Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

Chapter 1

Introduction

Several published games use PCG as a main part of its game play, to maintain a challenge for the player and present a new experience on re-playing the games.

Chapter 2

Existing research and framework

In this chapter we describe the existing research on the topic of game- and level-generation, which is of relevance to the subject. Additionally the framework used for game- and level-generation, and testing is described thoroughly.

2.1 Content generation

Procedural content generation is a topic that is being increasingly researched and implemented in games of various types and genres (source). PCG can help game designers in creating game elements like levels/maps, environments, weapons or AI's.

Hendrikx et al. [2013] suggests a six-layered taxonomy to describe PCG at different stages of games, while performing a survey of different ways in which PCG have been implemented.

2.1.1 Game-level generation

Almost all video games are highly dependent on designed level or map (or levels/map) in which the game play takes place. Levels are often responsible for creating increasingly challenging situations, and thereby entertaining game play. Super Mario would be way less interesting if enemies and platforms were spread randomly across each level, which would almost certainly result in the game being ether too easy or too difficult.

In recent years, game developers have begun using automatic generation of game-levels, to decrease the human work-load while still creating interesting content for games. Using PCG for creating "dungeon"-type maps have long been a successful endeavor for certain type of games as seen in *Rogue* [1980], the *Diablo*-series, and in newer times *Minecraft* [2011] and *Spelunky* [2009]. PCG were additionally successfully implemented in strategy games like "Civilization II", "Command & Conquer: Tiberian Sun" and "XCom: UFO Defense", but otherwise automatically generating levels have not been hugely successful in other game genres.

Several scientific projects have focused on the topic as well with an array of different approaches, some requiring player-input, or simply a helper for human level-designer. In addition to the other game genres mentioned above, a lot of work has been done in generating levels for puzzle games (ie. Sokoban and Cut The Rope). Most of the work is focused on generating levels for single game, where some attempt to create general level generation procedures.

2.1.2 Other uses of PCG

Other games have been published applying PCG in different ways. In *Borderlands 2* [2012] the player have access to an extreme amount of different weapons, designed by automatic generation.

2.1.3 Advantages and disadvantages using PCG

Having a procedure for generating content in games has a number of advantages:

- Content generation allows games to potentially have an endless amount of possibilities, often resulting in higher re-play value than linear games.

- PCG can help game designers generate game levels, often simply by letting the designer explore possible layouts, add decorations or finishing up human-designed levels.

PCG also has some problems for generating content for levels:

- Generated content can have a feeling of being unauthentic and/or too random. For instance, the positions of decorations in a level (barrels, paintings etc.) can be important for humans, while they are not for computers.

- In games where the player follows a story it can be difficult to automatically generate levels, that follow what the game designer intend to happen

2.2 Game generation

Generating complete games through algorithms is a problem that is being increasingly researched, but work has been done on the topic for last decade. Because the problem is in general quite large, a subset of the problem is usually handled: Only generating certain types of games-, or using different (restricted) frameworks. Video games may consist of a large number of tangible and intangible components, including rules, graphical assets, genre conventions, cultural context, controllers, character design, story and dialog, screen-based information displays, and so on Cook and Colton [2014], Liapis et al. [2014], Nelson and Mateas [2007].

In this project we look specifically at generating the game-play and -setting of games, i.e. defining a set of game-rules and -objects, and specifying the levels in which the game-play takes place. The two main approaches that have been explored in generating game rules are reasoning through constraint solving Smith and Mateas [2010] or search through evolutionary computation or similar forms of stochastic optimisation Togelius and Schmidhuber [2008], Browne [2008], Font et al. [2013]. In either case, rule generation can be seen as a particular kind of procedural content generation Nelson et al. [2014].

