Imperial College London
Department of Computing

Co-operative Coevolution for Computational Creativity: A Case Study In Videogame Design

Michael Cook

October 2015

Supervised by Simon Colton

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

This work is my own. All other work has been referenced.

Abstract

The term *procedural content generation* (PCG) refers to writing software which can synthesise content for a game (or other media such as film) without further intervention from a designer. PCG has become a rich area of research in recent years, finding new ways to apply artificial intelligence to generate high-quality game content such as levels, weapons or puzzles for games. Such research is generally constrained to a single type of content, however, with the assumption that the remainder of the game's design will be fixed by an external designer.

Generating many aspects of a game's design simultaneously, perhaps ultimately generating the *entirety* of a game's design, using PCG is not a well-explored idea. The notion of automated game design is not well-established, and is not seen as a task distinct from simply performing lots of PCG tasks at the same time. In particular, the high-level design tasks guiding the creative direction of a game are all but completely absent in PCG literature, because it is rare that a designer wishes to hand over such responsibility to a PCG system.

We present here *ANGELINA*, an automated game designer that has developed games using a multi-faceted approach to content generation underpinned by a co-operative co-evolutionary approach which breaks down a game design into several distinct tasks, each of which controlled by an evolutionary subsystem within *ANGELINA*. We will show that this approach works well to automate game design, can be ported across many game engines and game genres, and can be enhanced and extended using novel computational creativity techniques to give the system a heightened sense of autonomy and independence.

Acknowledgements

Nine years ago I stood outside one of my first mathematics tutorials wondering if I would manage to finish my first degree year, let alone go on to do a PhD. I asked someone I hadn't met before if they could help me with a question I was struggling with, and since that day Azalea Raad's support, respect, advice, teaching and love made it possible for me to make it through the decade that followed, and ultimately submit this thesis. Thank you for always having the patience to teach me new things, and the perseverance to push through the hardest times with me. I wish I could dedicate a grand truth about humanity to you, or a groundbreaking revelation about our universe, but all I have is this text about videogames. It is not enough, but nothing ever could be. I love you.

I am indebted to the rest of my family too, for letting me love games and encouraging me to keep working with them. My brother, Andrew, has always had something new to show me about videogames even when he was a few years old watching me play games with an unplugged controller in his hands. My mother, Michele, has provided an amazing foundation ever since I was a child, and has been an inspiration throughout her life - I am proud to be attending her university graduation in the same year as I submit this thesis.

As for my father, Simon - it's fair to say that had he not sat down to fudge the wires into a ZX Spectrum as often as he did, I might not be here at all. Had he not spent those nights in his shed building and upgrading computers for me, or borrowing new bits of technology off of friends; had he not suggested that I might enjoy Computer Science; had he not done these or any other number of things, I would not have been come close to doing any of the work contained within this document. More than anyone else, this thesis is dedicated to him, because he is not able to be here to read it for himself.

I'm privileged that over the course of this PhD I was able to meet hundreds

of wonderful, interesting people. This PhD has been influenced in so many ways by developers, critics, players, designers, academics, artists and more, and without so many people pitching in with their thoughts, opinions, ideas and criticism this work would have been much less ambitious, much less interesting, and much less fun. Thank you to everyone who I met (or tweeted at) over the years and who helped make this work what it is today.

Thank you for reading this thesis. It's been a weird five years, during which time just about every assumption and opinion I've had about games and academic research has been challenged in some form. While a lot of this work now seems strange and alien to me, I think I am glad it is here, collected into some kind of narrative. For all the issues I have with the worlds of academia and games development that this work sits between, it was a privilege to be involved in an area of games research still defining itself, and all I can hope is that I contributed in some way to its development and growth.

"The popular stereotype of the researcher is that of a skeptic and a pessimist. Nothing could be further from the truth! Scientists must be optimists at heart, in order to block out the incessant chorus of those who say 'It cannot be done.' "

–Sid Meier's Alpha Centauri

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

–Alan Turing

Contents

List of Tables

List of Figures

Published Work

The work described in this thesis has also been published in the following papers and articles:

- *Multi-faceted Evolution Of Simple Arcade Games* - Michael Cook and Simon Colton - CIG 2011 - Material from this paper appears in chapter 5.
- *Initial Results From Co-operative Co-evolution For Automated Platformer Design* - Michael Cook, Simon Colton and Jeremy Gow - EvoGames 2012 - Material from this paper appears in chapter 6.
- *Aesthetic Considerations For Automated Platformer Design* - Michael Cook, Simon Colton and Alison Pease - AIIDE 2012 - Material from this paper appears in chapter 7.
- *Mechanic Miner: Reflection-Driven Game Mechanic Discovery And Level Design* - Michael Cook, Simon Colton and Azalea Raad - EvoGames 2013 - Material from this paper appears in chapter 8.
- *Nobody's A Critic: On The Evaluation Of Creative Code Generators* - Michael Cook, Simon Colton and Jeremy Gow - ICCC 2013 - Material from this paper appears in chapter 8.
- *Creativity In Code: Generating Rules For Videogames* - Michael Cook - ACM XRDS, 2013 - Material from this article appears in chapter 8.
- *From Mechanics To Meaning And Back Again: Exploring Techniques For The Contextualisation Of Code* - Michael Cook and Simon Colton - AI and Game Aesthetics Workshop, AIIDE 2013 - Material from this paper appears in chapter 8 and chapter 12.

- *Automating Game Design In Three Dimensions* - Michael Cook, Simon Colton and Jeremy Gow - AISB 2014 - Material from this paper appears in chapter 9.

- *A Rogue Dream: Automatically Generating Meaningful Content For Games* - Michael Cook and Simon Colton - Experimental AI in Games Workshop, AIIDE 2014 - Material from this paper appears in chapter 12.

- *Towards The Adaptive Generation Of Bespoke Game Content* - Cameron Browne, Simon Colton, Michael Cook, Jeremy Gow and Robin Baumgarten - Handbook of Digital Games - Material from this book chapter appears in chapters 5, 6 and 7.

- *Playable Experiences at AIIDE 2014* - Michael Cook - AIIDE 2014 - Material from this paper appears in chapter 9.

- *Creating Code Creatively - Automated Discovery Of Game Mechanics Through Code* - Michael Cook - Videogames and Creativity, 2015 - Material from this book chapter appears in chapters 5, 6 and 8.

Work from the thesis as a whole contributed to the following papers and book chapters:

- *On Acid Drops And Teardrops: Observer Issues In Computational Creativity* - Simon Colton, Michael Cook, Rose Hepworth and Alison Pease - AISB 2014.

- *Assessing Progress In Building Autonomously Creative Systems* - Simon Colton, Alison Pease, Joseph Corneli, Michael Cook and Maria Teresa Llano - ICCC 2014.

- *Towards The Automatic Generation Of Fictional Ideas For Games* - Maria Teresa Llano, Michael Cook, Christian Guckelsberger, Simon Colton, Rose Hepworth - Experimental AI in Games Workshop, AIIDE 2014.

The following two journal papers are currently under review, and summarise a large portion of the work described here.

- *The* ANGELINA *Videogame Designer, Part I* - Michael Cook, Simon Colton and Jeremy Gow - Submitted to the *Transactions on Computational Intelligence and AI in Games*.
- *The* ANGELINA *Videogame Designer, Part II* - Michael Cook, Simon Colton and Jeremy Gow - Submitted to the *Transactions on Computational Intelligence and AI in Games*.

1 Introduction

The medium of videogames is constantly changing, as the state of the art is pushed forward by both artists and technologists. In the latter case, scientific research is often motivated by the needs of modern games [15], as well as being the motivating force behind the creation of new kinds of games [136]. More than any other cultural medium, videogames are inexorably linked to what it is possible for computers to do.

The history of videogames is littered with examples of people writing software to create parts of a videogame automatically. Sometimes these endeavours had technological motivations, efficiently compressing large games inside small algorithms that would unfold as the game ran [36]. Sometimes they were motivated by artistic intentions, producing unpredictable or pseudo-infinite spaces and systems [7]. The culture of *procedural generation* within videogames is approaching ubiquity, but it is also somewhat stagnant. Players have become accustomed to the idea that simple kinds of content can be generated, but the repetitive nature of these generators and the kinds of content they generate have led to the notion of procedural generation being stereotyped as 'random placement' of basic game components [59].

The thesis primarily describes an approach to automated videogame design centred on evolutionary techniques, and the development of a piece of software called *ANGELINA*, which was built over the course of many iterations and can be considered an ongoing development even past the completion of the work described here. *ANGELINA* is situated between the fields of c*omputational creativity*, a branch of artificial intelligence concerned with building software which can perform tasks via behaviours which we call 'creative', and *procedural content generation*, a broad field but normally defined (as indeed we define it here) in terms of the videogames industry, referring to software which can produce content for videogames automatically.

The work we present here approaches procedural generation with a new

perspective, with the aim of producing software capable of generating any and all aspects of a videogame. In doing so, we shift the focus from how to generate a piece of content in isolation, to how to generate many different pieces of content that relate to one another in subtle ways. We will show that co-operative co-evolution is an appropriate technique for this, that there are ways to augment and extend this approach, and that there are many interesting philosophical and technological issues that arise when tackling this larger problem. We hope that in doing so we present an argument for *automated game design* to be recognised as a research area distinct but parallel to procedural content generation, with a unique set of problems and opportunities to uncover.

In this chapter, we first present our motivation for undertaking the work, and we discuss our position in the wider context of the games industry in particular. We then provide a brief overview of the contributions that this work has made, and an outline of the remaining chapters of the thesis.

1.1 Motivation

Videogames occupy a unique position in human history at the conflux of cultural expression and technological progress. The development of a modern blockbuster game involves hundreds of highly-skilled people, millions of dollars, and the leveraging of a huge variety of resources, from symphony orchestras [63] to cutting-edge motion capture [11]; from Hollywood actors [120] to experimental scientists[1]. Despite this huge variety of resources employed in their production, videogames are becoming a popular form of cultural and artistic expression, both on a local, personal level (as evangelised by Anna Anthropy in [4]) as well as individual work becoming financially successful and globally popular. For example, Braid [8] was designed primarily by Jonathan Blow with contract work for art and music. The game sold over half a million copies in its first few years. Gunpoint [37] was similarly made by a single developer, Tom Francis, with musicians and artists working on commission. Francis posted a blog after the game's releasing stating that Gunpoint had made its development costs back in 64 seconds, since the only cost besides the hours put in by people was the $30 cost of a game development tool three years previously [42]. While this is in

[1] Valve Software currently employs both in-house psychologists and economists.

some ways a comical exaggeration (the artists, musicians and Francis himself all had other forms of employment during this time so technically the hours spent were free time work) it underlines the small-scale, low-budget work that nevertheless is able to have huge impact in a market dominated by products costing hundreds of millions of dollars to produce.

Automating the production of artefacts in creative domains is not a new concept. It has often been attempted in media such as visual art [22], poetry or music. However, there has not been a serious attempt to automate the production of videogames. One reason for this may be their relative complexity: their composition of music, artwork, narrative and programming posing a great challenge to be tackled simultaneously. Gaming's perceived frivolity as a pastime may also contribute to a lack of attempts at automating their design. It is worth noting that the cultural media whose automation has been most often attempted by researchers are those which are held up as traditional intellectual pursuits [65]. Regardless of the reasons for it, the automation of game design is an exciting and desirable research question for a number of reasons.

For Computational Creativity research, the meeting of creative media and the complexity of artistic expression through interactive systems offers myriad exciting research questions that will both improve our understanding of creative systems and offer a challenging validity test for our models and evaluation techniques. This thesis will provide a foundational body of work for the area, and ultimately ask more questions than it answers, as research often does.

For artificial intelligence at large, the automation of a fundamentally technological task – that of programming a game – forces us to reconsider one of computer science's most daunting unsolved problems: automated code generation. This thesis will show how automated game design offers a unique platform for investigating this problem, and we will propose methods for managing the complexity of it.

For the growing and diversifying group of videogame consumers, automatic game design offers a long-term solution to a growing issue of content scarcity in modern videogames, as well as offering a chance to produce games that could not be designed today. This thesis will point to a future where artificial intelligence software can express creative ideas, and will highlight how they may ultimately be able to stand, alongside people, as autonomous

creators.

1.2 Contributions and Achievements

This thesis makes contributions to the state of the art in both computational creativity and procedural content generation, as well as accomplishing several milestones for the fields in terms of major achievements by generative software.

- We developed *ANGELINA*, the first piece of computationally creative software that tackles full-game creation. *ANGELINA* has designed numerous games in different genres throughout the course of the work described here, and its games have been played by tens of thousands of people.
- In developing *ANGELINA*, we demonstrated that the evolutionary programming technique of co-operative co-evolution (CCE) can be applied to the task of videogame design in many different ways. We show that CCE can be applied to many different game genres, to evoke specific game features, to tackle artistic and creative tasks as well as technical ones, and can produce emergent and subtle effects in what it creates.
- We developed techniques for generating game content without intermediate abstract data structures, through the direct manipulation and generation of program code. This led to systems with a strong degree of creative autonomy, and has influenced research directions in procedural content generation.
- We have presented studies of computationally creative software engaging with various groups, including the general game-playing public, in multiple scenarios. This includes the first example of a piece of software entering a public game jam alongside human game designers.

1.3 Breakdown Of The Thesis

We hypothesise that co-operative co-evolution is an appropriate technique for generating complete videogames. We present *ANGELINA* in evidence of

this hypothesis, a piece of software built over many iterations, all of which will be described in turn. We also hypothesise that an automated game designer such as *ANGELINA* can be accepted within a creative community, as an independent creator in its own right. This thesis will not provide proof of this second hypothesis, but we will lay the foundations for this hypothesis and show clear progress towards this goal.

We now step through the remaining chapters in turn and summarise their contents. Chapters 2, 3 and 4 provide a background for the work in this thesis, introducing concepts that are relevant to both of the fields which have inspired and influenced the construction of *ANGELINA* and the research surrounding it.

- Chapter 2 provides a foundational introduction to videogames, and covers selected aspects of their history and the current state of the art.
- Chapter 3 gives an overview of evolutionary computation, additionally covering coevolution and the application of evolution to art and design tasks.
- Chapter 4 considers generative software, particularly in the context of videogames, and gives an extensive background on its use in design, as well as research surrounding the generation of content for videogames.

Chapters 5, 6 and 7 describe the first three versions of *ANGELINA*, and demonstrate a repeatable structure for automated game design systems based on co-operative co-evolution.

- Chapter 5 describes $ANGELINA_1$, a system which evolved simple arcade games. This represents the first system we built which uses co-operative co-evolution for the purposes of game design.
- Chapter 6 describes $ANGELINA_2$, which took forward the ideas of $ANGELINA_1$ and applied them to a new design domain, Metroidvania games. $ANGELINA_2$ demonstrates the ability of our approach to generate games with very genre-specific features, as well as showing the flexibility of the techniques we use.
- Chapter 7 describes $ANGELINA_3$, moving the system into Computational Creativity, extending $ANGELINA_2$ to demonstrate how our

approach can be used to incorporate real-world ideas, concepts and information into a game's design.

Chapters 8, 9 and 10 extend the development of *ANGELINA* with two major versions that make contributions both to the high-level philosophy of Computational Creativity and the low-level technical cutting edge of procedural content generation.

- Chapter 8 describes $ANGELINA_4$, which uses code generation to generate game mechanics. It also shows how co-operative co-evolution can give rise to surprising results and emergent behaviour between individual evolutionary subsystems.
- Chapter 9 describes $ANGELINA_5$, bringing together many of the generative techniques explored in various earlier versions of *ANGELINA*, with a new emphasis on presenting the system as an independent, autonomous game designer with an understanding of the real world. We also present details of how $ANGELINA_5$ entered the Ludum Dare game design competition.
- Chapter 10 brings together a mix of evaluation techniques to provide an evaluation of *ANGELINA* from many different angles, providing results of studies and surveys, as well as analysis of *ANGELINA* along technical and philosophical lines.

Chapters 11 and 12 look both at what has gone before, and what remains to be done. We consider *ANGELINA* in the context of other work both in Computational Creativity and procedural content generation, and look at the questions that were opened as a result of this research, which we intend to explore next.

- Chapter 11 presents related work to *ANGELINA* across both Computational Creativity and procedural content generation. We consider other attempts to automatically generate games either in whole or part, and other software developed to be taken seriously as a creative entity.
- Chapter 12 looks at prototypes and open research questions that are still being pursued as the work outlined in this thesis draws to a close.

We examine early results and the implementation of prototypes, and theorise as to where this work might lead.

1.4 Summary

This chapter introduces the remainder of this thesis, which describes the research and development of *ANGELINA*, a piece of software capable of designing videogames autonomously. We provided a motivation for the work, which will be further strengthened by the background chapters which follow this one, situating the work as an extension of several lines of enquiry and research. We then summarised the contributions of the thesis, which we tackle in the later chapters of the thesis that provide details of *ANGELINA*'s implementation during various versions of the software. Finally, we gave an outline of the thesis, providing an overview of what else follows.

2 Background - Relevant Concepts in Videogames

2.1 Introduction

In this chapter, we introduce some important concepts related to videogames and their design. These concepts will provide background for the work we present in the main part of this thesis, as well as a basis for some of the discussion of topics such as procedural content generation in chapter 4. This is by no means a comprehensive overview of videogame history or design theory; instead we limit ourselves to the most relevant topics for this thesis.

In §2.2 we describe some common methods for categorising videogames, both according to the kinds of interactivity they offer as well as the more specific design ideas the game contains. These classifiers divide important background material as they describe established game archetypes which will influence the implementation of versions of our automated game design software. They also provide an opportunity to think about how designers innovate within established genres, by making small changes to well-known formulae. We will revisit this idea later from a computational standpoint in chapter 8.

In §2.3 we introduce perhaps the defining feature of videogames as a medium: *game mechanics*. We define mechanics, and its child concept *verbs*, giving some simple examples from games to illustrate them. This section will help us understand the underlying systems at work in the games generated by our game design system ANGELINA, and will also be particularly relevant when discussing the generation and evaluation of game mechanics in chapter 8.

In §2.4 we introduce approaches to measuring the worth of a videogame, either in terms of how fun it is, what positive effects it might have on the player, or other purposes games can fulfil. We explore some theories about

what gives games value, but also note that no single theory adequately summarises what gives games their worth. This will influence our approach to implementing game evaluation in ANGELINA, which we lay out in later chapters in this thesis.

2.2 Game Classification

Games are typically classified according to which genre they belong to, in line with other media such as film or music. The identification of artistic movements or design philosophies in game design is less common, although attempts have been made to identify them, such as Brendan Caldwell's identification of the *punk* movement in [13]. Caldwell refers to a quote by independent developer *thecatamites* who is asked what inspires him and responds:

> Punk rock! Especially the DIY spirit of it, and the sense that being amateurish didn't necessarily mean just making cut-price versions of bigger commercial stuff; it could be about trying to find different ways of looking at it altogether.

Caldwell identifies punk as a movement within independent games development, rising out of better development tools and distribution channels which enable people outside the traditional games industry to make games and make others aware of them. According to Caldwell, this '*democratised videogame development in the same way the lousy vocals and poor strumming of the late seventies democratised the notion of forming a band*'.

Genres are the preferred classification for mainstream game criticism, retailers, and general discussion, partly because of their deep rooting in the popular culture of videogames. Genre classification in media can segregate according to many different qualities. In film, for instance, genre can indicate emotional tone as well as imply an audience or theme. Videogames are subject to what we call *high-level* and *low-level* genre classification; high-level genres tend to indicate the skill-set required to complete the game, while low-level genre classifications are closer to an artistic movement, identifying common features or design ideas that players can expect from games adhering to it. Below are some common classifiers, some of which are particularly relevant to the topics we will cover in the remainder of this thesis.

2.2.1 High-Level Classification

High-level genres indicate the skill-set required to play or enjoy a particular game. Their generality can make them unhelpful classifiers, although they are still a popular way of classifying games on major review sites such as Metacritic[1] or game stores such as Steam[2]. Some examples of high-level genres are:

- **Action Games** - Games that typically rely on reflexes, accuracy or dexterity. Many older games that were not electronic versions of boardgames fell into this genre, such as *Spacewar!* [105] and *Pong* [62]. This genre is now extremely broad, as any game that requires real-time control of an object or group of objects can be described as an Action Game.

- **Adventure Games** - Sometimes called *Point-and-Click Adventures* to distinguish them from more action-oriented titles, adventure games are story-driven puzzle games which try to connect combinatorial problem solving to real-world logic. The *Monkey Island* [82] series is a famous example, in which narrative obstacles are overcome using items and information found throughout the world, while *The Walking Dead* [50] provides a more contemporary example of the genre. This genre declined in popularity from the late nineties onwards, but has experienced a revival in recent years.

- **Role-Playing Games** - Games in which the player is encouraged to take on a character within a fictional universe, a feature inherited from tabletop role-playing games such as *Dungeons and Dragons* [57]. Modern usage of the term more often refers to statistics-driven game systems where player strength is a function of various numbers describing their abilities, personality or physical characteristics. These systems were often features of games such as Dungeons and Dragons, and have taken greater prominence in the transition to computer games.

- **Platform Games** - While technically a subclass of action games, platformer is such a broad classification that it is easy to consider

[1] http://www.metacritic.com
[2] http://store.steampowered.com/

it a high-level classifier. Primarily focused on the traversal of 2D or 3D worlds, platformers sometimes include combat, puzzles, or item collection as game features or goals.

2.2.2 Low-Level Classification

The multitude of low-level classifications that exist for games make it impossible to comprehensively cover them here. Instead, we describe those that act as context for the work outlined in this thesis, either as domains that ANGELINA works within, or domains with relevance to further research questions or philosophical discussions.

Puzzle Platformer

In the past decade, the platformer genre experienced a surge in popularity as the astronomic rise of independent and amateur game development largely focused on the development of platform games. This is thanks in part to the development of several game libraries geared towards the production of platformers, such as *Flixel* and *FlashPunk* as well as the proliferation of the genre during the late 1980s/early 1990s – children who had grown up playing platformers were now developing games themselves [1]. The saturation of the genre encouraged experimentation, and puzzle platformer-style games emerged as a common subgenre as a result.

The primary components of platform games - moving and jumping - became commonplace in the time since their emergence in the 1980s and are now considered part of a general gaming skillset on which new concepts can be built. Therefore, while early platformers such as *Donkey Kong* [88] focused on dexterity and jumping problems, modern platformers assume these skills inherently in the player and focus more on learning new skills and using them in conjunction with the basic platforming skills to solve problems. Puzzle platformers are games which rely on the player's ability to use skills to solve problems in a physical space, often applying concepts that are not found in other games, thereby demanding learning and experimentation.

Puzzle platformers often introduce new rules or concepts that explicitly build upon existing ones as the game progresses. In *Braid* [8] the player is initially given the ability to rewind time indefinitely. This is used to undo mistakes which cause the player's death, such as mistiming jumps (a skill

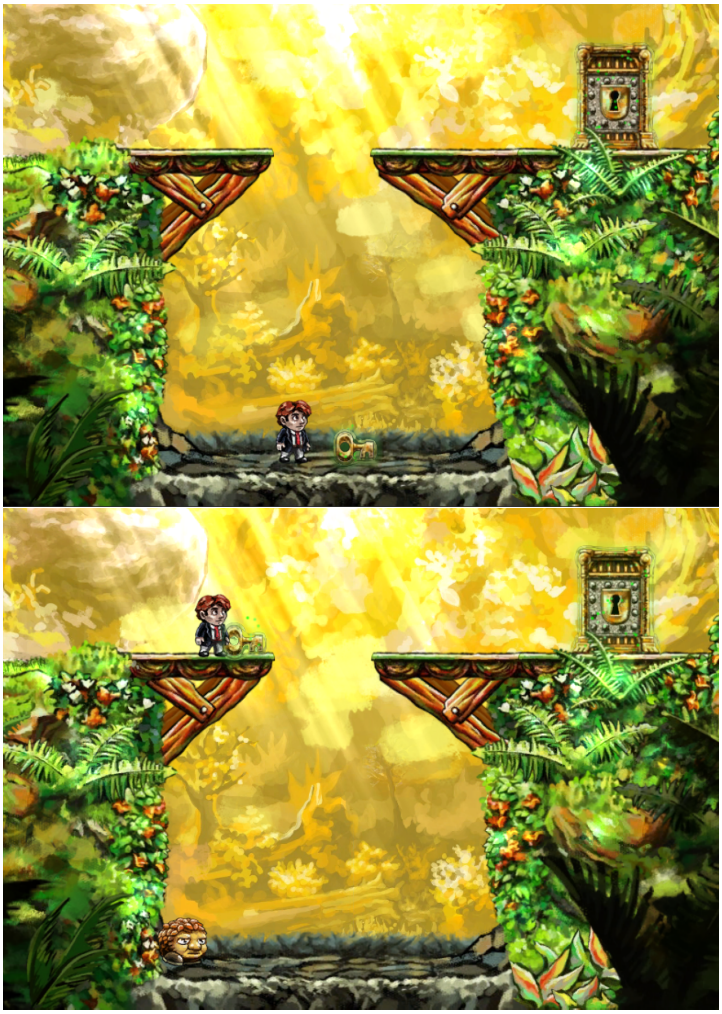


Figure 2.1: A simple puzzle from *Braid* [8]. The player drops down to pick up the key, and then rewinds time backwards. The key, unaffected by the rewind, follows the player back.

inherent to the platform genre). Later, the player is introduced to objects that are not affected by the reversal of time. This allows them to solve new puzzles by making changes to the object, and then rewinding the rest of the world to its original state. Figure 2.1 shows an example puzzle. The player must retrieve a key from the bottom of a pit. The key is unaffected by rewinding time. Therefore, by falling into the pit and grabbing the key (first image) the player can rewind themselves back out of the pit while still holding the key (second image).

Metroidvania

Metroidvania games are a subclass of platform games that put an emphasis on exploration and item acquisition. The term is a portmanteau of two game names, *Metroid* [68] and *Castlevania* [74], popularised by games journalist Jeremy Parish. Neither Metroid nor Castlevania are considered the first examples of such games, but they are landmark examples that defined the subgenre.

Metroidvania games typically situate the player within a large game world, of which only a small percentage is initially accessible to the player. Figure 2.2 shows the size of the original Metroid game. The initial area accessible to the player (that is, without obtaining items or bonuses of any kind) accounts for less than 2% of the entire game world. By acquiring items, the player can expand this accessible area, and find further items. Combat is often a central feature in these games, as the items that expand the player's accessibility often also augment their ability to fight. Fighting is necessary to progress through the game, although combat is largely skill based and the collected items typically make combat easier rather than being a requirement.

Some items exhibit basic lock/key mechanics, where physical obstructions to progress (such as a locked door) are removed automatically when a particular item (such as a key) are acquired. More interesting items or abilities have indirect effects that the player must experiment with to fully utilise, however. For instance, in *Metroid* the player can find the Ice Beam item. This allows the player's shots to freeze an enemy in place, which in turn allows the player to use it as a platform to stand on and gain additional height or reach new areas, as seen in figure 2.3. This forces the player to reconsider environments she has already passed through because the new item may be applied to them to open up new areas. As a result, subtlety and variation in items is celebrated in the genre.

Roguelikes

Roguelikes are a subclass of role-playing games, focusing on statistics-driven turn-based combat, with less of an emphasis on story (although many modern roguelikes have narrative elements). The term *roguelike* is derived from *Rogue* [127], a hugely influential game which defined the subgenre. While

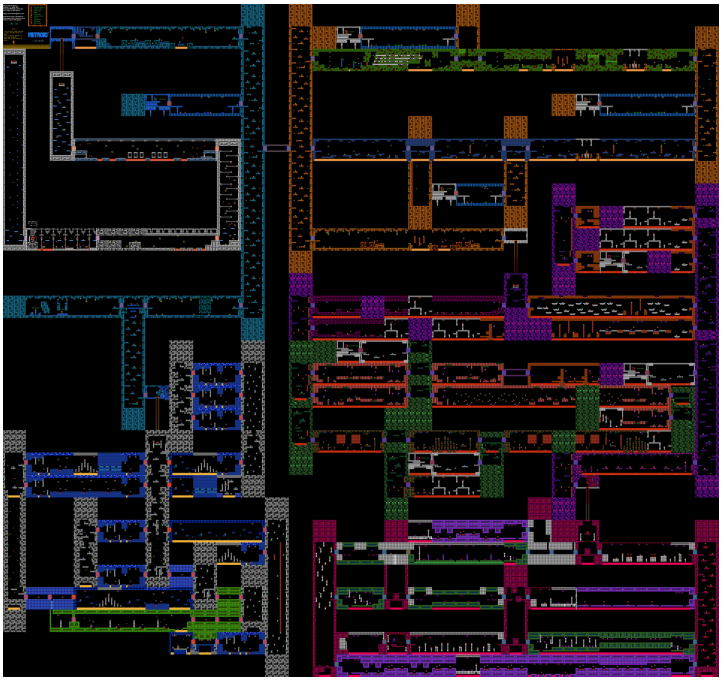


Figure 2.2: A composite image showing the entirety of the game world in *Metroid*. Image from [61].

many conflicting descriptions of roguelikes exist, modern games describing themselves as part of the genre tend to select from a list of features, including:

- Turn-based gameplay, meaning the player is giving indefinite time to think before taking an action.
- Procedurally-generated content, particularly level generation.
- Permadeath, meaning that games cannot be saved and loaded to attempt a situation differently.
- An emphasis on understanding the game through exploration and experimentation, rather than being taught explicitly how each game system works.

Many of these features (such as permadeath and obscured game rules) may lower some players' quality of experience, modern roguelikes have altered some of these features in order to present different or more approachable experiences. For instance, in *Dungeons of Dredmor* [46], permanent

Figure 2.3: A screenshot from *Metroid*, demonstrating how the Ice Beam can augment player accessibility. The player-character is standing on a frozen enemy in order to access a gap in the wall on the left.

death (or *permadeath*) is optional, allowing players to save and load if they wish. In *FTL: Faster Than Light* [48] there are no game rules or concepts that are obscured from the player, for example. Nevertheless, obscurity and difficulty is still prized by some parts of the roguelike community.

2.3 Verbs and Game Mechanics

Anna Anthropy describes a game in her book *Rise of the Videogame Zinesters* as '*an experience created by rules*' [4]. When we talk about rules in games

Figure 2.4: A simple game mechanic from *Spelunky*.

from a design perspective, we often discuss *game mechanics*. The term game mechanic has many different (and conflicting) definitions in game criticism and design. When used throughout this thesis, we will use it to mean the following: a game mechanic is a system of rules which affects entities in the game world. Figure 2.4 shows a simple game mechanic from *Spelunky* [47]. Spiders cling to the ceiling until a player passes underneath them, at which point they drop down onto the player. This is one small set of rules that makes up part of the game's systems of rules as a whole, which affect the whole game experience.

A *verb* is a special type of game mechanic which is initiated by a player action. This could mean an explicit game interaction, such as pressing a button to cause a particular system in a game to be altered in some way. It can also refer to indirect causation, where the player interacts with another system through a series of button presses to bring about a particular game state.

As an example, in *Super Mario*, pressing the A button on the controller causes the player character to jump up in the air. Jumping is a very basic game mechanic, and integral to many types of game, particularly Platformer games (§2.2.1). Jumping is a verb: pressing a button instantly causes the desired change in game state, propelling the player upwards. In the same game, if the player character falls on top of certain enemy types, that enemy will be damaged and change state in some way, often dying and being removed from the game entirely. By positioning the player character above another game object, the player is bringing about a subsequent game state, where the character lands on top of the object. This mechanic allows the player to clear their path of obstacles and enemies.

Some game mechanics are fundamental to all game types, while others are highly specialised and only apply in very specific games. Very common or important game mechanics such as jumping often become part of the shared vocabulary of videogames, and become second nature to people who play games regularly. Other game mechanics may be commonplace to players of a particular type of game. In *first-person shooter* (FPS) games, a subset of the action genre (§2.2.1), the act of moving left and right while looking forward is called *strafing*. Strafing is an important ability in such games, as it allows the player to look in an important direction while moving orthogonally to it. While strafing is uncommon in games with less action or quick decision-

making, it is unusual for a modern FPS game to omit it.

Some game mechanics are designed for specific games, to extend a game's ruleset in a particular way. They may be designed by altering existing common mechanics in some way, or they may be completely novel. For instance, Valve Software's *Portal* relied on the titular portals as a core gameplay mechanic. They allowed the player to designate two flat surfaces in the game world such as walls or floors, which were then connected in space by opening a 'portal' on each surface through which the player, or any other object, could pass. This game mechanic has appeared in very few games since then, but was explored in great depth by the series.

2.3.1 Emotion and Meaning

Recall the quote from Anna Anthropy's *Rise Of The Videogame Zinesters* which we began this section with: '*a game is an experience created by rules*'. Rules, systems, mechanics and verbs are what set videogames apart from other forms of media. They allow their players to interact with the game and see the results - this allows players to explore and understand these systems, and to infer things from how rules interact with one another.

Games can convey meaning and evoke emotion through the same techniques used by other media like film and music. Horror games like *Silent Hill 2* [108] use cinematographic techniques similar to those used in horror films to make the player feel uneasy, such as skewed or unusual camera angles, or reverse camera angles where the player can't see what the character is about to walk into. Figure 2.6 shows an example of this. However, games also have the unique ability to convey meaning through their systems and rules, and Anthropy's definition of games in these terms highlights how this is a defining feature of the medium.

As we discussed earlier with respect to puzzle platformers, some mechanics and verbs are fundamental to games and generally accepted as being universal across certain genres. To give a very simple example, in two-dimensional platform games, the game is viewed from the side, meaning that objects fall downwards on the screen to simulate gravity. The player knows that if they jump upwards, they will eventually fall down again. This doesn't communicate a complex emotion or message to the player, but is nevertheless important because it establishes a connection between a game system (objects falling down on the screen) and a real-world concept (grav-

Figure 2.5: A screenshot from *Silent Hill 2*, showing a reverse camera angle that prevents the player from seeing what the character sees.

ity and physics).

Games can communicate non-literal meaning through systems, too. Alan Hazelden's *By Your Side* is a good example of a simple system of rules which conveys a simple artistic message while still retaining the form of a game. Initially the player appears to control two characters, a man and a woman, but in fact the player is controlling the man, and the woman mimics the man's movement if there is no obstacle blocking her path. This can be used to move one character without moving the other, and navigate the maze to get both characters to the exits. Later in the game, the woman starts performing the opposite move to the one the player enters for the man. The change in the relationship between the two game entities also conveys a change in the relationship between their analogues in real-life, and asks the player to think about what the the system of mimicking or opposing actions might represent.

2.4 Value

In order to design games, automatically or otherwise, we require some understanding of why people play games and what qualities good games have. In academic circles this discussion is often prone to circular arguments and

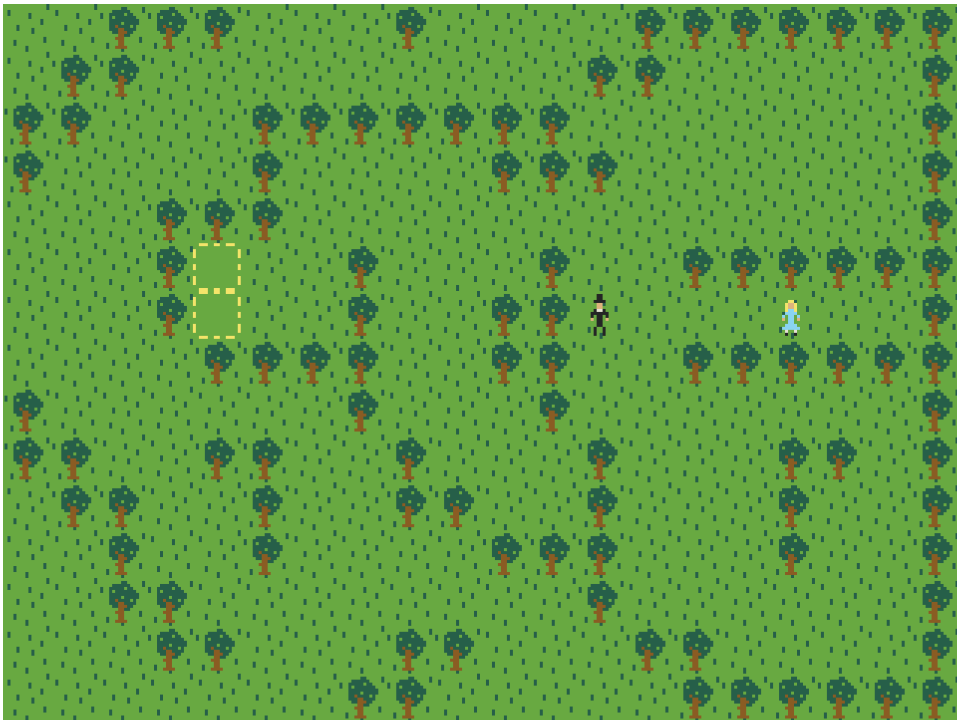


Figure 2.6: A screenshot from *By Your Side*.

conflicting definitions of overloaded terms. We therefore aim only to give a high-level overview of some leading theories regarding the worth of games. We discuss each evaluation criterion used within ANGELINA, and the motivation for it, in subsequent chapters with reference to this section.

2.4.1 Fun

Koster and Learning as Fun

Raph Koster is an American game designer and theorist whose game credits include the massively-multiplayer online games *Star Wars: Galaxies* and *Ultima Online*. In 2004 he published an influential book on games entitled *A Theory Of Fun For Game Design* [75]. It has received over 650 academic citations to date and is frequently referenced in industry talks and criticial discussions online. In a ten-year anniversary of the book, Koster states the following:

> Fun in games arises out of mastery. It arises out of comprehen-

> sion. It is the act of solving puzzles that make games fun. With games, learning is the drug. [75]

Koster's assertion is that learning is what provides the core 'fun' feeling players experience when playing games. He distinguishes this from other experiences that may be conflated with fun, claiming they are in fact unrelated or outside of his definition of fun. 'Comfort', 'Meditation', 'Story' and 'Practice' are all concepts he excludes from fun, while noting that they are 'perfectly valid non-fun reasons to play games'.

The argument can be criticised somewhat for oversimplifying the definition of 'fun', yet it seems that learning is certainly *one of* the reasons that people play games, if not the only one. *Masocore* games, which we discuss in chapter 4, and *roguelike* games, which we have already discussed, put great focus on obscure game systems that must be mastered through repeated exposure and experimentation. A guide for the roguelike game *Angband* states that:

> Part of the fun of playing Angband is NOT knowing in advance what an item, a monster or a spell will do... You get only one chance to play the game *without* full knowledge, but you'll have many chances to play the game *with* full knowledge.

This is not the only kind of learning that players experience through games, as Koster points out. Many games exist to encourage players to understand them through the careful introduction of concepts gradually. Figure 2.1 shows an example of gradual concept introduction in the context of puzzle platformers, for instance. While we may not accept Koster's theory of fun as an all-encompassing one, learning is clearly one component of a game's enjoyability.

Critiques of Fun and Alternative Evaluation

Journalist, game designer and author of *This Gaming Life*, Jim Rossignol, says of fun:

> Games can be evocative, entertaining... I was playing Stalker when it first [launched]... I was crouched behind a pipe, there

> were monsters nearby, and I'd run out of ammo, and it was absolutely terrifying... quite horrific. Is that fun? Is that summed up by a three-letter word? ... I think it's a lazy word. [104]

Fun is still an oft-used word to describe games that are considered enjoyable or good in some way, and players often argue that such negative experiences as described by Rossignol above are still fun in the sense that they enjoy experiencing sensations of terror or fear within a simulation. However, it is worth bearing in mind that videogames have evolved past their origins as pure sources of entertainment.

Koster's theory has been used in academic research relating to games. In [125], the authors use machine learning as a means to evaluate videogame rulesets, aiming to provide a ruleset which is neither too difficult nor too easy to learn and adapt to. The authors explicitly cite Koster as the inspiration for this approach. Other researchers, however, are concerned about explicitly attempting to evaluate fun when generating parts of videogames. For instance, Gillian Smith writes:

> What designer is ever going to ask for a level that scores 0.742 on the "fun" scale? What does it mean for a level I create to be 0.2183 fun and a level that you create to be 0.5312 fun? How is this meaningful feedback? Didn't we both just fail? Do we always want 100% "fun" content?

Smith's objection is not to Koster's theory specifically, but to the use of quantitative measures of fun in the evaluation of games research. What makes a game 'fun' is not widely agreed upon in games criticism, but even if it were it seems unlikely that we could simply reduce the notion to a real number. Like Rossignol, Smith also questions whether 'fun' should be the foundational concept by which we judge videogame experiences. One might liken it to evaluating films on the basis of how moving they are. Many films are moving, many intend to be moving, and film is an excellent medium for evoking such emotions in a person. But it cannot be said to be the primary motivation for the medium as a whole, and directors may have many other goals when creating a film.

2.4.2 Self-Improvement

McGonigal and Productivity

Jane McGonigal is an American author and game designer who specialises in designing games with social or personal impact, with game design credits on projects run by organisations like the New York Public Library and the American Heart Institute. In 2011 she published *Reality Is Broken* [85], a book whose core message was that the benefits of games are derived from the social and psychological effects they have on the player, and that these benefits could be leveraged to improve society and individuals. McGonigal states that:

> Games are showing us exactly what we want out of life: more satisfying work, better hope of success, stronger social connectivity, and the chance to be a part of something bigger than ourselves. [85]

Many of the ideas espoused by McGonigal relate to multiplayer games and the interaction between people, but some points affect all kinds of gaming, particularly statements she makes in reference to productivity. In a TED talk entitled *Gaming can make a better world*, she says (emphasis added):

> You know there's a reason why the average World of Warcraft gamer plays for 22 hours a week, kind of a half-time job. It's because we know, when we're playing a game, that we're actually happier working hard than we are relaxing, or hanging out. **We know that we are optimized, as human beings, to do hard meaningful work.** And gamers are willing to work hard all the time, if they're given the right work. [84]

The idea of humans being happier when working meaningfully can be seen in the appeal of many abstract games that do not have any direct learning involved in them, such as *Bejewelled*, a puzzle game in which players swap neighbouring tiles on a grid. When three tiles of the same type form in a row or column, those tiles disappear and new ones descend from above. There are few rules besides this in the game, yet players will happily play for many hours. *Bejewelled 2* included an *endless* mode in which the player can never lose, allowing the game to be played indefinitely for no other reason

than to interact with the game's match-based puzzling. McGonigal calls this 'blissful productivity' [85].

Similarly, many games balance repetitive content (sometimes referred to as *grind*) with tangible progress. The ratio of progress to activity can be seen to be closely linked with enjoyment for many types of game, particularly role-playing games where a sensation of growth and gradual increases in strength is integral to the game's core mechanics.

2.4.3 Other Assessments of Games

Games as Escapism

Games are often described as a form of *escapism*, whereby an individual immerses themselves in an alternative reality to avoid boredom, pain or other discomfort. Jim Rossignol, who we referenced earlier in the context of games as fun, says that escapism may be a fundamental part of who we are as human beings. In an article on Warren Ellis' blog [103], Rossignol quotes philosopher Yi-Fu Tuan and summarises his argument thus:

> What Tuan suggests is that we change the world around us because we are able to imagine things as they are not, as they could be, or even as they may never be... Farming, then, is an escape from the harsh nomadic existence of the hunter-gatherer. Cooking is an escape from the ugly acts of evisceration of animals, and the unpleasant nature of many individual ingredients. [103]

Rossignol identifies games as one of the purest examples of escapism, possibly because of their highly structured nature.

> Imagination is stimulated, exercised, by structuring itself against something external. Modelling the world and seeing how it could be. How it could be better, how it could be worse, how it could be **controlled**. (Rossignol's emphasis) [103]

While most media is passive, games allow the player the freedom to influence its systems through verbs, and achieve goals. Yet there remains structure, objectives, direction, so that the player is not left aimless. Many games are valuable precisely because they allow the player to immerse themselves within their fictional worlds and systems of rules.

Games as Self-Expression

Games are designed with varying degrees of player freedom. There are no aspects of gameplay that are guaranteed to be controlled by the player in every game. The term *on-rails* is used to describe games where the player's movement is automated, for instance. *Visual novels* often have entirely linear narratives that can only unfurl in a single way. The ways in which the player is afforded freedom in a game, then, are often very important as they are the only ways in which the player can differ their experience from that of another player.

Individualisation and self-expression through games has become more and more important as online multiplayer and social gaming communities have grown in popularity. The illusion of the player as the lone hero in the world, capable of amazing feats, is no longer sustainable when there are millions of others with an identical status. As a result, many multiplayer (often competitive) games promote themselves on the basis that players can develop a unique playing style that separates them from other players. Online games often feature cosmetic shops where players can purchase items which alter their appearance in-game. Often these cosmetic changes confer no gameplay advantage, yet are still hugely popular. Valve Software operates cosmetic stores in games such as *DOTA 2* [116] and *Team Fortress 2* [114]. In 2013, Valve paid $10.3m in *commissions* alone to the makers of cosmetic items on sale in these stores. Team Fortress 2 is affectionately referred to as 'America's #1 War-Themed Hat Simulator' by Valve Software employees and fans [113].

Self-expression can also be found in games in a more traditional sense. Minecraft [99] is a three-dimensional exploration and crafting game where players can break up the environment around them to gain resources which can then be recombined into other objects. The game's vast community interact with one another frequently through streaming services such as *Twitch.tv*[3] and *YouTube*, and one common type of content shared between players is video recordings of large construction projects, complex machinery or simply feats of endurance. The game has a relatively low barrier to entry, which has made the game very popular with children. As a result, Minecraft has been used in education[4] with one of the educational benefits

[3] http://www.twitch.tv
[4] http://minecraftedu.com/page/

being that it encourages children to act creatively and express themselves[5]. Anecdotal evidence suggests that such activity may benefit children with certain disabilities.[6]

2.5 Summary

In this chapter, we discussed some key background information about videogames as a creative medium. We showed how games can be classified according to different criteria, and gave examples of what kinds of classifications are in common use. We will return to these classifications in later chapters as we introduce our work in automated game design. We also defined *game mechanics* as fundamental building blocks of game systems, and gave details about how specialised game mechanics called *verbs* allow the player to interact with a game and influence these systems for its own ends. Verbs and mechanics are one of the focal points of the work we present in this thesis, and chapter 8 in particular will reference this background knowledge as we introduce work on generating and evaluating mechanics automatically. Finally, we considered ways in which different critics and communities ascribe values to videogames, from fun to the improvement of society. While all of these theories have weaknesses, they give us insight into why people play games. This influences not only the work we propose in the thesis, but will also become relevant in evaluatory chapters reflecting on the role of automated game design in the future of the games industry.

[5] http://www.edutopia.org/made-with-play-game-based-learning-minecraft-video
[6] http://www.autcraft.com/

3 Background - Evolution

3.1 Introduction

In this chapter, we give some background on the idea of *evolutionary computation*, a technique for solving combinatorial optimisation problems that is commonly applied to projects throughout computer science. This chapter will introduce techniques which are fundamental to the basic operations of the systems which we present in later chapters.

In §3.2, we introduce evolutionary computation and give some background on the origins of its terminology. We then describe a basic evolutionary algorithm at a high level, and step through that description in turn, discussing how evolutionary systems are initialised, executed, and how they terminate. This description introduces much terminology that will be important in subsequent chapters, which discuss multi-part evolutionary systems at length.

In §3.3 we describe *coevolution* as a specalised kind of evolutionary computation. Co-evolution is the core technique used by the ANGELINA system. We discuss its strengths and applications, and how it differs from standard evolutionary computation. We also differentiate between *competitive* and *co-operative* coevolution. The work described in this thesis primarily uses the latter, however we introduce both here for the sake of completeness and to highlight the difference that defines the co-operative approach.

In §3.4 we look at how evolutionary computation has been applied to creative domains, in particular the production of artworks. We look at the origins of evolutionary computation in art, and how it was extended from focusing primarily on evolution to focusing primarily on artistic expression. We look at different kinds of evolutionary art, and also the task of evaluating artworks for fitness. The issues raised of evaluation in a creative domain, will be directly applicable to the problems we encounter when evaluating evolved videogames created by ANGELINA.

Finally, in §3.5 we describe a simple example system which evolves basic two-dimensional mazes. This example uses much of the terminology and concepts described in this chapter, and shows some problems and variations that can occur when building an evolutionary system. Later in this thesis we will be describing much more complex, multi-population co-evolutionary systems, and this example serves as a simple introduction to how these systems and their execution will be described.

3.2 Evolutionary Computation

Evolutionary computation is a term describing a class of algorithms for solving optimisation problems. It derives its name, as well as much of its terminology, from the process of organic evolution which inspired its design. Evolutionary algorithms maintain a population of solutions to a particular problem, which are evaluated for fitness of purpose, and selectively recombined to produce a new population. This process is iterated upon until a solution of sufficient fitness is found, or a minimum number of iterations has passed. Listing 3.1 shows pseudocode for a generic evolutionary system. In this section we will step through this algorithm and examine each line in turn, giving an overview of how each line is implemented in evolutionary systems.

```
Initialise a population
Evaluate the population
Do:
        Select fittest from population
        Produce new generation from fittest
        Evaluate the new generation
        While not terminating
```

Figure 3.1: Pseudocode for an evolutionary system.

3.2.1 Initialising Evolutionary Algorithms

Evolutionary algorithms explore a search space of solutions, S, to a particular problem. Solutions tend to have two representations - a *genotype* and a *phenotype* representation. In biology, the *genotype* is an encoding of the

information required to produce a particular biological organism, through DNA. A *phenotype*, meanwhile, is the physical expression of the organism encoded by a genotype. Evolutionary computation mirrors this relationship somewhat – a genotype refers to the low-level representation used to encode information about a particular solution in a population, while a phenotype is the 'uncompressed' realisation of the genotype, in its final form.

Evolutionary algorithms start with a population, P, of solutions drawn from the search space S. This population is typically generated by randomly assigning values to each genotype's data. The population then goes through an iterative process of selection and recombination. The representation of a genotype is an important decision made by the system's designer, as it defines which aspects of a phenotype can be changed through evolution, and the ways in which they can be changed. Simple data structures such as lists and trees are often used – we describe an example system later in this chapter which uses two-dimensional arrays to represent genotypes.

3.2.2 Evaluation & Selection

Evaluation is the process by which an evolutionary algorithm orders a population, P, according to some *objective function* F. Typically, F is a function which maps $p \in P \longrightarrow [0, 1]$, although all that is really required of F is that it expresses an ordering on the search space S. The evaluation of $p \in P$ under F is called p's *fitness*.

Once P has been sorted using F, some kind of *selection* criteria is applied to choose which members of P are recombined to produce the next generation's population, P'. A simple selection method might take the solutions in order of fitness, from highest to lowest, recombining pairs in order until a new population of sufficient size is created. If a pair of solutions from the population are recombined to produce ten new solutions, for example, then the top twenty percent of P are used to create P'. A wide variety of selection criteria have been used in many different types of evolutionary system such as roulette selection, where fitness represents the chance that a solution will get chosen, or universal sampling, where individuals are selected proportional to their overall fitness.

In the analogy with biological evolution, the application of a fitness function is related to the notion of *survival of the fittest*, hence the terminology used. In nature, the 'fitness' of an organism is directly related to its ability

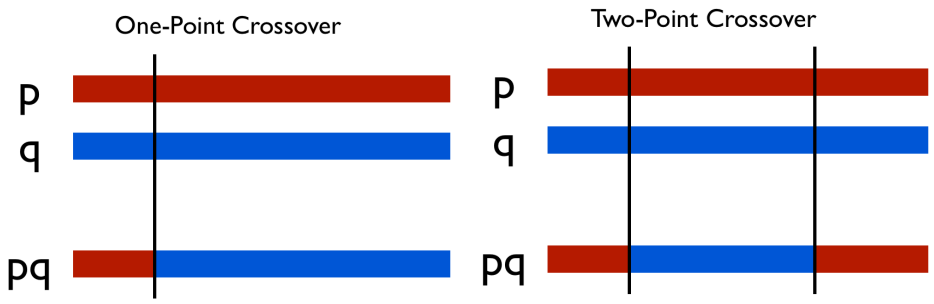


Figure 3.2: Diagram of two crossover techniques. In both cases, black lines denote crossover points where inheritance switches between parents p and q. pq is the resulting child solution.

to procreate, so the process of selection and fitness evaluation are closely related. Some types of selection criteria in evolutionary computation can be less directly related to fitness, however, such as the probabilistic roulette sampling mentioned above.

3.2.3 Recombination

After selecting a portion of the population P using a selection criterion and the fitness function F, the selected members of P must be *recombined* to create a new population P'. This is typically done through two processes: mutation and crossover.

Crossover takes a pair of solutions (p, q), or *parents*, from P and creates a new solution, or *child*, pq by combining attributes from both p and q. This can be done in any number of ways depending on the structure of P's genotype. If the genotype contains individual pieces of data, values can be inherited directly from a single parent, chosen probabilistically, or linearly combined using averages of both parents. For larger data structures such as lists or trees, more involved procedures are normally used instead of direct inheritance. Figure 3.2 shows two common crossover techniques, one- and two-point crossover. In this case, crossover points are chosen at random. A child inherits the properties of one parent until it reaches a crossover point, at which it switches and inherits from the second parent.

