# Towards generating arcade game rules with VGDL

Thorbjørn S. Nielsen*, Gabriella A. B. Barros*, Julian Togelius†, and Mark J. Nelson‡
*Center for Computer Games Research, IT University of Copenhagen, Denmark
†Department of Computer Science and Engineering, New York University, NY, USA
‡Anadrome Research, Houston, TX, USA
{thse@itu.dk, gbar@itu.dk, julian@togelius.com, mjn@anadrome.org}

*Abstract*—**We describe an attempt to generate complete arcade games using the Video Game Description Language (VGDL) and the General Video Game Playing environment (GVG-AI). Games are generated by an evolutionary algorithm working on genotypes represented as VGDL descriptions. In order to direct evolution towards good games, we need an evaluation function that accurately estimates game quality. The evaluation function used here is based on the differential performance of several game-playing algorithms, or Relative Algorithm Performance Profiles (RAPP): it is assumed that good games allow good players to play better than bad players. For the purpose of such evaluations, we introduce two new game tree search algorithms, DeepSearch and Explorer; these perform very well on benchmark games and constitute a substantial subsidiary contribution of the paper. In the end, the attempt to generate arcade games is only partially successful, as some of the games have interesting design features but are barely playable as generated. An analysis of these shortcomings yields several suggestions to guide future attempts at arcade game generation.**

## I. Introduction

Could a computer independently design a complete game? This question has been posed in the past by us as well as other researchers [1], [2], [3], [4], [5], [6], and a number of attempts have been made to generate complete games – usually with limited success. We will discuss some approaches to game generation in the next section, but for now let us instead ask: why would a computer not be able to design a complete game? After all, computers have been able to independently design tables [7], antennas [8], theorems [9], pictures of many different kinds [10], [11], music [12] and many other things. Evolutionary algorithms are often used for computational design; in fact, Evolution Strategies were originally developed in the context of automated design of aircraft wings [13]. Designing *parts* of games – levels, textures and items – is a burgeoning enterprise under the name procedural content generation in games [14]. It therefore stands to reason that complete games, including their rule sets, should be as computer-designable as other types of artifacts.

However, we have a problem with quality evaluation. An aircraft wing can be tested in a wind tunnel or with aerodynamic simulation, and the correctness of a theorem can be verified by following the steps in the proof; in either case, the testing activity are mostly mechanizable. But what about games? It would seem that the quality of a game is much more ephemeral than that of an antenna or wing, more like the quality of a picture or piece of music. But whereas pictures or music can be experienced more or less passively, games require active appreciation: you must interact with a game, specifically by playing it, in order to understand it.

For a computer program to design games, it therefore needs to also be able to play its own games. To use evolutionary computation or any similar search/optimization algorithm as a generation method, many thousands of bad games will need to be evaluated before we find a good game. We therefore need artificial players that are able to play all those unseen games with some skill, and who can do so very fast. In other words, we need competent and fast general game playing algorithms.

If this sounds like a tall task, note that the challenges don't end there: just because we can play a game automatically does not mean that we have a way of evaluating its quality. (Game-playing algorithms tend not to have opinions on the games they play.) However, we could tell something about the quality of the game by observing how one or several algorithms play the game. Specifically, in recent work we explored the idea of estimating game quality by how much better good players play a game compared to how bad players play it [15]. In other words, a good game would allow for more skill differentiation. This idea was operationalized as the Relative Algorithm Performance Profile (RAPP), where a number of game-playing algorithms are tested on all games and the relative performance of these algorithms are compared. We showed that human-designed games did indeed exhibit higher performance difference between strong and weak game-playing algorithms than random or mutated games did.

The current paper builds on our previous work, deepening our analysis of game quality using RAPP, devising a concrete fitness function and variation operator for game evolution, and analyzing the results of several attempts to generate games. We focus on simple arcade games expressed in the Video Game Description Language (VGDL). We introduce two new game tree search algorithms with very different performance profiles, one of which is the currently top-performing algorithm for the original test set of games[1]. The original contributions of this paper thus include the first attempt at generating games using VGDL and two new game playing algorithms.

## II. Related work

The concept of RAPP is similar to that of player performance profiles [16], [17], which compare players' performances against opponents that have various levels of strengths.

[1] One of the algorithms, "Explorer", was briefly described previously; the current paper describes an updated version and provides more detail.

Performance profiles are used to compare different player strategies in a given-game, to evaluate these strategies. On the other hand, RAPP involves comparing different strategies against different games, in order to evaluate the game, not the strategies. Furthermore, this work is also related to general video game playing and game generation, as described below.

### A. General video game playing

Research on general game playing (GGP) aims to create artificial players that can proficiently play not just one but a large number of games, in particular games that are *unseen* (not known by the algorithm designer). The arguably most famous outlet and impetus for this research is the General Game Playing Competition, organized since 2004 [18]. Competitors submit game-playing agents to this competition; the agents are provided with descriptions of several unseen games written in a specialized game description language (GDL) and judged on their capability to play these games. The GDL used provides a rather low-level description of games, and players will have to build up their own mechanics representation from the description. Due to constraints of the GDL, games used in the GGP competition are mostly board or puzzle games.