It is clear that generating a set of rules that makes for an interesting and fun game is a hard task. The arguably most successful attempt so far, Browne's Ludi system, managed to produce a new board game of sufficient quality to be sold as a boxed product Browne [2008]. However, it succeeded partly due to restricting its generation domain to only the rules of a rather tightly constrained space of board games. A key stumbling block for search-based approaches to game generation is the fitness/evaluation function. This function takes a complete game as input and outputs an estimate of its quality. Ludi uses a mixture of several measures based on automatic playthrough of games, including balance, drawishness and outcome uncertainy. These measures are well-chosen for two-player board games, but might not transfer that well to video games or single-player games, which have in a separate analysis been deemed to be good targets for game generation Togelius et al. [2014]. Other researchers have attempted evaluation functions based on the learnability of the game by an algorithm Togelius and Schmidhuber [2008] or an earlier and more primitive version of the characteristic that is explored in this paper, performance profile of a set of algorithms Font et al. [2013].

A typical approach for generating complete games is searching in a space of possible games. This basically requires two things: That the ratio of enjoyable games in the set is not too low - otherwise those games might never be found. To increase this ratio a game description language (GDL) is often used (searching through all possible Java or C programs would lead to an enormous amount invalid games). Also, to search through a set of games it is necessary to be able to calculate a fitness value for each game, valuing how enjoyable the game is.

2.2.1 Using AIs to analyze games

A large amount of games contain puzzle-elements, but only a few has puzzles as the main game-play.

2.2.2 Something to note: Levels

The problem of generating complete games is heavily linked to the problem of generating levels since most games are deeply tied to the geometry and object-placement defined in levels.

2.3 Framework: Game description languages

Regardless of which approach to game generation is chosen, one needs a way to represent the games that are being created.[1] For a sufficiently general description of games, it stands to reason that the games are represented in a reasonably generic language, where every syntactically valid game description can be loaded into a specialised game engine and executed.

Stanford GDL

There have been several attempts to design such GDLs. One of the more well-known is the Stanford GDL, which is used for the General Game Playing Competition Genesereth et al. [2005]. That language is tailored to describing board games and similar discrete, turn-based games; it is also arguably too verbose and low-level to support search-based game generation.

PuzzleScript

Another attempt at an VGDL is called PuzzleScript, created by game designer Stephen Lavelle. The language (as its name suggest) is focused on turn-based puzzle games, but the engine allows for simple forms of animation and movement. The language is relatively high level compared to Stanford GDL, but does not support that large set of video games.

2.3.1 VGDL

The various game generation attempts discussed above feature their own GDLs of different levels of sophistication; however, there has not until recently been a GDL for suitably for a larger space of video game types- and genres. The Video Game Description Language (VGDL) is a GDL designed to express 2D arcade-style video games of the type common on hardware such as the Atari 2600 and Commodore 64. It can express a large variety of games in which the player controls a single moving avatar (player character) and where the rules primarily define what happens when objects interact with each other in a two-dimensional space. VGDL was designed by a set of researchers Levine et al. [2013], Ebner et al. [2013] (and implemented by Schaul Schaul [2013]) in order to support both general video game playing and video game generation. In contrast to other GDLs, the language has an internal set of classes, properties and types that each object can defined by, which the authors suggest the user to extend.

Objects have physical properties (i.e. position, direction) which can be altered either by the properties defined, or by interactions defined between specific objects. Playing the games also required a specified level which defines a set of game-tiles deciding the initial locations of sprites. When running games sprite can move in between game-tiles (if the **speed** is a fraction).

A VGDL description has four parts:

SpriteSet Defines which sprites can appear in the game. Each sprite must be designated by a class, in which a set of predefined actions exists. Also a set of parameters can be fed to each sprite, configuring for instance the speed or how often the sprite takes an action, and the parameters used for the different sprite-classes – for instance, for the class **Flicker** (a simple extension to the base sprite, the sprite is destroyed after a speecifed amount of time) the lifetime can be adjusted. Sprites can additionally be designed in a tree structure, where multiple sprites have the same parent sprite, making more possibility for the interactions and terminations described below.

InteractionSet Each line defines what happens when a set of two sprites collide with each other (located on the same game-tile). Each interaction is represented by a class defining the action to take (i.e. push back, or kill sprite), and a set of parameters specific to each interaction-class, as well as the score achieved for letting the interaction happen.

TerminationSet Defines how the game can end. Each line in this set has a win parameter, which is set to true or false; winning or losing the game. Each termination-function is represented by a class defining under which conditions the game should end, which can for instance when there exists 0 of a certain sprite (when all of the sprites are killed).

