this system is a generated browser-based 2D platform video game.

This report is organized so as to first provide the necessary background information on video games and their genres as well as brief explanations on DSLs and code generation. The analysis of the platformer genre follows, which lays the foundation of the language and generator. By discovering the commonalities of platformers and grouping them together, the terms and structure of the GypPO language are directly formed. The remainder of the report provides a look at the designed GypPO DSL and generator.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Video games with common functionalities and elements can be categorized under the same *genre*. GypPO, a unique acronym for "Generative Platformers", is a research project focused on determining if games falling under the same genre can be generated. The research process is broken into 2 parts. The first is to determine if these family of games are similar enough for a Domain Specific Language (DSL) to be created, which encapsulate these games' core elements and logic. The second portion of this research project is to generate the game code based on specifications defined using the DSL.

A video game genre is a broad term without a standard definition for which games can fall under a genre, chapter 2 will explain how genres are defined for the purpose of GypPO. There are a large variety of different genres, so to narrow the scope of this project, the 2D Platformer family of games will be the main focus. By analyzing a number of Platformer games, the common game components defining a Platformer are discovered leading to a complete coverage of the domain's main mechanics and elements in the designed DSL.

The DSL is created with the intention of reallocating and shortening the time developers spend creating their 2D platformer game. With the use of the GypPO DSL and code generator, developers can spend less time in the implementation stage of core platformer functions and focus more on the design of their game. Based on the analysis of Platformers, GypPO generates the common elements and logic of Platformers, thus allowing the developer's focus to be solely on the aspects of their game which make it "unique" (be it art or a unique game mechanic). GypPO's target group of users is very broad, thus by designing the language to be as readable as possible, knowledge of programming video games is not necessary in order to define a game using GypPO and having it generated.

It is important to note that GypPO is an extension of the work previously done by Collman (2014) on generating Bullet Hell games with MAKU.

Chapter 2

Video Game Genres

Genres are used to categorize things based on similar attributes. These attributes can range from content to art-style. With respect to games, Adams (2014) defines genres based on similar gameplay elements/challenges. The issue with video game genres is that there is no standard definition as to what falls under the different video game genres. The existing guidelines for genres are based on what a game is thought to fall under by either the creators or the players. As games begin to get bigger, they have game challenges which can fall under multiple genres, making hybrids that are hard to categorize. For example, *Grand Theft Auto V* (2013) is labeled as an action game but it consists of elements of shooters, racing, and sports games.

Common genres for video games are shooters, action, sports, strategy, role-playing, simulation, and adventure. These genres are further broken into subgenres to group games together more accurately. This is to avoid the issue where games like *Risk of Rain* (2013) and *Soulcalibur* (1998) can both be considered action games. However, they have very little in common to be useful for players to determine what sort of mechanics/challenges these games have. With subgenres, these games can be labeled correctly as a platform game (*Risk of Rain*) and a fighting game (*Soulcalibur*).

As for the GypPO research project, the focus is on the subgenre of Platformers rather than the entire broad action game genre. The games used for a detailed analysis, found in chapter 5, were gathered based on using common gameplay mechanics and elements to define what the platformer genre is.

Chapter 3

Domain Specific Languages

This chapter provides a brief introduction into Domain-Specific Languages (DSLs) and the importance of domain analyses. Most of the information is obtained from the existing work of Mernik *et al.* (2005), Fowler (2010) and Collman (2014).

3.1 What are DSLs?

Mernik *et al.* (2005) define Domain-Specific Languages (DSLs) as "languages tailored to a specific application domain." A very particular domain is analyzed and through this analysis, a language can be designed which solves problems belonging to this domain. This differs from the more well-known General Purpose Languages (GPL), such as C, Java, and Python, which are not constrained to any one domain. GPLs, as their name suggests, are useful when applied to a variety of general problem domains. Whereas the sole purpose of a DSL is to solve one very specific problem, making them less useful in solving problems belonging to other domains. Some existing DSLs are the Excel macro language for spreadsheets, and $\LaTeX$ for typesetting (Mernik *et al.*, 2005).

To develop a DSL, Mernik *et al.* (2005) breaks the process into steps: identify and analyze the application domain, then design and implement the DSL. The foundation of a well-designed DSL is a detailed domain analysis, which, if done well, can translate directly to creating the vocabulary of the DSL. After this task, design decisions need to be made regarding how the DSL should be created, such as whether the DSL should be an extension of an existing language or its own standalone language.

3.2 Domain Analysis

As a DSL is intended to solve a specific problem domain, an in depth understanding of this domain is crucial. The domain could be analyzed informally or formally using existing methodologies such as DARE (Domain Analysis and Reuse Environment) developed by Frakes *et al.* (1998). Analyzing existing source code related to the application domain can be used in both the informal and formal analysis and provides good insight into how existing solutions go about solving the domain problem (Mernik *et al.*, 2005). Through the use of the domain analysis, common terminology used in the domain will help with creating the syntax of the DSL.

3.3 DSL Design and Implementation

Once the domain analysis is done, an important decision needs to be made regarding the creation of the DSL. Fowler (2010) separates DSLs into "internal" (embedded) and "external" (standalone) DSLs. These differ with regards to their connection to other general purpose languages. Embedded DSLs are created using an existing GPL as the foundation and the DSL is simply an extension of the language, whereas standalone DSLs are independent of a GPL (Mernik *et al.*, 2005).

The use of a host language has both its advantages and disadvantages when creating a DSL. This approach has the benefit of the DSL having access to already existing features of the host language. The DSL also does not need a new compiler to be created as the GPL will already have one. The issue with the embedded approach is the lack of flexibility in creating syntax that closely matches the domain terminology, as the freedom to use domain-specific notations is restricted by the GPL (Mernik *et al.*, 2005). Another problem is that the users of the DSL will require some knowledge of the underlying GPL in addition to knowledge of the problem domain to use the DSL correctly.

Standalone or external DSLs are made with no reliance on any existing GPLs. These DSLs' syntax is not constrained by any underlying GPL. The terminology found in the domain can be freely used to form the language (Mernik *et al.*, 2005). The advantages to this approach is that the DSL can be designed to closely resemble natural language. This allows users to easily read and write a file written in the designed language without the requirement of knowing how to program in a GPL. The downside to this approach is that it is more complex and time-consuming to develop as the DSL creator requires knowledge of compilers to create one for their designed language (Mernik *et al.*, 2005).

3.4 Relevance to GypPO

While both types of DSLs have their own merits, the GypPO DSL was created with the idea that its syntax will closely resemble the common terms used to define a platformer. The domain analysis for GypPO, found in chapter 5, is done informally through playing and observing gameplay as well as examining source code. From the domain analysis step, many keywords used in describing platformers are found to create a language that is able to clearly define a platformer game. Therefore, by making an external DSL, GypPO can be readable and used by domain users without the need to first learn how to code using a host general programming language. While the difficulty increases when creating a new compiler/processor for GypPO, it is necessary to increase the readability of the created DSL.

Chapter 4

Code Generation

This chapter provides a brief introduction into code generation. It focuses on the general information about the generation process, leaving the implementation details to chapter 8, the GypPO code generator. This information on code generation is obtained from the earlier work of Czarnecki and Ulrich (2000), Mur (2006), Fowler (2010), Szymczak (2014) and Collman (2014).

4.1 What is Code Generation?

Code generation is the automated production of source code from an input file written in a higher-level language (Mur, 2006). With regards to GypPO, the higher-level language is the DSL created for describing the platformer domain. In essence, a generator, such as a compiler for example, takes an input file written in a higher-level language (such as C or Java), processes the input file, translates to and outputs the source code (machine-readable code for compilers) (Fowler (2010), Czarnecki and Ulrich (2000)). For GypPO, instead of generating machine-readable code, the generator creates JavaScript source code based on the specifications detailed by an input file written in the DSL.

4.2 Process of Generating Code

From the information obtained from Czarnecki and Ulrich (2000) and Fowler (2010) the process of code generation could be generalized into 5 steps:

1. Compile the generator framework and parser

2. Parse the high-level input file

3. Process the parsed information

4. Translate the processed information into the target output language

5. Print/Output the translated code

4.3 Why Generate Code?

Code generation is useful for being able to take a higher-level language input file and generate code in any target output language (Mur, 2006). For the case of using domain-specific languages as the high-level language, the first three steps of the code generation process stay the same regardless of any changes to the target output language. The translation to, and printing of the output file are the steps which require alterations depending on the target language of generation. While chapter 8 goes into the specifics of how GypPO generates the game code from the DSL, an important prerequisite to the translation step is to have an abstract syntax tree (AST) for JavaScript. This tree provides a representation of the syntax of the languages necessary to generate working code which is written correctly in JavaScript. By swapping the AST of the target language by an AST of another, and modifying the translator and printer accordingly, the code generator can generate to any language without needing to alter the input file and parser. This allows for users to focus their efforts on the domain problem and correctly providing specifications through the DSL, while relying on the generator to produce the source code in whichever language is required.