One notable development in GGP agents is the emergence and dominance of agents based on Monte Carlo Tree Search (MCTS), a statistical tree search algorithm. It requires very little to no prior knowledge of the game's domain, and has been used successfully in many games [19].

Besides the board-game-focused GGP competition, another domain that has seen GGP research is playing Atari games. The Atari Learning Environment (ALE) is a framework for testing agents using emulated Atari 2600 games [20]. Agents are given the raw screen data and the score of the game as inputs, and return joystick commands. The Atari 2600 was originally released in 1977, and today it is considered useful for research purposes due to its large collection of titles, that go from board games to shooters. Approaches using MCTS [20], temporal difference learning [20] and neuroevolution [21] have been applied to ALE.

Finally, and what we base our work on here, the General Video Game AI Competition framework (GVG-AI) is a benchmark for general game playing that uses games described in the Video Game Description Language (VGDL) [22], [23], [24]. VGDL can be used not only for testing GGP algorithms, but also for game generation, in a 2D environment.

### B. Game generation

Game generation could be done in a number of different ways. Generally speaking, any system for generating complete games including their rules is likely to be either constructive (e.g. grammar-based), solver-based or search-based [6]. An example of a solver-based method is Variations Forever, which uses Answer Set Programming (ASP) to explore the search space of 2D game design rules [25]. The majority of attempts to generate games are however search-based. Browne's Ludi system is capable of creating good quality board games, by searching a constrained space of games [3]. Similarly, ANGELINA evolves new games, expanding the search space to not only rules, but also characteristics and levels [4].

Due to its complexity, the task of complete game generation is usually not attacked in full. This work focuses on the generation of 2D arcade-style games, due to their more restrictive nature: being focused on the interaction between game elements in a two-dimensional space. It has previously been argued that this class of games should be relatively "generatable" when compared to other types of games [6].

## III. Methods

### A. VGDL and the GVG-AI framework

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 [22], [23] and implemented [24] in order to support both general video game playing and video game generation. A VGDL game is composed of a description and one or more levels. The description consists of four parts: *SpriteSet*, *InteractionSet*, *LevelMapping* and *TerminationSet*. The *SpriteSet* describes all game object's types and properties. *IntectionSet* defines how these objects will interact with each other. The *LevelMapping* describes how game objects are represented in a level text file, and the *TerminationSet* defines what are the conditions for this game to finish.

The GVG-AI framework is a testbed for general game-playing controllers on VGDL descriptions. Controllers are called once at the beginning of each game for setup, and then once per clock tick to select an action. They do not have access to VGDL descriptions of games, and receive only the games current state: the position and type (e.g. portal, immovable, resource) of each sprite. However these states can be forward-simulated to future states. Thus game rules are not directly available, but a simulatable model of the game can be used.

The framework comes with a number of example controllers, including a controller that acts randomly and an implementation of Monte Carlo Tree Search (MCTS).

### B. New controllers

We introduced three controllers of an increasing degree of cleverness: *One-step*, *Deep-search* and *Explorer*. Each uses a different strategy when simulating the forward model provided by the GVG-AI framework. They are described below:

The *OneStep* algorithm advances the game's current state once for each allowed action. State value is based on score and win/lose result, and the action with the best resulting state value is chosen. If no state is better, a random action is chosen.

The *DeepSearch* controller is a mix between a depth-first search and MCTS, seeking to advance deep into a tree with a few initial actions. It starts out in a similar fashion as *OneStep*, by expanding all possible actions. These game-states are then simulated further upon by advancing each game state a single time, with a partly random action, without copying the previous state. This continues until the controller runs out of time. Each action is evaluated by considering the score and win/lose-value for the states that can be achieved from each of the initial states. Also, the controller inputs the initial game state to be simulated from again when other simulations have failed or it has reached a certain search depth.

**Algorithm 1:** DeepSearch algorithm

```
queue ← initial gameState
justExpandedFromInitial ← false
while has time left do
    gameState = queue.poll()
    if gameState == initialGameState then
        for action : PossibleActions do
            queue ← gameState.copy().advance(action)
        justExpandedFromInitial ← false
    else
        queue ← gameState.advance(random, but not opposite to first-, action)
        d ← depth of search //amount actions performed
        value[action] ← value[action] + value(newGameState) / PossibleActions.length$^d$
        if d == 4 and !justExpandedFromInitial then
            queue ← initial gameState
            justExpandedFromInitial ← true
return action leading to highest value, or random action if values are equal
```

