Chapter 6
Rules and mechanics (DRAFT)

Mark J. Nelson, Julian Togelius, Cameron Browne, and Michael Cook

So far in this book, we have seen a large number of methods for generating content for existing games. So, if you have a game already, you could now generate many things for it: maps, levels, terrain, vegetation, weapons, dungeons, racing tracks. But what if you don't already have a game, and want to generate the game itself? What would you generate, and how? At the heart of any game are its rules. This chapter will discuss representations for game rules of different kinds, along with methods to generate them, and evaluation functions and constraints that help us judge complete games rather than just isolated content artefacts.

Our main focus here will be on methods for generating interesting, fun, and/or balanced game rules. However, an important perspective that will permeate the chapter is that game rule encodings and evaluation functions can encode game design expertise and style, and thus help us understand game design. By formalising aspects of the game rules, we define a space of possible rules more precisely than could be done through writing about rules in qualitative terms; and by choosing which aspects of the rules to formalise, we define what aspects of the game are interesting to explore and introduce variation in. In this way, each game generator can be thought of an executable micro-theory of game design, though often a simplified, and sometimes even a caricatured one [21].

6.1 Encoding game rules

To generate game rules, we need some way of *representing* or *encoding* them in a machine-readable format that our systems can work with.[1] An ambitious starting point for a game encoding might be one that can encode game rules *in general*: an open-ended way to represent any possible game. The game generator would

[1] There are many other uses for machine-readable game rules, such as for use in game-playing AI competitions [**?**, **?**] and in game-design assistants targeted at human game designers [15, 7]. This chapter focuses on encodings for *generating* rules, but multi-use encodings are often desirable.

then work on games in this encoding, looking for variants or entirely new games in this space. But such a fully general encoding provides a quite unhelpful starting point. A completely general representation for games cannot say very much specific about games at all. Some kinds of games have turns, but some don't. Some games are primarily about graphics and movement, while others take place in an abstract mathematical space. The only fully general encoding of a computer game would be simply a general encoding for all software. Something like "C source code" would suffice, but it produces an extremely *sparse* search space. Although all computer games could in principle be represented in the C programming language, almost all things that can be represented in C's syntax are not in fact games, and indeed many of them are not even working programs, making a generator's job quite difficult.[2] Instead of having a generator search through the extremely sparse space of all computer programs to find interesting games, a more fruitful starting point is to pick an encoding where the space includes a more dense distribution of things that are games and meet some basic criteria of playability. That way, our generator can spend most of its time attempting to design interesting game variants. Furthermore, it's helpful for game encodings to start with a specific genre. Once we restrict focus to a particular genre, it's possible to abstract meaningful elements common to games in the genre, which the generator can take as given. For example, an encoding for turn-based board games can assume that the game's time advances in alternating discrete turns, that there are pieces on spaces arranged in some configuration, and that play is largely based on moving pieces around. This means the game generator does not have to invent the concept of a "turn", but instead can focus on finding interesting rules for turn-based board games. An encoding for a side-scrolling space shooter would be very different: here the encoding would include continuous time; entities such as terrain, enemies, physics, lives, and spawn points; and events such as shooting, object collision, and scrolling. Of course, the encoding cannot be *too* narrow: at the limit, an encoding that specifies exactly one game (or only a few) is not very interesting for a game-generation system. The most productive point on the spectrum between complete generality and complete specificity is one of the key tradeoffs in designing an encoding for game generators to use: smaller spaces typically are more dense in playable, interesting candidates, but larger spaces may allow for more interesting variation [18].[3]

In addition to being a more fruitful space for game generators to work in, genre-specific encodings also make it easier to produce *playable* games. Whereas a computer could generate purely abstract rule systems, making interesting games that are playable by humans requires connecting those abstract rules to concrete audiovi-

[2] This is not to say generating games encoded as raw programs would be *impossible*: genetic-programming techniques evolve programs encoded in fairly general representations [19], and applying genetic programming to videogame design could produce interesting results. But the techniques in this chapter focus on higher-level representations, which allow the generators to work on more familiar game-design elements rather than on low-level source code.

[3] Some interesting future work lies in modular encodings: instead of choosing a specific genre, a generator might pick and choose a generative space consisting of a combat system, 2d grid movement, an inventory system, etc. [14].

sual representations [13]. For example, the abstract notion of a “capture” in board games is often represented by physically removing a piece from the board. The idea of “hidden information” in card games is represented by how players hold their cards, and which cards on the table are face-up versus face down. Concepts such as “health” can be represented in any number of ways, ranging from numerical display of hitpoints or health percentage on the screen, to more indirect methods such as changing a character’s color, or even varying the music when a player’s health drops below a threshold. Matching generated rules to these concrete representations can be a challenging research problem in itself [16], but working with encodings of specific genres allows us to sidestep the issue, by having a standard concrete representation for the genre being considered.

Finally, using a genre-specific encoding provides a first step towards answering a key question: how do we evaluate what constitutes a good set of game rules? Rather than the extremely general question of what makes a good game, we can take ask what makes a good *two-player board game*, a good *real-time strategy game*, or a good *first-person shooter*. That lets us take advantage of existing genre-specific design knowledge, which is usually better-developed and more amenable to being formalized. Design of new board games may focus on properties such as balance, availability of multiple nontrivial strategies, etc. Criteria for designing a good sidescrolling shooter, meanwhile, may instead focus on the pace of the action, patterns of enemy waves, and the difficulty progression—very different kinds of criteria. When we generate the rules for games using encodings of these well-defined genres, we can use a wide variety of existing design knowledge to made our playability and quality judgments. This allows rule-generating PCG systems to start from the basis of being *domain experts* in a specific genre, to use Khaled *et al.*’s terms for PCG system roles [9].

The two sections that follow describe game-generator experiments that a number of researchers have undertaken in two domains that have seen the most study: board games, and 2d graphical-logic games.