The downside to using code generation is the added complexity of generated code compared to handwritten code which performs the same functions. Generated code may be harder to read and the increased complexity can lead to an increase in difficulty debugging and testing the generated code (Szymczak, 2014).

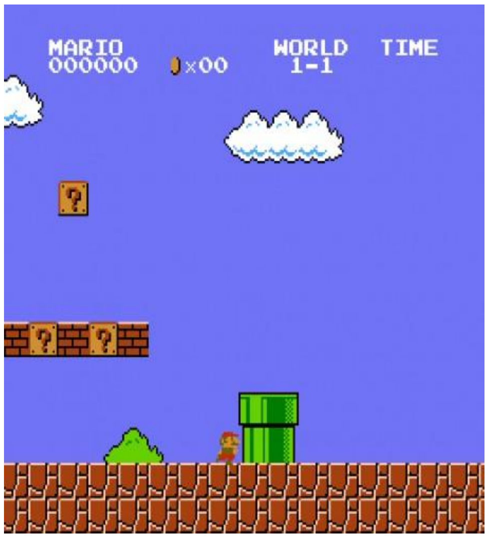
Chapter 5

Platform Games

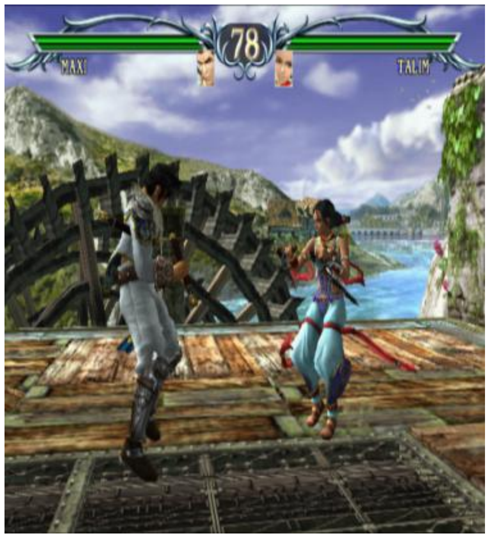
Based on the Adams (2014) definition of games being grouped by common gameplay challenges to form genres, modern games are becoming hybrids of many different genres with a variety of challenges. A testament to this are games known as Platformers. As Adams (2014) provides descriptions of various game genres, the platform section is relatively short, stating that the commonalities of these games consist of "unrealistic physics" (jumping) and platforms to traverse in a game level. Therefore, all other challenges of these games would come from other genres (commonly 2D shooters). This brief description of the domain of platform games is not detailed enough to provide the required knowledge to form a complete domain-specific language. Thus, this chapter thoroughly analyzes Platformers in the hopes of having a better understanding of what this domain is. The complete list of games used for this analysis are found in Appendix A. Some games were very briefly and informally looked at while others were analyzed more thoroughly providing a clearer picture of the commonalities of platformers.

5.1 Action Games

Before conducting an in depth analysis of platformers, it is important to first understand the parent genre, action games, that platformers fall under. These games are ones which require some form of physical skill to overcome their gameplay challenges (Adams, 2014). With this general definition, action games encompass a variety of different games, leading to the creation of subgenres for more accurate groupings of games. Some action subgenres and their physical challenges include: platformers - jumping from one platform to another while overcoming obstacles, fighting games - requiring reaction time and timing skills to combo moves and attacks, shooters - need hand-eye coordination and reflex skills to accurately aim and shoot at enemies before they shoot back.



(a) Super Mario Bros.

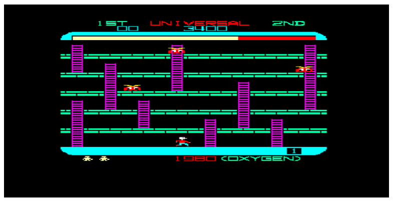


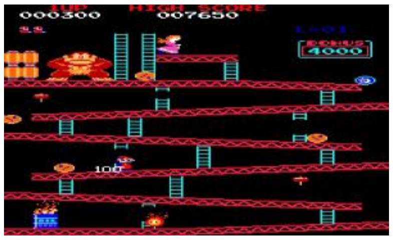
(b) Soulcalibur 3

Figure 5.1: Two very different action games

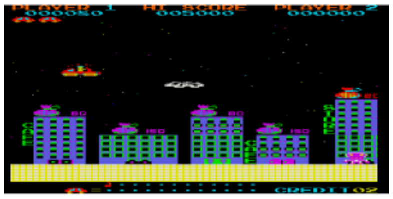
5.2 Platform Games

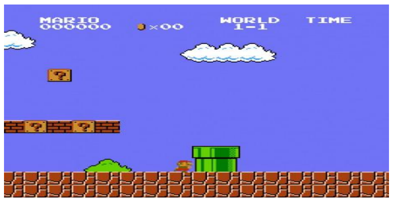
Platformers have been a part of the gaming world since the 1980s, where they became big hits in arcades. Arguably, beginning with the 2D arcade game *Space Panic* developed by Universal Entertainment Corporation (1980), the platform genre went through a roller coaster of popularity, picking up throughout the 1980s and 1990s. 2D platformers in particular, the focus of GypPO, had a flux in popularity throughout the years. With the introduction of 3D platformers, 2D platformers took a backseat until the release of modern games such as *Risk of Rain* (2013), and *Cuphead* (2017) bringing a revival of the 2D platformer genre. While *Space Panic* can be considered the first platformer, the game lacks the fundamental elements that define the platform genre today. The two defining game elements of platformers are the jumping mechanic, and platforms to traverse by means of jumping between them. While *Space Panic* has a terrain which can be considered a "platform", it extends the entire width of the screen with travel between platforms constrained to ladders placed around the level. It was not until *Donkey Kong* was released by Nintendo R&D1 (1981), the most well known pioneer of the genre, that jumping was introduced and became the norm. Another element of platformers that is now expected, but not mandatory, is the placement of collection items around a level for the player to acquire. These items are sometimes included as bonus score increases while other times they are the objective of a game level.


(a) *Space Panic* (Universal Entertainment Corporation, 1980)

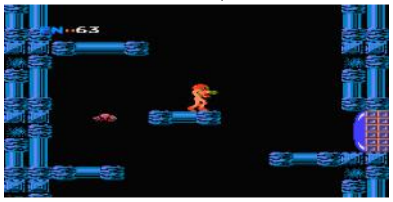


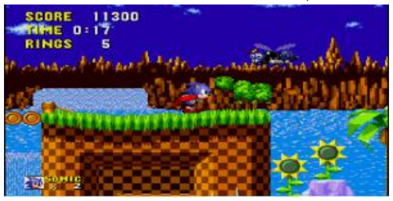
(b) *Donkey Kong* (Nintendo R&D1, 1981)


(c) *Jump Bug* (Hoei and Alpha Denshi Corporation, 1981)

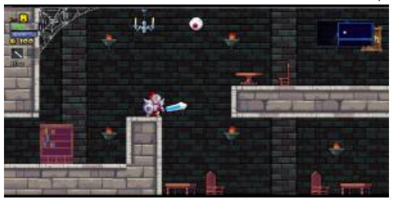


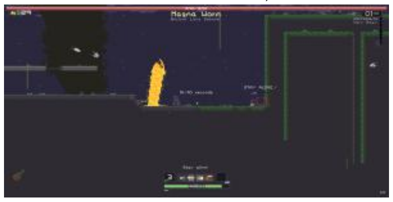
(d) *Super Mario Bros.* (Nintendo Creative Department, 1985)

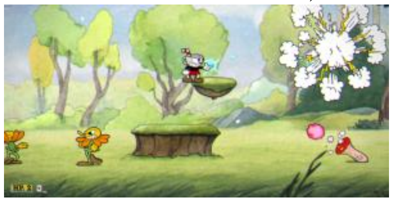

(e) *Metroid* (Nintendo R&D1 and Intelligent Systems, 1986)



(f) *Sonic the Hedgehog* (Sonic Team, 1991)


(g) *Rogue Legacy* (Cellar Door Games, 2013)


(h) *Risk of Rain* (Hopoo Games, 2013)


(i) *Cuphead* (StudioMDHR, 2017)

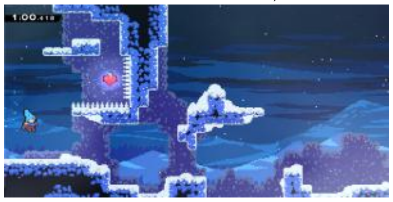

(j) *Celeste* (Matt Makes Games, 2018)

Figure 5.2: Screenshots of different Platformer games

When creating a 2D platformer, the first important design decision to be made is regarding the camera type. The three camera options used for platformers are **fixed screens** in which the entire level is displayed on the screen, **one-way scrolling** where the camera can only move in one direction preventing the player from traveling back, and **free-scrolling** where the camera focuses on the player avatar in the centre of the screen but has no constraints as to which directions it can move in.