Finally, the *Explorer* was designed specifically to play arcade-style games. Unlike controllers that only use the current state to decide on an action, it stores information about visited tiles and prefers visiting unexplored locations. In addition, it uses three sets of accumulated statistics: how probable death is from each action; the score that can be achieved from each actions; and how boring each action is. If the controller has died too many times, it takes the action with the lowest death rate. This death rate of an action uses the forward simulation of that action, and if it leads to a death the death rate is given by the sum of $1 * k^n$ per action in this sequence, where $n$ is the total of actions in the sequence, and $k == 4$, chosen by testing different values. Otherwise, it takes the action with the best score. If several actions have similar scores, boringness breaks the tie. In this context, boringness refers to how often the avatar has visited a certain tile, i.e. the fewer times it has been visited, the less boring it is. The controller also addresses a common element of VGDL games: randomness. It gains an advantage in several games by simulating the results of actions repeatedly, before deciding the best course to chose.

## IV. Playing existing and generated games

The GVG-AI framework comes with 21 human-designed games. Some are adaptations of classic arcade games, which can be plausibly assumed to represent good games. Others were newly written for the competition. We excluded the newly written games, and in addition excluded games where no algorithms played well (all of which were puzzle games), since they don't provide us useful information with our current approach. We then added two newly encoded adaptations of classic arcade games, bringing us to 13 games to use as examples of non-generated "good" games: Aliens, Boulderdash, Frogs, Missile Command, Zelda, DigDug, Pacman, Seaquest, Eggomania, Solar Fox, Crackpots, Astrosmash, and Centipede.

**Algorithm 2:** Explorer algorithm

```
queue ← initial gameState
deathActions, scoreActions, boredActions ← []
while has time left do
    gameState = queue.poll()
    if gameState == initialGameState then
        for action : PossibleActions do
            queue ← gameState.copy().advance(action)
    else
        d ← depth of search //amount actions performed
        firstAct ← first action performed
        deathActions[firstAct] + = isDead(gameState) / PossibleActions.length$^d$
        scoreActions[firstAct] + = scoreValue(gameState) / PossibleActions.length$^d$
        boredActions[firstAct] + = boredValue(gameState)
        queue ← gameState.advance(with least boring action)
if deathActions values higher than threshold then
    return action for lowest deathAction value
if scoreActions difference higher than threshold then
    return action for highest scoreActions value
return action for lowest boredActions value
```

We then produce two kinds of generated games. One set are variants of the existing games, which we call "mutated" games. Each of the 13 designed games is mutated in 10 different ways (described below), to produce 130 mutated games. The other set are randomly generated without reference to existing games; we generate 400 of these, each accompanied by a randomly generated level (of size 15x15).

Six different controllers play through each game in all three sets: *SampleMCTS*, *Explorer*, *DeepSearch*, *OneStep*, *Random* and *DoNothing*. *SampleMCTS* and *Random* are from the GVG-AI framework. The first is a vanilla MCTS algorithm, while the second simply returns a random action each turn. *DoNothing*, as the name suggests, always returns a null action.

We modified the GVG-AI framework so that players cannot score below 0, which was already the case in the human-designed games. This makes comparing scores across controllers more straightforward.

### A. *Mutating games*

In the first set of generated games, we produce 130 "mutated" versions of the 13 human-designed games. How to best mutate games is not immediately apparent, however, since VGDL games have a number of elements (rules, sprites, levels, etc.) and some are interdependent. The sprite definitions and rules of a VGDL game are constructed from a combination of a class and a series of parameters specific to the class, so it is necessary to change the parameters if the class is changed, but the parameters of an existing class can freely be mutated.

In order to increase the chances of creating a playable mutation, we limited mutations to only be able to change interaction rules from the *InteractionSet*, with 25% mutation probability per interaction rule. Levels, win/loss rules, and sprite sets were left unchanged.

### B. *Generating random games*

To generate new games completely from scratch, we need to construct the four parts of a VGDL description: an array of sprites for the *SpriteSet*, interaction rules for the *InteractionSet*, termination rules for the *TerminationSet*, and level mappings in the *LevelMapping*.

We need some limits on the search space, so somewhat arbitrarily limited the number of sprites, interaction rules, and termination rules: games must have $3-8$ sprites, $3-10$ interactions and 2 terminations (one "win" and one "lose" termination). In addition, no parent-child structure was allowed in the *SpriteSet*, i.e. no sprite could be a sub-type of another. All objects were then given random sprite images.

Generating a game is intimately linked to generating levels appropriate for the game's rules. One could end up with generating high quality game descriptions, but if the level does not fit the gameplay the games might be deemed bad. In these experiments, we don't address that issue in detail, and instead randomly generate levels with the only constraint that it should not make the game engine crash, e.g. objects in the level must exist in the game definition, avatar sprites cannot be spawned and sprites cannot transform into an avatar.