6.2 Board games

Board games were the first domain in which systems were built to procedurally generate game rules. They have several features that make them a natural place to start. For one, there is a discrete, finite structure to the games that simplifies encoding; unlike computer games, which are defined by an often complex body of code, games like chess are defined by simple sets of rules. Secondly, there is already a culture of inventing board-game variants, so automatic invention of game variants can draw from existing investigations into *manual* generation of game variants, and the design books that have been written about those investigations [6, 2].

6.2.1 Symmetric, chess-like games

The earliest rule-generating system, predating the more recent resurgence in PCG research, was METAGAME [18], which generated “symmetric, chess-like games”. The *chess-like* part means that the games take place on a grid, and are structured around two players taking turns moving pieces according to certain rules; these pieces can also be removed from the board in certain circumstances. The *symmetric* part means that the two players start on opposite ends of the board with symmetric starting configurations, and all game rules are identical for each player, just flipped to the other side of the board. For example, if METAGAME invented a chess variant in which pawns could capture sideways, this would always be true for both the black and white player; the space of games METAGAME represents doesn’t include asymmetric games where players start with different pieces, or make moves according to differing rules.

The symmetric aspect of the game rules is enforced by construction: only one set of rules is encoded in the generator, and those rules are applied to both players, so any change to an encoded rule automatically changes the rules for both sides. The space of possible rules is encoded in a hierarchical *game grammar* that specifies options for the board layout, how pieces can move, how they can capture, win conditions, and so on. Specific games are generated by simply stochastically sampling from that grammar, and then imposing some checks for basic game playability. The generator also has a few parameter knobs available, allowing the user to tweak some aspects of what’s likely to be generated, such as the average complexity of movement rules.

Pell’s motivation for building METAGAME was not game generation itself, but testing AI systems on the problem of general game playing. By the early 1990s, there was a worry that computer chess competitions were causing researchers to produce systems *so* specifically engineered to play chess and only chess, that they might no longer be advancing artificial intelligence in general. Pell proposed that more fundamental advances in AI would be better served by forcing game-playing AI systems to play a wider space of games, where they wouldn’t know all the rules in advance, and couldn’t hard-code as many details of each specific game [17]. To actually set up such a competition, he needed a way to define a larger space of games, and a generator that could produce specific games from that space, to send to the competing systems. METAGAME was created to provide that more general space of test games, and as a result, also became the first PCG system for game rules.

6.2.2 Balanced board games

While METAGAME generated a fairly wide range of games, the end result was controllable only implicitly: games were not selected for specific properties, but chosen randomly from the game grammar.

One property that is frequently desired in symmetric games is game balance: there shouldn't be a large advantage for one side or the other, such that the outcome is too strongly determined by who starts with the white pieces versus the black pieces. METAGAME produces games that are *often* balanced by virtue of having symmetric rule-sets, which tend to produce balanced gameplay. But a symmetric rule-set does not automatically mean a game will be balanced: moving first can often be a large advantage, or it might even in some cases be a disadvantage. Hom and Marks [8] decided to address the goal of balance directly. They first took a much smaller space of chess variants, to allow the space to be more exhaustively searched. Then, they evaluated candidate games for balance by having computer players play against each other a number of times, and rejected games with simulated win-rates that deviated too far from 50/50.

This process ends up feeding the original motivating application of METAGAME back into the generation of game rules. METAGAME had been designed to test general game-playing agents, which were new at the time. Over the years, a number of research and commercial systems were developed, which could take an arbitrary game encoded in a description language, and attempt to play it. Hom and Marks took one such general game-playing system, Zillions of Games, and set it to play their generated games as a way of evaluating them.

The changes from METAGAME introduced here are fairly general ones which are seen in other PCG systems: the idea of an *evaluation function* to decide what constitutes a good example, and *simulation* as a way of specifying an evaluation function in a complex domain, where it's difficult to specify one directly. Here, simulation is done by the computer playing the game against itself, and the evaluation function is how close its win rate comes to being 50/50 from each side of the board.

6.2.3 Evolutionary game design

The obvious next step is to use this ability to simulate and evaluate general games to guide the automated search for new games. For example, the evaluation function can be used as a *fitness function* to direct the evolution of rule sets, to search for new combinations of mechanics that produce fit, interesting games. This section describes an experiment in evolutionary board game design called LUDI, which produced the first fully computer-invented games to be commercially published [3].

6.2.3.1 Representation

Games are described in the LUDI system as symbolic expressions in simple *ludemic* form (a *ludeme* is a unit of game information). For example, Tic-Tac-Toe is described as follows:

```
    (game Tic-Tac-Toe
    (players White Black)
    (board (tiling square i-nbors) (shape square) (size 3 3))
    (end (All win (in-a-row 3)))
)
```

This game is played by two players, White and Black, on a square 3×3 board including diagonals (*i-nbors*), and is won by the first player to form 3-in-a-row of their colour. By default, players take turns placing a piece of their colour on an empty board cell per turn.

The LUDI language is *functional* rather than *declarative* in nature, being composed of high-level rule concepts rather than low-level machine instructions or logic operations, as per the Stanford GDL. This makes the language less general as every rule must be predefined by the programmer, but has the advantages of simplicity and clarity; most readers should be able to recognise the game described above despite having no prior knowledge of the system. Further, it allows rule sets to be described and manipulated as high-level conceptual units, much as humans conceptualise games when playing and designing them.

6.2.3.2 Evaluation

The LUDI system evaluates a rule set by playing the game against itself over a number of self-play trials. A rule set is deemed to be “fit” in this context if it produces an a non-trivial and interesting contest for the players. The basic approach is similar to that used by Althöfer [1] and Hom and Marks [8], but in this case a much broader range of 57 aesthetic measurements are made, divided into:

- *Intrinsic* criteria based directly on the rule set.
- *Playability* criteria based on the outcomes of the self-play trials.
- *Quality* criteria based on trends in play play.

The intrinsic criteria measure the game at rest directly from its rule set. However, the true nature of a game does not emerge until the game is actually played, so it was not surprising that no intrinsic criteria ultimately proved useful when these criteria were correlated with human player rankings for a suite of test games.