In Platformers, the player is in control of a single entity on the screen, labeled as the **protagonist** which needs to traverse over the platforms in a single level to achieve a win condition. These win conditions can be broken down into four different objectives: **collection** of some game object, **destroying all enemies**, **eliminating a specific "boss" enemy**, or **reaching the end of the level**. Using the games shown in figure 5.2 as samples representing the platformer genre, many observations and commonalities can be found between them. As stated above, while jumping may not have been a feature belonging to the first platformer, it became a must in most platform games following *Donkey Kong*. These protagonists have the ability to jump to allow players to move across the platforms. Most of the platformers in 5.2 have these platformer-defining elements but also include some other gameplay mechanic that is usually found in other types of games. Commonly this is an attack method to defeat enemy entities in the levels. This attacking method is through the means of a weapon (the protagonist can be a weapon if, like *Super Mario Bros.* (1985) 5.2d, jumping onto enemies is an attack). *Celeste* (2018) 5.2j, on the other hand, does not provide the players with an attack method to destroy enemies as the focus of that game is on having fluid movement mechanics and a dash ability to increase the distance a player can reach. *Celeste* therefore, requires good level design to make the most of the basic platformer movement mechanics. With the inclusion of a dash feature, *Celeste* increases the possibility of where platforms can be placed, increasing the difficulty of getting to the end of a level.

There are various different types of platforms found in the levels of these games. These platforms include simple, **stationary platforms**, **moving platforms** - requiring players to time their movement to reach the next platform, and **one-way platforms** that allow players to phase through them when moving in one direction but not allowing the player to move back in the direction they came from. To add to the variety of platforms used in these games, any of these platforms could have **size variations**, with smaller platforms requiring players to have more skill to successfully reach them.

While some games in 5.2 may not appear to have any form of enemies, that can only be true if we are considering the traditional enemies found in games. Every game shown in 5.2 has some form of enemies labeled as **antagonists**. While most of these games have entities which actually attack the player, thus being traditional enemies, other games have elements in their environments that still perform the same task as these common enemies (kill the protagonist). *Celeste* (2018) 5.2j may appear to lack any typical enemies, but the obstacles found in *Celeste* perform the same functions as enemy entities, thus they are also labeled as antagonists. Each antagonist consists of a **movement pattern** and an **attack type**. While the antagonists that are just obstacles or traps for the players may not appear to have any form of movement or attack, being **stationary** is one common option for antagonists' movement pattern, while just coming into contact with a protagonist is a form of attack.

5.3 Detailed Analysis of Platformers

With section 5.2 providing an overview of platform games, this section expands on the commonalities found through analyzing the games in Figure 5.2 and Appendix A.

5.3.1 Entities

Entities of a platformer are split between **protagonists** and **antagonists**. The biggest difference between these 2 types of entities is whether they are player-controlled in movement and attacking or have specific patterns assigned to them. The other difference is an optional **score** that can be associated with an antagonist when they are destroyed, a feature found in *Jump Bug* (1981) 5.2c and *Super Mario Bros.* (1985) 5.2d. Through observation of the games of 5.2, the common properties of platformer entities were discovered and used to form an understanding of what defines an entity. Table 5.1 shows these common properties.

Property	**Description**
Health	The initial health of the entity.
Lives	The number of lives the entity has before being permanently destroyed.
Speed	Used by the engine to provide the correct movement speed to the entity upon creation.
Spawn Location	A set of coordinates of where to place the entity at the start of a level.
Linked Weapon	The name of the weapon which the entity uses to attack. Optional.

Table 5.1: The common properties used by all entities in Platformers

Health and lives of entities are easy to understand, games like *Metroid*

5.2e, *Rogue Legacy* 5.2g, *Risk of Rain* 5.2h, and *Cuphead* 5.2i explicitly show the player's health properties and lives, whereas the others, while not actively showing these properties, still have them. In the games where the health property is hidden, the game still assigns a value to each entity. Most of these games have a "1-hit, 1-kill" policy, where coming into contact with any antagonist results in death. For these games the health property is considered to be set at a value of 1, while the damage value of the attacking weapon is also considered to be at 1 health unit. Lives, like-wise may be shown like in *Super Mario Bros.* or can be hidden. Any of these games where an entity death leads to respawning rather than permanently having the entity destroyed (if it is the protagonist, this leads to losing the game) means that the game assigned a number of lives to the entity which is greater than one. For entities with no respawning, this means that the life value is just one.

The speed value is required for any moving object within a game. The game engine requires this property to move an entity at the correct speed. Spawn location is another property required by the game engine. The engine needs each entity to have intial position coordinates so as to place these entities in their correct positions in a level.

The last common property found for entities is a weapon linked to the entity. Section 5.3.3 goes into details regarding weapons found in platformers but the "linked weapon" property for entities is to specify which weapon/attack method an entity uses. This is an entirely optional property as not every platformer has weapons and attacking as a gameplay feature. *Celeste* has no attack methods and weapons, however all the others in Figure 5.2 have some form of a weapon. Games such as *Sonic the Hedgehog* and *Super Mario Bros.* both use the protagonist's body as a weapon (more specifically the avatar's feet) as jumping onto enemies is a form of attack. Similarly, when player entities come into contact with enemies, the player takes damage, indicating that the enemy bodies are also considered weapons.

Recognizing that both protagonists and antagonists are very similar is key to the creation of the game engine for GypPO. This is to abstract specific platformer elements from the engine and leave this information to the DSL and code generator.

5.3.2 Antagonists

While subsection 5.3.1 discusses the commonalities found between all entities in platform games, this section dives into further detail regarding the unique properties of the antagonist entities. These are the movement patterns and attack patterns followed by antagonists while still alive in a level. The attack patterns are left for the weapons section 5.3.3 to discuss. To understand movement of antagonists, it is broken into five properties: **pattern** - movement pattern assigned based on table 5.2, **tracking** - whether or not the antagonist can track the protagonist and always face towards the protagonist (useful for attacking as an enemy with tracking will always attack in the direction of the protagonist), **speed** - discussed in the entities section, **flying** - whether or not the antagonist is ground-based or in the air, and **"vee" movement** - which is whether or not the entity moves in a "v" like pattern. This pattern occurs when an entity is moving towards the left or the right of the screen but jumps each time the entity lands on the ground, moving this way repeatedly will form a v-shaped pattern. Flying entities could also have this movement pattern but they need a maximum and minimum elevation value to correctly form the v-shape. The "vee" pattern could have been a movement pattern itself, but an antagonist with "vee" movement still requires a defined movement pattern of the four shown, thus "vee" is its own property rather than another pattern. The patterns found in table 5.2 are based on observations of enemy movement in the games of figure 5.2.

Pattern	**Description**
Patrol	Antagonist moves between two coordinate points in the level.
Charge	Antagonist moves in the direction of the player.
Still	Stationary antagonist.
Random	The direction the antagonist moves in changes randomly.

Table 5.2: Movement patterns used by antagonists

5.3.3 Weapons

Weapons can be found in all Platformers that also have killable enemies. In subsection 5.3.1, the "linking weapons" property is there to link an entity to the weapon it uses. The most important property of these weapons is the **damage** dealt by the weapon when it attacks and comes into contact with an entity. For games such as *Risk of Rain*, these weapons have a fixed weapon value that the entity's health value is decreased by. The games with no explicit health property, leading to a fixed value of 1, means that the damage value of weapons is also fixed to 1. This creates the "1-hit, 1-kill" policy mentioned above. The other common attributes of the weapons in platformers are **range**, **rate of fire**, **speed**, and **attack type**. The weapons used by these games are very simple, divided into 2 types: melee weapons and ranged weapons. These weapons have a specific range value that sets the furthest distance a weapon can reach before being destroyed. For melee weapons, this range value specifies the length of the melee weapon. For ranged weapons, this value provides the game with a maximum distance a projectile/bullet can travel before it is destroyed. The rate of fire property is the frequency at which the weapon can fire. This property is found to be common with any weapons that are external from the entity's body. In *Super Mario Bros.* and *Sonic the Hedgehog*, the melee weapon is the protagonist's body, so there is no actual firing of a weapon. Similarly, speed is another property only required by external weapons. This value provides the game engine with how quick a projectile should travel or how fast a melee weapon swings/stabs.

