# General Video Game Level Generation

Ahmed Khalifa
New York University
New York, NY, USA
ahmed.khalifa@nyu.edu

Diego Perez-Liebana
University of Essex
Colchester, United Kingdom
dperez@essex.ac.uk

Simon M. Lucas
University of Essex
Colchester, United Kingdom
sml@essex.ac.uk

Julian Togelius
New York University
New York, NY, USA
julian@togelius.com

## ABSTRACT

This paper presents a framework and an initial study in general video game level generation, the problem of generating levels for not only a single game but for any game within a specified domain. While existing level generators are tailored to a particular game, this new challenge requires generators to take into account the constraints and affordances of games that might not even have been designed when the generator was constructed. The framework presented here builds on the General Video Game AI framework (GVG-AI) and the Video Game Description Language (VGDL), in order to reap synergies from research activities connected to the General Video Game Playing Competition. The framework will also form the basis for a new track of this competition. In addition to the framework, the paper presents three general level generators and an empirical comparison of their qualities.

## Keywords

procedural content generation, level generation, video game description language, general video game playing

## 1. INTRODUCTION

Procedural Content Generation (PCG), or algorithmic creation of game content, has been an important research topic for several years [1]. Generators have been constructed for everything ranging from textures to maps to game rules. In this paper, we focus on level generation. Level generation is one of the oldest PCG problems, and arguably the first to find practical use in games; see for example Rogue [2], which generated levels in realtime in 1980. Level generation was invented due to the technical limitation of the hardware devices. These limitations drove developers to use PCG techniques to support more content with a small disk/memory footprint. Although modern technology is much less limiting, level generation is perhaps more important than ever in the game industry. This is largely because the increased diversity in gamer demographics and tastes, and an increased expectation of content quality and quantity, necessitate the automation of some content creation, but also because some types of games use level generation as an aesthetic in its own right.

Almost all of level generation work (either in the research or in the industry) is done for specific games (the exception are level generators that work on a somewhat abstracted notion of levels, such as Sentient Sketchbook [3]). This direction of work is important for having a deeper understanding of the level design for a specific game and an understanding of how the game rules interact with each other. Even more importantly, at our current level of understanding we simply do not know how to construct competent level generators for multiple games. On the other hand, a core limitation of such generators is the amount of work that needs to go into building a generator for each game. This precludes the incorporation of level generation in AI-assisted game design tools that lets you construct new games. It would be a great boon especially to small game developers if level generators were available that did not need to be re-constructed for specific games. From a more academic perspective, PCG could be a grand AI challenge, but only if the challenge of generating specific levels is somehow dissociated from game-specific domain engineering—very much in the same way that General Game Playing and General Video Game Playing have dissociated the AI game playing challenge from the details of particular games [4, 5, 6, 7].

From these concepts arose the idea of general level generators, and general level generation as a challenge. We define the problem of general video game level generation as follows: *Construct a generator that, given a game described in a specific description language and which can be played by some AI player, builds any required number of different levels for that game which are enjoyable for humans to play.* Very recently a few researchers have addressed variants of this challenge.

Khalifa and Fayek [8] created two level generators for games described in PuzzleScript language. PuzzleScript [9] is a Description Language created by Stephen Lavelle to help game designers/developers to prototype Puzzle games quickly. The generators are tested on five different games and the results are judged using human players. The results show that almost all of the levels are playable, and that the fitness function's evaluation of the levels correlate with



*WOODSTOCK '97 El Paso, Texas USA*



DOI: 10.475/123_4

player preferences. The level generators used in this paper are heavily inspired by these two level generators. The work on PuzzleScript level generation in turn builds on work by Lim and Harrell [10].

Neufeld et al. [11] used Answer Set Programming (ASP) to generate levels for any game described using the Video Game Description Language (VGDL), a game description which will be discussed further below. The system by Neufeld et al. translates VGDL scripts into ASP constraint rules. Further, Evolution Strategies (ES) are used to find constraints that can increase the percentage of the generated levels that are playable. The system is tested using three different games and the results show that the generated levels have a similar structure to human-authored levels.

While the work discussed above is a good start, there is no framework with which to compare the different methods and they are based on different description languages. In this paper, we therefore introduce the General Video Game Level Generation (GVG-LG) framework to help with the General Level Generation problem and to help with the unification problem of different systems. The GVG-LG framework is developed in Java and supports a Java interface to allow different users to create their own general level generator. For describing the games, this framework uses the Video Game Description Language (VGDL). VGDL is a game description language designed to be able to represent a large variety of games that feature graphical logics in a 2D world, for example arcade games from the eighties [5, 6]. VGDL is flexible enough to describe games ranging from Space Invaders to Sokoban.