Mutation takes a single solution p from P and creates a child solution simply by varying the properties of p randomly. This does not require processes like one-point crossover as there is only one parent to inherit

from. Mutation can also be used after crossover has taken place to further differentiate the child from either of its parents. Typically, a *mutation rate* defines the probability that a given population member will be mutated (or the probability that a gene within a population member will be mutated). Normally this value is quite small - if too many of the new solutions are mutated, the population can become too divergent. Divergence is a problem because evolutionary systems are designed to converge through repeated iteration and consistently picking the highest-fitness individuals through selection. Mutation randomly alters an individual after the selection and recombination process, meaning that it can easily make the individual less fit. If this happens to too many individuals then the overall fitness of the system doesn't improve. A small amount of mutation, however, can be important in exploring areas of the state space that simple crossover would not normally consider.

A new population is formed using the children generated by the recombination step. However, it may also contain other individuals as well, depending on the design of the system. For example, *steady state* evolutionary systems allow the parents from the previous generation to be included as individuals in the following population along with their children. Many evolutionary systems also include randomly generated individuals in new populations, to improve the diversity of the population and stop the population from stagnating around a local maxima.

In search, a *local* maxima is a solution which is much higher quality than the solutions it is adjacent to in the search space, but is lower quality than other maxima further away in the search space. This can cause problems for search techniques like computational evolution, as they gravitate towards areas of high fitness and avoid moving towards lower fitness solutions. If a local maxima is isolated from other maxima, a system may settle on it and not search further away for potentially better solutions. Mutation is one way computational evolution tries to deal with this problem, by randomly enforcing variation on solutions, and including new randomly generated individuals also helps the system search further afield which can help find new maxima in the case of stagnation.

3.2.4 Termination

An evolutionary system terminates when certain conditions are met, specific to the individual system. Common termination conditions are when a certain minimum fitness is met, or a set number of generations has passed. Evolutionary systems may also terminate based on properties of the population as a whole, such as the average difference between each individual in the population (being an indication of how stable the population currently is). When a system terminates, the most fit individual in the current population is normally selected as the 'output' of the system. A system may also sample from its population for multiple outputs.

3.3 Coevolution

In nature, coevolution describes a reciprocal relationship between two independent species where a change in one species causes a related change in the other species. This can form mutually beneficial relationships, such as the relationship between plants and the animals they rely on for pollination; it can also result in competitive relationships, so-called 'evolutionary arms races' where predatory creatures evolve characteristics in response to an evolutionary change in their prey, and vice versa.

In co-evolutionary computation, the evaluation of members of a population is a mixture of both *subjective* and *objective* fitness criteria. Objective criteria are the kind of fitness evaluations found in ordinary evolutionary systems, and the fitness scores they return are independent of any other part of the evolutionary system. Subjective criteria, however, evaluate an individual according to its relationship with the rest of the population, or members of other populations (often called *species*) in the case of a multi-population system.

Highly fit individuals in a co-evolutionary system not only exhibit objectively fit qualities but are also adapted to other individuals from either its own population or specifically selected competing populations. This means that two runs of the same co-evolutionary system can produce vastly different outputs, because the subjective criteria causes individuals to evolve characteristics relative to the current population, which is initialised very differently between two runs of a system because of the randomness that is

inherent in evolutionary computation.

3.3.1 Competition and Co-operation

To further the biological metaphor that runs throughout this chapter, coevolution in nature can be classified as 'co-operative' or 'competitive' depending on whether organisms are mutually benefiting from the coevolution (such as animals helping plants pollinate, and benefiting from a source of food) or not (as in the predator/prey 'arms race' example). Co-evolutionary approaches to evolutionary computation can also be classified in such a manner.

Some co-evolutionary systems are designed to encourage competition between individuals or populations. In these systems, the fitness of an individual is either mostly or entirely based on its subjective fitness relative to other individuals. The individuals are therefore in competition with one another: if one individual's fitness increases, the fitness of the rest of the population decreases, because they are performing less well in relation to the increasingly fit individual. The term 'arms race' is often used to describe this relationship. Competitive coevolution is particularly effective at producing solutions to problems involving multiple parties or antagonistic relationships between solution spaces. For an example of the former, [109] describes a system which co-evolves controllers for 3D creatures, whose fitness was evaluated by whether they could catch a ball faster than their opponent. For an example of antagonistic elements of a solution space, [131] gives the example of a system which evolves a sorting network for data, and simultaneously evolves harder input data sets for it. As the quality of the sorting networks improves, the data sets compete to be harder for the network to handle. This competition drives up the quality of the resulting sorting networks.

Co-operative co-evolutionary systems work differently. Instead of competing between individuals in a population, co-operative co-evolutionary systems maintain multiple populations whose individuals are evaluated according to how well they co-operate with individuals from other populations. This means that a co-operative co-evolutionary system represents the decomposition of a large problem into several smaller problems, which are solved individually. In order to evaluate the fitness of an individual from one of the subpopulations, a member of each population is combined together, to form a solution to the original problem. The quality of this synthesised

solution represents how well each individual co-operates with the others. In [51] the authors describe a co-operative co-evolutionary system which evolves neutral network designs by passing off the task of designing subnetworks to different populations. Often co-operative co-evolutionary systems perform this problem decomposition step automatically as a feature of the system.

3.3.2 Applications for Co-evolution

In [131] Wiegand cites three motivations for the use of coevolution: infinite or extremely large solution spaces, lack of objective measures, and highly complex structures.

Large Solution Spaces

One of the strengths of evolutionary computation is its ability to find optimal solutions across very large solution spaces, by encouraging random iterations on existing solutions, and maintaining diversity in its population. However, extremely large or near-infinite solution spaces can exist which are difficult for standard EC approaches to deal with. This often happens when EC is applied to the product of two solution spaces, the result of which is a space far larger than either of the originals. The task is no longer simply finding optimal solutions in two spaces, but finding optimal solutions that compliment each other in two spaces simultaneously.

Rather than aim to find globally optimal individuals, coevolution can identify subsets of the solution space and then search for optimal individuals within this smaller space. This is made possible by having different species of a co-evolutionary system represent the original solution spaces which made up the problem. The co-evolutionary system is then tasked with finding intersections of these two spaces where interesting local optima exist, which it does through the application of its subjective fitness criteria to compare the interactions of different species.

In other words, suppose we have two search spaces, P and Q, and the space created by the product of the two, PQ. PQ is too hard to search with a standard evolutionary system looking for global optimisation, because it is much larger than P or Q alone. Multi-population co-evolutionary systems can represent P and Q as separate populations which co-evolve by evaluating

individuals $p \in P$ and $q \in Q$ against each other. In this way, the system identifies fit individuals in the space PQ by separately identifying which areas of P and Q are suited to each other (according to its subjective fitness criteria).

Lack of Objective Measures

Evolutionary computation is often applied to global optimisation problems where clear objective criteria for fitness exist. Even if a globally optimal result may be hard to find in many cases, it is nevertheless clear when one solution is better than another. However, many problems do not have clear objective measures of quality, and single-objective single-population EC approaches may struggled in these cases. [131] gives an example of such a situation: evolving strategies for games, where potential strategies may beat each other in an intransitive fashion. In such a situation a population cannot be well-ordered under any single-objective fitness function, and individuals cannot be easily separated from one another.

In [131] Wiegand posits that co-operative co-evolutionary systems naturally tend towards finding *Nash equilibria*. A Nash equilibrium is a game theory term describing a solution to a multiple-player game which is optimal in the sense that no player benefits from changing their strategy if no-one else changes theirs. What this means for co-operative coevolution is that it is capable of finding balanced solutions to problems where measures of fitness may conflict or be non-monotonic in some way. As we will see later, this is particularly apt for creative applications, where the notion of optimality is often entirely absent.

Complex Structures

The earliest work in co-operative coevolution was aimed at decomposing highly complex but also highly *structured* problems [101]. According to Wiegand, the primary motivation for this was the hypothesis that co-operative co-evolutionary approaches would be able to optimise smaller substructures in parallel more efficiently than a standard EC approach could optimise the original structure in its entirety.

The authors in [101] focus on an example relating to function optimisation, where the target solution is a vector of values for parameters which

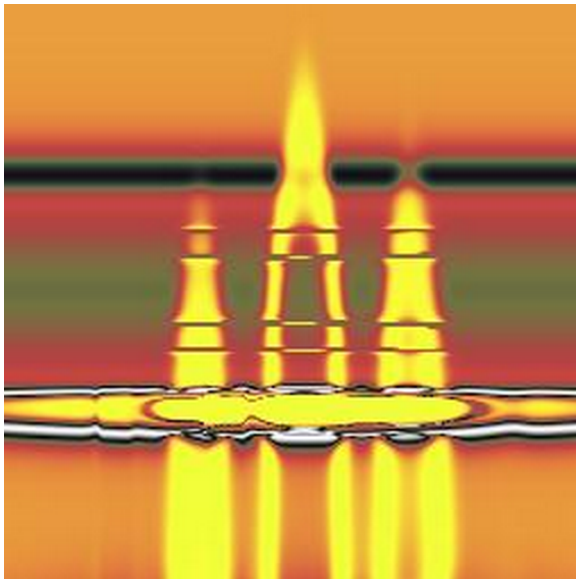


Figure 3.3: *River Temple* by Opah, an image created by the interactive evolution software Picbreeder.

are fed into a complex function, the aim being to find a vector of input parameters which maximise the value of the function. Potter and De Jong show that a co-operative co-evolutionary approach, where each parameter is assigned to a species, strongly outperforms a standard single-population EC system for the problem [101].

A crucial element when dealing with complex structures is finding an appropriate decomposition of the original problem structure. While some problems have trivial decompositions (such as a vector of parameters being decomposed into several single parameters) other structures, such as the ones which we describe in this thesis, are more harder to break down.

3.4 Evolutionary Art & Design

The use of evolutionary computation in the production of artworks is very well-established, perhaps moreso than any other creative domain that evolution has been applied to. Art created using evolutionary computation can vary between highly abstract, such as the work produced by Picbreeder [107] in figure 3.3, or more representational, such as the non-photorealistic rendering techniques shown in figure 3.4 by Machado et al.'s Photogrowth.

Figure 3.4: *Nude #7*, an evolved non-photorealistic rendering by Machado et al. and Photogrowth.

Evolutionary art has also been widely exhibited – artists such as William Latham have had their work installed in galleries around the world [78].

In [77], Lambert et al. cite Dawkins' *The Blind Watchmaker* [35] as an early populariser of computer-generated visuals with an evolutionary underpinning. Dawkins' pictures of evolved *biomorphs* in his book showed that evolutionary computation could produce organic forms as well as evoke surprise and interest in the viewer. While Dawkins' motivation was primarily to demonstrate evolution as a natural phenomenon, later proponents of evolutionary art such as William Latham explored the technique as an artform. Figure 3.5 shows a 3D sculpture produced by Latham's *Mutator*, an evolutionary computation-driven tool for generating both static and moving 3D art [123].

3.4.1 Fitness and Interactivity

Earlier in this chapter, we discussed how a population in an evolutionary system is evaluated using a fitness function, which sorts a population and influences which individuals will be chosen to produce the next generation of individuals. As we mentioned when discussing coevolution, many problems do not have objective measures of fitness which can be applied. In [40],

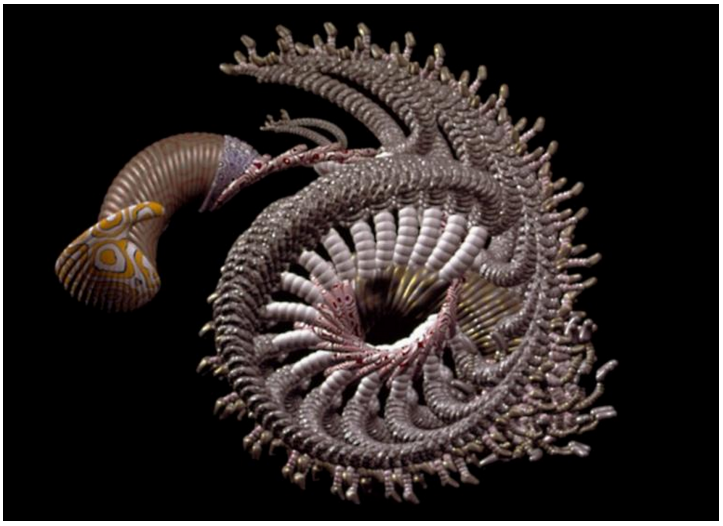


Figure 3.5: 3D sculpture evolved by Latham's *Mutator* program.

Eigenfeldt et al. point out that this is particularly true of problems in creative domains, where notions of optimality are often entirely absent.

Instead, evolutionary art systems are programmed with fitness functions that represent subjective notions of fitness. These notions may come from the programmer's own personal aesthetics, as was the case with Dawkins' biomorphs, or they may be generated by the software itself. Another common source for notions of fitness in evolutionary art is to derive them from the users of interactive software. This process of *interactive* evolution does away with traditional fitness functions, and instead asks a user to explicitly select images they like which are then used as the parents of a new generation of images.

We have already shown an example image from Picbreeder in figure 3.3, one of the most famous examples of interactive evolutionary art. The Picbreeder website contains over 9500 images published by its users as end results of interactive evolution. Figure 3.6 shows a screenshot of the evolutionary interface presented to users. In the top-right is the source image the user is developing, and in the center of the page are possible recombinations or mutations of this image. By selecting ones which the user likes, Picbreeder can generate further elaborations along the same lines, as well as presenting more diverse possibilities to suggest new ideas to the user.

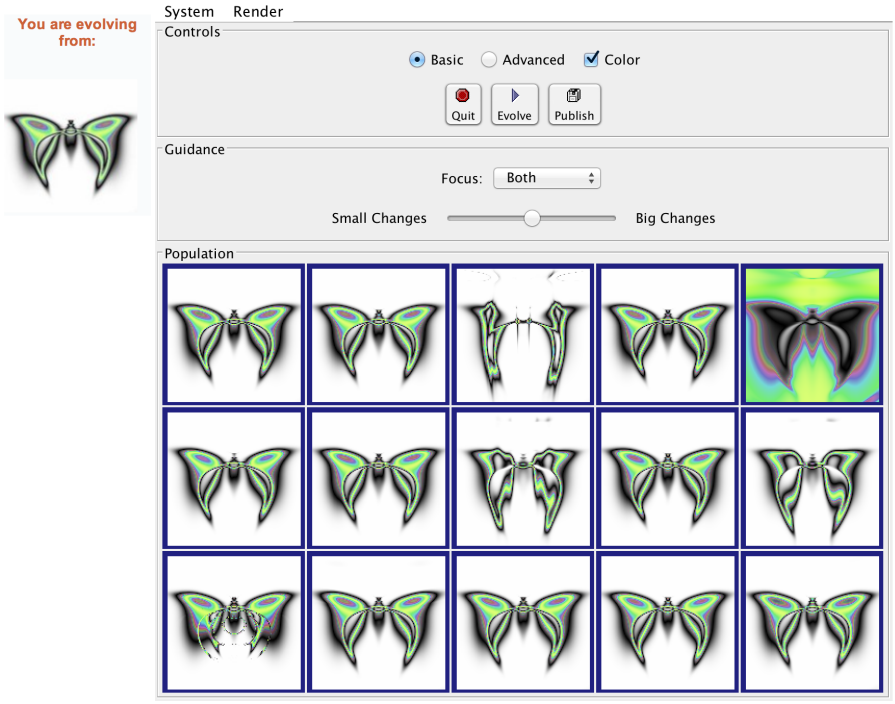


Figure 3.6: A screenshot of Picbreeder's user interface.

3.5 An Example System

In this section, we describe a simple evolutionary system for creating mazes, as a way of highlighting some of the concepts we have discussed above. We define a maze as a two-dimensional grid of tiles, some of which are marked as solid. Solving a maze consists of finding an unbroken path of non-solid tiles from a start tile, S, to an exit tile, X. Figure 3.7 shows an example maze evolved using the system we describe here. Recall Listing 3.1:

```
Initialise a population
Evaluate the population
Do:
        Select fittest from population
        Produce new generation from fittest
        Evaluate the new generation
        While not terminating
```

First, we choose a genotype representation appropriate to the phenotype. In this case, the representation is fairly straightforward: we use two-

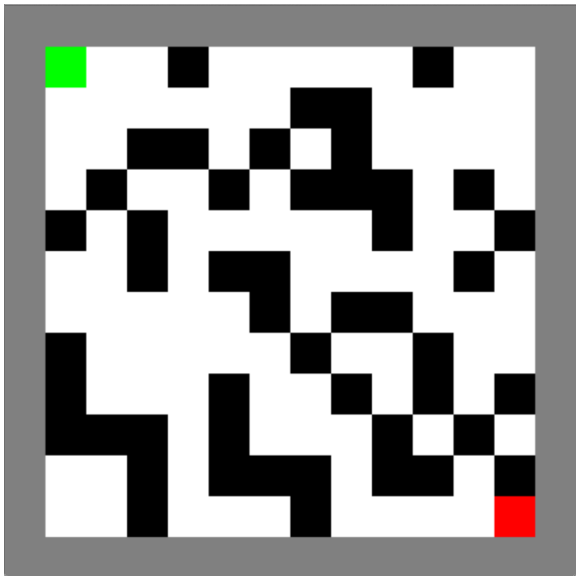


Figure 3.7: A simple maze evolved by our example evolutionary system. The start tile S is green, and the exit tile X is red.

dimensional arrays of booleans to represent a maze, with true entries in the array corresponding to solid tiles in the maze grid. We initialise our population by generating random arrays of booleans, with a 50% chance of each tile initially being solid.

We then evaluate the initial population according to a fitness function of some kind. A good maze has many different properties, but the simplest might be that we require a long path from the start of the maze to the exit. In order to normalise our fitness evaluation to the range $[0, 1]$, we'll calculate the theoretical longest path for the maze, and then an individual's fitness will be how close its path length is to that. For our mazes, since solid blocks occupy grid space, the longest theoretical path would be:

$$W * (W + 1)/2$$

Where W is the width of the maze (assuming, as in our example, that the mazes are square). Our fitness function, then, works as follows: it calculates the length of the path from the start to the finish for the current maze, using a simple A* search. Call this parameter *length*. The fitness f of an individual maze is then:

$$length/(W * (W + 1)/2)$$

We can sort our population according to this metric, assigning a fitness of zero to mazes that do not have a path from start to finish. We then select parents to generate a new population from. For this example, we will use elitism – the top 10% of the population in terms of fitness will be selected to produce the next generation. We take these pairwise, and generate new mazes using recombination and mutation.

Since our phenotype is a two-dimensional array, we can use one-point crossover to combine two mazes into a new maze design. To do this, we randomly pick a point (rx, ry) in the maze array. Then the new child maze is constructed as follows:

```
if(y < ry || (y == ry && x < rx)
    child[x][y] = parent1[x][y];
else
    child[x][y] = parent2[x][y];
```

This uses the selected point as crossover, reading left-to-right and top-to-bottom to decide at which point inheritance switches between parents. We can also randomly mutate a maze, by selecting grid points and flipping their boolean value between solid and non-solid. We balance recombination and mutation in the new population, so that mutated mazes make up approximately half of the new generation, while the remainder come from crossover.

Our termination condition in this case is a fixed number of generations - the maze generated in Figure 3.7 was evolved using a population of size 200 over 1000 generations. Many aspects of this evolutionary system can be varied to influence the types of mazes generated, however. The source code for this system, including visualization and parameters relating to some other concepts discussed in this chapter (such as steady-state populations) can be found at [30].

3.6 Summary

In this chapter, we introduced the field of *evolutionary computation*, involving techniques for solving combinatorial optimisation problems. We walked through a basic evolutionary algorithm, identifying different stages in the algorithm and giving examples of systems which exhibit these properties.

This basic algorithm will form the basis of many systems described in this thesis. However, we are most interested in a specialised type of evolutionary algorithm called co-operative coevolution. We introduced this notion as an extension of basic evolutionary computation, and discussed some of the strengths of the approach. We then looked at how evolutionary computation is used, in particular in creative domains such as the visual arts. We gave a brief introduction to evolutionary art, and discussed some different ways in which the concept of fitness is dealt with in a domain for which there are often no clear objective concepts of quality. This will be a familiar topic that we return to later in this thesis, in the context of another creative domain – videogames. Finally, we described a very basic evolutionary system to highlight some of the concepts we introduced in this chapter.

4 Background – Generative Software And Games

4.1 Introduction

In chapter 2, we introduced videogames as a creative domain, described methods for classifying games and assessing their value, and talked in detail about certain genres of game which will feature prominently in this thesis. In this chapter, we focus on one specific element of modern game development: *procedural content generation*.

We begin by defining procedural content generation (henceforth PCG) in §4.2, and describe the motivation behind its application to videogame development, both historically and contemporaneously. The history of PCG has had an important influence on the work described in this thesis. As a game design concept, PCG is a crucial underlying technique in the technical contributions of this thesis' work, as we show in chapters 5, 6 and 9. As a cultural concept, PCG's prevalence in videogame culture and the community's familiarity with the technology is an important factor in understanding the response and perception of the work, which we discuss in chapter 10.

In §4.3 we introduce a taxonomy proposed for procedural content generators, describing each classification in detail and giving examples of each type identified by the taxonomy. We propose an extension to the taxonomy that will have an impact on the work we describe in later chapters, and enable a richer comparison of related work in chapter 11 by precisely describing the contributions we make to procedural content generation with our approach.

To understand how procedural content generation is applied in a large commercial game, we provide a case study of *Spelunky* in §4.4, a game which is closely related to some of the game genres we highlighted in chapter 2 as being relevant to this thesis. Spelunky's level design is not only a

classic example of procedural content generation in videogames, it is also an example which relates directly to the level design system which we describe in chapter 6.

In §4.5 we turn to a specific subtype of procedural content generators: those which employ evolutionary algorithms, as described in chapter 3. We show that while there are only a few examples of evolution being employed within games, these examples are diverse and effective, including applications to intelligent design tools and commercial game releases. We show a variety of evolutionary techniques in use, which sets the scene for our own approaches, to be introduced in subsequent chapters.

4.2 History and Contemporary Application

The term procedural content generation (henceforth PCG) refers to any generative software process that produces *content* for use in digital software, usually video games. Content may be any asset used by the software in its execution, from art to music to code segments. While PCG is used in many creative mediums, its most common use is within videogames to help to produce game content, such as levels or visual assets. This section will give a background on the origin of procedural content generation within the games industry and how its role has evolved over time to its current state.

4.2.1 Needs-Driven and Wants-Driven PCG

The use of PCG within games originally arose as a way of circumventing technical restrictions, particularly those imposed by storage space limitations. Whereas a game such as *Pac-Man* [90] would store its level designs in data, this became impractical for games that wished to portray large world spaces, such as the galaxies of *Elite* (1984) [36] or the fantasy worlds of *The Elder Scrolls* (1994) [117]. Instead, data such as the arrangement of planets in galaxies or the heightmaps of a continent could be computed at runtime using seeded random number generators and carefully designed generative algorithms which used random numbers as input.

A simple example is Braben and Bell's use of the Fibonacci sequence to generate planetary data in Elite. Fibonacci sequences are deterministic and easily computed, requiring only two starting numbers (since every number

is the sum of the two preceding integers in the sequence). By relating integers in the sequence to features of the planet (such as mass, habitation types, imports, exports, and so on) an entire planet could be stored as just two digits. This huge compression allowed for the storage of vast numbers of planets, but in doing so Braben and Bell relinquished control over the individual planets' design, as their properties were inherited from the pseudorandom output of whatever Fibonacci sequence generated them.

The loss of authorial control over content generated in such a way is an interesting feature of procedural content generation in modern games culture. Games are fundamentally interactive pieces of software, and so the task of controlling what a player experiences is already harder than in other creative fields such as film or music. If a film director wishes the audience to see a specific thing, they point the camera at it. In a game, the player is typically in control of what they experience. They may choose to visit game areas in a different order, ignore certain tasks, or skip through plot exposition. A game designer can choose to restrict the player's freedom, of course, but in doing so they may also reduce the player's sense of freedom, control and enjoyment.

PCG complicates the issue of authorial control even further, by removing authorial control over certain areas of the game completely. In *Spelunky*, the game's designer cannot know the exact nature of the levels the player will be asked to complete. Their PCG system defines a possibility space, which they can control in a limited fashion by making adjustments to the PCG system itself. Ultimately, however, they have no way of knowing for sure what content will be generated when. We will show later in this chapter how this can be used as a strength rather than a weakness, however.

We call the application of PCG to circumvent technical limitations, such as the galaxy generation in *Elite*, *needs-driven PCG*, as it was initially adopted to solve a problem rather than to provide an attraction to the consumer. Needs-driven PCG still abounds in the modern games industry, as it is frequently used to generate content that is technically infeasible to generate by hand. A good example of this is the SpeedTree system [64] for generating foliage, which has been used in many blockbuster games such as *The Elder Scrolls IV: Oblivion* [119]. Manual design and placement of trees which look unique and naturally 'random' is not only hard for a designer to do, but also time-consuming and tedious, for a task that amounts to

set-dressing. Needs-driven PCG techniques neatly solve such problems.

The benefits of procedurally generating content go beyond merely overcoming budget constraints, however. The vast galaxies of Elite were not just technically impressive – they provided a major selling point for the game. Games designed by hand had worlds that were easily exhaustible. Players could draw maps of the entire game space, and know everything there was to know about a particular game. Elite's vast galaxies were too big to explore completely, and this offered a complexity and sense of scale that had not been seen before in videogames. Procedural content generation has consistently been used in this way: not simply to make a game take up less space on a disk, but to offer a new kind of gameplay. We call this use of PCG as a selling-point for games *wants-driven* PCG.

Possibly the most famous example of wants-driven PCG in modern video games is the world generation of Mojang's *Minecraft* [99], which situates the player in a world made of cubes that can be destroyed, collected, placed elsewhere and combined into new items. Minecraft's worlds extend to eight times the surface area of the Earth in size, each one randomly generated. The potential size of the world is so vast in comparison to the area a player will likely explore that they are sometimes described as pseudoinfinite.

Figure 4.1: An unusual formation found in a particular *Minecraft* world.

PCG's role within the game is to offer unpredictable worlds to explore, which accentuates the game's themes of pioneering and survivalism. It also

engenders a sense of connection between the players and the PCG system that generates the worlds, to the point where players will share the random strings that seed a world with particularly interesting features so that others can experience them[1] (figure 4.1 shows a screenshot from one such shared world seed). Here, PCG is not merely solving a content problem – it is augmenting the mechanics of the game by adding in unpredictable content, something that human designers could not provide without continuously developing new content by hand.

Another way in which PCG can contribute to the worth of a game is by producing content on such a large scale and with such diversity that a sensation of personalisation is created. *Dwarf Fortress* [43] generates a finite two-dimensional world, in contrast to Minecraft's pseudoinfinite spaces. Once generated, Dwarf Fortress then continues to generate the world's history, founding civilisations with mythologies and successions of leaders. Though simplistic and almost entirely random, the generative process is extensively detailed, which leads to players investing heavily in the specific history of the world they are playing in. Dwarf Fortress is often experienced by people who do not play the game at all – the detail and variety in the worlds generated has made the game extremely popular among a community of gamers who write reports of the games that they play (colloquially referred to as *Let's Plays*), which are often read by people with no interest in learning how the game works. Without PCG, the game would have no unpredictability in the worlds that the game takes place within, and hence unlikely to be of interest in Let's Play scenarios. Earlier in this chapter we discussed the problem of relinquishing authorial control by using PCG to generate content. In Dwarf Fortress, the loss of control is a strength of the game – the players feel invested in the fact that no-one else has possibly seen this world before, and the act of reading world histories and inferring narratives from the simple account of events is itself part of the game's appeal.

4.3 Classification of PCG Systems

In the previous section, we offered a broad partitioning of PCG systems based on the motivation for their use. More detailed classifications of procedural content generators exist, based on the features of the generator

[1] http://www.minecraftseeds.info/

itself, such as in [126], in which Togelius et al. put forward five axes along which PCG systems can be classified. Most of these, as we describe below, are not binary classifiers but rather linear gradients along which many different kinds of PCG systems may lie. We give the five axes below, along with examples of games which fit at either ends of the spectra:

Online – Offline

An *online* PCG system may operate at runtime within the game, generating content while the player consumes it, such as the level generation in *Spelunky* [47] (see below). An *offline* PCG system may be used prior to the game being released to generate static content the game will subsequently use (such as the generation of a galaxy map in *EVE: Online* [44]). Offline PCG systems are more likely to be needs-driven, since they solve a design-time content problem, such as the generation and placement of trees, but the player only experiences static content in the final game.

Necessary – Optional

Necessary PCG systems generate content which the player is guaranteed to interact with on the *critical path*[2] through the game. For example, the levels in Spelunky are procedurally generated, and it is not possible to play the game without experiencing the content produced through this system. *Optional* PCG systems generate content which the player either chooses to experience, or experiences in a way which is not vital to the completion of the game. For example, optional quests in *The Elder Scrolls V: Skyrim* [120] have a procedurally-generated component, but the content may not be experienced by some players, depending on their chosen path through the game.

Random Seeds – Parameter Vectors

Informally, this feature of the taxonomy describes how controllable the input to the PCG system is. Some PCG systems derive their input from streams of random data, such as world generators which use Perlin noise [98]. The

[2]In game design terminology the critical path refers to the minimum path from the start to the finish of the game, i.e. content which is necessarily experienced by the player in completing the game.

primary way of interacting with or varying these PCG systems is through providing a random seed to the generator that drives the system. For instance, Minecraft's world generator works on this basis: players can provide different seeds to the world generator, but cannot influence the process in any other way. In contrast, some PCG systems use parameterised inputs, allowing them to offer some control to players or designers. For instance, map generation in the *Civilization* series [87] has many parameters that can be altered prior to world generation that affect geographic factors such as sea level, erosion rates or tectonic activity. These influence not only the starting point of the world generation, but also the way the world develops as it is generated, affecting the final output.

Stochastic – Deterministic

This feature is closely related to the feature *Random Seeds/Parameter Vectors* above. *Stochastic* systems produce different pieces of content if they are run multiple times, even if they are set to the same starting configuration each time. *Civilization*'s world generator is an example of a Stochastic system, as it produces a different world map each time it is run, even if the settings are the same. In this case, the settings for the generator dictate a particular style of generator, rather than guaranteeing a certain output. *Deterministic* systems produce the same piece of content given identical input parameters, although Togelius et al. do not consider the random seed to be a parameter in this case (since all systems would be deterministic under this definition). PCG systems which use random seeds, like the galaxy generator in *Elite*[36] we described earlier in this section, would be an example of a deterministic procedural generator. Given the same two initial numbers, the resulting planet always has the same set of features.

Constructive – Generate-and-Test

Constructive PCG systems produce their content in discrete phases, additively contributing content until finished. These PCG systems are guaranteed to produce usable content at the end of the process, and normally include expert knowledge embedded by a designer that ensures that usable content is always the result. *Spelunky*'s level generation (described in detail in §4.4) adds content to a level in stages, and each stage preserves the level's

playability without evaluation, meaning it is not possible for the generator to produce content that is not usable. *Generate-and-Test* approaches, as their name suggests, go through two phases – the generation of a piece of content, and then an evaluatory phase which considers whether it is usable or not. Whenever a piece of content is not usable, the Generate-and-Test system will normally restart the generation of that piece of content from the beginning. Generate-and-Test is not common in online PCG, because players are often not willing to wait for a system to produce content, and such systems provide no guarantees that content will be created within a particular time period. *Dwarf Fortress* [43] uses generate-and-test in its world generation phase, despite being an example of online PCG, because its playerbase is willing to wait for a considerable time for a world to be generated. The resulting worlds are often used for dozens of hours of resulting gameplay, which also offsets the issue of waiting for a long period for content to be generated.

4.3.1 Extensions to the PCG Taxonomy

In the previous section, we described a taxonomy for categorising search-based procedural content generation systems according to the manner in which they generate content. The classifiers help to define a wide space of PCG systems and highlight many important distinctions between such systems. The taxonomy speaks to the role of the PCG system within the wider game (*Online/Offline* and *Necessary/Optional*), the variability and repeatability of the system's execution (*Seeds/Vectors* and *Stochastic/Deterministic*) and the generative approach taken to content creation (*Constructive/Generate-and-Test*). However, the taxonomy does not offer a way to talk about PCG systems which are composed of multiple generators, or the ways in which multiple generators may interact. This is understandable, as PCG systems tend to be perceived as black boxes whose internal workings are not typically studied or analysed. However, this need not be the case, and in order to analyse systems which autonomously design games, including the *ANGELINA* system we describe later, it will be helpful to be able to talk about the degree to which the content generators inside an autonomous game designer interact and execute in concert with one another.

We propose two further classifiers for search-based procedural content

generation systems here. These new classifiers are based on the premise that content generation often occurs in distinct stages, in which multiple types of content may be generated. For example, Ed Key's *Proteus* [71] takes place on a procedurally generated island. While this can be thought of as a single generator under the taxonomy in [126] that is *online*, *seeded*, producing *necessary* content, and so on, it is also informative to consider it as a system composed of many generative steps. For example, the heightmap of the island is generated separately from the placements of the landmarks or flora and fauna. These generation phases are not only distinct, but interrelated, for instance the placement of content such as types of plant life is dependent on the content generated in previous stages.

By considering such a system as performing many generative steps, we can classify it in greater detail and build up a more expressive taxonomy of procedural systems. This is particularly important when considering automated game design systems that may tackle several generative tasks simultaneously, such as the systems we describe in this thesis. For a series of procedural content generation tasks, our new classifiers distinguish the degree to which the tasks are interrelated (*dependence-independence*) and whether the generative acts are interleaved (*sequential-parallel*). Both of these features are useful in analysing our approach with ANGELINA as well as distinguishing related work in automated game design from other existing work in procedural content generation.

4.3.2 Dependent versus Independent

Independent PCG systems produce new content without reference to the content which already exists within the game, either as output from other generators, or statically placed by designers. For example, early versions of the village placement algorithm in *Minecraft*'s world generator forced the placement of villages in the world, regardless of the surrounding geography. This often resulted in villages that were embedded in the sides of mountains, or otherwise completely disconnected from the main world space. By contrast, *Dependent* PCG systems generate content with respect to the context the content will eventually be placed in. Many games in the *Roguelike* genre, such as the eponymous *Rogue* [127], place items and monsters in levels after the level geometry has been generated, ensuring that content is evenly spread throughout the world.

The Dependent/Independent classifier is a linear rather than a binary classifier. Some games exhibit high levels of dependence in their generation, including *Dwarf Fortress* [43], whose world generation is performed in discrete stages which use geographical and geological models to place rivers, settlements and natural features according to the features of the generated world's topology. Other games exhibit some dependence in their generators but not as comprehensively, only having small amounts of information shared between generators. For instance, *Spelunky* [47] places monsters and items with little regard for level flow, but always ensures level exits are accessible based on the level layout.

4.3.3 Sequential versus Parallel

Sequential generators act in distinct stages of generation, with PCG systems acting one after another in sequence to produce content. This often facilitates *dependent* generation (as described above), as it allows for one phase of content generation to completely finalise its output before the next type of content generation begins. While this is largely a binary classification, we can imagine generators which perform some computation in parallel but whose generative steps are mostly performed sequentially – for example, a system which generated a dungeon one room at a time, but fills each room with content using parallel generators. Here, at the highest level, the dungeon is being generated sequentially, one room at a time. However, the individual rooms are generated in a parallel fashion.

Most procedural content generation in commercial or hobbyist videogame development is performed sequentially. We believe that the reasons for this include a more direct analogy with human content generation such as level design, as well as the fact that sequential systems have a conceptually simpler system design. For example, ambient exploration game *Proteus* [71] designs its islands in distinct generation phases which complete fully before the next phase begins. Parallel examples are rare, but work in [70] using multiple agents to lay content in a level simultaneously can be thought of as a parallel content generation system. In this case, each agent is placing content at the same time, allowing interactions between the separate content-generating agents.

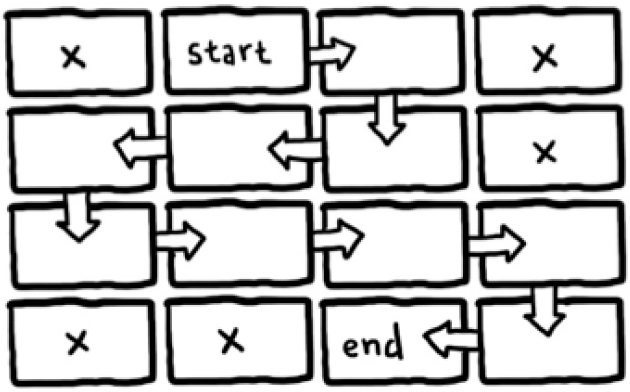


Figure 4.2: An illustration by Derek Yu describing Spelunky's level generation process [135].

4.4 Case Study: Spelunky

Spelunky [47] is a 2009 action-adventure game developed by Derek Yu for the PC and later ported to the XBox 360 in 2012. Spelunky received huge critical acclaim throughout its initial release and subsequent redevelopment for console, and particular praise was given to its use of procedural content generation in the platform genre. Yu envisaged the game as a combination of platform game and roguelike (see §2.2.1 and §2.2.2 respectively for descriptions of these genres), pointing to games such as *La Mulana* and *NetHack* as inspirations.

La Mulana is a popular Japanese platform game that belongs to a sub genre of platform games that pride themselves on being extremely difficult to play. The official guide to *La Mulana* cheerily states that "... players learn to play this game as they get killed over and over. Many people might give up!" While this may seem counterintuitive as a design ideal for a piece of entertainment, these games (sometimes dubbed *masocore* games [2]) enjoy a large following among gamers. Games in this genre tend to use the replayability of a game as a tool for the player to learn with – because the game's content is static, the player can more easily relearn patterns and understand how to quickly play the game.

Spelunky disrupts this notion by using procedural content generation to automatically design game levels. Derek Yu describes the process by which levels are designed in [135], which we paraphrase here. The world is com-

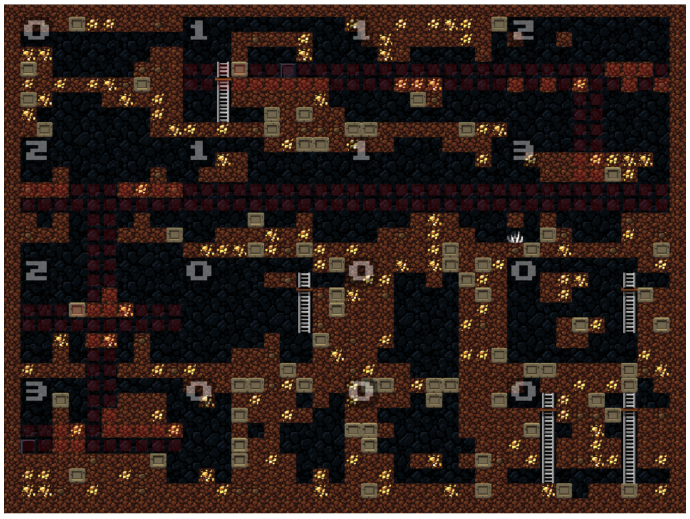


Figure 4.3: A sample Spelunky level, generated by Darius Kazemi's interactive generator [69]. The player starts in the second tile from top-left. The exit in this level is in the bottom-left.

posed of screen-sized templates which Yu creates himself beforehand, and categorises them according to which directions they can be entered and exited from. The level generator then arranges a series of template screens from the start point to the exit point, ensuring that the exit can be reached from the start. Remaining sections of the map that were not filled in by the generator are randomly assigned from the list of template tiles.

Figure 4.2 shows a high-level overview of a Spelunky level generation, used with permission from [135]. The arrows indicate the path composed of tiles selected by the level generator. The tiles marked with an X are tiles which were not filled in on the initial generative step, as the player doesn't need to pass through them to reach the exit. These tiles are filled in with randomly-selected templates, which may or may not be accessible to the player. Figure 4.3 shows a level generated using an interactive tool based on Yu's original generator [69]. A line of highlighted tiles shows a path from the beginning of the level (the second tile from the left in the top row) to the exit (the bottom-left tile).

Notably, many gamers do not initially realise that Spelunky's levels are generated from such large template chunks. This is partly due to the high number of templates added by Yu in the final game release, but another contributing factor is the post-generation adjustment phase, where the level

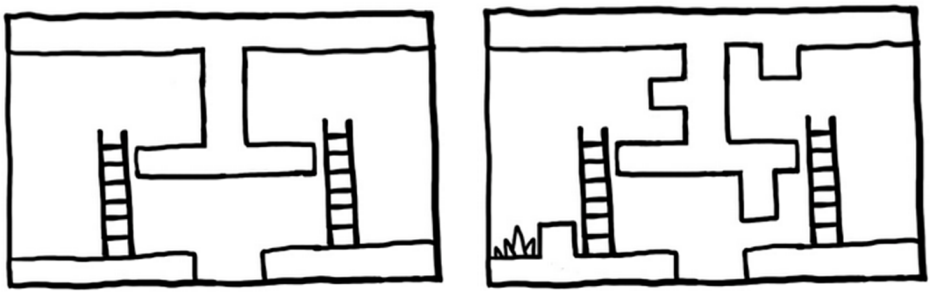


Figure 4.4: A screen-sized template for a level in Spelunky, before random adjustment (left) and after adjustment in-game (right), from [135]

is randomly tweaked to change its layout without affecting how traversable the level is. Figure 4.4 shows how a screen-sized template can be adjusted randomly to change its visual signature, without altering how the tile slots into the overall level design.

In a postmortem of the game's development, Yu notes that platform games tend to exhibit repetition and that "not much improvisation is required to play". Procedural content generation is used to great effect here to overcome this, which highlights how PCG can be used as a unique gameplay element used by a designer, rather than a replacement for more content.

4.5 Evolutionary Procedural Content Generation

Computational evolution is not a commonly used technique in procedural content generation. In particular, it is rarely used in online PCG, because of a lack of guarantees that the system will produce output of a particular quality, or within a given time. However, games exist which use content that was evolved offline during its development, such as *EVE Online*'s galactic map. Attempts in academic work to use evolution both online and offline are more common. We describe some approaches in this section.

4.5.1 Evolution and Online PCG

Galactic Arms Race

Galactic Arms Race [45] is a multiplayer action game where players fight each other in spaceships, gaining new and more powerful weapons over time.

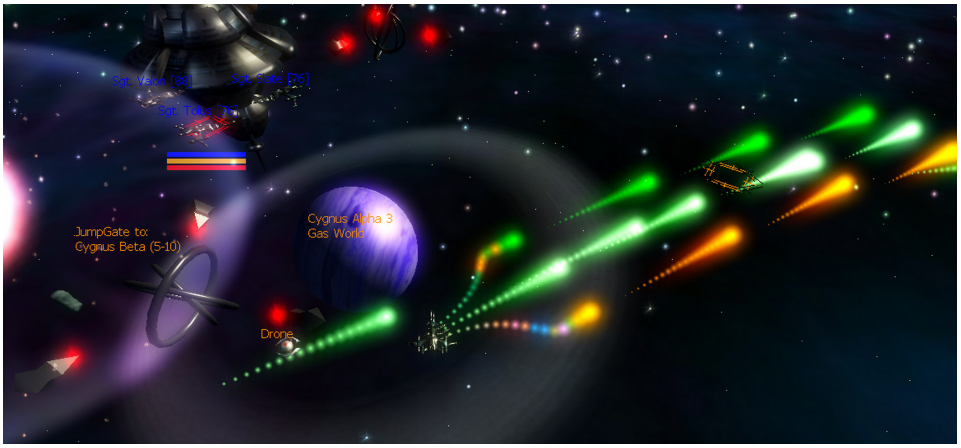


Figure 4.5: A screenshot from *Galactic Arms Race* showing an evolved weapon being fired.

The weapons the players find in the game are procedurally generated, and have varying features such as the rate of fire, the shape and speed of the projectiles, and the visual effects associated with the weapon when it fires. Figure 4.5 shows a player firing an evolved weapon during gameplay. The weapon generation uses neuro-evolution of augmenting topologies (NEAT) [118] to adapt weapon characteristics to player preference while simultaneously maintaining diversity and novelty. The amount a weapon is used by a player is interpreted by the system as an expression of preference, and acts as the fitness function which directs the NEAT system.

New weapons appear in the game at major game milestones, such as the defeat of a particularly strong enemy. These may be readymades from a stock of pre-evolved content called the *spawning pool*, or a new weapon evolved from the current population of weapons, which consists of any weapon currently being used by a player in the game. This therefore uses two kinds of player-driven selection – not only how often a player has fired a weapon, but also whether or not they decided to continue carrying the weapon, since players can swap out weapons at will for new ones.

Galactic Arms Race was originally envisaged as a research project [58], but is still under development four years after the publication of the initial research papers, and was recently accepted onto the Steam marketplace, the largest digital download store for games. The commercial success of the project demonstrates that evolution can have applications to online PCG

for games.

Racing Games

Evolutionary PCG has been applied to content generation in racing games on more than one occasion. In [124], Togelius et al. present a level generator for a racing game which tries to learn a model of the player before using that model to guide an evolutionary algorithm in designing a race track. A partially-optimal neural-network-based controller is trained on recordings of a player controlling a car through several calibration tracks, in order to develop the controller to more closely mimic the player's behaviour. This trained controller is then used by the system to evaluate tracks during the evolutionary process. Since the trained controller models aspects of the player's racing style, the controller's performance on unseen tracks can be used to infer how well the player might perform on them. The tracks are also evaluated according to other criteria, some of which interact with the model of the player. For example, the top speed the player model obtains when playing the track is affected by both the presence of straight sections of track (which is independent of the player model), as well as the player's propensity to accelerate during such sections.

Evaluation of the results in [124] takes place mostly anecdotally by the authors. Tracks evolved for different controllers are visually compared for variation as an indication that the system is personalising the tracks, and further compared to tracks evolved without a controller to show that adaptation is taking place.

In [14] Cardamone et al. present another designer for racetracks, this time focusing not on a model of the player, but the qualities of the racetracks themselves, independent of any particular playthrough. In particular, the authors focus on *diversity* within an individual racetrack, maximising the variation in types of track segment, driving speeds and turn rates. They present multiple types of evolutionary track designer focusing on diversity in each of these areas, and compare both single- and multi-objective evolutionary approaches, showing that both methods were capable of producing tracks that players rated highly.

The authors present an evaluation through player surveys, first showing people pictures of the evolved tracks and asking them to express preference over pairs of tracks. This was then followed up by multiple play sessions

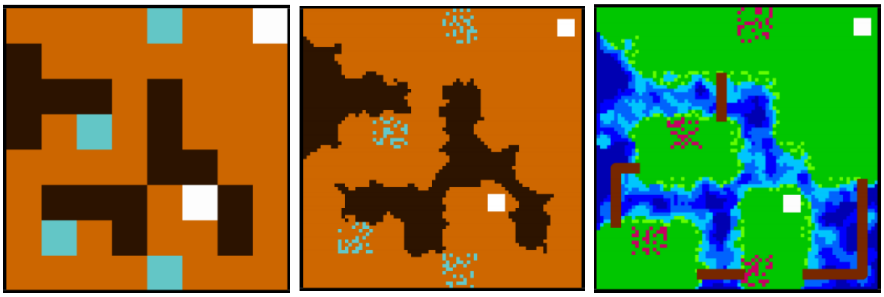


Figure 4.6: A low-resolution sketch by a user (left) and two high-detail visualisations of the sketch made by The Sentient Sketchbook (centre and right). In the original sketch, black tiles denote impassable areas, blue tiles denote resources, and white tiles denote player starting locations.

where both evolved and hand-designed tracks were supplied to the participants. Responses from the participants were recorded in terms of pure track preference, but also more specific features such as the perceived challenge of a track. Evolved tracks were preferred overall, and results also indicated they were seen as much more challenging than hand-designed tracks, particularly for less experienced players. This did not appear to affect player's willingness to replay the tracks, however, despite their difficulty.

4.5.2 Evolutionary Assistance – Mixed Initiative Tools

The Sentient Sketchbook is a mixed-initiative design tool for designing game worlds, described in [80]. The tool works in a user-led fashion, initially beginning with a low-resolution sketch of a game map by the user, which is then translated to a greater level of detail by an evolutionary system, using a NEAT technique similar to that employed in *Galactic Arms Race*. These translations can be done in multiple styles: Figure 4.6 shows a user-drawn map sketch alongside two different translations of the sketch at higher levels of detail, generated by the tool. This translation method allows a designer to roughly draw out important map features (such as inaccessible paths, or approximate routes through a world) and have the lower-level detail of a world, such as the natural flow of geographical features, handled by the procedural content generator.

The Sentient Sketchbook is also capable of providing suggestions about

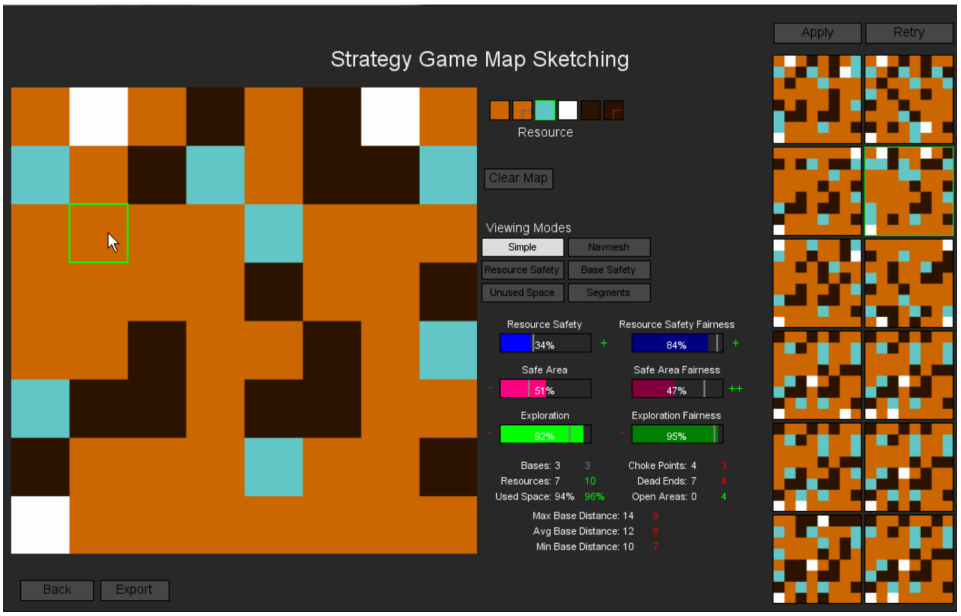


Figure 4.7: A screenshot from The Sentient Sketchbook. The user sketch is on the left, and live suggestions appear on the right-hand side of the screen. Options and metric information about the sketch appear in the central column.

the current user sketch in real-time. Suggestions appear on the side of the user's screen, evolved by the system using novelty search [79], an evolutionary computing technique which incorporates a preference for diversity into its selection process. This results in suggestions which relate to the user's current sketch, but adapt it in a variety of different ways, with the intention of serendipitously sparking new ideas in the designer, or possibly accelerating the design towards a goal which the user already had in mind.

The Sketchbook is a good example of evolutionary techniques being applied to offline procedural content generation, and assisting in high-level design tasks alongside a game designer. The Sketchbook is able to rapidly provide natural-looking interpretations of abstract design ideas, and offers suggestions to designers to lead the development of a game world in unexpected directions.

4.6 Computational Creativity

Computational Creativity is a field of artificial intelligence which concerns itself with

> ...the philosophy, science and engineering of computational systems which, by taking on particular responsibilities, exhibit behaviours that unbiased observers would deem to be creative.

This description of the field, quoted from [29], carefully sets out many important aspects of the field. Deconstructing creative activities into *responsibilities* allows them to be studied in isolation, and composed with one another to build more complex systems. This pattern will be evident to the reader in this thesis, as we build an automated game design system out of diverse creative responsibilities that interact with one another to create full games. The notion of *unbiased observers* touches on another important area of the field, namely evaluation.

4.6.1 The Challenge of Evaluation in Computational Creativity

Agreeing on whether something is creative or not is difficult for any group of people to do, including among communities of creative people and within the Computational Creativity community. A lack of consensus on what constitutes creativity makes evaluating software which claims to be creative difficult. We argued in [26] that this is due to creativity being an *essentially contested concept*[52] – a term used to describe certain concepts where the lack of agreement relating to the concept is inherent in the concept's function in culture. That is, the fact that we disagree and debate the meaning of terms such as justice, art and creativity is part of their value to society, because that discourse drives the development of these concepts and improves society in doing so.

Even if we were able to agree on what constitutes creative activity, however, the issue remains that assessing how *well* a creative action is being performed is still extremely difficult. This is particularly noticeable in Computational Creativity when trying to assess the performance of a piece of creative software. As Eigenfeldt et al. note in [40]:

> Rational problem solving [in Artificial Intelligence] is evaluated by comparison to some optimal solutions... Computational Creativity is faced with the dilemma that, while creative behaviour is intelligent behaviour, such notions of optimality are not defined.

