Abstract

In this thesis a framework for generating games- and levels in the video game description language VGDL is presented. The generator is able to automatically generate games, test the playability and difficulty using AI controllers, and score each game according to a series of evaluation criteria.

The research and framework is separated into two parts, divided by different game genres: 1) An analysis of 2-dimensional action-arcade games, with the goal of producing a generator focusing on evolving a set of game-rules using the results from a series of general game-playing algorithms. And 2) An examination of 2-dimensional turn-based puzzle games- and the especially the levels of puzzle games, with the ambition of being able to generate simple sets of game-rules and focusing on the evolution of levels for those games.

The results of this thesis show that automatically generating game design is indeed very promising, with several interesting complete Sokoban-like puzzle games generated, but indicating that creating action games is in general a more difficult terrain for generating rules- and mechanics enjoyable for humans.

Acknowledgement

I would like to say "thanks brah"

Contents

Chapter 1

Introduction

In this chapter the idea of generating complete games-, and the content within them is introduced. The research goals of the thesis is then shown, and an overview of the project is presented.

1.1 Introduction to content generation

Procedural generation of game content (levels, textures, items, quests, audio, characters etc.) has become a widely used tool for video-game developers, and has been applied to an array of different game- genres and types. Automatically generating content allow game developers to provide more content, with more differentiation, from a limited supply of assets, most often by helping to produce levels, maps or dungeons for their games. PCG is often used to maintain a challenge for the player by presenting an unpredictable and renewed game experience on re-playing the games in which it is used.

Rogue [?] is one of the earliest examples of a game using procedurally generated levels (dungeons) as a main part of the game design. The space-trading game Elite [?] used PCG to both save disk space, and to generate a large array of planets and galaxies, each with a different set of unique properties generated for every play through.

Since then several highly successful games, from genres including platform-, first person shooter- (FPS) and strategy games, have used PCG to generate levels-, maps and worlds, as either used in their main game mode (e.g. Spelunky [?], Minecraft [?]), or as an extra feature, making the games have additional re-playability (e.g. Civilazation II(-V) [?], Heroes of Might and Magic III [?]). PCG has additionally been used to create a wider array of weapons and items, especially in role-playing games (RPGS) where "looting" (finding new gear) is a main part of the game design (e.g. Borderlands-series [?], Diablo-series [?], Torchlight-series [?], Dark Age of Camelot [?]).

Several game-developers, companies and research groups has additionally used PCG to create content of other types of game-content, or at different stages, including complete worlds, environments and stories ([?]), and graphics and audio (RoboBlitz [?], .kkrieger [?]).

1.2 Introduction to game generation

An interesting next step to take is to not only generate content for a video game, but to generate completely new games and game design – i.e. to generate the set of rules and mechanics of a game, and necessarily to generate the content for actually playing the game (even simple games like Tetris or Pong is dependant on a playing field (or level) and a definition of each object).

A possible approach to automatically generating complete games might be to search through a space of programs represented in a programming language like C or Java. However, the proportion of programs designed in such languages that can even be considered a game is rather small – and much less, the amount games that a human player would find enjoyable. To reduce this problem a game description language (GDL), designed to encode games (and only games), can be used, severely increasing the density density of well-defined games.

Even searching a fairly well defined space of possible games, I still have a need for some way of actually telling good games from bad ones (or at least, mediocre games from really bad games) – a fitness function is needed. A fitness function could partly consist of inspecting the generated rules as expressed in the GDL, e.g. to make sure that the player can interact with the game in some way, and that there are winning- and losing conditions, which could in principle be fulfilled. However there are many bad games that fulfil such criteria, so intuitively it seems one need to actually play the games that are generated.

A fitness function for generating complete games therefore needs to incorporate a capacity to automatically play the games it is evaluating, giving each game a score dependant on certain values from the results of playthroughs.

1.3 Research Objectives

The main objectives of this study was to construct a generation process able to create simple games and levels using the game description language (VGDL). With the goal of making games as enjoyable for human players as possible. The main focus of the study was to create a fitness function able to differentiate between games (and levels) of different quality, to be able to search and sort a set of generated games, and thereby create interesting content.

The main approach used to create the fitness function was to use the results of a series of general game-playing (knowledge free) algorithms, and using the collection of their performance profiles to construct individual fitness features. A series of human-designed games, assumed to be human-enjoyable, was used as a baseline for "good games" and was intensely used in choosing and weighing fitness features. This approach was focused on a specific genre and type of games (*arcade-action games*), which is in the focus of the framework used (the GVG-AI framework). Additionally this type of games fit very well with general game-playing algorithms used, which are not very good at exploring deep into a space of game-states (which is often not need in the game-genre – quick movements to avoid enemies and increase the score is in the focus).

The second part of this thesis is focused on games of the *puzzle*-genre described in VGDL, which overall has a much smaller game-state space (the amount of possible arrangements of sprites and values) and so potentially are more well-suited for a more in-depth analysis. This approach is mainly focused on generating new game-levels for either existing games, or generated VGDL programs, by completely searching through all possible states of a game (using a breadth-first algorithm), finding all possible solutions, with the goal of being able to judge the difficulty, and thereby the quality of a game/level pair.

Assumptions

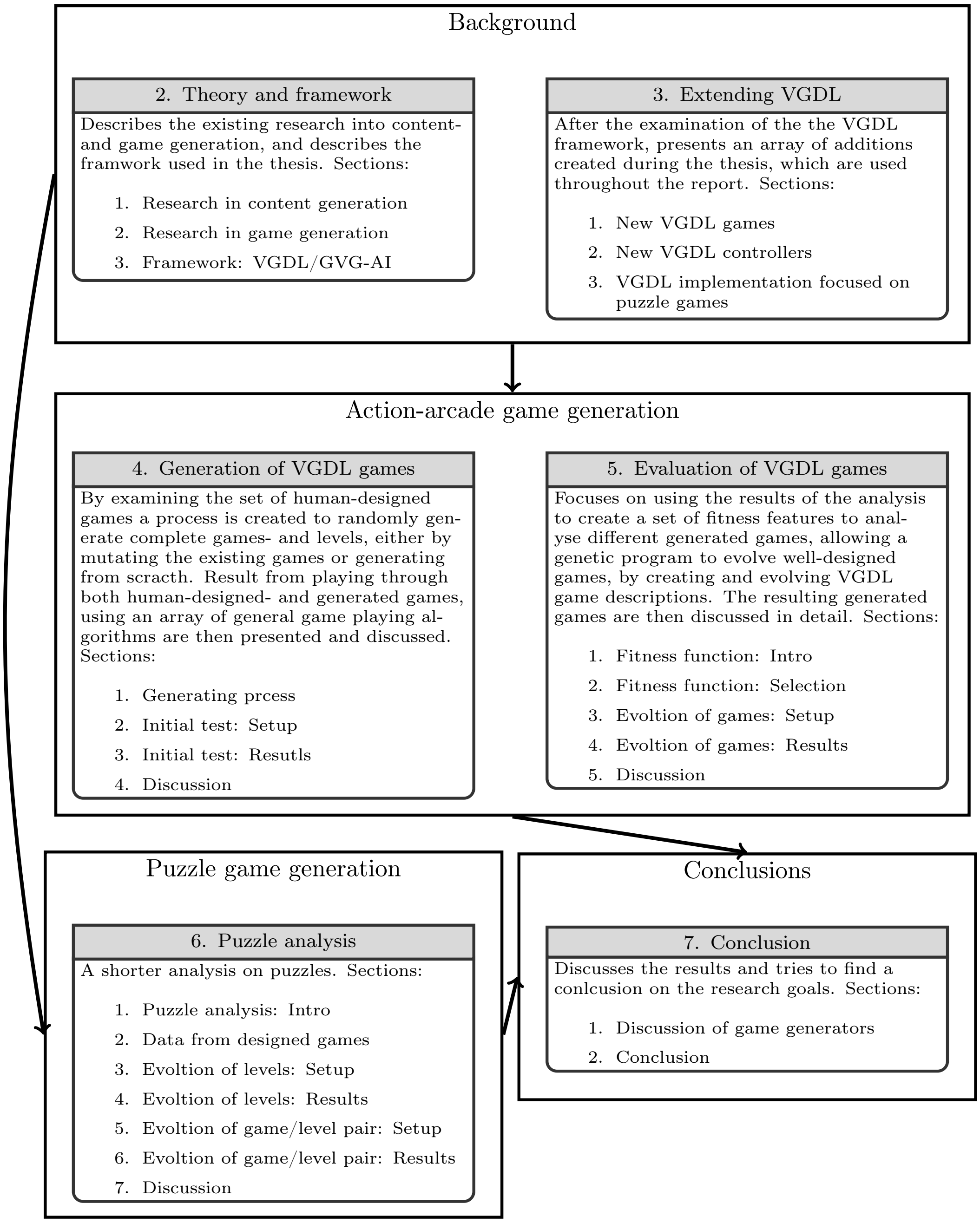
As mentioned above, I assume that the a subset of the human-designed VGDL games described (Section 2.3, 3.1) – the ones which are interpretations of commercial games – are of high quality (i.e. enjoyable for human players). This is not always clear when actually playing the games in the framework used, since they mostly need crucial element like, audio, proper graphic assets- and effects and an input mapping scheme fitting to the game.

