

MAKU: A Code Generator for Bullet Hell Games

MAKU: A CODE GENERATOR FOR BULLET HELL GAMES

BY

NATHAN COLLMAN, B.Sc.

a thesis

submitted to the department of Computing and Software

and the School of Graduate Studies

of McMaster University

in partial fulfilment of the requirements

for the degree of

Master of Applied Science

© Copyright by Nathan Collman, May 2014

All Rights Reserved

Master of Applied Science (2014)
(Software Engineering)

McMaster University
Hamilton, Ontario, Canada

TITLE: MAKU: A Code Generator for Bullet Hell Games

AUTHOR: Nathan Collman
B.Sc., (Computer Science)
The University of the West Indies (Mona Campus),
Kingston, Jamaica W.I.

SUPERVISOR: Dr. Jacques Carette

NUMBER OF PAGES: xii, 81

In loving memory of the Reverend Dr. Sandra Morgan.

She was an excellent person.

Abstract

In each genre of video-game, there are always commonalities that bind different titles to each other. In classifying these similarities, a game can be thought of its base genre-specific features and its further elaborations, to this set of commonalities. Specifying a game in this way allows the developer to focus on these elaborations, while ensuring conformity to preexisting genres and player biases.

This thesis describes MAKU as a fully customizable system for generating HTML5 Canvas browser based games belonging to the Bullet Hell genre of video game. It consists of a domain specific language that encapsulates the core features and this functionality of this genre of video game along with an intelligent code generator, that interprets MAKU language game specifications and generates specialized source code.

MAKU is equally accessible to domain-novices and domain-experts, and allows for the generation of games with variable sophistication. It shows that genres are indeed specifiable and well suited to code-generation.

Acknowledgments

I would like to thank my supervisor, Dr. Jacques Carette, for his patience, enthusiasm, advice and dedication to the completion of this project. I would also like to pay special tribute to the other members of my examination committee, Dr. Spencer Smith and Dr. Rong Zheng for their invaluable feedback on this thesis.

I wouldn't be here without the support of my family and my fellow graduate students and friends. I must thank them for their patience in fielding my stupid questions and thank them for their dedication to seeing me over the line.

光陰矢のごとし, 災難に友達が知られる。

Notation and Abbreviations

- AST = Abstract Syntax Tree
- DSL = Domain Specific Language
- HTML5 = The 5th version of the HTML standard.
- JS = JavaScript
- BGM = BackGround Music

Contents

Abstract	iv
Acknowledgments	v
Notation and Abbreviations	vi
1 Introduction	1
2 A Word on Genre	5
2.1 Genre	5
2.2 The Implementor’s perspective	6
2.3 The Player’s perspective	7
2.4 Conclusions	8
3 Bullet Hell	10
3.1 Shoot ’em up	10
3.2 Drilling down on Bullet Hell	14
3.3 Bullet Hell as a Program Family	16
3.4 Summing it all up.	18

4	Domain Specific Languages	21
4.1	What are DSLs?	21
4.2	What makes a “good” DSL?	23
4.3	DSLs: Embedded vs. External	23
4.4	Relevance to this work	24
5	Code Generation	25
5.1	What is code generation?	25
5.2	The stages of generation	26
5.3	Why use code generation?	26
6	Diving into MAKU: A <i>Hodgepodge</i> of an Example	27
6.1	A high level description of <i>Hodgepodge</i>	28
6.2	The elements in <i>Hodgepodge</i>	29
6.3	The logic in <i>Hodgepodge</i>	33
7	The MAKU DSL	36
7.1	A word on abstraction	36
7.2	A MAKU game	38
7.2.1	The elements of a game	39
7.2.2	With respect to movement	40
7.2.3	With respect to <i>TurretType</i>	42
7.2.4	The logic of a game	44
7.2.5	The MAKU protag	45
7.2.6	The MAKU timeline	46
7.2.7	<i>WinConditions</i> in the MAKU language	46

7.3	Extensibility	47
8	The MAKU Generator	48
8.1	The ASTs	48
8.1.1	The Design AST (DAST)	49
8.1.2	The JavaScript AST (JSAST)	49
8.2	The translator	50
8.2.1	Part One: <code>init()</code>	51
8.2.2	Part Two: The engine assembly	51
8.2.3	The <i>pretty</i> printer	52
9	The MAKU Engine	54
9.1	Static vs. Dynamic	55
9.2	Optimizations	56
10	Solution Decisions	63
11	Testing	65
11.1	Testing for description errors	65
11.2	Testing the MAKU engine	66
12	Conclusion	68
12.1	Related Work	69
12.2	Limitations of the MAKU language	69
12.3	Future Work	70
A	Example MAKU Files	72

A.1	Example.maku	72
A.2	hodgepodge.maku	74

List of Figures

3.1	The final boss in <i>Mushihimesama</i> . A great example of the difficulty inherent in Bullet Hell games. The protagonist is currently center-bottom of the screen and every pink dot is an enemy bullet.	11
3.2	Screenshots of SHMUPs organized by viewpoint as described in Table 3.1	12
3.3	<i>Gradius</i> ' power-up bar. Shows the different types of power-ups that were available to the player.	13
3.4	Screenshots of Bullet Hell games	16
3.5	Terms derived from the Bullet Hell genre domain analysis.	17
6.1	Hodgepodge: bullets	29
6.2	Hodgepodge: antag <i>b1</i>	30
6.3	Hodgepodge: antag <i>b1, two</i> and <i>three</i>	31
6.4	Hodgepodge: upgrades	32
6.5	Hodgepodge: randoms	32
6.6	Hodgepodge: singles	33
6.7	Hodgepodge: level one	34
6.8	Hodgepodge: level two	35
7.1	A diagrammatic representation of how a MAKU game is generated. . .	37
7.2	MAKU game in pseudo-BNF	38

7.3	A MAKU elements description	39
7.4	A MAKU bullet description	39
7.5	A MAKU <i>SSCW</i> description	39
7.6	A MAKU antag description	40
7.7	A MAKU movement description	40
7.8	The MAKU turret descriptions	41
7.9	MAKU turret configurations	42
7.10	A MAKU upgrades description	43
7.11	The MAKU logic description	44
7.12	The MAKU grid description	44
7.13	The MAKU level description	45
8.1	The game data type as defined in Haskell	49
8.2	The logic data type as defined in Haskell	50
8.3	The movement data type as defined in Haskell	50
8.4	Translation function for a MAKU shot	52
8.5	Pretty printer function for a MAKU shot	53
9.1	A diagrammatic representation of the translation process.	55
9.2	The model for MAKU bullets in Javascript	56
9.3	The controller for MAKU bullets in Javascript	57
9.4	<i>PanHorizontal</i> movement pattern as defined in JSAST	57
9.5	<i>PanHorizontal</i> movement pattern as defined in Javascript	59
9.6	Random.prototype.assignPosition: all sides defined	61
9.7	Random.prototype.assignPosition: only left defined	62

Chapter 1

Introduction

At its core, this thesis describes a solution to a modeling problem. There are many varieties of video game released each year and in quantifying these games through play, there are many common threads that bind each one to the next. These threads are normally collected and isolated as so-called *genres* of game.

Through analysis of seemingly “similar” games it should be possible to isolate and categorize commonalities and through this categorization organize them by the crossover between their commonalities. Indeed, if all games of a particular genre share common traits, there is a finite set of games where each member is differentiated from the other by the various permutations and combinations of choices relative to that genre sphere. The bold statement could then be made that within a particular genre, there is no “original” game. We posit that if there was a language of description for these commonalities, then video games are wholly specifiable. Also, if they are specifiable, then they are capable of being generated as well.

Apperley [2006] suggested that by focusing on genres of interactivity, it allows the scholar to examine games in a way that can classify them according to their underlying

similarities rather than their superficial visual or narrative differences.

The goal of this thesis is to develop a domain specific language (DSL) and code generator for games of a particular genre of video game, the Bullet Hell genre, to prove this point. The domain specific language put forward in this paper is believed to be a complete representation of the core game elements and logic that make up the set of Bullet Hell games. The DSL allows users to spend the majority of their development process focusing on which elements make up their game as opposed to being confounded by implementation details. MAKU is built in an extensible fashion to allow those who are familiar with the underlying technologies to extend it. DSLs solve specific problems, and genre, as will be discussed later, is not standardized. This means, that due to its variability, we must define our own category of what makes a Bullet Hell game for any progress to be made.

The Gaming Scalability Environment Project (G-Scale) lab at McMaster University is particularly concerned with technical and experiential issues surrounding the scaling of digital games and virtual environments across a wide spectrum of display devices. The G-Scale lab theorized attention overload to be a non-trivial factor in scaling games. How does one's perception of the same game change with respect to scaling those on screen elements of the game linearly? Differences in speed and difficulty normally become more apparent as a result of this "scaling". Bullet Hell was chosen as the target genre because it perfectly typified this problem. In a previous experiment, we developed three Bullet Hell game prototypes and through that process and subsequently, the domain knowledge garnered from it, it was discerned that this domain was one in which a generative programming solution could be applied.

Organization of the thesis

Chapter 2 will introduce the reader to the idea of video game genre from two perspectives: that of the player and that of the implementor.

Chapter 3 introduces the reader to Bullet Hell games. This is done via a thorough domain analysis of both Bullet Hell and their super genre of shoot 'em up games (SHMUPs).

Chapters 4 and 5 will introduce domain specific languages and code generators, respectively. This will serve as an introduction to both topics, the different techniques inherent to each and discuss why the juxtaposition of the two was deemed appropriate in the solving of this particular problem.

In Chapter 6 introduces *Hodgepodge* as an example of MAKU in practice. We will move from a high level specification of the game *Hodgepodge* to a MAKU description and then towards the final generated source along with screenshots of *Hodgepodge* running in the browser.

Chapter 7 delves into the inner workings of the MAKU language. Whereas Chapter 6 dealt specifically with the choices relative to *Hodgepodge*, this chapter shows the design of the MAKU language using a psuedo-BNF notation and outlines all the domain specific terms and structure inherent in MAKU.

Chapter 8 will discuss the ASTs used to represent the MAKU language internally, along with a brief explanation of the rudiments behind the generation process.

Chapter 9 will discuss the specification of the MAKU engine. Taking into consideration the lessons learned from the domain analysis and describe their representation in the engine.

Chapter 10 will discuss the various implementation details explicitly related to the

languages used in designing MAKU. Questions related to why a functional paradigm was used in addition to other implementation concerns related to why a browser based solution was chosen.

Chapter 11 discusses the different testing methodologies employed whilst testing MAKU and Chapter 12 concludes the paper by discussing limitations of the current system and potential future work.

Chapter 2

A Word on Genre

Genres ‘evolve’ akin to biological species is second nature to the discourse of most knowledgeable gamers, gaming press people, industry veterans and game studies academics. Arsenault [2009].