[1] See Nelson et al. [2014] for a discussion of game-rule representation choices.

LevelMapping The job of the `LevelMapping` is to translate from a character (`char`) in a level-file (explained below), to sprites from the `SpriteSet`. A single mapping can be shared by several sprites, causing the sprites to be created on the same game-tile.

```
BasicGame
  SpriteSet
    city  > Immovable color=GREEN img=city
    explosion > Flicker limit=5 img=explosion
    movable >
      avatar  > ShootAvatar stype=explosion
      incoming >
        incoming_slow  > Chaser stype=city color=ORANGE speed=0.1
        incoming_fast  > Chaser stype=city color=YELLOW speed=0.3

  LevelMapping
    c > city
    m > incoming_slow
    f > incoming_fast

  InteractionSet
    movable wall  > stepBack
    incoming city > killSprite
    city incoming > killSprite scoreChange=-1
    incoming explosion > killSprite scoreChange=2

  TerminationSet
    SpriteCounter stype=city   win=False
    SpriteCounter stype=incoming win=True
```

Figure 2.1: Example of VGDL description - a simple implementation of the game Missile Command

Additionally each game can only be played using level files. A level file is written using the `LevelMapping` characters, where each character defines a tile of the game, and spaces defines empty tiles.

```
w    m   m    m   m   m m mw
w                          w
w                          w
w                          w
w                          w
w                          w
w            A             w
w                          w
w                          w
w                          w
w   c  c  c  c  c  c  c  c w
wwwwwwwwwwwwwwwwwwwwwwwwwwww
```

Figure 2.2: Example of VGDL level description - a level for the implementation of the game Missile Command

The GVG-AI framework is a testbed for testing general gameplaying controllers against games specified using VGDL. Controllers are called once at the beginning of each game for setup, and then once per clock tick to select an action. Controllers do not have access to the VGDL descriptions of the games. They receive only the game's current state, passed as a parameter when the controller is asked for a move. However these states can be forward-simulated to future states. Thus the game rules are not directly available, but a simulatable model of the game can be used.

GVG-AI example games

The framework additionally contains 20 hand-designed games, which mostly consist of interpretations of classic video games. Figure 2.2 shows the VGDL description of the game *Missile Command*. Most of the games are inspired by classic arcade- and Atari games (e.g. Boulderdash, Frogger, Missile Command and Pacman), while some are original creations by the General Video-Game AI Competition's organizers. The games can in general be described (except for *Sokoban*) as action arcade games, in that the player controls a single avatar which must be moved quickly around in a 2D-setting to win, or to get a high score (the player is able to increment a score counter in all of the games).

Below is a short summary of each of games, describing the winning conditions, and how the player can increase his/her score:

Aliens VGDL interpretation of the classic arcade game *Space Invaders*. A large amount of aliens are spawned from the top of the screen. The player wins by shooting all the approaching aliens.
Goal Destroy all the incoming aliens, without being hit by them or their projectiles.
Scoring Destroy all the incoming aliens, without being hit by them or their projectiles.

Boulderdash VGDL interpretation of *Boulder Dash*. The avatar has to dig through a cave to collect diamonds while avoiding being smashed by falling rocks or killed by enemies.

Butterflies The avatar has to capture all butterflies before all the cocoons are opened. Cocoons open when a butterfly touches them.

Chase About chasing and killing fleeing goats. However, if a fleeing goat encounters the corpse of another, it get angry and start chasing the player instead.

Digdug VGDL interpretation of *Dig Dug*. Avatar collects gold coins and gems, digs his way through a cave and avoid or shoot boulders at enemies.

Eggomania VGDL interpretation of *Eggomania*. Avatar moves from left to right collecting eggs that fall from a chicken at the top of the screen, in order to use these eggs to shoot at the chicken, killing it.

Firecaster Goal is to reach the exit by burning wood that is on the way. Ammunition is required to set things on fire.

Firestorms Player must avoid flames from hell gates until he finds the exit of a maze.

Frogs VGDL interpretation of *Frogger*. Player is a frog that has to cross a road and a river, without getting killed.