We additionally assume that completely randomly generated games, using randomly generated levels in the VGDL language (without sorting/searching using a fitness function) produce games with an extremely (almost negligible) low chance of being human-enjoyable. This assumption is mostly made from initial tests into game generation using the VGDL language.

1.4 How to read this thesis

In the following chapters I will define the problem in more detail, explain the existing research and framework used, present the results of fulfilling the research goal and finally the ultimate products of the analysis of this thesis: Generated games.

The figure below shows the structure of the thesis, explaining the content and reasoning behind each chapter and how it fits in to the larger picture.



Background

2. Theory and framework

Describes the existing research into content- and game generation, and describes the framwork used in the thesis. Sections:

1. Research in content generation
2. Research in game generation
3. Framework: VGDL/GVG-AI

3. Extending VGDL

After the examination of the the VGDL framework, presents an array of additions created during the thesis, which are used throughout the report. Sections:

1. New VGDL games
2. New VGDL controllers
3. VGDL implementation focused on puzzle games

Action-arcade game generation

4. Generation of VGDL games

By examining the set of human-designed games a process is created to randomly generate complete games- and levels, either by mutating the existing games or generating from scracth. Result from playing through both human-designed- and generated games, using an array of general game playing algorithms are then presented and discussed. Sections:

1. Generating prcess
2. Initial test: Setup
3. Initial test: Resutls
4. Discussion

5. Evaluation of VGDL games

Focuses on using the results of the analysis to create a set of fitness features to analyse different generated games, allowing a genetic program to evolve well-designed games, by creating and evolving VGDL game descriptions. The resulting generated games are then discussed in detail. Sections:

1. Fitness function: Intro
2. Fitness function: Selection
3. Evoltion of games: Setup
4. Evoltion of games: Results
5. Discussion

Puzzle game generation

6. Puzzle analysis

A shorter analysis on puzzles. Sections:

1. Puzzle analysis: Intro
2. Data from designed games
3. Evoltion of levels: Setup
4. Evoltion of levels: Results
5. Evoltion of game/level pair: Setup
6. Evoltion of game/level pair: Results
7. Discussion

Conclusions

7. Conclusion

Discusses the results and tries to find a conlcusion on the research goals. Sections:

1. Discussion of game generators
2. Conclusion

1.4.1 Terms

This project is focused a games of two restricted genres and types – these restrictions mostly stems from the framework used in the work, but allows a more focused analysis:

1) Single-player, 2-dimensional, top-down, action games, in which the player controls a single avatar which can (maximally) be moved in the four cardinal directions, possibly with an "action"-(shoot etc.) button. These games will be simply be referred to as **Action-arcade games**.

And: 2) Single-player, 2-dimensional, top-down, turn-based puzzle games, in which the player controls a single avatar which can (maximally) be moved in the four cardinal directions, possibly with an "action"-(shoot etc.) button., and all events occur as a result of player actions (i.e. no NPC's or falling boulders). This type of games will be referred to as **Puzzle games** in the project.

Chapter 2

Existing research and framework

In this chapter I describe the existing research on the topic of procedural content generation (PCG) for games, and specifically game- and level-generation, which is of relevance to the project. Additionally the framework used for game- and level-generation, and testing is described thoroughly.

2.1 Content generation

Before discussing generation of complete games, the research on procedural content generation (PCG) for games are investigated. The task of PCG is to create elements for specific parts of a game, for instance levels, weapons, environments, textures and sounds. **?** proposes a layered taxonomy to describe PCG of different depths in games, while performing a survey of different ways in which PCG has been implemented in commercial games, noticing that some layers has not been thoroughly explored.

Several different approaches are used in PCG (both in commercial games and in research), however the "search-based" approach (explored by **?**, **?**) dominates the problem. In search-based generation a *fitness function* is used to score generated content by their quality, making it possible to ensure a high quality of the generator, by either simply generating a large amount of content and choosing the best, or using an evolutionary strategy to evolve better content.

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.

Several scientific projects have focused on the topic of level-generation 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 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 **???**.

In this project I 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 **?** or search through evolutionary computation or similar forms of stochastic optimisation **???**. In either case, rule generation can be seen as a particular kind of procedural content generation **?**.

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 **?**. 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 **?**. Other researchers have attempted evaluation functions based on the learnability of the game by an algorithm **?** or an earlier and more primitive version of the characteristic that is explored in this paper, performance profile of a set of algorithms **?**.

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 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. A part of the success of Ludi stem from the fact that levels (boards) were as much in focus, as the game-rules themselves.

2.3 Framework: VGDL and the GVG-AI framework

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. 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 **?**. That language is tailored to describing board games

[1] See **?** for a discussion of game-rule representation choices.

and similar discrete, turn-based games; it is also arguably too verbose and low-level to support search-based game generation. 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 at the same time the complete space of possible games is smaller.

2.3.1 VGDL game description

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 **??** (and implemented by Schaul **?**) 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. Each sprite can move a distance specified by its `speed`-parameter in the four cardinal direction, which can have decimal values causing sprites to be able to move in-between game tiles.

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 (and sometimes, must) be fed to each sprite, configuring for instance the `speed` or how often the sprite takes an action – using the `cooldown`-parameter – and a set of parameters unique to 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, and the `Chaser` sprite class (sprite that move towards a specified other sprite each move) it is necessary to define which object to actually chase. Another interesting sprite types is the `Resource`-class, which, in combination with interaction-rules, can be "collected" by different other sprites and several effects can occur (again defined in the interaction-rules), like "if avatar has collected more than 10 coins, he/she is killed when touching wall sprites". Sprites can additionally be designed in a tree structure, where multiple sprites have the same parent sprite, making descriptions for interaction- and termination rules (described below) less verbose.

InteractionSet

Each line of the `InteractionSet` specifies a pair of sprites and an effect, defining the resulting effect of the two sprites interacting with each other (i.e. colliding with each other). Possible interaction effects include `stepBack`, causing the first sprite specified to be pushed back to its last position, `bounceForward`, causing the sprite to be pushed one tile forward, and `killSprite`, which simply kills the first sprite defined on collision. Many interaction effects additionally have parameters which must be set for the rules to function, for instance for the rule `transformTo` (which converts the first sprite defined from one type to another), it must be defined which sprite to transform to. All interaction effects also have a possibility of changing the players score on collisions.

TerminationSet

The termination set 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, for instance the `SpriteCounter` class can cause the game to be won or lost, when there exists 0 of a certain sprite (when all of the sprites are killed). Almost all of the VGDL games analysed or created in this thesis contain two termination rules: A win- and a lose- termination.

LevelMapping

The job of the `LevelMapping` (see figure 5.3) 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
        forestDense > SpawnPoint stype=log prob=0.4  cooldown=10 img=forest
        forestSparse > SpawnPoint stype=log prob=0.1  cooldown=5 img=forest
        structure > Immovable
            water > color=BLUE img=water
            goal  > Door color=GREEN img=goal
        log     > Missile   orientation=LEFT  speed=0.1 color=BROWN img=log
        safety > Resource  limit=2 color=BROWN img=mana
        truck  > img=truck
            rightTruck  > Missile   orientation=RIGHT
                fastRtruck  > speed=0.2  color=ORANGE
                slowRtruck  > speed=0.1  color=RED
            leftTruck  > Missile   orientation=LEFT
                fastLtruck  > speed=0.2  color=ORANGE
                slowLtruck  > speed=0.1  color=RED
        # defining 'wall' last, makes the walls show on top of all other sprites
        wall > Immovable color=BLACK img=wall

    InteractionSet
        goal avatar  > killSprite scoreChange=1
        avatar log   > changeResource resource=safety value=2
        avatar log   > pullWithIt   # note how one collision can have multiple effects
        avatar wall  > stepBack
        avatar water > killIfHasLess  resource=safety limit=0
        avatar water > changeResource resource=safety value=-1
        avatar truck > killSprite scoreChange=-2
        log    EOS   > killSprite
        truck  EOS   > wrapAround

    TerminationSet
        SpriteCounter stype=goal   limit=0 win=True
        SpriteCounter stype=avatar limit=0 win=False

    LevelMapping
        G > goal
        0 > water
        1 > forestDense water       # note how a single character can spawn multiple sprites
        2 > forestDense wall log
        3 > forestSparse water       # note how a single character can spawn multiple sprites
        4 > forestSparse wall log
        - > slowRtruck
        x > fastRtruck
        _ > slowLtruck
        X > fastLtruck
        = > log water
        B > avatar log
```

Figure 2.1: Example of a VGDL game description: An interpretation of the classic arcade game *Frogger*

Level description

Besides the four sets used to describe a game in VGDL, to actually play a game a level file is additionally needed, defining the initial positions of a set of sprites using the `SpriteSet` combined with the `LevelMapping`, in which each textual character defines which sprites to appear in which tile-, while spaces defines empty tiles of the game.