While traditional artificial intelligence paradigms might use an optimal solution as a benchmark, in Computational Creativity we cannot use the notion of a 'best' painting or a 'best' videogame to compare the performance of creative software against, because the idea of an optimal creative work is anathema to most creative domains. This further complicates the process of evaluating computationally creative software.

4.6.2 Evaluation Approaches in Computational Creativity

Product

In [102], Ritchie describes several criteria which can be used to assess a system which generates content of some kind. The criteria are defined mathematically, and so can be applied precisely to a system, assuming certain definitions can be made, such as a notion of 'quality' for the domain the system works within (which can be problematic for reasons we outline above). Many of the criteria refer to the system's output directly, which Ritchie calls the *result set*. This distinguishes the approach from ways of evaluating systems which followed Ritchie, which we describe later, by focusing purely on the system's input and output, rather than the system itself.

The criteria given by Ritchie evaluate the system's output along the lines of typicality (the degree to which a particular artefact is an example of the class of artefact which it is intended to belong in – does the joke generator output things which look like jokes, for example) and value (a combination of the perceived quality of an artefact and its novelty). Ritchie proposes eighteen criteria in total.

Ritchie's criteria are also the basis of the *curation coefficient* measure for a piece of software's performance. The curation coefficient is the proportion of a system's output that the system's creator would be comfortable sharing with others publicly as the work of the system, to be assessed as a creative artefact. A curation coefficient of 1.0 implies that the system is capable of working completely independently with no filtering or selection on the part

of the designer. Increases in curation coefficient, therefore, can be used to argue that one system is improving, or better than another system, because it is better at assessing the quality of what it is creating.

Process

Ritchie's focus on the output of a system means that creative software is treated as a 'black box' whose internal working is not important in evaluating its worth or success. In [25] Colton et al. present a model for describing creative software according to the kinds of creative *acts* they perform. The authors say:

> It is clear that such [creative] acts can occur at the ground level, whereby generative systems produce new artefacts such as theorems, pictures, compositions... However, it is also clear that creative acts can occur at the process level, where new ways to generate and assess artefacts are invented.

In other words, although the output of a system is a creative act, there can also be creative activity that occurs internal to a computationally creative system. Often we find that this activity is as important, if not more important, than the assessment of what the system creates. Colton et al. propose four categories of creative act in the FACE model presented in [25]: Framing, Aesthetics, Concepts and Expressions of concepts. Each act can be performed at a generative level, meaning an activity that generates examples of each of the four categories, or at a process level, where new methods for generating examples are created by the system.

The FACE model recognises creative acts beyond the simple generation of artefacts, which is important when evaluating whether progress has been made in the development of creative software. The creativity of a system (or indeed a person) does not always relate to the perceived quality of what is produced. In [26] we introduce the notion of *latent heat* in Computational Creativity. This term describes the phenomenon whereby a piece of software is iterated upon and improved in terms of its creative ability and autonomy, but that resulting increase in independence initially results in a lower quality output. This drop in quality ultimately disappears over time as the software is further improved, but in the short term the system appears

to be producing worse quality artefacts despite being a more sophisticated system internally. An assessment that only focuses on the system's output fails to recognise the steps forward made by such a system.

Perception

The proposed shift in focus from evaluating product to evaluating process has not entirely been accepted by the Computational Creativity community, and many different evaluatory methods are often used to give a mixed picture of a system's performance, such as in [89]. A lack of consensus is no doubt partly down to creativity being essentially contested, as we mentioned earlier, and mixed approaches also help in providing many different angles to consider a system from.

An emerging feature of evaluation alongside the system's output and the actions it performs to generate such output is the notion of the *perception* of the system by consumers, critics, creatives and other stakeholders both in and out of the creative domain the system works in. The interest in the perception of a piece of software stems from the same line of thought that led researchers to evaluate the processes within a system. In [19] Colton states that:

> As we developed The Painting Fool[3], we began to understand that when consumers of paintings assess them, they do not strictly separate the process and the artefact.

In the previous two subsections we described approaches to evaluation that focused on the quality of artefacts, and approaches that focused on the processes which were undertaken to produce them. Colton's assertion above makes an important additional point, however: that outside of an academic context, it is difficult for most people evaluating creative software to keep the two separate.

Charnley et al further this point in [16], describing artworks in which the information and context surrounding the artwork are more important than the physical artefact itself. They refer to this as *framing* information, and discuss how this information helps shape someone's perception of

[3]The Painting Fool is a computationally creative artist developed by Colton over many years. We discuss the work in chapter 11.

the artwork. Without context, an artwork may be hard to understand or appreciate, or may not be recognised as a piece of art at all.

Although the perception of a piece of software is hard to judge quantitatively, we can look at major milestones in a project's development as indicators of how it is viewed in relation to other creative software, other creative people, and the history and context of a creative domain – such as acts of recognition from creative communities (accepting work into exhibitions, for instance) or analysing the language used by people when discussing the software. Crucially, however, assessing the perception of a piece of software relies on being open about its identity as a piece of software, rather than staging anonymous Turing-style tests [129]. Pease and Colton have argued strongly in the past against Turing-style tests for evaluating creative software, citing reasons such as the fact that the test encourages 'window-dressing' and trickery to deceive evaluators into thinking a piece of software is being more creative than it really is [97].

4.7 Summary

In this chapter, we introduced the concept of *procedural content generation*, and gave a brief history of its role in videogame development. We identified two types of motivation for the use of PCG in games – *needs-driven* and *wants-driven* – and further classified PCG systems according to their design, structure, application and execution. In doing so, we introduced two new axes for classifying PCG systems: *dependent-independent* and *sequential-parallel*. These will help us highlight the unique qualities automated game design has as a research problem later in this thesis. We looked in detail at *Spelunky* and its use of procedural content generation in creating its levels. We looked at how its PCG system combines human-designed fragments in new ways, and tweaks its levels to obscure the static nature of its starting templates. Finally, we looked at the use of evolutionary techniques in procedural content generation, considering both *online* and *offline* examples, as well as its application to design tools. Later in this thesis we will use these examples, and the taxonomy, to compare and contrast our approach with other PCG systems, from both academia and the wider games industry.

5 Coevolution in Arcade Game Design

5.1 Introduction

In this chapter, we describe *ANGELINA*$_1$, the first iteration of *ANGELINA* and the foundation for the ways in which *ANGELINA* develops through subsequent chapters. *ANGELINA*$_1$ focuses on creating the most simple game archetype which we discussed in chapter 2: arcade games. The focus of this chapter is the fundamental question: can co-operative coevolution (CCE) be used as the basis for automated game design? We present *ANGELINA*$_1$ as a proof of the basic concept, and then develop this with further study through subsequent chapters and versions of the software. This chapter will describe the design space *ANGELINA*$_1$ occupies, the structure of its internal CCE system, and the games it can produce.

In §5.2 we give details of *ANGELINA*$_1$'s design space, describing the motivation for some of our choices of problem to solve, and the preliminary work which inspired this initial system design. This design space defines the kinds of games that *ANGELINA*$_1$ can design, and clearly divides up the different tasks which we give to the various components of the CCE system in this version of *ANGELINA*.

In §5.3 we describe each species in the CCE system of *ANGELINA*$_1$ in turn. We give details of the individual species' design space, how it generates its population, how each population is evaluated (according to the type of game content the species deals with) and how generations are recombined and mutated. We discuss in particular detail the evaluation approach taken with simulating gameplay, as this is closely related to both our use of CCE and the core challenge of automated game design: evaluating an entire game design at once, to assess all generated parts of a game simultaneously.

In §5.4 we provide examples of games designed by *ANGELINA*$_1$, including

screenshots and brief descriptions of their structure. Some of these games are generated with complete autonomy, with the CCE system acting to design each aspect of the games itself. Other games are designed with partial hand-designed content, to show the flexibility of the system and to give an indication of how species can respond to content changes within the CCE system.

5.2 Design Space

In [125], Togelius and Schmidhuber describe a series of experiments in generating rulesets for videogames. We describe the work in greater detail in chapter 11. This work inspired the foundation of $ANGELINA_1$, in particular the design space it uses as a basis for generating games. While Togelius and Schmidhuber focused on rulesets only, in a turn-based environment, we built $ANGELINA_1$ as a modular system capable of evolving multiple aspects of a game simultaneously, both evaluating and outputting games with real-time gameplay.

The version of *ANGELINA* described in this chapter works in a design space of arcade-like games. The games are played on a two-dimensional grid of tiles, each of which can be either solid or empty. In this grid, a number of coloured circles or *entities* are placed, as well as a single grey dot denoting the player. There is always one player, but there may be any number, or none, of each of the three entity types (red, blue, green).

Each of the three coloured entity types have movement behaviours associated with them, chosen from a series of predesigned movement styles:

- *Clockwise* and *Anticlockwise* - the entity moves in an initially randomised cardinal direction (north, south, east, west). When it hits a solid tile, it rotates either clockwise or anticlockwise (depending on the behaviour) and then continues in a straight line.

- *RandomShort* and *RandomLong* - the entity moves in a randomly selected direction for a randomised period of time, before changing to a random cardinal direction. The time period is randomised between two limits, which are dependent on the specific behaviour.

- *Static* - the entity does not move.

The rules of these games are centred around collisions between tiles and entities, or entities and other entities. All rules can be expressed in the same basic 5-tuple format:

```
<Object1,Object2,Effect1,Effect2,ScoreChange>
```

Such rules are interpreted as having the following meaning within the game: when an object of type `Object1` collides with an object of type `Object2`, apply the effect `Effect1` to the first object, `Effect2` to the second object, and then add `ScoreChange` to the overall score. The object types are drawn from the following:

- `Player` - A small grey circle controlled by the player using the arrow keys.
- `Red`, `Green` or `Blue` - The three entity types described earlier, governed by their own motion rules.
- `Wall` - Any solid tile that makes up the level layout.

The effect types represent self-contained mechanical concepts that can be generically applied to both the non-player entities and the player itself. The effects do not apply to wall objects, and so have no effect should a rule be generated applying them. These effects are: `Teleport`, which chooses a random empty tile on the map and moves the object to that position; `Kill`, which removes the object from the game, causing a 'Game Over' if that object was the player; `Nothing`, which has no effect but is useful for defining rules which only affect one party.

5.3 Coevolutionary Setup

5.3.1 Species - Level Design

A *Level* is a 20 by 20 array of boolean values, where an entry of `True` indicates that the tile at that position in the map is solid. Solid tiles are represented as black squares in the figures in this chapter.

Generation

Levels are generated by randomly placing single tiles on the grid, with an initial density of 0.4 – that is, there is a 40% chance that a tile will be solid during the initial random generation phase. We prototyped many level generators during this stage of our work [30], and experimented with many different initial densities. 0.4 was found to give a good balance between sparseness and overwhelming density.

Evaluation

Map designs are scored using two metrics which we call *fragmentation* and *domination*. A map's fragmentation score is equal to the number of islands present in the map, where an island is a set of blocks that are adjacent to one another and disconnected from both the wall and other islands. A map's domination score represents the number of tiles which dominate sections of the map. A tile is said to dominate two other tiles if all paths between those two tiles must pass through the dominated tile. A similar definition can be found in [6]. In the vocabulary of game design, a dominating tile represents a doorway or a corridor, i.e. a tile separating two regions of the map.

Calculating which tiles dominate others is costly to do, given that the calculation must be performed on every tile in every map in the population at every generation. We check if a tile is dominating by calculating how many tiles are reachable from that tile in the map. We call this set of reachable tiles a *flood plain*. For a tile, t, the flood plain is defined as:

$$fldplain(t, map) = |\{t_1 \text{ such that } rchable(t, t_1, map)\}|$$

where

$$rchable(p, q, m) \iff \exists \text{ a path from p to q in map m}$$

We then modify the map to obtain map_{mod} by blocking out t and making it inaccessible. We then perform the same calculation for a neighbour of t, t_{nb}, in the modified map map_{mod}.

$$map_{mod} = map[t = \texttt{wall}]$$

Where $map[t = \texttt{wall}]$ represents a remapping of the map such that tile t is

now inaccessible. Therefore, tile t is a dominant tile for map if the modified floodplain is more than one tile smaller than the unmodified floodplain, i.e.

$$fldplain(t, map) < fldplain(t_{nb}, map_{mod}) + 1$$

Since we have only blocked off one tile in map_{mod}, if the floodplain is more than one tile smaller then blocking t must also have blocked access to other tiles. In other words, t dominates at least one tile. In general, fragmentation controls the number of obstacles in a map, while domination controls the openness of the play areas. These map-generation metrics are simple, but include a range of arcade-style archetype maps varying from open arena-based games, such as Pong [62] or variants of Snake[1] with minimal obstacles, to the more labyrinthine maps with maze-like properties such as those used by Pac-Man [90] or Sokoban [60].

$ANGELINA_1$ has two parameters: t_{frag} and t_{dom}, which represent a target amount of fragmentation and domination respectively, to achieve in its generated levels. We describe now the fitness calculation for fragmentation; the calculation for domination is analogous with the replacement of the necessary variables.

$$fragmentationFitness = 1 - \frac{|frag(m) - t_{frag}|}{t_{frag}}$$

This is clamped within the range $[0, 1]$ to avoid negative fitnesses. This calculates fitness as the distance of the fragmentation variable ($frag(m)$ for map m) from the target value. The same calculation follows for domination. The two values are then summed to achieve the overall fitness and discounted according to whether the evolution is weighted more towards one than the other. For the examples in this chapter, we weight the variables equally:

$$fitness = 0.5 \times fragmentationFitness + 0.5 \times dominationFitness$$

[1] No accepted citation for Snake exists, as the game has no definitive version. The reader is directed to http://playsnake.org/ to play the game.

Crossover and Mutation

Levels are crossed over using three approaches: binary AND, in which a square is solid in the child level if the square is solid in both parent levels; binary OR, in which a square is solid in a child level if the square is solid in either parent level; and what we call 'scattered inheritance', where a square in the child is decided as follows:

$$map_{child}[i][j] = \begin{cases} map_1[i][j] & \text{if } rnd() < 0.5 \\ map_1[i][j] & otherwise \end{cases}$$

Where map_1 and map_2 are the parent levels and $rnd()$ is a random number generator. $ANGELINA_1$ uses each crossover technique once on a pair of parents. No mutation is used in this species.

5.3.2 Species - Layout Design

A *Layout* is a 20 by 20 array of integer values, where the value 0 denotes an empty space in the layout, value 1 indicates the starting position of the player (there can only be one of these in a layout) and values 2 to 4 indicate starting positions for entities of the other types. There can be multiple entries of these in the layout, indicating many entities of the same type, or no entries at all if that entity is not in the layout.

Generation

Layouts are generated by randomly placing an amount of the entity values (2 to 4) in the array. The amount of each entity type placed is also randomly selected from an interval set as a parameter to $ANGELINA_1$. In the case of the games shown in this chapter, for layouts generated by $ANGELINA_1$, between zero and 15 entities of a given type were allowed. More than 15 entities tended to result in an overcrowded level which was found to be hard for $ANGELINA_1$ to design rulesets for or evolve good layouts which don't conflict with the level geometry.

Evaluation

Layouts are scored using two metrics: *sparseness* and *volume*. These are defined as follows: a layout is sparse if the average distance between two

NPCs is high. For a layout with n items in it (of any colour) the sparseness of a layout is measured as:

$$2 \times \left(\sum_{i=1}^{n} \sum_{j=1}^{n} dist(i,j) \right) \times \frac{1}{(n \times n - 1)}$$

Where the *dist* function measures the Euclidean distance between two items. Volume is the measure of how many entities there are on-screen. For a map m, its volume is defined as:

$$vol(m) = |reds(m)| + |greens(m)| + |blues(m)|$$

Where $colour(m)$ is the set of entities in map m with colour *colour*. The layouts verify themselves by attempting to apply their layout to the highest-fitness map in the current population, to ensure that they are producing legal, playable layouts – that is, layouts which do not place entities over obstacles such as walls. In this way, the layouts and the maps shape each other's development.

Crossover and Mutation

As with Levels, we use binary AND and OR crossover to combine layouts. To avoid maps exceeding the placement limit for a layout, we then randomly remove entities of each type until they meet the maximum entity placement limit. In addition to these two techniques, for each pair of parents two children are generated through colour set swapping, where a child inherits the entire colour set for a particular entity from the parent. The parent to inherit from is chosen randomly for each colour, and then all entities of that colour are copied into the child.

5.3.3 Species - Ruleset Design

A *Ruleset* is a set of parameters that define certain features of a game, including a list of Rule objects in the form of a 5-tuple, as we described above. A Ruleset contains:

- A Score Limit, which is the target score a player must achieve to win the game.

- A Time Limit, which is the number of seconds before the game ends with a loss.
- A movement policy for each non-player entity (described earlier in the chapter).
- A set of one or more rules, 5-tuples defining interactions between objects.

We have already discussed how rules and movement policies are implemented earlier in the chapter in section §5.2.

Generation

A Ruleset is generated by randomly assigning values to each parameter. In the case of score and time limits, values are randomly generated within a defined interval given as parameters to $ANGELINA_1$. In the case of this chapter, the minimum time limit is 20, and the maximum is 100, while the minimum score limit is 1 and the maximum 100. These values were chosen by playtesting randomly generated games as well as hand-designed games using the same design space as $ANGELINA_1$.

Rules are generated using the 5-tuple template described in §5.2 and randomly assigning each part of the tuple from the possible options. Score changes can either be +1, 0 or -1, also chosen randomly. The number of distinct rules generated is randomly chosen in an interval defined for $ANGELINA_1$. For the examples in this chapter we use 1 as the lower bound on the number of rules, and 4 as the upper bound. A value higher than 4 normally increases the likelihood of a high number of conflicting rules which causes the increase in fitness to slow down greatly, in most cases failing to reach acceptably fit solutions within a reasonable timeframe.

Evaluation

Rulesets are evaluated in two stages. The first checks for pathological situations and then assigns a zero fitness to those Rulesets. Rulesets which had rules in where, for example, the only way to gain score was by the player dying, were penalised in this way. Other penalties were less strong: Rulesets which included rules for entities which were not in the current fittest layout received a discount to their fitness by reducing the overall fitness by 50%.

In deciding on what form these checks should take, we took inspiration from McGonigal's definition [84] of a game's components as a *goal*, a set of *obstacles* and a *feedback* mechanism. In particular, we ensure the games are directed towards a goal using score gain and provide an obstacle to the player through the loss of score or death. These games shrink the state space for rulesets, whilst retaining a core space of interesting rulesets that can be explored through playouts. We also cull the lowest-scoring half of the population and take the remainder through to the second stage, which is a series of playouts.

In total, we simulate eight playouts of each game, one for each behaviour which we list below plus two additional playouts of the final behaviour time, adjusted for different levels of player reactivity. For each playout, we record the score and whether or not the player died. A playout begins by fixing a behaviour for the player-character using AI controllers, and optionally adding in extra constraints to test different aspects of the game. The types of playout are:

- *Empty*, where the player character is removed from the game and the game is executed until the time limit (or, in rare cases, the score limit) is met. This gives useful information on the game's natural tendency. For a game like *Pong*, this would result in a large negative score, while a game like *Pac-Man* would return a zero score.

- *Static*, where the player is included in the game but does not move. This might be considered a step up from an Empty playout. This playout represents a worst-case for player behaviour (assuming the player isn't actively trying to lose).

- *Random Walk*, where the player mimics the NPC entity behaviour of random walk. The player moves through the environment, changing direction at random intervals, until the score or time limit is met.

- *Pre-Collision*, which are playouts identical to Random Walk ones except that the player is collided with one or more types of entity before the first time the game update loop is called. These playouts are useful to compare to random walk as they are the extremes of player involvement in the game. A game like *Pac-Man* yields very useful information from these playouts, as they include a perfect run (colliding

with all the pills completes the level) and a worst-case (colliding with the ghosts ends the game).

- *Long Play*, which are identical to Random Walk playouts but are not limited by time, to investigate how the score and game-over events affect the game when less constrained.

- *Guided*, where the controller analyses the ruleset and detects first-order relationships that would cause score gain, score loss, or death. It then tries to avoid collisions that would cause score loss or death, and plots paths towards collisions that would cause score gain.

This sort of information about the role of the player in the current ruleset is used to filter out broken rulesets by detecting games which the player has little or no effect on, or that are too easily won. This is done in a second round of evaluation, where each ruleset is examined in turn and the score data for their playouts analysed. We expect some or no progression between the Empty, Static and Random Walk playouts, as they represent increasing player involvement and our definition of a game is strongly geared towards the player being the factor affecting the score. It is also useful to compare pre-collision data, as if all playouts of this type tend towards a positive or negative score, this can mean that a game's rulesets are either too difficult or too hard.

It isn't possible to know *a priori* what the theoretical maximum score is for a particular game design, because it can be a combination of various factors that affect the level, layout and ruleset. As such, we calculate the fitness of a Ruleset by considering the proportional difference between each increasingly intelligent playout technique. If we call the six playouts we list above P_n where $n \in \{1...6\}$ in increasing numerical order, then for two playouts P_x and P_y, P_x is considered more intelligent than P_y if $x > y$.

We calculate the *score progression* $sp(x, y)$ as the proportional change in score from P_x to P_y:

$$sp(x, y) = \frac{P_y}{P_x} - 1$$

This means that if the two playouts have the same score, this results in a zero score progress; if the score decreases then a negative score progression is calculated, otherwise positive. This value is clamped in the range $[-1, 1]$,

meaning that if a playout more than doubles the previous score then no additional score progression is calculated. The fitness of a ruleset is then calculated as the average of the sum of score progressions:

$$fitness = \frac{1}{5}\sum_{i=1}^{5} sp(i, i+1)$$

Crossover and Mutation

Crossover of two Rulesets selects a random number of rules from each parent and adds them to the child's rule list. It also randomly selects the other parameters (NPC motion and time and score limits) randomly from each parent as well. In addition to this, we mutate rulesets by randomly changing values in individual rules, and randomly regenerating parameters such as the time or score limit. The mutation rate is 20%, chosen through initial experimentation with the evolutionary parameters. We also introduce new randomly-generated rulesets into the population at a rate of 5% per generation to avoid stagnation, chosen in the same way.

5.4 Example Games

This section includes descriptions of several games designed by $ANGELINA_1$. Two of the games are *full* designs by the system: the CCE system evolved content for all of the species described in this chapter, with no hand-designed content involved. This demonstrates $ANGELINA_1$'s ability to design games, and provides a basic indication that CCE can indeed be used as part of an automated game design system. In addition to these games, we also describe two *partial* game designs produced by $ANGELINA_1$. In this case, some content was provided to the system in place of one of the CCE species, and the remaining content was then evolved in response to this content.

This semi-automated approach was taken to illustrate that CCE's modular structure allows for flexibility in the automated design process, allowing people or other content generators to involve themselves in the process. We also show that by adjusting the level design between the two partial game designs, $ANGELINA_1$ produces two very different game designs as a result.

Some of the games here have titles. Name generation was added during the development of $ANGELINA_1$ – it works by using a Markov chain using

a corpus of arcade game names to generate news ones. The corpus is limited to fairly well-known games from the 1970s and 1980s, since $ANGELINA_1$'s inspiring games are from archetypal games of this era – such as *Frogger*[73], *Pong*[62] and *Pac-Man*[90]. The Markov generator was trained on 2-grams from the corpus of names, although frequently created game names that were very similar to corpus names because of the small corpus size. We rectify this in future versions of *ANGELINA* with much larger corpora and more varied titling. Five randomly-selected examples from the generator: *Revolution*, *Skate Dance Dan*, *Street Fightmare*, *Golden*, *Final Adventure*.

The games shown below have been curated, as indeed have most of the example games presented in this thesis. In §4.6 we introduced the notion of a *curation coefficient*, defining the proportion of a system's output that its creator assesses as being good enough quality to distribute or show as demonstrative of the system's potential. Higher curation coefficients suggest that the system is better at independently assessing its work and creating better output. Through generating and playing a large number of games, we calculate the curation coefficient of $ANGELINA_1$ to be approximately 0.33. That is, one-third of the system's output are games good enough to be shown to people as representative of the system's potential performance. We will assess the curation coefficient of each version of *ANGELINA* presented in this thesis.

5.4.1 Full Designs

Untitled I

Figure 5.1 shows a dexterity-based 'steady hand' game (in which precision movement is the primary test) designed by $ANGELINA_1$. The red objects are static and don't move, but touching them causes the player to randomly teleport around the level and gain a point. Touching the walls of the level causes the player's death. Gaining points increases the difficulty level, as the player must rapidly work out where they have teleported to and avoid touching any walls.

Untitled II

Figure 5.2 shows a Pac-Man-like game designed by $ANGELINA_1$. The player must collect the blue objects while avoiding the moving red objects.

Collecting a blue object gains the player a point and destroys the blue object, while touching the red objects kills the player. This game design is almost exactly the basic format of Pac-Man, although it is missing certain mechanics such as power pills and screen wrapping.

5.4.2 Partial Designs

Revenge

Figure 5.3 shows a screenshot from a partial game design by $ANGELINA_1$. The hand-designed level design was provided to ANGELINA, and the rules and layout were evolved. In this game, the player must avoid touching the red dots, as they kill them. Touching the blue dots increases the player's score. Red dots teleport around the map rapidly, making it hard to avoid them.

After Squad

Figure 5.4 shows a screenshot from a partial game design by $ANGELINA_1$. The level design was provided to ANGELINA, and the rules and layout were evolved. In this game, the player must collect the blue dots by touching them. However, the player can't stop moving once they begin the game, and touching the walls causes the score to drop. The blue dots hug the walls as they move around, meaning the player must be very careful when collecting them.

5.5 Summary

In this chapter, we described $ANGELINA_1$, the first in a line of iterations of the automated game designer *ANGELINA*. This provided a foundation for the use of CCE in automated game design, and showed how a basic type of game – in this case, arcade games – could be broken down into multiple tasks and then solved individually by the species of a CCE system. We discussed the design space, which was then extended into a CCE system composed of three species: Layout, Level Design and Ruleset. We discussed each species in detail, and showed example games produced by the system. This helps to show our basic premise: that CCE can be used to automatically design games, albeit with a curation coefficient that gives much room

Figure 5.1: An untitled 'steady-hand' game designed by $ANGELINA_1$.

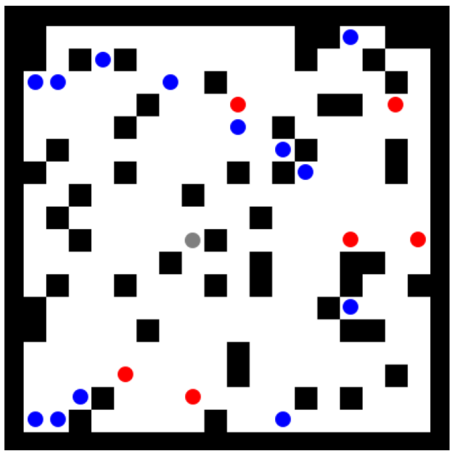


Figure 5.2: An untitled 'Pac-Man-like' game designed by $ANGELINA_1$.

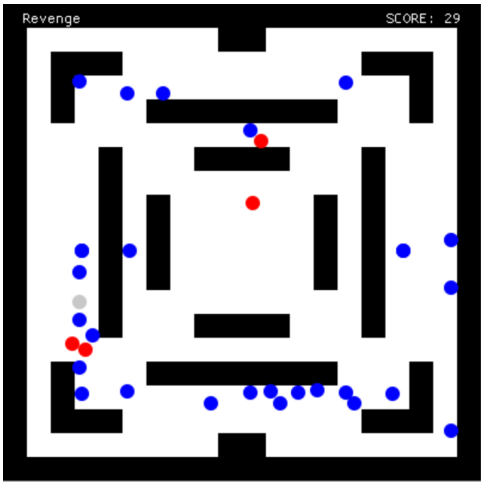


Figure 5.3: *Revenge*, a partial game design by $ANGELINA_1$.

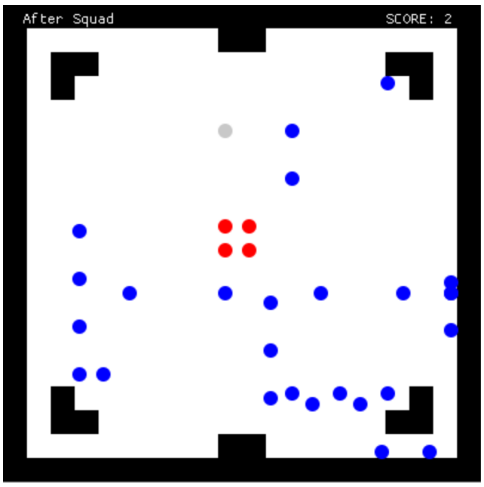


Figure 5.4: *After Squad*, a partial game design by $ANGELINA_1$.

for improvement. This result propels us into the following chapters in which we develop the use of CCE to tackle more specific kinds of problem, and investigate automated game design from different angles, with different engines, in different game genres – all of which we will see in the following chapter, describing the next iteration of the software.

6 Coevolution of Genre-Specific Features

6.1 Introduction

In this chapter we describe *ANGELINA*$_2$, the second iteration of *ANGELINA* with a focus on creating Metroidvania-style platformers. The change in genre necessitates a change in the internal structure of the CCE system, and allows us to focus on a new question: can a CCE system develop games which exhibit very specific genre features, rather than general high-level game-like properties. This chapter will give an overview of the system's architecture and walk through in detail an example game designed by the system.

In §6.2 we discuss the shift in genre and review the key identifying features of Metroidvania games. We extend this into §6.3 in which we discuss the design space chosen for *ANGELINA*$_2$, the structure of the games designed, and also discuss the engine which we use to implement these games in. This includes important information on the structure of the games that we will describe later in the chapter.

In §6.4 we step through the different species in *ANGELINA*$_2$ in order, describing their composition as CCE species in terms of their structure, how we generate new populations, how these individuals are evaluated, and how the most fit individuals are subsequently recombined into new populations. We describe the Powerup Set species, which develops the powers that underpin the Metroidvania-esque features of the games; we describe the Level Design species, which generates the levels the player has to explore; and we describe the Layout species, which places the level start and exit, as well as the player's obstacles.

In §6.5 we give details about *Space Station Invaders*, a game commissioned by the New Scientist and designed by *ANGELINA*$_2$. We will show the

overall design of the game's levels, but also go into detail to illustrate subtle design features that emerge from the interaction between the various CCE species. Finally, in §6.6 we summarise and conclude the chapter.

6.2 Motivation

In the previous chapter, we described *ANGELINA*, a basic system capable of evolving simple arcade-like games using a co-operative co-evolutionary approach. The games produced by this system are functional but generic. Indeed, the term 'arcade game' is a very loose term itself in videogame terminology generally, especially when compared to the more specific genre stereotypes that we described in §2.2.1.

Having established the basic utility of co-operative co-evolution in the previous chapter, in this chapter we turn to the more specific question of whether can we use co-operative coevolution to create games which exhibit specific genre features. We will describe $ANGELINA_2$, a new version of the software which develops the ideas and basic structure of $ANGELINA_1 1$, but refocuses the software on creating a different kind of game with highly specific design features: Metroidvania games.

We described the history of Metroidvania games and their defining characteristics in §2.2.2. Recall the following from our description:

> *Metroidvania games typically situate the player within a large game world, of which only a small percentage is initially accessible to the player... By acquiring items, the player can expand this accessible area, and find further items.*

This is the defining characteristic of Metroidvania games, and it is this property that we aim to incorporate in the version of *ANGELINA* which we describe in this chapter. In doing so, we will demonstrate that co-operative coevolution can be used to evolve games with very specific gameplay features or styles, in contrast to the higher-level design tasks of the previous chapter. $ANGELINA_2$ will also demonstrate how higher levels of detail in the design space of CCE systems can result in emergent design features. This becomes a continuing theme throughout the remainder of the work on *ANGELINA* we are presenting in this thesis.

6.3 Design Space

This version of *ANGELINA* targets the Flixel game engine[1]. Flixel was originally made as a library for ActionScript and Flash by game developer Adam Saltsman. The engine encapsulated a lot of common concepts for 2D game creation in a simple framework, and targeted Flash, a widespread platform that made distribution of games very easy. Flixel became a popular and widespread engine for game creation, and as a result Flixel has been ported to many programming languages. This version of *ANGELINA* produces games using the original ActionScript version of Flixel, while the co-evolutionary system itself is implemented in Java.

The most common way to represent game levels in Flixel is through *tilemaps* which are two-dimensional arrays of integers which Flixel automatically converts into a grid of tiles. A tilemap is parameterised by a number of variables: `drawIndex`, `collideIndex`, `tileWidth` and `tileHeight`. When a tilemap is added to a game, Flixel parses the underlying integer array. If the array contains a value v at co-ordinates (x, y) and if $v >$ `drawIndex` then a tile is added in the game world with the co-ordinates:

$$(x * tileWidth, y * tileHeight)$$

When collisions are processed between the tilemap and other objects in the game world, a tile representing the array co-ordinates (x, y) is considered solid if:

```
tilemap[x][y].value > tilemap.collideIndex
```

We use this feature of Flixel's tilemaps to make unlockable doors in the games produced by *ANGELINA* – the `collideIndex` is increased to make tiles with a lower value non-solid.

As with $ANGELINA_1$ described in the previous chapter, one of the species in the CCE system designs the levels for the games produced by the system. Previously the levels designed were very small: the games we described in §5.4 were just twenty tiles in both width and height. By contrast, a medium-sized game produced by the system we are describing in this chapter is eighty tiles wide and forty-five tiles high. This poses two problems for the system

[1] http://www.flixel.org

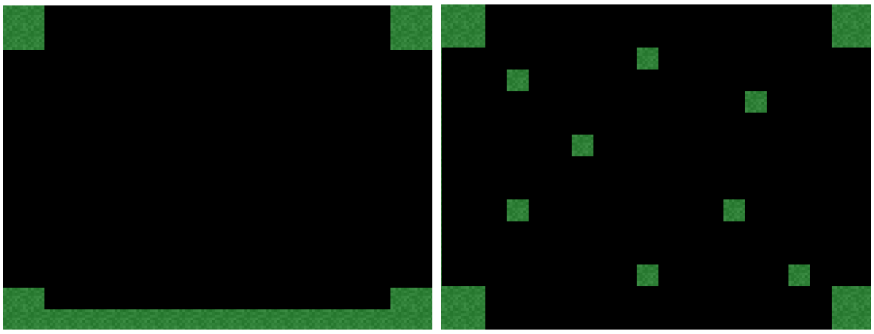


Figure 6.1: Map segments that *ANGELINA*$_2$ uses to compose larger map tiles. Left: a border map segment that has exits left, above and right. Right: a body map segment designed by hand.

if it were to modify maps at the resolution of individual tiles. Firstly, an efficiency problem, since the state space is now considerably larger than before. The second problem is an aesthetic one – levels designed at such a high level of detail don't look very pleasing to players because the system does not have any way of assessing the aesthetics of the levels it designs. As a result, the maps lack the organic feel that game levels often have when designed by a person.

In order to mitigate this, in *ANGELINA*$_2$ we implement a level generation system similar to that used by *Spelunky* which we detailed in §4.4. In this approach, a level is built out of tiles that are the size of a game screen. Spelunky's application of this technique involved a library of hand-designed tiles which had been tagged with the directions the player could exit the tile (those directions being any combination of above, below, left and right). Rather than use this approach, we instead allow *ANGELINA*$_2$ to decide on tile accessibility itself, by composing the tiles itself out of one *body* tile and one *border* tile. Figure 6.1 shows examples of body and border tiles. Body tiles are a collection of hand-designed tile innards. The one shown in Figure 6.1 is a collection of single-block platforms scattered across the tile. The border tiles are a set of every permutation of tile edge being blocked off – Figure 6.1 shows a border tile which blocks the floor off with tiles, while leaving the others open. There are sixteen border tiles in total, including all four sides blocked off making the tile inaccessible, and all four sides open.

The levels are populated with enemies, although *ANGELINA*$_2$ does not

simulate or evaluate combat, and as such the enemies are nuisances rather than true obstacles. The player cannot die, but contact with the enemies causes the player to be knocked backwards, which is often enough to disrupt the player's progress through the level by knocking them backwards off platforms. Simulating gameplay is a major part of the evaluation process for $ANGELINA_2$, as it is with $ANGELINA_1$, and the simulation of real-time combat is particularly difficult. It also complicates how reliable the simulations are, since combat can be resolved in many ways and there are many different ways a player might approach combat situations. This would mean simulating each playthrough many more times in order to balance effectively. The simulation of combat and other real-time elements is mentioned in the evaluation of our approach to *ANGELINA* in later chapters.

6.4 Coevolutionary Setup

6.4.1 Species - Powerup Design

In $ANGELINA_1$, the main influence *ANGELINA* had over the game mechanics was the ability to define rules which governed collisions between in-game objects. However, these rules involved the use of a lot of hand-designed mechanical concepts such as random teleportation or killing. Indeed, even the structure of rules and the notion of focusing on collisions could be considered to be elements of static hand-design in the system. In Chapter 11.3 we discussed the importance of system autonomy in Computational Creativity, and how the perceived creativity of a system is often closely linked to how independent it is and how much control it has over the artefact it is creating. In moving from $ANGELINA_1$ to $ANGELINA_2$, we aimed to have *ANGELINA* rely less on game design knowledge from its programmers.

In $ANGELINA_2$ we change the focus of the system with respect to game mechanics, from designing abstract game rules to designing smaller powerups for the player to collect. The main motivation for doing this is to move away from a mechanics species within *ANGELINA* which relies on hand-designed components, as was the case with $ANGELINA_1$, and instead design a species which can solve a more detailed design problem. This also ties in closely with the genre focus of $ANGELINA_2$ – Metroidvania games – in which items are important gates to play progress [133][132].

Species Definition

The *Powerup Design* species creates *powerup sets*. A powerup set P is a list of powerup items:

$$P = \{p_1, p_2, ..., p_n\}$$

A powerup is a game object with several important properties: an `x` and `y` co-ordinate defining where the object is placed in the world, a `variableTarget` and a `variableChange`. The latter two properties relate to how the powerup changes the game when the player collects it. The `variableTarget` property is selected from a list of hand-chosen game variables which already exist in-code. These are:

- `Player.jumpHeight` - the height in pixels the player is lifted up when the jump button is pressed.
- `Player.gravity` - the downward force applied to the player when they are in the air.
- `Game.collisionIndex` - the tileset value that determines which tiles are solid. Used for simulating locked doors.

The `variableChange` is a value which is applied to the `variableTarget` when the powerup is collected. For example, a powerup with a `variableTarget` of `jumpHeight` and a `variableChange` of 200 has the following method code for being collected.

```
public void collectPowerup(){
    player.jumpHeight = 200;
}
```

When generating a `variableChange` value for a new powerup, the value is randomly selected in a certain range, and that range is dependent on the variable being generated. For `gravity` and `jumpHeight`, the range is between 0 and 500. The variables can't take negative values in this version of *ANGELINA*, so the lower limit is sensible. For the upper limit, values higher than 500 have very little differential between them in terms of their effect on the game.

Generation

The Powerup Design species generates a random number of powerups, between a minimum and maximum limit set by the designer. For the examples given in this chapter, we set the limits to ensure there is at least two powerups (so that the system can assess progression and alternate orderings for collecting the powerups) and at most four, since for values larger than this it is hard for the system to continue designing levels that continuously expand in accessibility. For each powerup, a random variable is chosen from the list above, and assigned to the powerup's `variableTarget` property. The `variableChange` property is then randomly set conditional on the variable's type - the `jumpHeight` and `gravity` variables have a much wider range (between 0 and 500 in the examples given in this chapter) while the `collisionIndex` can only be between 0 and 4 because there are only a few tiles indices in the game. Finally, a random position in the game map is chosen for the powerup to spawn at.

Fitness Criteria

The acquisition of powerups is an important part of a Metroidvania game, as we discussed earlier in this chapter. A powerup set is evaluated by calculating all possible routes through the game. We define a route as an ordered sequence of powerups which the player can collect in a path from the start to the exit. A game may have multiple routes if powerups can be collected in different orders and still reach the exit, although this happens rarely in games made by $ANGELINA_2$ because the levels are relatively small and the number of powerup types is limited.

The fitness is calculated for a powerup set based primarily on the cumulative distance between each powerup across all routes, calculated using A* adapted for side-on platformers (taking into account jump height, gravity and such). The fitness is also reduced based on factors like inaccessible powerups and poor distribution across routes (such as having all powerups accessible from the start of the game). We aim to maximise the travel time between powerups and major game landmarks (the start, or the exit). As such, for each stage in the route we calculate its fitness as a ratio of the distance to the next stage to the length of the map's diagonal. The length of the map's diagonal is not the maximum distance two route stages could

possibly be apart, but it represents a good distance without forcing the co-evolutionary system to overfit the placement of powerups and exits so that they all appear in the extreme limits of the levels.

For a powerup set P, and a set R of routes describing the possible different orders in which powerups can be collected, define $dist(r, i, j)$ as the distance between the i^{th} and j^{th} powerup in the route $r \in R$. Let δ represent the distance between the top-left and bottom-right corners of the map. The distance-based component of a powerup's set fitness is calculated as follows: first, for each route $r \in R$, calculate the route's individual fitness, $f(r)$:

$$f(r) = \frac{\sum_{i=0}^{|r|-1} 1 - (dist(r, i, i+1) - \delta)}{|r|}$$

Average out this for every route possible for the powerup set being evaluated.

$$\frac{\sum_{r \in R} f(r)}{|R|}$$

This gives an average fitness for the routes possible with the current powerup set, indicating average distribution across the map and how good the different ways of exploring and completing the level are. There are also several penalties that can be applied to a powerup set: if the same powerup is included twice, for example, or if a powerup reduces the accessibility, then the fitness is discounted by a fraction. If any powerups are completely inaccessible or not part of any route, the fitness is also reduced.

Crossover and Mutation

When two powerup sets are crossed over to produce a child, one-point crossover is used on the array of powerups, as described in §3.2.3. Mutation is applied at the level of individual powerups within the powerup set, and randomly selects one of the variables within the powerup and changes it. If the `variableTarget` of the powerup changes, the mutation operator also randomly changes the `variableChange` value in case the type is no longer compatible with the variable change (for example, setting the `collisionIndex` to 300). The mutation rate for this is 5%. Another mutation operator is applied to powerup sets with the same likelihood - this operator has a chance to remove, replace or add powerups to a powerup set,

as long as the total number of powerups remains within the bounds set for the system which we described above - between 2 and 4, for the experiments reported in this chapter.

6.4.2 Species - Level Design

In §6.3 we described the notion of a tilemap, how it is implemented in Flixel as a two-dimensional array of integers, and how we use this in $ANGELINA_2$ to design levels using screen-sized map templates. This allows $ANGELINA_2$ flexibility in the kinds of level it can design, while retaining an ordered, organised appearance in the level designs, because the individual components have been designed by hand.

Species Definition

The *Level Design* species creates *levels*. A level L is a two-dimensional array of integer pairs:

$$L = [(body_1, border_1), ..., (body_n, border_m)]$$

A pair (p, q) at $L[x, y]$ denotes a tile at screen co-ordinates ($x \times screenWidth$, $y \times screenHeight$) which is composed of the body tile of index p and the border tile of index q superimposed on one another, as per the description in §6.3. This two-dimensional array of body and border indices represents the *genotype*, since it has to be expanded by looking up the tile templates in a database and copying them into the game's tilemap before the level can actually be played - this final tilemap being the phenotype form of the level design.

Generation

Level designs are generated by randomly initialising the two-dimensional array at a fixed height and width, with randomly-initialised pairs such that the integers in the pairs are within the bounds of the tile database. The array must be the same height and width for all members of the population, because the nature of co-operative coevolution relies on being able to synthesise a full game design using any members of each species. This means that if two maps have different dimensions, combining them with the same

powerup sets may result in some powerups being outside of the game world for some levels, but inside them for others. While we might expect an evolutionary system to be able to solve this problem over time and co-evolve solutions which have the same dimensions, it was not deemed to be an interesting design problem to solve since it offers little impact on the kinds of levels designed, and so we avoid unnecessary additional computing overhead by setting the dimensions of the game level in advance of evolution.

Fitness Criteria

As we have already stated earlier in this chapter, a key feature of Metroidvania games is a slow growth in the size of the space the player can access in the game world. The player should initially have access to a small portion of the world, and then as subsequent powerups are collected the player should be able to access new areas.

Similar to how the fitness of a powerup set is calculated, $ANGELINA_2$ calculates all possible routes through the game in terms of the order in which powerups are collected. At each stage of a route, $ANGELINA_2$ recalculates what fraction of the game world is now accessible to the player. The first stage of the route is the beginning of the level, and new stages begin after the collection of each powerup. For a level design L, there is a set R of routes from start to finish, where $max(R)$ is the fraction of the level accessible after the exit has been reached, and $min(R)$ is the fraction of the level accessible before any powerups are collected. The fitness of L is defined as:

$$\frac{\sum_{r \in R} max(r) - min(r)}{|R|}$$

That is, the difference between the maximally reachable fraction and the initially reachable fraction, averaged out across all possible routes. This is designed to encourage smaller initial fractions, and also penalises game designs which have multiple routes which contain uninteresting possible routes through the game, even if the other possible routes encourage exploration. That is, it encourages a consistency in the degree to which exploration is part of the gameplay, rather than having some routes which rely on exploration, and others which shortcut it.

Crossover and Mutation

When two level designs are crossed over to produce a child, we employ a technique we term *parent-weighted crossover* which randomly selects a parent to be the *dominant* influence in the makeup of the child level design. For two parents p, q, where p is the dominant parent, the contents of the level design array in the child pq at indices (i, j) is:

$$pq[i,j] = \begin{cases} p[i,j] & \text{if } rand() < lim_{dom} \\ q[i,j] & otherwise \end{cases}$$

Where $lim_{dom} \in [1, 0]$ defines the probability that the child will inherit from the dominant parent. For the experiments outlined in this chapter, we use $lim_{dom} = 0.75$. The high value and emphasis on a single parent means that children only gradually change with each generation, which is important in this case because of the structure of the phenotype. Single-point crossover can work (reading from top-right to bottom-left as a single-dimension array) but has the potential to lose contiguous segments of game space. This approach produced better results in pilot experiments, where largely finished game designs were tweaked with information from other designs, rather than being spliced wholesale.

Mutation randomly selects a tile in the level design and replaces it with a randomly-generated tile. The mutation rate is 10%.

6.4.3 Species - Layout

As described in chapter 5, *ANGELINA*$_1$ contained a species which laid out the game's active entities and key objects like the player. This was separate to the solid geometry of the level, as there are many different configurations a single level can take, given the same ruleset but a different layout. *ANGELINA*$_2$ has a similar species working on the same principle. We describe it in brief here for two reasons: its structure is largely similar to that of *ANGELINA*$_1$; and its role in the game is less important in this version of *ANGELINA*.

6.4.4 Species Definition

The *Layout* species creates layout objects containing a two-dimensional array of enemy starting co-ordinates, as well as the co-ordinates for the start and end of the game (that is, the player start, and the exit location respectively).

6.4.5 Generation

Layouts are generated by randomly initialising the co-ordinates for all elements in the layout, within the range set by the map width and height.

6.4.6 Fitness Criteria

The location of the start and exit have a large impact on the evaluation criteria of both the level design and powerup set design species, and so play a large role throughout the CCE system. In terms of their own fitness, we calculate fitness as a composite of several factors, for a layout L represented a single-dimension array of co-ordinates where the last two entries in the array are the start and exit co-ordinates. First we calculate L_v, the proportion of placement co-ordinates (for both the enemies and the start/exit) which are valid: that is, not overlapping with solid space.

$$L_v(m) = \frac{\sum_{l \in L} valid(l, m)}{|L|}$$

Where m is the map, or the phenotype of the level design, currently under consideration, and $valid(l, m) = 1$ if the co-ordinates of point l are empty space in the map m, and 0 otherwise. We also calculate L_d, which reflects the degree to which enemies are evenly distributed across the map. Define max_e as the number of enemies in the most-populated tile, and min_e as the number of enemies in the least-populated tile. We calculate L_d thus:

$$L_d = 1 - (max - min)$$

This parameter is designed to discourage clustering in certain tiles and to evenly spread enemy placements throughout the level. Note that the density calculations do not take into account the placement of the player start or the exit, as we are only interested in the even spread of enemies specifically.

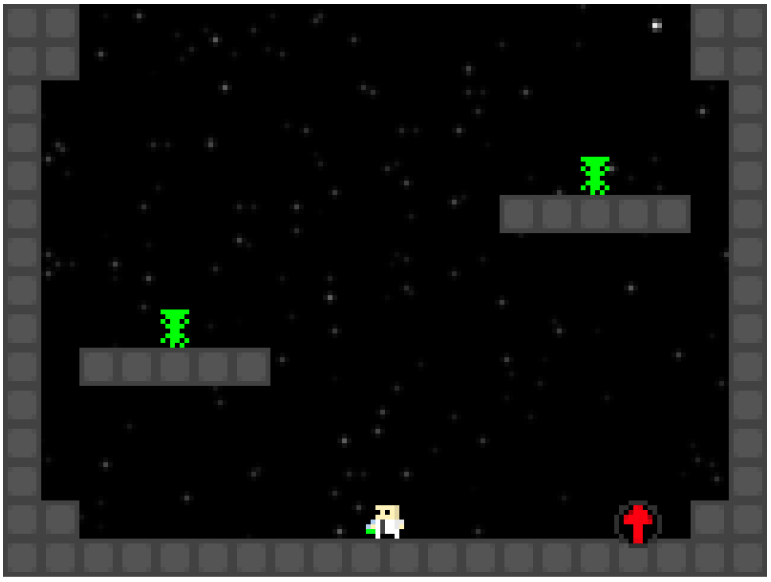


Figure 6.2: A screenshot from *Space Station Invaders*, a game commissioned by *The New Scientist* in 2012.

Crossover and Mutation

In order to perform crossover, one-point crossover is performed on the array of enemy co-ordinates, and then the start and exit co-ordinates are randomly assigned to the child layout from either of the two parents based on a separate random check. Mutation randomly removes, or replaces an enemy co-ordinate, or adds a new one, up to a maximum limit of enemies. It can also randomly generate a new start or exit co-ordinate. The mutation rate is 10%.

6.5 Example - Space Station Invaders

In early 2012 *ANGELINA* was commissioned by the popular science magazine *The New Scientist* to design a game for them [5]. We used the latest version of $ANGELINA_2$ at the time to create a science-themed Metroidvania game (although the theme and visuals of the game were supplied by the author). Figure 6.2 shows a screenshot from this game.

The game is composed of three levels that were generated separately and compiled into a single game. The game levels in all cases are four screens

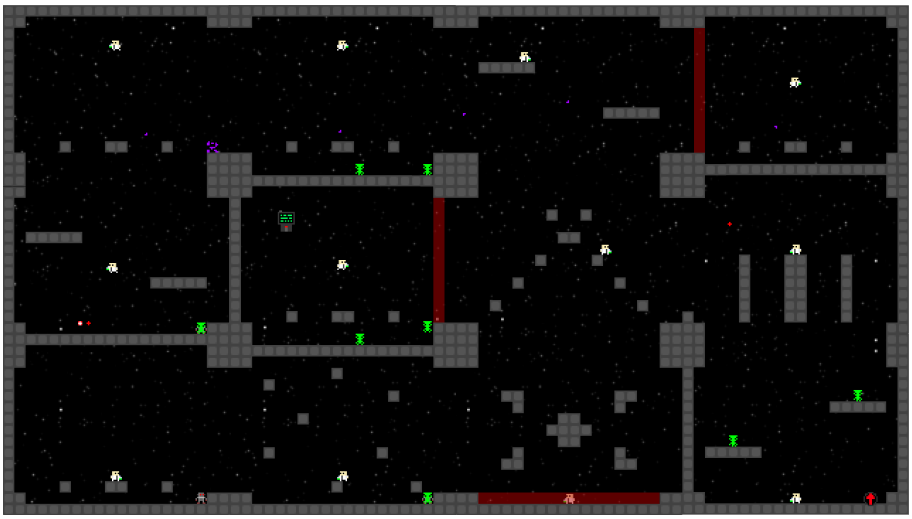


Figure 6.3: A compiled image showing the entirety of Level 1 from *Space Station Invaders*. Red areas are impassable blockades that can only be removed by finding a key.

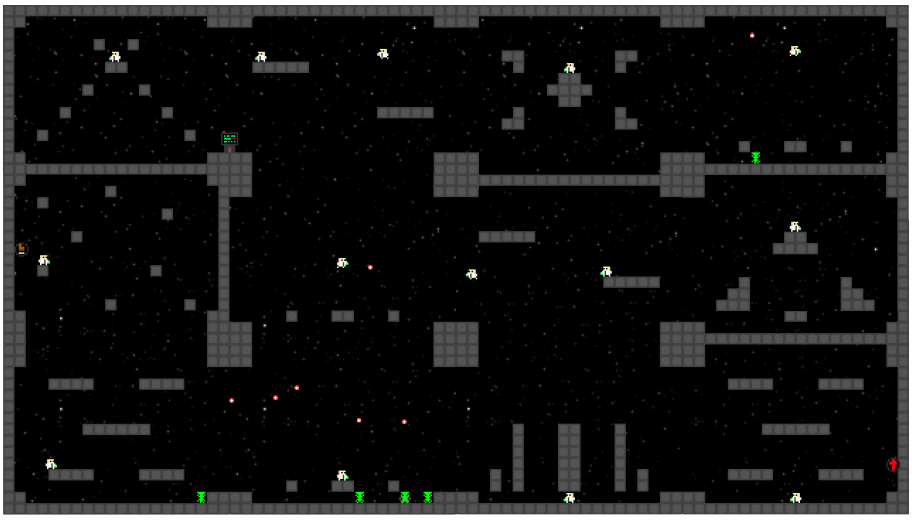


Figure 6.4: A compiled image showing the entirety of Level 3 from *Space Station Invaders*.