It was found that the playability criteria, based on the game outcomes, provided a useful and robust estimate of the basic playability of a game. Four of these criteria proved particularly good at identifying unfit rule sets, constituting a *playability filter* that formed the first line of defense to quickly weed out games that:

- result in draws more often than not,
- are too unbalanced towards either player,
- have a serious first or second move advantage, or
- are too short or too long on average.

Games that pass the playability filter are then subject to a number of more subtle and time-intensive quality measurements, based on the *lead histories* of the simulated games. The lead history of a game is a record of the difference between the estimated strength of the board position of the eventual winner and the eventual loser at each turn. Such quality measurements are more subtle and less reliable than the playability measurements, but offer the potential to capture a richer snapshot of the player experience.

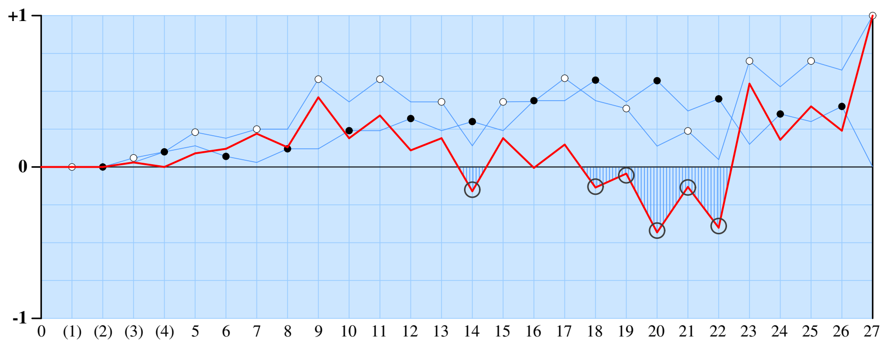

Fig. 6.1: Lead history showing drama in a game.

For example, Figure 6.1 shows the lead history of a game lasting 27 moves. The white and black dots show the players' estimated fortunes, respectively, while the bold line shows the difference between them at each move. This example demonstrates a dramatic game, in which the ultimate winner (White) spends several moves in a relatively negative (losing) position before recovering to win the game. Such *drama* is a key indicator of interesting play that human designers typically strive to achieve when designing board games.

6.2.3.3 Generation

Rule sets are evolved using a *genetic programming* (GP) approach, summarised in Figure 6.2. A population of games is maintained, ordered by fitness, then for each generation a pair of relatively fit parents are selected and mated using standard *crossover* and *mutation* operations to produce a child rule set. The symbolic expressions used to describe games constitute rule trees that are ideal for this purpose.

Each child rule set is checked for correctness according to the LUDI language, playability, performance and similarity to other rule sets in the population. Rule sets that pass these checks are given a unique name, officially making them a game, and are then measured for fitness and added to the population. The name for each game is also generated by the system, based on letter frequencies in a list of Tolkien-style names.

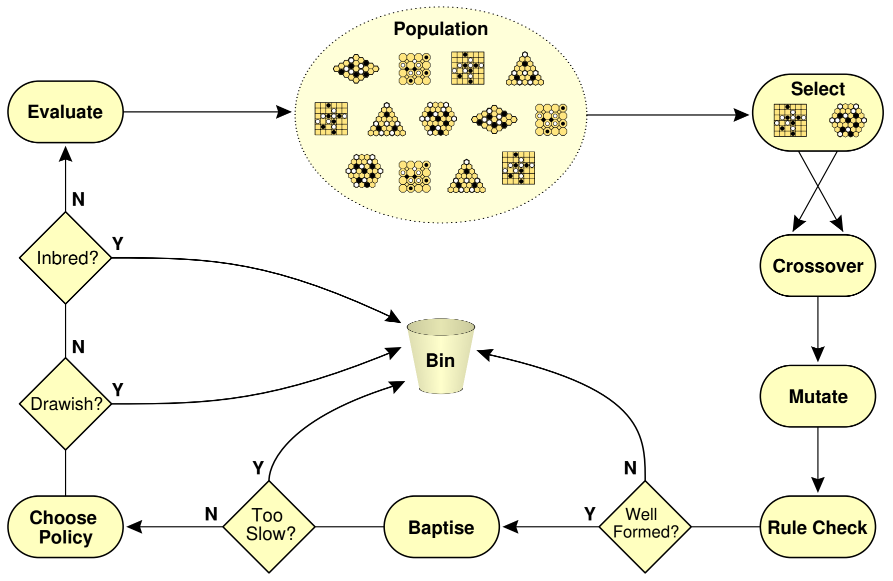

Fig. 6.2: Evolutionary game design process.

6.2.3.4 Evolved Games

LUDI evolved 1,389 new games over a week, of which 19 where deemed “playable” and two have proven to be of exceptional quality. The best of these, Yavalath, is described below:

```
(game Yavalath
   (players White Black)
   (board (tiling hex) (shape hex) (size 5))
   (end (All win (in-a-row 4)) (All lose (in-a-row 3)))
)
```

Yavalath is similar to Tic-Tac-Toe played on a hexagonal board, except that players win by making 4-in-a-row (or more) of their colour but lose by making 3-in-a-row beforehand. This additional condition may at first seem a redundant afterthought, but players soon discover that it allows some interesting tactical developments in play.

For example, Figure 6.3 shows a position in which White move **1** forces Black to lose with blocking move **2**. Such forcing moves allow players to dictate their opponent’s moves to some extent and set up clever forced sequences. This *emergence* of complex behaviour from such simple rules provides an “aha!” moment that players find quite compelling, and is exactly what is hoped for from an evolutionary search.