```
wwwwwwwwwwwwwwwwwwwwwwwwwwwww
w             wGw           w
w00==000000===0000======000=2
w0000====0000000000=====00012
w00===000===000=====0000===02
www    ww    www     www  wwwww
w    ———     ——    –   ———    w
w–         xxx        xxx      xx  w
w  –     ——        –     ——— ——   w
w           A                  w
wwwwwwwwwwwwwwwwwwwwwwwwwwwww
```

Figure 2.2: Example of a VGDL level description: A level for the interpretation of *Frogger*

2.3.2 The GVG-AI framework: AI controllers and testing

The GVG-AI framework is a testbed for testing general game-playing 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; the position of each sprite, and their "types" (e.g. portal, immovable, resource), passed to a controller when it is asked to select a move. However these states can importantly 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 sample controllers

A series of general game playing controllers is additionally available in the framework, using a series of different approaches to find select an action on each clock tick. Two of these controllers were chosen to be used in this project to analyse- and evolve new games. Below is a short description of these controllers :

MCTS A simple "Vanilla" MCTS approach using UCT, with a specified static .

OneStep-Heuristic Heuristically evaluates the states reachable through one-step lookahead. The heuristic takes into account the locations of NPCs and certain other objects.

GVG-AI example games

The framework additionally contains 20 hand-designed games, which partly consist of interpretations of classic video games (e.g. Boulderdash, Frogger, Missile Command and Pacman), while some are original creations by the General Video-Game AI Competition's organisers. Note however, that the interpretations of class games all have different stages of simplifications to them, causing several of the games' features to be missing – for instance, the player cannot push boulders in the VGDL version of Boulderdash and they do not "roll" as in the original, and the ghosts in Pacman behave in a much more simplified fashion than in the original. A short summary of each of the example games can be seen in Appendix A

The games are almost all (except for the game *Sokoban*) of the *action-arcade* genre, 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). Figure 5.2 shows the VGDL description of the game *Missile Command*, and 5.3 a level description from the same game.

Restrictions of VGDL and the GVG-AI framework

The current state of VGDL, using the implementation of the GVG-AI framework, the games that can be described are rather restricted to certain genres- and types, without great extensions. For instance, the lack of possibility to traverse different levels in the same play through 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.

As mentioned in the introduction, this project is only focused on the two main type of genres describable in VGDL: Action-arcade games and turn-based puzzle games.

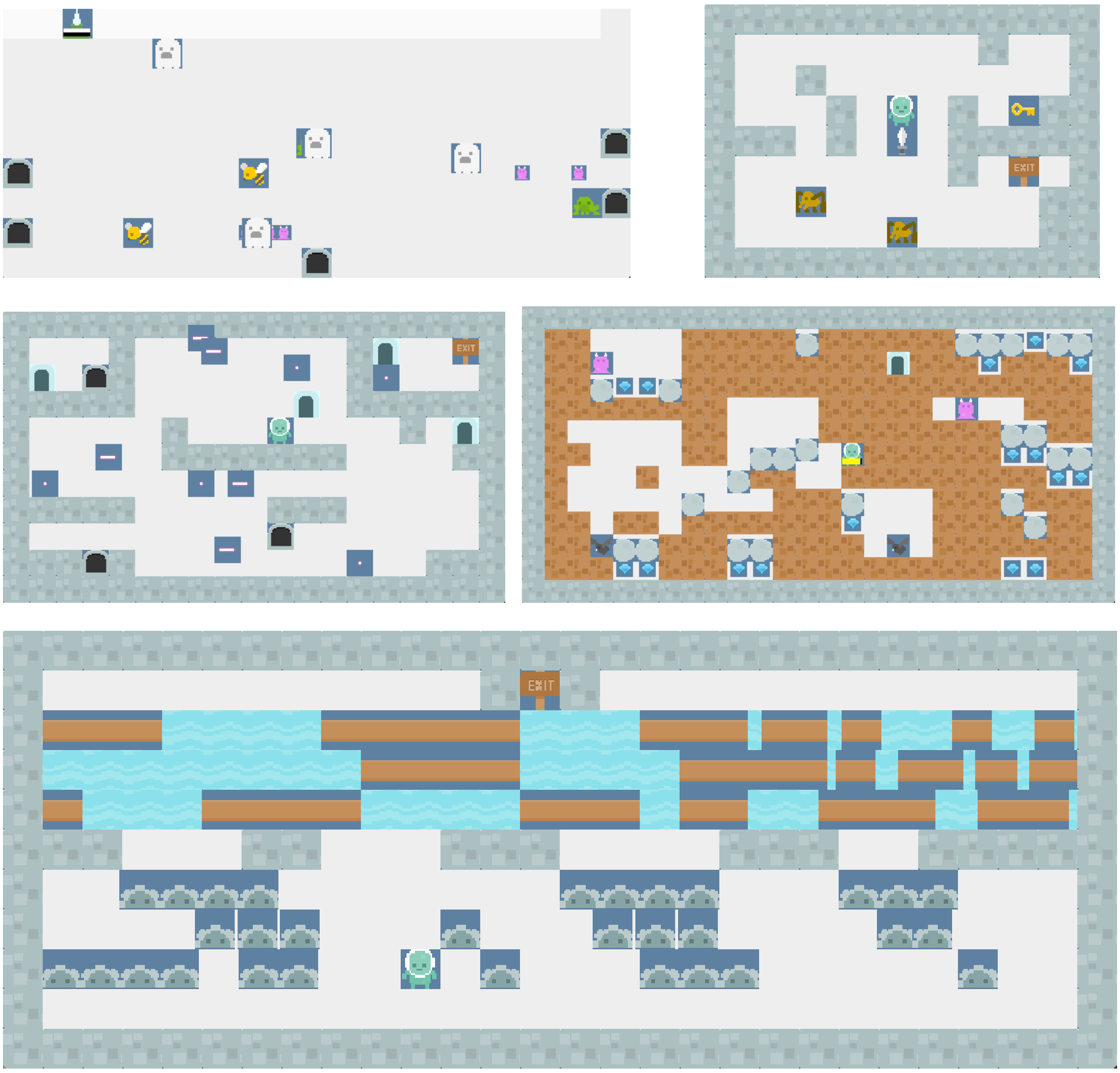


Figure 2.3: The visual representation of a few of the VGDL example games which are interpretations of different (successful) commercial games. From top-left: *Seaquest*, *Zelda*, *Portals*, *Boulderdash* and *Frogs* (*Frogger*).

I 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. As a results, certain sprites- and interaction classes are restricted to the action-arcade games (e.g. NPC- and Missile sprites), making the space of possible *puzzle games* slightly smaller.

Chapter 3

Extending VGDL

This chapter describes a series of extensions I constructed for VGDL and the GVG-AI framework, making a more in-depth analysis possible. I created a series of VGDL games, AI controllers and a implemented a new, simplified framework to play through *puzzle games* with less time and memory use.

3.1 Writing new VGDL games

To increase the size of the set of designed games, to allow for a more precise analysis of what makes a good game, I created 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 are almost all of *action-arcade* genre.

3.1.1 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 types/genres, and so only a limited number of games can be translated without severely changing the game-play.

The games I created, and described below are in general "exact" copies of the originals - 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 points (like fruits in Pac-Man), infrequently appearing enemies (UFOs in Space Invaders) or features which only appear in later levels of the games.

3.1.2 Results

A series of fourteen commercial games was re-created in VGDL to be used in future tests. Figure 3.2 shows one of the games translated to VGDL. Minor changes to the GVG-AI framework was made for a small number of the games, to be able to correctly copy the games' core gameplay features. The original game levels were additionally translated into VGDL level description, with five levels being created for each game. For some of games the levels could not be completely copied, for instance because the levels were too large in size (*Bolo Adventures*), or because a lack of game features implemented in the clones (*Chip's Challenge*), and the resulting levels are therefore simplifications of the originals..

Action-arcade games

A set of four Atari -arcade and -2600 games were found to be suitable for interpretation; *Solar Fox*, *Crackpots*, *Centipede* and *Astrosmash*, and a VGDL game description was written for each. A description of each of the games and their interpretations can be seen in Appendix B

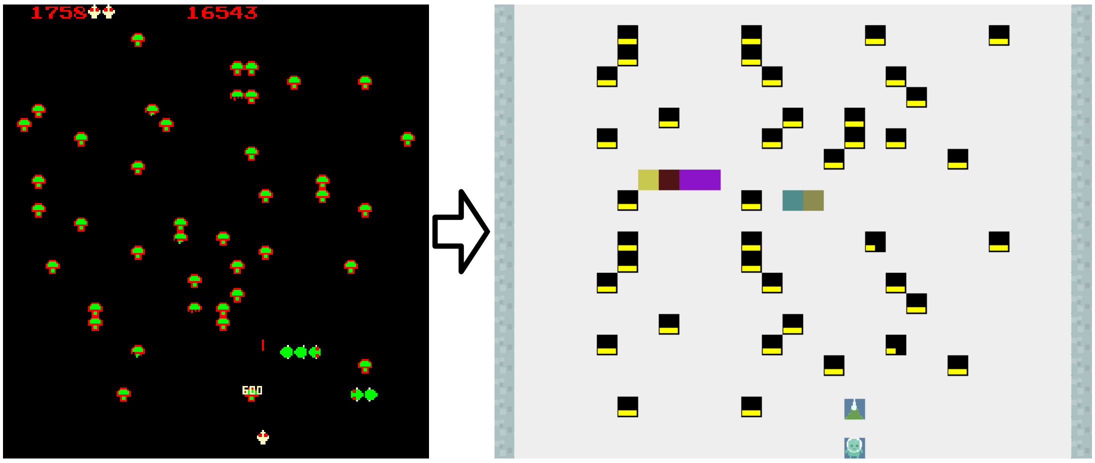


Figure 3.1: Interpretation of the classic arcade game *Centipede* [?]

Puzzle games

A set of puzzle games from several different platform, all featuring a single avatar, was found to be suitable to be described in VGDL. The games can mostly be described as *Sokoban*-clones with different spins on the gameplay. Some of the games have action- gameplay aspects, with enemies hindering the progression of the player, but as per the defintion of *puzzle games* used in the thesis these parts were removed only leaving the puzzles of games.

The games interpreted were *Sokoban*, *Bait*, *Bombuzal*, *Bolo Adventures*, *Zen Puzzle*, *The Citadel*, *Brainman*, *Chip's Challenge*, *Modality* and *Painter*.

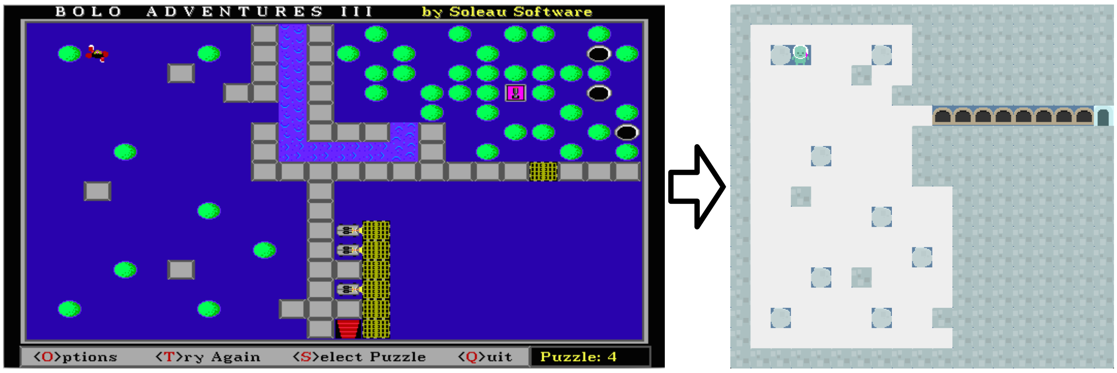


Figure 3.2: Interpretation of both the game and part of a level, from the DOS game Bolo Adventures III

3.2 Creating new VGDL controllers

To be able to probe VGDL games in more detail, I created a series of new AI controllers with various approaches to finding a results. In total four new controllers was generated: Three to play the action-arcade style games, and one only focused on solving puzzle games.

3.2.1 Action-arcade controllers

Three controllers of an increasing degree of cleverness was introduced: *One-step* (least clever), *Deep-search* and *Explorer* (most clever). The controllers greatly differ in their strategy in simulating the forward model (accessible when running the GVG-AI framework) using different actions, and different series of actions.

One-step

The One-step AI only attempt to advance the forward model once, for each allowed action in the game. The score and win/lose state of the resulting game states are then considered, and the action with the resuling best state is chosen – and a random action is chosen when no state is better than others. The controllers approach can be seen below:

Algorithm 1: One-step algorithm