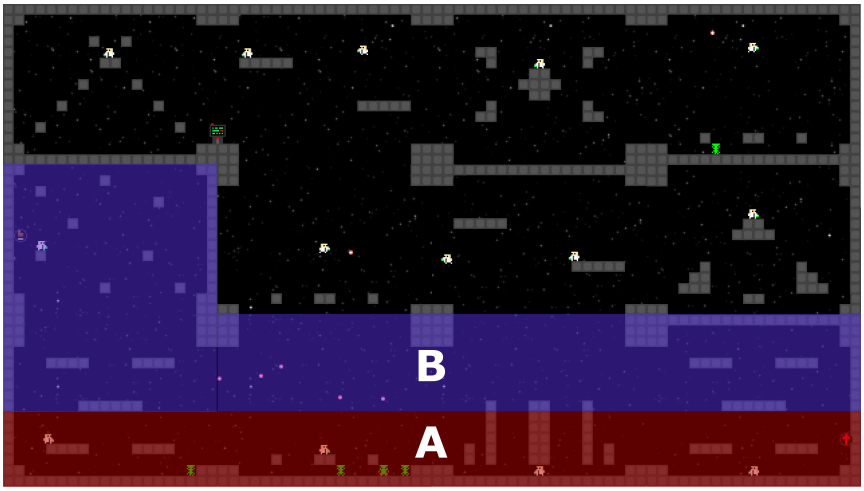


Figure 6.5: An annotated version of Figure 6.4 showing accessibility regions for the first two stages. The red region, labelled A, is the initially accessible region. The blue region, labelled B, is accessible after the first powerup is collected.

wide and three screens high. Figure 6.3 shows a compiled view of the first level for a better idea of the size of the game.

The levels generated for this game have familiar features that are commonly seen in games generated by this version of *ANGELINA* - the player starts near the bottom of the map and initially finds jump powerups which makes reaching the top of the level easier. The later powerups are typically lock powerups which force the player to backtrack to find locked sections that were passed earlier in the level.

Space Station Invaders has some levels which demonstrate interesting properties of the co-evolutionary approach we took. Figure 6.5 shows an annotated version of the level shown in Figure 6.4. The region marked A is the area the player can initially access without any powerups. There is only one powerup in this region: a jump height modifier which increases the player jump height from 100 to 184.

Collecting this powerup increases the part of the map that is accessible, now that the player can reach higher platforms. Now the accessible region is the area $A \cup B$ on the annotated map. The value 184 may seem somewhat arbitrary to the reader: why not 174, or 194, as long as it is high enough to reach higher platforms? Figure 6.6 shows a zoomed in region of the lower

part of the map. The player is standing at the highest point on the lower part of the level (region A in Figure 6.5). The red line in the image shows how high the player can jump with a jump height of 184 from this position. It is just lower than the upper-level platforms.

The preciseness of this value emerged through co-operation between the level designer and the powerup designer. If different tiles were placed in the level, it might be possible to gain more height and jump up to the higher platforms. If the powerup gave any more jump height, it might be possible for the player to bypass part of the level and leap to the next layer. Instead, the player must find a second powerup (in the region marked B in Figure 6.5) to lower gravity enough to reach the upper levels.

This is not only a nice illustration of co-operative coevolution - it also shows the power of giving *ANGELINA* more precision in designing its powerups. We could imagine supplying $ANGELINA_2$ with specific values for each powerup, and only allowing it to select the type of powerup for a given level design. But by allowing free variation in the powerup's associated parameter, we give $ANGELINA_2$ the ability to explore a larger state space, and in turn find more interesting co-operations between the powerups and other species, in this case the level design.

6.6 Summary

In this chapter we described $ANGELINA_2$, the second iteration of the *ANGELINA* system which creates simple Metroidvania-style games using a new CCE system. We began by motivating the work, describing the basic features of a Metroidvania game. The objective with $ANGELINA_2$ was to show that CCE would be able to produce games which demonstrated these key defining features. We described the system's design space, introducing *Flixel* as a game library and describing our approach to constructing 2D levels through a series of overlaid tiles. We introduced three new CCE species which make up $ANGELINA_2$: Powerup Set Design, Level Design and Layout Design. Each of these were described in terms of their structure as an evolutionary system. Then, to conclude, we looked at a large game created by $ANGELINA_2$ and saw how the flexibility and modularity of CCE allowed interesting properties to emerge in the resulting games.

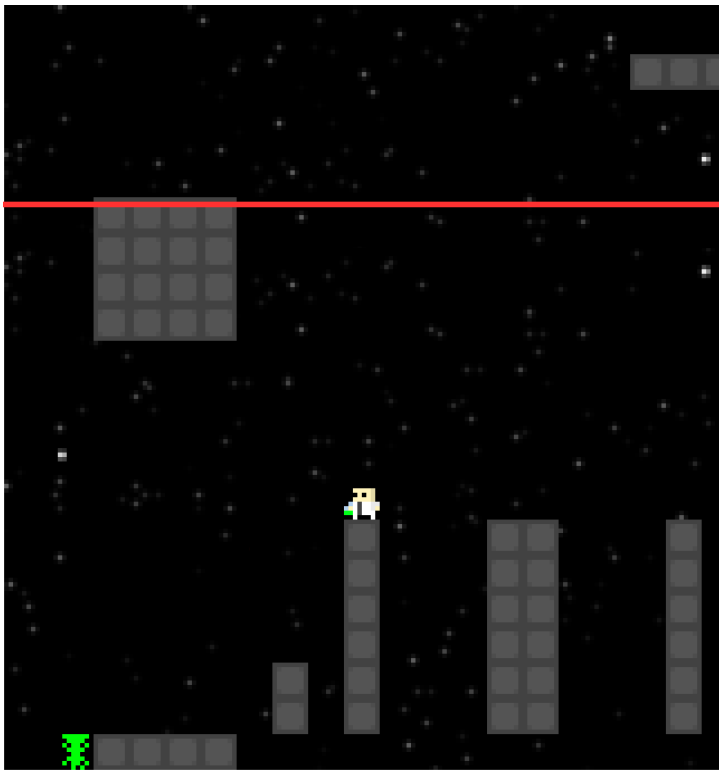


Figure 6.6: A smaller section of the level in Figure 6.4. The red line shows the maximum possible jump height with the first powerup from the player's current position. It is slightly too short to reach the next layer of platforms.

7 Creative Art Direction in Coevolutionary Game Design

7.1 Introduction

In this chapter we introduce *ANGELINA*$_3$ which builds upon the *ANGELINA*$_2$ system that we described in the previous chapter. It marks the beginning of *ANGELINA*'s ability to make games about a real-world topic. It does this in two ways: by adding a new predesign phase to the game design process, in which *ANGELINA*$_3$ prepares a theme and gathers media to use in the game; and by adding a new species to the CCE process which organises the media within the game. We focus on the transition from *ANGELINA*$_2$ to *ANGELINA*$_3$ in this chapter.

In section 7.2 we discuss the reasoning behind the additions made in *ANGELINA*$_3$, particularly from a computational creativity perspective. This leads us into section 7.3 in which we give more specific details about how this motivation affects the space that *ANGELINA*$_3$ works in.

In section 7.4 we describe the two major changes in *ANGELINA*$_3$: the introduction of a predesign phase and the addition of a new species to the CCE system. The predesign phase selects a newspaper article to base the game on, scours the Internet for various kinds of media and additional metainformation from social media, and then collects this ready for the CCE system to take over. We describe all of these processes in turn, and then go through the Media Arrangement species which takes a lot of the output of the predesign phase and evolves appropriate placements for it within the game space.

Finally, in section 7.5 we give three examples of games made by *ANGELINA*$_3$, exhibiting both the variety of the system's output as well as several important features such as the ability to bias the representations of people in the games; the difficulties in dealing with sensitive subject mat-

ter; and how different elements of the newspaper article can contribute to different kinds of media being retrieved. Finally, we summarise the chapter and look forward to the following sections of the thesis.

7.2 Motivation

In chapter 4.6 we introduced several concepts integral to building creative software. One of these concepts was *handing over responsibility* to software as it is developed further. This process often involves carefully considering the division of labour between designer and software in a computationally creative program, and identifying areas that can be given over to the software.

In a lot of practical applications as well as research into generating content for videogames or boardgames the focus is on generating abstract content, for example rulesets or arrays of data describing the layout of game levels. Such content is often generated without knowledge of the context it is being placed in – a level generator for a game about war typically generates levels in much the same way as it might if the game were about romance or politics. $ANGELINA_1$ created games which were highly abstract in nature, using primitive shapes and interactions to describe the game with. $ANGELINA_2$ created many games with visual themes and context, but this was all sourced from its designers rather than from $ANGELINA_2$ itself.

This state motivated us, in designing $ANGELINA_3$, to build a system that was able to take on the responsibility of sound and visual design for parts of its games. We drew further inspiration from work by Krzeczkowska et al. in [76], in which art software generated collages of images based on news articles drawn from the web pages of British newspaper The Guardian [1]. The websites of newspapers serve as useful sources of data for creative systems for a number of reasons. The data is ever-changing, making it both a challenge from a research perspective, since it is less predictable, but also making the results more surprising, potentially raising the likelihood of serendipitous output as well. In addition, newspapers offer data which is high quality and well-organised. Data on the Guardian news pages is tagged and sorted according to the topics involved in the news story, and the text can be relied on to be well-formed, varied in vocabulary, and largely correct

[1] http://www.guardian.com

in terms of grammar and spelling.

As we will see in this chapter, the results of interacting with real-world data and embedding them into the games vastly increases the variety of the system's output, as well as making them more valuable as cultural artefacts, which is important for *ANGELINA*'s acceptance as a game designer in the wider world. The motivation to have *ANGELINA* draw on and use real-world knowledge pays into later work with $ANGELINA_5$ also, a philosophical successor of sorts to the work done in this chapter.

7.3 Design Space

The primary addition to the design space in $ANGELINA_3$ in terms of the structure of the game is the addition of several new kinds of level object which act as a means of presenting sounds and images to the player. Two kinds of objects, image triggers and sound triggers, are new parts of the space $ANGELINA_3$ explores when designing a game. An image trigger has two components. The first is a rectangle defined by a 4-tuple $< x, y, w, h >$ where the first two parameters describe the co-ordinates of the top-left corner of the rectangle and the latter two describe the width and height respectively. The second component of an image trigger is a filepath which references an image to be displayed in the game.

The image is added to the game, stretched to meet the shape of the rectangle defining it. Its alpha value is set to zero, meaning the image is invisible at first. When the player sprite overlaps with the trigger rectangle for the first time, the alpha value is interpolated to the maximum value over a few seconds, fading the image into visibility.

Sound triggers act similarly to image triggers but have fewer parameters: a 3-tuple $< x, y, r >$ where r is the radius of a circle in pixels, and a filepath pointing to the target sound file. The game triggers the sound effect the first time the player comes within r pixels of the origin point (x, y). The sound effect only plays once.

Games now come with a set of image and sound triggers which are added to the game at runtime and triggered by the player as they play. There are also additional parameters which are not evolved by the system and instead are set during a predesign phase, which we explain in a following section. These parameters are:

- A background image, which is displayed at all times behind the level, slightly moving according to a weak parallax-like effect.
- A music file which plays looping in the background of the game.
- The game's title, which is displayed at the start of the game.

7.4 Coevolutionary Setup

7.4.1 Phase - Predesign

In previous versions of *ANGELINA*, the software was entirely defined as a CCE system. The start and endpoint of an execution of $ANGELINA_1$ or $ANGELINA_2$ was the beginning and end of the evolutionary process respectively. $ANGELINA_3$'s design necessitates a new structure which allows for preprocessing in which data can be gathered, filtered and added to the CCE process before it runs. In the case of $ANGELINA_3$, this *predesign phase* is responsible for selecting a Guardian news story to focus on, analysing the news story, and then gathering media to parameterise the CCE system that runs in the main execution phase. In this section we describe this predesign process and the systems at work.

The predesign begins by selecting a newspaper article from the website of The Guardian. It does this by downloading the current headlines at the time of execution, and ranking them according to several criteria. These are:

Story novelty $ANGELINA_3$ records the headlines of any story it has read before, and ranks new stories higher than old ones. This is to avoid repetition in the output of the system.

Tag novelty Each news story has tags attached which define the story's topics or theme. $ANGELINA_3$ keeps a record of seen tags. If it encounters a new tag, it ranks the related story more highly.

Person novelty $ANGELINA_3$ also records any people it detects in news stories it has read. We describe how it does this below. If it detects that a news story includes someone it has not encountered before, it ranks the

story more highly. This is to encourage attention on emerging stories or new characters in the narrative of the news.

Opinion shift *ANGELINA*$_3$ uses Twitter to gauge public opinion on people it detects in news articles. We describe this process below. If it detects a large shift in public opinion about a person featured in a news story it ranks a story more highly. This gives *ANGELINA*$_3$ multiple points of influence (both the news story, and social media) and helps it react to important or controversial events.

The ordering above is least important to most. That is, *ANGELINA*$_3$ will prioritise stories which include a shift in opinion about someone it has a record of. If there are none of this type, it will look for new people, then new tags, and then simply stories it hasn't read. If it finds no such stories, it will randomly select a headline. Ties are broken randomly, so that if two stories contain opinion shifts about a person, ANGELINA chooses one of them with equal probability.

We mentioned two key abilities in the above list: person detection, and public opinion mining. *ANGELINA*$_3$ assesses if a named person in a news article is prominent or not by searching Wikipedia and checking if a page exists about a person with the same name who is currently alive. We found this approach to be effective in determining whether a name referred to someone currently in the public eye or relevant to a news story. When it detects a person it hasn't seen before, it makes an entry in a database file, along with a new value for the current public opinion of the person. *ANGELINA*$_3$ can similarly use Wikipedia to identify countries, using a list of sovereign states.

Public opinion is assessed by querying social media, such as Twitter, for completions of the phrase "$<$ *name* $>$ is...", based on a technique proposed in [130] which Veale term *milking Google*. The words following the phrase are looked up in the AFINN sentiment word list [93], which ranks a list of common words with a sentiment rating in $\{-5, ..., 5\}$, where -5 expresses an extremely negative sentiment, and 5 an extremely positive one. *ANGELINA*3 averages out the sentiment rating of the tweets returning from its search query, and adds this to a running average of opinion that is updated each time the person is encountered in the news.

Figure 7.1: Three results from an example augmented image search of UK Prime Minister David Cameron, to show the variation in outcome. Left, happy. Center, no augmentation. Right, angry.

Once a story has been selected as the topic for a game, the headline, subhead, body text and tags are downloaded to be used as starting points for the next stage of predesign. *ANGELINA*$_3$ then looks for visual and aural media it can use inside the game. For each tag, and for each person or country identified in the article, *ANGELINA*$_3$ can perform searches to extract images from both Google Images and Flickr. The latter is used specifically for images related to countries, which act as backdrops to the games. For people, the searches are augmented with emotionally-loaded keywords as described in [31] according to the public perception of the person recorded in *ANGELINA*$_3$'s database. An example of the results of such augmentation is shown in figure 7.1. A tag may be selected to be a *focused* tag if it meets certain criteria – if a tag is mentioned in the headline or if it is mentioned in the article more than ten times. In this case, the image search will find additional images for this tag, and it will also impact the generated commentary (see below).

Other searches use the article tags unchanged as search terms. The results, along with any pictures of people, form the *image set* that is part of the output of the planning phase. These images are later selected at random to be placed in the final game using the new species in *ANGELINA*$_3$'s CCE system, described below. *ANGELINA*$_3$ also creates a *sound set* by searching sound effect and recording libraries using tags from the article, on

the website FreeSound[2]. This results in a wide variety of recordings, from spoken word to singing, from ambient environmental noise to staged sound effects. Specific selections are made by sorting according to different metrics provided by the site's search engine: in this case we sort by the number of times a sound effect has been downloaded and prioritise the highest first, but we also avoid sound clips that are longer than one minute in length. This is partly to reduce the file size, but also because shorter sound effects tend to be more focused and expressive, which is useful because the player will likely only experience them briefly as they move through the level. The metric used for a particular execution is selected randomly by $ANGELINA_3$ from the site's list.

To complete the audio set, $ANGELINA_3$ downloads a piece of music from the website of Kevin Macleod[3]. Macleod organises his music according to many criteria, including mood. *ANGELINA*3 performs a sentiment analysis on the body text of the sourced Guardian article, using the AFINN database of word sentiments [93] to assess individual words used, and to gain an average sentiment for the article. Using this analysis, the system can select an appropriate piece of music for the game by selecting either a randomly-selected positive mood (such as *bouncy* or *bright*) if the average valence is positive, or a randomly-selected negative mood (such as *dark* or *unnerving*) if the average valence is negative.

Finally, $ANGELINA_3$ generates a title for the game. This is done using two sources of information: first, several corpora of pop culture references were assembled for $ANGELINA_3$ to search through: the Internet Movie Database Top 250 Films[4], the Guardian Newspaper's 1000 Best Albums Ever[5], several Top 100 games lists from major websites[6], and a list of proverbs and sayings[7]. This was combined with code written to access the online rhyming dictionaries RhymeZone[8] and WikiRhymer[9]. Tags, countries and the surnames of people detected in news articles are selected randomly and fed through the rhyming dictionaries. The resulting rhymes,

[2]http://www.freesound.org
[3]http://www.incompetech.org
[4]http://www.imdb.com/chart/top
[5]http://music.guardian.co.uk/1000albums
[6]Such as http://www.gamesradar.com/best-games-ever/
[7]http://www.phrases.org.uk/meanings/proverbs.html
[8]http://www.rhymezone.com/
[9]http://wikirhymer.com/

I was reading the Guardian website today when I came across a story titled 'Obama to urge Afghan president Karzai to push for Taliban settlement'. It interested me because I'd read the other articles that day, and I prefer reading new things for inspiration. I looked for images of United States landscape for the background because it was mentioned in the article. I also wanted to include some of the important people from the article. For example, I looked for photographs of Barack Obama. I searched for happy photos of the person because I like them. I also focused on Afghanistan because it was mentioned in the article a lot.

Figure 7.2: An excerpt from the commentary for the game *Hot NATO*

if any exist, are then matched against results in the pop culture corpus. If any results are found, the rhyming word in the result is swapped out with the tag, surname or country originally used in the search. This creates a pun-like effect where a pop culture reference is related some way to the article. Examples of titles are given later in this chapter in the example games section.

Commentary Generation

In chapter 4.6 we discussed many different ways in which a piece of software can demonstrate creative behaviour and increase the likelihood that others perceive it to be creative. One important part of this is demonstrating an understanding of the context in which its work exists, and being accountable for the decisions it makes when acting creatively. A common way to do this is to provide framing information with the created artefact.

Following on from work on computational poetry in [27], $ANGELINA_3$ records key decisions made during the creation of each game, and creates a 'commentary' which frames part of the creative process. A template is used as the basis for each commentary, with key segments replaced with prepared text depending on the types of decision made, and key data from the news story (such as the names of relevant people). An excerpt from a sample commentary is shown in figure 7.2. The accompanying game is described in more detail below in the subsection '*Sample Games*'.

The decisions recorded for the final commentary are as follows:

- The reason for selecting the news article initially (i.e. which metric was used to decide it).

I was reading the Guardian website today when I came across a story titled HEADLINE. It interested me because ARTICLE-JUSTIFICATION. I decided to make a videogame themed around the story.
I went online to look for resources I could use in my game. I looked for images of COUNTRY landscape for the background because BACKGROUND-CHOICE-JUSTIFICATION. I also wanted to include some of the important people from the article. For example, I looked for photographs of PERSON-OF-INTEREST. I also focused on FOCUS-TAG because FOCUS-TAG-JUSTIFICATION.
I wanted some appropriate music for the game, but I'm not very good at choosing music (I think most of it sounds the same!). However, I made sure to choose something MUSICAL-VALENCE because the story had that feel to it. My thanks to Kevin for allowing me to use his music - you should take a look yourself at www.incompetech.com.
I designed the rest of the game as I would normally. Soon I'd like to experiment with capturing more of the story in the game design itself. For instance, one sentence in the article said that 'QUOTE'. It would be great if I could convey important details from the article in the gameplay, not just the illustrations.
If you play my game, I would really like to hear what you think of it. Email me a short (100 words max) review to angelina@gamesbyangelina.org. One day in the future I hope to use reviews to learn how to become a better designer, so I need lots to practice my reading on!

Figure 7.3: The template commentary filled in by the system. Some of these phrases are conditionally dependent on the game, and so do not always appear.

- The reason for choosing the background image (i.e. whether it was related to a person, or a country being mentioned).
- If at least one person is mentioned in the article, the reason why photos of the person are happy or sad.
- The reason for focusing on a tag (if applicable).
- The reason for choosing the piece of music (i.e. whether the article's mood was positive or negative).

7.4.2 Species - Media Arrangement

As we have described already in this chapter, *ANGELINA*$_3$ goes through an extensive phase of analysis and media acquisition prior to designing its games. When it has collected all of the media it needs to complete a design, it begins evolving the layout, levels and powerups in much the same way as *ANGELINA*$_2$, described in the previous chapter. Unlike *ANGELINA*$_2$, however, *ANGELINA*$_3$ features an additional species for laying out some of the media collected in the predesign phase.

Species Definition

The *Media Arrangement* species creates *arrangements*. An *arrangement A* is defined as two lists of objects, one of image triggers, and one of sound triggers, described earlier in the section *Design Space*.

Generation

Arrangement generation is parameterised by an *asset manager* object which contains references to all the images and sound effects downloaded during the predesign phase. For each sound and picture, a trigger is generated and added to the arrangement object. For images, the (x, y) origin is randomly generated within the boundaries of the map, and the width and height of the image are then randomly generated within a pre-defined minimum and maximum pixel length. In the case of the example games shown in this chapter, the minimum is 32 pixels and the maximum is 118. These numbers are chosen through experimentation - any smaller than 32 and the image is too small to make out, any larger and it is extremely difficult to find

an optimal placement for and therefore pointless trying to generate. The aspect ratio of the image is always kept the same: to generate the height of an image, we first randomly generate the width and then multiply by the appropriate scale factor to ensure we are not skewing the image in any way.

For sound effects there is a similar minimum and maximum range, 20 and 100 pixels respectively. These were chosen as sensible limits to stop sound effects being triggered prematurely or being too hard to trigger, since there is no visual indicator to help the player discover them.

Fitness Criteria

For each image placement, we calculate its fitness as the percentage of the image that is in accessible empty space. This means that fitness is reduced for any part of the image that is covered by solid tiles, or is in inaccessible regions of the level, or is entirely off the edge of the screen. We round the dimensions of the image rectangle up or down to the nearest tile border, which are multiples of eight, and then calculate which tiles match any of these negative criteria, such as being covered by tiles. Call this calculation $inacc(image)$ for an image trigger $image$.

The fitness of an image set, IS, in which the i^{th} image IS_i is w_i tiles wide and h_i tiles high is calculated as follows:

$$\frac{\sum_{i=0}^{|IS|} inacc(IS_i)/(w_i \times h_i)}{|IS|}$$

The process for evaluating sound placement is less precise since we are primarily concerned with whether or not the sound can be triggered; whether the trigger space is partially obscured is less important as there is no visual content that can be blocked. The fitness of a set of sound triggers, SS, is simply the proportion of the set whose trigger circle overlaps with the accessible part of the level.

If we define SS_f as the fitness of the sound trigger set and IS_f as the fitness of the image trigger set, the overall fitness is calculated as a weighted average of the two based on the size of the respective sets, thus:

$$\frac{(SS_f \times |SS|) + (IS_f \times |IS|)}{|SS| + |IS|}$$

Crossover and Mutation

The Media Arrangement species uses one-point crossover to recombine two arrangements into one child. It does this by concatenating the sound and image trigger sets together, and then picking a single point for crossover within that larger set. This process involves two random choices: first, it randomly chooses which parent is selected from first; then it randomly selected whether to append the sound set to the end of the image set, or vice versa. The child inherits placements from the first parent until a randomly-selected point in the set is reached, after which point the child inherits from the second parent instead.

Mutation randomly selects a property from a randomly-selected trigger object (either sound or image) and then regenerates a value for that property. It doesn't affect filepaths, meaning that it will only affect the origin co-ordinates x or y, or the dimensions, width and height for image triggers and radius for sound triggers.

7.5 Sample Games

7.5.1 The Conservation Of Emily

The Conservation of Emily was developed on the 10th May 2012, based on a news article titled *Lord Mandelson confirms he is advising company accused of illegal logging.* The news article discussed Mandelson's involvement with a company who had cut down protected rainforests in Indonesia. The game's title is a pun on a 1964 film called *The Americanization Of Emily*, with the word *Conservation* swapped in because it was one of the tags associated with the news story.

Screenshots from the game are shown in Figure 7.5. Four images were retrieved for use as in-game illustrations, which are shown in Figure 7.4. Clockwise from top-left, the photos are: Peter Mandelson, the subject of the news story, retrieved from an image search for his name; a collage of animals with the words 'help us' written on it, retrieved from a search for the story tag *endangered species*; a photo of a puppy and a kitten, retrieved from a search for the story tag *animals*; and a photo of two people rappelling down a skyscraper hanging a poster which says 'Oil fuels war', retrieved from a search for the story tag *Greenpeace*.

Figure 7.4: Images used in *The Conservation of Emily*

There is only one sound effect in the game - the sound of a man screaming. The music is a fairly dark and slow music track. The game can be played online at

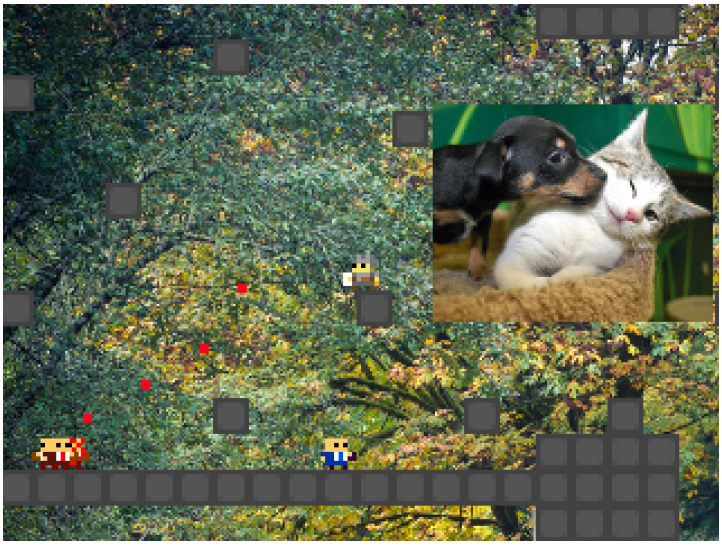
http://www.gamesbyangelina.org/aiide/emily

7.5.2 Hot NATO

Hot NATO was developed on the 20th May 2012, based on a news article titled *Obama to urge Afghan president Karzai to push for Taliban settlement*. The news article reports on discussions between US President Obama and Afghan Prime Minister Karzai about the withdrawal of American and NATO troops from Afghanistan and the handover to local forces. The game's title is a pun on the slang phrase *Hot Potato*, with the military force NATO swapped in, chosen from one of the story's tags.

Screenshots from the game are shown in Figure 7.7. Eight images were retrieved for use as in-game illustrations, some of which are shown in Figure 7.6. The top two rows show photographs of people relevant to the news

(a) A photo of Peter Mandelson in-game.



(b) A photo of some animals. A scream plays when the player triggers this image.

Figure 7.5: Screenshots from *The Conservation of Emily*.

Figure 7.6: Some of the images used in *Hot NATO*

story, in this case Karzai and Obama respectively. Notice how Karzai's photographs show him in more aggressive stances, while Obama is smiling in the photos. This is down to $ANGELINA_3$'s social media analysis which told it that people disliked Karzai but liked Obama, and then the subsequent augmented searches to achieve the happy and angry effects. The last row of images are photographs of Afghanistan, depicting soldiers and helicopters.

There are two sound effects in the game: one is the sound of a very large machine-gun spinning up and firing several volleys of bullets. The other sound is a warning siren. The music is another dark and depressing music track. The game can be played online at

http://www.gamesbyangelina.org/iccc12/hotnato

(a) A photo of Hamid Karzai.

(b) A photo of the Afghani countryside.

Figure 7.7: Screenshots from *Hot Nato*.

7.5.3 Sex, Lies and Rape

Sex, Lies and Rape was developed on the 8th May 2012, based on a news article titled *Rochdale gang found guilty of sexually exploiting girls*. The news article discussed the conviction of nine men charged with child abuse offences in the UK. The game's title is a slightly distasteful pun on the 1989 film *Sex, Lies and Videotape*, with the word *rape* swapped in from the story's tags.

Screenshots from the game are shown in Figure 7.9. Four images were retrieved for use as in-game illustrations, some of which are shown in Figure 7.8. The images represent the themes of family, children and crime quite appropriately. The fourth image, not shown in the figure but visible in 7.9b, was a Renaissance-era painting depicting the rape of Lucrece. While partially a result that came about by chance, its inclusion is nonetheless quite poignant given the game's topic. Many other, less appropriate, images might have come out of the image search.

While this game can be challenging to discuss or think about, it is interesting as an example of $ANGELINA_3$'s output, as it shows how the use of unpredictable source material can lead to the system dealing with scenarios which we might not normally present to the software in ordinary experimentation. It also raises questions about the interaction between people, software and culture, such as whether it is appropriate for a piece of software to comment or reference such a sensitive event. We discuss issues such as this further in a later chapter.

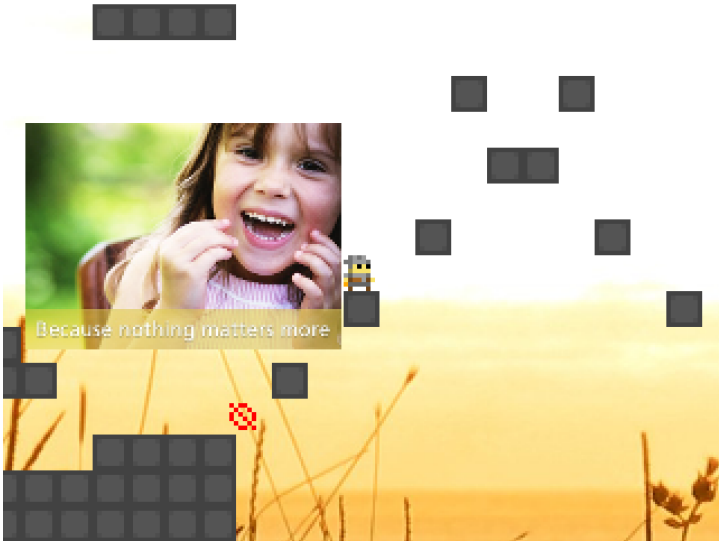
This game has just one sound effect: the sound of a woman singing a children's lullaby in Greenlandic. The music is dark and tense. The game can be played online at

http://www.gamesbyangelina.org/aiide/slar

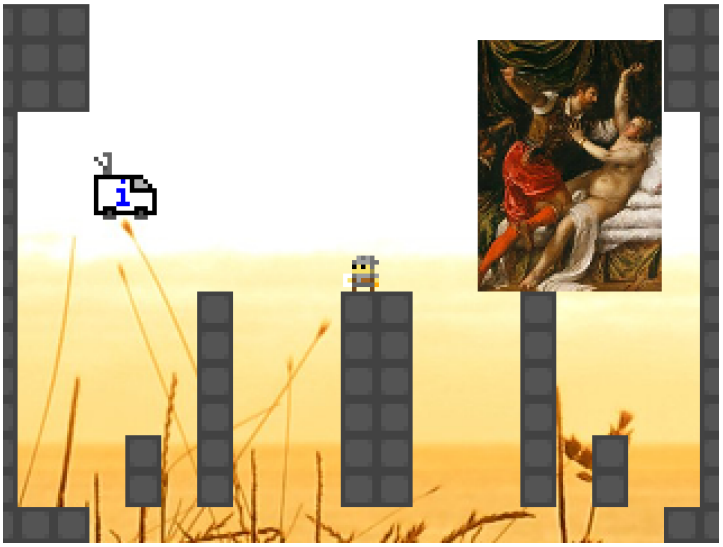
7.6 Summary

In this chapter we introduced $ANGELINA_3$, the first effort to integrate audio/visual content into the design process of *ANGELINA*. We described the motivations for the software to take on more responsibility in the creative process, and outlined how it did this by using data from the Internet, mining media which it then integrated into a design. We described the new

Figure 7.8: Some of the images used in *Sex, Lies and Rape*



(a) The text under the child's photo reads 'Because Nothing Matters More'.



(b) A Renaissance painting, Titian's *Rape of Lucrece*, by the level exit.

Figure 7.9: Screenshots from *Sex, Lies and Rape*.

structure of *ANGELINA*$_3$, using a predesign phase to choose an article for inspiration, analyse it and then acquire images, sound effects and music from the web. We then showed how this was used in the CCE process to create a game. We ended the chapter by giving example games produced by *ANGELINA*$_3$, putting them in the context of the article which inspired them.

In the next chapter we'll be taking forward the idea of adding creative responsibility, examining how we achieve this by increasing the depth of *ANGELINA*'s work, in contrast to *ANGELINA*$_3$ which added more breadth to the process. As we'll see, both approaches improve *ANGELINA* as a designer, in different ways.

8 Coevolution and Reflection-Driven Mechanic Design

8.1 Introduction

In this chapter we describe *ANGELINA*$_4$, the fourth iteration of *ANGELINA* which was build to design puzzle platformer games, which we introduced in chapter 2. *ANGELINA*$_4$ deviates from previous versions in its internal structure and its major contributions. In previous chapters we described how *ANGELINA* was expanded to take on more tasks as a designer, and in doing so we examined how cooperative coevolution (CCE) allowed these tasks to interrelate, as well as showing that evolutionary approaches could work well on many different aspects of the game design task. We describe here how we have eschewed breadth for depth, and present a system consisting of just two generative subsystems which work at much finer levels of detail.

In section 9.2, we discuss the motivation for building such a system, and its relevance as part of *ANGELINA*'s development as a computationally creative system. We refer back to earlier discussions of computational creativity and relate this thinking to the work described later in this chapter. We complete the background for the chapter in section 8.3 by laying a technical foundation, discussing the metaprogramming technique *reflection* which underpins *ANGELINA*$_4$, and describing the design space that *ANGELINA*$_4$ searches in terms of its mechanic generation, which sits at the center of the system's capabilities.

In section 8.4, we give an overview of the evolutionary system in *ANGELINA*$_4$. We describe the two species comprising the system – the mechanic generator, and the level designer – and give a detailed walkthrough of

the simulation algorithm that both species use to evaluate their output. Finally, in section 8.7 we show the results of $ANGELINA_4$, first by describing *A Puzzling Present*, an Android and Desktop game developed using levels and mechanics designed by $ANGELINA_4$, and then by giving a more detailed example of how $ANGELINA_4$ was able to produce results beyond the scope of what its designers thought possible. We link this back to notions of creativity in software.

8.2 Motivation

In §4.6 we introduced the area of Computational Creativity, and discussed ways in which software can be designed to increase perceptions of it being creative. Recall the definition of Computational Creativity from Colton and Wiggins we quoted in §4.6:

> The philosophy, science and engineering of computational systems which, by taking on particular responsibilities, exhibit behaviours that unbiased observers would deem to be creative. [29]

Taking responsibility for something means many different things for a piece of software. It can mean that the software is performing more tasks; it can mean that the software is less constrained in the ways it can undertake those tasks; it can also mean that the software is in charge of evaluating how well those tasks were completed. In progressing from $ANGELINA_1$ to $ANGELINA_2$ the tasks the system was in charge of remained somewhat the same, but the freedom it had in completing those tasks was expanded to allow it to have fine-grained control over certain elements of the game design (such as the parameterisation of powerups). In progressing from $ANGELINA_2$ to $ANGELINA_3$, we explicitly expanded the task set $ANGELINA$ is capable of undertaking, in order to hand over the responsibility of visually theming the game so that the system is in charge of it.

In this chapter, we return to the changes we made between $ANGELINA_1$ and $ANGELINA_2$ with respect to the game's mechanics. By allowing the system more specificity in how it designed powerups for the Metroidvania-like games, we showed in chapter 6 that $ANGELINA_2$ was capable of designing games with interesting emergent properties as a result of co-operation

between different species. However, the way in which *ANGELINA*$_2$'s mechanics were designed still relied heavily on designer input, in order to choose the variables *ANGELINA*$_2$ could affect, and to set the ranges within which the system could act.

In this chapter we describe, *ANGELINA*$_4$, a system developed to reduce the influence of external designer input and to examine the benefits of a system which has more freedom in how and what it chooses to alter in the game code when designing mechanics. As with *ANGELINA*$_2$ and *ANGELINA*$_3$, *ANGELINA*$_4$ designs small parts of the game's mechanics, allowing the player to alter game variables at runtime to affect the game's systems. Unlike previous versions, *ANGELINA*$_4$ uses metaprogramming techniques, explained in the next section, to select variables to modify itself, and the degree to which to modify them. This expands the design space explored by *ANGELINA* from a few hand-picked variables to the entire codebase of the template platformer game, giving it much greater freedom.

There are numerous technical challenges that come with taking such an approach; including how to explore a codebase programmatically; how to simulate gameplay using mechanics not known *a priori*; and, crucially, how to evaluate game mechanics with no background knowledge about the nature of the mechanics being generated. We deal with these issues in turn in the following sections.

8.3 Design Space: Reflective Mechanic Design

8.3.1 Reflection

Reflection is a metaprogramming technique that allows a program to inspect and modify its own code, or that of another program, at runtime. Many modern languages support reflection-like behaviour, including Java and C#. Figure 8.1 shows Java code using reflection to obtain a list of the methods and fields declared within a particular class. Java's reflective capabilities allows for the use of `Field` and `Method` objects to retrieve information about their associated fields and methods at runtime. Fields can have their value accessed or set, including member fields of objects by passing a particular object to the `Field` object, as shown in Figure 8.2.

Metaprogramming techniques can extend much further than this, to method

```
//A sample object of class type 'A'
A anObject = new A();

//A class object representing the class 'A'
Class aClass = anObject.getClass();

//The class object can be queried to get lists
//of declared fields and methods
Field[] fields = aClass.getFields();
Method[] methods = aClass.getMethods();
```

Figure 8.1: A reflection example showing an object's Class being obtained and declared methods and fields being extracted.

```
//For static fields, the parameter to getValue() can be null
Field staticField = aClass.getField("someStaticField");
staticField.getValue(null);

//For member fields, we pass the object
//that we wish to query the the value of
Field memberField = aClass.getField("someMemberField");
memberField.getValue(anObject);

//We may wish to check for additional metadata about the classes
//such as their visibility level or other field modifiers
//These calls return boolean values
Modifier.isStatic(memberField.getModifiers());
Modifier.isFinal(memberField.getModifiers());
Modifier.isPublic(memberField.getModifiers());
```

Figure 8.2: A reflection example showing accesses to field objects and other metadata.

invocation, and in some cases the generation of executable code blocks or the implementation of new classes. *ANGELINA*$_4$ uses Field access only, so to understand how it operates for the purposes of our system the examples in figures 8.1 and 8.2 will suffice.

8.3.2 Mechanic Generation

In this thesis we have discussed the value in handing over responsibility to a piece of software with regards to how a videogame is designed, and with each successive version of *ANGELINA*, we have tried to hand over some kind of new responsibility to the system. However, this has often necessitated the use of intermediate representations to abstract the specific details of how a mechanic is implemented. In *ANGELINA*$_1$ we had premade code segments that encapsulated the notion of an object being `kill`ed or `teleport`ed. In *ANGELINA*$_2$ we targeted a small set of hand-selected variables, such as `jumpHeight` or `gravity` and allowed the system to choose which variable it affected and how it affected them.

ANGELINA$_2$'s generated mechanics were also not very complex in gameplay terms, as they came in the form of powerups which were collected once and then always active. In other words, once a powerup had been collected by the player, the change it made to the game was permanent - the player's jump height changed forever, and so on. In most action games, including the Metroidvanias we drew inspiration from for *ANGELINA*$_2$, at least some of the player's verbs are skill-based in some way. Recall the definition of a player verb from §2.3:

> A *verb* is a special type of game mechanic which is initiated by a player action.

In §2.3 we described the Ice Beam from Super Metroid, which froze enemies into a solid block that could be used as a platform. In order to use this to traverse the world, the player needs timing and precision to freeze enemies in the right place.

There is a broader game design philosophy behind our motivation for generating more complex mechanics too, regarding what makes games interesting to play and interact with. In [94] game developer Jan Willem Nijman states 'you must give the player a reason not to push a button'.

```
//Called once every frame
public void update(){
  if(FlxG.keys.justPressed("SPACE")){
    //A negative velocity temporarily pushes the player up
    velocity.y -= 200;
  }
}
```

Figure 8.3: A sample game mechanic. Pressing spacebar causes the player to jump.

By making all of *ANGELINA*$_2$'s generated powerups permanent, the player didn't have to make any decisions. The powerups never even had negative consequences, meaning that there was no decision whether or not to collect them. In a sense, this makes the task of designing these powerups considerably easier, since the system did not have to consider how the player might use the powerup.

ANGELINA$_4$ tackles both the simplicity of *ANGELINA*$_2$'s powerups and the abstracted nature of the work done so far in all versions of *ANGELINA*. We do away with abstract representations of game concepts like death or teleportation, in favour of a direct manipulation of pure game code. *ANGELINA*$_4$ uses Reflection to examine, modify and execute game code at the same level of detail that a programmer works at.

8.3.3 Toggleable Mechanics

ANGELINA$_4$ generates simple player verbs that are attached to keyboard buttons, such that when a button is pressed, a mechanic is activated in some way. A simple verb might make the player jump when a button is pressed. The code for a jumping verb is shown in figure 8.3, written in Java using the Flixel game engine. *ANGELINA*$_4$ uses a similar template to the one shown in Figure 8.3 to design game mechanics. The major difference is that the generated mechanics can be toggled on and off, meaning that the player can switch between two states in order to change things about the game world and overcome obstacles. For a variable `someVariable` and some method `op` with inverse `invop`, the template code is shown in Figure 8.4.

That is, pressing the spacebar once causes the variable to be changed in some way, and then pressing the spacebar again causes that change to be

```
public boolean toggled = false;

public void update(){
  if(FlxG.keys.justPressed("SPACE")){
    if(toggled){
      op(someVariable);
    } else{
      invop(someVariable);
    }
    toggled = !toggled;
  }
}
```

Figure 8.4: A template for a verb generated by $ANGELINA_4$.

inverted. As an example, `op` might add a fixed number to `someVariable` if it is of a numeric type, while `invop` subtracts the same value. This is not perfectly reversible – if `someVariable` is changed by other code regions between two spacebar presses, then the original state will not be reached by the second press. However, it does offer a two-function verb to the player, and this simple template can capture many mechanics in use in videogames.

Figure 8.4 shows the basic structure of verbs generated by $ANGELINA_4$. The following section will outline in detail how reflection is used to generate these mechanics, and how they fit the structure shown in the figure.

8.4 Coevolutionary Setup

In previous chapters describing the different versions of *ANGELINA*, we described the CCE systems in terms of the species, as the system ran as a tightly-knit set of species which interacted with each other on a regular basis after each iteration of the underlying evolutionary processes. $ANGELINA_4$ is structured differently, using two evolutionary processes which act sequentially. Typically, the mechanic generator precedes the level design, with the latter only beginning execution after the mechanic generator has fully completed evolving a possible mechanic. However, it is also possible to run the process in reverse, with level design occurring first, followed by mechanic design. We discuss this later in the chapter.

The motivation behind this change for $ANGELINA_4$ is that the mechanic

generation is a complex process that requires quite detailed simulation in order to evaluate the resulting mechanics, and as a result is sandboxed in a template level environment in order to provide a consistent way in which to evaluate the generated mechanics. In past versions of *ANGELINA*, the mechanical aspects of the CCE were more akin to tweaking an existing mechanic design rather than inventing something new from scratch. Just as the collection of media content in $ANGELINA_3$ was performed in a predesign phase, we believe that mechanic design is also suited to a predesign phase, in order to have the primary idea in place before the main game design is attempted. We might envision CCE designers then making small balance changes to the mechanic while the game is being designed, without changing the fundamental mechanical idea at work.

In $ANGELINA_4$, first the mechanic is designed during a predesign phase. Then when a mechanic has been found, a series of level designs can be created that use the mechanic. In this section, we will describe both the act of designing and evaluating mechanics, and the resulting process of designing levels for new, unseen mechanics. The result is a system which can generate novel mechanics and then generate levels for them without having any heuristics or ideas about how that mechanic might work. The section that follows will show how $ANGELINA_4$ was subsequently applied to release a videogame based on its creations.

8.5 Species - Verbs

8.5.1 Species Definition

A *Verb*'s phenotype is the information required to create a simple toggleable game mechanic similar to the one shown templated in Figure 8.4. The phenotype consists of three parts: a Java `Field` object denoting the target variable that will be changed; a function that describes a primitive operation such as addition, or boolean inversion; and in the case that the function is binary, the phenotype also includes a value which acts as the second argument to the binary function (such as the value to be added to the `Field` if the function is additive).

8.5.2 Generation

Operation and Value Selection The set of functions the system can choose from include all common Java operations for arithmetic and Boolean types that have an inverse, that is:

- Addition and Subtraction (Binary functions)
- Multiplication and Division (Binary functions)
- Double and Halve (Unary functions)
- Assign (Binary function)
- Invert sign (Unary function)
- Invert boolean (Unary function)

Most of these functions are self-explanatory. Double and Halve are present in addition to the Multiplication and Division, inspired by a quote from game designer Sid Meier, who is quoted by his colleague Soren Johnson in a column in Game Developer magazine:

> **Double It Or Cut It By Half**
> ...When making gameplay adjustments, developers should aim for significant changes that will provoke a tangible response.
> If a unit seems too weak, don't lower its cost by 5%; instead, double its strength. If players feel overwhelmed by too many upgrades, try removing half of them... [66]

Continuing down the list of operations: assignment specifically sets the field's value to a fixed number. The original value is stored and then the inverse of the function reverts the original value back.

In the case of binary functions, the values passed as parameters to the operations for the numeric type fields are randomly generated in a fixed range, similar to how they were generated for $ANGELINA_2$. The range for all numeric fields is $[-500, 500]$, for the same reasoning as we gave in chapter 6:

> For the upper limit, values higher than 500 have very little differential between them in terms of their effect on the game.

Because we are now using a broader variable selection process that has access to the entire codebase, we do now allow the selected values to be negative. However, the upper and lower limits are still in place for the above reason: that values beyond this range have little effect on any game element, and they expand the search space significantly for little gain.

Field Selection

Earlier in the chapter we introduced the concept of Reflection, and explained how it can be used to find representations of fields declared in classes, as well as getting and setting the current value of that field for a particular class or object. We use this technique to select a variable to attach to each Verb.

In order to select a field to target, we first collect all *reachable* fields from the game's codebase. A field F is *reachable* if there is a chain of references starting with a static field that ends with a reference to F. We are limited, in a technical sense, to objects which are referenced as class or instance fields within the game. Local variables are not able to be referenced by the system, because they require the code that references them to be inserted into the same scope, and our analysis of the game codebase is not sophisticated enough to do this – the generated Verb code is always placed in the same place in the codebase, in the `Player` object's `update` method.

Once we collate a list of field objects, we randomly select from this list to generate a verb. If the object is statically referenced, we can create the `Field` object straight away, and store it as part of the Verb population member. The reason for this is that we don't need to pass anything to the `Field.get()` and `Field.set()` methods (recall Figures 8.1 and 8.2). In the case of a field which is at some point referenced by an instance field rather than a static field, we store a reference to the first point in the chain of references that was non-static. For example, the player's `x` co-ordinate can be found through the following reference chain:

```
Registry.gamestate.player.x
```

`Registry` is a class which we can statically reference to access its `gamestate` field. However, the `gamestate` field is set at runtime, so we can't access any

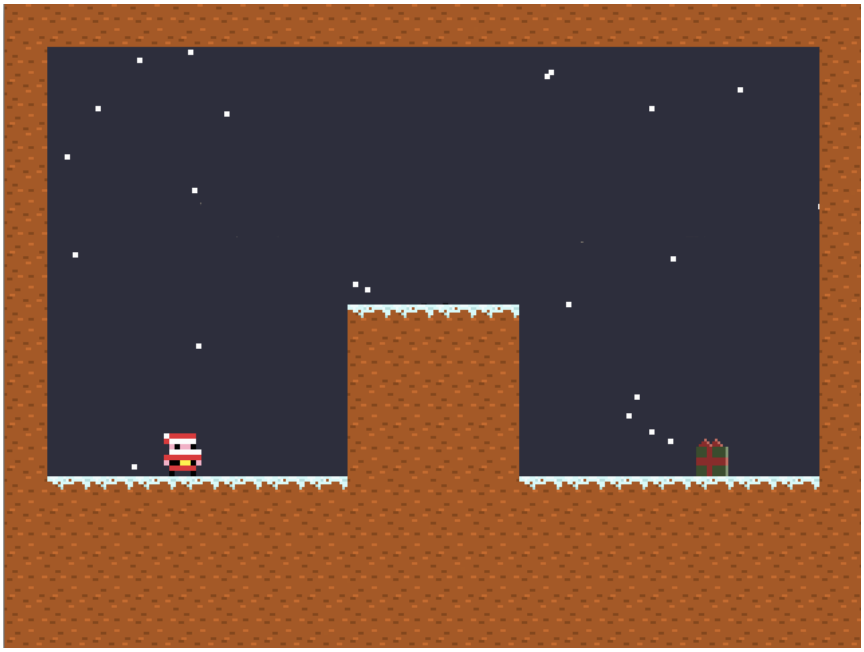


Figure 8.5: A template level used to evaluate game mechanics in *ANGELINA*$_4$.

further down this reference chain until the game is running. To circumvent this, we store a reference to the first non-static field in the chain – `Registry.gamestate` – and then store the remainder of the reference chain as a String which is unpacked at runtime to access the specific instanced field.

8.5.3 Fitness Criteria

Evaluating the quality of a game mechanic is extremely difficult, and no standard formal metrics exist – nor are they ever likely to in an objective sense – that is, in the absence of directly interrogating players to evaluate mechanics directly or machine learn preferences. The quality of a mechanic is often intrinsically tied to subjective notions like enjoyment or satisfaction, which are hard to define in a static fitness function. Mechanics may also be linked to complex concepts like the conveyance of meaning or a message in the game. Researchers have also strongly criticised research which attempts to quantify notions like 'fun' for the purposes of fitness functions or similar [111].

To avoid such criticisms ourselves, while still giving *ANGELINA*$_4$ some way of measuring the worth of a game mechanic, we focus on the concept of *utility*. While there are examples of player verbs in games that serve no utilitarian purpose (e.g. in *Transistor* [49] the player can hold a button to hum a song, for example), on the whole verbs help the player overcome obstacles, particularly in the genre we are working in with *ANGELINA*$_4$, puzzle platformers, which as we discussed in chapter 2 rely on the player verbs manipulating the world to solve problems. Our metric, therefore, can be summarised as considering whether the mechanic helps the player do something they couldn't do before.

To establish this for a given mechanic, we use the mechanic to try to solve a previously-unsolvable level. Figure 8.5 shows a small, simple platformer level. The player controls the Santa sprite, and must reach the present on the right-hand side. In the example shown, this isn't possible because the player can only jump slightly above its own height. *ANGELINA*$_4$ can verify that this level is unsolvable by simulating gameplay and attempting to exhaust gameplay traces, down to a certain level of detail.

8.5.4 Simulation

The simulation process works by exploring a possibility space represented by different keypresses, doing so in a breadth first fashion, and exploring a single keypress combination until nothing new is happening in the game world. Listing 8.6 shows pseudocode for the simulation algorithm, which we will now step through line by line.

ANGELINA$_4$ maintains a list of *game states* that are yet to be explored by the search algorithm. A game state is a representation of the game at a certain point in time. For efficiency's sake, this does not encapsulate the entire game. Instead, we explicitly define what data needs to be stored in a game state. In *ANGELINA*$_4$'s case, we need only store information about the player, but for more complex games we might need to record enemy data, moving platforms or physics objects, logical states like locked doors or completed objectives. Anything that might be affected by the game - or by the generated code - needs to be stored.