2.1 Genre

A genre is a term used to describe something based on some stylistic criteria generally denoted by conventions as agreed upon by society over time. The general problem that is encountered when assessing genre is that there is no regulation as to what each particular genre *really* encapsulates. The term in and of itself is not specifically related to the domain of video games or gaming and so when discussing *video-game* genre or one its subsets we are categorizing a set of criteria directly related to the type of video game we are seeking to discuss.

Bolter and Grusin [2000] describes the “logic of remediation”. Remediation being “the formal logic by which new media refashion prior media form”. The space of video

game genre has been defined by the genres of other associated media, such as film and television. However, the inherent difference is the interactivity of video games. Bolter and Grusin [2000] suggest that interactivity is secondary to its more representational aspects. *Representational* denoting the visual aesthetic elements of a particular game. Following on from this, Frasca [2003] suggests two approaches to the study of video games, the *narratological*, which focuses on the narrative, more representational aspects of a game and *ludological*, which is more focused on understanding of their structure and elements. Arsenault [2009] suggests a conflation between both saying that they are both intrinsically linked to gameplay.

Different genres are defined differently in different places, and often times one man's genre is another man's sub-genre. Is it *Action*? Or *2D Action*? Do you organize by perspectives and viewpoints? (e.g. *1st/3rd Person Shooter*). Indeed, it is common that most video-game review sites leave the selection of genre up to the developer. This is an interesting concept, because it means that the genres that currently exist have been informed by what the developers thought their game to be. However, different review sites can categorize the same game into a plethora of different genres. Traditionally, games are grouped into their different genres by either commonalities between their mechanics or commonalities between their themes. As an implementor, you become more aware of the former, whereas, as a player you become more aware of the latter.

2.2 The Implementor's perspective

The implementor's view on genre is based on what the difference between genres connote in terms of implementation details. The implementor is much more token to the ludological line of thinking. Adams [2009] states "*video game genres are determined by gameplay:*

what challenges face the player and what actions he takes to overcome those challenges". As an implementor, it becomes clear that certain genres *are* their mechanics.

For example, you expect *racing* games to have some notion of a "track" and an avatar that moves around it. Indeed without this mechanic you would struggle to call that game a *racing* game. Other game mechanics that are liable to be found in racing games include "laps", position ranking and steering at the very least. From an implementation point of view, the game is made up of its differing game mechanics working in harmony. Conceptually, one of the major processes involved in building a game is really the incorporation of mechanics common to its genre.

Daniel Cook straightforwardly points out the significance of mechanics and interactivity in "My Name is Daniel and I am a Genre Addict": *"In the game industry a genre is a common set of game mechanics and interface standards that a group of titles share. [...] Warcraft and Starcraft have very different plots and settings, but they still belong to the same genre of RTS [Real-Time Strategy]"*. Cook [2013]

2.3 The Player's perspective

The last citation is instructive, because it shows us not just the implementor's perspective but the player's as well. The player while possibly being cognizant of some of the game mechanics (especially the ones announced proudly on the box!) is more likely to recognize more narratological/thematic elements. Both Halo: Combat Evolved (Bungie Inc. [2001]) and Battlefield 1942 (EA Digital Illusions CE. [2002]) are described as first-person shooters. However, whereas as implementor might think of these games in terms of their mechanics, there is a chasm between the thematic elements of both these games. The player would indeed pick up the fact that Battlefield 1942 (EA Digital Illusions CE.

[2002]) is a World War II simulation game where as Halo: Combat Evolved (Bungie Inc. [2001]) is based in a fantasy world where planets and aliens exist. Two games with the same mechanics can be considered different strictly based on their thematic elements. This is an important point to realize because it means that a particular set of mechanics and rules of interaction can be consistent across different themes and produce different final experiences for intended players.

2.4 Conclusions

Arsenault [2009] proposed the *Great Genre Illusion*, showing that these inconsistencies in definition were known:

“In brief, the idea is that the word genre is an umbrella word, and that the bundling of disparate concepts under a single name gives them a false impression of unity. I claim that the word has no more internal coherence than the word “thingy”. To paraphrase Wittgenstein on games, consider, for instance, what is a thingy; what is common to all of them; what do all thingies share? It is natural to expect literary genres, speech genres and film genres to share certain characteristics - they are all genres, after all - but the reality might be far less logical and satisfying”.

It is important to note that the relationship between video game and genre is not injective. The very idea of using an external definition of genre and all its imprecision was insufficient for our purposes as we were seeking to very clearly define what were the main components of the Bullet Hell genre. We employ a classification method based on the implementor’s perspective and organize Bullet Hell games by their common game

mechanics and challenges.

Chapter 3

Bullet Hell

This chapter contains a short discussion on shoot ‘em up games and a domain analysis of Bullet Hell games. It builds upon work gathered by Whitehead [2007].

3.1 Shoot ‘em up

Traditionally, *Bullet Hell* games actually belong to the “shoot ‘em up” (SHMUP) genre of gaming. SHMUPs are actually a sub-genre of the “shooter” genre of video games. The “shooter” genre is itself a sub-genre of the “action” genre of video game. Bullet Hell games are differentiated from traditional SHMUPS by the sheer number of bullets on screen the player has to accommodate for. Figure 3.1 shows the final level of the game *Mushihimesama*. While this game is a SHMUP, the number of bullets on screen further classify it as belonging to the Bullet Hell genre.

Action games in particular are often intensively performative, [...] action games that will often require the player to engage in extreme nontrivial actions Atkins [2003]

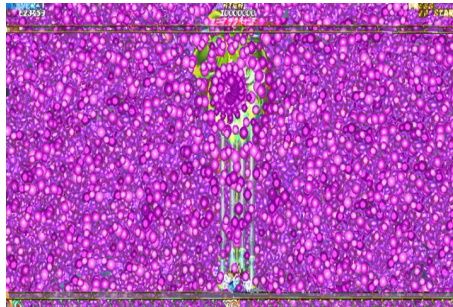
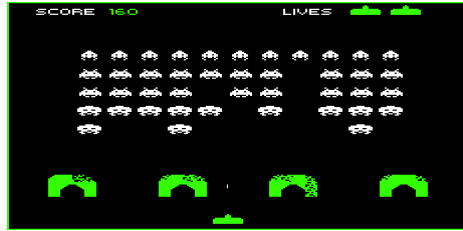


Figure 3.1: The final boss in *Mushihimesama*. A great example of the difficulty inherent in Bullet Hell games. The protagonist is currently center-bottom of the screen and every pink dot is an enemy bullet.

SHMUPs typically limit the control the player has over their movement. In 2D, a common tactic is to give the impression that the player is always moving in a particular direction. They are often characterized by viewpoint as shown in Figure 3.1. Broken down into **fixed** shooters on fixed screens, **scrolling** shooters that mainly scroll in a single direction, **top-down** shooters where the viewport into the game is from above, **rail** shooters where movement is automatically guided down a fixed path and **isometric** shooters which use an isometric perspective.

Typically, the player's avatar engages enemies and must simultaneously destroy and avoid. The games generally call for good reaction time and oftentimes for the player to memorize levels and attack patterns. Many video games normally test player speed and reaction time, but this is especially true of SHMUPs and even more specifically of Bullet Hell games.

Table 3.2 has screen-shots of a variety of SHMUP games from which a few observations can be made. Irrespective of the thematic elements in each of the games, they all have *protagonists* and *antagonists*. These protagonists oftentimes take the form of a space ship but this is not always the case, *Space Harrier* (3.2h), for example, features a human protagonist.



(a) *Space Invaders*



(b) *Galaxian*



(c) *Xevious*



(d) *Darius*



(e) *Time Pilot*



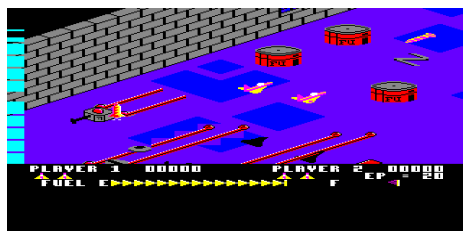
(f) *Bosconian*



(g) *Buck Rodgers: Planet of Zoom*



(h) *Space Harrier*



(i) *Zaxxon*



(j) *Viewpoint*

Figure 3.2: Screenshots of SHMUPs organized by viewpoint as described in Table 3.1

Viewpoint	Game(s)
Fixed	<i>Space Invaders</i> (1978), <i>Galaxian</i> (1979)
Scrolling	<i>Xevious</i> (1982), <i>Darius</i> (1986)
Top-down	<i>Time Pilot</i> (1982), <i>Bosconian</i> (1981)
Rail	<i>Buck Rodgers: Planet of Zoom</i> (1982), <i>Space Harrier</i> (1985)
Isometric	<i>Zaxxon</i> (1982), <i>Viewpoint</i> (1992)

Table 3.1: SHMUP games organized by viewpoint



Figure 3.3: *Gradius*' power-up bar. Shows the different types of power-ups that were available to the player.

Space Invaders 3.2a also has stationary, non-firing antagonists (obstacles). To be fair, they do serve the purpose of being barriers more than anything else, but they are considered to be antagonists all the same.

The heads-up displays (HUDs) of each games, contains at the very least, level information and the current protagonist's score and life count.

The antagonists in each of these games have many different *movement patterns* along with different bullets and shot patterns. They are sometimes grouped together or singular having no relationship with other antagonists currently on screen.

Conceptually, one could think of *Space Invaders* (3.2a) and *Galaxian* (3.2b) as the same game, with differences in their colour schemes and some variance in the movement patterns of their antagonists. *Gradius* (1985) introduced the idea of *power-ups* to weaponry and it became a feature integral to the realm of the SHMUP. Its original power-up bar is shown in Figure 3.3. The ability to not only change bullet types, but shot patterns and gain extra turrets (mechanism that provides the player with the ability to fire bullets from alternate sources) added complexity to the games and gave freedom

to player to choose *how* the games were played. SHMUPs do not traditionally have realistic physics. Protagonist movement is inertia free, projectiles often move at constant speed and bullets are often unlimited.

3.2 Drilling down on Bullet Hell

“Bullet Hell” is actually a transliteration of the Japanese 弹幕 (*danmaku*, “bullet curtain”). It is often know by other names such as “manic shooters”, “maniac shooters” etc. They normally require the player to maneuver a *seemingly* overwhelming number of enemy bullets (projectiles). The screen itself is normally covered with enemies. Figure 3.4 shows four examples of different Bullet Hell games.

With the advent of commercial 3D games, 2D game developers had to seek new ways to impress their players and so conceptualized Bullet Hell games. The goal was to provide the familiar experience of traditional SHMUPS boosted with a new impressive challenge. The challenge being that there were now *curtains* of enemies on the screen. Bullet Hell games are difficult and are supposed to be so.

Bullet Hell games may coax the user into ragged movements and in general seek to overwhelm the player into making rash decisions, but oftentimes the best strategy for survival is to recognize the complex “patterns” inherent in the movement of the bullets in the game. There should be a predictability of sorts *built* into the game that adept player should be able to recognize to further their mastery of it.