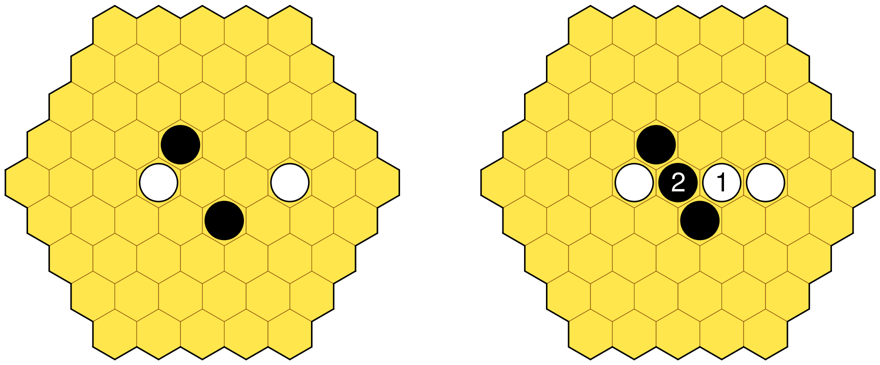

Fig. 6.3: White forces a win in Yavalath.

The other interesting game evolved by LUDI is called Ndengrod:

```
(game Ndengrod
   (players White Black)
   (board (tiling hex) (shape trapezium) (size 7 7))
   (pieces (Piece All (moves (move
      (pre (empty to)) (action (push)) (post (capture surround))
   ))))
   (end (All win (in-a-row 5)))
)
```

This is also an *n*-in-a-row game—this rule dominated the rule sets of evolved games—but in this case players capture enemy groups that are surrounded to have no freedom, as per Go. This rule set also demonstrates the emergence of interesting and unexpected behaviour, due to an inherent conflict between the “capture surround” and “5-in-a-row” rules, as shown in Figure 6.4.

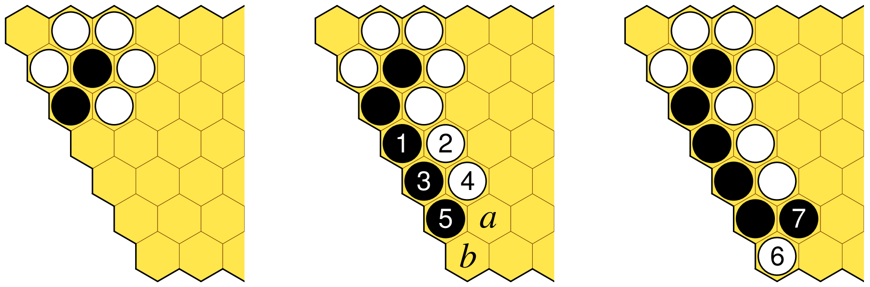

Fig. 6.4: Ladders don’t work as planned in Ndengrod.

White squeezes Black against the edge to force a ladder (left), which Black must extent each turn to keep their group alive (middle). However, once the ladder reaches four pieces long after move **5**, then White cannot continue the attack at point *a* but must instead block the line at point *b*, allowing White to escape with move **7**, and the game continues with White piece **6** now under threat.

6.2.3.5 Legacy

Yavalath and Ndengrod (renamed Pentalath) were the first fully computer-invented games to be commercially published. Yavalath was the first game released by Spanish publisher Nestorgames, and continues to be the flagship product in its catalogue of over 100 games.

Ndengrod is actually the better game of the two; it is deeper, involving a complex underlying friction between enclosure and connectivity, and is definitely more of a brain-burner. However, the more complex rules create a higher barrier to entry for beginners, hence it is destined to remain second choice. Conversely, the rules of Yavalath are intuitively obvious to any new player, and it has since been ranked in the top #100 abstract board games ever invented [4].[4]

The successful invention of board games by computer did not cause a backlash from players and designers as expected. The most common response from players is simply that they're surprised that a computer-designed game could be this simple and fun to play, while designers have so far dismissed this automated incursion into the very human art of game design as not much of a threat, as long as it produces such lightweight games. However, this attitude may change as PCG techniques—and their output—becomes increasingly sophisticated and challenges human experts in the field of design as well as play.

One near-miss produced by LUDI, called Lammothm, is worth mentioning to highlight a pitfall of the evolutionary approach. Lammothm is played as per Go (i.e. surround capture on a square grid) except that the aim is to connect opposite sides of the board with a chain of your pieces. Unfortunately, the evolved rule set contained the *i-nbors* attribute, meaning that pieces connect diagonally which all but ruins the game, but if this attribute is removed then the rule set suddenly becomes equivalent to that of Gonnect, one of the very best connection games [2]. LUDI was one mutation away from rediscovering a great game, but the very nature of the evolutionary process means that this mutation is not guaranteed to ever be tried for this rule set. We are currently investigating alternative approaches with stronger inherent local search, including *Monte Carlo tree search* (MCTS), to address this issue.

[4] BoardGameGeek database, October 2010 (http://www.boardgamegeek.com).

6.3 Graphical games

In the last few years, a small number of researchers have worked on representing and generating simple 2d graphical-logic games. By *2d graphical-logic games* we mean those games in which gameplay is based on 2d elements moving around, colliding with each other, appearing and disappearing, and the like.[5] While 2d elements moving around and colliding with each other constitutes a rather simple set of primitives out of which to build game rules, a quite large range of games can be built out of them, including such classics as *Pong*, *Pac-Man*, *Space Invaders*, *Missile Command*, and *Tetris*. These games have a different set of properties from those typically seen in board games. They are usually characterised by featuring more complex game-agent or agent-agent interaction that could easily be handled by human calculation in a board game, including semi-continuous positioning, timesteps that advance much faster than board-game turns, multiple moving NPCs, hidden state, and physics-based movement that continues even without player input. Many such games feature an avatar which the player assumes the role of and controls more or less directly, rather than selecting pieces from a board: the player "is" the Pac-Man in *Pac-Man*, which adds a new layer of interpretation [**?**] and experiential feeling to such games, and in turn a new axis of opportunity and challenge for rule generators.

6.3.1 "Automatic Game Design": Pac-Man-like grid-world games

In a 2008 paper, Togelius and Schmidhuber describe a search-based method for generating simple two-dimensional computer games [22]. The design principles of this system was that it should be able to represent a simplified discrete version of Pac-Man, that other games should be easy to find through simple mutations, and that the descriptions should be compact and human-readable.