Initially, the list of states only contains the starting state of the game. As long as there are still states remaining, the algorithm will keep running – if the algorithm runs out of states without the player sprite reaching

```
while(statelist.hasNext()){
        nextState = statelist.next();
        for(move in moveSet){
                game.loadState(nextState);
                game.applyMove(move);
        }
        if(isLegalState(game) and isNewState(game)){
                statelist.add(game);
        }
        if(isTerminalState(game)){
                return game.trace();
        }
}
```

Figure 8.6: Pseudocode for $ANGELINA_4$'s simulation algorithm.

the exit, it means the level was unsolvable. For each state in the list of unexpanded states, the simulation will try every valid move in a previously-defined *move set*. A move set is a list of button combinations which are valid in the game. This is slightly different from simply pressing every button individually, since this allows combinations of buttons such as pressing jump while moving. However, it is a smaller list than every permutation of button presses – pressing left and right simultaneously can never lead to meaningful progress in the specific template puzzle platformer that $ANGELINA_4$ is considering.

For every valid move set in this list, the game reloads the current state being considered, which sets the game's state to where it was when the state was originally recorded, setting things like the player position and velocity. It then applies the selected move set to the game state. To do this, $ANGELINA_4$ simulates holding down all buttons in the move set, and then lets the game run. While the game is running in this way, *ANGELINA* will periodically take snapshots of the game state at intervals. This interval will determine how accurate, and how slow, the execution of $ANGELINA_4$ is. More frequent snapshots will offer a higher-resolution search of the game space, but generating more of these game states will cause the search space to expand rapidly, delaying termination.

When a snapshot is taken, $ANGELINA_4$ will consider whether the snap-

shot's state of the game is *old* or *new*. There are two conditions to be met that make a game state old. Another state must already have been seen before which is identical to the current snapshot within some resolution. For $ANGELINA_4$, this means we compare all the game state data like the player's position, velocity or acceleration and see if they are less than a certain delta from any previously-seen state. This delta can be set to different values to affect the detail level (and by extension, the accuracy and computational complexity) of the search. Through experimentation we found that 4 pixels – half the size of a level tile – was a good tradeoff delta value. This means that a fairly comprehensive search is performed, without the search being intractable on an average development machine.

If we find any states that are similar, we check the second condition. The second condition is that the move history of the current snapshot must be *subsumed* by this older state. Each state contains a list of the moves that were made to reach this point in the simulation, and each time a new state is made, it concatenates the current move onto the list of the parent state. The previous state's moveset, M, is subsumed by the current state's moveset M' if the first n moves in M' are identical to M, where M is n moves in size. In other words, if we have already reached this state previously with the current history of moves, then the snapshot is not considered new. Otherwise, it gets added to the state list for consideration.

As an example, if the player moves left, then right, then left again, it will eventually reach places that it reached before simply by moving left. If $ANGELINA_4$ detects that the game is in the same state it was previously, and it got there by using fewer buttons, it won't bother adding a new game state to the list. Additionally, we may also check if a state is *legal*. *ANGELINA* checks that no exceptions have been thrown, for example, to catch cases where mechanics are generated that crash the underlying game engine.

$ANGELINA_4$ keeps taking snapshots and simulating the current move's button presses. The distance between each snapshot is set by a parameter in the simulator, and can be used to approximate a kind of reaction time for the system, since the snapshot is also the point at which the simulator can enter a new move from the moveset. The longer the period between snapshots, the more time it takes the simulator to press another button, somewhat simulating a player who does not react as quickly as one with a

shorter snapshot interval.

The system stops taking snapshots once no progress is being made. This is assessed by comparing snapshots against other snapshots taken for this move set. If the system reaches a snapshot that has already been taken for this move set, we assume that the move set is no longer changing things in the game (perhaps the player is running into a wall, or stuck in a loop of jumping on the spot) and $ANGELINA_4$ stops considering this move. It then moves on to simulate other move sets in the list until no more are left, and then continues onto the next state in the state list.

If, when simulating a move, the player reaches the objective – in our case, they overlap with the exit object represented by the wrapped Christmas present in Figure 8.5 - the state is considered to be successful, and the move history of the current state is recorded as a solution to the level. At this point, $ANGELINA_4$ stops simulating. This simulation process can easily detect that the level shown in figure 8.5 is unsolvable, as it eventually exhausts its list of world states without reaching the present.

In order to evaluate a particular mechanic, $ANGELINA_4$ adds the code for the mechanic to the game's codebase, using reflection, and then extends the set of move sets used by the simulator to include pressing the button for the new verb, as well as pressing the button in conjunction with jumping and moving. $ANGELINA_4$ can then run the simulation again on the previously unsolvable level, this time using the new mechanic as one of the buttons that can be pressed.

The fitness of a mechanic is 1 if the simulation manages to reach the level objective – thereby completing the level. Otherwise, the fitness of a mechanic is the percentage of the level that the mechanic made accessible. The reasoning behind this decision is that as the mechanics enable the player to access larger portions of the level, and as this portion becomes larger the mechanic ultimately allows the player to reach the exit. However, this biases the evolutionary process towards mechanics which affect reachability and positioning. We can imagine a game mechanic that doesn't help the player move much further than before, but might allow very specific changes to the game world, such as destroying blocks in the scenery to allow the player to pass through walls. This is an area of the system that could be further developed in future iterations.

Solvability

Note that when we say *ANGELINA*$_4$ 'exhausts' its list of world states we do not mean that *ANGELINA*$_4$ truly proves that the level is unsolvable – only that it exhausts the search space defined by the parameters of the simulation. The two variables that most strongly impact this are the interval between snapshots, which affects how often the game is examined by the simulation, and the delta measuring similarity between states, which affects how often the search branches new nodes to explore. Setting these parameters too high or too low can transform the simulation from a trivial exercise to an intractably dense search. Our delta setting of 4 pixels and snapshot intervals of around 2-8 snapshots per second (varying to represent different reaction times) resulted in manageable solution times (under a second for a 300 tiles^2 level).

However, it is possible that solutions exist which require a higher snapshot frequency, or that require the search to branch on lower-resolution delta values. We believe this likelihood to be extremely low, particularly for the kinds of mechanic that *ANGELINA*$_4$ ultimately ended up generating – in fact, we encountered no mechanics which had unintended solutions not known to *ANGELINA*$_4$ through the simulation process. It is worth noting that playtesting in commercial game development often fails to find extremely detailed flaws in their games, resulting in 'glitches' which often involve extremely precise 'pixel-perfect' movement on behalf of the player to exploit. These glitches often appear in the speedrunning gamer subculture. Having encountered no problems in the use of *ANGELINA*$_4$, the potential existence of solutions missed by the simulator therefore only contribute towards a known challenge within game development, rather than being a failing unique to *ANGELINA*$_4$.

8.5.5 Crossover and Mutation

Crossover is difficult in this species because the structure of the phenotype can be very different depending on the type of the `Field` being targeted. All numerical types are compatible with one another (such as `Double` or `Int`), but `Boolean`s are not compatible with numerical types. In the case that two numerical or two Boolean types are crossed over, we create a child verb that has the `Field` target from one parent, and the operation and value

from the other parent, randomly selected between the two available. In the case that a numeric type is crossed over with a boolean type, we create a new child with the `Field` target from the numeric type, and a unary sign-invert function for the operation. While not a perfect solution, it makes the system flexible enough to deal with the full set of primitive types.

When mutating a verb, one of two mutations can take place: the root class of the `Field` object can be searched again to find a new field to replace the current one. This kind of mutation changes the specific field being altered, but the search is local to the class already being selected, which means the change is somewhat localised to a particular segment of the codebase. The second mutation keeps the `Field` unchanged, and instead randomly reassigns the operation and the target value of the mechanic. The mutation rate is 15%, a figure which we settled upon after performing some experiments with no mutation at all, and a variety of increasingly high rates. Small amounts of mutation were found to be useful without stopping the system from converging on stable results.

8.6 Species - Level Design

In the previous section we described how mechanics are generated and evaluated using a template level. Searching a codebase for fields to modify, and then modifying them automatically at runtime, leads to many complications, and *ANGELINA*$_4$ commonly causes exceptions to be thrown and errors to occur because it is modifying code without an understanding of what might cause issues at runtime, such as null pointer exceptions or array indexing exceptions. Evaluating mechanics generated in such a way is therefore extremely difficult, but is made possible because of the clear-cut evaluation offered by the template level. We generate a mechanic, and use a static level to test it. The existence of a solution for that level acts as a witness to the quality of the mechanic.

In this section we tackle an equally difficult problem: how can levels be generated for a newly generated mechanic? Automatic level generation is a common concept both in games research and commercial game development, but such level generators are often built with a large degree of domain knowledge supplied by the game designer, such as the hand-designed templates in Spelunky's level designer, discussed in §4.4. For puzzle platformers,

levels are normally only solvable through the specific use of the mechanics the game is built around. Knowledge about these mechanics can normally be built into level generators, but in the case of *ANGELINA*$_4$ we can't provide this knowledge, because the mechanics are being generated by the system. The conflict, then, is between the need to embed domain knowledge, contrasting with the fact that we don't know the domain a priori, because mechanics are being generated for the first time.

We overcome this by inverting the premise of the previous section: instead of evaluating generated mechanics against static levels, we take a mechanic and use it to evaluate generated levels.

8.6.1 Generation

In *ANGELINA*$_2$ and *ANGELINA*$_3$ we generated levels using hand-designed templates. This was partly because of the large scale at which the levels were being generated, as well as a concern that the resulting levels would lack a feeling of organic design. For *ANGELINA*$_4$, the levels being designed are much smaller - the size of a single screen, compared to sixteen times that for some of the levels generated by *ANGELINA*$_3$. This is partly to accommodate the increased computational complexity in the simulation process, however it also allows *ANGELINA*$_4$ to generate levels on a per-tile basis. To retain an overall sense of structure, rather than having *ANGELINA*$_4$ place single tiles, we allow it to place primitive shapes: horizontal and vertical lines, and filled and unfilled boxes.

- A line is defined by five parameters: x and y co-ordinates, a Boolean indicating whether the line is horizontal, an integer length, and another Boolean indicating whether the line is made of tiles or of player-killing spikes.
- A box is defined by four parameters: x and y co-ordinates, how long the side of the box is (boxes are square, so only one parameter is needed) and a Boolean indicating whether the box is filled with solid tiles or whether only the perimeter of the box is solid.

Levels are randomly generated by starting with an empty level and then adding a number of elements to the level, randomly selected between a minimum and maximum number of elements (2 and 20 respectively for the

experiments described in this chapter). A level can therefore be represented in genotype form as a list of shape primitives, using only their parameters to define the level. The level's phenotype is represented as a two-dimensional array of integers. For a level L with tile width in pixels of t:

$$L[x, y] > 0$$

indicates that the tile at screen co-ordinates $(x \times t, y \times t)$ is solid, with a zero value in the array denoting empty space.

8.6.2 Fitness Criteria

As we discussed in the beginning of this section, the process of level design inverts that of mechanic design. We apply the same simulation process to a generated level, this time using a particular mechanic that we wish to employ as the target criteria for this level design. We simulate attempting to solve the level first without any additional mechanics, then again with the new mechanic added. For a level L and move set M, $solves(M, L)$ indicates that the level L can be solved with the move set M. We write M_d to indicate the default move set used by $ANGELINA_4$ which includes moving and jumping, and we let m be a new mechanic.

If $solves(M_d, L)$ holds, then we assign zero fitness to the level, since we can solve it with the default ruleset and it thus does not require the invented mechanic at all. If neither $solves(M_d, L)$ nor $solves(M_d \cup m, L)$ hold, then the level can't be solved either way and also receives zero fitness. Otherwise, assuming that $solves(M_d \cup m, L)$, $ANGELINA_4$ tries to evaluate how well this level fits the given mechanic.

There are three metrics this species is trying to optimise in the case that a level passes the initial solvability test:

- How long a solution is in terms of pixels travelled on-screen ('path length').
- How many times the new mechanic is used in the solution ('mechanic use').
- How many actions, including the mechanic, are used in the solution ('discrete actions').

In all three cases, the fitness is proportional to the absolute distance between the target value for the metric and the measured value in a given solution. For example, if the target number of actions is `targetActions` and the actual number measured in a simulation is `measuredActions`, then the fitness is calculated as:

```
Math.max(1 - act, 0)
```

where

```
act = Math.abs(targetActions - measuredActions)/targetActions
```

Hence, if the target number is met, the fitness is 1, otherwise it scales towards zero but never goes below zero fitness. The total fitness is a weighted sum of the three metrics; in the example runs in this chapter, we weight each metric equally, i.e. each contributing a third to the overall fitness.

An important additional note about the fitness calculation is that *ANGELINA*$_4$ uses the least fit solution from a level as its final fitness. When simulating the level, instead of terminating when a solution is found, it records the number of actions in the solution and then continues simulating until all paths of that length have been exhausted, adding any additional solutions to the output. It calculates the fitness of all of these and then returns the least fit one as the fitness. The reasoning behind this is that the search process sometimes finds more convoluted solutions with the same path length before it finds simpler ones, and since simpler solutions often imply a lower fitness (because they don't use the mechanic as much, or are much shorter paths). Hence *ANGELINA*$_4$ searches for all solutions of this length before ruling on the level's fitness.

8.6.3 Crossover and Mutation

Crossover of a Level object is performed by using one-point crossover on the level space itself, reading left-to-right and top-to-bottom across the two-dimensional array representing the level. This means that a point is picked in the level space, and one parent contributes any shapes which begin before this point, while the other parent contributes shapes which begin after this point. A shape is considered to 'begin' at the point defined by its x and y co-ordinates. In the case of all shape primitives used by *ANGELINA*$_4$, this is the top-left corner of the shape, so this makes sense for the top-to-bottom reading we use on the array.

Mutation of a level object randomly selects shapes from its list and

changes their properties. It can alter the starting co-ordinates, as well as flipping booleans such as the horizontal/vertical indicator on Line objects, or the Boolean indicating whether a line is solid ground or player-killing spikes. Elements can also be randomly added or removed, as long as this does not take the number of shapes above or below the limits set by the species at generation. The mutation rate is 20% – as with the mutation rate for verb design, we settled on this rate after some experimentation with the level designer. The figure is not intended to be particularly precise; rather, some mutation is beneficial, and we found 20% to be a good resting point.

8.7 Sample Games and Mechanics

8.7.1 A Puzzling Present

A Puzzling Present is a puzzle platformer game, developed in Java and released in December 2012 for Android and Desktop platforms. It featured three 'worlds', which were collections of levels that all shared a common theme. Each world contained ten levels which were designed to be solved using a particular mechanic. The three world mechanics, and the thirty levels in the game, were all generated by $ANGELINA_4$. The game was covered by the gaming and technical press, including features on *Ars Technica*, *New Scientist*, *Engadget* and *Gamasutra*, and reached the Android Top 500 games. To date, it has been downloaded over 15,000 times.

A Puzzling Present, or APP, was also designed to act as a survey to assess some of the aspects of $ANGELINA_4$ as a designer of both mechanics and levels. The levels had been designed with three different difficulty tiers in mind, based on the reaction times needed to complete the level, which $ANGELINA_4$ assessed according to the simulation done during the level generation process. At the start of the game, the player was asked if they were willing to participate in post-level surveys about their experience. If they consented, the level order was randomised to reduce bias introduced due to learning effects or fatigue, and after each level, the player was presented with the screen shown in figure 8.7.

The mechanics generated were as follows:

- Gravity Inversion - when toggled on, gravity would change from a downward force to an upward one, propelling the player towards the

Figure 8.7: Post-level survey from *A Puzzling Present*

ceiling. Toggling again would reverse the change.

- High Jump - when toggled on, the player's jump height was doubled. Toggling it again halved the jump height.
- Bounce - when toggled on, the player character became rubbery and would bounce off surfaces, slowly gaining height as it did so. Toggling it again reverted to a non-bouncy state.

Example Levels

Invert Gravity Figure 8.8a is a level generated for the Invert Gravity mechanic. The player begins in the bottom-left corner, underneath the holly, and must reach the present which is just above. The player can move around the edge of the screen without much trouble, inverting gravity at the far right of the screen to reach the top. When the player reaches the point shown in the screenshot, there is a harder task to perform to complete the level: the player must fall towards the holly and then invert gravity in mid-air to fall upwards towards the present.

Bounce Figure 8.8b is a level generated for the Bounce mechanic. The column of holly between the player and the exit is too high to jump over, and the horizontal holly blocks prevent the player from performing a running

jump over the gap. In order to clear the holly, the player must jump off the starting block, activate bounce, and then bounce off the floor to gain enough height to clear the holly.

High Jump Figure 8.8c is a level generated for the High Jump mechanic. The level involves two jumps which require the mechanic to be active: one jump to reach the first level of blocks, by jumping at the far left of the level, and a second jump to reach the exit at the very end. A third jump needs to be made to cross a portion of holly halfway through the level, although this can be done with a normal jump mode activated.

8.7.2 Surprise and Emergence

In chapter 6 we showed that the modular construction of the CCE system in $ANGELINA_2$ allowed for subtle game features to emerge as the different evolutionary systems co-operated with one another. We attributed this not only to the CCE's structure, but also the level of detail afforded to the individual species in designing their target components. In $ANGELINA_4$, we have made another step forwards in terms of handing over creative responsibility – now the system is capable of searching through a game's codebase and identifying interesting variable changes itself.

We have already shown through a description of *A Puzzling Present* that this can generate an interesting variety of game mechanics, and discussed how its use of elasticity properties exceeded the knowledge of its designers, demonstrating the strengths of our more general approach. Below, we go into more detail on another surprising output from the system.

Code Exploitation

Consider the following mechanic, which we termed *teleportation*, generated by $ANGELINA_4$:

(a) A level generated for the Invert Gravity mechanic.

(b) A level generated for the Bounce mechanic.

(c) A level generated for the High Jump mechanic.

Figure 8.8: Three levels from *A Puzzling Present* demonstrating the three different mechanics in the game.

```
if(FlxG.keys.justPressed("X")){
  if(toggled){
    Registry.player.x += 70;
  } else{
    Registry.player.x -= 70;
  }
  toggled = !toggled;
}
```

This mechanic moves the player along the x-axis by a short distance either left or right depending on whether the mechanic is currently toggled on or off. This mechanic appears to be quite simplistic, and is less flexible than teleportation-like mechanics as they typically appear in games. Normally such mechanics allow the player to select where they end up, or have a larger range.

After generating this mechanic, $ANGELINA_4$ was tasked with designing several levels which used it. When designing a level, $ANGELINA_4$ logs a sequence of actions that serve as a solution to the level, so that we are able to understand the routes the system is using to solve each problem. It logs these as a sequence of button presses, which in conjunction with a screenshot of the level is normally enough to understand what has happened. When generating levels for the teleportation mechanic, however, the solutions did not immediately make sense to us as we reviewed the results. Often, levels would involve scenarios such as the one depicted in Figure 8.9 in which the player has to scale a large height to reach the exit, despite the new game mechanic apparently only allowing lateral movement through the level.

Investigating further, we found that $ANGELINA_4$'s solution to these levels involved exploiting a bug in the core template game written by us. The following code snippet approximates the code used for jumping in the template game:

```
if(FlxG.keys.justPressed("SPACE")){
  if(Registry.player.isTouching(FlxObject.FLOOR)){
    Registry.player.velocity.x -= 100;
  }
}
```

When the player presses the jump button, the game checks to see if they

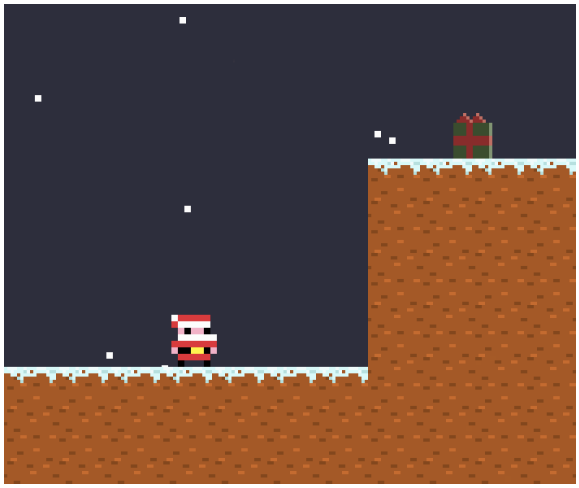


Figure 8.9: A segment of a level designed by $ANGELINA_4$. The player must climb up a high cliff face to reach the present. However, the current mechanic does not appear to allow them to do so directly.

are touching the floor. This is done using Flixel's built-in collision engine, and we simply use a flag to say we want to know if there are collisions below the sprite (the 'floor'). If this check passes, then the player is pushed upwards.

Figure 8.10 shows the effect of using the teleportation mechanic while near a wall. Because the generated mechanic has no conditions or checks on it, moving the player's x co-ordinate forces them inside a wall. We might intuit that this would leave the player stuck inside the wall; however, because $ANGELINA_4$ simulates gameplay by exhaustively pushing buttons, it was able to discover that the player can continue to jump upwards while inside a wall, because the collision system registers the player as standing on a floor tile. Repeatedly jumping upwards from the state shown in Figure 8.10 propels the player upwards until they reach the top of the wall, where they can safely reach the exit.

This is not what we might consider to be the intended use of the mechanic – it relies on a secondary effect within the game (a weakness of the jumping mechanic), an effect that is not documented as part of the core game's system either. It's also an emergent property of the mechanic with certain kinds of levels – this can't be used as a technique in all levels, only those

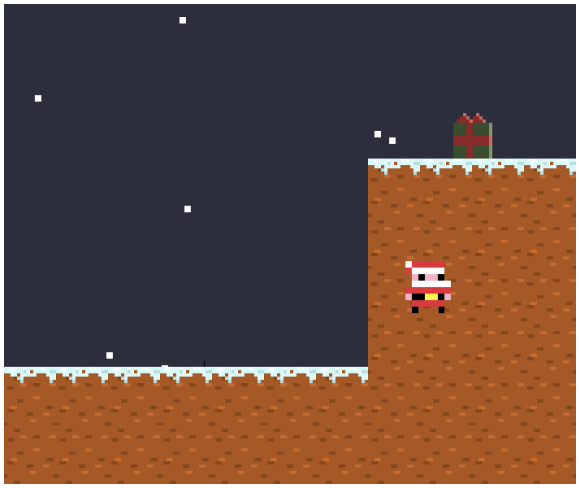


Figure 8.10: By teleporting inside the solid wall, the player can exploit an oversight in the game's jumping code, allowing them to jump directly inside the wall.

which involve the scaling of walls. This is an interesting result for two reasons: it had no heuristics or mechanics to guide its use of this mechanic in the first place, since it generated it itself; and secondly, it was dependent on both the mechanic *combined with* pre-existing game systems that it had no control over.

We discussed the value of surprise and novelty in assessments of computational creativity in chapter 4.6, and the discovery and use of the teleportation mechanic certainly represents $ANGELINA_4$ surprising its creators.

8.8 Summary

In this chapter, we described $ANGELINA_4$, a system capable of inspecting, modifying and executing Java code using metaprogramming techniques in order to generate and evaluate game mechanics. We showed how evaluation against a simple template level provided a good baseline to separate useful mechanics from useless ones, and then reversed the process to design levels by generating and testing them against mechanics. We described the evolutionary structure of $ANGELINA_4$, and then walked through an example game – *A Puzzling Present*, an Android and Desktop game built using mechanics and levels generated solely by $ANGELINA_4$.

9 Coevolutionary Game Design In The Wild

9.1 Introduction

In this chapter we describe $ANGELINA_5$, the fourth iteration of *ANGELINA* which incorporates some of the work from each of the previous versions, and advances the software to a new stage in its development where it can engage with communities of developers by entering game design contests. $ANGELINA_5$ was historically the first piece of software to enter a game jam and compete with other people in designing a game.

In section 9.2, we provide the motivation behind $ANGELINA_5$'s design and the direction it takes the project. We discuss the importance of engaging with a community of other creatives, and how we hypothesise it affects the perception of a piece of software as a creative entity. In section 9.3, we build on this motivation by describing in detail the technical groundwork for $ANGELINA_5$ and the design space that the system explores when designing games.

In section 9.4 we outline the structure of $ANGELINA_5$ as we have in previous chapters, including details of a preliminary design phase that prepares the system for designing a particular game, similar to that which we described for $ANGELINA_3$ in chapter 7.

In section 9.5, we describe the concept of a game jam, and in particular we detail the *Ludum Dare* game jam, a popular game making contest which we have entered $ANGELINA_5$ in multiple times. We contextualise game jams in the broader culture of games, and then in section 9.6 describe the games $ANGELINA_5$ ultimately ended up entering into Ludum Dare. Along with an outline of each game, we also detail the response from the Ludum Dare community and the rankings the games received after a process of peer review. This informs later discussion of *ANGELINA*'s place in modern

games culture later in the thesis.

9.2 Motivation

So far in this thesis we have seen work motivated by the aim of producing software capable of designing games. This has been tackled on both technical and artistic levels, considering issues such as the independence of the software, its freedom in discovering new concepts, its ability to use and understand elements of culture and the real world to express things. However, the act of designing games is only one part of being considered a game designer. The broader aims of *ANGELINA* as a project are to investigate how a piece of software can come to be accepted as a member of a creative community. This means more than just the act of creating – it means creating in certain contexts, understanding communities, and interacting with other people (and potentially software) who make up that community.

The most important motivation guiding the design of $ANGELINA_5$ was a desire to have the system take its first steps towards integrating itself with a community of game developers. Until $ANGELINA_5$, most people's interactions with the system were through playing games it had made, normally games which had received media coverage in the specialist games or science press. Encountering *ANGELINA* as a curio rather than a game designer, we hypothesised, affects how someone perceives *ANGELINA* as a system, and by extension the games that it produces.

Our intention with $ANGELINA_5$, therefore, was to have the system proactively engage with both game developers and game players, seeking exposure in the same channels that a person might when starting off as a game designer. A common way for game designers to improve their skills, meet other designers and get people to play their games is to enter game jams. Game jams are short contests in which entrants design games based on a given theme in a short time period. In order to enter a game jam, $ANGELINA_5$ needed to be designed in a way that would enable it to interpret any kind of word or phrase as a theme. Engaging with the theme in the game's design is important for submitting a serious entry to the jam, and being able to understand it on a basic linguistic level is the first step in doing this. We discuss both this process, and the basis of $ANGELINA_5$ as a CCE system, in this chapter.

We took *ANGELINA*$_5$'s entry to a game jam as an opportunity to examine people's reactions to the system as well. There are reports of both negative and positive bias associated with people responding to computationally creative software, some of which we have explored in section 4.6. Entering a game jam for the first time, with a version of *ANGELINA* that no-one had encountered before, allowed us to enter multiple games from the system and assess the different reactions to them from people both in the presence of, or absence of, an explanation of *ANGELINA* and how the game was made. These reactions, as well as the surrounding responses to *ANGELINA* as a system, will be important in the ongoing development of the system, which we discuss as Future Work in chapter 12 later in the thesis.

9.3 Design Space: 3D Exploration Games

ANGELINA$_5$ is built as a plugin to the Unity game development environment[1]. Unity is an extremely popular, versatile and powerful game engine that ships with a comprehensive development environment that is also easily extended by writing scripts that act as plugins to Unity itself. These plugins can interact with both the engine itself as well as the tools, and can modify Unity's user interface to add additional menus. We use this to somewhat customise Unity into having a frontend that allows for control of *ANGELINA*$_4$ more easily. Unity games can be deployed to web browsers, to all major desktop operating systems as native applications, to every modern games console and handheld device, and most smartphone operating systems including iOS, Android and Blackberry.

ANGELINA$_5$'s domain is 3D exploration games, although the domain is general enough to be expanded to many other more established game genres, such as first-person shooters. *ANGELINA*$_5$'s games consist of a 3D maze, in which several entities may exist, with different effects on both each other and the player. The primary objective is to reach the level's exit, but there may be secondary tasks the player can complete, such as collecting objects. Figure 9.1 shows three screenshots from a sample game. The top two are screenshots from the game as seen in the Unity editor. The maze-like organisation of the game can be seen clearly here, accentuated by

[1] http://www.unity3d.com

the first-person perspective (the bottom screenshot) which means that walls obscure the player's view of the rest of the space.

The levels are constructed from pre-designed tiles that are slotted together in a grid, similar to the system described in chapter 7 and the work of Spelunky, described earlier in §4.4. Unlike *ANGELINA*$_3$, each tile is assigned to a *zone*. Zones are defined by zone templates, which are collections of media that zones have in common. A zone defines:

- A texture to display on the floor tiles.
- A texture to display on the wall tiles.
- A 3D model file to display as scenery (where applicable).
- A sound effect file to play as ambient noise in the zone.

Zones are generated after a *predesign phase* has acquired media assets to choose from, which we describe in the following section. Some media is chosen at random, while others may have specific reasons for their selection depending on the knowledge *ANGELINA*$_5$ has about the theme or the game it is designing. We detail this selection criteria in the following section also.

The motivation behind such a design is to give structure to the individual level sections that players pass through. A zone template with birdsong sound effects and forest/tree textures might convey a natural area, whilst concrete and car sounds might convey a city. This expands *ANGELINA*$_5$'s design space to be able to treat locations as an abstract concept. In the past, *ANGELINA* has used the acquired media only if direct relationships to input material (such as the tags on a newspaper article in *ANGELINA*$_3$ - see chapter 7) could be established. This means that many more abstract creative decisions, like deciding on a location for a game to be set in, are hard to encode into the system. The introduction of zones, much like the use of free-text input themes, is part of a bid to make *ANGELINA*$_5$ a more general, less specialised version of the software.

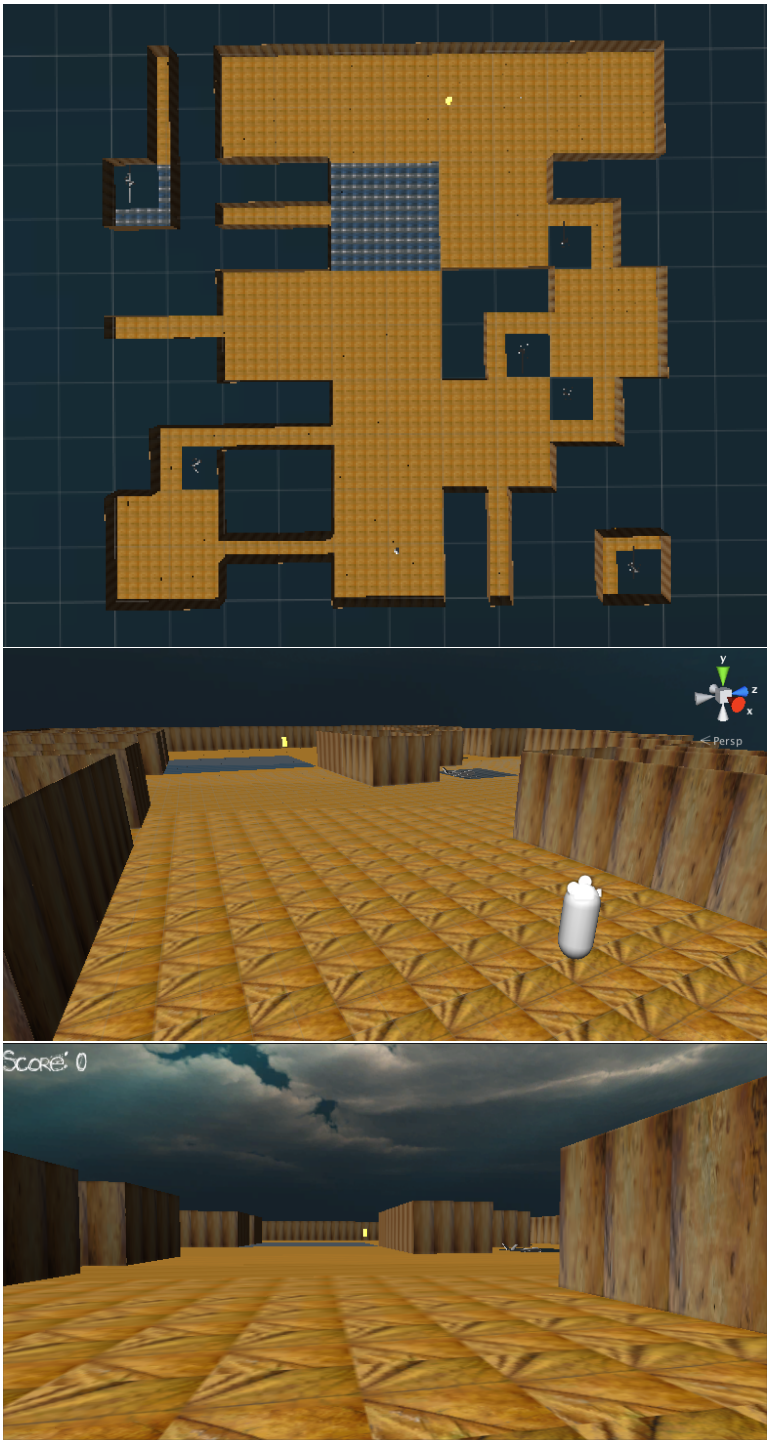


Figure 9.1: Screenshots from a game made by *ANGELINA*$_5$, showing an in-editor view of the level (top) an in-editor close-up of the game world (mid) and finally an in-game shot (bottom).

9.4 Coevolutionary Setup

9.4.1 Predesign Phase

As with *ANGELINA*$_3$, *ANGELINA*$_5$ has a predesign phase which it uses to prepare a theming for the game prior to executing the main CCE design process. In *ANGELINA*$_3$, the system began by scraping news articles from the front page of The Guardian newspaper. This version of *ANGELINA*, however, works by receiving a phrase or word which it uses as its theme. This is to match the setup of game jams, which we give details of later. If the inputted theme is a single word, then the predesign phase progresses using that word as a theme. If the theme is longer than a single word, *ANGELINA*$_5$ will analyse the theme to try and extract the most promising single word out of it.

Firstly, it looks up each constituent word in the phrase in a frequency database of English words [72]. It ranks words according to their frequency, culling any which are too common or too rare (more than 10,000 or less than 50 mentions in the database, an interval we set after hand-curation and experimentation). It then randomly chooses one of the remaining words as a candidate theme. If none of the words match this interval, it will take one of the more common words (a frequency of more than 10,000) and look the word up in a word associations database. It then randomly chooses one of these words and uses this as a theme. The word association database was observed to provide more specific instances of a word in most cases, and so selecting from this list often retrieves a theme that is usable. Finally, to test that the chosen word – whether taken from the original theme or from the word association step – is appropriate as a theme for a game, *ANGELINA*$_5$ performs several test searches on its media databases, which we list below, to see if it can find results for different kinds of media and linguistic data. If it can't find any results in one or more of its searches, it returns to word association to select another candidate theme. If no theme can be found this way, *ANGELINA*$_5$ simply stops – although this is rare, as word association provides many dozens of words of all kinds, including common nouns which normally produce good results in searches.

Once a theme has been selected that meets the search criteria, the system begins acquiring the media it needs to perform the main CCE design. Before doing so, it creates a small set of associated words by performing another

Figure 9.2: Title screen from *Cat That*, a game designed during prototyping $ANGELINA_4$. Note the cat-themed font used in the title.

word association lookup with the final theme, and selecting the first ten associations. These words are used as secondary search terms for many of the visual and aural media searches performed by the system, and they can help add variety to the content retrieved.

As with $ANGELINA_3$, media is acquired from many different sources. The system first obtains a font and a piece of music to accompany the game. It does the former by searching DaFont[2] for public domain or GPL-licensed fonts whose name or description includes the theme word. While simplistic as an approach, this often returns highly relevant fonts that add theme and atmosphere to the title screen. Figure 9.2 shows a game themed around cats, with a font that includes paw prints in the letters.

Music is selected by first associating emotions with the theme word, or words related to it. We do this by using a secondary function of a web service called *Metaphor Magnet*[3], a product of Computational Creativity research described in [130]. Metaphor Magnet uses Google N-grams to build a database of common phrases, which are used to construct metaphorical relationships, by extracting patterns of speech such as 'as X as a Y'. Metaphor Magnet stores such relationships and uses them to understand the connections between concepts. One of the other things it has mined from the N-grams is emotions people express in response to things, from patterns such as 'X makes me feel Y'. This data can be mined to under-

[2]http://www.dafont.com
[3]http://ngrams.ucd.ie/metaphor-magnet-acl/

stand common feelings expressed about a concept. *ANGELINA*$_5$ uses this to find out what emotions are most commonly associated with the primary theme word.

Once it has a primary emotion (for example, people feel *intimidated* by thieves) *ANGELINA*$_5$ tries to connect this emotion to a musical theme. As before, we use the Incompetech database of music as a source of background music for *ANGELINA*$_5$'s games. Incompetech provides emotional tags for all of its songs, but the list is limited to the following:

> Action, Aggressive, Bouncy, Bright, Calming, Dark, Driving, Eerie, Epic, Grooving, Humorous, Intense, Mysterious, Mystical, Relaxed, Somber, Suspenseful, Unnerving, Uplifting

Metaphor Magnet's emotional range is far broader than this, because it mines natural language from the Internet. To resolve this, we perform a narrowing process to map the found emotion to one of Incompetech's. For each emotion in the music database, we perform a search using DISCO[4], a semantic similarity tool for assessing how alike two words are in meaning. The database emotion with the strongest similarity to the discovered emotion is taken to be the target for the search, and we randomly choose a piece of music with this emotion from Incompetech's database.

ANGELINA$_5$ then searches for 3D models and sound effects. The latter search is performed similarly to the searches done for *ANGELINA*$_3$ – we use the FreeSound database and search using both the primary theme word and its association set. We sort by average rating, and download sound effects which are less than sixty seconds in length. We download multiple sound effects for each theme word, so that they can be randomly chosen from during the game design process. For 3D models, we use TF3DM[5], a database of free 3D models. *ANGELINA*$_5$ searches the database for .OBJ format models, as at the time this was the only file format that could reliably be imported into Unity. Since the design of *ANGELINA*$_5$, Unity's support for other model formats has improved considerably.

In order to create a title for the game, *ANGELINA*$_5$ uses a combination of rhyming dictionaries and a corpus of popular culture terms, in a similar approach to the one used in *ANGELINA*$_3$. To the existing corpora of film

[4]http://www.linguatools.de/disco/disco_en.html
[5]http://tf3dm.com/

and music titles, we added a list of common English idioms. *ANGELINA*$_5$ searches for words which rhyme with the primary theme word, or one of its associations, and then searches for idioms or other cultural phrases which include the rhyme, swapping it out with the theme word to create a pun. All of the game names in this chapter are named through this process.

This concludes the online part of the media acquisition. As we described in the Design Space section of this chapter (§9.3), *ANGELINA*$_5$ uses what we call zones to divide the game level up into differently-themed areas. Each zone has certain scenery objects associated with it, as well as sound effects and texture files. Unlike the other media used in its games, *ANGELINA*$_5$ obtains the texture files locally from a large corpus of textures. These textures are only identified by the name of the folder each group of textures is contained within, such as *wood*, *grunge* or *green*.

In the event that the theme or its associations match any of these folders, the textures inside are selected for use in the zone designs. However, this tends not to happen often as the folder descriptors are not always very accurate, and the words chosen do not cover all of the associations the textures themselves might have. To mitigate these problems without researchers personally annotating the textures (which could possibly reduce the perception of *ANGELINA*$_5$ as an autonomous system), we instead enable *ANGELINA*$_5$ to acquire additional labellings of each texture itself, by periodically uploading a texture to Twitter and asking its followers to suggest descriptions of textures. Figure 9.3 shows a screenshot from *ANGELINA*$_5$ asking about a texture, and some of its responses. *ANGELINA*$_5$ searches this catalogue of tags using the primary theme and the association set, and adds any matching textures to a list from which *ANGELINA*$_5$ will later make a selection to add to the game. If there are no textures found, or not enough to fully texture the game, it will choose randomly from the texture pool.

9.4.2 Species - Level Design

Species Definition

A *Level* is a two-dimensional array of tiles selected from a catalogue of predesigned tiles. A tile itself is a two-dimensional array of integers, where each entry in the array is one of the following:

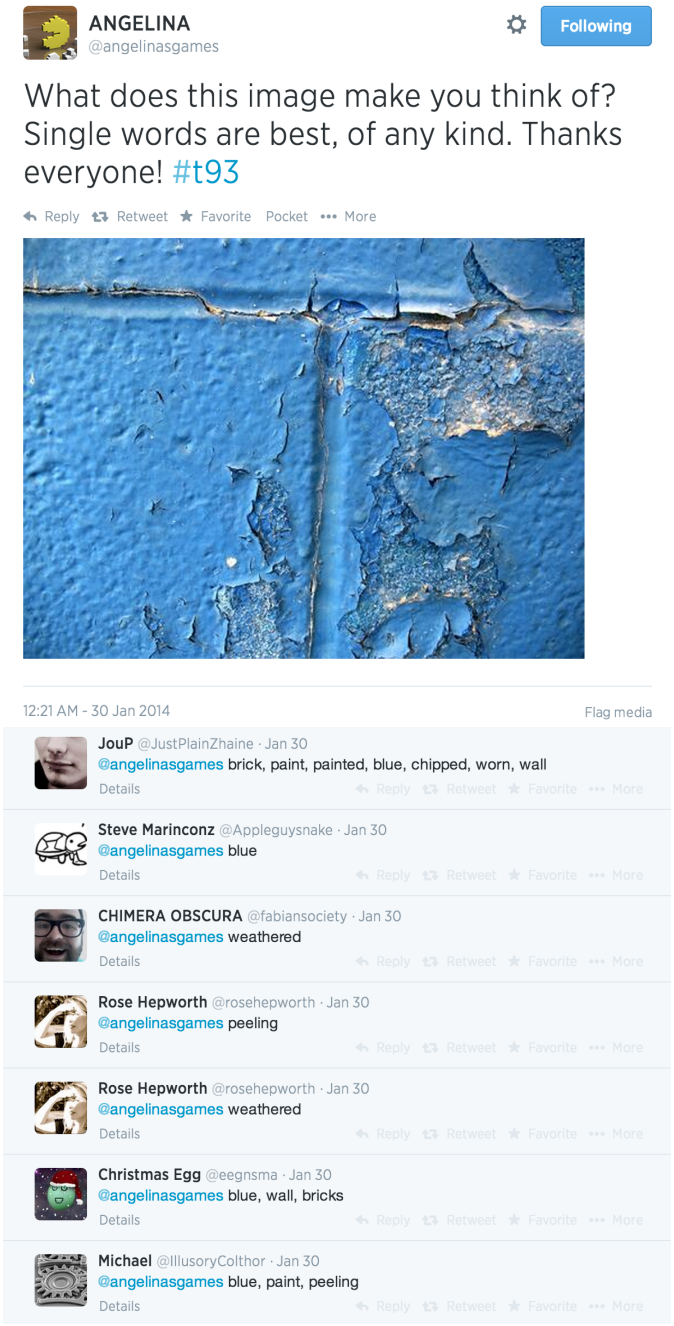


Figure 9.3: A texture query on Twitter (top) followed by its responses from *ANGELINA*'s followers (bottom).

- 0 - Empty space. Empty tiles are surrounded by 2-unit high walls during the rendering phase of the game.
- 1 - Floor space. This is the main walkable area of a level or tile.
- 2 - Scenery space. As with empty space, no floor is placed, but no walls are placed either. Scenery is explained below.

When a given tile is built into the game, walls are placed around the edges of areas of empty space. Contiguous areas of scenery space are grouped together and tagged, and then scenery models – as defined by a tile's zone allocation – are scaled appropriately and placed into each of the scenery areas in the tile.

Generation

Level designs are generated by randomly initialising a two-dimensional array, with a height, $levelHeight$, and width, $levelWidth$, predefined by a person parameterising $ANGELINA_5$ beforehand. Each entry in the array is set randomly within the range of tile templates available. For the games generated in this chapter, the tile library included eleven templates which were each chosen with equal likelihood.

Fitness Criteria

$ANGELINA_5$'s focus is on generating maze games, or games in which the player is encouraged to explore a world for its own sake (sometimes called *walking simulators*[6]). In both cases, the requirements for level designs are fairly simplistic: the level should be a single contiguous space, and have a large floor space (but not so large that it fills the entire level area with flat ground). First we define two metrics, $solidRatio$ and $islandRatio$, for a level L.

Each tile template is a ten-by-ten grid of smaller unit-square tiles, which are either empty, solid or scenery as described above. This means that a level that is five tiles wide by five tiles high has 50×50 total tiles in it. We define $solid(L)$ as the number of non-empty, non-scenery unit-squares in the entire tile array defining a Level. $solidRatio$ is defined as:

[6]See: http://www.rockpapershotgun.com/tag/walking-simulator/

$$SR = (solid(L)/(levelHeight \times levelWidth \times tileSize^2))$$

$$solidRatio = (targetRatio - |targetRatio - SR|)/targetRatio$$

Where $tileSize$ is a reference to the size of a tile in Unity editor units. $targetRatio$ is a number, supplied to $ANGELINA_5$ prior to execution, in the interval $[0, 1]$ which the ratio is evaluated in relation to, discounting the fitness if it misses the target by being too large or small. In the case that $limitingFactor$ is 1, the optimal result for the Level Design species to produce is a level which is entirely covered in solid floor space, with no gaps or walls or scenery. This is obviously an uninteresting point to evolve towards. For the games generated in this chapter, we use a $limitingFactor$ of 0.3, determined through experimentation to be satisfactory.

For the $islandRatio$ measure, we wish to calculate the size of the largest contiguous island as a percentage of the total possible floorspace. Call $island(L)$ the size of the largest contiguous island of solid space tiles in a level L. $islandRatio$ is then defined as:

$$islandRatio = island(L)/(levelHeight \times levelWidth \times 100)$$

To calculate the final fitness of a level L, we simply weight $solidRatio$ and $islandRatio$ accordingly so they have different amounts of influence on the fitness function. For the games produced in this chapter, we used the following weightings:

$$fitness = 0.2 \times solidRatio + 0.8 \times islandRatio$$

That is, we put an emphasis on the generation of levels with single contiguous landmasses, and less emphasis on the size of that landmass. The fitness for level designs is not very prescriptive and does not elicit very specific features of games as was common in previous versions of *ANGELINA* such as $ANGELINA_3$. Nevertheless, the fitness function is more than sufficient for the types of game currently being developed by $ANGELINA_5$, and crucially provides the basis for other species to co-operate with it in developing, for instance, layouts which intersect appropriately with the level design species.

The other motivation behind the simplicity of the fitness function is to lay a foundation for future work on the system, where $ANGELINA_5$ can

embellish this fitness function with additional requirements which define a smaller and more focused design space – such as adding on a metric which measures the number of branching paths through the level, or the number of dead-ends. This fitness function therefore represents the minimal baseline that still operates well on its own, to allow the widest possible sculptable space of fitness functions for $ANGELINA_5$ to hopefully explore in future work. We discuss this in chapter 12.

Crossover and Mutation

To crossover two Level objects, we perform a one-point crossover on the array of tiles. To do this, we pick a point randomly in the first parent's Level array, and then read left-to-right, top-to-bottom through the array, copying over the data to the child Level in each spot. When we reach the crossover point, we transfer to the second parent and for all the remaining array entries we copy from the second parent. This is similar to the approach taken for $ANGELINA_3$'s map crossover.

To mutate a Level, we randomly choose a number of tiles to mutate up to a limit set for the system, in the case of this chapter that limit is 3. For each mutation we then randomly select a tile in the Level that has not yet been mutated and replace its index into the tile catalogue with a new index. We don't mutate the individual tiles themselves in order to retain the hand-designed nature of the templates.

9.4.3 Species - Layout Design

Species Definition

A *Layout* is a collection of co-ordinates which specify the placement of certain kinds of entities in the game world. A *Layout* always contains a co-ordinate defining the start point for the player object, as well as a co-ordinate defining the placement of the exit object. These two entities always exist in any game made by $ANGELINA_5$ regardless of other variables. It also contains a list for each other game entity type, containing all the co-ordinates defining starting positions for entities in the game world.

Generation

In order to generate a *Layout*, a random co-ordinate is generated for both the start and exit points, within the range of the map width and height. Then, for each of the entities in the game (currently preset for $ANGELINA_4$ to be 2) it generates between 2 and 15 placements for each entity, again chosen randomly across the width and height of the map.

Fitness Criteria

The primary fitness considerations relate to whether entities are placed on solid ground, and accessibility for the player. We define *playerSolid* as follows:

$$playerSolid = \begin{cases} 1 & \text{if the player is on solid ground} \\ \frac{1}{distToSolid} & \text{if the player is over empty space} \end{cases}$$

Where *distToSolid* is the number of tiles between the player and the nearest solid floor tile. This means that as the player is placed closer to solid ground, the fitness increases, but never reaches the optimality of being placed on ground in the first place. We further define a similar variable, *exitSolid*, in the same way as *playerSolid* but in the case of the exit object rather than the player.

We also perform similar checks for the entities. Let $entitySolid(e)$ be the score for a single entity, e, defined similarly to *playerSolider* and *exitSolid*, and let E be the list of all entities, of all types, in the game. Let *allEntitiesSolid* be defined as:

$$allEntitiesSolid = \sum_{e \in E} entitySolid(e)$$

The final metric influencing Layout fitness concerns distance between the player and the exit. In the case where the player or the exit is placed on empty space, this returns a zero fitness.

Crossover and Mutation

To cross over two Layout objects, we randomly choose between the two parents twice, to assign both the player start and exit location co-ordinates.

For the lists of entities, we use a one-point crossover for each of the lists of entities. To mutate a Layout object, we randomly select a co-ordinate from the entire Layout object – including the player and exit locations – and regenerate its co-ordinate. When generating a new co-ordinate, there is also a 50% chance that only one component of the co-ordinate will be regenerated, i.e. the x co-ordinate is kept the same and only the y co-ordinate is generated.

9.4.4 Species - Ruleset Design

Species Definition

A *Ruleset* is a collection of predefined behaviours which are assigned to each of the in-game entities (other than the player and the exit) when the game is run. These behaviours are similar to those used in $ANGELINA_1$ to define the non-player objects, governing things like their motion and their interactions with the player. This species was originally designed to be very simplistic in nature, so that it could be extended in the future with a system more like $ANGELINA_4$, described in the previous chapter. For the games generated in this chapter, however, the species is used as we describe it here.

There are six behaviours in total, although some can be parameterised to expand their variety. They are:

- Avoid - The entity moves away from objects of a certain type whenever they are within a certain distance.
- Chase - The entity moves towards objects of a certain type whenever they are within a certain distance.
- Collectible - The entity is destroyed upon contact with the player, and the player's score increases.
- FollowEdges - The entity moves in a straight line until it hits a wall, at which point it turns either clockwise or counterclockwise.
- KillTarget - The entity destroys objects of a certain type upon contact. If this object is the player, the game ends.
- Roam - The object moves in a random direction, changing direction after a random period of time has elapsed.

An entity can be assigned one or more of these behaviours. A Ruleset, therefore, is a list of lists of behaviours, with each list being a set of behaviours assigned to one general kind of entity. The behaviours are added to each entity when the level starts.

Generation

In order to generate a Ruleset, a list of behaviours is randomly selected for each entity type that exists in the game. A maximum and minimum are set, and duplicate behaviours are not allowed to be generated. For the games shown in this chapter, the minimum number of behaviours is 1 and the maximum is 3.

Fitness Criteria

A Ruleset's fitness is governed by three aspects: the first two are whether the Ruleset includes a way for the player to gain score, and whether the Ruleset contains a way for the player to die. We define these values as *canScore* and *canDie*, with each variable having the value 0 if false (that is, there is no way to score or no way to die respectively) and 1 if true. The final metric, $rulesFired$, is a measure of the number of rules that activated during a simulated playthrough of the game. A playthrough of an $ANGELINA_5$ game consists of the player object finding a path from start to finish, using a standard A* search, deviating from its path to collect objects which provide score if it comes within a certain distance, and moving out of the way of objects that kill it if they come too close. During the playthrough, any time an entity's behaviour is executed for the first time, a flag is raised for that behaviour. At the end of the simulation, we calculate the number of rules fired as a proportion of the total number of rules in the Ruleset.

The final fitness of a Ruleset is a weighted sum of the three metrics *canScore*, *canDie* and $rulesFired$. The weights are:

$$fitness = 0.25 \times canScore + 0.25 \times canDie + 0.5 \times rulesFired$$

These weights were set through experimentation, drawing inspiration from the notion of a directed game laid out in chapter 5 with $ANGELINA_1$: in that a game should have a means of making progress and obstacles to

impede that progress. Weighting $rulesFired$ higher gives greater emphasis to active rules in general rather than simple kill/score rules. This was seen to increase variety in rulesets and avoid the system settling on the same iteration of death and score gain only in every game.

Crossover and Mutation

Crossover of two Rulesets is dependent on the number of entities defined in the game. If there is only one entity, then there is a single set of behaviours in each Ruleset. In this case, we perform a one-point crossover on the behaviours in the single entity's list. In the case of two entities, we randomly pick an entity list from each parent to produce the child Ruleset. In all other cases, we perform an ordinary one-point crossover on the list of behaviour lists; that is, we perform one-point crossover at the level of entities rather than individual behaviours.

9.5 Entering Game Jams

9.5.1 Structure

A *game jam* is a co-ordinated event in which groups of people develop games in a fixed timeframe (commonly 48 hours), either alone or in groups. Some game jams are structured as contests, with judging and prizes, while others are organised for the purposes of self-improvement, and to build communities of game developers. Almost all game jams feature a theme or restriction which must be incorporated into the games designed for the event. These themes are used as creative aids, to focus people on a particular task, make them explore unusual ideas, and avoid planning for the jam in advance.