Bullets are not necessarily aimed at the player, and due to the plethora of bullets and their complex patterns, they can weave truly beautiful displays on the screen. Bullet Hell games can be more more “manic” (reward continuous nimble movement) or more

“traditional” (reward calm and precise movement). Bullet Hell is characterized the complexity of these patterns, the vividness of the color schemes and the range of different weapon upgrades available to the protagonist.

DonPachi (1997) gave the player incredibly powerful weapons and a large number of power-ups, enemies fired large numbers of projectiles and introduces a chaining scoring system. *Radiant Silvergun* (1997) made all weapons available at the beginning, introducing the dynamic of the player choosing the best weapon for a particular task. Also allowed designers more control over the user experience. *Rez* (2001) synchronized music and vibration and gave feedback when enemies were destroyed by changing the background music of the game).

Titles like *DonPachi*, *Espagluda* (2003), allow the player to select each ship based on their different characteristics. Other iterations, such as *Deathsmiles* (2007) control for different interaction between protagonist, antagonist and antagonist bullets.

Dangun Feveron (1998) allows for a high degree of customization of the protagonist, allowing the player to select speed, and a variety of rapid-fire and sustained fire weapon. *Gundemonium* (2003), *Mushihimesama* (2004) the aforementioned *Deathsmiles* are interesting because they depart from the “ship in space” convention and have humanoid protagonists.

These games always provide a mechanism through which the player can recover life, either through accumulation of points or some other means (destruction of antagonists) along with different *shield* upgrades. Some also present the player with *bombs* which either clear the screen of all non-boss enemies or clear some predefined area surrounding the player’s avatar.

(a) *Deathsmiles*(b) *Espagluda*(c) *Dangun Feveron*(d) *Gundemonium*

Figure 3.4: Screenshots of Bullet Hell games

3.3 Bullet Hell as a Program Family

Domain analysis can be considered to be *commonality analysis*. That is, the approach of identifying those assumptions that irrespective of various variabilities can be said to be common to all members of a particular, proposed program family. Weiss [1997] put forward three assumptions that classify a set of software as a program family. Inherent in which is a strategy for the production of software that is intended to be highly variable and reusable. They are:

1. The Redevelopment Hypothesis: Most software development is actually software re-development, where one creates a variation of an existing system.
2. The Oracle Hypothesis: It is possible to predict the types of changes likely to be necessary in a system over its lifetime.
3. The Organizational Hypothesis: It is possible to possible to organize both the



Figure 3.5: Terms derived from the Bullet Hell genre domain analysis.

software and the organization that produces and maintains it, in such a way as to take advantage of these predicted changes.

The genre of Bullet Hell games meets each of these hypotheses. Most game development in general satisfies The Redevelopment Hypothesis. The very idea of genre and indeed sub-genre speaks to the broad commonality and innovation on the same idea that it prescribes. Bullet Hell games whilst different from title to title can certainly be considered to be variations of an existing system. Similarly, it is indeed possible to conceptualize the types of changes necessary to a game over its lifetime. The use of a road map that lays out features and deadlines relevant to a particular title satisfies both the Oracle and Organizational Hypotheses. The organization itself can be tailored to maximizing re-usability of their assets in order to facilitate any predicted change, whilst also ensuring that the game itself is developed in as modular a way as possible.

3.4 Summing it all up.

Bullet Hell games are a subset of SHMUP games. For a game to be a Bullet Hell game, it must have particular elements: a tremendous number of bullets to be dealt with, antagonists, upgrades and a protagonist. These elements should have a variety of different movement patterns. They should allow for the grouping of antagonists and can have strict developer defined paths or can be randomized. Figure 3.5 shows some of the main terms we derived from this domain analysis.

The protagonist should “destroy” antagonists to progress through the levels of the game. These levels could have different win conditions including but not limited to the score, the time elapsed or the defeat of a so-called “boss”. The protagonist should also have a mechanism by which the screen can be cleared of *lesser* enemies, a bomb if you will. From that point on, the only consistent factor between games are their differences and the fact that each game while bearing these similarities with respect to their mechanics and challenges, has a veritable plethora of differences with respect to thematic elements, color schemes etc.

We always thought that there should be a separation of concerns in MAKU. We started with entities and levels and progressed to logic and elements. We decided that logic should describe levels, which in turn should describe a timeline whilst also speaking about the different win conditions.

What really fluctuated however, was the makeup of each of those categories. Initially, we grouped the protagonist (protag), antagonists (antags), groups (groups) and any upgrades (upgrades) in elements. However, as we progressed and fleshed the domain out more, we realised that protags, although seemingly similar, should be independently definable across levels in order to allow the designer to have completely unrelated

protags across different levels.

This prompted us to start thinking about which things were *static* and which were *dynamic*. We wanted elements to be declarative. Anything not found there would not be able to be used in the definition of logic.

We did this because it allows the developer to concentrate on what types of things they would like to use in their game as opposed to what is actually represented there. A declarative approach would allow a developer to flesh out different types of antags, upgrades etc. while still being able to make progress in the logic development of their game.

Surely, the protagonist could be declarative as well, but we ended up needing to take the protag defined in elements and augment it based on different criteria which resulted in duplication of the keyword itself and a more complex internal data structure. We decided to bind them together to make the MAKU language easier to specify and at some level conceptually easier to understand.

We also resisted the temptation to include a *boss* attribute, thinking that bosses could more easily be represented in terms of singles and groups. This would also save us from having to represent antags as being solely *boss* and *minion*.

Obstacles were also originally part of our thinking, but it was decided that they were just non-firing (possibly non-moving) antagonists. There were also constructs like *progression* that dealt with how the levels followed each other as well as lower level structures which handled relationships between both protag and antag bullets respectively. Neither of these made the final cut, as it was decided that the outline of the levels in the specification should dictate the progression and that in order to free the designer from having to specify what would be inherent relationships between antag and protag that

relationships should be *built-into* the system itself.

Chapter 4

Domain Specific Languages

This goal of this chapter is to familiarize the reader with the concept of domain specific languages, what they are, what are their uses and most pertinently, the application to the problem of specifying the Bullet Hell genre. Much of the work is gathered from previous work by Mernik et al. [2005], who spoke about the situations in which DSLs are most applicable, and Beyak and Carette [2011], Curutan [2013] and Szymczak [2014] who implemented various DSLs themselves.

4.1 What are DSLs?

Essentially, an understanding of domain specific languages (DSLs) can be garnered from their opposite. Most people are indeed familiar with general purpose languages (GPLs) such as C or Python whose application is not limited to any specific problem domain but have far reaching applications to a *general* number of problems. DSLs, however, are designed with a specific problem domain in mind. Domain-specific languages (DSLs) are also called application-oriented, special purpose, task-specific, problem-oriented,

DSL	Application domain
BNF	Syntax specification
Excel macro language	Spreadsheets
HTML	Hypertext web pages
LATEX	Typesetting
Make	Software building

Table 4.1: Some widely used domain-specific languages

specialized, or application languages. They are designed to model a particular problem domain in a very straightforward and complete way. This makes them very restricted in their application to problems outside their problem domain. It follows then, due to this specificity, that most DSLs are not Turing complete. Table 4.1 shows some examples of widely-used DSLs.

The main task in the design of a DSL is the domain analysis. A DSL must reflect knowledge inherent in a particular domain. It is important that the domain to be modeled is very well understood. Any common terminology used in the domain will most likely be represented in the DSL to allow domain experts to properly use their expertise. These domain specific features allow experts to more efficiently design solutions to problems. The domain analysis can be done either informally, formally or via extraction from source code.

DSLs are implemented through either interpretation or code generation. Interpretation translates line(s) of the DSL source to their result at runtime, whereas code generation presents an intermediate step in this process. The DSL source is transformed to an intermediate structure, which can then undergo more processing before any output is given Fowler [2010].

4.2 What makes a “good” DSL?

Developing a new DSL is not easy. Due to the constraint that the domain to be modeled must be very well-defined. However, if this constraint is met and the developer possess the necessary expertise, then the gains to be had by developing a DSL become very hard to ignore. The best DSLs only contain what is *necessary* and are not over-designed. While principles of orthogonality and economy of form are foundations of language design, they are not necessarily well-applied to DSL design. Fowler [2010] This means that DSLs may be incredibly useful and well-suited to their intended domains without being particularly modular or very concise.

4.3 DSLs: Embedded vs. External

There are two types of DSLs: Embedded (sometimes called “internal”) and external (sometimes called “standalone” or “freestanding”). They differ in both design and implementation and should be chosen with their benefits and drawbacks in mind.

Embedded DSLs are embedded in a GPL. This language is known as the host language and the DSL can be seen as a micro-language of the host language, that specifies new constructs, while taking advantage of the predefined capabilities of its host language. In particular, the compiler is free. This does also mean that any restrictions in the host language are propagated to the embedded DSL and the expressiveness of the language can suffer as a result. This also means that any prospective users of the DSL will need to have a non-trivial amount of familiarity with the host language.

External DSLs are converse to their embedded counterparts, completely independent of any GPL. They therefore, are able to be much more expressive in terms of their syntax

and grammar, which can enable the language to much more natural for prospective users. However, external DSLs require their own compiler and this may not be a feasible option to the developer due to complexity of implementation.

4.4 Relevance to this work

The goal of MAKU is the generation of games. Prospective users can range from experienced game developers to those wanting to quickly create games. Both domain knowledge from existing source code and informal analysis were used in designing MAKU. The DSL is external, which allows the jargon to be human readable and as non-programming-language-like as possible.

Chapter 5

Code Generation

The processes involved in MAKU are based upon well understood techniques for code generation from DSLs and draw from previous work by Czarnecki et al. [2003], Carette and Kiselyov [2005], Hudak [1996] and Glück and Jørgensen [1997]. This chapter follows up on the topic of code generation as mentioned in Chapter 4.

5.1 What is code generation?

Code generation is a method of developing source code automatically via some higher-level language. Code generators generally take as input some high-level specification and convert that specification into source code.

Compilers can be thought of as really being code generators. The Java compiler takes Java which in this case is our high-level language and transforms it into Java byte-code. MAKU on the other hand, takes a specification in the MAKU language and generates to JavaScript as its final target.

5.2 The stages of generation

To generate code from some high-level specification, there are traditionally five steps:

1. Process the framework and parser.
2. Parse the high-level source into some internal representation.
3. Process this internal representation, which may involve the running of different optimization algorithms.
4. Translate to the target language representation.
5. Print the new source in the target language.

5.3 Why use code generation?

Code generation allows for code reuse and the ability to make very sophisticated changes to systems via small alterations of parameters whilst allowing the production of code to different target languages. Even one layer of abstraction will allow developers to concentrate on a particular domain while then using code generation to produce source code to run on any architecture, using any language.

It allows developers to write code in a non-repetitive fashion whilst keeping a high degree of traceability between the specification and the generated source. The use of optimization techniques also contributes to generated code having a high degree of efficiency. It is important to note that, code generation is not suitable for every purpose. The produced code is not normally human readable and due to optimizations very complex, which can result in far longer testing and debugging cycles.