```
for action : PossibleActions do
    newGameState = gameState.copy().advance(action)
    value[action] = value(newGameState)
return action leading to highest value, or random action if values are equal
```

Deep-search

This controller starts out in a similar fashion as the One-step, by expanding using all possible actions by copying the initial game-state. These game-states are then simulated further upon by advancing each game state a single time, with a random action, without copying the state (a rather costly procedure). This expansion is continued until the controller runs out of time, and a value for each action is calculated by considering the score and win/lose-value for each state that can be achieved from each of the initial states.

Algorithm 2: Deep-search algorithm

```
queue ← initial gameState
while has time left do
    gameState = queue.poll()
    if gameState == initial gameState then
        for action : PossibleActions do
            queue ← gameState.copy().advance(action)
    else
        gameState.advance(random action)
        d ← depth of search //amount actions performed
        value[action] + = value(newGameState) $*\text{PossibleActions.length}^{d}$
return action leading to highest value, or random action if values are equal
```

Explorer

The explorer controller was designed specifically to play the arcade-style games of the GVG-AI framework, and utilizes a series of methods to strengthen its decisions. Unlike the other controllers which utilise open-loop searches, it stores information about visited tiles and prefers visiting unvisited locations. Overall the controller uses three different arrays to choose an action: One considering how probable death is from each action, one examining the score that can be achieved from the actions, and one only looking at the boringness. If the *death*-array have too high values, that array is used to take the decision (the action leading to the lowest value is chosen), otherwise the best action from the score-array is used. If the values of the *score*-array happen to be too similar, the *boringness*-array is lastly used instead.

The controller also addresses a common element of the VGDL example games, randomness. The controller gains an advantage in many of the games by simulating the results of actions repeatedly, before deciding the best move.

The Explorer was proven to be of a decent quality by getting a 1st / 5th place in the GVG-AI's competition between controllers [1].

[1] The controller (nicknamed *thorbjrn_ other*) has of now (28/02-2015) a 1st place in the "training-set" (first 10 games of the GVG-AI framework), and 5th in the "validation-set" (the second 10 games).

3.2.2 Puzzle controller

The general game-playing algorithms from the GVG-AI competition and the ones described above, 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. To analyse an arbitrary puzzle game and be able to find the fastest solution, a breadth-first search algorithm was implemented.

Puzzle solver

The breadth-first search algorithm use the fact that most of the puzzle games contain wall-sprites which always push back the player, and so the controllers never try to move into a wall – this is achieved by storing the positions of every wall-sprite at the start of the game. Also, for each path a signature of the game state is calculated and stored, built up from the position and types of each (non-wall) sprite appearing in the given state. A path is cut off if the calculated game state is the same as has appeared before.

The controller was additionally geared up with a two different approaches to memory handling: 1) Storing the actual game state for nodes in the search queue, and 2) only storing the game state signature and path, making it necessary to re-create the game state from the initial state of the game, whenever an element is polled from the queue.

3.3 SimpleVGDL

Since the GVG-AI framework has some functions and properties which are not interesting for some parts of this work, I 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).

The puzzle solver controller was in addition adapted to run using FastVGDL. Below I present the results of a small test of running a increasingly difficult game/level-pairs using the puzzle solver, in both the GVG-AI framework and using FastVGDL, both with- and without the "low-memory" option.

framework	*time*	*mem.*
GVG-AI	962 ms	32 MB
FastVGDL	535 ms	6 MB
FastVGDL_LowMem	3642 ms	5 MB

Figure 3.3: Results from playing through the game *(Real) Sokoban*, level 4 (of 5) using the puzzle solver controller

framework	*time*	*mem.*
GVG-AI	9420 ms	270 MB
FastVGDL	4354 ms	59 MB
FastVGDL_LowMem	41691 ms	58 MB

Figure 3.4: Results from playing through the game *(Real) Sokoban*, level 2 (of 5) using the puzzle solver controller

framework	*time*	*mem.*
GVG-AI	314210 ms	1892 MB
FastVGDL	142547 ms	638 MB
FastVGDL_LowMem	163385 ms	386 MB

Figure 3.5: Results from playing through the game *Real Sokoban*, level 1 (of 5) using the puzzle solver controller

The results show a clear difference in time and space usage between the GVG-AI framework and FastVGDL, with FastVGDL being about twice as fast. There is a huge time difference between using the low-memory approach or not. The space usage is actually quite different between the controllers when the program running, because the main controller store a large amount of data in the queue, but the memory used to store each game state signature are the same, causing the almost the same amount of memory to be used up at the end of the process (the values shown in the figure). As can be see from the last test (Figure 3.5) the main advantage of the approach is that it can actually finish certain game/level-pairs, where the others reach a `OutOfMemoryError Exception`.

Chapter 4

Automatic generation of VGDL descriptions

This chapter describes the setup- and methodology used in generating new games (i.e. game descriptions). Additonally an initial play test of set of both human-designed and generated games, using an array of different AI-controllers explained in the previous chapters.

This chapter is only focused on *action-arcade* games - analysis and generation of *puzzle games* will be presented and discussed in Chapter 6.

Selected designed arcade-action games

A subset of the human-designed games mentioned in Section 2.3 and 3.1 were selected for the following analyses games, and for creating new games by mutation.

After my addition of VGDL game description to GVG-AI framework there is a total 34 human designed games which could potentially be used as a base-line for quality in games. However, using the definitions from the introduction, only 23 of the games are of the *arcade-action* genre explored in this section. Additionally, as mentioned in Section 2.3.1 a few of the example games from the GVG-AI competition are original creations, which cannot equally be assumed to be human-enjoyable.

With these limitation, thirteen interpretations of existing games are left and used as a baseline for testing and game generation: Aliens, Boulderdash, Frogs, Missile Command, Zelda, DigDug, Pacman, Seaquest, Eggomania, Solar Fox, Crackpots, Astrosmash and Centipede.

Besides the additions and changes to the GVG-AI framework to make new games possible, a slight change was done to remove the possibility of players scoring lower than 0, which is the norm in the original games mentioned, and importunely makes comparing scores more straightforward).

4.1 Generating processes

Using the GVG-AI framework I constructed processes for generating large sets of new game description. I used two different approaches in generating new game description in VGDL: Mutating existing (designed) game descriptions-, and randomly generating new ones (from scratch). In both approaches several constraints were set upon the generation process, either to prevent crashes or limit parameters to values known to be good in the designed games (causing a slight decrease in the space of potential generated games).

4.1.1 Mutation of example games

A process was built for parsing existing VGDL game descriptions, mutating certain sprites- or rules, and returning new, valid game descriptions,. The set of games described at the beginning of this chapter, were chosen to be used as a basis for mutating new game descriptions, analysed in this section.

Generating approach

It is not immediately apparent which approach to use in mutating the different elements of each VGDL game description. Since sprite- and rules are constructed from a combination of a class and a series of parameters specific to the class, it is necessary to change the parameters if the class is changed, but the parameters of an existing class can freely be mutated.

A process was built for mutating each of the different parts of game descriptions, i.e. changing the sprites available of the `SpriteSet`, the interactions rules (`InteractionSet`) and the termination rules (`TerminationSet`). The process additionally allowed for changing the amount of rules, i.e. generating new rules or removing existing ones.

Since the sprites and rules of each game description is described by both a class and a series of parameters, several partly subjective decisions was made on defining probabilities of different parameters' values to try reach "realistic" values. For instance it was decided that a sprite being mutated or generated only has a 25% chance of using the `cooldown` parameter (making sprites have pauses between acting), and that the parameter have a random value between 1 and 10. The values and probabilities used were mostly decided by examining the set of designed games descriptions. Several constraints were also used to avoid game with non-valid descriptions (which can cause crashes in the GVG-AI framework), for instance by ensuring that avatar-sprites cannot be spawned and sprites cannot transform to an avatar, but an avatar sprite can still transform to another type of avatar sprite.

In this work I decide to only mutate interaction-rules from the `InteractionSet` of designed VGDL games, and simply change every part (classes, parameters and references) of a random amount of rules, with each rule having a 25% change of being changed for each game to mutate. In addition to simplifying the mutation process, and having games more with a higher chance of being playable this allows us to use the original designed games' levels for testing.

Figure 4.1 shows a typical outcome of mutating the texttttInteractionSet of a VGDL game: Some of the new generated rules might never be triggered or have a neglible effect, while overwriting some of existing rules.