The games that this system can represent all take place on a grid with dimensions 15×15 (see figure **??**). The grid has free space and walls, and never changes; see figure [**?**]. On the grid, there is a player agent (represented in cyan in the screenshot) and *things* of three different colours (red, blue and green). Whether the things are enemies, food, helpers etc is up to the rules to define. The player agent and the things can move in discrete steps of one grid cell up, down, left or right. Each game runs for a certain number of time steps, and is won if the player reaches a score equal to or above a score threshold.

Representation: The game representation consists of a few variables and two matrices. The variables define the length of the game, the score limit, and the number of things of each colour. They also define the movement pattern of each colour. All things of a particular colour move in the same way, and the available movement patterns are standing still, moving randomly with frequent direction changes, mov-

[5] We borrow the term from Wardrip-Fruin [**?**].

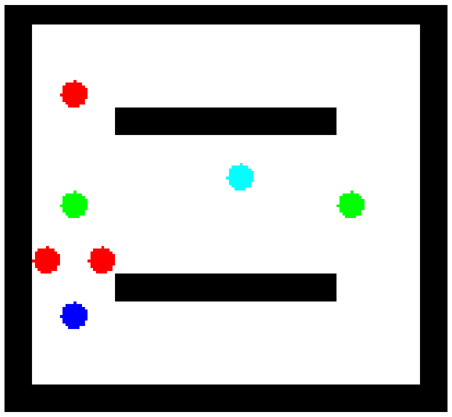

Fig. 6.5: The Automatic Game Design system by Togelius and Schmidhuger.

ing randomly with infrequent direction changes, moving clockwise along walls and moving counterclockwise along walls. The first of the the two matrices determines the collision effects of collisions between things, and between things and the agent. There is a cell for each combination of thing colours, and a cell for the combination of each colour with the player agent. The possible effects are that nothing happens, one or both things die, or one or both things teleport to a random location. For example, the matrix could specify that when a blue and a red thing collide, the blue thing dies and the red thing teleports. The other matrix is the score effects matrix. It has the same structure as the collision effects matrix, but the cells instead contain negative or positive changes to the score; for example, the player agent colliding with a blue thing might mean a score increment.

Evaluation: In the experiments described in the paper, the aim was to make games that were *learnable*. The motivation for this is the theory, introduced in various forms by psychologists such as Piaget and game designers such as Koster, that playing is learning and that a large part of the fun in games comes from learning to play them better [**?**, 10]. Translated to an evaluation function for game rules, the evaluation should reward games that are hard initially, but which are possibly to rapidly learn to play better. Under the assumption that learnability for a machine somehow reflects learnability for a human, the evaluation function uses an evolutionary learning mechanism to learn to play games. Games that are possible to win for random players receive low fitness, whereas games that can be learnt to play receive high fitness.

6.3.2 *Sculpting rule spaces:* **Variations Forever**

All of the possible games that can be specified in a particular rule encoding make up a *generative space* of games. We've just looked at one way to explore a generative space of games and pick out interesting games from the large sea of uninteresting or even unplayable games. If we define an evaluation function to rate games from the space, we can use evolutionary computation to find games that rate highly. A different approach is to carve out interesting subsets of the space, not by rating each individual game, but by specifying properties that we want games to have, or want games to avoid. This leaves a smaller generative space with only games that satisfy the desired properties; iterative refinement can then let us zoom in on interesting areas of the generative space.

Variations Forever [20] is a game-generator turned into a game, built with Answer Set Programming (ASP, see Chapter 8). In this game, the player explores different variations of game rules through playing games. The ontology and rule space is similar to but expanded compared to the rule space used in the Togelius and Schmidhuber experiment above. The games all contain things moving in a two-dimensional space, and the bulk of rules are defined by the graphical-logic effects of various types of interactions between the moving and stationary elements. However, the search mechanic is radically different. Instead of searching for rulesets that score highly on certain evaluation functions, the constraint solver finds rulesets where certain constraints are satisfied. Examples of constraints on the ruleset include: it should be possible to win the game by pushing a red thing onto a yellow thing, or it should not be possible to lose all blue things in the game while there are still green things. These constraints are specified by the game designer, and different choices of constraints will produce larger or smaller sets of games, with different properties. The player then gets a specific game randomly chosen from that constrained space (and then another one, and then another one), and part of the game is for them to try to figure out how the rules work, and what's in common between each successive game.

The aim of *Variations Forever* is not to produce a specific game deemed to be good, but to provide a way for game designers to define and "sculpt" generative spaces of games, where games can be included in or excluded from the space based on specific criteria. Players then in turn explore these designer-carved generative spaces, seeing a series of games that differ in specifics but all share the specified properties.

6.3.3 Angelina

Angelina is an ongoing project by Cook and Colton to create a complete system for automatically generating novel videogames. The system has gone through several iterations, each focusing on developing a different kind of game. In the first iteration, the focus was on discrete arcade-style 2D games, and the encoding system was along the lines of the Togelius and Schmidhuber experiment above [5]. The main change is that rather than keeping the map fixed and placing the agent randomly, Angelina sees the ruleset, the map, and the initial placement as three separate entities, and evolves all three of them using a form of cooperative coevolution. This means that each different design element is evaluated partly in isolation, according to objectives which are independent of the rest of the game. However, these individual elements are also combined into full games, which are then evaluated through automated playouts to assess how well the different elements cooperate with one another. For example, a level design might be individually fit by exhibiting a certain amount of branching or dead-ends, but be a bad fit for an object layout because it places walls over the start point for the player.