### C. *Testing and result analysis*

The six controllers each played through the set of human-designed, mutated, and randomly generated games. To constrain playthrough time, each game was played ten times, with 2000 maximum clock ticks per playthrough, and 50 ms maximum per tick. In addition, games were aborted if a clock tick ever took more than 50 ms. For each game playthrough, data recorded was: score, win-loss value, number of clock ticks used, and a list of actions performed.

### D. *Results*

Figure 1 shows the normalized average score for all three types of games, Figure 2 the win rate, Figure 3 average clock tick count, and Figure 4 the average action entropy, i.e. how much variety there is in action selection.

There is clearly a large difference between the results of the more intelligent algorithms and those of the intended "bad" controllers, with both a higher win rate and a significantly higher average score. Surprisingly, the *DeepSearch* algorithm has a significant lead over the other algorithms – in spite of its simple action selection approach – making it a suitable candidate for a "good player".

On the other hand, *clock-ticks* and *action entropy* do not show a similarly clear difference between more and less intelligent algorithms. Examining the results from each game, it is noticeable that the profiles for these values are very different from game to game.

The results suggests that there exists some relationship between the performance profiles of different controller types, on different sets of games, especially in relation to win-rate. This supports the hypothesis that relative algorithm performance profiles can be used to differentiate between games of different quality. However it should be noted here that some games from both the mutated and generated game set, contains games that when examined by themselves have similar distributions as the average designed game. 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.

In regard to randomly generated games, performances were mostly indistinguishable from designed games. Upon further examination of many of these descriptions, it was noticed that they were in general too trivial and aimless to be entertaining. However, by also comparing these games with others without "well-formed" performance profiles, it became clear that algorithms performance profiles could at least be used to separate "not-completely-bad" games from terrible games.

This examination strengthens our initial assumption that the generated games, at least from the randomly generated set, are of a overall low quality. Besides the above, a few re-occurring problems was found in the generated games: Sprites often completely leave the level playing field, the game can only be won in the first few (¡50) frames, the outcome of the game is too random (e.g. some sprites cannot be interacted with) or several of the sprites- and/or rules are never used.

## V. Evolving new games

### A. *Fitness*

The tests shown in Section IV-D indicate that general game playing controllers performance could potentially be used to determine the quality of games. But there were many different

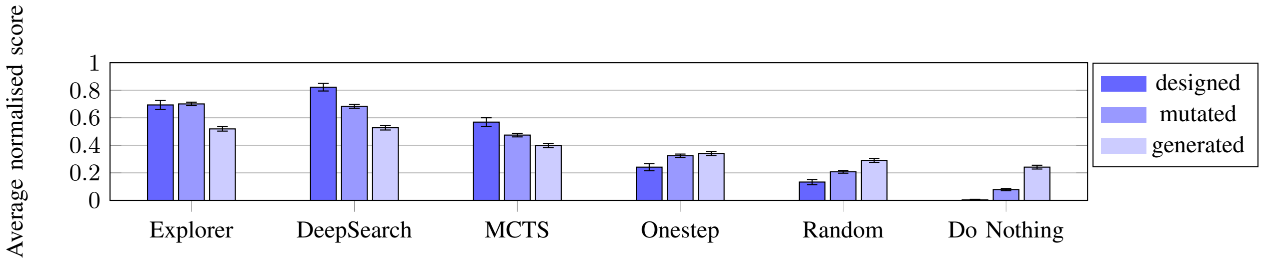

Fig. 1: Normalized average score across all games of the three different set of games: Designed-, mutated- and completely generated games