```
%zelda
InteractionSet
  movable wall  > stepBack
  nokey goal    > stepBack
  goal withkey  > killSprite scoreChange=1
  enemy sword > killSprite scoreChange=2
  avatar enemy > killSprite scoreChange=-1
  key  avatar   > killSprite scoreChange=1
  nokey key     > transformTo stype=withkey

%zelda_mutation_3
InteractionSet
  movable wall > stepBack
  monsterQuick monsterQuick > attractGaze
  sword sword > killSprite
  enemy sword > killSprite scoreChange=2
  avatar enemy > killSprite scoreChange=-1
  key avatar > killSprite scoreChange=1
  nokey key > transformTo stype=withkey
```

Figure 4.1: Mutation of the `InteractionSet` of the VGDL interpretation of the game *Zelda*

4.1.2 Random game generation

A similar process as mentioned above was used to randomly put game descriptions together creating new games completely from scratch, constructing the textual lines for the four parts of a VGDL description: Generating an array of sprites (for the SpriteSet), interaction-rules (InteractionSet), termination-rules (TerminationSet) and level mappings (LevelMapping).

Several partly subjective choices were again made for the range for the amount of different sprites, and interaction- and termination rules. Even though some of the original games contain a larger set of elements, I constricted the ranges to sprites: $3 - 8$, interactions: $3 - 10$ and terminations: 2 (a "win-" and a "lose"-termination), to reduce the space of generated games.

When generating descriptions, I used similar constraints to those mentioned in section **??**, partly to avoid generating descriptions with invalid elements, and partly to increase the proportion of interesting outcomes. As for the mutated games, some partly subjective decisions were made on defining the possible amount of sprites and rules, the proportion of sprites that has a level mapping, and when generating new sprite definitions.

To simplify the sprite creation process, no parent-child structure was used in the `SpriteSet`. However, the goal of the parent-child structure is only to make rule definitions less verbose, and does not make actual new game features possible.

All sprites were given a random sprite image, while the avatar-sprite was given the same sprite for each game, to make the game more easy to understand when visualising the games (and actually play through them). Figure 4.2 shows an example of a game generated using the process.

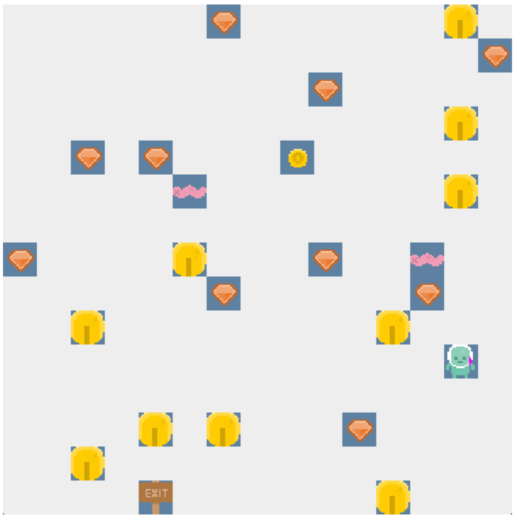


Figure 4.2: Screenshot of a playthrough in a randomly generated VGDL game

```
BasicGame
  SpriteSet
    avatar > HorizontalAvatar img=avatar cooldown=2
    gen1 > Resource limit=10 value=5 img=fire
    gen2 > RandomNPC img=wall
    gen3 > Resource limit=4 singleton=TRUE value=1 img=explosion
    gen4 > OrientedFlicker limit=23 orientation=RIGHT img=water
    gen5 > Spreader limit=14 stype=gen1 img=spaceship
  InteractionSet
    avatar gen1 > killIfFromAbove
    avatar gen2 > killIfOtherHasMore limit=4 resource=gen1
    gen5 gen1 > cloneSprite
    gen4 gen5 > pullWithIt
    gen3 avatar > changeResource value=1 resource=gen1
    avatar gen1 > undoAll
    avatar gen4 > stepBack
    avatar gen5 > stepBack
  LevelMapping
    $ > gen1
    % > gen2
    & > gen3
    ’ > gen4
    ( > gen5
  TerminationSet
    SpriteCounter limit=0 stype=avatar win=TRUE
    MultiSpriteCounter limit=0 stype1=avatar stype2=gen3 win=FALSE
```

Figure 4.3: A randomly generated VGDL game description