Infection Objective is to infect all healthy animals. The player gets infected by touching a bug. Medics can cure infected animals.

Missile Command VGDL interpretation of the classic arcade game *Missile Command*. Player has to destroy falling missiles, before they reach their destinations. If the player can save at least one city, he wins.

Overload Player must get to the level after collecting coins, but cannot collect too many coins, as he will be too heavy to traverse the exit.

Pacman A VGDL interpretation of *Pac-Man*. Goal is to clear a maze full with power pills and pellets, and avoid or destroy ghosts.

Portals Objective is to get to a certain point using portals to go from one place to another, while at the same time avoiding lasers.

Seaquest VGDL interpretation of *Seaquest*. Avatar is a submarine that rescue divers and avoids sea animals that can kill it. The goal is simply to a high score.

Survive Zombies Player has to flee zombies until time runs out, and can collect honey to kill the zombies.

Whackamole VGDL implementation of the classic arcade game *Whac-a-Mole*. Must collect moles that appear from holes, and avoid a cat that mimics the moles.

Zelda VGDL interpretation of *Legend of Zelda*. Objective is to find a key in a maze and leave the level. Player also has a sword to defend himself against enemies.

Camelrace Player needs to get to endpoint to win.

Sokoban The objective is to move the boxes to holes, until all boxes disappear or the time runs out.

2.3.2 Restrictions of VGDL

The VGDL implementation of the GVG-AI competetion is essentially (without great extension) only able to describe certain action-arcade games, and puzzle games. For instance, the lack of possibility to traverse different levels make **Adventure** games impossible to make, and the restriction of only being able move a single character with the keyboard (the avatar) makes great restriction in creating **Strategy** games.

In this project we will focus on the two main type of games describable in VGDL: Action-arcade games and turn-based puzzle games. We make the simple distinction that arcade games contain elements (sprites) that move by themselves, whereas interactions only occur as a result of the avatar moving in puzzle games. This puts the game *Portals* in the arcade genre, even though it contains puzzle elements.

Chapter 3

Research Objectives

The main objectives of this study was to create a generator being able to create simple games and levels, enjoyable for human players. The focus of the study was to create fitness functions able to differentiate between games (and levels) of different quality, to be able to create interesting content. Making the generator able to create a large amount of games, and then select the ones with the highest quality.

3.1 General Evaluation - Arcade-/Action Games

The first goal was to use the results of a series of general game-playing (knowledge free) algorithms to value a given game.

3.2 Breadth First Search - Puzzle Games

The second goal was to

3.3 Assumptions

We assume that the games metioned are of high quality (i.e. highly enjoyable for human players). This is not always clear when playing the games in their form, since they need crucial element to keep a game fun: Audio, graphic effects, control-scheme fiiting to the games. We reason this because they are clones of existing games.

Chapter 4

Extending VGDL

We created a series of extensions to VGDL during the project.

4.1 Writing new VGDL games

To increase the size of the set of designed games, and to allow for more precise analysis we created a series of fourteen new game descriptions in VGDL.

Another important reason for creating new descriptions, was to introduce a series of *puzzle games* to be analysed, since, as mentioned in Section 2.3.1 the example games from the GVG-AI competetion has a severe lack of that game-genre.

4.1.1 Describing existing games in VGDL

The main goal when developing a game - which also applies in this case - is to ensure that it is enjoyable to human-players. Therefore the games implemented are all interpretations of published existing games (and not original creations). As described in **Section 2.3.2** VGDL can only describe relatively simple games of certain type/genre, and so only a limited number of games can be translated without severely changing the game-play. The games are in general "exact" copies of the original ones - containing all the game-play features and interactions but lacking elements like audio, graphics and controls. However many lack certain (non-essential) game-play features like bonus point sprites, infrequently appearing enemies or features which only appear in some levels of the games.

Action-arcade games

A set of four Atari -arcade and -2600 games were found to be suitable for interpretation, which we then wrote the description for in VGDL.

Below is short description of each of the games that were translated, and used in later tests:

Crackpots [1983] VGDL implementation of the classic arcade game *Whac-a-Mole*. Must collect moles that appear from holes, and avoid a cat that mimics the moles.