Chapter 6

Diving into MAKU: A *Hodgepodge* of an Example

This chapter presents the game *Hodgepodge* as an example of MAKU in action. We start from a conceptual high-level definition of a game we are going to call *Hodgepodge* and then move into how we would go about specifying it using the MAKU language. We will be constructing the specification for *Hodgepodge* from the bottom up. Chapter 7 will describe the MAKU language formally from the top-down where any facet that is not discussed in this chapter will be formally addressed.

Hodgepodge is a result of the initial Bullet Hell game development undertaken by the G-Scale lab. It is a game with two levels, each of which is representative of two of the Bullet Hell games initially created in the lab.

6.1 A high level description of *Hodgepodge*

High-level in this context, really signifies “natural language”. The goal is the development of a game we will call *Hodgepodge* and its specificities are given in natural language as:

Hodgepodge is a game that consists of two levels. In each level, we wish to be able to navigate an avatar throughout a virtual world. In this virtual world it is also desirable that we encounter enemies. Destroying these enemies increases our score, and we in turn can be destroyed as well. We want to start each level with a particular number of lives.

This is the simplest definition of what really all SHMUP games are. We therefore convert this definition into:

Hodgepodge is a Bullet Hell game. There should be 2 bullets, one weaker with an overall value of 10 and the other stronger with an overall value of 30. There should be two levels. The first should have a succession of random antagonists and once the score gets to 20000, the level should end. In this first level, there should be two upgrades available. Both first/second should change the left/right turrets of the protagonist to a triple shot. The protagonist should start its main turret being a single shot. It should have an overall of 100, have 6 lives and 1 bomb. The second should have a succession of random antagonists and end with two larger stronger antagonists. The protagonist should be the same as the first level, except this time having no bombs.

Now that we have that defined let us try to coax it into a more MAKU-like fashion, but what does that mean? We define a game to be its elements and logic all relative

```
bullet
  name "tri"
  size s
  shape tri
  color pink
  weight 10
bullet
  name "circ"
  size s
  shape circle
  color pink
  weight 30
```

Figure 6.1: Hodgepodge: bullets

to a grid. The grid is overlaid on the screen and is used to specify the positions of elements later on. We define bullets (bullets), antagonists (antags) and all upgrades (upgrades) in these elements.

6.2 The elements in *Hodgepodge*

Figure 6.1 shows the two bullet definitions mentioned in our final high-level description 6.1 as defined using the MAKU language. As is shown, MAKU is indentation sensitive, similar elements must be placed beneath each other in the same line for the document to be parsed. This definition says that we have two bullets named *tri* and *circ* respectively. They both have the same size, *s* signifying small and have shapes representative of their names, *tri* denoting triangle and *circle* denoting a circle. They are both pink and whilst ‘tri’ has weight of 10, ‘circ’ has a weight of 30. Weights control what can destroy what inside the game and can be thought of as denoting strength. Having defined these two bullets, we can now use them in the definition of our other elements.

Figure 6.2 shows an antag definition for an antag named “b1”. Its movement

```
antag
  name "b1"
  movement
    pattern panX
    track no
  size 1
  shape circle
  color pink
  weight 4000000
  score 20000
  quad
    lb "tri" 3
    rb "tri" 3
    lt "tri" 3
    rt "tri" 3
```

Figure 6.2: Hodgepodge: antag *b1*

pattern: panX means that this antag will move across the horizontal axis of the screen and track being no means it will not follow the protag around. We represent movement in MAKU via the use of special keywords that denote common patterns found in Bullet Hell games. MAKU is intended to be relatively easy to use and we believe that this abstraction helps to make the language more accessible to those not well versed in physical equations. Its size is 1, which denotes large and it has a weight set to 4000000. Upon destruction the protag receives a score of 20000 and it has a quad turret type.

The turrets lb, rb, lt and rt denote the different turrets local to the quad turret type. Each of which has been initialized with the *tri* bullet defined previously with the number 3 here denoting that each turret will fire three *tris* in the conical fashion (a *triple shot*) common to multiple fire in the genre.

Figure 6.3 shows the other antags defined for *Hodgepodge*, *b2* is almost exactly like *b1* except for its color, while antag *two* and antag *three* have traditional turret types, and introduce two new patterns. The pattern tb here denotes that the antag moves

```
antag
  name "b2"
  movement
    pattern panX
    track no
  size 1
  shape circle
  color pink
  weight 400000
  score 20000
  quad
    lb "tri" 3
    rb "tri" 3
    lt "tri" 3
    rt "tri" 3
antag
  name "two"
  movement
    pattern tb
    track no
  size s
  shape pentagon
  color green
  weight 10
  score 200
  traditional
    mt "tri" 3
antag
  name "three"
  movement
    pattern zigzag
    track no
  size s
  shape square
  color red
  weight 10
  score 200
  traditional
    mt "tri" 3
```

Figure 6.3: Hodgepodge: antag *b1*, *two* and *three*

```
upgrades
  shot "a" lt "circ" 3
  shot "b" rt "circ" 3
```

Figure 6.4: Hodgepodge: upgrades

```
random
  name "r1"
  using "three"
  side top
  count 100
  wait 1
random
  name "r2"
  using "two"
  side top
  count 100
  wait 1
```

Figure 6.5: Hodgepodge: randoms

from top to bottom on the screen, whilst zigzag means that antag *three* zigs and zags across the screen.

Of course, none of these finer points were specified in our high-level description but, we are trying to illustrate what is available in MAKU after all and hence have supplemented the description with our own specificities.

Figure 6.4 shows the upgrades in *Hodgepodge*. There are 2 shot upgrades definitions, *a* and *b*. As taken from our high-level specification, *a* augments the protag's lt and *b* augments its rt. They both use *circ* as their bullet and the number three denotes that the shot type for both upgrades is a *triple shot*.

That brings us up to the end of description for the elements in *Hodgepodge*. We have declared every element we wish to use and now describe the logic of *Hodgepodge* where we will utilize them.

```

single
  name "s1"
  using "b1"
  iP 2 2
single
  name "s2"
  using "b2"
  iP 2 18

```

Figure 6.6: Hodgepodge: singles

6.3 The logic in *Hodgepodge*

From our high-level specification, we know that we need to have some notion of randomization. Figure 6.5 shows us two MAKU random definitions, *r1* and *r2*. The using attribute denotes that *r1* and *r2* use antags *three* and *two* respectively. In MAKU randoms need to have a side defined that controls which side of the screen they spawn from. As shown *r1* and *r2* both start from the top. They also have the same count, which specifies that there should be 100 of both antags *three* and *two* spawned. Finally wait denotes the time that must pass between each individual spawn, in the case of *Hodgepodge*, one time unit which we call T.

We have handled everything except our “two larger stronger antagonists” defined above. Figure 6.6 shows two singles named *s1* and *s2* respectively. MAKU singles signify any singular entity that moves independently of any other entity. As shown, *s1* uses *b1* and *s2* uses *b2*. The attribute iP denotes the initial starting position of each single. This iP is of course related to the grid and we can see that *s1* starts at (row: 2, column: 2) and *s2* starts at (row: 2, column: 18).

We have dealt with all the elements in *Hodgepodge* as prescribed by the specification. We just need to tie them all together by specifying our two levels. Figure 6.7 shows the MAKU level specification for level one of *Hodgepodge*. Its protag is defined to be

```
level
  protag
    color green
    mt "circ" 1
    weight 100
    bombs 1
    lives 6
  timeline
    timestamp 0 "r1" 15
    timestamp 0 "r2" 15
    timestamp 3 "a" 10
    timestamp 6 "b" 10
  score 20000
```

Figure 6.7: Hodgepodge: level one

green, have a weight of 100, one bomb and 6 lives as per the specification. The keyword `mt` denotes that the protag's main turret uses `circ` and 1 denotes a single shot.

If we look at Level One's timeline, we see that it has four timestamps. Each timestamp connotes to some time based event happening in *Hodgepodge*. Our first timestamp signifies that at time: 0, `r1` should start spawning and that each spawn should take 15 time units to move across the screen. The second timestamp denotes `r2` while the third and fourth handle the updates outlined in the high-level description for level one. The final part of this level is the bound on the score which represents its win condition. The win condition for Level One has been set to any score greater than or equal to 20000.

Figure 6.8 depicts the specification for *Hodgepodge* Level Two and follows the high-level description for level two as well. The third and fourth timestamps here denoting that at time: 10, `s1` and `s2` should be spawned. With that we have completed our MAKU definition of *Hodgepodge*.

The full description can be found in Appendix A.1. The following figures also show screen-shots of *Hodgepodge* running in the browser. The following chapter, Chapter 7

```
level
  protag
    color green
    mt "circ" 5
    weight 100
    lives 6
  timeline
    timestamp 0 "r1" 15
    timestamp 0 "r2" 15
    timestamp 10 "s1" 15
    timestamp 10 "s2" 15
  score 10000
```

Figure 6.8: Hodgepodge: level two

gives a much more thorough breakdown of the intricacies of the MAKU language.

Chapter 7

The MAKU DSL

MAKU really has two separate parts. The first being a DSL that describes Bullet Hell games and the second being a game engine that implements the features found in the DSL.

This chapter outlines the MAKU language and Chapter 9 describe the MAKU engine.

MAKU is text based. The language itself is relatively small. A game can be completely specified using these keywords and the language itself is meant to be very readable. All keywords are lower case and blocks are white-space sensitive. Figure 7.1 shows the process of generation in MAKU and outlines the steps we will be exploring in more detail later.

7.1 A word on abstraction

MAKU provides different abstractions that allow the user to interface with concepts like speed, size and shape among others. Of course, speeds could be defined differently, but by providing these abstractions we are ensuring that first of all that specifications stay

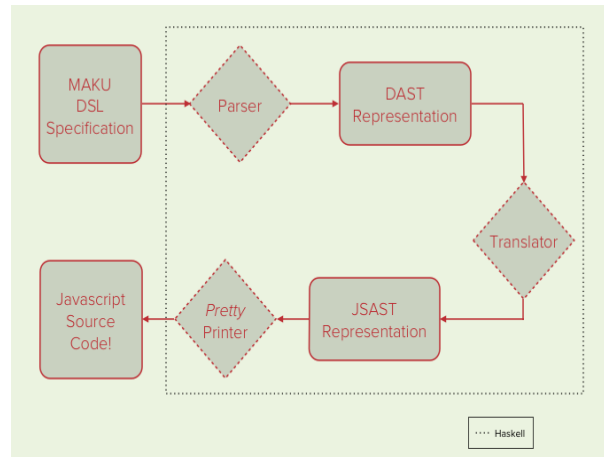


Figure 7.1: A diagrammatic representation of how a MAKU game is generated.

readable with very little context. A speed key with a value of *s* intuitively connotes to *this means the speed will be slow* and this sort of context-free readability was desirable for us.