Property	**Description**
Damage	Amount of health that is decreased upon coming into contact with an entity.
Range	The maximum distance the weapon can reach from its initial firing location.
Rate of Fire	Frequency at which the weapon can fire.
Attack Speed	The speed of the attack\projectiles.
Attack Type	Weapons can be either melee or ranged.
Attack Pattern	If the weapon is ranged, a projectile pattern needs to be defined.

Table 5.3: The common properties of weapons in platform games

As the weapons used in platformers vary between melee and ranged, additional commonalities can be found between weapons of the same type. As discovered for the movement patterns of antagonists, projectiles fired from ranged weapons also have attack patterns they follow. The common patterns used by these games are described in Table 5.4. One interesting weapon property discovered is that the modern games (2010s) and *Sonic the Hedgehog* also have weapons which can fire multiple projectiles at once. These weapons still require the same properties defined for single-shot, with one additional piece of information which is the directions that these bullets are fired. Without the different bullets having different directions to move in, they would overlap and appear as a single bullet on screen.

Pattern	**Description**
Straight	Projectile fired in a straight line in the direction the entity is facing.
Arc	Projectile fired in a curve in the direction the entity is facing.
V-attack	Projectile fired with a “v” pattern.
Homing	Projectile tracks the closest entity and moves to their tracked position.

Table 5.4: Movement patterns used by projectiles

5.3.4 Upgrades

Upgrades are the game element which most differ between the different platformers analyzed. The upgrades, which are common to platformers, are **weapon upgrades**, **health/life upgrades**, and **score upgrades/collectibles**. As games like *Rogue Legacy* and *Risk of Rain* have the win condition of beating a boss antagonist to win a level, adding these weapon and health upgrades provide the player with some progression and reward for overcoming the challenges. As weapons could vary in terms of the properties stated above, the way of playing can also change depending on which weapon a player is using. A player with a melee weapon may approach the game with more caution as they need to get in close to an enemy, risking death, in order to attack. With weapon upgrades in the game, this allows players to have a choice in which weapon they use, rather than just sticking with the original weapon assigned to the entity.

The collectibles have become the most famous upgrade within platformers. Beginning with *Donkey Kong*, these collectibles provide the players with another objective besides just getting to the end of a level or defeating an enemy. They are a must to have in platformers since their introduction. With games that keep score, attaining these collectibles provides the player with an incentive to increase their score. *Jump Bug* interestingly uses the collectibles in their game as a means to increasing the player’s lives. These collectibles also lead to the unique upgrades that are not common to all platformers. *Rogue Legacy*, *Risk of Rain*, and *Cuphead* use coins as both a collectible item as well as currency. By providing a form of currency, these games introduce economies that result in the currency being exchanged for other upgrades. All three of these games provide weapon upgrades but they add a cost property to these weapons. *Rogue Legacy* goes beyond by adding stat based upgrades amongst other things to the game. This results in the currency being used to exchange for more than just weapons. The game adds armor (a way of negating some amount of damage), mana as a resource to use special types of attacks, and new game mechanics which are all considered upgrades.

5.3.5 HUD

The heads-up display (HUD) of these platformers can be seen in the screenshots found in figure 5.2. Games like *Risk of Rain* provide a lot more information to the player compared to the others. The HUDs only display the most important information required by the player. This usually includes the protagonist's health, life, and score information. However, a HUD should only show information that is relevant to the game. If score is not an implemented feature, then of course it would not be shown on the screen. It is interesting to note that the latest two games have opted for a minimal HUD. *Cuphead* only needs to show the health of the player, and their power-up level for a special attack. *Celeste* was developed with consideration for speed-runners, so a timer is shown on the upper right portion of the screen. These games show a counter for collectibles when the player picks them up but they are only shown for a limited time before going away.

5.3.6 Levels

Levels in platformers are where all of the above subsections come together. When creating a level, a developer is required to define the camera model used, every spawn location of entities, the placement of platforms, the placement of upgrades, and the win condition. Platforms can consist of stationary platforms, moving platforms, as well as some boosted jump platforms like the springs found in *Super Mario Bros.* which allow the player to jump across large gaps. The moving platforms used in these games use the patrol pattern similar to the one used by antagonists. They move from one point to another and repeat. *Celeste* also has clouds as platforms that act like the boosted jump platforms with the added difficulty of disappearing as soon as they launch the player up, not allowing the player to come back down onto the platform. Most platformers have hand-crafted levels designed by the developers, but as procedural level generation is becoming more popular, some modern games are changing the way levels are created. Both *Rogue Legacy* and *Risk of Rain* have randomly generated levels, resulting in different levels being played in each playthrough of these games.

5.4 Leading to the DSL

This detailed analysis on the platformer genre of games is crucial to design a DSL that is a complete representation of this domain. Chapter 6 discusses the GypPO DSL and shows how the information obtained from this chapter translates to the formation of a domain-specific language. The common terms found to describe and analyze platformers are used as terms in the GypPO language. A summary of the variabilities found from the analysis is in Appendix B. These variabilities were discovered with help of the Family-Oriented Abstract, Specification, and Translation (FAST) approach developed by Coplien *et al.* (1998), the work of Ardis and Weiss (1997), and from the commonalities analysis on mesh generators done by Smith and Chen (2004).

Chapter 6

GypPO DSL

As explained in the introduction to domain-specific languages in chapter 3, the domain analysis in chapter 5 directly translates to the terminology used in a DSL. The resulting product is the GypPO DSL, a text-based language with a list of keywords related to the domain, used to create a specification file describing a platform game. Each section of the language is white-space sensitive to clearly separate the different elements that make up a platformer. GypPO uses Haskell to implement the data types representing the syntax of both the designed language as well as the target JavaScript language. The ASTs were created to closely resemble the languages, allowing for the system to efficiently parse and process the information from the input specification file. Haskell's `Parsec` library is a big help with the parsing of the game specifications.

The language is separated into two sections: the `elements` and the `logic` of a game. The elements portion is used to describe the game objects that can be found throughout the game, not just specific to any one level. This includes the antagonists, weapons, and upgrades. The logic section focuses on the specifications for each level within the game, from win conditions to the placement of platforms and entities. This chapter will go into further detail of what each of these consists of by using small examples of text written in the GypPO language. The explanation of the design decisions regarding why GypPO is organized in this specific way is left for chapter 7.

6.1 Grid System

Before the specifications of any elements or logic of the game, GypPO requires a relative `grid` system to be defined. GypPO uses such a system to provide relative coordinates for placing any elements into the game. As every entity in the game requires an initial spawn location, this coordinate is based on the grid system defined, rather than absolute coordinates. Using a grid system allows for the game to scale to any size without needing to change the placement positions of anything in the game from the specifications file. This method leaves any absolute coordinate placement to the GypPO engine which is the only portion of the GypPO system that needs to know the screen size used to play.

6.2 Elements

The elements portion of the GypPO language is for defining the properties of the weapons, antagonists, and upgrades used in a platform game. An example of each element that can be defined using GypPO is found below as the descriptions are explained. Properties such as `name`, `colour`, `shape`, and `speed` are used for describing all elements of the game. Each element of the game has a name assigned to them used for linking and placing these elements into the levels. There are a variety of colours that the user can choose from; all of which are found in the Haskell colour package. The common property `shape` can be either squares (`square`) or rectangles (`rectangle`). A `triangle` option is also available, however CraftyJS, the JavaScript game engine used to display the generated code, is restricted to only showing squares and rectangles. The triangle shape is only there to create a triangular hitbox for collision detection. These invisible hitbox options are important as they should be used when defining hitbox shapes but using custom art for the game elements. Since the generated game does not create art for a user, it can only use squares and rectangles to have visible game elements. The last common property is `speed` which can be found when defining any moving entity (weapon projectiles, antagonists, and protagonists). The speed property has six options for users to choose from: `rest`, very slow (`vslow`), `slow`, `medium`, `fast`, and very fast (`vfast`). The speed values, when generated, scale with the game size and are based on how much of the screen an entity is able to move within a second. With rest being 0 pixels per second, each successive speed value is an additional 5% of the game screen covered in a single second, with the maximum value being 25% of the game screen being traversed in a second.