Solar Fox VGDL interpretation of *Legend of Zelda*. Objective is to find a key in a maze and leave the level. Player also has a sword to defend himself against enemies.

Astrosmash [1981] The player controls a laser cannon at the bottom of the screen, with the goal of shooting down as many incoming meteors, bombs and other objects. Points are earned by destroying objects, but lost if the objects reach the ground. The game ends if the laser cannon is hit a few times by the incoming objects.

Centipede The objective is to move the boxes to holes, until all boxes disappear or the time runs out.

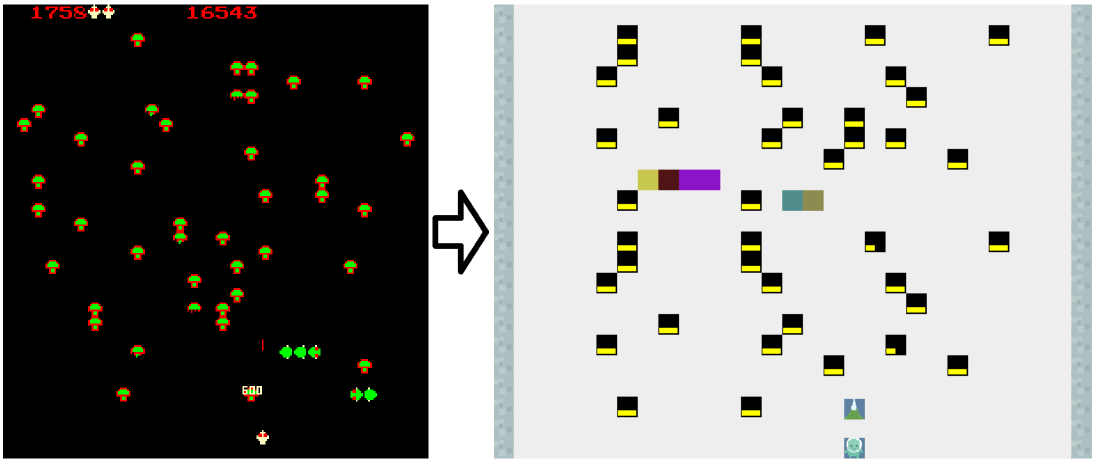

Figure 4.1: Interpretation of the classic arcade game *Centipede* [1983]

Puzzle games

A set of puzzle games featuring a single avatar was also found to be suitable for conversion to VGDL. Several of these games can be described as *Sokoban*-clones, but each containing unique game elements. Below is a description of each of ten puzzle games created.

Bait VGDL interpretation of the classic arcade game *Space Invaders*. A large amount of aliens are spawned from the top of the screen. The player wins by shooting all the approaching aliens.
Goal Destroy all the incoming aliens, without being hit by them or their projectiles.
Scoring Destroy all the incoming aliens, without being hit by them or their projectiles.

The Citadel VGDL interpretation of *Boulder Dash*. The avatar has to dig through a cave to collect diamonds while avoiding being smashed by falling rocks or killed by enemies.

Bombuzal The avatar has to capture all butterflies before all the cocoons are opened. Cocoons open when a butterfly touches them.

Chip's Challenge About chasing and killing fleeing goats. However, if a fleeing goat encounters the corpse of another, it get angry and start chasing the player instead.

Bolo Adventures VGDL interpretation of *Dig Dug*. Avatar collects gold coins and gems, digs his way through a cave and avoid or shoot boulders at enemies.

(Real) Sokoban VGDL interpretation of *Eggomania*. Avatar moves from left to right collecting eggs that fall from a chicken at the top of the screen, in order to use these eggs to shoot at the chicken, killing it.

Brainman Goal is to reach the exit by burning wood that is on the way. Ammunition is required to set things on fire.

Modality Player must avoid flames from hell gates until he finds the exit of a maze.

Painter VGDL interpretation of *Frogger*. Player is a frog that has to cross a road and a river, without getting killed.

Zen Puzzle Objective is to infect all healthy animals. The player gets infected by touching a bug. Medics can cure infected animals.