Representation: The representation of rules and mechanics in Angelina has changed through the different iterations of the software, in an attempt to increase the expressivity of the system and remove constraints on its exploration of the design space. In the first iteration of Angelina rules are composed out of a grammar-like representation of rule chunks, which produces good sets of rules, but is very dependent on the grammar it starts with. This is in turn dependent on the human that wrote the grammar. For Angelina, this is important because the research is very interested in questions of Computational Creativity. It's a good idea to think about issues like this when building a procedural content generator, however—if we want our systems to create things that are surprising and new, things that we could not have thought of ourselves, then it helps to consider whether our representation is constraining our systems with too many of our own preconceptions. Deciding how general or how specific your representation needs to be is a very important step in designing a generator of this kind.

To provide Angelina with more responsibility in designing the game's mechanics and rules, the second iteration of the software provided a less discrete domain for Angelina to explore. This version was focused on the design of simple Metroidvania-style platform games, where players incrementally gain powers that allow them to explore new areas of the world. Powerups are scattered through the game which change the value of one of a few hand-chosen variables in the game engine—such as the player's jump height, or the state of locked doors. The precise value associated with a given powerup was evolved as a design element in the coevolutionary system of this version of Angelina. This meant that Angelina could make fine-grained distinctions between the player's jump height being 120 pixels or 121 pixels, which in some cases was the difference between making the player suddenly able to access the entire game world, or carefully allowing access to a small part that would provide a more natural game progression.

This notion of game mechanics as data modifiers was carried through to the next iteration of Angelina, which took the idea a step further and opened up the codebase of the underlying game engine to Angelina. This time, instead of being given a fixed set of obvious variables to choose from, Angelina was responsible for choosing both the target value *and* the target variable, out of all the variables hidden away in the entire game's code. Below is an example 'mechanic' designed by the system. It finds the `acceleration` variable in the `player` object, and inverts the sign on its `y` component.

```
player.acceleration.y *= -1
```

In the Java-based game engine Flixel-GDX[6], which Angelina uses, this is equivalent to inverting the gravitational pull on an object, similar to the gravity-flipping mechanic in Terry Cavanagh's *VVVVVV* [**?**]. To generate this, Angelina searched through available data fields within a sample game, and generated a type-specific modifier for it (in this case, multiplying by a negative number). This exploration of a codebase was made possible by using Java's Reflection API—a metaprogramming library that allows for the inspection, modification and execution of code at runtime. Code generation and modification is a risky business, in general—the state space can very quickly become too large to explore in any reasonable timeframe, and modifying code at runtime is similarly perilous, particularly when done in as random a manner as Computational Evolution.

The approach used in Angelina tries to mitigate these problems in two ways: firstly, using Java as a basis for the system means that it has robust error handling. Generating and executing arbitrary code is liable to throw every kind of error imaginable. A typical run of Angelina will throw `OutOfMemoryExceptions` (by modifying data which triggers an infinite loop), `ArrayIndexExceptions` (by modifying variables which act as indexes into data structures) and `ArithmeticExceptions` (by modifying variables used in calculations, causing problems like division by zero). However, none of these errors cause the top-level execution of Angelina to fail. Instead, they can be caught as runtime errors, and suppressed. The mechanic which caused these errors is given a low or zero fitness score, and the system then proceeds to test the next mechanic.

The next and most important way that Angelina's design overcomes issues with code generation is the evaluation criteria used to assess whether a mechanic is good or not. Figure 6.6 shows the outline for a simple level from a Mario-like platform game. The player starts the level in the red square on the left, and can run and jump. The aim is to reach the blue square on the right. We can verify that this level is unsolvable for a given jump height—the player is simply unable to scale the wall in the center of the level. This is the game configuration that Angelina begins with when evaluating a new game mechanic. The system can then add this new game mechanic to the game's codebase, and try to solve the level by reaching the exit. If Angelina is able to make progress and get to the exit, since we know the level was previously unsolvable and only the mechanic has been added we can conclude that the mechanic adds some affordance which we did not previously have. In other words, it provides some *utility* for the player.
This black-and-white evaluation approach (either the simulation reaches the exit, or it does not) is helpful in directing search through this kind of unpredictable state space. There are a few things to note about this kind of evaluation, however. Firstly, because we are generating arbitrary code modifiers, we can't give Angelina any

[6] http://www.flixel-gdx.com

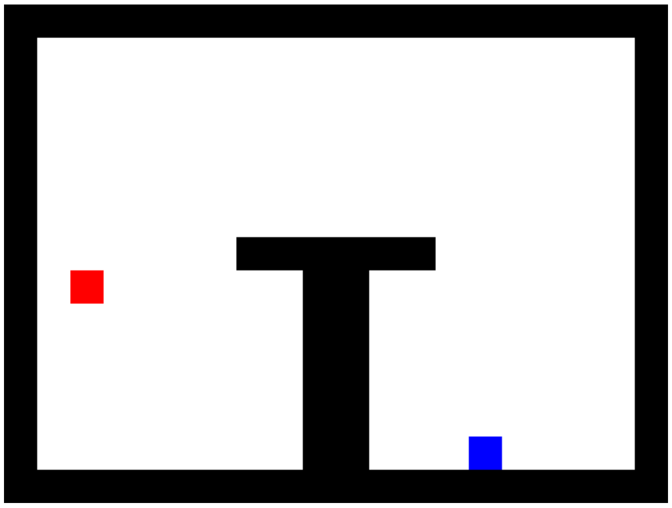

Fig. 6.6: A test level used by Angelina to evaluate generated game mechanics. The player starts in the red square on the left-hand side. They must reach the blue square on the right.

heuristics to help it test the mechanic out. We have no idea whether a given mechanic will affect the player, enemies, the level geometry, the physics system, or whether it will outright crash the game. This means that Angelina's approach to simulating gameplay with the new mechanic is to attempt a breadth-first exhaustive simulation of gameplay. While this is almost tenable for a small example level and a restrictive move set (left, right, jump and 'special mechanic'), it's hard to imagine expanding this to a game with the complexity and scale of *Skyrim*, for example, or even *Spelunky*.