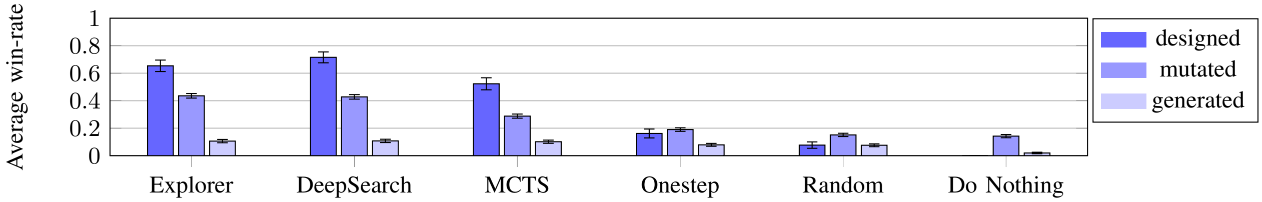

Fig. 2: Averaged win-rate for the three sets

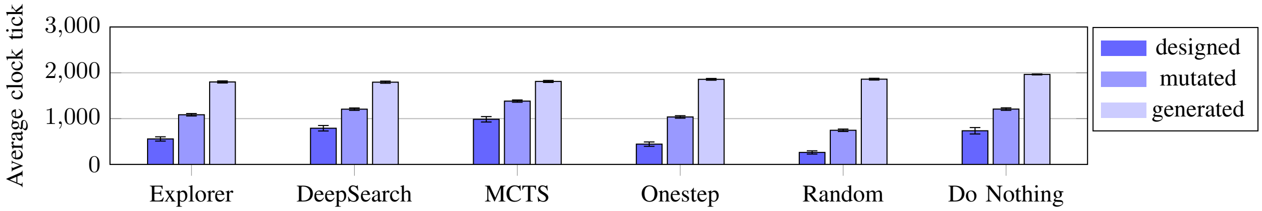

Fig. 3: Averaged clock tick for the three sets

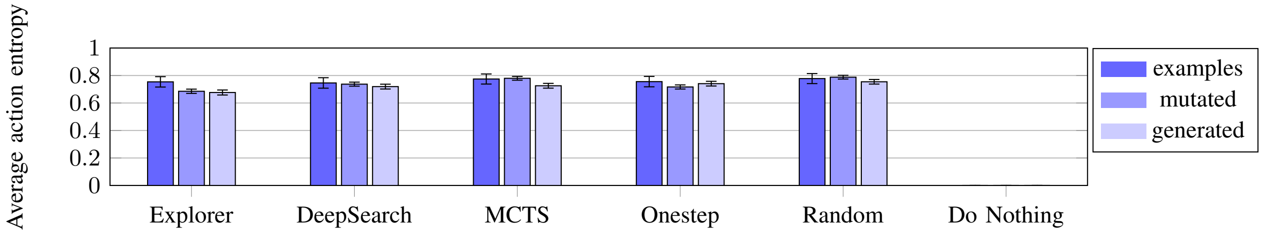

Fig. 4: Averaged action entropy for the three sets

features and relationships between algorithms. Therefore, it was necessary to identify the most important statistical values from said results for use in a fitness function, while trying to exclude statistics that do not signify game quality.

Out of all outcomes, the score and the win-rate seem to have the most distinct distributions for different type of games, making them interesting to use for an evaluation function. For example, the Explorer and the Random showed significant difference between them. Likewise, the *action entropy* might also be appropriate to use, but the difference in the statistical distributions are much smaller for these.

Even after limiting fitness features to score and win-rate, there are still many possible combinations. For instance, we could have a feature for the difference in average score between every single controller, but it would result in $\binom{6}{2} = 15$ features for a single statistic. We could also use a series of different score values: *minimum* and *maximum* score, *standard deviation* of score, *median*, *quartiles* or any form of normalised score, and of course we could use combined values between features (e.g. the score increase per clock tick).

It is also not straightforward to calculate a feature even after deciding which controllers and values to compare. We could use a simple relative difference formula (e.g. $f = |\frac{a-b}{max(a,b)}|$), but it might make more sense to hand-craft each feature to a specific goal. For instance, for action entropy, it appears that designed games cause intelligent controllers to have similar values as the Random controller, which is not true for randomly generated games – where the intelligent controllers often find a simple solution requiring similar actions. A feature could thus consist of having a value of 1 if entropies are close, decreasing towards 0 when Random has a higher entropy, and not rewarding the game for having the intelligent controllers action entropy be higher than Random.

To avoid a large set of possible fitness features, we started by using only two relative difference scores: between the mean score of DeepSearch and DoNothing, and between the win rates of the same two controllers. The total fitness of a game is the sum of these two values. Using these features, 12 designed games have a "perfect score" of 1, while the game DigDug only scores 0.5 (since the DeepSearch controller was never able to win). 30 mutated games have a score of 0.95 or higher (23 have perfect score), while only 4 of the generated games have a fitness above 0.5 (though these are all perfect scores).

After preliminary experiments, it became clear that we needed more fitness features to exclude certain classes of bad games given high fitness with the simple approach. First, games where intelligent controllers can win too quickly (under 50 clock ticks) are penalized. Secondly, games in which intelligent controllers can *only* win (and never lose) are penalized.

The result is the following fitness function:

$$f = \frac{RD(score) + RD(wins) + win\_50 + win\_lose}{4}$$

Where $RD$ means relative difference between algorithms for the given parameter, $win\_50$ is -1 if a controller can win in fewer than 50 clock ticks, 1 otherwise, and $win\_lose$ is 1 if the game can be both won and lost, -1 otherwise. The resulting fitness is always between -1 and 1.

In addition a game was automatically given a fitness of -1 if any controller is disqualified in any playthrough, the score and wins are always the same, or the controllers all finish in fewer than 50 clock ticks.

### B. Game generation methods

As with the experiments in Section IV, we evolve games here in two different ways: by mutating human-designed games, and by generating new VGDL descriptions. Both approaches use a simple evolution strategy (ES) algorithm, with mutation and crossover operations across several generations. In each iteration, a population of game descriptions was tested using two controllers: `DeepSearch` and `DoNothing`. Fitness is calculated for each game, with the lowest fitness games removed from the population. As in the mutation experiments in Section IV, only the `InteractionSet` was modified in each generation.