```
           %
   %%
     %’
           %
        %
 %
          %

$          &

       % &
 %

   (  %   A%
      %   %
```

Figure 4.4: A random generated level description, generated for the VGDL game above

Level generation

As mentioned in Section 2.2.1 the problem of generating a game is more often than not intimately linked to a generation of levels. I.e. I could end up with generating game descriptions of high quality games, but if the level descriptions does not fit to the game-play (by being too trivial, or too hard for instance) the games might not found if searching through a set of randomly generated games.

A simple level generator were constructed with the simple goal of making the randomly generated games playable. The generator designates a level built using a given size, and inserts sprite mappings in different positions which takes all the level mappings from a game description and put a random (but small) amount of each sprite defined, in random location(s).

Figure 4.3 shows a resulting game description-, and Figure 4.4 the level, of using the process described.

4.2 Initial test: Experimental setup

Using the generation presented in the previous section, I performed an initial test of playing through both the designed- and generated games using different general game playing algorithms (controllers), and discuss different ways to analyse the resulting data.

Controllers

Six different controllers were used to test the games. The controllers use an array of varying approaches, which can be described as a different degree of intelligence.

One of the more "intelligent" controllers used were included in the GVG-AI framework (*SampleMCTS*), while the remaining were implemented for this work (*Explorer*, *Deep-Search* and *OneStep*). In addition two extremely simple controllers were created to be used as a basis for "bad" play: 1) A *Random* controller, which always returns a random action, and 2) a controller which never acts at all and always return an "NIL"-action, called *Do Nothing*.

Generated games

A set of 130 VGDL games were generated by mutating each of the 13 designed games ten times, using the approach described in the previous section.

400 randomly generated games, each accompanied with a single level, were additionally generated and used in testing.

Testing and result analysis

The six controllers were used to play through the set of human-designed-, mutated and randomly generated games. Because of CPU budget limitations each game was played through ten times, with a maximum amount clock ticks of 2000 and 50 ms allowed for each tick. In addition, a function was implemented to stop playing through games if a clock tick ever takes more than 50 ms, which happens often when generated games have rules generating too many sprites, which can end up taking an extreme amount of time to play through if not discarded quickly. For each playthrough of games, only the most essential data from the GVG-AI framework was retrieved: The *score*, the *win-lose value*, the amount of *clock ticks* used and a list of actions performed.

A process to analyse the data was built, to calculate averages, standard deviations, max-, minimum and other statistics of each of the values, for each controller, for both each individual game and over a whole set of games. To more accurately compare the *score* for the different controllers when playing across a range of different games I additionally calculate a normalised score using a max-min normalisation ($\frac{score-min}{max-min}$), using the maximum/minimum score for each play through, across the controllers. The entropy of actions performed was also calculated, again with an average over all play throughs- or games.

4.3 Initial test: Results

Using the setup described above averages of all play-throughs for each controller, is presented and the results compared with each other.

In the results below, a series of data from the generated set of games were removed due to being either too difficult to analyse (because a controller was disqualified), or for having certain values which I deem impossible for a quality game. Namely I remove games by three different considerations: 1) When a controller was disqualified (too many sprites- and/or interactions happening), 2) when all controllers get the same score (and win-rate) for all playthroughs, and 3) where the game always end in less than 50 clock ticks (which is the maximum lifetime of generated `Flicker`-sprites).

Designed games

The six controllers were set to play through each of the human-designed VGDL games, with the constraints mentioned above.

I present the averages across each single game(can be seen in Appendix C) and averages across all of the games, seen in Figure 4.5. The distributions show that more intelligent controllers tend to have more success, with both a higher win-rate and a significantly higher average score. The *clock-ticks* and *action entropy* does not show any similar distribution for the collection of all games. Examining the results from each game it is noticeable that the profiles for these values (*clock-ticks* and *entropy*) are very different from game to game.

It should be noted that the uncertainties for average score, clock-ticks and action entropy assume a normal distribution of the data which in general is not true, and so these values should be taken with a grain of salt.

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	42.94 (5.73)	0.80 (0.02)	0.45 (0.02)	514.66 (12.40)	0.89 (0.01)
MCTS	23.14 (4.07)	0.49 (0.02)	0.25 (0.02)	629.58 (11.30)	0.90 (0.01)
GA	11.98 (2.41)	0.40 (0.02)	0.15 (0.02)	590.34 (12.93)	0.71 (0.02)
Onestep-S	14.48 (1.29)	0.41 (0.02)	0.08 (0.01)	477.72 (15.09)	0.88 (0.02)
Onestep-H	3.73 (0.46)	0.24 (0.01)	0.11 (0.01)	569.27 (14.91)	0.19 (0.02)
Random	7.52 (0.80)	0.25 (0.01)	0.06 (0.01)	395.88 (14.97)	0.90 (0.01)
Do Nothing	0.39 (0.19)	0.13 (0.01)	0.06 (0.01)	601.72 (14.07)	0 (0)

Figure 4.5: Averaged results across all the XX designed games

Mutated games

After removing games by the requirements mentioned above, I were left with a toal of 146 games in which I had useful data. The results of playing through these games with the controllers can be seen in Figure 6.1. The scores have higher means and standard deviations, indicating outliers in the data. The ordering of the *normalised score mean* and *win-rate*, however, shows a similar pattern as for the example games, with Explorer again excelling.

It should be noted here that the differences between the extreme values of *score* and *win-rate* are smaller than for the designed set of games.

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	392.08 (106.62)	0.84 (0.01)	0.31 (0.01)	607.87 (4.20)	0.84 (0.01)
MCTS	140.49 (18.17)	0.48 (0.01)	0.15 (0.01)	689.05 (3.40)	0.90 (0.00)
GA	88.96 (12.53)	0.43 (0.01)	0.13 (0.01)	671.44 (3.83)	0.67 (0.01)
Onestep-S	128.82 (11.70)	0.44 (0.01)	0.10 (0.01)	565.77 (4.79)	0.84 (0.01)
Onestep-H	82.67 (12.63)	0.26 (0.01)	0.19 (0.01)	568.66 (4.89)	0.20 (0.01)
Random	69.70 (11.04)	0.26 (0.00)	0.11 (0.01)	470.10 (5.19)	0.90 (0.00)
Do Nothing	81.55 (15.65)	0.18 (0.01)	0.10 (0.00)	642.16 (4.16)	0 (0)

Figure 4.6: Averaged results across the 146 mutated games

Generated games

Figure 4.7 shows results for the remaining 65 randomly generated games (of 400), with problematic games removed according to the same criteria as in the previous section.

First of all, the *score* have much more extreme values 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 used to compare scores across the different controllers. The *normalised score means* and *win-rates* both have values that are more closely clustered togther, than in the previous game sets. For this set of games the *action entropy* of the intelligent controllers has also fallen quite a bit (relative to the Random-controller), indicating that more of the games have simple solutions to win or increase the score

It should be noted that the description generation process can often end up making games be automatically won (for instance, the goal of a game could be to destroy all coins, but all coins are of the `Flicker`-class and disappear after a certain amount of time), which causes the relative high *win-rate* for the Do Nothing-controller.

controller	*ave. score*	*norm. score*	*win-rate*	*ave ticks*	*ave. act entr.*
Explorer	−18 207.91 (5638.19)	0.62 (0.01)	0.26 (0.01)	592.43 (8.42)	0.72 (0.01)
MCTS	−4035.85 (6422.29)	0.44 (0.01)	0.28 (0.01)	590.21 (8.50)	0.84 (0.01)
GA	−3501.65 (6512.05)	0.44 (0.01)	0.25 (0.01)	607.55 (8.23)	0.58 (0.01)
Onestep-S	−16 680.67 (5745.31)	0.39 (0.01)	0.22 (0.01)	632.94 (7.74)	0.82 (0.01)
Onestep-H	−25 728.97 (4846.43)	0.36 (0.01)	0.20 (0.01)	644.59 (7.63)	0.40 (0.01)
Random	−23 348.24 (4946.64)	0.32 (0.01)	0.21 (0.01)	637.22 (7.69)	0.86 (0.01)
Do Nothing	−3051.34 (6438.95)	0.37 (0.01)	0.18 (0.01)	643.60 (7.70)	0 (0)

Figure 4.7: Averaged results across the 65 generated games

Outcome and graphs

By examining the graphs (Figure 4.8 and Figure 4.11) of the data just discussed, it is clear that there is a strong relationship between the performance profiles of playing with controllers of different types, on different set of games. However it should be noted here that some games from both the mutated- and generated game-set, contains games, which when examined by themselves have similar distributions as the average designed game.

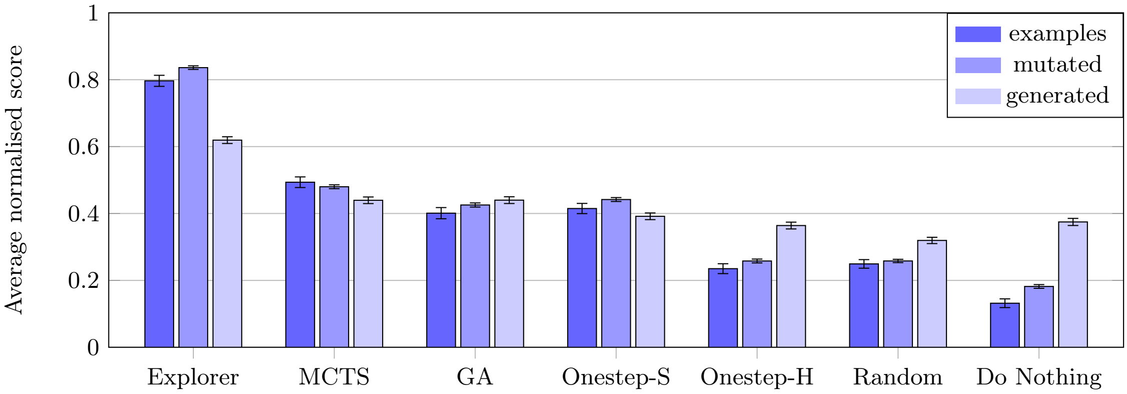


Figure 4.8: Averaged normalised score across all games of the three different set of games: Designed-, mutated- and completely generated games

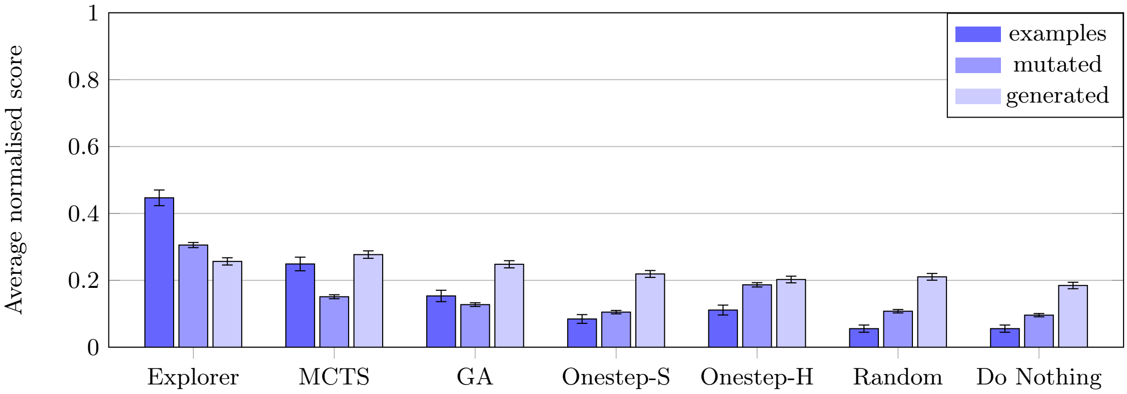


Figure 4.9: Averaged win-rate for the three sets

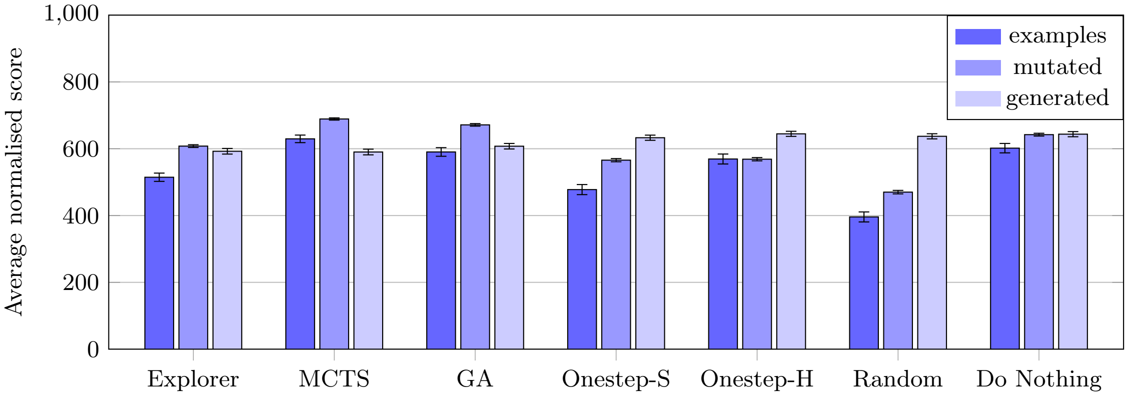


Figure 4.10: Averaged clock tick for the three sets

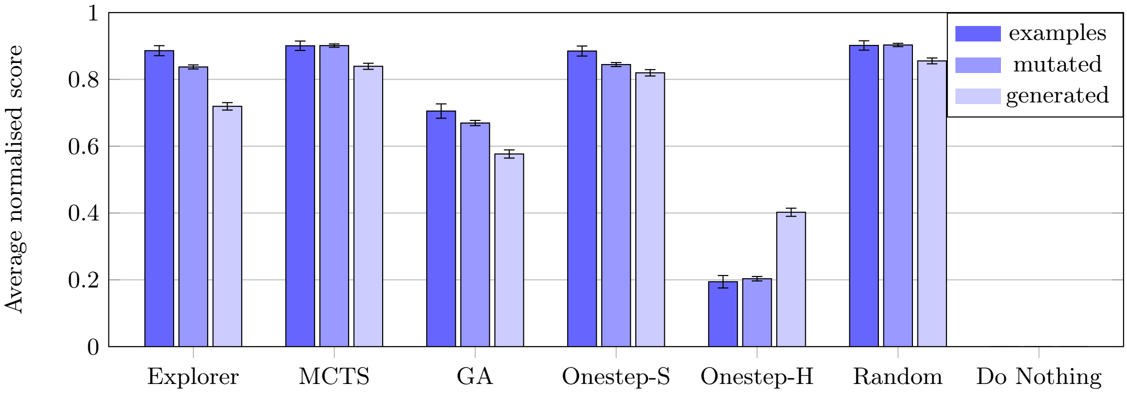


Figure 4.11: Averaged action entropy for the three sets

4.4 Discussion of initial test results

The results display some interesting patterns. Win rates suggest a relationship between intelligent controllers' success and better game design; for better designed games, the relative performance of different types of algorithms differ more. This corroborates the hypothesis that relative algorithm performance profiles can be used to differentiate between games of different quality. In randomly generated games, which arguably tend to be less interesting than the others, smarter controllers (e.g. Explorer and MCTS) do only slightly better than the worse ones (i.e. Random and DoNothing). This is due to a general a lack of consistency between rules generated in this manner. Mutated games, however, derive from a designed game. Therefore, they maintain some characteristics of the original idea, which can improve the VGDL description's gameplay and playability.

While it is possible that random actions can result in good outcomes, this chance is very low, especially when compared to the chance of making informed decisions. In spite of that both Random and DoNothing do fairly well in randomly generated games. The performance of DoNothing emerges as a secondary indicator of (good) design: in human-designed games, DoNothing very rarely wins or even scores.

Human play of generated games

As mentioned in Section **??** there was a series of generated games with performance profile basically indistinguishable from the designed games. I decided on examining many of these games further by studying their game descriptions, and playing the games with visuals enabled, both with AI controllers and human players. (SCREEEEEENSHTOS PL0X!!!) Besides a few mutated games which kept most of the original rules (and therefore were successful games), the tested generated games with well-formed performance profiles were in general too trivial and aimless to be entertaining. This however gives an indication of in important assumption: That I can assume that the generated games, at least from the randomly generated set, are of a overall low quality. I will carefully use this assumption to construct an evaluation function for games, in the next chapter,