The GVG-LG framework is built on top of General Video Game AI (GVG-AI) framework. That framework is the basis for the General Video Game Playing Competition [12] where competitors are able to develop a general playing agent to play different games. One significant advantage of building on the GVG-AI framework is the ability to reuse both games and controllers constructed for that competition.

The GVG-LG framework will be used during the Level Generation track of the GVG-AI competition. In this track, competitors will create level generators for the GVG-LG framework. The competition will use a protocol similar to that used in the Level Generation Track of the Mario AI Championship [13] where human players are used to judge the generated levels. Our intent is that this competition track will create a community of researchers and practitioners working on the general level generation problem using a common benchmark.

The paper is structured as follows: Section 2 gives an overview of the GVG-LG framework. Section 3 discusses the protocol used in the competition as well as in the empirical study in this paper. These are followed by Section 4 which describes three different algorithms that are incorporated in the GVG-LG framework. As a "dry run" of the competition we have organized a small pilot study to test our algorithms. The results are presented in Section 5 and discussed in Section 6. Section 7 provides an outlook on future paths and challenges.

## 2. THE GVG-LG FRAMEWORK

The GVG-LG framework is an extension to the GVG-AI framework. It allows competitors to integrate their level generators and test them against a variety of different games. Figure 1 shows the relationship between a supplied level generator and the GVG-LG framework. The framework provides the generator with a game description and in return it provides a generated level by overriding *generateLevel* function. The level generator is able to change the LevelMapping section by overriding *getLevelMapping* function.

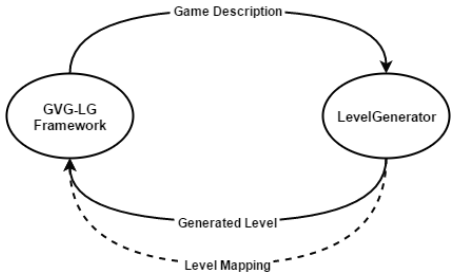


**Figure 1: Relationship between the GVG-LG Framework and the Level Generator**

```
public abstract class AbstractLevelGenerator {

    public abstract String generateLevel(GameDescription game, ElapsedCpuTimer elapsedTimer);

    public HashMap<Character, ArrayList<String>> getLevelMapping()
    {
        return null;
    }
}
```

**Figure 2: AbstractLevelGenerator Class functions**

Figure 2 shows the *AbstractLevelGenerator* class. The level generator must extend the *AbstractLevelGenerator* class and implement a *constructor* and the *generateLevel* function. The *generateLevel* function must return a string which represents the generated level layout. This string is parsed by the framework using the *LevelMapping* section in the game description file. Competitors can use their own *LevelMapping* section but they will need to override a *getLevelMapping* function. This function must return a *HashMap* object which maps a character to a list of corresponding sprite names.

Both of the *constructor* and the *generateLevel* function get a *GameDescription* object and an *ElapsedCpuTimer* object. The *ElapsedCpuTimer* object provides the generator with the maximum amount of time to spend in this function, while the *GameDescription* object encapsulates all the information about the current game. The *GameDescription* provides the generator with different functions that can retrieve information about the sprites, the termination conditions, the interactions, and the level mapping.

Sprites are the main game objects. Sprites are divided into six different categories: Avatar, NPC, Resource, Portal, Static, and Moving. Generators can get a list of *SpriteData* objects for all sprites or for a specific sprite category. *SpriteData* is a data structure holding all required information about the sprite. It has a name, a type, and a list of sprite names. For example: the avatar in *Space Invaders* is named *avatar*, is of type *FlakAvatar* (only moves horizontally and shoots), and it has a *bullet* sprite in its list (*bullet* spawns on shooting).

Termination Conditions define how and when the game should terminate. Generators can get a list of *TerminationData* objects. *TerminationData* is a data structure encapsulating all the related information about the termination conditions. It has a *type*, a *limit*, a *win flag*, and a *list of*

*sprite names*. The type defines how should the game end, the limit is used to check if the condition is satisfied, the win flag is for marking the condition as winning or losing, while the list contains all involved sprite names in the condition. For example: In *PacMan*, the losing condition is the number of the avatar sprite in the level is less than or equal to zero. This condition is of type *SpriteCounter* with a limit equal to *0*, marked as losing condition, and has a *PacMan* avatar name in the list of sprites.

Interactions are collision effects between any two sprites. Generators can get a list of *InteractionData* objects between any two sprite names (they must be defined in sprites). *InteractionData* is a data structure containing information about the collision effect between two sprites. It has a type, a *scoreChange*, and a list of sprite names. For example: In *Frogger*, when a frog collides with a car, the frog dies. The interaction type is *KillSprite*, the score is decreased by *1*, and it has an empty list of sprite names.