The evolution process stopped when the highest fitness no longer increased in 10 generations, when the best game's fitness was above 0.98 (an almost perfect score), or when the maximum 15 generations allowed were over.

In order evolve randomly generated games, it was necessary to verify if minimum criteria for a game were fulfilled (i.e. the description is well-formed and the game is winnable). Once this randomly generated game was assessed to be playable, it was then evolved in the same manner.

### C. Results

We evolved 54 total games: 45 from the human-designed games, and 9 from a randomly generated starting point. Of these, 40 mutated, and 6 randomly generated games had a perfect, or near-perfect fitness of over 0.98, making them the most interesting to examine further.

Figure 5 compares the original *InteractionSet* of *Boulderdash* with one of its mutated descendants. Figure 6 describes a game evolved from a randomly generated description - without its *LevelMapping* section.

```
        InteractionSet
dirt avatar > killSprite
dirt sword  > killSprite
diamond avatar > collectResource
diamond avatar > killSprite scoreChange=2
moving wall > stepBack
moving boulder > stepBack
avatar boulder > killIfFromAbove scoreChange=-1
avatar butterfly > killSprite scoreChange=-1
avatar crab > killSprite scoreChange=-1
boulder dirt > stepBack
boulder wall > stepBack
boulder diamond > stepBack
boulder boulder > stepBack
enemy dirt > stepBack
enemy diamond > stepBack
crab butterfly > killSprite
butterfly crab > transformTo stype=diamond scoreChange=1
exitdoor avatar > killIfOtherHasMore resource=diamond limit=9
-->
        InteractionSet
dirt avatar > killSprite
dirt sword > killSprite
boulder avatar > attractGaze
dirt diamond > killIfFromAbove
moving wall > stepBack
moving boulder > stepBack
avatar boulder > killIfFromAbove scoreChange=-1
butterfly dirt > killIfHasMore limit=15 resource=diamond
avatar crab > killSprite scoreChange=-1
boulder dirt > stepBack
butterfly avatar > killIfFromAbove
sword crab > cloneSprite
boulder boulder > stepBack
avatar EOS > killIfHasLess limit=13 resource=diamond
diamond dirt > transformTo stype=diamond
crab butterfly > killSprite
diamond avatar > collectResource
exitdoor avatar > killIfOtherHasMore limit=9 resource=diamond
butterfly boulder > killSprite
```

Fig. 5: The original (above) and evolved (below) set of interaction-rules of a VGDL description, generated by mutating the description of Boulderdash.

## VI. Discussion

In many cases, evolving games quickly breaks important aspects of the original games, often removing the core challenge of the game. Examples: giving the player the ability to walk through walls or boulders; not removing items after they have been picked up; or even not defining that enemies can harm the player. Nonetheless, many generated games do have interesting properties and features.

In one mutation of *Boulderdash*, `boulderdash_mut09`, gems do not disappear when collected, making the player able to win very quickly (one just has to stand still on a gem for a second, then head for the exit). But enemies have also become more mobile, so overall the game has some sense of challenge and difficulty when compared to most of the generated games. An interesting feature was that the avatar has gained an ability to push boulders, making them glide across the level, thus being able to kill certain enemies. However, the enemies that