6.2.1 Weapons

When defining the elements of a platform game in GypPO, it is necessary to group the various element types together. Figure 6.1 shows two different weapons being defined, where the difference lies in defining the attack type as brought up in the domain analysis. The common properties mentioned above are all found when describing weapons. A name for each weapon is required so it can be used when defining entities through linking a weapon to that entity if necessary.

The remaining properties of defining weapons are exclusive to the weapon elements. These include `damage`, attack type (`melee` or `ranged`), attack range (`atkrange`), rate of fire (`rof`), and `target`. Damage is a non-negative integer indicating how much health of an entity the weapon collides with is taken away by one successful attack. The attack type is the property which can be further defined based on what type is chosen. A melee weapon does not need any further specifications, but as the second weapon in Figure 6.1 shows, a ranged weapon needs the `speed` and `bulletpattern` to be defined. The bullet pattern is an attack pattern selected out of the ones built into the language and generator. The patterns found in Table 5.4 have all been implemented in the GypPO system for a user to select. The keywords for these are: `straight`, `arc`, `vatk`, and `homing`. The attack range is the maximum range in grid units that an attack can reach before being removed from the game. For a melee weapon, it is the length of the weapon element in the game whereas, for ranged weapons this value indicates how far a projectile can travel without colliding with any entity before it is destroyed by the game. Rate of fire is a property for how many times in a second a weapon can fire. This property prevents the enemies or players from firing a constant stream of attacks without any rest between them (unless that is the intention). The last property, `target`, is for the collision functions and homing bullet target-finding. The options for target and their descriptions are found in Table 6.1. Having options for attacking entities or other weapons is to provide some variety to weapons and giving users the options to define weapons with the sole purpose of blocking bullets.

```
weapon
  name "mel"
  damage 10
  colour green
  shape square
  melee
  atkrange 20
  rof 3
  target all
weapon
  name "str"
  damage 20
  colour green
  shape square
  ranged
    speed medium
    bulletpattern straight
  atkrange 40
  rof 1
  target characters
```

Figure 6.1: Example of both a melee and a ranged weapon

Target Option	**Description**
`characters`	Can only collide with entities in the game (antagonists and protagonists). Adjusts entity's health based on damage value.
`weapons`	Can only collide with other weapons in the game. Destroys both weapons.
`all`	Can collide with both entities and other weapons in the game.

Table 6.1: The options for `target`

6.2.2 Antagonists

Antagonists, labeled as `antag`, are the next element of a platformer described in GypPO. Figure 6.2 showcases an example antagonist and what properties are required to completely define the entity. An antagonist must have the common properties name, speed, shape and colour described as well as a few properties exclusive to these types of elements. One property to note that goes along with the colour and shape of the element is the `size` property. With the options being small (`s`), medium (`m`), and large (`l`), these values use the relative grid system to scale the antagonist appropriately for display on screen. This size property is required for the same reason as shapes, due to the lack of art generation, antagonists can have varying sizes that are only displayed on screen by different sized shapes. With game art being implemented, the size property remains important for simple collision hitboxes (the alternative being custom hitboxes with unique polygons). Antag `health` is the initial non-negative integer health value of the entity. This is the value which is always checked and adjusted after collisions with weapons, leading to the removal of the antagonist once this value reaches 0.

The important property for the antagonist is the `movement`, which is further broken down into the properties found in subsection 5.3.2. For the properties which have a simple true or false definition (`flying`, `vee`, `track`), the GypPO keyword for these properties is `yes` or `no`. The GypPO engine has each pattern found in Table 5.2 implemented to allow the ability for GypPO users to assign the commonly found movement patterns of platformers to any antagonist in their game. The list of keywords denoting each of these patterns is: `still`, `patrol`, `charge`, and `random`. As the patrol pattern requires two coordinates for the antagonist to move between, this information is left for the level description as that is where the spawn locations of these antagonists is described.

If an antagonist uses a weapon, the weapon name needs to be linked to a weapon described above in a GypPO file. Every other piece of information regarding the weapon is stored in the weapon definition rather than in the antagonist definition. Lastly, if the game keeps track of score points, an antagonist should have a `score` value to denote how many points are awarded to the player for successfully eliminating this enemy.

```
antag
  name "antag1"
  health 20
  weaponname "mel"
  flying no
  movement
    pattern patrol
    track no
    vee no
    speed slow
  colour blue
  size m
  shape square
  score 100
```

Figure 6.2: Example of an antag defined using GypPO

6.2.3 Upgrades

As discovered in the domain analysis chapter, the upgrades used in platformers are one of the more varying elements between these games. GypPO focused on the most important upgrades, found in Table 6.2, while leaving room for additional upgrades being added as future work with relative ease. An example of defining each of these upgrades is shown in Figure 6.3. The description of each of these elements follows the same pattern. Beginning with the keyword indicating which type of upgrade it is, next is the name of the upgrade, used for the same reason as every other element of the GypPO language, for identifying and placing in a level. The next property for an upgrade is the value assigned to the upgrade. For a `collectible`, this is a score value for how much the player's score is increased by upon pickup. `health` and `life` require a non-negative integer value specifying how much of their respective property to increase for the entity which picks up these upgrades. The `weapon` upgrade requires a name of a pre-defined weapon that will be replacing the entity's linked weapon. The last property for every upgrade is a colour assigned to each upgrade to display in the game screen.

```
upgrades
  collectible "coin" 50 violet
  health "hlth" 50 red
  life "lifeup" 1 yellow
  weapon "newW" "arc" blue
```

Figure 6.3: Example of the different upgrades defined using GypPO

Upgrades	**Description**
Weapons (`weapon`)	Replaces the weapon linked to the entity with the one picked up.
Collectibles (`collectible`)	Collectible pickup item that is one of the most defining elements of a platformer.
Health (`health`)	Increases the entity's remaining health property by a specified amount
Life (`life`)	Increases the entity's remaining lives by a specified amount.

Table 6.2: The upgrades currently implemented in GypPO

6.3 Logic

The logic portion of the DSL is where each level in the generated game is defined. This is where the elements defined earlier would be linked and actually placed into the game. Every level consists of a camera model, platform placement, a protagonist definition and a win condition. The remaining properties in Figure 6.4 (antagonist placement and upgrade placement) are optional as not every level in a platformer necessarily requires any of these. It is important to note that GypPO does not determine whether or not the placement of elements in a level is feasible. The game designer using GypPO is responsible for the feasibility of the placement of their game elements, meaning it is also possible to make levels which are impossible to play (such as spawning over nothing and falling to death). The exception is trying to place elements at coordinates which are not in the grid, GypPO must catch this error.

```
level
  camera fixed
  platforms
    0 1 m green
    1 2 l green
    5 1 m green
    4 3 s green
    4 2 m green
  antags
    "antag1" 1 0
    patrol
      start 0 0
      end 2 0
    "antag2" 2 2
  lvlUpgrades
    "coin" 0 1
    "hlth" 1 2
  protag
    spawn 0 0
    jump 1
    speed slow
    lives 3
    health 100
    colour red
    shape rectangle
    weaponname "hom"
  score 150
```

Figure 6.4: A level defined in GypPO

```
camera fixed
canera one-way 5 5
camera free-scroll 5 5
```

Figure 6.5: Defining all three types of cameras for GypPO

6.3.1 Camera Models

All three camera models discussed in chapter 5 have been implemented into the game engine and the DSL as options. Figure 6.5 shows how to define each of these as well as what data needs to go along with the camera (if any). The `fixed` camera is straightforward to define and implement. The entire level is scaled to fit on the screen without any need for scrolling. The other two camera models each require an X and Y value indicating how many grid units of the relative grid system are in the view.

6.3.2 Platform Placement

The placement of the platforms is the most important element of this game genre. The current method of placing platforms is by listing every platform required by a level with its placement coordinates (in the relative grid system), the size of the platform and the colour. By providing the platform location in the coordinates of the grid system used by GypPO, the generator and engine are tasked with translating these to absolute coordinates of the game screen. The next value of a defined platform is the size (`s`, `m`, `l`). The three options are small, medium, and large, with each taking a certain number of grid units. The small platforms take up one grid unit in width, with each increasing size adding another grid unit to the length of the platform. The colour data is the same as all other definitions of colour in GypPO, being required when the game element does not have any art to show, so it must have a colour instead. While this may not be the ideal way of placing platforms in games, improved methods are discussed in the section titled "Future Work".