Also, the G-Scale lab is interested in issues of visual scale. In playing with factors such as screen size, aspect-ratio and the user's physical proximity to the screen, we saw that the experience of a particular game varies with the augmentation of any of these aspects. Therefore, at least withing a linear scaling context (e.g. doubling the size of every element on the screen whilst keeping screen size constant), these abstractions help to keep game play consistent across a variety of different screen sizes and allow designers to specify their games in terms of how elements *should* look and behave. Especially, seeing as speed in MAKU is calculated with respect to the size of the player's screen. A unit of time is set in the system and every action in a game is given relative to this value of *T*.

MAKU has few constraints in order to allow the designer maximum control over their specifications. The designer has the freedom to create whatever they desire without any

```

<Game> ::= <Grid> <Elements> <Logic>
<Elements> ::= <Bullet>+ <Antag>+ <Upgrades>*
<Logic> ::= <Random>* <Single>+ <Group>* <Level>+

```

Figure 7.2: MAKU game in pseudo-BNF

regard for the sensibility of the final product. However, MAKU does institute particular constraints for certain elements. There are fixed options for speed, size, and many of the interactions between elements in games. For example, the relationship between an antagonists and protagonists is not available for manipulation.

7.2 A MAKU game

Games are made up of three aspects. Figure 7.2 shows the entire MAKU language description in a pseudo-BNF notation. We will use this notation to introduce each of the different aspects in MAKU as we move through them. We will first give the pseudo-BNF description, and then its corresponding MAKU language definition along with a brief explanation of the different choices available. Angle brackets denote *non-terminals*, a plus sign denotes one or more, and an asterisk denotes optionality. Words that are in this font face are MAKU language keywords.

Brackets denote *types* and are either:

- *Nat* - the natural numbers
- *AlphaString* - alphanumeric string
- *Color* - any colour that is defined in the SVG 1.1 specification(W3.org).

As shown, a game consists of a grid, the elements and the logic. We will discuss them by giving a high level description of each and then drilling down on each of the

```

<Elements> ::= <Bullet>+ <Antag>+ <Upgrades>*
<Bullet> ::= <Name> <SSCW>
<Antag> ::= <Name> <Movement> <SSCW> <Score> <TurretType>
<Upgrades> ::= <Shot>+ <Shield>* <Bomb>* <Life>*

```

Figure 7.3: A MAKU elements description

```

<Bullet> ::= <Name> <SSCW>

```

Figure 7.4: A MAKU bullet description

aspects that make them up. We will discuss elements first, talk about logic and in talking about it discuss grid as well.

7.2.1 The elements of a game

As seen in Figure 7.3, a MAKU element description consists of at least one bullet, at least one antag and optional upgrades.

What makes a bullet

MAKU bullets 7.4 must have a name, and an *SSCW*. *SSCW* as shown in 7.5 denotes an element's:

- size - can be either small, medium or large
- shape - can be either tri, square, rect, pentagon) or circle.

```

<SSCW> ::= <Size> <Shape> <Color> <Weight>
<Size> ::= <Small> | <Medium> | <Large>
<Shape> ::= <Triangle> | <Square> | <Rectangle> | <Pentagon> | <
  Circle>
<Color> ::= [Color]
<Weight> ::= [Nat]

```

Figure 7.5: A MAKU *SSCW* description

```
<Antag> ::= <Name> <Movement> <SSCW> <Score> <TurretType>
```

Figure 7.6: A MAKU antag description

```
<Movement> ::= <Pattern> <Track> <RotationData>
<Pattern> ::= <X>| <Y>| <PanY>| <PanX>| <Step>| <Zigzag>| <Spiral>|
  <Wave>| <Circular>| <LShaped>
<Track> ::= <Yes> | <No>
<RotationData> ::= <Direction> <Speed>
<Direction> ::= <CW> | <ACW>
<Speed> ::= <VS> | <S> | <M> | <F> | <VF>
```

Figure 7.7: A MAKU movement description

- `color` - any valid Colour.
- `weight` - corresponds to how *strong* an element is and is a natural number. A bullet with a weight of 60 will destroy an antag with a weight of 40. Weights are calculated using arithmetic and therefore a bullet with a weight of 20 will decrease an antag weight of 60, by 20 to 40.

What makes an antag

The structure of a MAKU antag is shown in Figure 7.6 and must have `name`, `movement`, `SSCW`, `score` and `TurretType` defined. The `score` denotes how much each antag is worth upon destruction. We will break to talk about `movement` and `TurretType` before continuing on to *upgrades*.

7.2.2 With respect to movement

The attribute `movement` defines as you would expect, how an antag moves. As Figure 7.7 shows, `movement` is defined with a `pattern`, `track` and `RotationData`. A `pattern` denotes the set of built in movement patterns and are based on common antag movement

Movement Pattern	Direction
x	Horizontally
y	Vertically
panX	Horizontally bounded by the edges of the screen
panY	Vertically bounded by the edges of the screen
step	In a step like movement bounded by the edges of screen
zigzag	In a so-called <i>zig-zag</i> like fashion
spiral	In a spiral
wave	Like a <i>sin wave</i>
circular	In a circle
lShaped	y followed by x bounded by the edge of the screen

Table 7.1: MAKU movement patterns and descriptions

```

<TurretType> ::= <Traditional> | <Double> | <Quad>
<Traditional> ::= <MainTurret> <LeftTurret>* <RightTurret>*
<Double> ::= <Turret> <Turret>
<Quad> ::= <Turret> <Turret> <Turret> <Turret>
<Turret> ::= <Bullet> <ShotType>
<ShotType> ::= [Nat]

```

Figure 7.8: The MAKU turret descriptions

patterns found in Bullet Hell games. Table 7.1 contains the patterns currently defined in the MAKU language.

Track controls whether or not the antag is attracted to the player's avatar and is either yes or no. *RotationData* denotes the rotation attribute of movement handles whether or not the antag rotates about a fixed position and how quickly it does so. It can be either anticlockwise (acw) or clockwise (cw), while the speed can be either very-slow(vs), slow (s), medium (m), fast (f) or very-fast (vf).

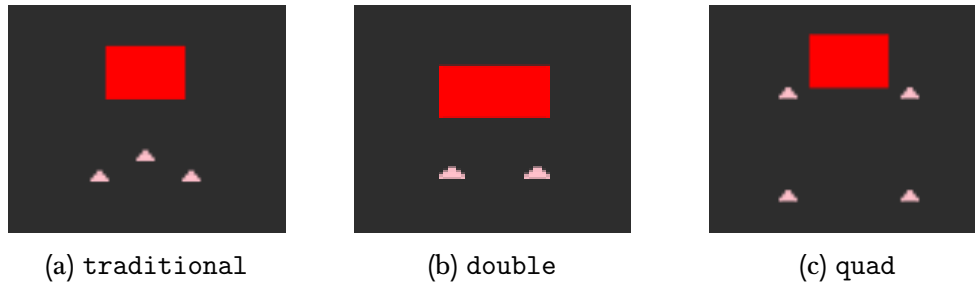


Figure 7.9: MAKU turret configurations

7.2.3 With respect to *TurretType*

Turrets have types in MAKU, and are currently either traditional, double or quad. They are, much like movement pattern, informed by the domain of Bullet Hell games and commonalities therein.

Figure 7.9 shows the different turret types and what they connote in the final analysis. One thing to note is that traditional turrets only need one turret defined. The first connotes to the main turret or `mt` whilst `lt` and `rt` denote the left and right turret respectively. double defines its `lt` and then `rt` while quad defines `lb` (left-bottom), `rb` (right-bottom), `lt` (left-top) and `rt` (right-top) in that order.

As shown in Figure 7.8 turrets, we focus on the makeup of said turrets and see that turrets must be defined with a bullet name and a *ShotType*. In the many different Bullet Hell games released over the years, there have been many different types of shot. We call the pattern in which the bullet(s) propagate themselves over time the *ShotType*.

The MAKU language currently has two *ShotTypes*. A *single* shot and a *many* shot. The non-negative integer one connotes to *single* shot. *Single* shot as you expect denotes that there is one bullet fired from that particular turret. Any number greater than one, invokes *many* shot. *Many* shot takes the non-negative integer and multiplies the specified bullet by it and fires them in a conical shape.

```
<Upgrades> ::= <Shot>+ <Shield>* <Bomb>* <Life>*
<Shot> ::= <Name> <Turret>
<Shield> ::= <Name> <Weight> <Color>
<Bomb> ::= <Name> <Count>
<Life> ::= <Name> <LifeType> <Count>
<LifeType ::= <Add> | <Mult>
<Count> ::= [Nat]
```

Figure 7.10: A MAKU upgrades description

What are upgrades?

The definable upgrades as shown in Figure 7.10 in MAKU are:

- shot - denotes the shot upgrades and must have a name and a turret defined. This turret specifies which of the protag turrets to augment upon being equipped.
- shield - denotes any shields that are available to the protag and must have a name, along with a weight specifying how much more weight to add to the protag along with a color specifying which color the protag changes to upon being equipped.
- bomb - denotes any bombs available to the protag and is defined with a name and a number signifying the increase in number of bombs to had upon being equipped.
- life - denotes the type of life upgrade available to the protag. *LifeType* here, can be either add (add some number to the current protag life count) or mult (multiply the current protag life count by some number) and of course a count, which signifies the number to be added or multiplied by.


```

<Logic> ::= <Random>* <Single>+ <Group>* <Level>+
<Random> ::= <Name> <Using> <Side> <Count> <Wait>
<Single> ::= <Name> <Using> <InitialPosition>
<Group> ::= <Name> <Using> <Lanes> <Count> <InitialPosition>
<InitialPosition> ::= <Row> <Column>
<Level> ::= <Protag> <Timeline> <WinCondition>
<Using> ::= <Antag>
<Side> ::= <Left> | <Top> | <Right> | <Bottom>
<Row> ::= [Nat]
<Column> ::= [Nat]
<Lanes> ::= [Nat]
<Count> ::= [Nat]
<Lanes> ::= [Nat]
<Wait> ::= [Nat]

```

Figure 7.11: The MAKU logic description

```

<Grid> ::= [Nat]

```

Figure 7.12: The MAKU grid description

7.2.4 The logic of a game

As seen in Figure 7.11, a MAKU element description must consist of at least one single and at least one level, and optional randoms and groups.

There is an inherent time cycle build into all MAKU games. This time cycle is simply called as T . Every action in the game is interpretable in units of T .

Position in the MAKU language really denotes *InitialPosition*. Every single or group that appears on screen must have some *InitialPosition* (called iP) defined. We use a grid system (Figure 7.12) to control for this and overlay grid lines on the screen. The user can then specify the initial position in terms of row and column.

With all of that in mind, they are outlined as follows:

- single - must have name, using and *InitialPosition* defined. This name is how the single will be referenced and the using (which is common to groups as well)

```

<Level> ::= <Protag> <Timeline> <WinCondition>
<Protag> ::= <Colour> <Traditional> <Weight> <Lives> <Bombs>*
<Lives> ::= <Infinite> | [Nat]
<Bomb> ::= [Nat]
<Timeline> ::= <Timestamp>+
<Timestamp> ::= <Start> <What> <Duration>*
<Start> ::= [Nat]
<What> ::= <Random> | <Single> | <Group> | <Upgrades>
<Duration> ::= [Nat]
<WinCondition> ::= <Boss> | <Score> | <Time>
<Boss> ::= <Single>
<Score> ::= [Nat]
<Time> ::= [Nat]

```

Figure 7.13: The MAKU level description

denotes which previously defined antag is being utilized.