4.2 Creating new VGDL controllers

To be able to probe VGDL games in more detail, we created a series of new AI controllers for various purposes. In total five new controllers was generated: Three to play the action-arcade style games, and two only focused on turn-based puzzle games.

4.2.1 Action-arcade controllers

One-step

Deep-search

Explorer

The Explorer was proven to be of a decent quality by getting a 2nd / 7th place in the GVG-AI competition.

4.2.2 Puzzle controllers

Breadth first

Best first

4.3 FastVGDL

Since the GVG-AI framework has some functions and properties which are not interesting for some parts of this work, we created an implementation of a lighter version of VGDL, solely focused on analysing puzzle games in more detail. The implementation is basically a clone of the GVG-AI framework, but with several time- and memory consuming features removed, which in part was possible due to the puzzle games being relatively more simple (for instance, only movement from one tile to another is possible in FastVGDL, whereas sprites can move and collide in-between tiles in the GVG-AI framework).

SMALL TEST SHOWING TIME AND MEMORY DIFFERENCE PL0X!!!

Chapter 5

Automatic generation of VGDL descriptions

This section describes the experimental- setup, and methodology used generating VGDL descriptions.

5.1 Generating descriptions

5.1.1 Example games: Arcade-/action

Two of the 20 games from the GVG-AI framework were deemed too monotonous after initial tests. In these two games the controllers all had similar scores for each run - or with only one controller being able to increase its score. The remaining 18 hand-designed VGDL game descriptions were chosen to be used as a baseline for testing and game generation.

5.1.2 Mutation of example games

A mutation process was repeatedly applied for each of the 18 example games mentioned in section **??**. The process consisted of changing the set of interaction rules (i.e. lines from the InteactionSet) defined in each game description. For each mutation, each interaction rule had a 25% chance of being mutated, but with a requirement that at least one rule were changed. Mutation occured by changing the objects in that interaction rule, the function on collision between said objects, and/or the function's parameters.

Several contrains were used during each mutation to avoid games with non-valid descriptions (which can cause crashes in the GVG-AI framework). Additionally, several constraints were used for the different function parameters, as to only allow "realistic" values. The range of these constraints were extrapolated (and slightly extended) from the example games. For instance, the parameter *limit* used by certain rules was limitied to values between 0 and 10, as the same is true for the rules of the example games. This process was applied 20 times for each example game description, resulting in 360 generated game descriptions.

When testing the mutated games the same level descriptions as for their original counterparts were used (those mentioned in section **??**).

5.1.3 Random game generation

A set of 400 random VGDL game descriptions were generated by constructing the textual lines for different parts of a VGDL description: Generating an array of sprites (for the SpriteSet), interaction-rules (InteractionSet), termination-rules (TerminationSet) and level mappings (LevelMapping).

Before generating descriptions, we used similar constraints to those mentioned in section 5.1.2, partly to avoid generating descriptions with invalid elements, and partly to increase the proportion of interesting outcomes. The number of sprites, interaction- and termination-rules were randomly chosen, limited to 25, 25, and 2, respectively. Furthermore, a simple level description (only containing one of each sprite) was generated for each of the generated game descriptions for test purposes.

5.2 Results of playing

The seven controllers mentioned in section ?? were used to play through a set of example-, mutated and randomly generated games. Because of CPU budget limitations, each game was played for a maximum of 800 clock ticks, and each controller was restricted to use 50 ms on each tick. In the following sections, we show results of these tests, analyse the average of all play-throughs for each controller, and compare the results with each other.

To more accurately compare the score for the different controllers across the range of different games, we normalise each score using a max-min normalisation. Normalised averages and win rate averages are shown in Figures ?? and ??, respectively. In Figure ??, it is possible to see that the difference between the highest and lowest scores is greater in the example and mutated games than in the generated games. On the other hand, the average win rate of generated games surpasses both examples and mutated games, as shown in Figure ??.

In addition to the score and win-rate, the average entropy of actions chosen for the player avatar is shown in the tables below.