6.3.3 Antagonist and Upgrade Placement

The placement of elements defined in the elements portion of a GypPO DSL file is fairly straightforward. With the exception of antagonists with the patrol pattern, all other elements require the name of the defined element (antagonist or upgrade) and their initial position in the level. The exception, patrol, requires necessary additional information for antagonists to have them move correctly. `patrol` has two properties which need to be defined; the `start` grid coordinates and the `end` grid coordinates. The antagonist will then move between these points until they are destroyed or the level is over.

6.3.4 Protagonist

In the domain analysis, antagonists and protagonists are both labeled as entities of the game. While antagonists are defined in the elements section of the DSL, a protagonist is defined in each level created for a platformer. This is because, while the same enemies can be found in multiple levels, protagonists in different levels can vary. In games such as *Celeste*, when the player is in an underwater portion of the level, they have altered properties to account for the change of environment such as the maximum jump height or there is a variation in speed. In *Rogue Legacy*, different protagonists start with a variety of different properties. These protagonists can differ with regards to their weapons, health, speed, and maximum jump height. The DSL took these variations of protagonists into account leading to each level having their own defined protagonist, rather than one general protagonist defined in the elements part.

Figure 6.6 shows a description of a protagonist in a single level. As noted above and in chapter 5, antagonists and protagonists can both be considered entities, thus they have similar properties that need to be defined. With the exception of `spawn`, `jump`, `lives`, the rest of the properties found under a `protag` are the same ones found under an `antag` in the elements side of the DSL. Like the antagonist placement above, the protagonist requires an initial position (`spawn`) for the engine to place the player when the level is loaded. Platforms as terrain in levels and jumping protagonists are the two crucial elements which define the entire genre of platformers. The `jump` property takes a non-negative integer as a value indicating the maximum height in grid units that the player can jump. If platforms are placed just one unit above or below the preceding platform, then this jump value is typically 1, but that is not always the case, going back to *Celeste's* change in maximum jump height in different environments ("levels"). The last unique property for the protagonist is `lives`. While some games may provide only a single life to the player (*Rogue Legacy*, *Risk of Rain*), others can have multiple lives before "game over". To take both types of games into account, `lives` can range from 1 - n.

```
protag
  spawn 0 0
  jump 1
  speed slow
  lives 3
  health 100
  colour red
  shape rectangle
  weaponname "hom"
```

Figure 6.6: A `protag` description in GypPO

6.3.5 Win Condition

Through analyzing the various ways of winning a level in platformers, three win conditions were implemented as options for GypPO. The win conditions boss kill (`boss`) and end of the level (`eol`) are directly taken from section 5.2. The boss win condition requires the name of an antag that is placed in the level to define it as the boss. This results in moving onto the next level once that antagonist is destroyed. The end of the level definition requires an (X,Y) coordinate of the relative coordinate system indicating where the end of the level is, so once the player enters this cell in the grid, they progress onto the next level of the game. The last win condition implemented in GypPO is `score`, which is that the player moves on once a certain score is reached. This is a combination of the two win conditions pickup every collectible and kill all of the enemies from section 5.2. As both collectibles and enemies have score values assigned to them, GypPO uses this property to determine whether those win conditions are met. Figure 6.7 shows how each win condition is defined in GypPO.

```
score 200
boss "antag1"
eol 19 18
```

Figure 6.7: Defining all three types of win conditions for GypPO

Chapter 7

GypPO DSL Design Decisions

The influence of the MAKU DSL by Collman (2014) is very evident on the design of GypPO. Throughout the design process, the initial goal was to make GypPO similar to MAKU in the hopes of combining the two DSLs later on. The resulting DSL would cover both the bullet hell and the platformer domain. Ideally, this DSL would be able to generate games such as *Cuphead*, which is a hybrid of the shoot 'em up and platformer families of games.

The basic organization of a game described in GypPO is to have separate definition sections for the constant game elements and the varying logic (levels) of a game. The purpose of separating these two portions of a platformer is to have them not knowing and affecting any of the specifics regarding the other section. This is so that modifications to the definitions of one portion do not result in any changes to the other. If a level uses an antagonist, the level description only requires the name of the `antag`. This results in any changes in the antagonist's specifications immediately being displayed when the level is generated without needing to modify the level's specifications. Entities in platformers, such as basic enemy units, can be found throughout the different levels of a game. GypPO thus places these sort of game elements' definitions under the elements portion so they are only defined once. By adding a unique name attribute to each game element, they can be linked to any game levels defined under the `logic` section without the need to redefine the element each time. Weapons, enemies, and upgrades all fall under the `elements` section of the DSL as their definitions remain unchanged throughout the GypPO specifications file and the generated game. While the domain analysis was crucial in discovering the required element properties in GypPO, these attributes also needed to be independent of any specific level information so they could truly be used anywhere in the game. The prime example of this is the movement pattern `patrol`. While the pattern type is defined in the elements section of GypPO, the necessary patrol point information is left to each level definition so the same antagonist could be placed at different locations across the game while still having the exact same properties (movement data, health, weapon, shape and size).

As mentioned in chapter 6, while the protagonist is considered an entity in the domain analysis, this is the one element of the game that is not defined under the elements section. Since the elements should be unchanging in their definitions, protagonists were moved to the logic portion as they had the potential to change throughout the game. A protagonist may not vary much between levels, but with there being a possibility of variation, it is easier to define the protagonist in each level including the specific spawn location for that level. The alternative method would be to define a single protagonist in the elements section but then include the ability for users to modify the protagonist entity in the logic portion as well. This would result in providing the logic portion with permission to read and write the properties of an element already defined, defeating the purpose of having these definitions be separate.

The `logic` definitions are for describing levels using the elements already defined in a specification file. As mentioned above, it is crucial that this section only uses the names of the elements to use them, without needing to go into any further details regarding their definitions. It is up to the generator and engine to implement the game elements correctly for each level. The only exception to just needing the name is of course the patrol definition for antagonists which require it. Most of the definitions for a level are easily understood such as element placing, with the requirement of a defined element and coordinates for placement. This is the case for upgrades and enemy placements. The interesting decisions occur with regards to camera models and platform placement.

Camera models were decided to be placed under level descriptions as a change in camera models between levels does occur in platformers. Most commonly in boss levels, even when most of the game is free-scrolling, these levels become fixed. *Metroid* is an example of this as the majority of the game uses a free-scrolling camera model but during boss fights, this becomes fixed.

Platform placement is similar to protagonists as it could be defined under elements as an obstacle or map element and reused throughout the game. The reason for not doing this is that even if the same platform is used to create an entire level, it would still need to be linked in the level definition with a coordinate for placement. This just needlessly adds additional definitions to the DSL with the user first needing to define a platform as an element (using the same properties already defined in the current GypPO logic section). Then the level definitions would still need to call that defined platform and provide coordinates. The current method, Figure 7.1, is much simpler with all platforms in a level being listed with the coordinates, the size, and the colour. There is no need to also have the platforms defined under `elements`.

An important note considering the design of GypPO is that the ability

```
platforms
  0 1 m green
  1 2 l green
  5 1 m green
  4 3 s green
  4 2 m green
```

Figure 7.1: Placing platforms in a level

to add additional content is relatively easy. Although in its current state, GypPO only allows for simple platforms to be placed, there is still room to add the more "complicated" platforms described in chapter 5 in the future. More details regarding how these other platform types could be added can be found in section 9.1.

Chapter 8

GypPO Generator

With an explanation of code generation provided in chapter 4, this chapter provides a brief look into how GypPO's generator follows the steps of section 4.2 to generate a complete game. The process involves first parsing the GypPO DSL specification file, and storing the relevant information into the data structures of the GypPO abstract syntax tree (AST). A translator then converts the data from the designed AST to the target language's (JavaScript) AST. Finally, a pretty printer needs to output the generated code. This chapter uses snippets of the project code to explain each portion of the generator. The entire project code can be found at `https://gitlab.cas.mcmaster.ca/G-ScalE/GypPO`.

The target language of GypPO did not necessarily have to be JavaScript. One alternative could have been Unity scripts generated in C#. JavaScript was the chosen output language as it allows for browser-based Platformers to be generated. Having games which run on browsers allows for them to be run on any machine capable of opening a web page, and on any screen size. Alongside JavaScript, the Crafty engine is used to create the graphics of the generated video game.