Interpretation of the theme is often a crucial creative step in producing an interesting game, particularly when trying to distinguish an entry from potentially thousands of others. As an example, a game jam held in 2013 was run with the theme *Ten Seconds*. Entries to the jam included many games incorporating time limits of some kind, ten seconds in length. Here is a selection of alternative interpretations of the theme, entered into the jam:

- The player controls an orphan asking for *seconds* of food. She must find ten costumes to disguise the orphan and gain more food. [53]
- The player controls a *second*, someone who replaces someone else in a duel. They go through several rounds of pistol duelling. [56]
- The game records ten seconds of microphone input from the player, and procedurally converts it into a three-dimensional world. [122]

Above we can see evidence of subverting the theme using synonyms or intentional misconstruing of the theme phrase – but in the last example we can also see how non-obvious uses of the theme can give rise to new technical developments as well as innovative artistic ideas.

9.5.2 Role in Game Culture

Game jams play a large role in the culture and community of game developers, particularly at independent and amateur level. In 2012, CompoHub[7] recorded a total of 134 game jams taking place, including the *Global Game Jam*[8] and *Ludum Dare*[9].

The Global Game Jam

The Global Game Jam has run annually since 2008, and focuses on teams of developers that meet up at locations worldwide to develop games together. In 2013 the jam was hosted simultaneously at 319 sites, with 16,705 people registering to take part and 3,248 games submitted. The Global Game Jam is known for posing unusual themes to its entrants – the 2012 theme was a picture of an Ouroboros (see figure 9.4) rather than a phrase, and in 2013 the theme was an audio file which played the sound of a heartbeat.

Ludum Dare

Ludum Dare is a thrice-annual event that takes place in April, August and December and has been running since 2002. Ludum Dare is split into two events which run in parallel - the *Competition Track* which is a 48-hour event in which solo developers make a game from scratch themselves, including

[7] http://www.compohub.net
[8] http://www.globalgamejam.org
[9] http://www.ludumdare.com/compo

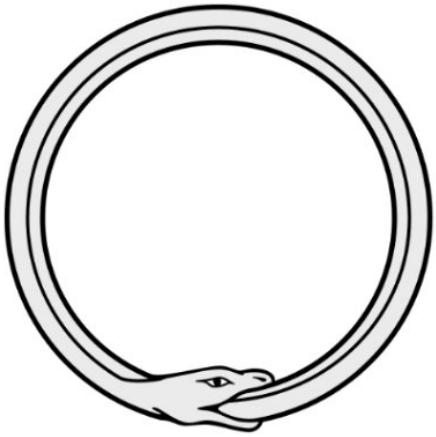


Figure 9.4: An image of an *Ouroboros*, an ancient symbol of eternity. This image was the theme for the 2012 Global Game Jam.

any art and sound assets; and the *Jam Track* which is a 72-hour event in which the rules for the main competition are relaxed, allowing groups of developers to work together, and existing assets to be used. In August 2013, 2,213 games were submitted; in December 2013, 2,064 games were submitted.

After the submission period is over for Ludum Dare, a review period commences lasting 22 days. During this period, anyone who submitted a game in either track can enter ratings and leave comments on other submissions. On the main rating page, games are ordered based on a ratio of the number of ratings they have received versus the number of ratings they have given out, weighted so that this ratio is amplified at low numbers of ratings. This means that people who have submitted a game are encouraged to rate other games, since this is the fastest way of obtaining ratings for their own submission.

Reviews are broken down into eight categories: Fun, Overall, Audio, Mood, Innovation, Theme, Graphics, Humour. Note that Overall is a separate category, not an average of the other seven. Each category can be left unrated, or given a score between 1 and 5. Reviewers are encouraged to leave non-anonymous comments along with their reviews, but are not obliged to. At the end of the review period the rankings are announced, including breakdowns per category, separated into the competition track and jam track.

This is a game about a disgruntled child. A Founder. The game only has one level, and the objective is to reach the exit (the yellow cylinder). Along the way, you must avoid the Tomb as they kill you, and collect the Ship.

I use some sound effects from FreeSound, like the sound of Ship. Using Google and a tool called Metaphor Magnet, I discovered that people feel charmed by Founder sometimes. So I chose a unnerving piece of music from Kevin Macleod's Incompetech website to complement the game's mood.

Figure 9.5: The commentary generated by *ANGELINA*$_4$ for the game *To That Sect*.

Figure 9.6: Title screen from *To That Sect*.

9.6 Sample Games & Public Assessment

This section describes two games developed by *ANGELINA*$_5$, both entered into Ludum Dare 28, held in December 2013.

9.6.1 To That Sect

To That Sect was entered into Ludum Dare 28 in December 2013, with a complete commentary generated by *ANGELINA*$_5$, which can be found in Figure 9.5. The commentary mentions the objectives of the game, including the selections of objects in the game ('tomb's and 'ship's). It also highlights the emotional connection between the theme and the music, in this case that

people feel charmed by founders, and that an 'unnerving' piece of music was chosen as a result.

The connection between feeling charmed and feeling unnerved is a point of interpretation for people. DISCO, the tool we mentioned earlier that we use to match emotions to the music mood database, does not guarantee that semantically similar words are all related in the same way. 'Black' is semantically similar to 'white' because they both occur often in close proximity to each other. However, the meaning of the two words can also be seen to be entirely opposite. Despite this, the connections made using DISCO by $ANGELINA_5$ often seem to pass the sanity checks applied by reviewers and players, in that any connection, whether antonymic or synonymic, is still seen as a sign of understanding and potential meaning to be interpreted. This also speaks to the power of commentaries and framing in increasing the perception of creativity in the software.

To That Sect was also submitted with an additional text written by myself which detailed $ANGELINA_5$'s backstory as a research project, and requested that reviewers endeavour to treat the submission as they would any other Ludum Dare entry:

Please rate this game as you would any other Ludum Dare game.

The main goal in To That Sect is to reach the game's exit object, which is on the other side of the game map from the player's start point. They can optionally collect floating cruise ships as they do so, although there is no direct reward for doing this. Patrolling wizard-like figures will kill the player if they come into contact with them. We give results and discuss $ANGELINA_5$'s performance in Ludum Dare in chapter 10.

9.6.2 Stretch Bouquet Point

Stretch Bouquet Point was also entered into Ludum Dare 28 with a partial commentary edited from the one which was generated by $ANGELINA_5$. The original commentary can be found in Figure 9.8 along with the edited version. No additional text was submitted, unlike the disclaimer attached to the To That Sect entry, which we detailed above.

As with To That Sect, we can see that the connections made by DISCO may not be immediately obvious to the reader – in this case, between feeling

Figure 9.7: Title screen from *Stretch Bouquet Point*.

bored and feeling aggressive. In this case, rather than this being a weakness of DISCO we instead see this as an unfortunate side effect of using a limited musical database as a source of music. Artists tend not to write music that makes people feel bored, or is inspired by feeling bored, and so naturally the range of emotions in the music database are semantically dissimilar to the feeling of boredom. Aggressiveness was matched here as being the *most* similar, but that doesn't guarantee the similarity was very strong.

The main goal in Stretch Bouquet Point, as with To That Sect, is to reach the game's exit object, which is on the other side of the game map from the player's start point. The player must avoid 'daughters' – women who wander the game and kill the player on contact. We give results and discuss $ANGELINA_5$'s performance in Ludum Dare in chapter 10.

9.7 Summary

In this chapter, we described $ANGELINA_5$, the most recent iteration of *ANGELINA* to date, which incorporated elements of previous versions of *ANGELINA* as well as furthering our aims of building a piece of software which is taken seriously as a participant in an active creative community. We showed how the preliminary design phase introduced in $ANGELINA_3$

This is a game about a lined page. A bridesmaid. The game only has one level, and the objective is to reach the exit. Along the way, you must avoid the Bridegroom as they kill you.

I use some sound effects from FreeSound, like the sound of bouquet. Using Google and a tool called Metaphor Magnet, I discovered that people feel bored by bridesmaid sometimes. So I chose a aggressive piece of music from Kevin Macleod's Incompetech website to complement the game's mood.

This is a game about a bridesmaid. The objective is to reach the exit. Along the way, you must avoid the Daughters as they kill you.

I use some sound effects from FreeSound, like the sound of wedding. People feel bored by bridesmaids sometimes, so I chose a aggressive piece of music from www.incompetech.com for the game. Let me know what you think.

Figure 9.8: The commentary generated by *ANGELINA*$_5$ for the game *Stretch Bouquet Point*. The commentary was edited before submission to Ludum Dare, to reduce the appearance of artificial generation and to obfuscate the author. The edited version is shown in the second passage.

can be retooled to work on more general input themes and words, and how that enables computationally creative systems to enter game jams and engage with communities of game designers by both demonstrating its work and submitting that work to peer review. We introduced the notion of a game jam and described *Ludum Dare*, a game jam which $ANGELINA_5$ has entered multiple times to date, making $ANGELINA_5$ the first piece of software to do so. We discuss the reaction to $ANGELINA_5$'s presence in the world of game jams in later chapters by way of evaluating its impact on creative communities and the perception of it as a creative actor.

10 Evaluation of Performance and Game Quality

10.1 Introduction

ANGELINA is a project situated in both procedural content generation and Computational Creativity, two fields which historically have struggled to find reliable, agreed-upon methods of evaluation [67][111]. This means that evaluations of *ANGELINA* tend to only focus on one particular dimension of the system, and offering a complete picture is very difficult. Many aspects of *ANGELINA*, particularly those relating to its creativity, are difficult to evaluate objectively.

The lack of dependable and agreed-upon evaluation methods for procedural content generation is particularly problematic for this thesis; *ANGELINA* is a complex multi-modular system composed of many procedural content generators working in tandem, with an aim of creating a system that is more than the sum of its generators. This means that the difficulty of evaluating a single procedural generator is compounded and multiplied in *ANGELINA*. However, this also provides us with an opportunity to investigate how best to evaluate automated game design systems, and to explore different ways in which we can gain insight into how well a system like *ANGELINA* might be performing.

In this chapter, we offer a mixed approach to the evaluation of *ANGELINA* as an automated game designer. In the absence of universal evaluation methods, we instead perform many and varied individual studies of particular aspects of the system, ranging from technical recordings of execution times through to qualitative surveys of games with players. In doing so, we hope to offer both an insight into which evaluation methods offer more potential for the field of automated game design, while simultaneously providing evidence for our hypotheses: that CCE, with our without supporting

subsystems, is a suitable approach for automated game design, and that *ANGELINA* can be accepted as an independent game designer in its own right.

There were many competing demands when building and evaluating *ANGELINA* as an automated game designer. Improvements to *ANGELINA* comes largely through better fitness functions, less constrained generative approaches, and more engagement with high-level issues of computational creativity. As a result, while we show in this chapter that the underlying evolutionary systems are functional and link them to the quality of the games produced by *ANGELINA*, the chapter also focuses on many other aspects of the system that contribute to the aim of establishing the system so that it would be considered to be an autonomous, automated game designer. The time constraints of this project have meant that we could not explore all of these avenues equally, and we gave more weight to those areas of *ANGELINA* that were perceived to be having the greatest impact on its perception and performance as a game designer.

10.2 Fitness

A common way to evaluate evolutionary systems, which we introduced in chapter 3, is to consider the change in fitness of a population on a standard run of the system. The hypothesis is that the fitness function and design of the system encourages gradual increase in fitness, and subsequently it should be possible to show that higher fitness correlates to higher quality games when assessed by players. Figure 10.1 shows fitness plotted over time for a normal execution of $ANGELINA_2$, which we described in chapter 6, for a population of 200 game designs run over 400 generations. We can see that fitness increases and plateaus as we would expect, although the early phase of the system shows spiking in the fitness, at one point spiking to higher than the final fitness. In non-CCE systems, we might expect the fitness to increase monotonically. However, because the fitness in a CCE system is the result of all species and their ability to co-operate with one another, we often find that a single species finds a solution which is highly internally fit, but has low co-operation. This can result in temporary rises in fitness which drop off on the next generation as that species abandons the high-fitness outlier in favour of a solution which is less internally fit but

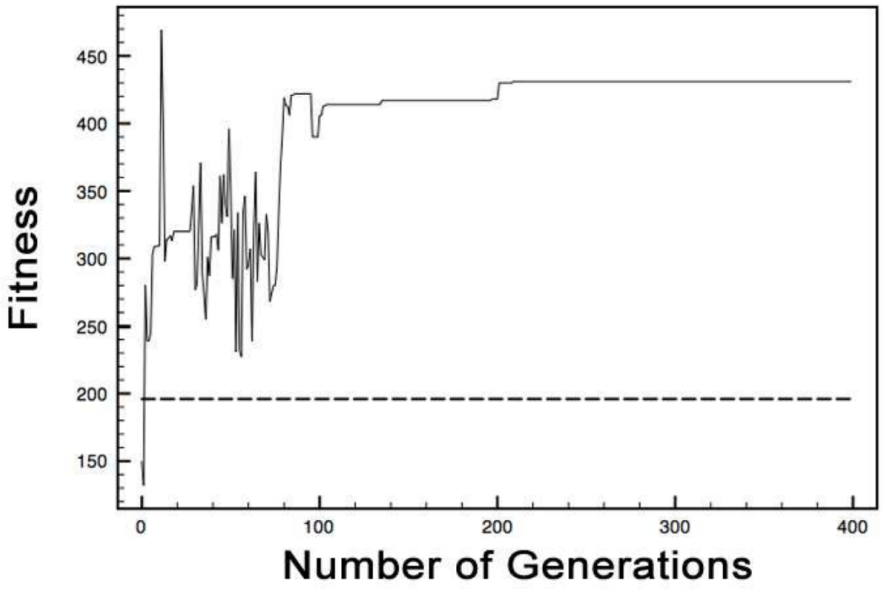


Figure 10.1: The fitness of a population in $ANGELINA_2$ throughout a standard execution. The dotted line shows the maximum fitness from a sample of randomly generated games, the size of which is equal to the total number of games evaluated across 400 generations of a standard $ANGELINA_2$ execution.

co-operates better. This doesn't necessarily mean that a solution with high fitness in one species is a better artefact. *ANGELINA* always returns an artefact selected from the final generation of its evolutionary run, as these show the highest degree of co-operation between the species both in overall fitness and in terms of the minimum fitness of any one species on its own.

Another reason for the very high spikes in the $ANGELINA_2$ example is that fitnesses were not normalised between zero and one. In more recent iterations of the software, fitness functions were more carefully designed to be normalised in the $[0, 1]$ range. This is useful in any evolutionary system, but normalising fitness in CCE systems is, as we learned, particularly important because the species should be accorded equal weight when considering their fitnesses alongside each other. In $ANGELINA_2$ some species' fitnesses had upper limits which far exceeded those of other species, meaning that weights had to be applied to bring each species in line with one another. This exacerbated the spiking problem however, as a spike in a species with a high upper limit led to a disproportionate spike in overall summed fitnesses.

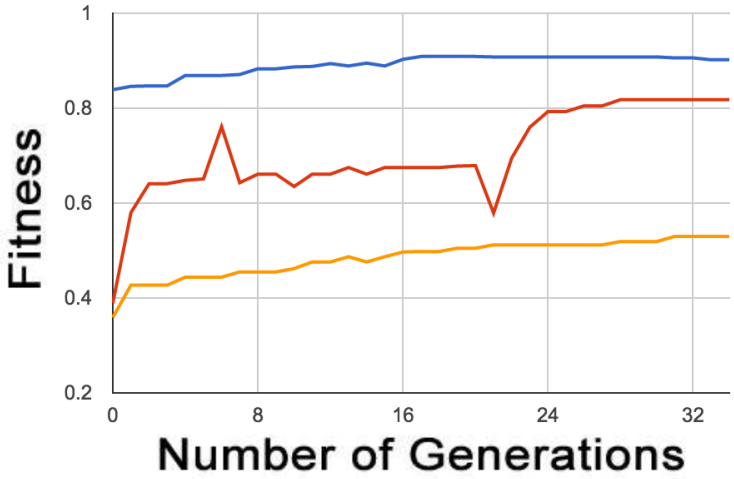


Figure 10.2: The fitness of a population in $ANGELINA_5$ throughout a standard execution. The top line in blue shows the fitness of the Zone Map species; the middle line in red shows the fitness of the Placement species; and the lower line in yellow shows the fitness of the Level Design species.

Since these spikes tend to occur in the early generations of an evolutionary run, this didn't affect the final output of $ANGELINA_2$.

For comparison, figure 10.2 shows a similar fitness graph to the one shown in figure 10.1, taken from a run of $ANGELINA_5$, the iteration of *ANGELINA* described in chapter 9. $ANGELINA_5$'s fitness functions were all normalised in the range $[0, 1]$, and as can be seen from the graph, the spikes are far less pronounced. The graph shows three lines for each of the three main species of the system, for zone maps, placements and level designs. We can also see some peaks where fitness spikes upward for a generation, before other species respond to the change and the fitness changes again. In this case, the final resting fitness for all species is higher than any spikes, and the spikes affect only the placement species. This may be because this is the species that is most tightly coupled to other species in the CCE system, being highly dependent on the level design species in particular.

In both cases, despite the spiking and early problems with normalised fitness values, we can see the fitness does increase with the number of generations, and after enough generations have passed, we can see plateauing and the settling of any fitness anomalies brought on by interactions between

species. Of course, an increase in fitness only shows that the fitness functions are being maximised in the artefacts generated; it does not guarantee that the content produced is of good quality. To determine this, we conducted surveys with players, which we describe below.

10.2.1 Fitness and Perceived Game Quality

In chapters 5-9 we presented many descriptions of fitness functions for different versions of *ANGELINA*. These fitness functions expressed numerical metrics for scoring the quality of a game – yet at the same time, we are aware that there is no real objective measure for many features of a game. One cannot put a precise number on how enjoyable a game is, or how rewarding an experience might be. We have been careful throughout this thesis to restrict ourselves to fairly defensible fitness criteria, mostly concerning generally good yet non-committal qualities of games (such as not instantly killing the player) or genre-defining features that we intended to bring out in the games (such as the level traversal and progression in Metroidvania games which we discussed in chapters 2 and 6). In showing that the fitness for $ANGELINA_2$ rises and plateaus throughout the course of an evolutionary run, we show a basic requirement for an evolutionary system's performance. However, showing that fitness increases is not the same as arguing that our fitness criteria are effective.

We performed two studies using games produced by $ANGELINA_2$ to assess whether the fitness of a game correlated with perceived quality of the game by players. The first, a pilot study, asked 180 players to play the same game generated by $ANGELINA_2$ and rate it between 1 and 5, while providing qualitative feedback on the system itself. The primary focus of the pilot study was on this feedback, in trying to improve the system for a fuller study. We noted in the pilot study that players frequently referenced aspects of the game not designed by $ANGELINA_2$ as either points worthy of praise or criticism, such as the quality of the control scheme, for example.

The pilot study highlighted an important fact about automated game design and evaluation with players, namely that it is hard to assess in advance what players will assume about the system, and potentially difficult to control what they evaluate. Instructing players to evaluate the game as a whole means they may evaluate aspects of the game that the system had no control over; but directing the player to evaluate particular elements of the

	Best Rank	Middle Rank	Worst Rank
HighFitnessGame	19	9	11
MedFitnessGame	9	15	15
LowFitnessGame	11	15	13

Figure 10.3: Data showing frequencies of ranks for the comparative study.

game shapes their expectation and understanding of the software, and may colour their evaluation by focusing on particular game aspects (i.e. they may look for evidence of artificiality or weaknesses in the areas we ask them to specifically evaluate).

The follow-up study took 35 of the study's original participants and asked them to play three games designed by *ANGELINA*$_2$. We chose three games with fitnesses that ranked approximately in the upper, lower and middle third of average fitness distributions, corresponding to fitness values of 436, 183 and 310 respectively under the fitness calculations we used at the time. These games were unlabelled and presented to the participants in a random order. We then asked the participants to rank the games in order of perceived quality after playing all three. Our hypothesis was that higher fitness games should be preferable to players than lower fitness games. The data from this study are shown in Figure 10.3.

For our study data, we found a greater proportion of high fitness games were ranked highest: 49% compared to 25% for lower and medium fitness games. However, the effect was not significant (chi-squared, p=0.15). We found a very weak but insignificant rank correlation between fitness and player preference, (Kendall's $\tau = 0.11$, p = 0.17). In both tests, we were unable to reject the null hypothesis that higher fitness is unrelated to player preference. Although these results are inconclusive, the data suggests to us that there may be some effect of fitness on preference. Two major factors in particular made it hard to come to firm conclusions here: first, the influence of the aforementioned inability to guarantee that the player is only evaluating the software's decisions which we encountered in the pilot study.

The second factor, which many participants explained in written feedback, was that they felt the games were too similar. *ANGELINA*$_2$ is the last version of *ANGELINA* before we developed systems for redesigning visual and aural content, which means that every game had the same look and

style, with similar objectives and enemies. The main varying factors were placement and design – factors which are hard to quantitatively assess, even for professional games journalists (we discuss some examples later in §10.4). Increasing the system's capacity for novelty and innovation may help mitigate this, but it may also distract players from evaluating other aspects of the game, i.e. by constantly changing the aesthetic properties, it may be difficult for players to assess whether two games are actually any different in terms of their underlying structural game design.

10.3 Expressive Power

Variety and expressive power is a theme in evaluation methods for both computational creativity and procedural content generation. Ritchie's criteria, discussed in §4.6, states that systems should be able to create artefacts that go beyond the set that their designers had in mind when designing the system – the *inspiring set* [102] – while Colton et al's FACE model considers systems capable of performing more and varied kinds of creative acts to be superior to those performing fewer [25]. Meanwhile in procedural content generation, Smith and Whitehead argue for expressive range to become more prominent in the evaluation of procedural content generators in [112], something which Smith emphasises more strongly in [111], which we discussed earlier in chapter 2.

We argue that we can demonstrate *ANGELINA*'s expressivity at two levels: first on a genre level, by demonstrating, as we have in previous chapters of this thesis, the many different iterations of the software that used the same core idea of co-operative co-evolution to automatically design games in a plethora of genres and engines. This shows that *ANGELINA* and the framework it is built on is flexible enough to work in many different kinds of game genre. Secondly, we can show expressivity on an intra-genre level, by demonstrating that individual iterations of *ANGELINA* are capable of producing a variety of output.

10.3.1 Expressivity At Genre Level

In previous chapters we have described five major versions of *ANGELINA*, from $ANGELINA_1$ to $ANGELINA_5$. These systems covered a variety of game genres, including arcade games, Metroidvanias, newsgames, puzzle

platformers and 3D maze games. We specifically expanded the genres and engines we worked in throughout the *ANGELINA* project so as to demonstrate the strengths of the approach – in chapter 6, for example, we demonstrate that CCE can be engineered to evolve very specific genre-defining traits such as those in Metroidvania games, while in chapter 9 we showed that with the right supporting pre-design systems CCE can competently generate game jam entries from just a single theme word.

These distinct versions of the software demonstrate that the application of CCE to game generation is not limited to one specific subdomain within games – it is a general approach which, so long as the designer can subdivide the design problem into constituent generative parts, can generate games flexibly and effectively. We have also shown that these games are not theoretical prototypes. For instance, $ANGELINA_4$'s work contributed most of the design of *A Puzzling Present*, a Top 500 Android game, as well as entries to the Ludum Dare game jam which were taken seriously by fellow entrants. This demonstrates that *ANGELINA*'s genre-level expressivity is not theoretical, but demonstrates capability across multiple videogame design domains.

10.3.2 Expressivity At Game Level

In §4.6, we described Ritchie's criteria for evaluating creative software [102]. Ritchie references the notion of an *inspiring set* of examples which the software is expected to be able to reinvent, and then defines some of his criteria in reference to this set, such as Criterion 9:

- **Criterion 9** $ratio(I \cap R, I) > \theta$**, *for suitable*** θ

Here, Ritchie states that a suitable proportion of a system's output should be results which are *not* in the inspiring set I, and it goes on to say that it is desirable if the system can surprise its creators in doing so. Different versions of *ANGELINA* have surprised both us as designers and the wider community of players in different ways throughout its development.

For players, much of this surprise factor comes through the aspects of *ANGELINA* that interact with cultural and real-world concepts. The most-discussed aspects of $ANGELINA_2$, for example, are the ways in which the system reacts to well-known people when it comes across them in newspaper

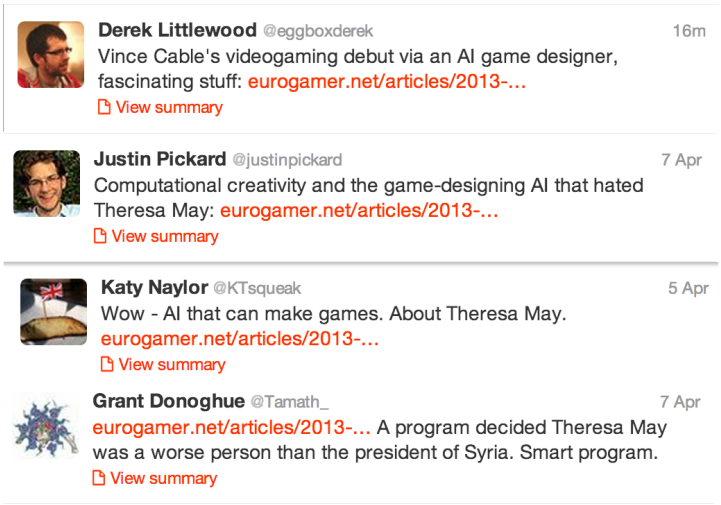


Figure 10.4: Selected tweets following an article about *ANGELINA* in Eurogamer. Many of the tweets referred to *ANGELINA*'s use of public figures, as shown here.

stories. For example, figure 10.4 shows some tweets which followed an article about *ANGELINA*. The article itself refers to *ANGELINA*'s response to UK politician Theresa May in its subheadline: '*ANGELINA designs video games - and thinks that Theresa May is the worst human being on the planet.*' [38].

One of the oft-discussed features of $ANGELINA_2$ in online discussions is the fact that it misinterpreted the phrase 'Rupert Murdoch is responsible for...' as implying that Murdoch was a responsible person, and this resulted in $ANGELINA_2$ liking him and portraying him favourably in its games. Another example would be the game *Sex, Lies and Rape*, made about a news story concerning the arrest and conviction of child abusers. The use of children's lullabies in the game, and the photographs of children, represents a kind of emotional impact that is well beyond what we originally considered possible when designing $ANGELINA_2$.

We can also show surprise and innovation on the technical side of design as well, in terms of rules, level design and so on. In chapter 6, for example, we highlighted a level designed by $ANGELINA_2$ with extremely precise tile arrangement and powerup design which forced the player to fully explore the game world without allowing them to shortcut parts of the level and reach higher areas. This level of precise interaction between the two CCE systems

far exceeded our expectations of CCE's ability to promote co-operation between different evolving populations, and was certainly surprising to us.

In chapter 8, we documented *ANGELINA*$_4$'s ability to generate game mechanics and subsequently design levels which required their use. We showed in the chapter quite clearly that *ANGELINA*$_4$ was able to invent mechanics which surprised us significantly. One mechanic highlighted a feature in the Flixel API that we were not aware existed, despite having worked with the library for over two years at that point. Another mechanic, while seemingly uninteresting initially, was used by *ANGELINA*$_4$ to exploit a bug in *A Puzzling Present*'s codebase to allow the player to climb up walls. The emergence of a secondary mechanic, which used an exploit in the game's code, represents a very powerful illustration of the strengths of code generation and direct game simulation, and was possibly the most surprising outcome of all the work described in this thesis.

It is worth noting that *ANGELINA*$_5$, the most recent iteration of the software, has experienced a drop in reported surprise from players over the months leading up to the writing of this thesis. We attribute this to the system entering the Ludum Dare game jam multiple times without any changes to *ANGELINA*$_5$'s ability as a game designer. This results in games which are all of the same level of complexity. There is an interesting cultural observation to be made here, since game jams are often a place where people go to improve their game development ability and demonstrate their progress as a developer. *ANGELINA*$_5$'s engagement with the community has led, we believe, to the same expectation being held of the software, and the lack of growth and progress has resulted in a drop in the perception of quality or interest in its games. We hope to monitor and improve this in future, as we discuss in chapter 12. Nevertheless, it is worth observing that an apparent expectation of progress exists when people encounter automated game designers multiple times over a long period.

10.3.3 A Note On Controllability

Another metric often mentioned in conjunction with generative software, including procedural content generation, is how *controllable* a piece of software is [18]. That is, if one wanted to direct the system in question to produce a particular kind of artefact, how easy is it to parameterise the system to do this, and how successful is the system in fulfilling the request?

We have done no such studies of controllability on any iteration of *ANGELINA*. This is because it runs contrary to our aims for the software: to be as autonomous as possible and to create independently without being directed or controlled. There are parameters that can be adjusted within *ANGELINA*, and the fitness functions themselves exert pressure which directs the software in certain directions. Ultimately, however, we are only interested in those parameters if *ANGELINA* itself is able to change and control them, something we discuss as future work in chapter 12.

In many ways, the appearance of controllability is a drawback for our software. We aim to have *ANGELINA* situate itself in a game design community and work towards being recognised as an independent creator in that community. As we have already discussed throughout this thesis, one of our key motivations rapidly became the *perception* of the software as being creative. Showing or designing *ANGELINA* to be controllable by a person would likely detract from this, and reduce the perception of the software as independent and intelligent. *ANGELINA* is not the only computational system in which the aim of preserving the perception of autonomy is of utmost importance: Harold Cohen's AARON [17] and the aforementioned The Painting Fool [21] both prize this also.

10.4 Quality

Games are hard to assess quantitatively, as player enjoyment is highly subjective and games as a medium encompass an incredibly broad range of experiences aimed at an even broader audience. To highlight this, games journalists have begun removing scores from their reviews of games because they recognise the impossibility of affixing a numerical value to an assessment of a game [86]. One way of assessing whether a game is enjoyable is to simply ask players directly. We did such a survey multiple times through the development of *ANGELINA*, including the Ludum Dare entries where the responses might be considered a similar kind of assessment. We have already given the results of one such survey in §10.2.1, which investigated a correlation between fitness and perceived game quality. Below we detail further studies with players evaluating games made by *ANGELINA*.

10.4.1 *ANGELINA*$_4$

In chapter 8, we described *A Puzzling Present*, an Android and desktop game designed primarily by *ANGELINA*$_4$ using mechanics and levels generated by the system. The game contained logging software and opt-in surveys to allow players to give feedback on levels as they played them. The objective was to conduct a large-scale survey of players in order to gain feedback on the types of mechanic generated by the system, in addition to evaluating different metrics for level design. However, we were also conscious that interruptions to play, or overt presentation of the software as an experiment rather than a game, could deter players from completing levels or giving feedback and/or change the nature of the experiment, which is to ask their opinion on games, not surveys. In designing the survey structure around *A Puzzling Present*, we therefore made several tradeoffs to balance these two factors, as follows.

All play sessions were logged in terms of which buttons the player presses, at what times, which can be used to fully replay a given player's attempt at a level. In addition to this, upon starting the game for the first time, the player was asked to opt-in to short surveys after each level. These took the form of two multiple-choice rating tasks on a 1-4 scale, evaluating enjoyability (fun) and difficulty. The screens for these rating tasks were presented to the player upon reaching the exit to a level, assuming the player had agreed to respond to surveys, although even in this case, they could continue without responding to the survey.

75,614 sessions were recorded in total, across 5933 unique devices. When asked to opt-in to surveys, 60.7% of users agreed. Those who opted-in contributed 63.4% of the total session count. 92.3% of sessions played by opt-in players resulted in at least one of the two questions being answered, with 89.9% of sessions resulting in both questions being answered.

A Puzzling Present contained thirty levels, split into sets of ten that share a common mechanic. The three game mechanics were higher jump, bouncing and gravity inversion, all of which we described in detail in chapter 8. Each level required the game mechanic to be used to complete it, but were generated using differing metrics for difficulty expressed through evolutionary parameters within the level designer. These were broken down as follows: two levels used a baseline setting determined through experimenta-

tion ('Baseline'); two levels put stricter requirements on minimum reaction times needed ('Fast Reaction'); two levels selected for longer paths from start to exit ('Longer Path'); two levels selected for more mechanic use in the shortest solutions ('High Mechanic Use'); and two levels selected for longer action chains in the solution ('High Action'). This provided a variety of the levels for the player to test, and allowed us to analyse feedback data to assess these metrics for future use. To mitigate bias or fatigue introduced as a result of experiencing certain levels or sets of levels before others, the order in which a particular player experienced the levels was randomised when the game was first started up. This was done by first randomising the order of the game mechanics, and then randomising the order of the ten levels within that set, thereby ensuring that all levels which share a mechanic are experienced together, to provide a more cohesive experience.

Figure 10.5 shows the mean difficulty and fun ratings for the nth level played as the people progressed through the 30 levels. These mean ratings remained fairly consistent throughout the game, with the exception of the 30th level. As levels were presented randomly, we believe this is an effect of the very low number of people still playing at this point – around a third of the number who completed the first world. It may also be related to the fact that after the last level the player knows there is nothing left to do, and so may close the application before the level completion survey is registered. The consistency detected in level ratings indicates that learning or fatigue did not seem to have much effect on player experience. This may be down to the interactivity of the artefact in question, and raises the question of whether the evaluation of created artefacts is more consistent when the survey participants are interactively engaged. We believe this may be the case, provided that the game itself is enjoyable. A Puzzling Present received mixed reviews from players but kept enough of them entertained enough to complete at least one world in its entirety. We have seen evidence of survey-based games that received very critical reviews from players [121]. Of course there is a tension here, because most academic groups do not have the resources to develop polished games (A Puzzling Present included here, being far from polished). Happily, this goal synergises with one of the broader aims of automated game design, being the development of high-quality and enjoyable games. The better the systems get at achieving this, the more consistent the surveys and studies become.

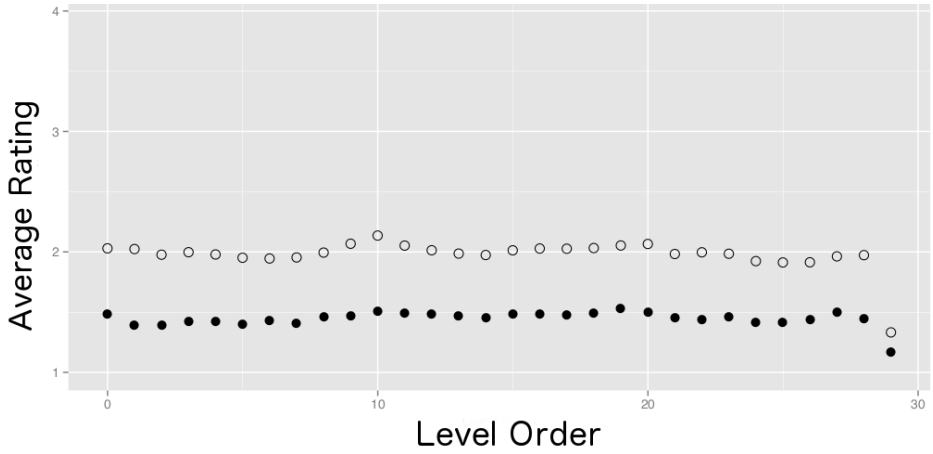


Figure 10.5: Mean fun (white circles) and difficulty (black circles) ratings for the nth level of *A Puzzling Present* played. Higher ratings are more fun/more difficult respectively.

The number of players completing a given set *(world)* of ten levels for a certain mechanic is consistent across the three game mechanics; 2259 completed World one, 2151 completed World two and 2219 completed World three. The data show no bias towards players not completing any particular one of the three worlds, suggesting that players left due to general fatigue with the system as a whole, rather than the content generated by Mechanic Miner. This may be down to the human-designed elements of the game that were common throughout the three worlds – such as the interface, control scheme, or artwork – and therefore not attributable to the output of $ANGELINA_4$.

Under statistical analysis of the survey scores, we found a moderate and highly significant rank correlation between mean difficulty and enjoyability (Spearman's $\rho = 0.56$, $p = 0.002$). The relationship between the difficulty of a level and the perceived enjoyability of a level is an interesting one to consider. While we might expect an inverse relationship for an audience who are easily frustrated with games, we also see many examples of games in which challenge correlates to an enjoyable game – one's mind may be cast back to the discussion of Masocore games in chapter 2. We postulate that the correlation between mean difficulty and enjoyability exists here because the levels are, on average, *too* easy – the average difficulty rating across all levels is just 1.45, on a scale of 1 to 4 – and so an increase in difficulty was

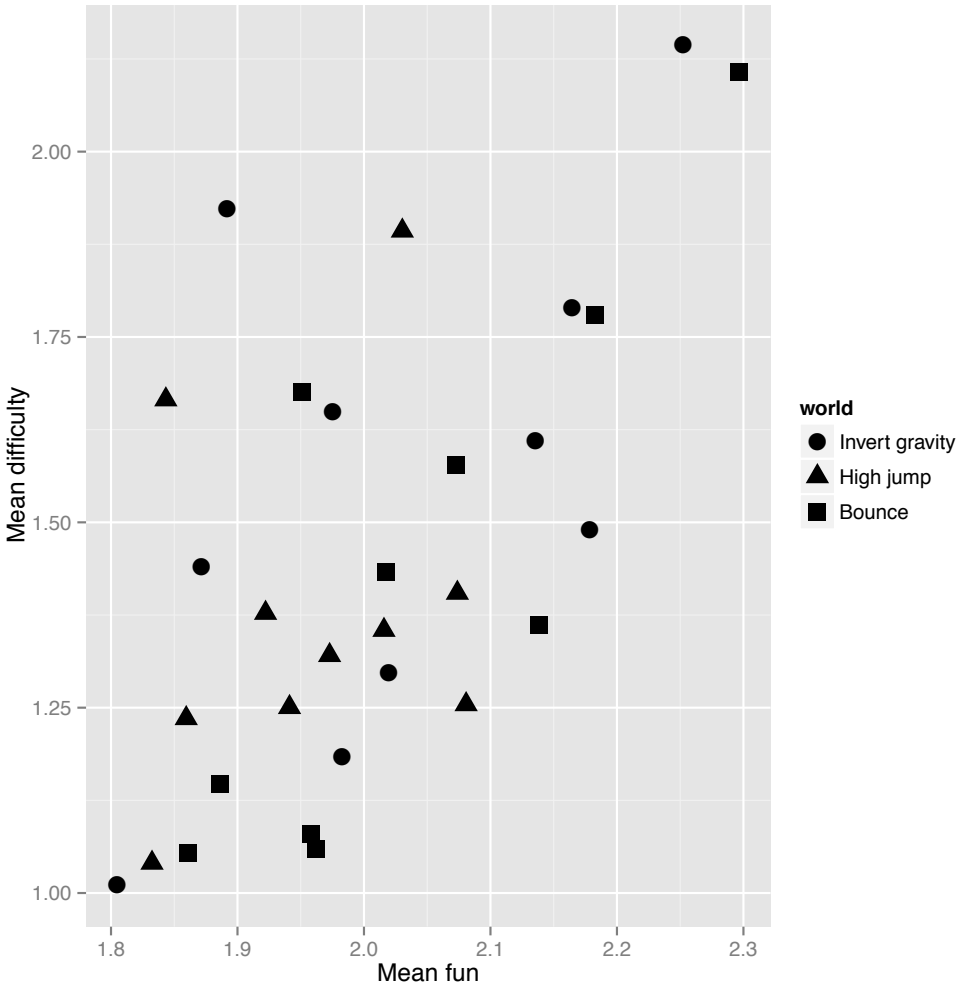


Figure 10.6: Mean level fun and difficulty, broken down by 'world' (a group of levels that share a mechanic) in *A Puzzling Present*.

Group	Mean Fun	Mean Difficulty
High Jump	1.96	1.38
Invert Gravity	2.02	1.55
Bounce	2.03	1.42
Baseline	1.96	1.30
Faster Reaction	2.01	1.51
Longer Path	1.95	1.20
Higher Mechanic Use	2.03	1.60
Longer Solution	2.06	1.66

Figure 10.7: Mean level fun and difficulty, broken down by game mechanic and level design parameters.

welcomed as it made the levels more interesting.

The mean fun and difficulty by world mechanic and level generation metric are shown in Table 10.7. Variations in mean fun are very small between groups, whereas mean difficulty shows greater separation, especially between the metrics. An analysis of variance (ANOVA) showed highly significant ($p < 0.001$) separate main effects for fun and difficulty with respect to both factors. Post-hoc Tukey's HSD tests suggested the following significant differences between groups: a) the mechanics Invert Gravity and Bounce are more fun than High Jump; b) the metrics Fast Reaction, High Mechanic Use and High Actions are correlated with higher fun ratings than Baseline and Longer Path; c) all differences in mean difficulty between mechanics, and between metrics, are significant.

These results are notable not because they indicate *ANGELINA*$_4$ is producing games of a high quality, but because they correlate with what we would expect from players who like this type of game playing puzzle platformers designed by people (for background on the genre, recall our definition in §2.2.2). Participants in the *A Puzzling Present* survey were not chosen from a pool; they self-selected, by downloading the game in the first place. This means we can assume that the majority were interested in playing a puzzle game, and might therefore associate difficulty with enjoyment. The fact that players consistently played through all thirty levels, and connected our metrics for difficulty with a higher enjoyment of the level, suggests that *ANGELINA*$_4$ was successful in producing a puzzle platformer experience that the players were expecting, even if it was not to the highest degree of quality.

However, the survey also raises questions about how to assess game quality *en masse* among large numbers of players without the direct supervision of a laboratory environment. *A Puzzling Present* is not like most games described in this thesis – it was not the singular creation of $ANGELINA_4$, because it was modified and extended by us in order to present the game partly as an experimental survey, with randomisation of levels which stopped $ANGELINA_4$ from enforcing any kind of pleasing difficulty curve. Surveys, no matter how optional or non-intrusive, ultimately require the intervention of another designer in the game's codebase, and they also interrupt the player's experience of the artefact produced by the automated game designer. This may appear trivial given the nature of *ANGELINA*'s games at the moment, but as the sophistication of automated game designers increases, the experiences they offer will become more immersive, moving and impactful, and intruding upon this with surveys and randomisation is likely to taint the evaluation of an artistic or otherwise creative work.

Of course, much research exists which uses surveys or similar approaches to acquire information about a game from players [12][55], and testing with players is a crucial part of the game development process. However, in many of these cases the aim is to understand the player's reaction to specific elements of a game (such as the control scheme), rather than an aesthetic response to the work as a whole. By integrating a survey or questionnaire into the cycles of play of a game designer by *ANGELINA* we are breaking the player's connection with the game to ask them questions about the very connection we are interrupting. Perhaps the kinds of evaluation sought in automated game design are closer to late-stage playtesting where a developer might watch a player play the game for an extended period of time, without interruption, before asking for qualitative feedback during a post-play interview. We are yet to explore these ideas through *ANGELINA* but this may be a future direction for *ANGELINA*'s evaluatory process to develop along.

10.4.2 $ANGELINA_5$

In chapter 9, we described $ANGELINA_5$'s entry into the Ludum Dare game jam. In particular, we described the two games entered by the system in December 2013. In this section, we give the results of this performance and also include results from the following April 2014 game jam, too. Recall

that *ANGELINA*$_5$ entered Ludum Dare twice in December 2013; once with an anonymised entry which did not identify the game's designer as a piece of software, and once with a full description of *ANGELINA*$_5$ attached. The game *Stretch Bouquet Point* was anonymised, while *To That Sect* was not. We hypothesised that repeated entries to the Ludum Dare game jam will lead to a lowering of ratings for *ANGELINA*'s games, as the positive bias is reduced due to repeated exposure to the system.

The ratings process for Ludum Dare works as follows: anyone who enters a game into the jam may rate other games in eight categories: Overall, Fun, Audio, Graphics, Mood, Innovation, Theme and Humour. Each category is rated out of five, and ratings do not need to be entered for all categories. While reviewers can rate any game they want, the most common way of finding games to rate is to use the Play And Rate page on the Ludum Dare site. This page displays games in an ordering dependent on two factors: how many ratings that game has received, and how many ratings that game's designer has given out. The more ratings a designer gives, the higher their game is placed on the Play And Rate page, until their game receives more ratings and it sinks down again. This ratio is weighted such that games with less than twenty ratings receive a boost. This is to weight it in favour of games with fewer ratings, so that the final scores are more statistically significant (and so that people get a more evenly distributed amount of feedback).

There are some methodological complications here. Firstly, we can only gain ratings for *ANGELINA*$_5$'s games by reviewing games ourselves. We do this by reviewing them as we would normally, but being careful to leave no written comments that might sway a game designer one way or the other (since it is common to visit and rate the game of someone who has visited and rated you). The second complication is that we can't be certain that all ratings given for *ANGELINA*$_5$'s games are equally diligent and honest in reviewing the game. Ludum Dare operates on the basis of an 'honour code' because it is possible to just deliver quick ratings on many games without playing them, in order to boost your position in the Play And Rate ordering. Finally, in the case of the anonymised entry, we needed to ensure that both games would not rise to the top of the Play And Rate page simultaneously, as the similarities between the games could become more obvious. To mitigate this, we ensured that we reviewed games under

	Ranking (TTS)	Ranking (SBP)	% (TTS)	% (SBP)
Overall	500	551	36	29
Fun	515	543	34	30
Audio	211	444	73	43
Graphics	441	520	43	33
Mood	180	479	77	39
Innovation	282	525	64	33
Theme	533	545	32	30
Humour	403	318	48	59

Table 10.1: Overall scoring for *To That Sect* (TTS) and *Stretch Bouquet Point* (SBP). The first two columns show the position in the final entry list (lower is better), while the second two columns show the percentile this places the game in (higher is better). There were 780 total submissions to this track.

both accounts at separate times of the day, at least six hours apart. This meant that each game rose to the top of the Play And Rate rankings, was reviewed, and then sank again before the other game was pushed up.

Table 10.1 shows the standings for *To That Sect* and *Stretch Bouquet Point* in Ludum Dare, as well as the percentiles this places each game in. For all categories, with the exception of Humour, the non-anonymised entry is ranked higher than the anonymised entry. We believe that Stretch Bouquet Point outperforms To That Sect on humour unintentionally, because the audio content surprised and bemused many of its players, judging by the comments left by them underneath the game's entry. We believe that this disparity in ratings shows that there is a clear *positive* bias towards creative software in the domain of videogames, which is at odds with similar experiments conducted in other creative domains [40][89].

These results also give a rough estimation of game quality for $ANGELINA_5$, although the presence of positive bias does call this into question somewhat. The bias is also evident in other areas of the ratings. For example, To That Sect ranks in the 64^{th} percentile for innovation, when the game is very evidently not innovative at all. Combined with comments praising the innovative nature of $ANGELINA_5$, this suggests that reviewers found it hard to separate To That Sect from $ANGELINA_5$ when reviewing, and ended up rating the system rather than the game it had created, despite being directly asked not to in the game's description.

Despite this, we believe that some of the ratings can be considered hon-

	% (TTS)	% (JFG)	% (CAU)
Overall	36	29	27
Fun	34	26	27
Audio	73	74	79
Graphics	43	36	30
Mood	77	81	-
Innovation	64	59	-
Theme	32	26	-
Humour	48	51	-
Coolness	67	63	26

Table 10.2: Percentile data for *To That Sect*, $ANGELINA_5$'s non-anonymised December 2013 entry, *Jet Force Gemini*, its April 2014 entry, and *Cut And Upside*, its August 2014 entry.

est and supportive. Discounting Innovation as anomalous, $ANGELINA_5$'s best-performing categories were Audio and Mood. This is a clear strength of this iteration of *ANGELINA*, with good justification and selection of music through emotive connections with Metaphor Magnet, and very targeted sound effect selection which is made possible by the breadth of the FreeSound database and the improvements made to *ANGELINA*'s ability to break down, expand and interpret a game jam theme. Unfortunately, we were unable to obtain individual ratings data for each reviewer that played $ANGELINA_5$'s games, and so cannot determine more revealing statistics such as variance of each category.

10.4.3 Trends Across Multiple Ludum Dare Entries

ANGELINA entered Ludum Dare a further two times, in April 2014 and August 2014, with no changes made to the system in that time. The results for these entries are shown in Table 10.2 alongside To That Sect's results, this time in percentiles only.

Note the additional row here denoting 'Coolness' and the fact that the third game entry does not contain ratings data for many of the categories. Coolness represents the percentile the game's designer is in when he or she is ordered according to the number of games reviewed. We can see here that in the August 2014 entry, we rated far fewer games than in the previous two jams. This was due to unforeseeable commitments around the time of the jam. This in turn leads to a lower billing on the Play And Rate games page, and fewer ratings overall. This led to some categories not having enough

data to return ratings. As such, the ratings for *Cut And Upside* are less reliable than the previous two entries. We nevertheless include them here for completeness and to draw extensions to some of the trends we have identified.

Most categories see some kind of drop from the debut entry To That Sect to the subsequent entries. The Overall category in particular drops seven percentage points down to the same level as the anonymised entry in the original jam, suggesting that bias is reduced and people are more honestly critical in their ratings upon repeated encounters with $ANGELINA_5$. What is interesting to note is that despite this drop, the ratings for Mood and Audio remain high, actually increasingly slightly in the case of Audio (no Mood rating is available for Cut and Upside so this is hard to tell definitively). We believe this provides evidence for our claims that $ANGELINA_5$ is considered to perform well in these categories and this is not necessarily a case of bias.

One might be skeptical about the Audio rating, however, since $ANGELINA_5$ is not creating the audio from scratch but searching and utilising audio from various sources. We would argue that $ANGELINA_5$ is demonstrating skill in selecting and applying the sound effects and music in its games, however, and this is evidenced by the fact that the ratings for the Graphics category are far lower than Audio, and continue to get lower in subsequent jams, despite $ANGELINA_5$ using external graphics databases in the same way it uses external sound databases. Here, the presence of premade handcrafted content did not artificially increase the ratings in this category. We claim that this shows that performance in the Audio and Mood categories are more likely because of something that $ANGELINA_5$ is doing that results in better games, and not merely a function of the human-made content the system is leveraging.

10.4.4 Qualitative Review Analysis

In addition to the scores recorded by the Ludum Dare site, it is also worth considering the written comments left by some reviewers underneath the Ludum Dare submission. Reviews for *To That Sect* largely balanced positive with negative remarks. No comments were universally negative, tempering any criticism with positivity. For example, one comment states that "Angelina seems really good at creating an atmosphere with both sound

and visuals. But the game part of it seems a bit lacking still." This praises the game for its use of visual media and sound content, cushioning the blow for the criticism that the gameplay of running around a maze and looking for the exit is not as fun as they had expected or hoped. Another comment states: "The game itself is too simple. It seem the AI got the mood, but not the [game]play." *To That Sect* highest rating was in the Mood category. We consider this to be quite an achievement for the system, and we believe that the process of music selection through emotion recognition plays a particularly important role in making Mood one of $ANGELINA_5$'s strong points.

There are 33 comments in total on the page for *To That Sect*[1]. Sorting them into four categories – entirely positive, entirely negative, mixed, and neutral/other – we find that 18 of the 33 comments are entirely positive, 14 a mix of positive and negative comments, and 1 comment is neutral (a suggestion about solving a technical bug in the game). None of the comments are entirely negative.

Comments on *Stretch Bouquet Point*, where the reviewers did not know that the author was a piece of software, have a different and altogether more passive-aggressive tone. Some are outright negative, such as "this was a rather annoying experience." Others are more indirect in their criticism, which we interpret as the commenter attempting to shield the developer from directly being insulted while still noting their bemusement at the game. One comment states, for example, "You made me feel something there. Don't make me put it into words though." while another says "This was certainly an experience."

It's possible that the tempered tone is because many entrants to Ludum Dare are just starting out in game development, and reviewers do not wish to directly insult or discourage novices or young people. Because the entries are done through semi-anonymous usernames, it's hard to tell where a game is coming from, and reviewers may therefore be cautious before directly criticising someone's work (the author has experience of this – direct criticism of his own submissions to Ludum Dare come mostly from people who know him well, and feel capable of discussing and sharing their opinion frankly). It should also be noted that there are, of course, differences between To That Sect and Stretch Bouquet Point. Stretch Bouquet Point

[1] http://www.ludumdare.com/compo/ludum-dare-28/?action=preview&uid=29184

includes much stranger and more powerful audio clips than To That Sect, and is considerably less subtle – very loud chanting starts the second the player begins the game. We did not select *Stretch Bouquet Point* to be the anonymised entry – $ANGELINA_4$ created both games in turn, and the first game it created was always intended to be the non-anonymised submission. Nevertheless, it is possible that had the games been swapped and *Stretch Bouquet Point* been the non-anonymised submission, responses would have been different. This is difficult to ascertain through open submission to Ludum Dare, however.

There are 21 comments in total on the page for Stretch Bouquet Point[2]. Sorting them into the same four categories as before – entirely positive, entirely negative, mixed, and neutral/other – we see that 2 of the comments are entirely positive, 4 are entirely negative, 3 are a mix of positive and negative comments and the remaining 12 are categorised as neutral or other. This includes comments with no content but implying a judgement on the game, for example "Umm... ...." While the comment has no explicit praise or criticism, the implication is that the game was confusing or strange to the reviewer. We categorise these comments as neutral although even here they lean towards a negative assessment.