Besides the above, a few re-occurring problems was found in the generated games.

- Sprites often leave the level playing field.
- Several of the sprites- and/or rules are never used.
- The game can only be won in the first few (<50) frames.
- Too much random stuff going on.

Chapter 5

Evaluation of games (fitness functions)

The initial tests of the previous chapter shows that there is reason to explore the relationship of AI's perfomance profiles to analyze a given VGDL game, and give some indications of interesting statistical values to analyse.

This chapter focuses on, using the results of the previous chapter and several analysis methods, creating a set of fitness functions to analyse different generated games, allowing a genetic program to evolve well-designed games, by creating and evolving VGDL game descriptions. The resulting generated games are then discussed in detail.

5.1 Fitness function: Introduction

The tests of the previous chapter indicates that the relationship between general game playing controllers performance could potentially be used to determine the quality of games. To make it possible to use the performance relationships, while not overfitting the parameters to the small set of "good" data points (13 human-designed games), it is necessary to find the most important and selective statistical values from playthrough results for use in a fitness function, while trying to exclude statistics that do not signify game quality.

As mentioned in Section **??** the *score* and *win-rate* seems to have the most distinct distributions for the different type of games, with human-designed games all having a signficant difference between e.g. the Explorer- and Random controller, making them interesting to use for an evaluation function. However, the *action entropy* might also be appropriate to use, but the difference in the statistical distributions are much smaller for these values.

Limiting fitness features to score, win-rate and action-entropy, there is still an enormous amount of possible ways to calculate features-values for the fitness function. For instance, I could have a feature for the difference in average score between every single controller but it would result in $\binom{7}{2} = 21$ features for a single statistic. Also, I could use a series of different values simply to compare the score: *minimum*- and *maximum* score, *standard deviation* of score, *median*, *quartiles* or any form of normalised score, and of course I could use combined values for the features, e.g. the *score* increase per *clock tick*.

Additionally it is not straightforward to calculate a feature even with if I decide to compare e.g. *score* or *clock ticks* between Explorer and Random: I could use a simple relative difference formula (e.g. $f = |\frac{a-b}{max(a,b)}|$), and use the result as a feature, but it might make more sense to hand-craft each feature to a specific goal, for instance for the *action entropy* it seems (from the data of the last chapter) designed games cause the intelligent controllers to have a similar value as the Random controller, while the same is not true for the randomly generated games – in the generated games the intelligent controllers often find a simple solution requiring similar moves. A feature could therefore be made to have a value of 1 if the entropies are the close (and going toward 0/-1 when random has a higher entropy), while not rewarding the game for having the intelligent controllers *action entropy* be higher than Random (which the data do not show as a relationship in designed games).

Also, how each fitness feature adds to the complete fitness function is not straightforward. One could use a simple sum of each chosen feature, maybe with a weight constant attached signifying how much each feature adds the overall value ($f = w_1a_1 + ... + w_na_n$), but approach can have the outcome of some features overruling others, unless best possible know weights are know (which they typically are not).

In the following I will try to shed light on which features are the most interesting, by both analysing the data in more detail, and by evolving games using the features and analysing the evolved games.

5.2 Performance relationship analysis

Since the relationships between controllers acting intelligent, and acting random or doing nothing, seems to be a possible indication of game quality, an analysis of the relationship between different controllers using varying values was performed. As indicated in the previous section, choosing how to evaluate game for the fitness function almost certainly have an element of subjectivity. To make an as informed decision as possible I ran a series of test described below, using different features, automatically choosing different set of features, evolving weights for the features, and examining the results using data mining principles.

In the following I repeatedly to use a *relative difference*-function ($f = |\frac{a-b}{max(a,b)}|$) to compare statistical values between controllers performance results, retrieved from playthroughs. Additionally I chose to simply construct the fitness function be a sum of the individual features.

Also, the assumption that the games from the generated- and mutated set simply are of a low quality (discussed in Section **??**), is used throughout the following analysis.

Analysis of possible relationship features

As mentioned in the previous sections, the relationship of score and win-rate between controllers seems to be obvious choices, to adopt as fitness features. Also, the there seems to be a consistent difference in the values between the intelligent controllers (especially Explorer and DeepSearch) and the less intelligent ones: The OneStep, Random and Do Nothing-controllers.

To measure the strength of different possible choices from I performed a small test, counting how often a specific intelligent controller has a higher value than a simple one, and calculating the average relative difference between the two. The results can be seen in Figure 5.1.

Some of the features in the table immediately seems interesting and significant, for instance the score average- and win-rate difference between the intelligent- and the DoNothing controller is consistently higher for the intelligent controllers, while the same is not true for several of the games from the mutated- and randomly generated set. The average clock ticks between, for instance Explorer and OneStep additionally could be interesting to use as an indication of a bad game, since the value is never higher for OneStep in the generated game set. However the actual relative difference between the controllers is tiny, and so the relationship might not be that significant. Also the designed games do not have a clear relationship profile for the clock ticks (as discusses in Section 4.4).

Analysis of feature sets

From the above analysis, a selection of interesting feature values are available, and of these I continuously tested the results of selecting different sets of features (between 2-20) of relative difference values between pairs of controllers.

From the varying sets of feature selections, I first of all calculated and compared the resulting (total) fitness of games from both the designed- and generated set, noticing how each game was scored (while examining high-fitness generated-, and low-fitness designed games further).

In choosing a set of evaluation features it is natural to value each with a weight, making the addition from each feature count in a balanced fashion.

For most of the set of chosen features (when many features were selected) there was no clear indication of which values were more important, and so I continuously used a CMA-ES approach to evolve a set of weights. For this task another problem arise however: Generating a fitness function to score a chosen set of weights. Several different approaches were tried for this task, but all relying on calculating the fitness for each of designed- and randomly generated games, and making the overall fitness for the designed game set higher than for the generated games.

data type	Designed		Mutated		Rnd. gen.	
	count	*diff.*	*count*	*diff.*	*count*	*diff.*
Mean score:Explorer>Onestep-S	13 (of 13)	0.717	78 (of 90)	0.494	43 (of 66)	0.240
Mean score:Explorer>Random	13 (of 13)	0.848	81 (of 90)	0.651	43 (of 66)	0.334
Mean score:Explorer>Do Nothing	13 (of 13)	1.000	82 (of 90)	0.823	45 (of 66)	0.404
Mean score:DeepSearch>Onestep-S	13 (of 13)	0.736	73 (of 90)	0.474	35 (of 66)	0.214
Mean score:DeepSearch>Random	13 (of 13)	0.864	81 (of 90)	0.659	41 (of 66)	0.307
Mean score:DeepSearch>Do Nothing	13 (of 13)	1.000	83 (of 90)	0.821	42 (of 66)	0.375
Winrate:Explorer>Onestep-S	11 (of 13)	0.675	36 (of 90)	0.282	4 (of 66)	0.035
Winrate:Explorer>Random	11 (of 13)	0.756	39 (of 90)	0.327	5 (of 66)	0.033
Winrate:Explorer>Do Nothing	11 (of 13)	0.846	36 (of 90)	0.349	6 (of 66)	0.086
Winrate:DeepSearch>Onestep-S	12 (of 13)	0.762	35 (of 90)	0.299	3 (of 66)	0.029
Winrate:DeepSearch>Random	12 (of 13)	0.846	39 (of 90)	0.350	5 (of 66)	0.026
Winrate:DeepSearch>Do Nothing	12 (of 13)	0.923	35 (of 90)	0.345	6 (of 66)	0.081
Mean tick:Explorer>Onestep-S	8 (of 13)	0.197	39 (of 90)	0.026	0 (of 66)	-0.076
Mean tick:Explorer>Random	9 (of 13)	0.439	56 (of 90)	0.280	1 (of 66)	-0.084
Mean tick:Explorer>Do Nothing	7 (of 13)	0.082	34 (of 90)	0.015	2 (of 66)	-0.072
Mean tick:DeepSearch>Onestep-S	8 (of 13)	0.251	42 (of 90)	0.098	1 (of 66)	-0.083
Mean tick:DeepSearch>Random	10 (of 13)	0.504	59 (of 90)	0.328	2 (of 66)	-0.086
Mean tick:DeepSearch>Do Nothing	6 (of 13)	0.160	35 (of 90)	0.082	2 (of 66)	-0.075

Figure 5.1: Simple analysis of possible fitness values

The construction and evolution did not

Outcome

Because In the end I chose to only use the Explorer, OneStep-, Random- and Do Nothing controller in calucalting fitness.

I decided on a set of 19 features, where 15 are calculated from the relative difference between a pair of controllers, and 4 values, each with a unique calculation. The chosen features were:

3 features comparing score mean between Explorer and OneStep, Random and Do Nothing.

3 features comparing score median between Explorer and OneStep, Random and Do Nothing.

3 features comparing score quartion 1 between Explorer and OneStep, Random and Do Nothing.

3 features comparing score quartion 3 between Explorer and OneStep, Random and Do Nothing.

3 features comparing win-rate between Explorer and OneStep, Random and Do Nothing.

In the end I chose the following set of values to compare: *ave. score*, *min. score*, *max. score* and *win-rate*.

5.3 Evolution of games: Setup

A process for generating evolving new VGDL games were built using the fitness function found in the previous section. Two different approaches were used to create new games: 1) Evolving from an existing, human-designed game description, using human-designed levels, and 2) evolving from randomly generated games, using randomly generated levels.

From the project it should be obvious that the first approach, using existing games and levels, have a much better chance of both achieving a high fitness-value and be an enjoyable game. However, the second approach allows for magnitudes of more options and possibilities, since it is not restricted to any set of sprites- or levels.

The evolution process was stopped when the highest fitness stopped increasing over ten iteration, or when the best games fitness were above 0.98 – a very good fitness score.

The figure below shows the process which was used in both evolution approaches.

HERE SHOULD BE A PICTURE OF THE GENERATION PROCESS

Generate game –> if not player runs out of time –> if not any sprites out of bounds –> get results –> if no disq –> if no low time wins –> if no all squares equal

5.4 Evolution of games: Results

XX games were generated over a duration of XX days,