5.2.1 Designed games

controller	*score mean*	*std.dev.*	*normalised-mean*	*winrate*	*act-entropy*
Explorer	15.36	30.53	0.8295	0.1011	0.9796
MCTS	5.76	10.48	0.5052	0.0389	0.9954
GA	4.76	7.90	0.5004	0.0189	0.8318
Onestep-S	7.17	16.55	0.4603	0.0161	0.9780
Onestep-H	−2.77	22.76	0.2985	0.0556	0.2632
Random	3.02	7.69	0.3136	0.0033	0.9997
DoNothing	−1.44	5.03	0.1630	0	0

Figure 5.1: Results from the 20 example games

Averages and win-rates from the 18 human-designed example games are shown in Figure 5.1. The distributions of normalised scores show that more intelligent controllers tend to have more success. It is worth noticing that the *score mean* and *normalised score mean* have slightly different orderings. Notice also that distributions are slightly different when analysing the results of individual games. For instance, in *Aliens*, Random has a higher average than Onestep.

5.2.2 Generated games

Figure 5.2 shows results for the 65 randomly generated games, with problematic games removed according to the same criteria as in the previous section.

First of all, *score std. deviations* are much higher than in the previous games, with the minimum being 199,406.58, over 1500 times larger than the highest in the set of example games (i.e. 121.55, by Explorer). Clearly, only the *normalised mean* can be on this set to compare scores across the the different game types. The *normalised score means* and *win-rates* both have values that are more closely clustered togther, than in the previous game sets.

5.2.3 Mutated games

When mutating games, two types of games are problematic: Games where the controllers never increase their score (and never win), and games where too many objects are created and each frame

controller	*score mean*	*std.dev.*	*normalised-mean*	*win-rate*	*act-entropy*
Explorer	319.44	4852.31	0.5583	0.1967	0.8674
MCTS	542.32	5773.92	0.4401	0.2200	0.9818
GA	581.33	6263.91	0.4514	0.1978	0.7783
Onestep-S	344.16	5026.86	0.4132	0.1622	0.9640
Onestep-H	689.14	6159.38	0.4309	0.1744	0.5591
Random	322.81	4492.68	0.3195	0.1611	0.9981
DoNothing	566.19	5065.79	0.3625	0.1667	0

Figure 5.2: Results from randomly generated games

end up taking too long (> 50ms). We exclude both types of games in the following analysis.

Averages from playing the remaining 146 mutated games (of 200 total) are shown in Figure 5.3. The scores have higher means and standard deviations, indicating outliers in the data. The ordering of the *normalised score mean*, however, shows a similar pattern as for the example games, with Explorer again excelling.

controller	*score mean*	*std.dev.*	*normalised-mean*	*win-rate*	*act-entropy*
Explorer	45.22	203.71	0.8049	0.0736	0.9491
MCTS	24.14	168.33	0.4495	0.0267	0.9957
GA	26.13	170.08	0.4545	0.0226	0.8179
Onestep-S	30.21	156.71	0.4200	0.0130	0.9466
Onestep-H	10.87	147.38	0.3037	0.0863	0.2376
Random	17.47	176.04	0.2567	0.0127	0.9978
DoNothing	13.59	159.16	0.1889	0.0068	0

Figure 5.3: Results from mutated games

5.2.4 Outcome

The conclusion of the above tests is that the the result distribution of controllers can be used to score a game, with a high probability.

Chapter 6

Evaluation of games (fitness functions)

6.1 Action-arcade games: Fitness functions

From the above tests we can see that it is interesting to write a fitness function for controllers results, to be able to find out if new generated games are of a high quality.

6.1.1 Action-arcade games: Generated levels

Results of what happens when generating levels for the example games.

Chapter 7

Puzzle generation

7.1 Turn-based puzzle games

Definition of games

The games described in this section are games where the puzzles are the only game-play. There is no fast movement, or quick reaction time required to win. Additionally, because the games are described using the GVG-AI framework, the games focus around a player avatar which can only move up, down, left or right (also, in the games described below, all movement are from one game-tile to another).

7.1.1 Games

Description of the puzzle games used in tests.