The balance of comments between the two submissions definitely suggests that people were less willing to be critical of *ANGELINA*'s entry to the game jam when they were aware that it was created by a piece of software developed in the course of scientific research.

ANGELINA's entry to Ludum Dare 28 received some attention in the game development world, in the press (such as [39]), social media and the Ludum Dare community specifically[3]. The last comment on Stretch Bouquet Point, posted less than twelve hours before the results were announced, reads "Hah. You're a control, aren't you?" – in other words, members of the Ludum Dare community are now well aware that *ANGELINA* is entering the jam, and because its games all have a similar structure and visual style, repeating the anonymised experiment was considered to be difficult if not impossible.

The reviews for *Jet Force Gemini* have a similar tone to those for *To That Sect*, although more of the comments convey an awareness of what

[2]http://www.ludumdare.com/compo/ludum-dare-28/?action=preview&uid=32167
[3]https://twitter.com/ludumdare/status/420677578383302656

ANGELINA is and some mention having played or read about its entries in Ludum Dare 28. The game received 28 comments in total, which were categorised as with the previous entries. 13 comments were entirely positive, 2 comments were entirely negative, and 11 comments were a mix of positive and negative comments. The remaining 2 were neutral or unrelated - asking for a Linux build of the game, for example.

The presence of two completely negative comments for *Jet Force Gemini* suggest that the positive bias weakened slightly between *ANGELINA*'s debut in Ludum Dare and its second entry. This argument is strengthened by some of the comments explicitly stating that they perceived a drop in quality or that *Jet Force Gemini* offered nothing new over *ANGELINA*'s past entries. The version of $ANGELINA_5$ that entered Ludum Dare 29 is almost identical to the version which entered Ludum Dare 28 save for some slight improvements to theme parsing and level design, and some bugfixes. This all means that the games are very similar. Despite this similarity, other reviewers felt that *Jet Force Gemini* was a better game than *To That Sect*, and said so in their comments.

10.5 Cultural Impact

Research in the artifact generation paradigm of artificial intelligence [29] often focuses on a search for objective quality metrics that can indicate how good a piece of software is at producing things of worth. This is often quite effective when the artifacts in question are in domains governed by precise equations and the output can be quantifiably evaluated. As Eigenfeldt and Pasquier note in [40], such notions of optimality are often lacking when the output is a creative work, leading to a search for alternative evaluation criteria, some of which we explored in §4.6. Another important factor is to observe not the work itself but the influence of the work on the context it is in, and to try and ascertain whether or not *ANGELINA* has affected the modern culture of videogames, if at all.

10.5.1 *ANGELINA* as an Exhibit

In August 2014 $ANGELINA_5$'s Ludum Dare entry, *To That Sect*, was featured in an exhibition at REVERSE in New York City[4], hosted by a games and arts group called *babycastles*[5]. The exhibition was described as an event that:

> *...explores the the wonder, banality, comfort, humor, and terror that can arise, often simultaneously, out of designed systems.*

$ANGELINA_5$'s presence in the event in particular is described as follows:

> *...even step into a world imagined, itself, by a system.*

These somewhat charged descriptions of both the exhibition and the work itself paint a particular picture of *ANGELINA* as something mysterious and perhaps worthy of science fiction. Nevertheless, its inclusion as part of this exhibition shows that the research itself is taken seriously as a new frontier in the culture and art of videogames, even if $ANGELINA_5$ itself is not yet accepted as an individual creator – Michael Cook was listed as the primary exhibiting artist, rather than the software itself.

10.5.2 *ANGELINA* as a Gendered Icon

ANGELINA's name was originally chosen out of a desire for a humourous acronym[6] but after the project moved increasingly towards Computational Creativity research and interacted more with the public, the fact that its name is also a name given to people began to present issues of representation and questions as to whether *ANGELINA*'s name was offering a false sense of humanity to people who played its games or otherwise observed it. In February 2014, Alexis Ong wrote an article for Dazed Digital entitled *Ten women reshaping modern tech* in which it named *ANGELINA* alongside female academics and artists. Ong opens the article describing those named in the list as 'exploding online gender stereotypes' [96].

ANGELINA is frequently referred to using gendered female pronouns, a practice that we ourselves have struggled to stop doing, since the natural

[4] http://reversespace.org/safety-in-nebulous-710-83/
[5] http://babycastles.com/
[6] 'A Novel Game-Evolving Labrat I've Named *ANGELINA*'

flow of conversation using a person's name simply causes one to default into speaking in such a way. Ong's article created some discontent among many people in the games industry who were aware of *ANGELINA*, as its publication came at a time of increasing awareness of diversity and representation issues in the games industry [134]. Many, ourselves included, felt that including a piece of software in such an article was problematic because it took a spot that could easily have been occupied by a person who perhaps deserved the spot more than a piece of software that had been accidentally gendered at some point in the past.

This shows a different kind of impact on the domain of videogames, perhaps one that we do not see very often but one that we feel will be seen increasingly often as Computational Creativity research produces systems which interact in such direct ways with communities of people. *ANGELINA* is simultaneously seen as equal and unequal. As researchers, we struggle for *ANGELINA* to gain validity and recognition as a game designer, but feel uncomfortable when it gains recognition for other things, such as a humanity that it lacks and can never possess, at least in a literal sense. Despite this, other opinions have been voiced on the subject, such as by videogames and social justice researcher Amanda Phillips, who wrote about *ANGELINA* after encountering the research at the AIIDE conference [100]. Phillips writes:

> *One particular line from Cook's talk got me thinking: he said one of his struggles is to get Angelina recognized as a legitimate game designer. Against a backdrop of human women struggling to achieve legitimacy in the games industry (and games journalism and gamer culture), the accident of Angelina's gender becomes quite a bit more complicated – and potentially problematic. I'm looking forward to watching how this project develops and thinking about how it might have a place in commentary or even intervention on the current troubles in game development.*

In Phillips eyes, the fact that *ANGELINA* is struggling for acceptance as a game designer provides an interesting twist to its accidental gendering as female, mirroring the difficulties faced by many women in the games industry to gain a similar level of acceptance that *ANGELINA* is searching for. Both Phillips' reaction and the collective reaction to Ong's piece shows that *ANGELINA*'s status in the games industry, while confusing and exper-

imental, is provoking discourse on topics that perhaps have not been faced before. We see this as ultimately a positive outcome for the system, albeit one we did not foresee when the system was originally conceived.

10.6 Evaluation In Automated Game Design

At the beginning of this chapter we stated that there existed no agreed-upon routes to evaluation for procedural content generators, and that evaluation in computational creativity research is still lacking in consensus. As a result, we tried as many different approaches to examining and evaluating *ANGELINA* as possible, as a way of investigating how automated game designers can be evaluated.

Generating complete games is not a straightforwardly technical task. The outcome cannot simply be evaluated quantitatively in the absence of players – the 'observers' in the definition of computational creativity we gave in chapter 4. Attempting to evaluate using metrics like fitness alone do not adequately capture whether or not an automated game designer is achieving its goals. Fitness merely indicates that the underlying evolutionary systems are functioning correctly, but as we saw from the follow-up studies, fitness does not necessarily correlate with enjoyment, or the perception of quality.

Just as we augmented co-operative co-evolution with additional design phases as *ANGELINA* developed further, we similarly adjusted our evaluatory approaches to capture higher-level qualities of the games being developed – such as the system's ability to be novel, to create new ideas or knowledge, or to act in ways that others would consider skilful, appreciative or imaginative [20]. Examining whether the system is having an impact on the creative community in which it is situated, for example, is a valuable way of assessing whether a system is making progress at the highest level, on a social or cultural level. Although this approach is perhaps impractical in a general sense, smaller-scale kinds of impact are also revealing. The discussion of surprise earlier in the chapter is a good shorthand for demonstrating that an automated game design system is capable of producing output that is innovative or unexpected in some way, even if that innovation is at a more local level rather than innovation on a global scale (see Boden [9] for a discussion of creativity on different scales).

Ultimately, we are most satisfied with our evaluations of *ANGELINA*

through peer review, through the ratings process of *Ludum Dare*. Although there are interesting methodological complications to evaluating *ANGELINA* in this way, such as the nature of the reviewers' self-selection or the difficulty of obtaining voting data, it represents a promising method for the future of evaluating automated game designers, particularly those working in the context of computational creativity research (which we would argue all automated game designers are, to some degree). Peer review through game jams and similar events are powerful because they use a large population of potential reviewers with a variety of experience in the medium as creators; they provide a mix of qualitative and quantitative through numerical ratings and verbal discussion; they offer the potential for *ANGELINA* to reciprocate and participate in rating and reviewing in the future, too. Ludum Dare's structure as a system of creativity, interacting peers, reviews and promotion for the winners, closely mimics ideas of social creativity described by Saunders in [106]. In the future such evaluations might offer both evaluation and new research opportunities at the same time, by examining the system's relationship with the community it finds itself in.

10.7 Summary

We began this chapter by claiming that evaluation is difficult in the fields of research that this work is situated within, namely Computational Creativity and procedural content generation. This led us, over the course of the work described in this thesis, to pursue a variety of different evaluatory methods. While none of these methods leads to a strong conclusion about *ANGELINA*'s ability to design games of high quality autonomously, we believe that our mixed evaluation approach shows many different facets of *ANGELINA*, and shines a light on the many and conflicting aspects of research in automated game generation, which future research in the area will surely encounter as well.

We looked at the improvement of fitness scores as an indication of an evolutionary system's performance, and then compared this to surveys done with players about perceived game quality to see if a connection could be made. The connection was difficult to make, which we believe is partly due to the simplicity of the games generated by $ANGELINA_2$, but also sheds light on the complications of evaluating automatically designed games, with

participants finding it hard to critique only the parts of the game that were designed by the software.

Taking inspiration from both procedural content generation literature and Computational Creativity research, we considered the expressive range of *ANGELINA*. We argued that the breadth of *ANGELINA*'s output throughout the course of this thesis shows the wide applicability of computational co-evolution and our methodology of breaking procedural generation tasks into smaller evolutionary systems. We also discussed expressivity at a lower level, showing that *ANGELINA* is capable of surprising its audience and its creators, both intentionally and unintentionally.

We extended the discussion of game quality that began when considering evolutionary fitness, and looked at two further surveys we conducted to assess $ANGELINA_4$'s sense of level design and difficulty, and $ANGELINA_5$'s interactions with the peer review system of Ludum Dare. In both cases, we find that firm conclusions are hard to draw, but we also see indications that *ANGELINA* is capable of designing levels that people find interesting and enjoyable to play, and that despite the unclear evaluation conditions raised by the surveys of $ANGELINA_2$, there are still strengths to the system that reviews are able to identify, as evidenced by the reviews of $ANGELINA_5$'s Ludum Dare entries and their high ratings for Mood and Audio.

Finally, we discussed cultural impact, perhaps the hardest aspect of a project to evaluate. How has *ANGELINA* impacted the culture and thought processes behind videogames, if at all? We discussed two aspects in particular, relating to its struggle to be recognised as an individual artistic entity, and the response to it being treated as a gendered entity by the technology and videogames community. We believe that as we push *ANGELINA*'s interactions with creative communities further, through projects like Ludum Dare, we will see more of these issues arise as people encounter and interact with the system in different ways.

Evaluating such a large and multi-faceted project is difficult, and this chapter does not offer firm answers to many of the questions raised in this thesis. For this reason, the reader may find more answers in Chapter 12 in which we discuss the future work that is likely to lead from the unanswered questions that remain. Many of the issues raised in this chapter directly influenced the intended trajectory of future work.

11 Related Work

11.1 Introduction

Throughout the background chapters of this thesis, we introduced many important concepts and prominent research that inspired and directed the development of our project. *ANGELINA* has roots in many different areas, including videogames, computational creativity theory and computational evolution. In this chapter, we take the opportunity to more specifically consider work related to *ANGELINA* and look at our work in the context of these other projects.

In section 11.2, we look at work undertaken in the field of procedural content generation to implement automated game design systems, some focusing on specific elements of a game's design, others attempting a more holistic approach similar to that attempted by us with *ANGELINA*. We assess the kinds of activities these different projects undertake when designing or generating content, and consider how *ANGELINA* differs from each, both from a technical standpoint as well as a cultural and creative one. Often we find that while the technical aims of the projects align with those of *ANGELINA*, the overriding philosophical objectives of our work continue to set it apart from contemporary research, and the desire to have the software accepted as a game designer in the future provides a unique motivation behind some of our design decisions in contrast to other work.

In section 11.3 we consider projects from the field of Computational Creativity, and look at how they developed over time, and how they dealt with issues of evaluation. In particular, we focus on how the two systems – the artificial painter The Painting Fool and the artificial soup chef PIERRE – dealt with evaluation directly with the public, and how they interacted with communities of experts and laypeople/consumers alike. While not directly comparable in the traditional sense of 'related work', the results from the two systems offer interesting comparisons for *ANGELINA* nonetheless.

11.2 Automated Game Design

11.2.1 Togelius and Schmidhuber

In [125] the authors describe an unnamed system, henceforth referred to as TS, which designs simple 2D arcade games. We mentioned this system earlier in chapter 5 – it served as the inspiration behind $ANGELINA_1$ as well as other work outlined later in this section. We describe below the design space of the TS project, as it has been influential as a baseline for automated game design tools since, including *ANGELINA*. The following is an edited description of the design space from [125]:

- The game takes place on a discrete grid with dimensions 15×15.
- Each cell on the grid is either free space or a wall.
- A game will run for a finite number of time steps, starting at $t = 0$ and continuing until either $t = tmax$, $score >= score_{max}$ or the flag *agentdeath* has been set.
- At the beginning of a game, one of the cells (randomly selected among the 4×4 centralmost cells) contains the agent. At any time step, the agent can and must move one step either up, down, left or right.
- At the beginning of a game, zero or more cells are occupied by *things*. These cells are randomly chosen from the free space on the grid, except that no thing starts closer than two steps from the agent. Every thing can be either red, green or blue. Things can, but do not have to, move one step every time step.
- Each colour has an associated movement logic that determines how things of that colour move; a collision effects table determines what happens to things when two things of the same or different colours collide, or when a thing and the agent collide; a score effects table determines how the score changes when a collision between two things or between a thing and the agent occurs.

The design space of $ANGELINA_1$ inherits many of these features. We use the terminology *entity* in place of 'thing'.

TS begins by randomly generating a population of game rulesets, by setting the values for the game's specification to random values. This population is then evolved over 100 generations, and each generation is evaluated using neural network player controllers which attempt to learn the game's ruleset through repeated play. The evaluation measures how quickly, and to what degree, the neural network was able to learn how to play and win the game, if at all. This is inspired by Koster's Theory of Fun [75] in which he states that fun in videogames is derived directly from how difficult or easy the game is to learn and how much learning potential it continues to afford over time. We discussed Koster's work earlier in chapter 2.

The learning process and design of the neural networks was such that a typical evolved controller 'sometimes wins the game, and other times fails through some mistake of varying severity' [125], thus achieving a balance between trivial failure and trivial optimisation for most game rulesets. The fitness rewarding the performance of a neural network is the fraction of the player's final score for that run divided by the target $score_{max}$. Thus, controllers that make progress in gaining score are rewarded more until they eventually complete the game. The best fitness achieved through the evolution of player controllers for a given number of generations indicates how rapidly the neural network evolution was able to find controllers which performed well at the game, which the authors argue informs how learnable the game is and thus how fun.

The authors note that 'in the current rule space, the vast majority of games are unplayable' in the sense that they are either far too hard or far too easy for a human player, regardless of the neural network performance. They cite several examples of good, curated games however, including *Chase The Blue* (a human-assigned name) where the player must catch a blue object twice in a short timeframe.

While $ANGELINA_1$ inherits the design space from TS, it expands on it considerably, attempting to lay out the initial configuration of game elements as well as design parts of the levels. This was later developed in subsequent versions of *ANGELINA* to more deeply alter and vary parts of a game such as the specific value of variables governing game mechanics, or the aesthetic and decorative aspects of the game. This progresses *ANGELINA* as an automated game designer on both a technical level (in systems such as $ANGELINA_4$, see chapter 8) and an artistic or creative

level. It should be noted however that TS only aims to design rulesets, and the authors make no claims of it being an automated game designer.

The philosophy of using learning and neural networks as evaluators also differs considerably from the approach we have taken. $ANGELINA_1$ also used player controllers to evaluate evolved game designs. However, these controllers were statically defined to have certain properties which were intended to investigate particular aspects of the games. Learning is an appealing means to evaluate game designs because it is extremely generalisable across all kinds of game, and is fairly high-level in terms of its application (it doesn't differentiate strategic thinking from reflex action, which is useful in comparing games of different types).

We avoid using learning as a metric in *ANGELINA* for two reasons primarily. Firstly, we don't believe learning-as-fun adequately captures enough elements of why people play games, and certainly doesn't capture why people *design* games. While it might help to evaluate a game in retrospect, people have very different motivations for designing games, and this is something we ultimately wish to capture in *ANGELINA*. Games can be designed to express a view, convey a message, instil a feeling or sensation in the player, experiment with an aesthetic or mechanic, or for the designer's own learning and self-improvement. While *ANGELINA*'s evaluatory process is simplistic currently, we intentionally kept it basic so that it can be built on in the future by the system itself. We want to keep our own prescription of what constitutes fun or quality in a game to a minimum.

The second reason is that learning is difficult to quantify. TS represents an interesting approach to the problem and a good attempt at using learning in a concrete generative system. Nevertheless, it's hard to connect a simple neural network's ability to learn to that of a person, particularly when paired with a fitness function that only deals with score, and Schmidhuber's 'Theory of Artificial Curiosity' which further complicates the model by adding in subjective theories of robotics. Learning is a complex process that we don't fully understand, and while we believe it will be useful in future work in the field of automated game design, it feels premature to use it now. With that said, we expect models of learning to be useful in designing difficulty curves and balancing games in the future of our work.

11.2.2 Game-O-Matic

In [128] Treanor et al. describe the *Game-o-Matic* (GOM), a tool for authoring videogames by describing relationships between objects that will be in the game. The tool was originally developed to help journalists rapidly develop *newsgames* that express ideas or elements of a news story. The emphasis was on developing a tool that was usable by non-programmers, but still capable of developing interesting, playable games that conveyed something meaningful about an event or a situation.

Game-o-Matic's development is based on a philosophy of game design called *proceduralism*, not to be confused with procedural content generation which we described in chapter 4. Proceduralism is a term proposed by game designer and critic Ian Bogost to describe an approach to game design in which the game's systems convey the meaning of the game, rather than its appearance or presentation. In [10] Bogost writes of proceduralist games:

> In these games, expression is found in primarily in the player's experience as it results from interaction with the game's mechanics and dynamics, and less so (in some cases almost not at all) in their visual, aural, and textual aspects.

Bogost also makes other claims about proceduralist games in [10] (including, interestingly, "the strong presence of a human author" which is of relevance to this thesis in a broader sense perhaps). However the primary factor is the belief that meaning is conveyed through mechanics and systems, rather than the audio and visual content of the game.

The process of interacting with the GOM begins by drawing up a conceptual diagram of the relationship the user wishes to express through a game. Figure 11.1 shows an example of such a diagram. Nodes in the graph represent objects in the world; in the figure, the concepts are related to a food chain of corn, cows, burgers and humans. Vertices in the graph represent relationships between concepts. Corn feeds cows, cows make burgers, and humans eat burgers.

This graph is then taken by the Game-o-Matic and processed to create a game. The system attempts to find graphics in a database that match the concept nodes; if no graphics can be found then coloured circles with text labels are used. The relationships linking concepts are then mapped

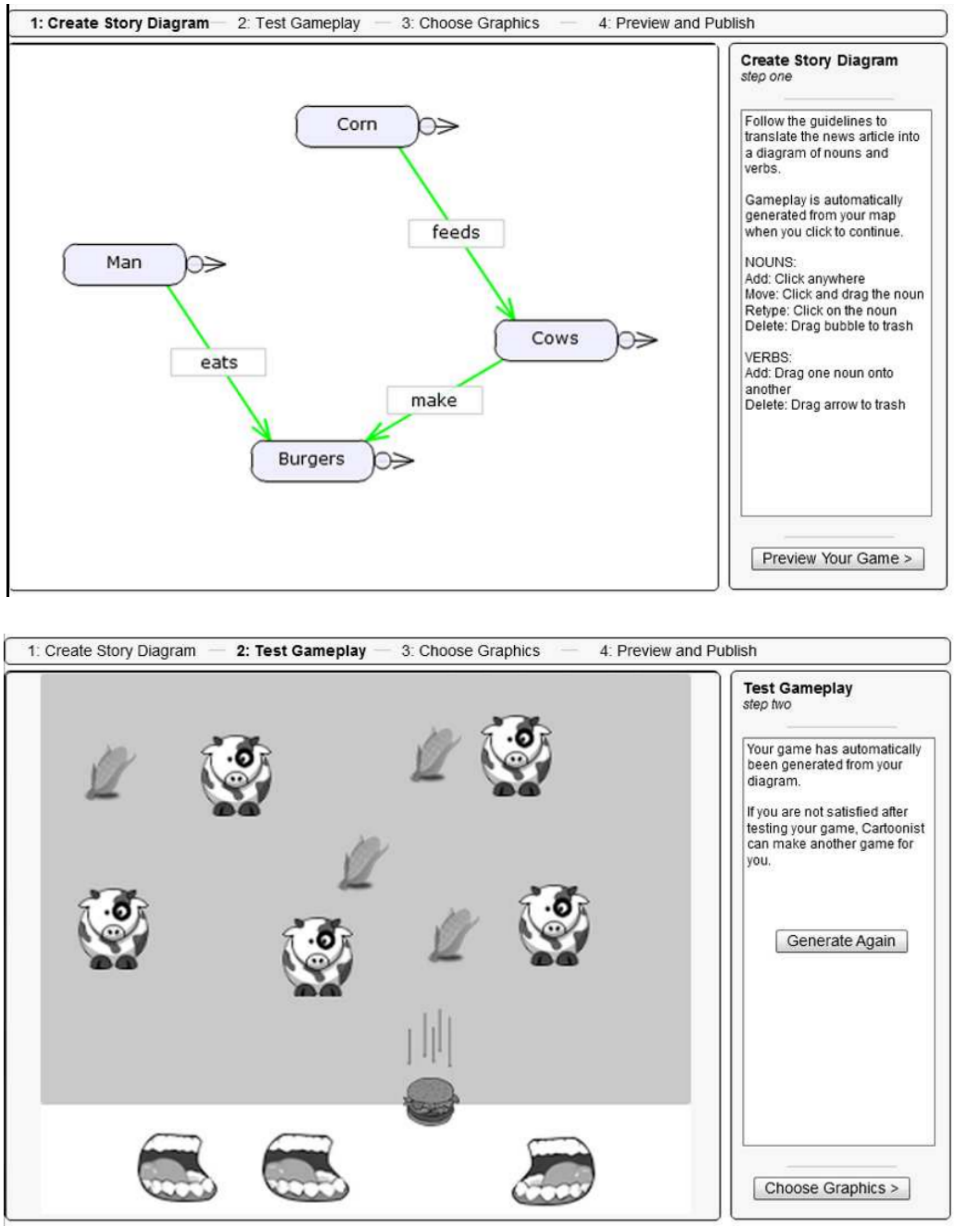


Figure 11.1: Screens from the *Game-o-Matic*. Top: A concept graph showing related concepts. Bottom: A game based on that concept graph.

by the system to known mechanics in GOM's database. These mechanics, when combined with thematic information from the game's concept graph, form what the authors call *micro-rhetorics*; small independently-meaningful units of gameplay. For example, if a cow touches a corn and the corn disappears, this conforms to the micro-rhetoric of the cow eating the corn. This relationship is meaningful in isolation, but can also be inserted into a larger game to supply meaning to a bigger system. Figure 11.1 shows one such game, along with the concept graph tools that are supplied with versions of GOM. Here, the relationship between corn and cows is in the larger context of a system in which cows can be eaten by people in the form of burgers. The connections between these elements all contribute to a particular kind of game and a particular kind of meaning (even if, as in this case, that meaning is quite simple).

On a technical level, GOM and *ANGELINA* share some similarities in that they both aim to produce complete games and design multiple aspects of the games – GOM designs both the game's mechanics as well as its visual content, for example. The GOM works sequentially, designing each aspect of the game in stages, while *ANGELINA* attempts to design all aspects of its games simultaneously, allowing content produced by one part of the system to influence potentially all other content generation processes, and to be influenced in turn by those systems too.

On a philosophical level, the aim of the GOM project is to create a tool that is used by people to design games, positioning the GOM as a design *assistant* rather than a standalone game designer. This is an elementary distinction of *ANGELINA* and the GOM. People are frequently contracted to create games in which they have no creative input themselves, for example. In the case of software, however, it is important to consider the sources of knowledge and data, particularly when considering the creativity of a piece of software. The GOM relies heavily on the input concept graph to make sense of the world it is depicting and understand the relationships between objects. *ANGELINA*, meanwhile, aims to work from much looser input information and to try and mine this knowledge about the real-world itself.

While this might seem like a minor difference, relating knowledge about the real world to a system of game mechanics is something we consider to be an AI-complete task [32]. The authors of the GOM carefully created

a database of micro-rhetorics specifically relating game mechanics to real-world concepts, and even with this rich background of hand-crafted data the GOM can only recognise 17 relationships between objects in its input concept graphs. Although *ANGELINA*'s results are often bizarre or hard to interpret, it demonstrates a fundamental difference in the goals of the two projects.

It should be noted that at no point do the authors claim that the GOM is a computationally creative system, nor do the project aims include claims of creativity. The GOM is a powerful tool for enabling people to design a wide array of games with little programming knowledge. *ANGELINA*, meanwhile, is a project which aims to produce an autonomous, standalone system capable of designing games and being recognised and treated as a game designer. At this early juncture in the field of automated game design the systems seem similar, but we feel that as the field develops, projects such as the GOM and *ANGELINA* will be seen as pursuing separate but complementary research directions in the field.

11.2.3 Nelson and Mateas

In [92] Nelson and Mateas describe an unnamed system which generates simple *microgame* in the style of *WarioWare* [95]. WarioWare was a game composed of many microgames, in which the player had only a few seconds to complete a simple task such as pressing a button a number of times or guiding a car around a racetrack. The microgames all take place on a single screen – the player has to assess and complete the task in a matter of seconds, so simplicity is essential.

The system presented in [92] (hereby referred to as NM) takes a verb and noun pair as a primitive description of the game (such as 'shoot ducks') and then creates a simple single-screen microgame based on the input words. The authors express two aims for games created by NM:

> The game should "make sense" in terms of the roles its thematic elements are playing, and it should be "reasonably close" to what the user requested.

NM employs a database of pre-made mechanics which are tagged with certain verbs or actions, contextualised based on the role of the player in

each scenario (thus a game where one object chases another might relate to 'attacking' if the player is controlling the chasing object, but 'escaping' if the player is controlling the object being chased). NM searches ConceptNet [81] to find the verb in the database that has the shortest number of links to the verb chosen by the user (as explained below). Once a verb has been chosen, the mechanic it is associated with it selected as the core of the game.

The mechanics have defined nouns that they require in order to complete a game. To use the previous example of one object chasing another – called *Avoid* in the authors' work – this requires two objects, one to be chased and another to do the chasing. Three competing criteria are used to select nouns:

> We choose nouns from those for which we have sprites that meet the constraints of the game and are close to what the player requested. [92]

By "constraints of the game", the authors mean that the chosen nouns should be close to nouns in ConceptNet with certain properties associated with the game mechanics in the hand-crafted database. The *Acquire* mechanic template involves two nouns, with the player controlling one noun and collecting multiples of the other noun. According to [92], the two nouns should be connected in ConceptNet via the relationships `DesireOf` – in other words, the player object desires the secondary noun according to ConceptNet. As with the verbs, the noun selection is also traded off against whether the nouns are close to the original request by the player. Distance of nouns is measured using hypernym/synonym relationships in WordNet.

Although NM designs smaller games with fewer components than *ANGELINA*, the two systems have similar approaches and the use of ConceptNet and WordNet to narrow free input down to a database of understood concepts feels closer in spirit to our approach with *ANGELINA* of designing a system that can ostensibly design a game about anything. Our approach is partly distinguished by the development of Mechanic Miner, as described in chapter 8, as this allows us to progress *ANGELINA* past the point of needing hand-designed game mechanics and enables it to generate mechanical concepts on its own.

ANGELINA's use of CCE to design game elements simultaneously also sets it apart from NM, since this again allows different game elements to

interact with one another and influence the process of game design. This is less integral for NM since there are few interrelated parts, and the stripped-down nature of the microgames means there is less of an emphasis on the design of large spaces. Nevertheless, it is another feature that sets the two systems apart.

Finally, as with all systems described in this subsection, the authors of [92] do not explicitly intend for NM to be recognised as a creative system, nor do they aim to identify the software as an independent game designer. Although largely a philosophical stance, this has clearly impacted the development trajectory of *ANGELINA* and raised the importance of things such as commentary and framing generation in the system, something which is not present in systems like NM. This is not a weakness in any of the other systems; it simply highlights the contrasting aims of *ANGELINA* and other proto-automated game design projects.

11.2.4 Variations Forever

In [110] Smith and Mateas describe *Variations Forever* (VF), a 'ruleset generator' which uses answer set programming (ASP) [83] to explore a design space of simple 2D arcade games. Unusually for academic research, VF is framed as both a game and a research project. As a game, VF tasks the player with exploring a search space of game mechanics by playing and completing short generated games. As games are completed, new game concepts are unlocked in the design space, allowing the player to potentially uncover them on repeated playthroughs. The authors cite the independent videogame *ROM, CHECK, FAIL* [41], a game in which several mechanical and decorative elements rapidly change between famous game archetypes, as inspiration.

A major contribution of the work in [110] was the introduction of answer set programming to the space of procedural content generation techniques and automated game design, which the authors argued offered many appealing features for generative software. ASP allowed for constraints to the design space to be easily and precisely defined, allowing a space to be 'sculpted' either automatically or by a user. Much of the paper is spent underlining the advantages of ASP and offering an introduction to working with it as a language. We will avoid going into detail about the precise implementational details here and instead limit ourselves to a higher-level

description of VF's operation.

Much like *ANGELINA*$_1$ and elements of later *ANGELINA* versions, VF's design space includes many distinct categories of game content with discrete, hand-designed values for the system to choose between. For example, the game's level space can wrap toroidally like a level from Pac-Man, wrap spherically or not wrap at all. While the entire design space of VF is not given in the paper, below is a selection of the design choices made by the system:

- Agent colour.
- Selection of which agent is controlled by the player.
- Behaviour of non-player agents.
- Movement style of player agent.
- Collision effects between agents and other agents, or agents and obstacles.

Like *ANGELINA*$_1$, VF is inspired by the work described in [125], which is evident through the shape of the games produced by it. Nevertheless the system does have slightly more flexibility – the ASP solver can choose, for instance, to have fewer kinds of agents, and there is a slightly wider space of control schemes and mechanics.

Variations Forever differs from *ANGELINA* in several important regards. Firstly, the generative system is a game in itself, so the aim of the project is to apply the generative nature of the system to provide many different game experiences for a single player. This is often a question raised in the context of *ANGELINA* – whether it will ever be released as a standalone system so that people can play many generated games made just for them. However, the aim of the *ANGELINA* project is to produce a system which behaves as a game designer might; producing finished, static games that are played by everyone in the same way. This distinction is important for several other computational systems such as The Painting Fool [21] and AARON [17].

VF is also difficult to analyse from a technical standpoint because ASP is a monolithic process that is hard to pick apart without describing the underlying processes at work in an answer set solver. Even then, such a solver treats all decisions equally – the decision of whether the player is a

red object or a blue one is the same as whether the objective of the game is to kill or to avoid being killed, and so on. Treating the systems as black boxes, *ANGELINA* and VF are similar pieces of software that both design games, albeit very different kinds of games. *ANGELINA*'s internal structure, however, is more modular and more easily broken down into smaller, comprehensible conceptual pieces. Although from a software engineering standpoint the systems are both very flexible in different ways, *ANGELINA* has a more readily understood narrative of how it produces games, with a process that is perhaps closer to how people imagine themselves producing games.

This confers no technical benefit, but from a philosophical and cultural perspective, it enhances *ANGELINA*'s attempts to become accepted as a game designer. The ineffable nature of intelligent software and algorithmic processes to laypeople (and even to computing experts not versed in particular fields of AI) is a real barrier when attempting to argue that a piece of software is engaging in creative activity – activity that is strongly associated with being human. The 'readability' of *ANGELINA*'s internal processes, in contrast to systems like VF, is a strength that helps in it being accepted as an independent designer.

Finally, VF's games are less sophisticated and polished in some ways than those of *ANGELINA*. While the authors make no attempt to do so, VF's games are nevertheless lacking in real-world context or connection, do not have names or other framing information, and remain very abstract. Abstraction in game design is not an artificial trait nor an indication of poor quality, but we would argue that *ANGELINA*'s push to design games which have a connection to the physical world and human culture shows sophistication in certain areas of game design that comparable systems such as VF lack.

VF is notable in being the only system in this subsection to explicitly refer to computational creativity and connect itself with the research, noting that one of its results was novel to the authors and highly unexpected. The publication of [110] precedes that of work on the FACE model and other contemporary methods for evaluating computationally creative systems. We evaluate VF in [34] using the FACE model [25], and it remains an interesting example of an automated design system from both a videogames and a Computational Creativity perspective.

11.3 Computationally Creative Systems

11.3.1 The Painting Fool

The Painting Fool is a computationally creative painter and artist. The aim of the project is commonly stated as 'to have the system taken seriously as an artist in its own right, one day'[1]. The system is largely the work of Simon Colton, who has also used The Painting Fool as a platform for investigating broader questions of computational creativity [27].

The development of The Painting Fool has taken the system from a series of experiments in rendering, composition and the technical details of painting, through to more current issues of creative autonomy, ideation and cultural relevance. In this section, we are particularly interested in the ways in which The Painting Fool has been exposed and evaluated by external groups. The Painting Fool is not the only artificial artist to have been evaluated by the public or other stakeholders, however the project's original aims of autonomy, independence and integration in a creative community have directly inspired and influenced the development of *ANGELINA*.

In [28] Colton and Ventura describe an exhibition called *You Can't Know My Mind* which was held in Paris in 2013. The exhibition featured demonstrations of several computational creativity systems, but its primary purpose was to exhibit work by The Painting Fool. The exhibition lasted for a week, and included live portraiture by the system, which would ask those attending the exhibition to sit for a painting, but would not always paint portraits depending on fluctuations in the system's internal 'mood'.

The exhibition was carefully designed to introduce the exhibition's visitors to the notion of computationally creative software. Posters were put up explaining some of the behaviour of the software, and the motivations behind it, and some aspects of the system were anthropomorphised (such as the aforementioned 'mood') to try 'to enable [the public] to make an informed opinion about whether it was appropriate to call the software 'uncreative' or not.' The authors of [28] note that, after talking to many of the attendees of the exhibition, they received no salient answers as to why the software should be considered uncreative, which the authors believe 'indicates how well we handled public perception of The Painting Fool during the festival'.

[1] http://www.thepaintingfool.com/

Dedicating an exhibition to the output of a single piece of software is a significant achievement, and can be seen as taking a major step towards integrating the software with an artistic community, and subsequently being seen as an artist in its own right, as a part of that community.

While *ANGELINA* has never had an exhibition dedicated to its games alone, this is uncommon for game designers in general. $ANGELINA_4$'s *To That Sect* was included as part of an exhibition in New York by Babycastles[2], which describes itself as an 'independent games arcade' and has wide cultural influence in the videogames community and organises many exhibitions and events around the world. Although we might see exhibitions as analogous in the worlds of art and videogames, one could argue that a closer equivalence to a dedicated art exhibition would be for *ANGELINA* to have a booth at a major games event such as GamesCom, Penny Arcade or Indiecade. Booths dedicated to a single game developer are common, and offer a way for the general public to both interact with a game designer and to see both its past, current and sometimes future work. For now this remains a goal for the future of this project.

An interesting contrast to consider between The Painting Fool and *ANGELINA* is the relationship between the system in question and the medium that it targets. The aim of The Painting Fool project is to have those who interact with it appreciate the output of the software, and view The Painting Fool itself as one might view any other artist. The *ANGELINA* project has very similar aims, but because videogames are already an inherently technological medium, submissions of *ANGELINA*'s work often confuse whether they are evaluating *ANGELINA* itself or the games produced, as we discussed in chapter 10. Part of the motivation behind $ANGELINA_4$ entering a game jam is that it strengthens the position of the system as a creator and a designer, rather than an intermediate procedural generation system. $ANGELINA_4$ has a Ludum Dare account, enters games, receives ratings like any other designer. In doing so, we try to reinforce the notion that *ANGELINA* is a system that deserves to be treated equally with other designers, in the same way that The Painting Fool looks to be treated as an artist.

[2] http://babycastles.com/

11.3.2 PIERRE

PIERRE is a computationally creative culinary tool designed by Morris et al. and first presented in [89]. It is notable in the context of this thesis as it is one of only a few computationally creative to have been evaluated in a public context.

PIERRE is built on a foundational *inspiring set*, a term the authors borrow from Ritchie's terminology which we discussed in section 11.3. The inspiring set in this case is a database of 4,748 existing soup recipes which the authors collected from online cookery websites such as the Food Network[3]. The authors manually parsed each recipe into a common format by normalising ingredient measures. In addition to formatting the recipes, the ingredients were also roughly categorised both at a high-level (whether they were fruits, vegetables, seasonings and so on) and at lower levels (vegetables might be subclassed into leafy vegetables, root vegetables, etc.). For each ingredient, statistics on that ingredient's usage is also calculated over the entire corpus of recipes, such as the maximum amount used, minimum amounts, standard deviation and so on.

PIERRE uses a genetic algorithm to generate new recipes through one-point crossover and random mutation, with the initial population drawn directly from the initial inspiring set of pre-made recipes. The recipes are evaluated using multi-layer perceptrons (MLPs) trained on the user ratings of the recipes which formed the original inspiring set, with additional randomly-generated recipes added in with a zero-rating to add more breadth to the MLP training, since even a bad recipe on a cookery website is likely to be of at least a minimum level of quality.

In [89], the authors perform several evaluations of the software, including assessing the software's performance in terms of Ritchie's criteria from [102]. They compare their original inspiring set of 4,748 recipes with a more refined set of only 594 chilli-based recipes, and then compared the generated recipes output by each system. The authors looked for what they called *rare n-grams*. A rare n-gram is defined as:

> a combination of n ingredients that does not occur in the inspiring set and does not contain a rare (n-1)-gram as a sub-combination

[3] www.foodnetwork.com

They noted interesting results, such as the chilli inspiring set weighting the recipes towards a certain style of ingredient combinations (which they deemed a 'chilli profile') and hypothesised that inspiring sets might be blended to combine flavour profiles.

A particularly interesting element of PIERRE's evaluation is the investigation into the presentation of the recipes and the responses from the general public, as well as selected survey participants. PIERRE's raw output is a list of ingredients and their quantities. These quantities are often counterintuitive to a person familiar with cookery, for two reasons: first, they are often extremely specific, asking for 2.87 teaspoons of a particular ingredient, for example; second, they often ask for unusual or negligible amounts of ingredients, such as 0.26 of a slice of bacon in a huge pot of soup.

The authors designed PIERRE to be able to render its output slightly differently when presenting them to the public - it generated titles for its recipes (such as *Soup Over Bean Of Pure Joy* or *Exotic Beefy Bean*) and rounded its ingredient amounts to normalise their appearance, adding in common phrases such as *a dash of* to replace certain values. In a survey of 38 participants, they found that there was no significant difference in rating the recipes regardless of whether they were presented in their 'raw' form or when adjusted and made more presentable. However, when submitting ten of the recipes to the website *Food.com*[4] the authors found a very different response, noting that:

> the online community was outraged enough by some of the ingredient quantities (e.g. a dash of green beans) – which, though absurd by human standards, would not negatively affect taste – that even without [considering] the quality of [the recipes]... they removed our recipes from the site and suspended our account. [89]

It should be noted that these recipes were submitted to the site anonymously, akin to $ANGELINA_4$'s anonymised Ludum Dare submissions. There are clear parallels here; both systems are submitting their output for evaluation by a community of enthusiasts at the very least, with many users probably considered highly experienced or expert in the domain. In both

[4]http://www.food.com/

cases, the output is met with some level of disbelief and confusion. However, the authors note an important fact that sets PIERRE's experience apart from *ANGELINA*$_4$'s – the users of Food.com are asked to rate recipes without necessarily cooking or tasting them. The authors refer to this as the 'raw' presentation of the work as opposed to the 'cooked' version, and posit that there are analogous states in other creative domains. By contrast, Ludum Dare entries must be playable, and so people were able to interact directly with the finished version of *ANGELINA*$_4$'s work.

This is an important distinction. PIERRE's attempt to introduce a dash of green beans put people off and angered them. Most of *ANGELINA*$_4$'s game design is not explicitly mentioned in the game's commentary, because games are designed to be experienced through interaction and play. One side effect of this is that *ANGELINA*$_4$ does not have to draw attention to many of its design decisions, and so choices that players might find strange – such as making the collectible objects move away from the player so they can't be caught – remain unnoticed by the player. If something does not directly impede the player in a videogame, it seems reasonable that they are willing to ignore it or not comment on it. If *ANGELINA*$_4$ doesn't draw attention to it either, then it allows the system to get away with adding its own equivalent of a dash of green beans to the mix.

PIERRE is the only example of a computationally creative system that we are aware of which has entered its output to be rated and compared directly against the output of humans working in the same field – HR [24] is a comparable example, perhaps; a mathematical discovery system which invented an integer sequence which was included in the Encyclopedia of Integer Sequence. While *Food.com* is not a contest in the same vein as Ludum Dare, the ratings of a recipe influence where it appears in search results and how likely readers of the site are to find or read it.

11.4 Summary

In this chapter we reviewed several projects related to *ANGELINA* either in overall aims, technology used, or issues encountered. We looked at contemporary attempts to automate game design in various ways and to various degrees, and compared their approaches with the methodology we employed for the *ANGELINA* project. We also considered how *ANGELINA*'s place-

ment within the field of Computational Creativity altered its aims compared to other automated game designers. We also looked at projects in the field of Computational Creativity specifically, and considered their interactions with the general public and practitioners of their chosen medium. Ultimately, *ANGELINA*'s engagement with the videogame community is a mix of success and failure, something which seems common throughout other projects with similar goals of integration within a creative community.

The related work described in this chapter motivated and contrasted with *ANGELINA* throughout its development, but importantly they also contributed to the thinking that drives the future of the project. In the next chapter we will cover future work plans which will show the influence of many of the works in this chapter on the directions we wish to take the project in the near future.

12 Future Work

12.1 Introduction

In this chapter, we bring together some recurring themes from the thesis and point out how they lead into the future for *ANGELINA*, Computational Creativity, and the emerging field of automated game design. For the most part, these future directions are already being worked on in some small way by us, mostly through the development of prototypes and small experiments that help to identify the potential of the idea. We take a somewhat optimistic attitude to the topics covered in this chapter, looking forward to what the future might hopefully hold.

In section 12.2, we tackle issues of perception on a deep technical level, by proposing a system which would allow *ANGELINA* to make subjective decisions during the creative process without relying on the opinions of its creators, social media, or random number generation. We describe some early results with such a system, and posit that further work could lead to a large step forward for the perception and autonomy of creative systems like *ANGELINA*.

In section 12.3, we explore another technical question with higher-level implications, namely how to embed real-world knowledge and an understanding of culture into the mechanical systems that make up a game. We describe how, so far, *ANGELINA* has largely kept the design of its games and the surface-level theming completely separate, and discuss how it might be possible to merge the two and produce more meaningful, deeper experiences as a result. We describe a prototype game, *A Rogue Dream*, that makes some initial steps towards achieving such a result.

We then discuss smaller points of future work that are less developed in section 12.4. In particular, in section 12.4.1 we discuss how $ANGELINA_4$'s mechanic generation could be extended with richer code generation abilities, allowing it to generate more complex game mechanics that affect games in

more interesting ways. In section 12.4.2, we discuss ideas for extending *ANGELINA*'s use of commentary and framing information throughout its development process to heighten engagement with players. In section 12.4.3, we pose the question of whether *ANGELINA* could contract and collaborate with other people to create content for use in its games, and the new research questions that could open up.

12.2 Code Generation For Artificial Subjectivity

In chapter 8, we described in detail $ANGELINA_4$, and the metaprogramming system within it that allowed it to select, alter and evaluate the codebase of a videogame in order to invent new game mechanics. We believe that code generation has enormous potential for automated game design, and it also drives forward computational creativity research by opening up opportunities to have software invent using the same language that the writers of the software themselves invented.

Throughout this thesis, we have described many fitness functions that work as selection mechanisms in the various evolutionary species that *ANGELINA* is composed of. A common discussion point when talking to people about *ANGELINA* was the source of these fitness functions, and whether they express the subjective belief of us, the system's designers, about how a game should look or function. This was often seen as limiting the perceived creativity of the software, since it was strongly influenced by the subjective beliefs of someone else.

This led to two philosophies being employed when designing fitness functions for *ANGELINA* – one way was to aim for justifiable objective standards where they existed, for example in $ANGELINA_2$, the use of powerups to traverse a level design is a recognised trope of the Metroidvania genre (recall our discussion of the genre in §2.2.2), or in $ANGELINA_4$ where we used utility as a measure of how good a game mechanic was. Alternatively, we would try and make the fitness functions as general as possible to encompass many different subjective aims for a game. This is most clearly seen in $ANGELINA_5$ where the fitness functions are defined to be intentionally broad so as to not restrict the system too much with the subjectivity of us as designers. This latter approach is perhaps less successful, as it results in games that are lacking in focus on a particular style and often feel

underdeveloped as a result.

Subjectivity is important to creative work – the subjective decisions made by a creator are often what defines the work and provides the human connection that people value so much. Yet our attempts to introduce subjectivity from external sources, such as the integration of social media opinions in *ANGELINA*$_2$, were largely met with bemusement and treated as a curio, rather than something that was of the system itself, e.g. [91]. If we are to incorporate subjectivity within *ANGELINA*, then the source of that subjectivity cannot be another person – it has to come, somehow, from the software itself. The question of how to achieve this remains open. Some researchers feel it is fruitless to claim that software can express subjective opinion; others feel that while expression of opinion is possible, it is hard to escape the Catch-22 situation of embedding personal opinion within the system in the course of achieving this.

Generating Preference Functions

We have begun to investigate alternative solutions to the problem of introducing subjectivity into *ANGELINA*, by generating *preference functions* – code segments which express a preference between two objects – for use in computationally creative software. This means that judgements about part of the creative process, such as which members of a population to select in an evolutionary system, can be made by the system without any overt involvement from the system's designers.

Our existing prototype system generates simple methods, modelled after the `java.util.Comparator` interface in Java. Comparators take two objects, x and y, of a certain type, and then return one of three values which obey some ordering $<_{ord}$:

- `-1` if $x <_{ord} y$
- `0` if $x =_{ord} y$
- `1` if $x >_{ord} y$

Where $<_{ord}$ is defined implicitly by the code of the preference function, such as sorting level designs according to the distance between the start and the exit. All of *ANGELINA*'s fitness functions that we have defined in this

thesis are written in code as comparators that order populations in this way. Comparators are very useful in a wide variety of programs, but are simple to generate because they can only return three values. This makes them a good target domain for experiments towards simple subjectivity generation.

Our prototypical system generates code segments in the C# language using an evolutionary system, with a fitness function that assesses whether a particular piece of code is a 'better' opinion than another. This might seem counterintuitive – to express a meta-level opinion on opinions is presumably to allow our own subjectivity into the system. To mitigate this problem, we have defined several metrics that measure certain features of an opinion without expressing judgements on the nature of the opinion being expressed. These metrics capture what we believe people would consider to be general properties of *defensible* opinions. That is, these features describe opinions that one may disagree with, but that are not open to attack on the basis of logical inconsistency or vagueness. The metrics are:

Specificity The *specificity* of a preference function f for a set of artefacts A is defined as:

$$1 - \frac{|\mathbf{0}_f|}{|P|}$$

with

$$\begin{aligned} P = &\quad \{(a,b) \mid (a,b) \in A \times A \wedge a \neq b\} \\ \mathbf{0}_f = &\quad \{(a,b) \mid (a,b) \in P \wedge f(a,b) = 0\} \end{aligned}$$

Specificity is expressed as the proportion of pairings within $A \times A$ for which f expresses a definite preference; that is, $f(a,b)$ does not return zero. Note that P excludes identity preferences, but it does not assume transitivity on f and it includes reflexive preference, i.e. $f(a,b)$ and $f(b,a)$.

Transitive Consistency The *transitive consistency* of a preference function f for a set of artefacts A is defined as:

$$\frac{|T_f|}{|Q|}$$

with

$$Q = \left\{ (a,b,c) \;\middle|\; \begin{array}{l} (a,b,c) \in (A \times A \times A) \\ \wedge\, a \neq b \wedge b \neq c \wedge a \neq c \end{array} \right\}$$
$$T_f = \{(a,b,c) \mid (a,b,c) \in Q \wedge tight_f(a,b,c)\}$$

Where $tight_f$ holds for a triple (a,b,c) if the triple is transitively non-contradictory under the preference function f. That is:

$$a \geq_p b \wedge b \geq_p c \implies a \geq_p c$$

or any permutation thereof. Transitive consistency is a measure of how well the preference function is internally consistent. A high transitive consistency means that the preference is well-ordered, and expresses a preference that doesn't contradict itself. As with other metrics, this is not inherently good or bad. Low transitive consistency can imply that the preference selects based on unconnected or competing features in the artefacts, which is not uncommon in everyday preferences (indeed, it is quite common in some cases, such as deciding between complex multi-objective alternatives like voting in an election).

Reflexivity The *reflexivity* of a preference function f for a set of artefacts A is defined as:

$$\frac{|P_r|}{|P|}$$

with

$$P_r = \{(a,b) \mid (a,b) \in P \wedge f(a,b) = -f(b,a)\}$$

Reflexivity is a measure of how dependent f is on the ordering of its arguments. A high reflexivity suggests that the preference being expressed is not dependent on the arguments being supplied to it. This discourages arbitrary or randomised judgements in generated code.

Agreement

Two preference functions f_1 and f_2 are said to be in {k,n}-agreement iff:

$$k \leq \frac{|P_{f_1,f_2}|}{|P|}$$

with

$$P_{f_1,f_2} = \left\{ (a,b) \;\middle|\; \begin{array}{l} (a,b) \in P \wedge \\ \left(\begin{array}{l} f_1(a,b) = f_2(a,b) \vee \\ f_1(a,b) = 0 \vee f_2(a,b) = 0 \end{array} \right) \end{array} \right\}$$

Where $|P| = n$. That is, the proportion of pairs (a, b) for which f_1 and f_2 either evaluate the same value or one of them has no preference, is greater than k for a set of inputs of size n. Two preference functions are in *strict* {k,n}-agreement iff:

$$k \leq \frac{\left| P^s_{f_1,f_2} \right|}{|P|}$$

with

$$P^s_{f_1,f_2} = \{(a,b) \mid (a,b) \in P \wedge f_1(a,b) = f_2(a,b)\}$$

The above definition states that the functions precisely agree on preference for at least $k \in [0, 1)$ of the set of possible pairings. This is a good measure of how close two preference functions are on the same sample of inputs.

Generating Preferences

For the preference functions we give here as examples, we used the following combination of metrics for our fitness function:

$$\begin{aligned} fitness(f) &= 0.5 \times reflexivity(f) \\ &+ 0.25 \times specificity(f) \\ &+ 0.25 \times consistency(f) \end{aligned}$$

This was developed through manual experimentation – reflexivity was found to be very important in evolving functions which operated on the arguments to the function rather than making arbitrary calculations that did not rely on what was being evaluated. Again we stress that this objective function is not considered optimal or better in any way. Specificity may be more important in some domains, while totally unimportant in others, for instance – it depends on the nature of the preference functions that the programmer wishes to generate. In our case, reflexivity was found to be important in ensuring a perception of defensibility in the resulting preference

functions. A high weighting for reflexivity might be preferable in many application domains, this will need to be seen in further development and use of these criteria.

Because of the nature of code generation, particularly our code generator's implementation, it is possible for a code segment to either fail compilation, or to throw exceptions during evaluation. While industrial code generation is often structured to never produce non-compiling code, our code generation is not prohibited from dividing by zero, for example, because it used a variable without checking its contents first. We catch and ignore any errors in this process and assign a negative score to the code segment as a result.

Crossover of two code segments uses one-point crossover on the list of code statements making up the segment. This is currently acceptable for the subspace of the programming language specification we cover, although if local variables are introduced in the future, this approach will need revision to avoid constantly introducing scope errors (where a local variable is referenced in the latter half of a function, but its declaration was not carried over during crossover). Mutation is applied by randomly regenerating one of the code statements in the list of statements. As with crossover, once the system's focus moves to more complex method constructions, a more fine-grained mutation process may be required that is capable of making small changes to individual statements in a method.