```
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 5.2: 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.

HERE GOES A GRAPH OF HOW FITNESS INCREASES OVER TIME IN EVOLUTION

5.5 Discussion of results

The resulting games are unfortunately pretty bad.

It would be interesting to retrieve more data from playing through each game. For instance, by examining the proportion of sprites- and rules that have been in use in the game, the amount of sprites that has left the level field, or the amount of objects that have been killed or created.

```
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
wwwwwwwwwwwwwwwwwwwwwwww
```

Figure 5.3: Example of VGDL level description - a level for the implementation of the game Missile Command

5.6 HUUUUSK Designed action-arcade games: Generated levels

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

Chapter 6

Puzzle generation

In this chapter I will explore the possibility of automatically generating puzzle games in the VGDL language.

6.1 Puzzle analysis introduction

To analyse and generate puzzle games more quickly, my implementation of VGDL: FastVGDL, was used to test- and generate levels, instead of the GVG-AI framework.

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).

Games

From the GVG-AI framework (Section 2.3) and my own additions (Section 3.1), there is a total of 10 puzzle games to use in the following analysis.

However, several of the games were deemed unfit for a complete examination, because either the levels were too difficult (Bolo Adventures and Chip's Challenge), that the game had features difficult to solve for a the puzzle-solving AI controllers (Bombuzal), or that the gameplay was too simple (Painter). The games used in the following are Bait, Brainman, The Citadel, Zen Puzzle, Sokoban, Modality

6.2 Data from designed games

To analyse the six designed puzzle games mentioned above, the puzzle solving controller was set to play through all of the levels of the games, finding both the fastest possible solution, and all other possible routes.

Below is a summary of the results:

game	*lvl*	*found sol.*
bait	0	true
bait	1	true
bait	2	false

Figure 6.1: Averaged results across the 146 mutated games

As can be seen, many of the levels were never completed – because the solutions were too complex.

6.3 Evolution of levels: Setup

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:

Algorithm 3: How to write algorithms

```
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;
```

6.4 Evolution of levels: Results

6.5 Discussion of results

6.5.1 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”.

Chapter 7

Conclusion

A huge problem in evolving games with the process used in this project, is that intelligent AI controllers are used which spend a lot of time in playing the through each game – just playing through a single game ten times takes the Explorer controller up to 2000 ticks * 40 ms * 10 = 800 seconds.

A more clever approach might be to first probe the games with some more simple controllers, or decreasing the allowed time for intelligent controllers, allowing the games to be played through more quickly, and decide if the game is worth spending time on for a more precise analysis (using intelligent controllers).

It might be interesting to analyse games using a wider array of controllers, or analysing by letting the same controller (or controllers) play with a different amount of time allowed per clock-tick, or even using controllers designed to "play bad" (trying to die, decrease score) could be interesting

Also, it could be interesting to retrieve more extrinsic data from playing through a game (e.g. the proportion of interaction-rules that was triggered).

Maybe one could make an algorithm to play a game in a more similar fashion as humans: First reading, and understanding the rules (in pac-man: eat all the cheese, in boulderdash: get to the exit after getting X gems). Then, using a learning-approach, playing through the game/level, not know much about possible interactions, but maybe assuming that colliding with moving sprites is dangerous (they could be enemies!). And then playing the game repeatedly to learn to play the game better.

Appendices

Appendix A

GVG-AI example 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.

Appendix B

Games designed during thesis

B.1 Action arcade games

Crackpots [?] 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 [?] 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.

B.2 Puzzle games

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.

Appendix C

Designed games results

This chapter shows the individual data for each game, for the test discussed in Section **??**.

Aliens

controller	*score mean*	*std.dev.*	*normalised-mean*	*winrate*	*act-entropy*
Explorer	82.64	1.91	1.0000	1.0000	0
MCTS	69.16	4.47	0.8392	1.0000	0
GA	70.08	4.53	0.8499	0.9600	0
Onestep-S	55.08	12.74	0.6710	0.1600	0
Random	45.32	17.24	0.5555	0.0800	0
Do Nothing	−1.00	0.00	0.0000	0.0000	0

Boulderdash

controller	*score mean*	*std.dev.*	*normalised-mean*	*winrate*	*act-entropy*
Explorer	82.64	1.91	1.0000	1.0000	0
MCTS	69.16	4.47	0.8392	1.0000	0
GA	70.08	4.53	0.8499	0.9600	0
Onestep-S	55.08	12.74	0.6710	0.1600	0
Random	45.32	17.24	0.5555	0.0800	0
Do Nothing	−1.00	0.00	0.0000	0.0000	0