- random - must have name, using, side which denotes which size the randomizer spawns antags from, either left, top, right, bottom. It must also have a count specifying the number of antags to be spawned along with a wait (how many units of T until the next spawn cycle.)
- groups - must have name, using, lanes which denotes how many lanes of antags there are to be displayed on the screen as well as a count which details how many antags should be present in each lane.
- level - consists of the protag, timeline and *WinCondition* shown in Figure 7.13 which we will discuss next.

7.2.5 The MAKU protag

The protag in accordance with the previous definition must have color, weight, lives and particularly, a traditional turret type defined. lives of course, denote how many

chances the protag has to complete the level. It can be stated using numbers or defined as infinite. bombs are optional and denote how many bombs the protag begins each level with.

7.2.6 The MAKU timeline

MAKU's timeline is a powerful construct allowing for the specification of non-trivial levels. As shown, each timeline contains one or more timestamps. The timestamp is a combination of *Start* which denotes the value of T at which the spawn cycle should be started. The *What* value denotes the what should be spawned: either random, single, group or upgrades and finally the optional *Duration* denotes how many cycles of T the spawned element should remain on screen. There is an inherent velocity built into the MAKU engine, the duration specification controls the movement of entities across the screen. This specification alters the speed of an entity with respect to the size of the screen. Of course, if none is specified every spawned elements velocity defaults to the built in value.

7.2.7 WinConditions in the MAKU language

There are currently three *WinConditions* in the MAKU language:

- score - which denotes a particular value at which, once attained a level finishes.
- time - which denotes a particular value of T at which if the protag is still alive a level finishes
- boss - which denotes some single that acts as the final enemy to be defeated, which upon defeat the level finishes.

7.3 Extensibility

We have given a complete description of all the facets of the MAKU language. MAKU is completely extensible and new features can be added at will. It is really a combination of other *micro* DSLs, that specify things like weighted shapes and shot types and patterns to name a few. Each level description can be thought of being written in a level-specific DSL. Extending any of the smaller DSLs in turn extends the MAKU language itself.

Chapter 8

The MAKU Generator

This chapter will give a very basic explanation of how a DSL is translated into actual GPL source code. In this case, our DSL being the MAKU language and the GPL being JavaScript. The generator is actually three components that supply input for each other.

These components are:

1. Two internal Abstract Syntax Trees (ASTs).
2. The translator between the two. This translator normally performs optimizations on the code, such as *function inlining* and loop elimination.
3. The *pretty* printer that produces the GPL source code.

8.1 The ASTs

We will not go through each and every aspect of the ASTs, just enough to give the reader a clear understanding of what indeed is going on.

```
data Game = Game {
  getGrid :: Grid,
  getElements :: Elements,
  getLogic :: Logic
} deriving (Show)
```

Figure 8.1: The game data type as defined in Haskell

The first AST is the AST that mirrors the design of the MAKU language, while the second is an AST that describes a subset of the JavaScript language.

It should be noted that ANY language could be used as the target language for a MAKU game. We chose JavaScript as our final target for the portability reasons we specified in Chapter 10, but any language can be used. All that would be required is an AST for the language along with a reworking of both the translator and printer.

8.1.1 The Design AST (DAST)

The DAST is a complete reflection of the pseudo-BNF notation of the MAKU language as given in Chapter 7. Figure 8.1 shows the data structure that the game upon parsing is stored in as it is represented in Haskell. Figure 8.3 shows how MAKU movement is represented in Haskell, in particular the use of Haskell's `Maybe` to build optionality into the data structure itself. Figure 8.2 shows how MAKU logic is defined and shows the use of the `NEList` data type which enforces the rule of *at least one* for singles, groups and levels.

8.1.2 The JavaScript AST (JSAST)

The JSAST is a trimmed down representation of JavaScript as defined by Crockford [2008] in *JavaScript: The Good Parts*. It was adapted from previous work by Seefried

```
data Logic = Logic {
  getRandoms :: [Random],
  getSingles :: NEList Single,
  getGroups  :: NEList Group,
  getLevels  :: NEList Level
} deriving (Show)
```

Figure 8.2: The logic data type as defined in Haskell

```
data Movement = Movement {
  getPattern :: Maybe MovementPattern,
  getTrack   :: Bool,
  getRotation :: Maybe RotationData
} deriving (Show)
```

Figure 8.3: The movement data type as defined in Haskell

[2012], in particular the precedence of JavaScript operators. There are no function declarations in the JSAST, function literals can however be assigned to variables. The use of an internal JS data structure allows us to be confident that any generated JavaScript is *correct by construction*. We did not need *all* the features of JavaScript and so the AST has been tailored to our needs.

8.2 The translator

The most important part of the process, the translator in MAKU is responsible for the heavy lifting that transforms a MAKU language game description into pretty-printable JavaScript source representation. The translator is really two parts:

1. Part One: The translation of a MAKU description into a MAKU engine `init()` method.
2. Part Two: The assembling of a version of the engine specific to the requirements

of *that* description.

8.2.1 Part One: `init()`

As mentioned earlier, a MAKU game description is really an initialization method. This part of the translator takes the internal representation of the description given in terms of internal representation data types (DAST) and produces JSAST representation. In this step, the translator traverses the data structure and via the invocation of the relevant functions translates the tree between representations. There are a few decisions made in this step:

- On the description side upgrades, groups etc. are specified in terms of shot, shield etc. in timeline descriptions. Internally, these are represented in terms of integers, 0 corresponding to groups and so on and so this analysis and decision is made.
- The translation from Nat representation of the shot type to the *SingleShot*, *ManyShot* representation used internally by the system.

Figure 8.4 shows the translation function that interprets the DAST: *Shot* into the JSAST: *Expr*. We will see the corresponding functions, denoted here as `P.straightShot` and `P.manyShot` respectively when we discuss the pretty printer, the P denotes functions native to the pretty printer.

8.2.2 Part Two: The engine assembly

The MAKU Engine is actually a combination of different functionalities collected relevant to the specific description of a particular game. It is sensitive to the selections in the


```

shot :: Shot -> Two -> [Bullet] -> Expr
shot x@(Shot _ t) Zero b =
  let tt = typ t in
  case tt of
  1 -> P.straightShot s' P.true w'
  _ -> P.manyShot tt s' P.true w'
  where
    (s',w') = bulletShape x b

shot x@(Shot _ t) One b =
  let tt = typ t in
  case tt of
  1 -> P.straightShot s' P.false w'
  _ -> P.manyShot tt s' P.false w'
  where
    (s',w') = bulletShape x b

```

Figure 8.4: Translation function for a MAKU shot

description and eliminates conditional expressions and loop constructs appropriately. A far greater detailed discussion can be found in Chapter 9.

8.2.3 The *pretty* printer

The final stage of the process that takes the JSAST representation of both designed game's `init()` method and engine. The printer contains functions analogous to that of the translator and produces human readable source code. Granted, as MAKU *minifies* all JavaScript at the end of the generation process, this human readability is lost, but the pretty printer is what actually turns the representation into source and is vital to the generation process. Figure 8.5 shows the corresponding pretty printer functions for the function mentioned in Figure 8.4. The function `paramInvk` is a helper function that helps to make an otherwise ungainly JSAST representation *writable*. The printer makes use of many functions in this vain for this very purpose, indeed the function `n`, applied to `w` in both functions is a function corresponding to a JSAST number.

```
straightShot :: Expr -> Expr -> Double -> Expr
straightShot b polarity w = paramInvk __straightShot [b, polarity, n
w]

manyShot :: Double -> Expr -> Expr -> Double -> Expr
manyShot x b polarity w = paramInvk __manyShot [n x, b, polarity, n
w]
```

Figure 8.5: Pretty printer function for a MAKU shot

Chapter 9

The MAKU Engine

For all intents and purposes, the MAKU engine has only one requirement: It must implement the features as prescribed by the DSL. As far as implementation languages or techniques goes, given that the features in the DSL are represented in the engine, they do not really matter. Essentially, the MAKU engine is an interpreter for a game description's generated `init()` method. This means that each incarnation of engine is directly related to its game specification. Figure 9.1 shows the process of the engine during interpretation of a specification.

The engine is made up of models and controllers (MC) directly related to the MAKU language construct. This notion of model and controller is derived from the software-architectural pattern often used in implementing user interfaces. The separation of concerns which depicts a model as describing the problem domain independent of implementation details while the controller is responsible for the manipulation of the model's state and served as an interface between the *front-end* and *back-end*.

For example, the `bullet` model (see: Figure 9.2) is called *Bullet* while its controller (see: Figure 9.3) is *Bullets*. All models are built on the base *Entity* model, which provides

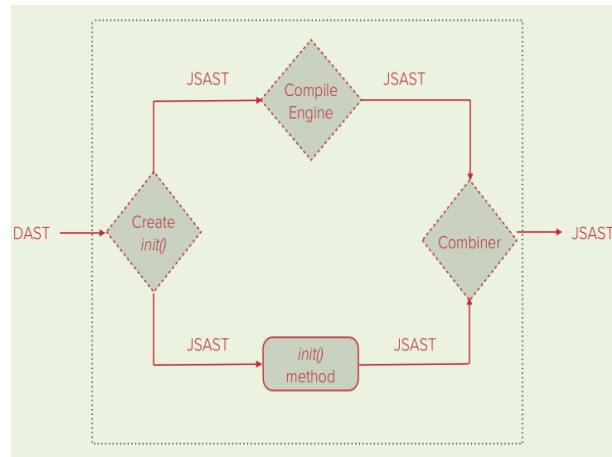


Figure 9.1: A diagrammatic representation of the translation process.

functionality common to all children. The engine itself manipulates different Paper.js *layers* and shows and hides entities via manipulation of their children. The *Environment* construct is the global controller and delegates the start/pause/end of the game as well as other functions.

9.1 Static vs. Dynamic

In order to prioritize what *should* be generated, information which we classified as being either static or dynamic. Traditionally, static and dynamic have conventional meanings in the domain of code generation which are analogous to their meaning here. Static typically denotes that information that is known at compile time whilst dynamic depicts information that the system cannot know until runtime. In MAKU, we can see this relationship between static and dynamic as static denotes that information vital to every generated game whilst dynamic refers to the compositional nature of the information provided at runtime based on that particular game's specification.

The initial incarnation of the engine was one large file with all the functionality to

```
var Bullet = function(shape, weight) {
  Entity.call(this);
  var self = this;
  self.shape = shape;
  self.weight = weight;
  self.inView = true;
};
Bullet.prototype = new Entity();
Bullet.prototype.constructor = Bullet;
Bullet.prototype.getPosition = function() {
  var self = this;
  if (self.alive)
    return self.shape.getPosition();
  return false;
};
Bullet.prototype.updateWeight = function(weight) {
  var self = this;
  self.kill();
};
```

Figure 9.2: The model for MAKU bullets in Javascript

interpret the method. We sought to chop this file up into our two categories. We found that for particular representational constructs, what fluctuated was the necessity of their various models and controllers. Figure9.1 shows this breakdown.