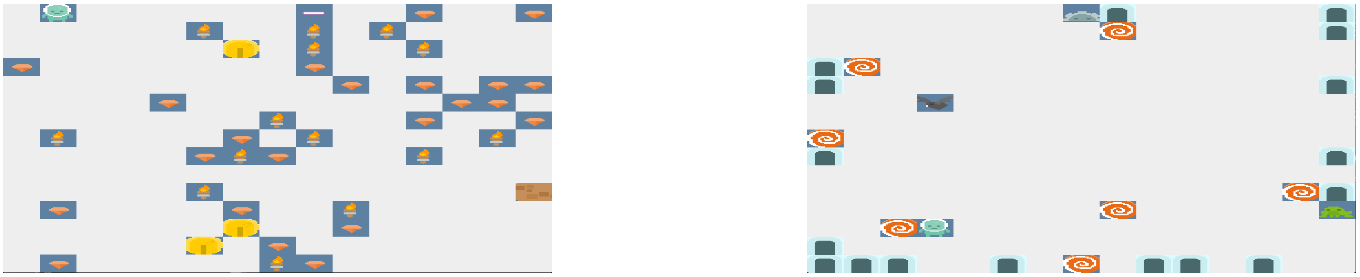

Fig. 7: Visual representation of two “evolved-from-random-generated” VGDL games: `evol_game001` (left), having a fitness score of 1, and `evol_game002` having a score of ≈0.98

```
BasicGame
 SpriteSet
  avatar > MovingAvatar img=avatar
  gen1 > Passive img=honey
  gen2 > Resource limit=4 singleton=TRUE img=missile value=2
  gen3 > Immovable img=pellet
  gen4 > Bomber orientation=DOWN stype=gen5 img=frog prob=0.22090000000000004
  gen5 > Spreader limit=33 stype=gen2 img=fire
  gen6 > Fleeing stype=avatar img=dirt
 InteractionSet
  gen6 gen4 > attractGaze
  gen5 EOS > wrapAround
  gen1 gen3 > killIfHasLess limit=3 resource=gen2
  gen2 avatar > transformTo stype=gen1 scoreChange=2
  gen2 avatar > wallStop
  gen2 avatar > changeResource resource=gen2 value=0 scoreChange=7
  gen6 gen2 > killIfOtherHasMore limit=0 resource=gen2
  avatar EOS > killSprite
  avatar EOS > stepBack
  gen1 EOS > stepBack
  gen3 gen3 > stepBack
  gen3 EOS > stepBack
  gen1 gen1 > spawnIfHasMore limit=11 stype=gen6 resource=gen2 scoreChange=−2
  gen5 EOS > stepBack
  gen6 EOS > stepBack
  gen2 gen5 > collectResource
 TerminationSet
  SpriteCounter limit=0 stype=gen6 win=TRUE
  SpriteCounter limit=0 stype=avatar win=FALSE
```

Fig. 6: The VGDL game description for an evolved game, without *LevelMapping*

can be killed by boulders no longer harm the avatar, and so the new ability does not cause much change to the actual game.

Randomly generated games, on the other hand, appear to have more unique designs. A generally undesirable feature that recurs in several games is that the outcome is partly or completely independent of the player’s actions. Since the fitness function was supposed to explicitly search for games where better players perform better, games in which performance doesn’t matter at all should have been eliminated. That they weren’t is likely a product of using a rather small sample of playthroughs, six per controller. The games are overall quite simple, but many contain some interesting game design for human players, some which seemingly appeared by accident:

In the evolved game `evol_game001` (Figure 6 shows its description and Figure 7 a screenshot), the goal is to kill a block of dirt that whizzes around the level, trying to flee from the avatar. However, true to the action-arcade genre of the game, the player can additionally increase his/her score by moving the avatar to a laser-sprite (`gen2` in the description). Only one laser-sprite can exist in the level at any given time, and the player can only increase his/her score by repeatedly going to each laser that is spawned. If the player picks up a laser in the same position of a previously picked up laser, the player loses all his points.

Another game, `evol_game002` (Figure 7), is rather simple: its goal consists in killing the avatar, which can be achieved by walking to the “cog” sprite (in the screenshot, it appears at the top of the screen). However the game is lost if the bat sprite (a `RandomNPC`) ever collides with the same object, making the outcome rather random.

Overall, the VGDL game generation process was not able to create any game of reasonably high quality, especially in comparison to the human-designed arcade games. To increase the game generator’s quality, it seems necessary to refine the fitness function, possibly by identifying more aspects of games and playthroughs. For instance, one could examine how each controller increased its score across a playthrough, 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.

Seeing that a main problem of many of the generated games is their requirement for superhuman responses or otherwise miscalibrated timescale, another way of improving the generation process could be to restrict all controllers to only act on a human timescale, for example, by restricting the number of times per second the active decision could be changed, or by introducing some temporal indeterminacy into when exactly a chosen action would be executed.

One major issue with a straightforward evolutionary approach is that it takes a lot of time for intelligent controllers to play each game. Even the quite severe limits we put on controllers’ playthroughs here can lead to large total time budgets. If a controller is allowed up to 800 ticks of 50 ms each, one simulated playthrough can take up to 40 seconds. With a population size of 50, and 6 samples per fitness evaluation, that results in over 3 hours for a single generation in the worst case. Addressing this might require better ways of allocating limited CPU budget to fitness evaluation, such as failing fast on “obviously bad” games. Alternatively, one could probe each game or level with simple controllers, which can play the game quickly, and decide if the game is worth spending time on for a more precise analysis using “intelligent” controllers.

We do not, at the moment, verify with humans if the games are really good or bad. This should be done, possibly by letting a set of experienced and novice users play each game and compare their performance, removing games where the performance does not significantly differ between players.

## VII. Conclusion

The goal of this project was to automatically generate a complete "as-enjoyable-as-possible" 2D game using VGDL. In order to do so, we needed a manner of evaluating said games. Firstly, generating the VGDL game descriptions of *action-arcade games*, we attempted at correlating the quality of a game with the performance of general game playing algorithms. Our results using six different controllers and human-designed, mutated human-designed and randomly generated games show indication that a relation between a game's quality and a relative performance evaluation exists.