In order to speed up the evolutionary system, we compile an entire population of preference functions simultaneously, passing each comparator as a separate file along with the template comparators they inherit from. If errors are thrown during the compilation of a particular comparator, they do not affect the compilation of the other files passed. Testing of this batch compilation method showed it was far more efficient than single-file compilation, even when done in parallel, because most of the overhead of compilation is in initialising and shutting down the compiler itself. This may change in the case of generating extremely large code blocks, but we do not expect it to be an issue in the near future.

Example Preferences

Figures 12.1, 12.2 and 12.3 show three preference functions generated using our approach. Their captions describe the purpose of each method, since the actual code is often hard to read and contains unnecessary code segments

that are either never evaluated or have no impact on code flow. These examples were generated from a population of 20 code segments over 15 generations of evolution. When evaluating the preference functions, we supplied a test set of 100 random integers (or characters, depending on the function's type) for the preference function to be tested on.

These figures were generated on simple primitive types, integers and characters. To demonstrate generation of preferences for more complex objects, we used a template class shown in Figure 12.4, which describes a simple Item class from a videogame. Items have damage and speed statistics, a name, and may or may not be droppable. A compact, human-rewritten version of a generated preference function can be seen in Figure 12.5.

To demonstrate the application of the Agreement metric, Figure 12.7 shows a human-rewritten version of another preference function generated using the system. However, this function was generated with a modified fitness function:

$$\begin{aligned} fitness(f) &= 0.25 \times reflexivity(f) \\ &+ 0.125 \times specificity(f) \\ &+ 0.125 \times consistency(f) \\ &+ 0.5 \times \{0, 100\} - agreement(f, a_1) \end{aligned}$$

Where a_1 is the preference function shown in Figure 12.5. In other words, the fitness function has been augmented to evolve functions which strongly disagree with the preference function a_1. The resulting function, shown in Figure 12.7, is the inverse of the function shown in Figure 12.5. This was evolved from the same starting point as all the other functions shown in this section, simply with the augmentation of the agreement metric. Although the original source of the function is again very complex, shown in Figure 12.6, the underlying functionality is accurate. We believe this demonstrates considerable power and flexibility in the metrics.

The Future

The flexibility that this kind of preference generation would provide us in building a system that can generate its own reasoning for its decisions, even

```
public int compare(int i, int j) {
    if ((i < i)) {
        return 0;
        return 0;
    }
    if (((j + i) < j)) {
        i = i;
        return -1;
    }
    else {
        j = -491;
        return 1;
    }
    return 0;
}
```

Figure 12.1: In this preference function, if `i` is negative, it is preferred over `j`; the second conditional check is true if $i < 0$.

```
public int compare(int i, int j) {
    if ((i <= j)) {
        return -1;
        j = ((i * 335) % j);
    }
    else {
        return 1;
        j = j;
    }
    return 1;
    return -1;
}
```

Figure 12.2: This preference function orders numbers from largest to smallest. The first conditional returns a reverse ordering (-1) if the first argument is smaller than the second. Note the copious amount of unreachable code. This constitutes a compile-time warning in C#, which is suppressed here.

```
public int compare(char i, char j) {
    if ((((int)(j)) <= ((int)(i)))) {
        return 1;
        return -1;
    }
    else {
        i = ((char)(((((int)(i)) -
        ((int)(i))) * (((int)(j)) -
        ((int)(j))))));
        return -1;
    }
    return 0;
    return 0;
}
```

Figure 12.3: Reverse lexicographic ordering on characters. The first conditional block is entered if the second argument, j, has a smaller ASCII code than the first argument, i. This returns a correct ordering (1). Otherwise, a reverse ordering is returned (-1). As with Figure 12.2, there is much unreachable code here. Also note that explicit casts to `int` types has caused a lot of excess bracketing.

```
public class Item{
    public bool droppable;
    public string name;
    public int damage;
    public int speed;
}
```

Figure 12.4: A dummy class specification used for generating preference functions. `speed` cannot have a negative value, but `damage` can.

```
public int compare(Item i, Item j){
  i.name = j.name;
  if ((j.speed > i.speed)){
    i = j;
    return 1;
  }
  else{
    return -1;
    j.name = j.name;
  }
  i.cost = (i.speed / i.speed);
}
```

Figure 12.5: An ordering on Item objects based on their `speed` variable.

```
public int compare(Item i, Item j){
  if ((i.speed == j.speed)
            == (i.speed >= j.speed))
    j = j;
    return -1;
  }
  if ((i.droppable != j.droppable)
            != (i.droppable || i.droppable)){
      return 1;
      j.name = j.name;
  }
  else {
      return 1;
      j.speed = ((i.speed * i.speed)
          - (i.speed + j.speed));
  }
  if (((i.speed * j.speed) > j.speed)){
      return -1;
      return 1;
  }
  else {
      return -1;
      j = i;
  }
}
```

Figure 12.6: An ordering on Item objects based on their `speed` variable, directly opposite to the one shown in Figure 12.5. A compacted, human-translated version of this function is shown in Figure 12.7

if the reasoning is ultimately arbitrary, is considerable. It not only allows *ANGELINA* to make decisions about objects it has never seen before (for example, if *ANGELINA* invents a new kind of game object, we cannot offer heuristics to evaluate it since we don't know anything about the object *a priori*). It also allows *ANGELINA* to reason about its decisions and hopefully will lead to an increased perception that the software is acting creatively and independent of us as its designers.

There are two important avenues along which to extend this work: first, to generate more complex preference functions that express richer opinions, using method calls to other parts of the codebase, and intermediate value calculations to allow for more interesting branching within a preference function. We acknowledge this is a considerable task, and code generation is not simple – however even small advances in this area could lead to new and interesting applications.

The second avenue is researching methods for the system to verbally communicate to a person what the plain-English effect of a preference function is. This is relatively important particularly for commentary generation, but also potentially for communicating this information in-game (consider a generated preference that governs how an NPC reacts to players – the game should be able to explain to the player what this means so that they can react to it). We believe that this is extraordinarily difficult. Not only does this imply the system is able to explain the functional performance of the code line-by-line, which is a complex task on its own, but it also means the system may need to relate the meaning of the code to the surface-level theme of the game. That means it should not only be able to know that a variable is important to a preference function, but it needs to understand that this variable represents, say, a player's knowledge of magic, or their public approval rating, or their current high score.

Providing a system with governance over its own subjectivity is a major step forwards for the autonomy and creativity of a system like *ANGELINA*. It opens up the potential for new kinds of framing information [26] that justify decisions, it allows the system to play with its own opinions and generate contrary examples that go against a particular preference function (as we saw earlier by leveraging the agreement metric), and it increases the perception that this software is acting alone or at the very least is the majority stakeholder in the creative artefacts it produces.

12.3 Understanding Culture and Meaning

As we saw in chapter 4, much work in procedural content generation focuses on generating the parts of games that are represented by abstract data – level designs, rulesets, layouts for parts of the game. This kind of content can be transferred between games, regardless of the game's theme, setting, meaning or relation to the real world. Since $ANGELINA_3$, we have been interested in bringing an understanding of real-world concepts and cultural ideas into the automated game design process, and letting *ANGELINA* work such things into its games – this encompasses a large amount of *ANGELINA*'s functionality, from reading the Guardian newspaper to inspire newsgames in $ANGELINA_3$ through to using corpora of proverbs and sayings to create pun titles for $ANGELINA_5$'s games.

Understanding and being able to use knowledge about how the world works is important to an automated game design system, if it is to design games that reference situations, people and ideas that the player can relate to. Such understanding provides an opportunity not only to demonstrate intelligence and creativity (by creating artistic work that appeals to human experiences), but to have real impact on both individuals and society at large, by making statements and observations about the world at large and challenging the player's views and ideas about how the world is.

While we made some progress in this area through the many iterations of *ANGELINA*, it was largely constrained to using real-world knowledge on a level that was completely detached from the rest of the game. Visual redesigns in $ANGELINA_3$'s newsgames do not affect the mechanics of the game to convey a political or social message; the game jam themes inputted to $ANGELINA_5$ do not filter through to shaping the objectives of the game on a functional level. In [4], Anna Anthropy describes games as '*experiences shaped by rules*'. A central hypothesis to the work we intend to do in the future is that this definition of rules compels us to find deeper ways of connecting real-world knowledge to games, embedding this knowledge in its rules, its systems and its design, to produce experiences that are interesting and perceived as creative works.

12.3.1 A Rogue Dream

A Rogue Dream is a prototype game designed to demonstrate the potential for extending *ANGELINA*'s use of real-world concepts. The game begins by asking the player for a noun, completing the phrase '*Last night, I dreamed I was a...*'. The noun is used to theme the game in a very simple sense, primarily surface-level visual theming, but reflecting an understanding of the given noun without the use of hand-designed databases within A Rogue Dream.

Figure 12.8 shows a screenshot from the game, with the input word 'cat'. The player character is represented by a cat in the center of the screen. The water droplets are enemies, and hurt the player if they come into contact with them. The grey objects are long grass, which heal the player if they pick them up. At the bottom of the screen, the game indicates that the player has a special ability – to *throw*, attacking an enemy at range.

These abilities along with non-player characters (NPCs) were generated by the game in response to the input word by using a technique called *Google milking*. The phrase was coined by Veale [130] to describe a process of sending incomplete questions to Google as search terms and then scraping the autocomplete suggestions from Google to find what people commonly type to complete the question. Autocompletion suggestions for the incomplete question '*why do cats always*' are shown in Figure 12.9. Veale noticed questions phrased in this way are actually valuable sources of information, because they report beliefs held by the person asking the question. The question 'why do cats always land on their feet' is asked by someone who believes that this is true, or has observed this happening directly, but this information is not explicitly written down on websites because the information is too obvious to arise in everyday discussion.

We adjust the technique of Google milking to extract more targeted information to generate game content with. Having fixed the player to be the input noun, a cat for example, we then mine Google using specific questions to extract information that will work to generate certain kinds of content. For example, *Why do cats hate...* is milked in order to obtain nouns that might function as enemies in the game (in this case, *Why do cats hate water*, leading to water droplets being used as enemies). *Why do cats eat...* or *Why do cats like...* are used to obtain nouns that would function as sources of

```
public int compare(Item i, Item j){
  if (i.speed <= j.speed) {
    return -1;
  }
  else{
    return 1;
  }
}
```

Figure 12.7: A retranslation of the generated code shown in Figure 12.6, to more clearly show the inverse relationship with the function in Figure 12.5. This is a direct functional translation of the code in Figure 12.6 with unreachable or nonfunctional code removed for readability.

Figure 12.8: A screenshot from *A Rogue Dream*, given the input word 'cat'.

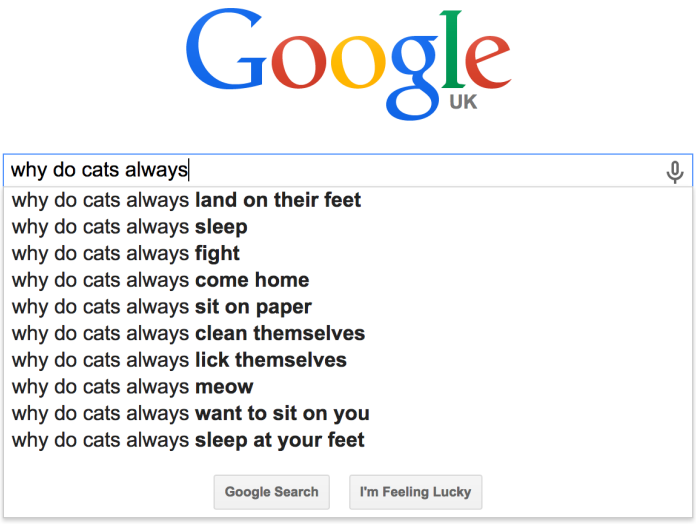


Figure 12.9: Google autocompletions for a partial search term question.

health points or score (in this case, *Why do cats eat grass*, leading to grass as a health collectible in the game – of course, in reality cats eat grass for a very different reason, but our approach only connects the verb *eat* to the game mechanic of health gain).

The images are generated for A Rogue Dream using *Spritely*, a tool we wrote in Java that searches several online image corpora – Google Images, Open Clipart and Wikimedia Commons. Spritely augments its image searches, adding on words like 'cartoon' and 'silhouette' to the original search terms, in order to obtain simple images which are easily scaled down to the size of a sprite. It downsamples the image, optionally applying colour palettes (which A Rogue Dream does not use), and then outputs a low-resolution pixel art sprite. We open sourced Spritely as a tool for game developers[1].

Finally, A Rogue Dream will attempt to generate a special ability for the player, based on several keywords being detected in Google milking searches. These are hard-coded to relate to specific in-game mechanics. For example, if a synonym of the word *throw* or *shoot* appears in a Google milking search, the 'ranged attack' mechanic may be selected, which allows the player to damage an enemy without being adjacent to it in the game world. In the

[1]https://github.com/gamesbyangelina/spritely

Figure 12.10: A screenshot from *A Rogue Dream*, given the input word 'musician'.

cat game, Google milking returns the autocompletion *Why do cats always throw up*. This erroneously is identified by the system as indicating that cats can throw certain kinds of objects, and thus the 'ranged attack' mechanic is selected with the name 'throw'.

12.3.2 Illustrative Examples

A Rogue Dream's reliance on volatile, noisy data means that it does not always find results, and even when it does, they do not always make sense – as we can see from the earlier example of a cat throwing up. To give an indication of its effectiveness, we performed an experimental run of the system on 30 words, picked from three Top Ten lists of animals, jobs and countries[2]. 60% of the words resulted in a full skinning of the game (i.e. the player, enemy and collectible item all succeeded in finding images), while 96% of the words resulted in all but one game element being properly skinned. In terms of quality, we performed a curation coefficient analysis as described in [29] – we measured what proportion of the results we would be willing to show to someone as acceptable output of the system. We considered 66% of the results to be good and worthy of showing others. The remaining third were let down either by bugs in the system, misinterpretation of terms through Spritely, or through bizarre or unusual output from Google.

[2]http://www.thetoptens.com/lists/

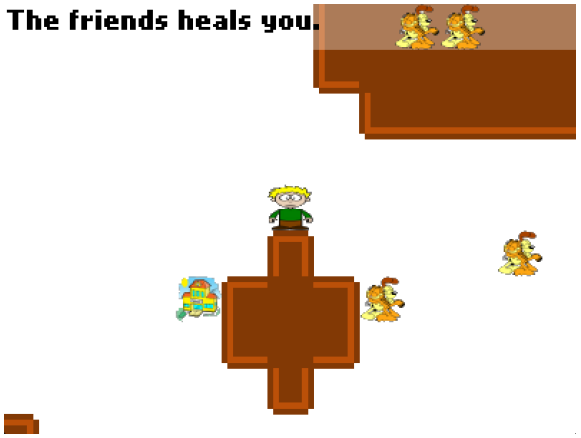


Figure 12.11: A screenshot from *A Rogue Dream*, given the input word 'kid'.

It should be noted that the three categories we chose – jobs, animals and nations – are all things to which sentience can be ascribed, or are spoken in terms of being sentient. This affects the nature of Google results significantly. A Rogue Dream certainly doesn't work on all words, or even all nouns. Nevertheless we consider it an interesting starting point for further expansion.

Below we give two further examples of full games, in addition to the cat example already given:

- **Kid** - The player controls a small child. They must avoid schools, which damage them, and collect friends, which heal them. Figure 12.11 shows a screenshot from this game.
- **Musician** - The player controls a man blowing a trumpet. They must avoid Kenny G albums, which damage them, and collect long hair. Figure 12.10 shows a screenshot from this game.

12.3.3 The Future

While A Rogue Dream is only a simple prototype, we believe that relating real-world concepts to the vocabulary of videogames is set to become the defining problem in automated game design. By focusing primarily on content which has no relation to real world content, modern approaches to procedural content generation have avoided encountering this problem up until now. But automating the entire process of game design forces us to

address it, and raises questions of what new technologies, tools and game designs might be possible if we have software which can generate content with an awareness of its real-world meaning.

When a player watches Mario bump into a mushroom and the subsequent chain of events that occurs – the mushroom disappears, Mario grows in size – they understand what has happened without anyone telling them explicitly. Anna Anthropy has written, at length, about how *Super Mario Bros.*' first level explains many of the game's mechanics without writing anything on the screen [3], and much of why it does this relies on both the game designer and the game player having a shared understanding of how the real world works and how to communicate that through shapes on a screen. Reaching the same level of shared understanding with a piece of software will be a serious step forward both for computational creativity and for automated game design.

12.4 More Directions For Future Work

12.4.1 Code Generation For Mechanic Invention

In chapter 8, we described *ANGELINA*$_4$ and showed how the system was capable of inventing new game mechanics, designing levels which exploited those mechanics, and even discovering emergent properties of its own code base that surprised us as the creators of the system. This shows the huge potential for code generation to have an impact on the future of game design, and also the potential for it to transform *ANGELINA* into a system that can influence the creative domain it is situated in, by making discoveries and influencing creative thinkers in its domain of game design.

Currently, *ANGELINA*$_4$'s code generation is limited to the modification of variables and does not generate larger code blocks or attempt to invent new game objects. There are many open areas of inquiry that we hope to look at in the future, including the generation of new objects with usefulness in solving game tasks, the development of more complex code segments that affect the game in more fundamental ways, and building a system which can identify places within a game's code to insert newly generated code. This last point is particularly complex – in *ANGELINA*$_4$ we assume that any generated code will be placed in a particular part of the player object's class file, activated whenever a button is pressed. In practice, game

development often necessitates that code is spread across many class files, in many different areas, in order to ensure data visibility, encapsulation or simply that the ordering of execution doesn't cause bugs. These are all complex concepts and will require many challenges to be overcome in program analysis and general game playing [54].

We have begun developing extensions to $ANGELINA_5$ in the Unity game engine that emulate $ANGELINA_4$'s ability to generate and test code. Unlike $ANGELINA_4$, this new implementation can generate short code segments rather than simple variable adjustments. Unity's engine also means that playouts can be viewed as they run, giving a better idea of what the generated code is doing to the game. However, the tradeoff is that the system is much slower as a result. Code generation, compilation and execution are all slower in C# under Unity than in Java as with $ANGELINA_4$. Unity is also a more complex use-case because the API is more general and less geared towards particular kinds of games, as was the case with the Flixel engine we used for $ANGELINA_4$. We believe this may make discovering simple game mechanics harder, as more code is required to cause similar effects to Flixel. Overall, there are many complications to pursuing this line of research, however we deem it to be valuable enough to warrant continued efforts.

12.4.2 Developer Commentary And Devlogging

We believe that the focus on commentary and framing throughout the growth of *ANGELINA* served the system well and increased the perception of creativity among observers. We also think it is useful in increasing the value of the artefacts produced by the system, and increasing interest in the games from the people playing them. We intend to develop and extend this emphasis on framing throughout the development process, mimicking the act of 'devlogging' that many game developers do through social media and blogging. The act of chronicling creative decisions *while the artefact is still being created* is not common in Computational Creativity, but in game development it increases the value of the resulting artefact, and offers insight into the creativity of the person developing the game in question.

In particular, we hope to implement two behaviours: adding developer commentary to finished games, and blogging about the ongoing progress of a game's design. Developer commentary is the act of embedding information

about the game's design into the game itself, normally as audio logs that can be played over the top of the normal game audio. Many modern games use this as an opportunity to shed light on certain aspects of a game, cut content, or creative decisions. Games made by small teams have used the technique to great effect, such as Gunpoint [37], but equally very large teams have also used developer commentary. For instance, Valve Software's Portal 2 [115] included commentary from many members of the development team, including programmers, designers, artists and sound engineers. Because developer commentary is often targeted to particular areas of the game or certain actions the player performs, it can provide much more detailed and targeted insight than a general commentary is able to. This should increase the information conveyed to the player, and hopefully raise their estimation of the game and their perception of the software that created it.

Blogging about a game's development allows for a different kind of commentary on the creative process. It allows players and potential players to see how a game is being developed actively, and to gain insight on decisions made by the software not retrospectively but while those decisions are still influencing the direction of the game. This might be an opportunity to demonstrate the intelligence and skill of the software more directly, and it also mitigates the concern that the software may be lying to the player in its commentaries – something which has come up in discussions with players of *ANGELINA*'s games in the past. For example, Figure 12.12 shows a comment from an online discussion about *ANGELINA*'s games, which states that the commenter does not believe *ANGELINA* when it justifies its choice of music or other design decisions, even when expicitly justified in commentaries and framing information.

Blogging also offers a chance for *ANGELINA* to do something it has not done to this point, namely generating interest in a game that it hasn't released yet. As the development cycles become longer for *ANGELINA*'s games, and the system potentially involves third parties such as musicians and artists in the generation of its content, this opens up the opportunity for *ANGELINA* to generate excitement about something it has produced prior to its release. This is a new kind of interaction for computationally creative software that will be interesting to explore.

```
fHueColour = iFloatRand(0.0f,360.0f);
Texture_Wall = iIntRand(0,10);
Music = iIntRand(0,5);
GameMode = iIntRand(0,10);

Ai, or, just basic random number generation?
```

Figure 12.12: A comment on an article about *ANGELINA* on Slashdot, a popular discussion site on the Internet. The commenter is implying that *ANGELINA*'s decisions are made using single random numbers rather than any intelligent process, despite the commentaries explaining otherwise.

12.4.3 Third-Party Asset Development

In developing *A Puzzling Present*, we worked with an artist to create promotional material such as store art for the game. This was to provide a professional front to the game so as to increase the pool of users and gain a larger volume of data. Contracting artists and musicians temporarily is very common in game development, and many solo game developers will work with contractors to complete parts of a game design they are not as skilled in. Generating art and music has been a persistent problem for *ANGELINA* and we believe this will continue to be a theme in automated game design. One possible solution might be to take the same approach that people do when designing games, and contract out these problems to other people, or other software such as The Painting Fool [21].

For most research problems, such a solution might be considered avoiding the problem, by simply conceding that the software can and should seek out the assistance of a skilled person instead of trying to find techniques to solve the problem itself. However, from a Computational Creativity standpoint, we can view this approach in a different light. Communicating with other creative people and engaging in the same creative problem is an interesting task for a piece of software to undertake, particularly when the software, not the people, is the driving creative force. To our knowledge, all research in mixed-initiative creativity, which we discussed in chapter 4, assumes that a person is the primary motivating force behind the creative act. In our case, a piece of software takes on that role.

Our intention would be to give *ANGELINA* a budget and connect it with pre-selected individuals who are happy to fulfil requests sent to them by *ANGELINA* via email. *ANGELINA* will be able to identify simple art and music assets it requires, send a specification to an appropriate contractor, and also respond to simple questions. Dialogue between the two parties is important in establishing this as a creative activity – *ANGELINA* could offer fixed questions it can respond to or topics it can talk more about, in the style of interactive fiction, to clarify elements of the specification. The contractor then produces the piece, and sends it to *ANGELINA* via email. The system could then analyse the piece to assess how much it meets its own expectations, perhaps asking for small modifications if it considers changes necessary. Finally, the content is placed into a game.

This dialogue between creative actors provides a platform for researching new interactions in a creative process, and putting more leadership responsibilities onto *ANGELINA*, as well as investigating how people react to working with a piece of software in a scenario in which they are perhaps more equal in creative input. It also has the potential to feed forward into framing and commentary, by providing *ANGELINA* with more opportunities to discuss the decisions it made and the ways game content came about. While definitely experimental, we are interested in pursuing this as a new direction in the philosophy of Computational Creativity.

12.5 Conclusions

In this section, we brought together some of the open threads of investigation that this thesis has presented, and looked at them in the context of ongoing future work. In many cases, we were able to highlight prototypes that point to promising new areas of expansion for *ANGELINA* and related research, such as the prototype roguelike game A Rogue Dream which trials new flexible methods for obtaining real-world knowledge that could feed into game designs at a systemic and structural level.

We also expanded the emerging theme of code generation present in *ANGELINA*$_4$, first by proposing it be used to augment *ANGELINA*'s creative ability rather than the games themselves by generating code that can make subjective judgements. This plays into the common Computational Creativity narrative of 'handing over responsibility' to the software by slowly

removing the influence of its designers from the creative process. Alongside this, we also proposed the extension of the mechanic invention work started with $ANGELINA_4$, extending it to make it a major component of future versions of *ANGELINA*, alongside the evolutionary game design aspects.

We also highlighted a number of areas of future work that we intend to explore but have not yet had a chance to develop very far. Extending existing areas of *ANGELINA* such as its framing and commentary, as well as offering the possibility for *ANGELINA* to co-ordinate creative activity with other people, both offer interesting areas of development that make *ANGELINA* more valuable as a game designer, but also further the field of Computational Creativity at the same time. Altogether, we hope these future directions will lead to *ANGELINA* becoming more dependent, and viewed in a more positive and respectful light, as it moves towards becoming a truly independent game designer.

13 Conclusions

Computational Creativity is a flourishing field of research, and the range of techniques used and media it covers is expanding rapidly. Prior to the work described in this thesis, videogames had received relatively little attention in this field, despite being (in our opinion) the most important cultural medium of the 21^{st} century. As we have seen throughout the presentation of *ANGELINA*, however, videogames are the perfect medium for Computational Creativity to investigate: highly complex, requiring many smaller creative tasks, offering the potential to investigate creativity in the design of *systems* as well as static pieces.

In this thesis, we presented our work building and developing *ANGELINA*, a system which autonomously designs videogames. The system has been developed across many iterations, each one focusing on a different aspect of game design, or a new research question uncovered by previous iterations. Over five distinct versions of *ANGELINA*, we developed games in many different languages, engines, platforms and genres. *ANGELINA* has been responsible for designing levels, enemy layouts, rulesets, game mechanics, visual themes, soundscapes and framing its own games in a cultural context. Perhaps most importantly, we showed that this work can have a real impact on the culture of games, by interacting directly with game designers, game players, and game critics.

In this chapter we review some of the contributions we opened the thesis with, and reflect on the aims of the project.

13.1 Reviewing Our Contributions

In Chapter 1, we presented several contributions which we stated would be covered during the thesis. Here we review these with reference back to the work presented in the chapters preceding this.

- Our primary practical contribution is the development of *ANGELINA* itself. Through chapters 5, 6, 7, 8 and 9, we charted the development of the system from developing simple arcade games to fully 3D experiences. The chapters included examples of games made by each system, showing the variety of game types we have tackled throughout. This has laid a broad foundation for future work in automated game design by offering baseline examples for future systems to be compared against.

- We showed that co-operative co-evolution is a sufficient technique for automated game design through the presentation of the simplest iterations of *ANGELINA* in chapters 5 and 6. However, through the course of our work, we believe we have also made a case that co-operative co-evolution can be enhanced by phases of analysis and preparation before the main design activities take place. We showed this first in chapter 7 where $ANGELINA_3$ downloaded newspaper articles and broke them down to find further assets to use in its games. The strongest case we made was in the last iteration of *ANGELINA* in chapter 9, where the system's ability to enter a game jam relied on the fact that $ANGELINA_5$ was capable of taking a single word or phrase as input and expanding that into the basis of an entire game design. Preliminary design phases in conjunction with a core CCE design system appear to provide a good basis for automated game design.

- In chapter 8, we introduced a novel technique for generating game mechanics through the direct inspection, modification and execution of code. This allowed $ANGELINA_4$ to generate puzzle platformer levels with unique game mechanics without the need for intermediate abstractions of the game world. This innovative approach led to surprising results, emergence in the designed mechanics, and has opened up new avenues of research for us in the future. We believe it may lead to a new phase of procedural content generation, as well as having a major influence on the state of the art in Computational Creativity research [33].

- In chapter 10, we went into detail regarding studies we had mentioned in earlier chapters describing the various versions of *ANGELINA*.

These studies covered *ANGELINA*'s interactions with a range of stakeholder groups, including game developers, not just in the context of evaluating the output of our research, but also in examining the interactions between artists working in a medium and a piece of computationally creative software seeking acceptance in the same medium. This is rare in Computational Creativity, and to our knowledge no such interactions have occurred in the field of videogame design. In the case of Ludum Dare, these interactions with a creative community represented the first time that a piece of software had entered a game design contest primarily aimed at people. We believe that the work we have presented here will provide a basis for future studies for other automated game design research, as well as our own.

13.2 Shifting Aims

As might be expected, this project changed considerably as we developed *ANGELINA* through its many iterations. We began with a desire to investigate co-operative co-evolution in automated game design, and as can be seen from the early chapters, we initially were mostly focused on the abstract data of videogames – level designs, rules, layouts. Most of these areas have been approached in procedural content generation projects before, albeit in individual aspects rather than as a concurrent system like *ANGELINA*, which attempts to generate all aspects at once. As our research progressed, we found ourselves faced with new and exciting questions, raised by exposure to a wider variety of game designers and critics, some of which we quote in background chapters 2 and 4.

Videogames are no longer distractions designed to attract more quarters for arcade machine owners. They are a medium of expression as broad as film or music[1], and like film and music, they are increasingly becoming accessible to everyone. Better development tools and the near-ubiquity of the Internet in the western world has enabled thousands of people to begin developing games who would not have considered the practice even a decade or so ago. The full title of Anna Anthropy's book [4], which we quoted in chapter 2, reads: *Rise of the Videogame Zinesters: How Freaks, Normals,*

[1]Of course, many videogames are still made to draw in quarters, and this is no bad thing. But they now represent only a portion of what games are today.

Amateurs, Artists, Dreamers, Dropouts, Queers, Housewives And People Like You Are Taking Back An Art Form. This explosion in accessibility, and the formation of a diverse array of subcultures in game development, gave us considerable pause for thought as we contemplated future directions for *ANGELINA*.

The overriding philosophy behind our development of *ANGELINA* was to keep adding responsibilities to the system, no matter how difficult those new responsibilities might sound to implement. In doing so, we uncovered new questions about computationally creative systems, about automated game design, and about building software that people would respect and treat as a creative individual. Influenced by our exposure to so many different game development scenes, we began to ask what it would mean for *ANGELINA* to express ideas, views about the world, or messages in the games that it creates. We pursued this through $ANGELINA_3$ and $ANGELINA_5$, while still pursuing technical innovations with our code generation work in $ANGELINA_4$. In the future, we hope to research ways to combine these two broad arms of automated game design, in order to build games in which the message of the game and the mechanics of the game's systems are intricately linked. These new ideas and challenges will be ultimately what separates automated game design from procedural content generation, allowing it to stand alone as a distinct subfield of research within videogames.

Developing *ANGELINA* has also brought us into the world of game development, as engineering an automated game designer typically necessitates that its designer also understands the game engines the system will be designed within. This has led us to be closer to the world of game development than we originally expected, and blurred the lines between independent developer and academic researcher. *A Puzzling Present*, the Android game developed during $ANGELINA_4$ as a subproject, was a collaboration between *ANGELINA* and us, and functioned as an entertaining game as well as demonstrating cutting-edge games research. We believe that this approach to research, while not suitable for every researcher, offers a new way to make games research relevant without the need to appeal to commercial potential. Researchers can do cutting-edge work that pushes the boundaries of both technology and art, and make that work meaningful by directly engaging with the game-playing public. We have observed one other group of people who work according to a similar process, over the years of research described

in this thesis: the same independent game developers Anthropy describes in [4]. We have fostered relationships with many developers through the course of this research, and their willingness to embrace new ideas and develop new technologies is encouraging and offers a promising future of collaboration between researchers and independent developers.

13.3 Automated Game Design

The development of *ANGELINA* involved overcoming many challenges. These challenges were a mix of technical, philosophical, engineering and cultural obstacles, and while we don't claim to have perfected the art of constructing automated game designers, we believe there are useful lessons to take away from this project for researchers looking to build automated game designers in the future.

More than any other area of games research, automated game design relies on people being able to play the results of the work done. While many research projects result in highly specialised games or software that can only run on a few computers, or that is too large and library-dependent to distribute, automated game design research cannot operate in this way. Priority must be given to producing games that are easily distributed and playable in as many ways as possible, to make large-scale surveys and feedback gathering as simple as possible. At each stage of *ANGELINA* we made sure to select engines and libraries that facilitated this: Monkey HTML5 for $ANGELINA_1$; Flixel and Flash for $ANGELINA_2$ and $ANGELINA_3$; Java and LibGDX for $ANGELINA_4$; and Unity for $ANGELINA_5$. Ultimately, all of these platforms were as good as each other.

Automated game design is less about the individual generative components, and more about what happens at the intersections of those components. That is to say, we consider questions such as 'What is the best way to design a level?' to be less important than 'How does a level design influence the design of the rest of the game?' Using existing generative techniques for the individual generative tasks, where possible, is preferable to attempting to reinvent those techniques or find small iterative improvements. Many of the individual generative approaches in *ANGELINA* are derived from existing methods for generating that content in isolation – a good example of this would be $ANGELINA_2$'s level design which uses tiles in a similar

manner to Spelunky, whose level design we covered in chapter 4. The focus can then shift from procedural generation tasks to the question of how these disparate tasks can be brought together, overlapped with one another, and solved as a composite problem by a game designer. An automated game designer is more than the sum of its generators – this is why we believe the field to be distinct, yet parallel, to procedural content generation research.

We also found that maintaining modularity in the system's design, and adopting an iterative design process, made it easier to rapidly develop *ANGELINA*, hand over creative responsibility, and extend it into new versions (*ANGELINA*$_3$ was built directly on top of *ANGELINA*$_2$'s codebase, and *ANGELINA*$_4$ used the basic platforming engine and some of its generative components). *ANGELINA*'s modularity is largely thanks to the structure of co-operative co-evolution, with each species being a distinct section of the software. New species were similarly easy to integrate into the system without affecting the existing species. The iterative design process – building new versions of *ANGELINA* regularly that refocus on new problem areas – helped us to incrementally hand over responsibilities to the system. In particular, we used Colton's approach of 'climbing the meta-mountain' [23], a term he uses to describe the process of examining a creative system, identifying areas in which a person is influencing the software's execution, and then adding new capabilities to the software to allow it to take on this task.

This process is extremely valuable when building automated game designers. Games are complex, multi-faceted artefacts with many different creative and technical tasks involved in their production. Climbing the meta-mountain allows small steps to be made, identifying the fringes of the software's current responsibility. Examining the involvement of people in the software also helps clear out bias in selecting areas of the system to develop. This helped us avoid iterating only on well-known areas of content generation, and encouraged us to focus on all aspects of game design equally. As a result, different versions of *ANGELINA* improve on areas from programming game mechanics to creating artistic directions for the game, and this variety helped uncover new and interesting problems to subsequently tackle.

13.4 Conclusion

In this chapter, we reviewed the contributions we laid out in chapter 1 and linked them back to the work that has been presented in the thesis. Our contributions range from practical demonstrations of ideas through to technical contributions of new approaches to content generation. We have also contributed methodological ideas about how best to structure automated game design systems, and how software can interact with groups of people in creative contexts.

We also reflected on the development of *ANGELINA*, and how although its core research thrust remained, we believe we have diversified *ANGELINA* as a system and opened up new research directions as time has gone on, largely inspired by changes to the medium of videogames that have happened in parallel with *ANGELINA*'s own development. We believe that we have laid out some foundational ideas for people who wish to pursue automated game design in the future, as a fusion of computational creativity and procedural content generation, but we also hope that we have demonstrated that this area is rich in problems to be solved, and many areas have not even had the most preliminary of initial work done in them. Automated game design is a new frontier for games research, one that does not have the usual plethora of obvious applications to games as an industry, but one that holds many promising new ideas, philosophical research directions, and potential contributions to videogames *as a medium*. We hope to have shown why this is a good thing over the course of this thesis.

The opening of this thesis includes two quotes, one from a videogame that lauds the optimism of scientists. The second, by Turing, comes at the end of the paper which proposed what is now called the Turing Test for artificial intelligence [97].

> "We can only see a short distance ahead, but we can see plenty there that needs to be done."

We find this quote to be no less true now, sixty years after it was first written, as it was then, as the horizons of artificial intelligence continue to expand and we find new and confusing challenges to face. We hope that this thesis has broadened the reader's vision of what is ahead, and similarly expanded the view of landmarks on the horizon to be worked towards.

Bibliography

[1] Nels Anderson. Why are so many indie darlings 2D platformers? `http://www.above49.ca/2010/07/why-are-so-many-indie-darlings-2d.html`, 2010.

[2] Anna Anthropy. Masocore games. http://auntiepixelante.com/?p=11.

[3] Anna Anthropy. Level design lesson: To the right, hold on tight, 2009.

[4] Anna Anthropy. *Rise Of The Videogame Zinesters: How How Freaks, Normals, Amateurs, Artists, Dreamers, Drop-outs, Queers, Housewives, and People Like You Are Taking Back an Art Form.* SEVEN STORIES PRESS, 2012.

[5] Jacob Aron. Ai designs its own video game. http://www.newscientist.com/article/mg21328554.900-ai-designs-its-own-video-game.html, 2012.

[6] D. Ashlock, C. Lee, and C. McGuinness. Search-based procedural generation of maze-like levels. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):260–273, Sept 2011.

[7] Tom Betts. *An Investigation of the Digital Sublime in Video Game Production.* PhD thesis, University of Huddersfield, 2014.

[8] Jonathan Blow. Braid, 2008.

[9] M.A. Boden. *The creative mind: myths & mechanisms.* Basic Books, 1991.

[10] Ian Bogost. Persuasive games: The proceduralist style. http://www.gamasutra.com/view/feature/132302/persuasive_games_the_.php.

[11] Team Bondi. L.A. Noire, 2011.

[12] Paul Cairns and Anna L. Cox. *Research Methods for Human-Computer Interaction*. Cambridge University Press, New York, NY, USA, 1st edition, 2008.

[13] Brendan Caldwell. Punk's not dead. `http://www.rockpapershotgun.com/tag/punks-not-dead/`, 2012.

[14] Luigi Cardamone, Daniele Loiacono, and Pier Luca Lanzi. Interactive evolution for the procedural generation of tracks in a high-end racing game. In *GECCO*, 2011.

[15] Alex Champandard. *Realistic Autonomous Navigation in Dynamic Environments*. PhD thesis, University of Edinburgh, 2002.

[16] John Charnley, Alison Pease, and Simon Colton. On the notion of framing in computational creativity. In *Proceedings of the 3rd International Conference on Computational Creativity*, 2012.

[17] Harold Cohen. Aaron. http://aaronshome.com/aaron/index.html, 2012.

[18] Simon Colton. *Automated Theory Formation in Pure Mathematics*. PhD thesis, University of Edinburgh, 2001.

[19] Simon Colton. Creativity versus the perception of creativity in computational systems. In *Proceedings of the AAAI Spring Symposium on Creative Systems*, 2008.

[20] Simon Colton. Seven catchy phrases for computational creativity research. In Margaret Boden, Mark D'Inverno, and Jon McCormack, editors, *Computational Creativity: An Interdisciplinary Approach*, number 09291 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[21] Simon Colton. The painting fool. http://www.thepaintingfool.com/, 2012.

[22] Simon Colton. The painting fool: Stories from building an automated painter. In Jon McCormack and Mark dInverno, editors, *Computers and Creativity*, pages 3–38. Springer Berlin Heidelberg, 2012.

[23] Simon Colton. The painting fool: Stories from building an automated painter. In Jon McCormack and Mark d'Inverno, editors, *Computers and Creativity*, chapter 1, pages 3–38. Springer, Berlin; Heidelberg, 2012.

[24] Simon Colton, Alan Bundy, and Toby Walsh. Automatic invention of integer sequences. In Henry A. Kautz and Bruce W. Porter, editors, *AAAI*, pages 558–563. AAAI Press / The MIT Press, 2000.

[25] Simon Colton, John Charnley, and Alison Pease. Computational Creativity Theory: The FACE and IDEA models. In *Proceedings of the Second International Conference on Computational Creativity*, 2011.

[26] Simon Colton, Michael Cook, Rose Hepworth, and Alison Pease. On acid drops and teardrops: Observer issues in computational creativity. In *Proceedings of the 7th AISB Symposium on Computing and Philosophy*, 2014.

[27] Simon Colton, Jacob Goodwin, and Tony Veale. Full-FACE poetry generation. In *Proceedings of the Third International Conference on Computational Creativity*, 2012.

[28] Simon Colton and Dan Ventura. You can't know my mind: A festival of computational creativity. In *Proceedings of the Fifth International Conference on Computational Creativity*, 2014.

[29] Simon Colton and Geraint A. Wiggins. Computational creativity: The final frontier? In *Proceedings of the European Conference on AI*, 2012.

[30] Michael Cook. Evomaze - a simple evolutionary system. https://github.com/gamesbyangelina/EvoMaze.

[31] Michael Cook and Simon Colton. Automated collage generation – with more intent. In *Proceedings of the Second International Conference on Computational Creativity*, 2011.

[32] Michael Cook and Simon Colton. From mechanics to meaning and back again: Exploring techniques for the contextualisation of code. In *Proceedings of the AIIDE Workshop on Artificial Intelligence and Game Aesthetics*, 2013.

[33] Michael Cook, Simon Colton, and Jeremy Gow. Nobody's a critic: On the evaluation of creative code generators. In *Proceedings of the Fourth International Conference on Computational Creativity*, 2013.

[34] Michael Cook, Simon Colton, and Jeremy Gow. The ANGELINA videogame design system, part i. In *Computational Intelligence and AI in Games, IEEE Transactions on*, 2015 (Under Review).

[35] Richard Dawkins. *The Blind Watchmaker*. Norton & Company, 1986.

[36] Frontier Developments. Elite, 1980.

[37] Suspicious Developments. Gunpoint, 2013.

[38] Christian Donlan. Plastic soul: One man's quest to build an ai that can create games. http://www.eurogamer.net/articles/2013-04-02-plastic-soul-one-mans-quest-to-build-an-ai-that-can-create-games, 2013.

[39] Christian Donlan. Can an AI win a game jam? http://www.eurogamer.net/articles/2014-01-14-can-an-ai-win-a-game-jam, 2014.

[40] Arne Eigenfeldt, Philippe Pasquier, and Adam Burnett. Evaluating musical metacreation. In Mary L. Maher, Kristian Hammond, Alison Pease, Rafael Pérez, Dan Ventura, and Geraint Wiggins, editors, *Proceedings of the Third International Conference on Computational Creativity*, page 140–144, Dublin, Ireland, may 2012.

[41] Farbs. ROM, CHECK, FAIL, 2008.

[42] Tom Francis. Gunpoint recoups development costs in 64 seconds. http://www.pentadact.com/2013-06-18-gunpoint-recoups-development-costs-in-64-seconds/, 2013.

[43] Bay 12 Games. Dwarf fortress, 2006.

[44] CCP Games. Eve online, 2003.

[45] Evolutionary Games. Galactic arms race, 2010.

[46] Gaslamp Games. Dungeons of dremor, 2011.

[47] Mossmouth Games. Spelunky, 2009.

[48] Subset Games. Ftl: Faster Than Light, 2012.

[49] Supergiant Games. Transistor, 2014.

[50] Telltale Games. The Walking Dead, 2012.

[51] N. Garcia-Pedrajas, C. Hervas-Martinez, and J. Munoz-Perez. Covnet: A cooperative coevolutionary model for evolving artificial neural networks. *Transactions on Neural Networks*, 14(3):575–596, 2003.

[52] Eugene Garver. Rhetoric and essentially contested arguments, 1978.

[53] Sam Geen. 1 + 10 more, 2014.

[54] Michael Genesereth and Nathaniel Love. General game playing: Overview of the aaai competition. *AI Magazine*, 26:62–72, 2005.

[55] Jeremy Gow, Simon Colton, Paul A. Cairns, and Paul Miller. Mining rules from player experience and activity data. In Mark Riedl and Gita Sukthankar, editors, *AIIDE*. The AAAI Press, 2012.

[56] Jay Griffin. The duellists, 2014.

[57] Gary Gygax and Dave Arneson. Dungeons and dragons, 1974.

[58] Erin J Hastings, Ratan K Guha, and Kenneth O Stanley. Evolving content in the galactic arms race video game. In *IEEE Symposium on Computational Intelligence and Games*, 2009.

[59] Claire Hosking. Opinion: Stop dwelling on graphics and embrace procedural generation. http://www.polygon.com/2013/12/10/5192058/opinion-stop-dwelling-on-graphics-and-embrace-procedural-generation, 2013.

[60] Hiroyuki Imabayashi. Sokoban, 1981.

[61] Metroid World Map image. Nes maps. http://www.nesmaps.com.

[62] Atari Inc. Pong, 1972.

[63] IO Interactive. Hitman: Blood Money, 2006.

[64] Inc. Interactive Data Visualization. Speedtree. http://www.speedtree.com/, 2002.

[65] Colin Johnson. The creative computer as romantic hero? or, what kind of creative personae do computational creativity systems exemplify? In *Proceedings of the Third International Conference on Computational Creativity*, 2012.

[66] Soren Johnson. Analysis: Sid meier's key design lessons. http://www.gamasutra.com/view/news/23458/Analysis_Sid_Meiers_Key_Design_Lessons.php, 2009.

[67] Anna Jordanous. *Evaluating Computational Creativity:A Standardised Procedure for Evaluating CreativeSystems and its Application*. PhD thesis, University of Sussex, 2012.

[68] Makoto Kano, Gunpei Yokoi, Hiroji Kiyotake, and Yoshio Sakamoto. Metroid, 1986.

[69] Darius Kazemi. Spelunky generator lessons. http://tinysubversions.com/spelunkyGen/, 2013.

[70] Manuel Kerssemakers, Jeppe Tuxen, Julian Togelius, and Georgios N. Yannakakis. A procedural procedural level generator generator. In *IEEE Conference on Computational Intelligence and Games*, 2012.

[71] Ed Key and David Kanaga. Proteus, 2013.

[72] Adam Kilgariff. Bnc database and word frequency lists. http://www.kilgarriff.co.uk/bnc-readme.html.

[73] Konami. Frogger, 1981.

[74] Konami. Castlevania, 1986.

[75] Raph Koster and Will Wright. *A Theory of Fun for Game Design*. Paraglyph Press, 2004.

[76] Anna Krzeczkowska, Jad El-hage, Simon Colton, and Stephen Clark. Automated collage generation with intent.

[77] Nicholas Lambert, William Latham, and Frederic Fol Leymarie. The emergence and growth of evolutionary art: 1980–1993. In *ACM SIGGRAPH 2013 Art Gallery*, SIGGRAPH '13, pages 367–375. ACM, 2013.

[78] William Latham. Mutator event website. http://latham-mutator.com/category/events/.

[79] Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evol. Comput.*, 19(2):189–223, 2011.

[80] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. Sentient sketchbook: Computer-aided game level authoring. In *Proceedings of the ACM Conference on Foundations of Digital Games*, 2013.

[81] Hugo Liu and Push Singh. Conceptnet: A practical commonsense reasoning toolkit. *BT Technology Journal*, 22:211–226, 2004.

[82] LucasArts. The Secret of Monkey Island, 1990.

[83] Victor W. Marek. Stable models and an alternative logic programming paradigm. In *In The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

[84] Jane McGonigal. Gaming can make a better world (ted talk). http://www.ted.com/talks/jane_mcgonigal_gaming_can_make_a_better_world, 2010.

[85] Jane McGonigal. *Reality Is Broken: Why Games Make Us Better and How They Can Change the World*. Penguin Group , The, 2011.

[86] Michael McWhertor. About kotaku reviews, 2009.

[87] Microprose. Sid Meier's Civilization, 1991.

[88] Shigeru Miyamoto. Donkey kong, 1981.

[89] Richard Morris, Scott Burton, Paul Bodily, and Dan Ventura. Soup over bean of pure joy: Culinary ruminations of an artificial chef. In *Proceedings of the Third International Conference on Computational Creativity*, 2012.

[90] Namco. Pac-man, 1975.

[91] Kathryn Nave. Meet angelina, the game-designing ai who loves rupert murdoch. http://www.wired.co.uk/news/archive/2014-01/20/angelina-gaming-ai, 2014.

[92] Mark J. Nelson and Michael Mateas. Towards automated game design. In *Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence*, 2007.

[93] Finn Årup Nielsen. A new anew: Evaluation of a word list for sentiment analysis in microblogs. *Computing Research Repository*, 1103.2903, 2011.

[94] Jan Willem Nijman. The art of screenshake. https://www.youtube.com/watch?v=AJdEqssNZ-U.

[95] Nintendo. Warioware, 2003.

[96] Alexis Ong. Ten women reshaping modern tech. http://www.dazeddigital.com/artsandculture/article/18631/1/the-top-ten-women-taking-over-tech, 2014.

[97] Alison Pease and Simon Colton. On impact and evaluation in computational creativity: A discussion of the turing test and an alternative proposal. In *Proceedings of the AISB symposium on AI and Philosophy*, 2011.

[98] Ken Perlin. Improving noise. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH, 2002.

[99] Markus Persson and Jens Bergensten. Minecraft, 2011.

[100] Amanda Phillips. Dispatch from a computer science conference: My time at aiide14. http://gamertrouble.wordpress.com/2014/10/07/dispatch-from-a-computer-science-conference-my-time-at-aiide14/, 2014.

[101] Mitchell A. Potter and Kenneth A. De Jong. A cooperative coevolutionary approach to function optimization. In *Proceedings of the International Conference on Evolutionary Computation*, 1994.

[102] Graeme Ritchie. Some empirical criteria for attributing creativity to a computer program. *Minds and Machines*, 17(1), 2007.

[103] Jim Rossignol. Guest informant: Jim rossignol. http://www.warrenellis.com/?p=13469, 2011.

[104] Jim Rossignol and John Walker. The rock, paper, shotgun electronic wireless show, episode 37, 2011.

[105] Steve Russell. Spacewar!, 1962.

[106] Rob Saunders. Multi-agent simulations of social creativity. Talk given at the PROSECCO Autumn School on Computational Creativity, 2013.

[107] Jimmy Secretan, Nicholas Beato, David B D Ambrosio, Adelein Rodriguez, Adam Campbell, and Kenneth O Stanley. Picbreeder: evolving pictures collaboratively online. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1759–1768. ACM, 2008.

[108] Team Silent. Silent hill 2, 2001.

[109] Karl Sims. Evolving 3d morphology and behavior by competition. *Artificial Life*, 1(4):353–372, 1994.

[110] Adam M. Smith and Michael Mateas. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *IEEE Conference on Computational Intelligence and Games (CIG)*, 2010.

[111] Gillian Smith. The seven deadly sins of PCG research. `http://sokath.com/main/the-seven-deadly-sins-of-pcg-papers-questionable-claims-edition/`, 2013. expanded from a panel discussion at the Foundations of Digital Games conference 2013.

[112] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the FDG Workshop on Procedural Content Generation in Games*, 2010.

[113] Valve Software. Team fortress blog - history. http://www.teamfortress.com/history.php.

[114] Valve Software. Team fortress 2, 2007.

[115] Valve Software. Portal 2, 2012.

[116] Valve Software. Dota 2, 2013.

[117] Bethesda Softworks. The elder scrolls ii: Daggerfall, 1994.

[118] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2), 2002.

[119] Bethesda Game Studios. The Elder Scrolls IV: Oblivion, 2007.

[120] Bethesda Game Studios. The Elder Scrolls V: Skyrim, 2011.

[121] Georgia Tech. Gwappy bird. http://www.newgrounds.com/portal/view/647126, 2014.

[122] themushroomsound. Wormholes, 2014.

[123] Stephen Todd and William Latham. *Mutator: a subjective human interface for evolution of computer sculptures.* IBM United Kingdom Scientific Centre, 1991.

[124] Julian Togelius, Renzo De Nardi, and Simon M. Lucas. Towards automatic personalised content creation in racing games. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

[125] Julian Togelius and Jrgen Schmidhuber. An experiment in automatic game design. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*, pages 111–118, 2008.

[126] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Trans. Comput. Intellig. and AI in Games*, 2011.

[127] Michael Toy, Glenn Wichman, Ken Arnold, and Jon Lane. Rogue, 1980.

[128] Mike Treanor, Bryan Blackford, Michael Mateas, and Ian Bogost. Game-o-matic: Generating videogames that represent ideas. In *Proceedings of the Third Workshop on Procedural Content Generation in Games*, 2012.

[129] Alan M. Turing. Computing Machinery and Intelligence. *Mind*, 1950.

[130] Tony Veale. From conceptual "mash-ups" to "bad-ass" blends: A robust computational model of conceptual blending. In *Proceedings of the Third International Conference on Computational Creativity*, 2012.

[131] R. Paul Wiegand. *An Analysis of Cooperative Coevolutionary Algorithms*. PhD thesis, George Mason University, 1999.

[132] Castlevania Wikia. Simon's quest inventory. http://castlevania.wikia.com/wiki/Simon

[133] Super Metroid Wikia. List of items in super metroid. http://metroid.wikia.com/wiki/List_of_items_in_Super_Metroid.

[134] Mike Williams. 1reasontobe panel shows female devs still struggling for equality. http://www.gamesindustry.biz/articles/2013-03-28-1reasontobe-panel-shows-female-devs-still-struggling-for-equality, 2013.

[135] Derek Yu. The full spelunky on spelunky, 2012.

[136] Zhengyou Zhang. Microsoft kinect sensor and its effect. *IEEE MultiMedia*, 19(2):4–10, April 2012.