8.1 GypPO AST

Figure 8.1 provides a glimpse into the AST created for the GypPO DSL. The figure shows the data structure created to store the necessary information of a health upgrade. It is up to the parser function, Figure 8.2, to read the input specification file for each health upgrade definition (explained in subsection 6.2.3) and store the information as a `HealthUpgrade`. A health upgrade requires a `String` for the name, a natural number (`Nat`) for the amount that the health upgrade increases the property by, and finally a `Colour` to show in the game level. Each section of the DSL in chapter 6 has a corresponding data structure in the designed AST as well as a parser function similar to the one above in order to store the information correctly. The complete GypPO AST

```
data HealthUpgrade = HealthUpgrade {
  getHealthName :: String,
  getHealthIncrease :: Nat,
  getHealthColour :: Colour
} deriving (Show)
```

Figure 8.1: The data structure for a health upgrade

```
healthUpgrade :: SourcePos -> Parser D.HealthUpgrade
healthUpgrade p = try $ do
  _ <- inline p
  reserved "health"
  n <- between quote quote (many alphaNum)
  spaces
  hlth <- digit'
  spaces
  clr <- colour'
  spaces
  return $ D.HealthUpgrade n hlth clr
```

Figure 8.2: Parsing a health upgrade from a specification file

```
data JSIfStatement = JSIfStatement JSExpression
  JSFunctionBody
    (Maybe (Either JSFunctionBody JSStatement))
```

Figure 8.3: The JavaScript AST representation of an if statement

is found in `Design.hs` under the AST folder, while the three DSL parsers are found in `Game.hs`, `Elements.hs`, and `Logic.hs` under the Parser folder of the project repository. It is important to note, these files are based on the MAKU AST and parsers by Collman (2014). The data types `NEList` (non-empty list) and `Nat` along with the functions related to them; the function `toNat` and the entirety of the file `Helper.hs` were taken directly from Collman (2014).

8.2 JavaScript AST

The JavaScript AST is an abstract representation of the JavaScript language, similar to how the GypPO AST represents the GypPO language. This AST file was created by Seefried (2012) with modifications to suit the needs of the GypPO generator. Figure 8.3 shows a snippet of the AST showing the data structure for a conditional statement. The if statement data structure has three fields of data required. The data structure requires a condition (as a `JSExpression`), and a list of statements to execute when that condition is met (`JSFunctionBody`). The third field has the options of being either `Nothing`, another set of statements for an `else` statement or another `JSIfStatement` to generate an `else if` statement in JavaScript. The complete AST can be found in `JSAST.hs` of the project repository.

8.3 Translation

The translation step is crucial to successfully generating code. By first parsing the platformer information and storing it into the data types of the GypPO AST, this step translates the designed AST into the AST of any target language. As the target AST can be swapped for any other, a new translator needs to be written for every target language. Going back to the health upgrades example, figure 8.4 shows how the `HealthUpgrade` data structure from the designed AST is translated into a `JSExpression` (a data structure from the JavaScript AST). This example takes the properties of a health upgrade

```
field :: JSName -> JSExpression -> JSObjectField
field a b = JSObjectField (Left a) b

object :: [JSObjectField] -> JSExpression
object = JSExpressionLiteral . JSLiteralObject . JSObjectLiteral
```

Figure 8.5: JavaScript `field` and `object` definitions in the JavaScript AST

and makes them fields of a health upgrade object in JavaScript. Figure 8.5 shows how the field and object definitions are written in the JavaScript AST. The translation of every GypPO AST data structure to a JavaScript AST data structure is found in the code file `Plat_Pretty.hs` in the GypPO repository.

```
healthUpgrade :: HealthUpgrade -> JSExpression
healthUpgrade h =
  object
  [
    field (nm "name")       (exstr $ getHealthName h),
    field (nm "health")     (n $ unNat $ getHealthIncrease h),
    field (nm "colour")     (exstr $ unClr $ getHealthColour h)
  ]
```

Figure 8.4: Translating a health upgrade from the `HealthUpgrade` data structure to a JavaScript expression

8.4 Printing

The pretty printing step of the generation process is where the target language's AST data is printed in the language's correct syntax. Figure 8.3 shows the data structure in the JavaScript AST, while Figure 8.6 depicts the pretty printing of an if statement as JavaScript code. The method first prints the necessary keyword `if`, followed by the condition and the statements to execute when the condition holds true. The next part is where the three options described in section 8.2 are used. Depending on if the last field of the `JSIfStatement` is `Nothing`, an else statement, or another if statement, the pretty printer can print the correct code syntax. The entire pretty printing code was also based on Seefried (2012) and is found in the `JSAST.hs` file of the GypPO repository.

The health upgrade example follows the entire process of code generation. First parsing and processing the upgrade into the GypPO AST, then translating to the JavaScript AST. The pretty printer will then convert the JavaScript object into the source code, shown in figure 8.7.

```
instance Pretty JSIfStatement where
 pretty (JSIfStatement cond thenStmts elseOrIf) =
   text "if" <+> parens (pretty cond) <+> pretty thenStmts <+> eOI
   where
     eOI = case elseOrIf of
     Nothing              -> empty
     Just (Left elseStmts) -> text "else" <+> pretty elseStmts
     Just (Right ifStmt)   -> pretty ifStmt
```

Figure 8.6: Pretty printing an if statement

```
healthUpgrades: [ {
   name: 'hlth'
   ,health: 50.0
   ,colour: '#ff0000'
} ]
```

Figure 8.7: Health upgrades defined in generated JavaScript

8.5 Generated Game

An example GypPO specifications file can be found in Appendix C. The antagonists and weapons defined in this sample file are based off existing game elements found in *Rogue Legacy*. Figure 8.8 displays a screenshot of this generated level with all entities being displayed as shapes (the protagonist being red). An important observation is that Figure 5.2g looks alot more polished and playable, stressing the importance of the art assets in a video game.

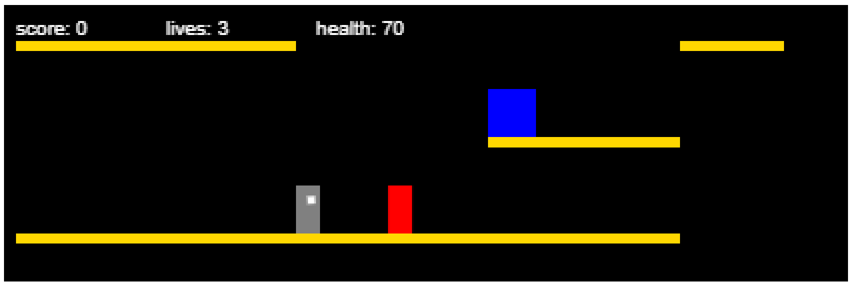


Figure 8.8: Screenshot of generated game

Chapter 9

Conclusion

This research project was done to determine if the platformer family of games could be generated. That is, to see whether or not the games could be generalized enough that a designed domain-specific language could define all games falling under this genre. GypPO, in its current state, does the job of defining and creating platforms well enough. There are some features which could be improved or added (discussed below in section 9.1), but the GypPO system proves that these games do, in fact, have many common elements and can be generated.

9.1 Future Work

As the current form of GypPO is entirely text-based, it makes it difficult to accurately design a level, forcing the DSL users to find other methods of visualizing the grid system and spawn locations of every entity in a level. The current method is functional however, for future work, the largest feature to add to GypPO would be a map editor. A map editor would open the door for accurate visual placement of game objects, new platform types, and movement information. Essentially this would simplify and graphically display the logic portion of the DSL. Users could add patrol points, platforms, and spawn locations accurately and easily. This could either be an entirely new map editor designed for GypPO or allow GypPO to support existing map editors.

One feature that the current state of GypPO is lacking is a variety of platform options. Besides the stationary platforms already implemented, the more complicated platforms found in Platformers would be difficult to define using only text. They require additional information, such as maximum jump increase a spring platform could give, or the coordinates and speed a moving platform needs. These platforms are left from the current version until an easier method of creating a level is implemented. The addition of a map editor would allow the functionality and properties of elements to still be defined with the GypPO language while allowing users to essentially drag and drop these elements into the levels when designing them.

An alternative way of improving level design could be having GypPO procedurally generate levels. Ways to generate levels in an automated way exist, such as the rhythm-based generator developed by Smith *et al.* (2009).

Another future update would be to add in more upgrades to the GypPO DSL. While the most common upgrades were implemented into GypPO, the newer upgrades found in modern games could be added. This includes some form of armor that negates some damage from enemies, as seen in *Rogue Legacy*. GypPO could also allow for economies and currency to be implemented adding costs to upgrades, in addition to the current “pickup” upgrades.

The modern platformers have also added new movement mechanics, such as the air-dash found in *Celeste* and *Rogue Legacy*. The ability to provide users with more control over the movement details of entities in platformers would be implemented into GypPO in the future.