A Level Mapping helps the framework to parse the level data. It is a *HashMap* which maps a character to a list of sprite names. The generator can retrieve this *HashMap* or even to replace it by implementing the *getLevelMapping* function. For example: In *PacMan*, *p* refers to a *pill*, while *a* refers to *PacMan* himself.

The generated level must follow two hard constraints. The first constraint is that the level string should not contain any unknown character that is neither in the original nor the new *LevelMapping* (by overriding *getLevelMapping*). The second constraint is that the level must only have one avatar.

# 3. THE GVG-LG COMPETITION

As mentioned before, the framework will be used for the level generation track of the General Video Game Playing Competition. The framework can automatically run level generators submitted as zip files. All submitted level generators must generate a level within a certain time (in this paper we are using a five-hour time limit) on a reference computer, the specifications of which are to be determined; for this paper, we are using a recent MacBook[1]. If the generator exceeds this time, it will be disqualified. There are no constraints on the programming language for the generator, but the competition only provides a Java interface. It should be easy enough for anyone working with another language to write an interface in that language. Based on the number of submissions, it might be impossible to judge all the submissions by humans; in that case, some submitted generators might be disqualified based on computational testing. This testing will be discussed later in Section 4.3.

These generated levels are tested using human judges. Our system will randomly choose two different generated levels from two different level generators. The system allows the human judges to play the selected levels for any number of times. The system also ensures that each level is picked at least once to be played. After the judges finish playing the two levels, they indicate a preference between them. All the judges' preferences are recorded into a SQL database and the most preferred generator across all games wins the competition.

# 4. DESCRIPTION OF GENERATORS

[1]2.9 GHz Intel Core i5 with 8 GB 1867 MHz DDR3

This section explains three sample level generators that are provided with the GVG-LG Framework, and which are also compared empirically in section 5.

## 4.1 Random Level Generator

This is a very simple level generator. It generates levels by placing the defined sprites at random empty positions, then surround the borders with solid tiles. Each tile position has a probability (set to 10% to give the best playable and visually appealing) to be filled with a random sprite picked using a uniform distribution. This probability can be adjusted to generate more or less cluttered levels. The generated levels have size proportional to the number of game sprites defined. The generator ensures that the produced levels have at least one of every sprite and only one avatar.

## 4.2 Constructive Level Generator

This level generator utilizes the information presented in the *GameDescription* object to generate better designed levels. For example if the *avatar* sprite is of type *FlakAvatar* or *HorizontalAvatar* (a type of avatar which can only move horizontally), it should be placed either at the top or the bottom of the level. The generator analyzes the *GameDescription* object using the *GameAnalyzer*, which is part of the competition framework. The *GameAnalyzer* divides the game sprites into five different categories:

- **Avatar Sprites:** as defined in the *GameDescription* object.
- **Solid Sprites:** block the movement of the avatar and do not have any other interaction with the avatar.
- **Harmful Sprites:** kill the avatar upon interaction (or spawn sprites that do this).
- **Collectible Sprites:** are destroyed upon interaction with the avatar and are not harmful sprites.
- **Other Sprites:** any sprites that do not fit into the above categories.

The *GameAnalyzer* object also provides two lists with information about sprites. The first list keeps track of game sprites created from other sprites (i.e. bullets), while the second list contains sprites that appear in the termination set. It also calculates a priority value for each sprite, based on how many interaction rules it occurs in. The level generator utilizes the information provided by the *GameAnalyzer* to generate a well-formed level. Figure 3 summarizes the core steps done by this approach. The generation procedure is divided into four core steps with a pre-processing step and a post-processing step:

1. **(Pre-processing) Calculate Cover Percentages:** This step calculates the percentage of tiles in the generated level that should be covered with sprites. It also calculates that percentage for each different sprite category. The total cover percentage is directly proportional to the number of collectible sprites, and it is inversely proportional to the number of harmful sprites and the number of sprites that are created by other sprites. All categories have a percentage directly proportional to the sum of the priority values of each sprite in each category.