Thereafter, we proposed a method for generating and evaluating games automatically. Our approach was successful at generating interesting game designs, and indicates that using the performance of game playing algorithms to generate content is worth examining further. Using an evolutionary strategy, a set of playable VGDL games of varying quality was created, some using existing human-designed games as basis, with the original game's levels, and others using randomly generated games and levels. Our process was able to generate several sets of interesting game-rules, but at the same time many games generated contain severely trivial game design, or game rules largely based on luck. Overall, the generators are not able to pin-point interesting games in the space of VGDL games – instead they are able to find a subset of more interesting game description that human game designers can examine further to decide which games are actually "good", which suggests that this process may be useful to idea generation. In addition, it can be applied to further evolve initial ideas created by a human game designer. Allowing the algorithm to select specific mutation operations [26] might vastly improve its ability to search for variations of a given game, while potentially decreasing the chances of "breaking" the game.

In addition, we found that time is a significant limitation. Tree search-based game playing algorithms spend a large amount of time playing through each game, since they need time to decide on an action on every frame. Even with the rather short time frames allowed in the tests and programs (50 ms per frame, 800/2000 frames per playthrough), it can take several days for a single run of the evolutionary algorithm.

## Acknowledgment

Gabriella A. B. Barros acknowledges financial support from CAPES Scholarship and Science Without Borders program, Bex 1372713-3. Thanks to Diego Perez, Spyros Samothrakis, Tom Schaul, and Simon Lucas for useful discussions.

## References

[1] M. J. Nelson and M. Mateas, "Towards automated game design," in *AI* IA 2007: Artificial Intelligence and Human-Oriented Computing*. Springer, 2007, pp. 626–637.

[2] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games*, 2008, pp. 111–118.

[3] C. Browne, "Automatic generation and evaluation of recombination games," Ph.D. dissertation, Queensland University of Technology, 2008.

[4] M. Cook and S. Colton, "Multi-faceted evolution of simple arcade games." in *CIG*, 2011, pp. 289–296.

[5] A. Zook and M. O. Riedl, "Automatic game design via mechanic generation," in *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, 2014.

[6] J. Togelius, M. J. Nelson, and A. Liapis, "Characteristics of generatable games," in *Procedural Content Generation in Games Workshop*, 2014.

[7] G. S. Hornby and J. B. Pollack, "The advantages of generative grammatical encodings for physical design," in *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, vol. 1. IEEE, 2001, pp. 600–607.

[8] G. S. Hornby, A. Globus, D. S. Linden, and J. D. Lohn, "Automated antenna design with evolutionary algorithms," in *AIAA Space*, 2006, pp. 19–21.

[9] D. B. Lenat, "Am: An artificial intelligence approach to discovery in mathematics as heuristic search," DTIC Document, Tech. Rep., 1976.

[10] S. Colton, "The painting fool: Stories from building an automated painter," in *Computers and Creativity*. Springer, 2012, pp. 3–38.

[11] P. Machado and A. Cardoso, "Nevar–the assessment of an evolutionary art tool," in *Proceedings of the AISB 2000 Symposium on Creative & Cultural Aspects of AI*, 2000.

[12] D. Cope, *Virtual Music: Computer Synthesis of Musical Style*. MIT Press, 2004.

[13] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies–a comprehensive introduction," *Natural computing*, vol. 1, no. 1, pp. 3–52, 2002.

[14] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2015.

[15] T. S. Nielsen, G. A. Barros, J. Togelius, and M. J. Nelson, "General video game evaluation using relative algorithm performance profiles," in *Applications of Evolutionary Computation*, 2015, pp. 369–380.

[16] W. Jaśkowski, P. Liskowski, M. Szubert, and K. Krawiec, "Improving coevolution by random sampling," in *Proceedings of the 15th Conference on Genetic and Evolutionary Computation*, 2013, pp. 1141–1148.

[17] W. Jakowski, M. Szubert, and P. Liskowski, "Multi-criteria comparison of coevolution and temporal difference learning on othello," in *Applications of Evolutionary Computation*, 2014, pp. 301–312.

[18] M. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the AAAI competition," *AI Magazine*, vol. 26, no. 2, pp. 62–72, 2005.

[19] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[20] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *arXiv preprint arXiv:1207.4708*, 2012.

[21] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, "A neuroevolution approach to general Atari game playing," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 4, pp. 355–366, 2013.

[22] J. Levine, C. B. Congdon, M. Ebner, G. Kendall, S. M. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson, "General video game playing," *Dagstuhl Follow-Ups*, vol. 6, 2013. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2013/4337/

[23] M. Ebner, J. Levine, S. M. Lucas, T. Schaul, T. Thompson, and J. Togelius, "Towards a video game description language," *Dagstuhl Follow-Ups*, vol. 6, 2013. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2013/4338/

[24] T. Schaul, "A video game description language for model-based or interactive learning," in *Proceedings of the 2013 IEEE Conference on Computational Intelligence in Games*, 2013, pp. 1–8.

[25] A. M. Smith and M. 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*, 2010, pp. 273–280.

[26] D. Ashlock and C.-K. Lee, "Agent-case embeddings for the analysis of evolved systems," *Evolutionary Computation, IEEE Transactions on*, vol. 17, no. 2, pp. 227–240, 2013.