The other thing to bear in mind is that how useful a mechanic is does not necessarily relate to whether it is a good idea or not. We could imagine a very useful mechanic which automatically teleports the player to the exit, but which trivialises the rest of the game's systems entirely. Similarly, many game mechanics are specifically designed to balance utility with risk (enchanting items in *Torchlight* might result in the item being destroyed, for example) or simply exist to entertain the player. This last category is very important—mechanics such as the infinite parachute in *Just Cause 2* certainly add utility to the player's mechanical toolkit, but is clearly designed to be enjoyable to interact with. Feelings like flow, tactility or immersion are difficult quantify at the best of times, and are certainly not captured by the extremely utilitarian approach taken by Angelina.

Despite these shortcomings, the use of code as a domain for procedural content generation is exciting, and holds much promise. Angelina was able to rediscover many popular game mechanics, such as gravity inversion (as seen in *VVVVVV*), and bouncing (as seen in *NightSky* [**?**]). The purely simulation-based approach also enabled Angelina to discover obscure and nuanced emergent effects in the generated code. In one case, Angelina developed a mechanic for simple teleportation, in which the player is moved a fixed amount in a particular direction when a button is pressed. This mechanic can be used for bypassing walls, but Angelina's breadth-first simulation of gameplay also discovered that by teleporting inside a wall, it was possible to jump up out of the wall, teleport back inside, and repeat the process. This technique could be used to wall-climb—even though the game had no code relating to this feature—all made possible by a single line of code modified by Angelina.

Evaluation: Evaluating Angelina as an autonomous game designer has proven difficult for a number of reasons. Cook and Colton note that during the development of Angelina many of the games features are still static and coded by hand, such as the control schemes or (in earlier versions of the software) the game's artwork. Focusing player surveys on the aspects of games which change is difficult. Comparative testing is also difficult when the expressive range of the software is as low as it has been in some versions of Angelina. The output of the system varies within a small subgenre which means it is difficult to make strong value judgements on whether one game is better than another, particularly mechanically.

Despite this, user studies have been conducted with Angelina and the results have indicated that it may be possible to get meaningful responses from players in this way. Analyses of the underlying evolutionary systems have also been conducted, to assess whether or not the fitness functions are optimising properly, and whether cooperation emerges as a result of the evolutionary structure.

6.3.4 The Video Game Description Language

The Video Game Description Language (VGDL) is an effort to create a generic and flexible but compact description language for video games of the types that were seen on early home game consoles such as the Atari 2600. In this sense, it is a direct follow-up to the efforts described above, and its conceptual structure is similar. However, it is intended to be more general in that can encode a larger range of games, and more flexible in that it decouples the description language from the game engine, the game evaluation metrics, and the generation method.

The basic design of VGDL was outlined in [**?**], and a first implementation of a working game engine for the language (together with several improvements to the design) was published in [**?**]. One of the design goals for VGDL is to be usable for "general video game playing" competitions, where learning artificial intelligence agents are tested on their capacity to play a number of games which neither the agent or the designer of the agent has seen before [**?**]. These games could be manually or automatically generated, and for the idea to be viable in the long run automatic game generation will need to be implemented at some point. The language is thus designed with ease of automatic generation in mind, though the initial stages of development have rather focused on re-implementing a range of classic games in VGDL to show the viability of doing this and test the limits of the game engine.

A VGDL game is written in a syntax derived from Python, and is therefore relatively readable. There are five parts to a VGDL game: level description, level mapping, sprite set, interaction set and termination set. The level description describes a level for the game as two-dimensional matrix of standard ASCII characters, where the level mapping defines which character maps to which type of sprite. The sprite set defines what types of sprites there are in the game and their movement behaviour, for example wall (stands still), guard robot (moves around the walls) and missile (chases the avatar). A special case is the player avatar, which the player controls directly. All sprites can obey different types of physics, such as grid-based movement or continuous movement with or without gravity. The interaction set defines most of what we call operational rules in the game, as it describes what happens when two sprites collide—similarly to the previous graphical game description efforts above, the list of possible interaction effects include death, teleportation, score increase or decrease and several others. The termination set describes various ways of ending the game, such as all sprites of a particular type disappearing, a particular sprite colliding with another etc.

6.3.5 Rulearn: Mixed-initiative game level creation

All the game generators described above have been non-interactive content generators, in that they generate a complete ruleset without human contribution. The Rulearn system by Togelius instead tries to realise interactive generation of game rules [**?**]. The system starts with the player controlling an agent obeying simple car physics in a 2D space containing agents of three other colours, moving randomly. Collisions will happen, but have no consequences. The player is also given an array of buttons which will effect consequences, such as "kill red", "increase score", "chase blue" and "split green". Every time the player presses a button, that consequence will happen. However, the system will also try to figure out why the player pressed that button. Using machine learning methods on the whole history of past actions, the system will try to figure out which game the player is playing, and induce the rules behind it. The result is a mixed-initiative system for game rules, which in early test has proved far from easy to use.

6.3.6 Strategy games

In a series of papers, Mahlmann et al. have described the evolution a system for generating key parts of strategy games. Strategy games are games, typically adversarial and themed on military conflict, where the player manages resources and moves units (representing e.g tanks, soldiers and planes) around on a board. Examples include the *Civilization* series, *Advance Wars* and *Europa Universalis*; this genre of games is closely related to real-time strategy games such as *Dune II* and *StarCraft*, except for being turn-based. They share characteristics with both traditional board games, such as typically being turn-based and playing out on a discrete board/map, and with graphical games in the relatively complex interactions between units and in the world, more complex than would be comfortably simulated in a non-digital games.

Mahlmann et al. developed a description language for strategy games, aptly called the Strategy Game Description Language (SGDL), together with a game engines that could allow a human or a computer player to play any game described in this language. In a series of experiments, different parts of strategy games represented in this language were evolved using genetic programming. In initial experiments, the focus was on evolving how much damage each type of unit could inflict on the others in a simple strategy game with the aim of creating balanced sets of units [11]. In a later set of experiments, the complete logics for the strategy game units were evolved, with the goal of finding sets of units of balanced strengths but which were functionally different between players [12]. In these experiments several new strategy game mechanics (previously unseen to the experimenters) emerged from the experiments, including units that modified the shooting range of other units based on their proximity.