As shown we see that most constructs have a both a static and dynamic portion. We separated the static portions into files to be retrieved and combined at compile time, whilst converting all dynamic portions into JSAST representation, to be printed at the appropriate time. Figure 9.4 shows the movement pattern *PanHorizontal* as represented in JSAST and Figure 9.5 the analogous *PanHorizontal* JavaScript method it replaced.

9.2 Optimizations

Knowing what exactly we *need* in the engine allowed us to make certain decisions about what was instrumental to the final generated engine, the generation process monitors for

```

var Bullets = {
  init: function() {
    var self = this;
    self.bullets = [];
  },
  move: function() {
    Protagonist.bulletMovement();
    Antagonists.bulletMovement();
  },
  manage: function() {
    Protagonist.manageBullets();
    Antagonists.manageBullets();
  },
  positiveCollision: function(shot) {
    ...
  },
  negativeCollision: function(shot) {
    ...
  },
  getBullets: function () {
    var self = this;
    return self.bullets;
  }
};

```

Figure 9.3: The controller for MAKU bullets in Javascript

```

_panH :: Program
_panH = complexMpProg __panH [] [directionLeft] [ self, _width,
_barrier, _leeway __width]
[
  iff (eq (ref (exnm __self) (prop __direction)) (exstr __right))
  [
    posXPlus,
    iff (gte shapePositionX (exnm __leeway)) [directionLeft]
  ]
,
  iff (eq (ref (exnm __self) (prop __direction)) (exstr __left))
  [
    posXMinus,
    iff (lte shapePositionX (exnm __barrier)) [directionRight]
  ]
]

```

Figure 9.4: *PanHorizontal* movement pattern as defined in JSAST

Construct	Model	Controller
<i>Entity</i>	✓	✓
<i>Bullets</i>	✓	✓
<i>Protagonist</i>	✓	◇
<i>Antagonists</i>	◇	◇
<i>Levels</i>	◇	◇
<i>Singles</i>	✓	✓
<i>Timestamp</i>	✓	●
<i>Environment</i>	✓	✓
<i>MovementPattern</i>	◇	●
<i>ShotType</i>	◇	●
<i>TurretType</i>	◇	●
<i>Randoms</i>	◇	●
<i>Groups</i>	◇	●

Table 9.1: Static and dynamic categorizations by model and controller

- Symbols:
- ✓ Fully static
 - ◇ Some static and some dynamic attributes
 - × Fully dynamic
 - Not applicable

```
var PanHorizontal = function() {
  MovementPattern.call(this);
  var self = this;
  self.direction = 'right';
}
PanHorizontal.prototype = new MovementPattern();
PanHorizontal.prototype.constructor = PanHorizontal;
PanHorizontal.prototype.move = function(shape, speed) {
  var self = this;
  var width = view.bounds.width;
  var barrier = 100;
  var leeway = width - barrier;
  if (self.direction == 'right') {
    shape.position.x += speed;
    if (shape.position.x >= leeway) {
      self.direction = 'left';
    }
  }
  if (self.direction == 'left') {
    shape.position.x -= speed;
    if (shape.position.x <= barrier) {
      self.direction = "right";
    }
  }
};
```

Figure 9.5: *PanHorizontal* movement pattern as defined in Javascript

different selections and optimizes where necessary. The current optimizations are :

- Only include relevant movement patterns MCs.
- Only include relevant shapes MCs.
- If protagonist tracking is enabled then include tracking methods.
- Likewise for rotation and turret definitions.
- Only include relevant shot type MCs.
- Only include relevant upgrade MCs.
- If there are defined upgrades/randoms/groups include their MCs and all relevant methods. This involves, customizing the level handler to handle only those defined types of timestamp.
- If the protagonist is defined with bombs, include the ability to deploy bombs in the key controller.
- Eliminate loops/conditionals for single definitions of upgrades/randoms/groups etc. For example the `Random.prototype.assignPosition` method that defines a physical screen position to randomly generated antags. Figure 9.6 shows the generated method when all sides are defined in the game description as opposed to when just one, top is defined (Figure 9.7).

Portability of the generated games was a major point in MAKU's development, the engine was implemented in JavaScript and HTML5 Canvas. It could have just as easily been C++, or Java. This of course allows for the many different games to be specified using the MAKU language while keeping the back-end implementation details changeable.

```
Random.prototype.assignPosition = function (grid) {
  var self = this;
  var p = arbitrary(grid);
  if (self.side === 'left') {
    return Environment.actualPosition({
      col: 0.0, row: p
    });
  }
  if (self.side === 'right') {
    return Environment.actualPosition({
      col: grid, row: p
    });
  }
  if (self.side === 'top') {
    return Environment.actualPosition({
      col: p, row: 0.0
    });
  }
  if (self.side === 'bottom') {
    return Environment.actualPosition({
      col: p, row: grid
    });
  }
  return false;
};
```

Figure 9.6: Random.prototype.assignPosition: all sides defined

It goes without saying that anything not prescribed by the DSL, must be defined by the engine. For example, the difference between how shot, shield and other upgrades is not prescribed by the DSL, which denotes that the engine must have methods of controlling for them. Similarly, attributes like protag shape and color are non-definable and as such must be built into the protag definition inherent in the engine.

```
Random.prototype.assignPosition = function (grid) {  
  var self = this;  
  var p = arbitrary(grid / 2.0);  
  return Environment.actualPosition({  
    col: 0.0,row: p  
  });  
};
```

Figure 9.7: Random.prototype.assignPosition: only left defined

Chapter 10

Solution Decisions

This chapter discusses the various design decisions made with respect to the languages used in the implementation of MAKU.

Why Haskell?

Functional programming languages are well suited to generic programming. Garcia et al. [2007]. Haskell, due to its strong type inferencing system, pattern matching, useful syntactic sugaring capabilities and the authors familiarity with it was chosen as the implementation language for parsing and translation of the DSL. Haskell also provided an interface to color keywords as defined in SVG 1.1 W3.org which proved to be very useful in the design of the MAKU language.

Why a browser based solution?

In the early stages when issues of visual scale were the principal motive, a desired feature of the final games was the ability to play them on a variety of different screen sizes.

JavaScript in tandem with HTML5 Canvas were chosen for their portability. Games built using HTML5 Canvas can be highly sophisticated whilst being completely device agnostic and that was a major factor in making the decision to use it. This sophistication freed us from having to develop native games across the different devices.

Paper.js is an open source vector graphics scripting framework that runs on top of the HTML5 Canvas. Jonathan Puckey It was the foundation on which the JavaScript game engine was built. Paper.js provides a clean interface to HTML5 Canvas with useful abstractions for useful HTML5 Canvas constructs.

Chapter 11

Testing

Testing MAKU required the testing of the following aspects.

1. Ensuring that descriptions given in the MAKU language were semantically and syntactically correct, along with ensuring that the generated JS source was valid.
2. The testing of the MAKU engine.

11.1 Testing for description errors

The process of evaluating a MAKU DSL game description is incremental. The description of antagonists and upgrades relies on the prior description of bullets. The description of logic requires the prior description of the elements.

The system therefore passes this relevant information freely throughout the different functions responsible for parsing and organizing into data types. There are 2 safeguards when parsing a description. The first being that, all input is expected to be from a particular set depending on what is being parsed and is compared to this set. Also, the

rigidity of the data types in Haskell, allow us to be sure that there will be no ill formed data types.

We used sample games with improper specifications for **each** attribute of each of the game elements and logic respectively. In each case, the parser returned blocking errors that do not allow the system to progress until valid data has been entered.m

We then needed to make sure that any changes made in the description were represented in the final generated source. This however, was a trivial endeavor due to the end result being completely visual. Each individual feature of the MAKU language tested in browser to make sure that any changes made to shapes, size, rotation, movement pattern, updating of weapons etc. would all be represented and working in any generated game.

11.2 Testing the MAKU engine

In testing the engine, we were trying to see if all behaviour happened as expected with respect to a game environment. It really consisted of implementing and testing the different mechanics in the MAKU engine

Mechanics such as:

1. Collision detection
2. Movement of the player's avatar
3. Updating the player's life count and score
4. Ensuring that each movement pattern corresponds to its to specified name.
5. Ensuring that they was no *level-bleed*, that is, that all on-screen entities are destroyed at the end of each level.

6. Ensuring that bullets propagate in an expected fashion.
7. Managing of protagonist and antagonist weight
8. Managing of time and level progression

All mechanics were tested in browser using a stripped down example game. We also performed optimizations on the MAKU engine to ensure as less latency as possible whilst playing the game. For example, ensuring that bullets are not tracked after leaving the player's viewport. During generation, the final step is to *minify* or compress the generated JS source with this optimization in mind.

Chapter 12

Conclusion

The Bullet Hell game generator described in this thesis allows for the design and generation of playable games from the set of all Bullet Hell games. The games are portable and error free within the constraints.

The project itself and the MAKU language and the MAKU engine are quite concise. That being, said, they are wholly extendable, and demonstrates that there are many games that are composable based on the sum of commonalities across a particular genre of game.

Indeed, MAKU describes not just a subset of the larger set of Bullet Hell games, but also describes a subset of the larger set of all SHMUPs as well. As more and more intelligence is added to the system, it becomes possible to create more permutations of games. However, the genre of the games themselves is changing, but MAKU is built with these changes in mind. Any change once added to the MAKU language and implemented in either the MAKU engine or any other engine, become representable using MAKU and adds more and more generatable varieties of Bullet Hell game.

12.1 Related Work

The main difference between MAKU and other systems is that MAKU's purpose is completely singular. That is the creation of playable Bullet Hell games. MAKU allows the designer to go straight from specification to gameplay. Other work in this domain has been primarily concerned with two things, the application of artificial intelligence (Smith and Mateas [2010], Love et al. [2008]), and procedural generation (either content or mechanics), (Cook [2013], Hastings et al. [2009]). These can be related to systems specifically developed to "play games well" (Pell [1996]), other which are focused on turn-based gaming (Jackson [2014]) and other specifically bound to the analysis of either board or card games, Font et al. [2013].

The most similar system to MAKU in concept is EGGG: The Extensible Graphical Game Generator (Orwant [2000]) which focuses on the generation of games from their "rules of play". EGGG is well suited toward the creation of game involving pieces, boards, cards and icons and implements a game engine using the Perl programming language. The main difference between MAKU and EGGG is that MAKU seeks to do one thing very well, (i.e. describing Bullet Hell games), whereas EGGG allows the user to describe different genres of games in a more general fashion without such fine-grain fidelity.

12.2 Limitations of the MAKU language

The final BH game that was built before this work was begun had one feature that differentiated it from the others.

The feature was the fact that the player's avatar was controlled via moving the mouse cursor. MAKU provides no such construct and as such, specifying a game that requires

anything other than keyboard control is impossible.