Appendix A

Games List

Game	**Developer and Year**
Space Panic	Universal Entertainment Corporation (1980)
Donkey Kong	Nintendo R&D1 (1981)
Jump Bug	Hoei and Alpha Denshi Corporation (1981)
Super Mario Bros.	Nintendo Creative Department (1985)
Metroid	Nintendo R&D1 and Intelligent Systems (1986)
Super Mario Bros. 2	Nintendo R&D4 (1988)
Sonic the Hedgehog	Sonic Team (1991)
Donkey Kong Country	Rare (2000)
Rogue Legacy	Cellar Door Games (2013)
Risk of Rain	Hopoo Games (2013)
BLACKHOLE	FiolaSoft Studio (2015)
SpeedRunners	DoubleDutch Games (2017)
Cuphead	StudioMDHR (2017)
Celeste	Matt Makes Games (2018)

Appendix B

Summary of Variabilities

$\mathbb{N}_0$ - Natural number starting from 0 (0,1,2,3...)
$\mathbb{N}^+$ - Natural positive number (1,2,3...)

Variability	Platformers can have different types of platforms
Parameter of Variation	Options: Stationary | Moving | Temporary | Spring

Variability	Weapons may be optional for entities
Parameter of Variation	Entities can have a weapon linked to them during specification. Options: Nothing | Unique ID of weapon : String

Variability	A timer may be used as a loss condition
Parameter of Variation	A time limit could be specified when defining a level. Options: Nothing | Specified amount of time (seconds) : $\mathbb{N}^+$

Variability	Score is an optional element found in Platformers
Parameter of Variation	Scoring could occur through killing enemies, collecting items, winning a timed level with time remaining. Options: Nothing | Antagonists have a score value : $\mathbb{N}_0$ | Collectibles have a score value : $\mathbb{N}_0$ | Time remaining is translated into score value. Example of time scoring: *Space Panic* sets a base score value, and the percentage of time left from how much was given is the percentage of the score value the game awards.

Variability	Platformer levels can have upgrades or collectibles
Parameter of Variation	Upgrades can be defined and placed in levels. Options: Nothing | Health upgrade | Life upgrade | Weapon upgrade | Collectible

Variability	Platformers can implement additional resources for use in game
Parameter of Variation	Additional resources are used as currency for the use of common elements found in Platformers. For instance, in Rogue Legacy, mana is a consumable resource to use special weapons and gold coins are used as currency to purchase upgrades. The cost and resource could be specified when defining weapons and upgrades. Cost value : $\mathbb{N}_0$, with 0 indicating no additional resource required.

Variability	Entities can have additional attributes assigned to them
Parameter of Variation	When defining the entities, optional attributes could also be defined such as armour which requires a natural number to indicate how much damage is absorbed before affecting health.

Variability	Players can have a double jump or a dash jump
Parameter of Variation	While not common in all Platformers, the specification for the players could have these additional movement mechanics be enabled : Boolean.

Variability	Players can have multiple weapons at once.
Parameter of Variation	When linking weapons, multiple weapons could be linked as a list

Variability	Enemies can drop loot when killed
Parameter of Variation	Loot is essentially an upgrade. A loot item could be linked to the antagonist so it appears once the antagonist is destroyed. Options: Nothing | Unique ID of upgrade

Variability	Levels can have checkpoint spawn locations when the player dies
Parameter of Variation	Checkpoint coordinates could be included when defining a level. Options: Nothing | Coordinates for checkpoint position

Appendix C

Example GypPO File

```
grid 20 20
elements
  weapon
    name "str"
    damage 20
    colour green
    shape square
    ranged
      speed medium
      bulletpattern straight
    atkrange 100
    rof 1
    target characters
  weapon
    name "arc"
    damage 20
    colour white
    shape square
    ranged
      speed slow
      bulletpattern arc
    atkrange 100
    rof 1
    target all
  antag
    name "facingTurret"
    health 18
    weaponname "str"
    flying no
    movement
      pattern still
      track yes
      vee no
      speed rest
    colour blue
    size m
    shape square
    score 100
  antag
    name "archer"
    health 20
    weaponname "arc"
    flying no
    movement
      pattern patrol
      track no
      vee no
      speed vslow
    colour gray
    size m
    shape rectangle
    score 100
  upgrades
    health "hlth" 50 red
logic
  level
```

```
camera free-scroll 4 4
platforms
  0 15 l gold
  3 15 m gold
  4 16 l gold
  7 16 l gold
  9 15 s gold
  10 14 m gold
  8 13 m gold
  6 14 m gold
antags
  "archer" 5 16
  patrol
    start 4 16
    end 8 16
  "archer" 10 14
  patrol
    start 10 14
    end 12 14
  "facingTurret" 9 15
lvlUpgrades
  "hlth" 6 14
protag
  spawn 0 15
  jump 1
  speed slow
  lives 3
  health 100
  colour red
  shape rectangle
  weaponname "str"
score 300
```

Bibliography

Adams, E. (2014). *Fundamentals of Game Design*. New Riders, Third edition.

Ardis, M. A. and Weiss, D. M. (1997). Defining families: The commonality analysis (tutorial). In *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, pages 649–650. ACM.

Cellar Door Games (2013). *Rogue Legacy*. Game [PC]. Cellar Door Games, Toronto, Canada.

Collman, N. (2014). *MAKU: A Code Generator for Bullet Hell Games*. Master's thesis, McMaster University. `http://hdl.handle.net/11375/16048`.

Coplien, J., Hoffman, D., and Weiss, D. (1998). Commonality and variability in software engineering. *IEEE software*, **15**(6), 37–45.

Czarnecki, K. and Ulrich, E. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA, USA, First edition.

DoubleDutch Games (2017). *SpeedRunners*. Game [Xbox One]. tinyBuild, Bothell, United States.

FiolaSoft Studio (2015). *BLACKHOLE*. Game [PC]. FiolaSoft Studio, Prague, Czech Republic.

Fowler, M. (2010). *Domain-specific languages*. Pearson Education, First edition.

Frakes, W., Prieto, R., Fox, C., *et al.* (1998). Dare: Domain analysis and reuse environment. *Annals of software engineering*, **5**(1), 125–141.

Hoei and Alpha Denshi Corporation (1981). *Jump Bug*. Game [Arcade]. Alpha Denshi, Ageo, Japan. ROM uploaded September 13, 2014. `https://archive.org/details/arcade_jumpbug`.

Hopoo Games (2013). *Risk of Rain*. Game [PC]. Chucklefish, London, England.

Matt Makes Games (2018). *Celeste.* Game [PC]. Matt Makes Games, Vancouver, Canada.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, **37**(4), 316–344.

Mur, R. A. (2006). Automatic inductive programming. In *Proceedings of the 23rd international conference on machine learning, tutorial.*

Nintendo Creative Department (1985). *Super Mario Bros.* Game [NES]. Nintendo, Kyoto, Japan.

Nintendo R&D1 (1981). *Donkey Kong.* Game [Arcade]. Nintendo, Kyoto, Japan.

Nintendo R&D1 and Intelligent Systems (1986). *Metroid.* Game [NES]. Nintendo, Kyoto, Japan.

Nintendo R&D4 (1988). *Super Mario Bros. 2.* Game [NES]. Nintendo, Kyoto, Japan.

Rare (2000). *Donkey Kong Country.* Game [GBC]. Nintendo, Kyoto, Japan.

Seefried, S. (2012). js-good-parts. `https://github.com/sseefried/js-good-parts`.

Smith, G., Treanor, M., Whitehead, J., and Mateas, M. (2009). Rhythm-based level generation for 2d platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 175–182. ACM.

Smith, S. and Chen, C.-H. (2004). Commonality analysis for mesh generating systems. Technical Report CAS-04-10-SS, McMaster University, Department of Computing and Software.

Sonic Team (1991). *Sonic the Hedgehog.* Game [Genesis]. Sega, Tokyo, Japan. ROM uploaded May 6, 2014. `https://archive.org/details/sg_Sonic_the_Hedgehog_Rev_1_1991_Sega_JP-KR_en`.

StudioMDHR (2017). *Cuphead.* Game [PC]. StudioMDHR, Oakville, Canada.

Szymczak, D. (2014). *Generating Learning Algorithms: Hidden Markov Models as a Case Study.* Master's thesis, McMaster University. `http://hdl.handle.net/11375/14101`.

Universal Entertainment Corporation (1980). *Space Panic.* Game [Arcade]. Universal, Tokyo, Japan. ROM uploaded August 7, 2014. `https://archive.org/details/arcade_panic`.