6.3.7 The future: 3D games?

As we can see, all existing work on generating graphical game has targeted games in the style of classic arcade games and home console games from the early 80's, or simple arcade games. There is still considerable work to be done here, and nobody has yet constructed a system that could generate novel graphical games of high quality, comparable to the novel high-quality board games produced by Cameron Browne's Ludi system. However, there is also considerable opportunities in developing game description languages that can effectively and economically describe other types of games, and game generators that take into account the specific game design affordances and challenges that come with such games. For example, what would it take to generate playable, interesting and original FPS games?

6.4 Exercise: VGDL

The main theme of this chapter has been that generating rules heavily depends on how we encode rules for a particular kind of game, since these encodings define a space of games. The Videogame Description Language (VGDL) provides a fairly straightforward encoding for a set of graphical-logic games, allowing for some variation in gameplay styles, without going all the way to the intractable complexity of trying to encode every possible kind of game. In addition, it includes an interpreter and simulator in Python, pyVGDL, so that games produced in the encoding can easily be played.

This exercise is in two parts, with an open research question suggested as an optional third.

Part 1: Understanding a VGDL game. Download pyVGDL, which is available at https://github.com/schaul/py-vgdl. It comes with a number of example games in the *examples* directory. Choose a game, and understand its encoding. You may do this by first playing it, and trying to figure out what its rules are. Then look at the rules as they're encoded in its definition file: are they the rules you figured out? Were there other rules you didn't notice? Play it again, this time with the rules in mind. Go back and forth between the written rules and the gameplay experience until you're confident you understand what happens in the game, and how that relates to what's written in the VGDL definition.

Part 2: Write a new game in VGDL. Choose a graphical-logic game suitable for representation using VGDL's vocabulary. What are the objects in the game, and what rules specify the game's mechanics? You may want to start by first listing these on paper in natural language, and then figuring out how to encode them in VGDL. You may make up your own game, or choose an existing arcade-style game to translate to VGDL.

Part 3 (optional): Write a generator for VGDL games. As of this writing, no generator that produces VGDL games as output exists. How would one work?

References

1. Althöfer, I.: Computer-Aided Game Inventing. Tech. rep., Friedrich-Schiller Univ., Faculty Math. Comp. Sci., Jena (2003)
2. Browne, C.: Connection Games: Variations on a Theme. AK Peters, Natick, Massachusetts (2005)
3. Browne, C.: Automatic Generation and Evaluation of Recombination Games. Ph.d. dissertation, QUT, Brisbane (2008)
4. Browne, C.: Evolutionary Game Design. Springer, Berlin (2011)
5. Cook, M., Colton, S.: Multi-faceted evolution of simple arcade games. In: IEEE Conference on Computational Intelligence and Games (CIG), pp. 289–296. IEEE (2011)
6. Dickins, A.: A Guide to Fairy Chess, 3rd edn. Dover (1971)
7. Dormans, J.: Simulating mechanics to study emergence in games. In: Proceedings of the 1st AIIDE Workshop on Artificial Intelligence in the Game Design Process, pp. 2–7 (2011)
8. Hom, V., Marks, J.: Automatic design of balanced board games. In: Proceedings of the 3rd Artificial Intelligence and Interactive Digital Entertainment Conference, pp. 25–30 (2007)
9. Khaled, R., Nelson, M.J., Barr, P.: Design metaphors for procedural content generation in games. In: Proceedings of the 2013 ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 1509–1518 (2013)
10. Koster, R.: A theory of fun for game design. Paraglyph press (2004)
11. Mahlmann, T., Togelius, J., Yannakakis, G.N.: Towards procedural strategy game generation: Evolving complementary unit types. Applications of Evolutionary Computation pp. 93–102 (2011)
12. Mahlmann, T., Togelius, J., Yannakakis, G.N.: Evolving card sets towards balancing dominion. In: 2012 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8. IEEE (2012)

13. Nelson, M.J., Mateas, M.: Towards automated game design. In: AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, pp. 626–637. Springer (2007). Lecture Notes in Computer Science 4733
14. Nelson, M.J., Mateas, M.: Recombinable game mechanics for automated design support. In: Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, pp. 84–89 (2008)
15. Nelson, M.J., Smith, A.M., Mateas, M.: Computational support for play testing game sketches. In: Proceedings of 5th Artificial Intelligence and Interactive Digital Entertainment Conference, pp. 167–172 (2009)
16. Orwant, J.: EGGG: Automated programming for game generation. IBM Systems Journal **39**(3–4), 782–794 (2000)
17. Pell, B.: METAGAME: A new challenge for games and learning. In: Heuristic Programming in Artificial Intelligence 3: The Third Computer Olympiad. Ellis Horwood (1992). Extended version available as University of Cambridge Computer Laboratory Technical Report UCAM-CL-TR-276.
18. Pell, B.: METAGAME in symmetric, chess-like games. In: Heuristic Programming in Artificial Intelligence 3: The Third Computer Olympiad. Ellis Horwood (1992). Extended version available as University of Cambridge Computer Laboratory Technical Report UCAM-CL-TR-277.
19. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming (2008). http://www.gp-field-guide.org.uk
20. Smith, A.M., Mateas, M.: Variations Forever: Flexibly generating rulesets from a sculptable design space of mini-games. In: Proceedings of the IEEE Conference on Computational Intelligence and Games, pp. 273–280 (2010)
21. Smith, A.M., Mateas, M.: Computational caricatures: Probing the game design process with AI. In: Proceedings of the 1st AIIDE Workshop on Artificial Intelligence in the Game Design Process, pp. 19–24 (2011)
22. Togelius, J., Schmidhuber, J.: An experiment in automatic game design. In: IEEE Symposium On Computational Intelligence and Games, pp. 111–118. IEEE (2008)