However, the movement of protagonists in BH games could be modeled and organised into its own DSL. This could then be integrated into MAKU to allow for the specification of games that use non-keyboard control.

In the development of this work, it was decided to adhere to the conventions and focus on the basics. This mentality represents itself in many of the design decisions in the MAKU language. The aforementioned movement control fixity, the shape and colour of the protagonist being fixed, the absence of constructs that allows for the developer to control the speed of bullets and the shapes and movement of the upgrades while deemed to be non-essential in the current MAKU specification, are intended to become the subject of future work.

12.3 Future Work

The possibilities for future work are immense, and there are many avenues for expansion, such as:

- Adding a chained scoring system to the MAKU language, where the scoring for destruction of enemies is linked to how many many enemies are destroyed at once.
- Identifying patterns and implementing constructs in the MAKU language for complex storytelling. For example, cutscenes and on-screen text.
- Adding new win conditions, such as *search and rescue* to the MAKU language.
- Allowing for antags to detach from their groups, and allowing different scoring mechanisms for pre and post departure.

- Adding constructs that allow the developer to specify different control structures for protags. For example, keyboard and mouse control.
- Allowing for much more user defined attributes with respect to both protag and upgrade size, shape, colour etc.

Appendix A

Example MAKU Files

A.1 Example.maku

```
grid 20
elements
  bullet
    name "single"
    size m
    shape tri
    color pink
    weight 10
  antag
    name "first"
    movement
      pattern panX
      track yes
    size s
    shape circle
    color white
    weight 10
    score 200
    double
      lt "single" 3
      rt "single" 2
  antag
    name "second"
    movement
      pattern circular
      track no
      rotation acw f
    size s
    shape circle
    color red
```

```

weight 10
score 600
quad
  lb "single" 4
  rb "single" 4
  lt "single" 3
  rt "single" 3
antag
  name "third"
  movement
    pattern circular
    track no
  size s
  shape circle
  color red
  weight 10
  score 600
  traditional
    mt "single" 1
    lt "single" 3
    rt "single" 2
upgrades
  shot "4shot" mt "single" 4
  shot "lt2shot" lt "single" 2
  shield "hyaku" 100 blue
  bomb "bakudan" 5
  life "fiveMore" add 5
  life "doubler" mult 5
logic
  random
    name "rand1"
    using "antagName4"
    side bottom
    count 5
    wait 2
  single
    name "bossSingle"
    using "boss1"
    iP 2 10
  group
    name "groupie"
    using "antagName"
    lanes 1
    count 2
    iP 3 4
  group
    name "frennie"
    using "antagName"
    lanes 2
    count 5
    iP 10 10
  level
  protag
    color green
    mt "single" 1
    lt "single" 2
    rt "single" 2
  weight 100

```

bombs 100	weight 100
lives 6	lives infinite
timeline	timeline
timestamp 0 "groupie"	timestamp 0 "groupie"
time 300000	timestamp 2 "frennie" 3
level	timestamp 3 "4shot" 0.5
protag	score 9000
color green	boss "boss1"
mt "single" 1	

A.2 hodgepodge.maku

grid 20	name "b1"
elements	movement
bullet	pattern panX
name "tri"	track no
size s	size 1
shape tri	shape circle
color pink	color pink
weight 10	weight 4000000
bullet	score 20000
name "circ"	quad
size s	lb "tri" 3
shape circle	rb "tri" 3
color pink	lt "tri" 3
weight 30	rt "tri" 3
antag	antag

```

name "b2"
movement
  pattern panX
  track no
size 1
shape circle
color pink
weight 400000
score 20000
quad
  lb "tri" 3
  rb "tri" 3
  lt "tri" 3
  rt "tri" 3
antag
  name "two"
  movement
    pattern tb
    track no
  size s
  shape pentagon
  color green
  weight 10
  score 200
  traditional
    mt "tri" 3
antag
  name "three"
  movement
    pattern zigzag
    track no
  size s
  shape square
  color red
  weight 10
  score 200
  traditional
    mt "tri" 3
  upgrades
    shot "a" lt "circ" 3
    shot "b" rt "circ" 3
  logic
    random
      name "r1"
      using "three"
      side top
      count 100
      wait 1
    random
      name "r2"
      using "two"
      side top
      count 100
      wait 1
    single
      name "s1"
      using "b1"
      iP 2 2

```



```
single                                     timestamp 6 "b" 10
  name "s2"                                score 20000
  using "b2"                                level
  iP 2 18                                   protag
level                                       color green
  protag                                    mt "circ" 5
  color green                               weight 100
  mt "circ" 1                              lives 6
  weight 100                               timeline
  bombs 1                                  timestamp 0 "r1" 15
  lives 6                                   timestamp 0 "r2" 15
timeline                                    timestamp 10 "s1" 15
  timestamp 0 "r1" 15                       timestamp 10 "s2" 15
  timestamp 0 "r2" 15                       score 10000
  timestamp 3 "a" 10
```

Bibliography

E. Adams. Sorting Out the Genre Muddle.

<http://www.gamasutra.com/view/feature/4074>, 2009. Accessed: 2014-06-11.

T. H. Apperley. Genre and game studies: Toward a critical approach to video game genres. *Simulation and Gaming*, 37(1):6–23, 2006. doi: 10.1177/1046878105282278.

URL <http://sag.sagepub.com/content/37/1/6.abstract>.

D. Arsenault. Video game genre, evolution and innovation. *Eludamos. Journal for Computer Game Culture*, 3(2), 2009. ISSN 1866-6124. URL <http://www.eludamos.org/index.php/eludamos/article/view/vol3no2-3/126>.

B. Atkins. *More Than a Game: The Computer Game As Fictional Form*. Manchester Univ Pr, 2003. ISBN 0719063655.

L. Beyak and J. Carette. SAGA: A DSL for story management. In *Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011, Bordeaux, France, 6-8th September 2011*, pages 48–67, 2011. doi: 10.4204/EPTCS.66.3. URL <http://dx.doi.org/10.4204/EPTCS.66.3>.

- J. D. Bolter and R. Grusin. *Remediation Understanding New Media*. 2000. ISBN ISBN-10: 0-262-52279-9 ISBN-13: 978-0-262-52279-3. URL <http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=3468>.
- Bungie Inc. Halo: Combat Evolved. CD-ROM, 2001. Published by: Microsoft Game Studios.
- J. Carette and O. Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, GPCE'05, pages 256–274, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-29138-5, 978-3-540-29138-1. doi: 10.1007/11561347_18. URL http://dx.doi.org/10.1007/11561347_18.
- M. Cook. Creativity in code: Generating rules for video games. *XRDS*, 19(4):40–43, June 2013. ISSN 1528-4972. doi: 10.1145/2460436.2460449. URL <http://doi.acm.org/10.1145/2460436.2460449>.
- D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008. ISBN 0596517742.
- B. Curutan. CPCG: A Cross-Paradigm Code Generator. Master's thesis, McMaster University, 2013.
- K. Czarnecki, T. O'Donnell, John, J. Striegnitz, and W. Taha. *DSL Implementation in MetaOCaml, Template Haskell, and C++*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72. Springer-Verlag, Berlin, Heidelberg, 2003. doi: 10.1007/b98156. URL <http://www.springerlink.com/content/nl620e17n94m161h/fulltext.pdf>.

- EA Digital Illusions CE. Battlefield 1942. CD-ROM, 2002. Published by: Electronic Arts, Asper Media.
- J. Font, T. Mahlmann, D. Manrique, and J. Togelius. A card game description language. In A. Esparcia-Alcázar, editor, *Applications of Evolutionary Computation*, volume 7835 of *Lecture Notes in Computer Science*, pages 254–263. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-37191-2. doi: 10.1007/978-3-642-37192-9_26. URL http://dx.doi.org/10.1007/978-3-642-37192-9_26.
- M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321712943, 9780321712943.
- G. Frasca. Simulation versus narrative: introduction to ludology. In M. J. P. Wolf and B. Perron, editors, *The Video Game Theory Reader*. Routledge, London/New York, 2003.
- R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17(2):145–205, Mar. 2007. ISSN 0956-7968. doi: 10.1017/S0956796806006198. URL <http://dx.doi.org/10.1017/S0956796806006198>.
- R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. *Lisp Symb. Comput.*, 10(2):113–158, July 1997. ISSN 0892-4635. doi: 10.1023/A:1007763000430. URL <http://dx.doi.org/10.1023/A:1007763000430>.
- E. Hastings, R. Guha, and K. Stanley. Automatic content generation in the galactic arms race video game. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(4):245–263, Dec 2009. ISSN 1943-068X. doi: 10.1109/TCIAIG.2009.2038365.

- P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), Dec. 1996. ISSN 0360-0300. doi: 10.1145/242224.242477. URL <http://doi.acm.org/10.1145/242224.242477>.
- S. Jackson. The Generic Universal RolePlaying System (GURPS), 2014. URL <http://www.sjgames.com/gurps>. Accessed: 29-08-2014.
- J. L. Jonathan Puckey. Paper.js - The Swiss Army Knife of Vector Graphics Scripting. <http://paperjs.org/about/>. Accessed: 2014-05-26.
- N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General Game Playing: Game Description Language Specification. 2008.
- M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005. ISSN 0360-0300. doi: 10.1145/1118890.1118892. URL <http://doi.acm.org/10.1145/1118890.1118892>.
- J. Orwant. *EGGG : The extensible graphical game generator*. PhD thesis, Massachusetts Institute of Technology, 2000.
- B. Pell. A strategic metagame player for general chess-like games. *Computational Intelligence*, 12(1):177–198, 1996. ISSN 1467-8640. doi: 10.1111/j.1467-8640.1996.tb00258.x. URL <http://dx.doi.org/10.1111/j.1467-8640.1996.tb00258.x>.
- S. Seefried. js-good-parts: Javascript: The Good Parts – AST & Pretty Printer. <http://hackage.haskell.org/package/js-good-parts-0.0.3>, June 2012.
- A. M. Smith and M. Mateas. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *IEEE Conference on Computational Intelligence and Games (CIG)*, 2010.

D. Szymczak. Generating Learning Algorithms: Hidden Markov Models as a Case Study. Master's thesis, McMaster University, 2014. URL <http://hdl.handle.net/11375/14101>. Retrieved from: MacSphere (McMaster University Libraries Institutional Repository).

The Gaming Scalability Environment Project (G-Scale). URL <http://www.gscale.ca>.

W3.org. SVG 1.1: Basic Data Types and Interfaces.

<http://www.w3.org/TR/SVG11/types.html#ColorKeywords>. Accessed: 2014-05-26.

D. M. Weiss. Defining families: The commonality analysis. *Submitted to IEEE Transactions on Software Engineering*, 1997. URL <http://www.research.avayalabs.com/user/weiss/Publications.html>.

J. Whitehead. Game Genres: Shmups.

<http://classes.soe.ucsc.edu/cmpe080k/Winter07/lectures/shmups.pdf>, 2007. Accessed: 2014-05-26.