7.1.2 Puzzle solving AIs

As mentioned in (WHERE ITS MENTIONED), the controllers from the GVG-AI competietion are not well-suited for playing puzzle games, because the goal can often only be achieved by applying a specific series of moves, which the controllers are not very good at. Two general puzzle solving algorithms were implemented. The overall goal of the controllers were to analyze the game as much as possibl, rather than finding a solution fast, or using as low memory as possible They controllers use fact that wall-sprites in the different games always push back the player, and so the controllers do not try to move into a wall – this is achieved by storing the positions of every wall-sprite at the start of the game. Also, the controllers store each path it is had tried. For each path a game state is calculated and stored, by finding the position and type of each (non-wall) sprite in the game. A path is cut off if the calculated game state is the same as has appeared before.

Breadth-first search A breadth-first search algorithm was implemented by using a queue and letting each node expand to the adjacent tiles, using the "tricks" described above. Additionally the controller was given a low- and high-memory option: In the high memor

Best-first search

7.1.3 Level generation

A setup for creating levels for a given game were constructed using an evolutionary algorithm. The level generator described in section **??** was used to generate simple levels for games. In addition two extra options were added: Wall-sprites and ground-sprites. This reduces a lot of troubles since the avatar would otherwise be able to escape the level in a lot of the games. Ground-sprites signify which sprite should appear on empty locations.

The fitness function used in evolving levels was found by letting the two "puzzle solving AIs" play through the level.

Mutation

Crossover

Two different levels were constructed into a new, by going over each tile on the level and with 50% chance take the sprite residing in the two different original levels.

The setup was as follows:

```
while not at end of this document do
    read current;
    if understand then
        go to next section;
        current section becomes this one;
    else
        go back to the beginning of current section;
    end
end
```

Algorithm 1: How to write algorithms

7.1.4 Level generation for generated games

The Title

Put here cool text like what is going on in the wauw

This is subsubsection with title “Level generation for generated games”.

Bibliography

Activision, Inc. Crackpots, 1983.

Cameron Browne. *Automatic generation and evaluation of recombination games*. PhD thesis, Queensland University of Technology, 2008.

Michael Cook and Simon Colton. Ludus ex machina: Building a 3d game designer that competes alongside humans. In *Proceedings of the 5th International Conference on Computational Creativity*, 2014.

Marc Ebner, John Levine, Simon M Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. Towards a video game description language. *Dagstuhl Follow-Ups*, 6, 2013.

José María Font, Tobias Mahlmann, Daniel Manrique, and Julian Togelius. Towards the automatic generation of card games through grammar-guided genetic programming. In *FDG*, pages 360–363, 2013.

Gearbox Software. Borderlands 2, September 2012. URL `http://www.borderlands2.com/`.

Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62, 2005.

Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 9(1):1, 2013.

John Levine, Clare Bates Congdon, Marc Ebner, Graham Kendall, Simon M Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General video game playing. *Dagstuhl Follow-Ups*, 6, 2013.

Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. Computational game creativity. In *Proceedings of the 5th International Conference on Computational Creativity*, 2014.

Mattel Electronics. Astrosmash, 1981.

Mojang. Minecraft, November 2011. URL `http://www.minecraft.net/`.

Mark J. Nelson and Michael Mateas. Towards automated game design. In *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, pages 626–637. Springer, 2007. Lecture Notes in Computer Science 4733.

Mark J. Nelson, Julian Togelius, Cameron Browne, and Michael Cook. Chapter 6: Rules and mechanics. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2014. URL `http://www.pcgbook.com`. (To appear.).

Tom Schaul. A video game description language for model-based or interactive learning. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

Adam M Smith and Michael Mateas. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Proceedings of the 2010 IEEE Symposium on Computational Intelligence and Games*, pages 273–280, 2010.

Julian Togelius and Jürgen Schmidhuber. An experiment in automatic game design. In *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games*, pages 111–118, 2008.

Julian Togelius, Mark J. Nelson, and Antonios Liapis. Characteristics of generatable games. In *Proceedings of the 5th Workshop on Procedural Content Generation in Games*, 2014.

Michael Toy and Glenn Wichman. Rogue, 1980.

Derek Yu. Spelunky, September 2009. URL http://www.spelunkyworld.com/.