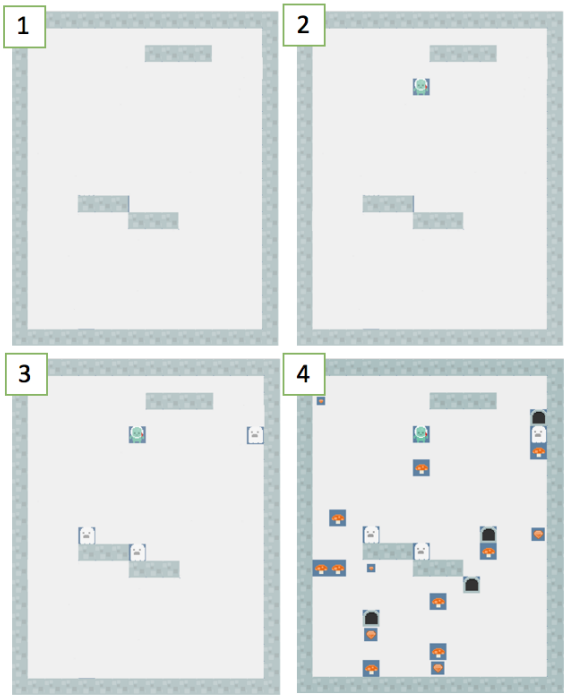


**Figure 3: Steps applied in the Constructive generator for Pacman: (1) Build a Level Layout (2) Add an Avatar Sprite (3) Add Harmful Sprites (4) Add Collectible and Other Sprites**

2. **Build a Level Layout:** This step only takes place when there are solid sprites. It picks a random solid sprite and surrounds the level with that sprite. Based on the calculated solid percentage it fills the internal level with solid sprites that are connected to each other without blocking any area.

3. **Add an Avatar Sprite:** This step places a randomly picked avatar sprite to a random free location.

4. **Add Harmful Sprites:** This step adds harmful sprites to the game based on the calculated harmful percentage. If the harmful sprite is a moving sprite, the generator chooses a free location away from the avatar sprite but if the sprite is a static sprite, the generator chooses any random free location.

5. **Add Collectible and Other Sprites:** This step places randomly picked sprites to randomly free locations. The number of added sprites depends on their cover percentages.

6. **(Post-processing) Fix Goal Sprites:** This step makes sure that the number of goal sprites are greater than the number specified in the termination set for that sprite type; sprites are added until this is the case.

## 4.3 Search-based Level Generator

This is a search-based level generator based on the Feasible Infeasible 2 Population Genetic Algorithm (FI2Pop) [14]. FI2Pop is a genetic algorithm which uses 2 populations at same time one for feasible chromosomes and the other

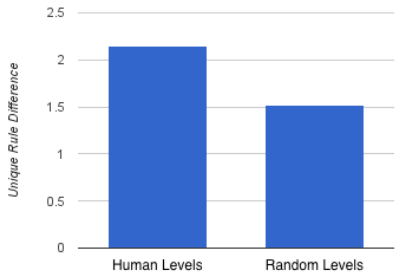


**Figure 4: Average number of unique rules used by AdrienCTX when playing human-designed levels and randomly generated levels for all VGDL games**

for infeasible chromosomes. The feasible population tries to improve the fitness of the overall chromosomes, while the infeasible population tries to decrease the number of chromosomes violating the problem constraints. Each population evolve on its own, where the children can transfer between the two population. The initial population is generated using the Constructive level generator described in Section 4.2. The levels are represented as a 2D array of tiles. Each tile consists of an array of strings representing all sprites at that tile. The generator uses one-point crossover which swaps the 2 chromosomes around a random tile. For mutation it uses 3 different operators:

- **Create:** creates a random sprite to any random tile position.
- **Destroy:** clears all sprites from a random tile position.
- **Swap:** swaps the sprites in two random tile positions.

The level generator uses an altered version of the *Adrienctx* controller for simulation-based fitness and constraint evaluation. *Adrienctx* is a well-performing controller, which won the 2014 edition of the GVGAI competition [12]. The controller has certain super-human skills (i.e. fast reaction time), and it was therefore altered to make its playing style somewhat more human-like. This is achieved through two modifications: adding action repetitions so that the controller has a tendency to repeat the last action for few time steps, and adding *NIL* repetition so that the controller has tendency to add *NIL* values between changing actions. These modifications make sure that the controller cannot handle situations which require extremely fast reactions, which in turn discourages the generation of levels that include such situations.

Some of the fitness and constraint evaluations are compared with the *OneStepLookAhead* or *DoNothing* players. The *OneStepLookAhead* player plays by greedily choosing among the immediate next actions. The *DoNothing* player simply applies the *NIL* action. Both players play for the same amount of steps as the altered *Adrienctx* controller. The feasible population is subjected to two different heuristic functions:

- **Score Difference Fitness:** difference between score achieved by *Adrienctx* and the best score achieved by *OneStepLookAhead* (over 50 runs), as suggested by Nielsen et al. [15]. That papers suggests that the relative difference between algorithms can be used to differentiate between well-designed and badly designed games. We compared with the score achieved by *OneStepLookAhead*, a weak player, to make sure to generate levels that require more skill to get a high score.
- **Unique Rule Fitness:** the number of unique events that happened in the level due to the avatar or any sprite spawned by it. We hypothesized that well-designed levels try to utilize almost all the different game rules while badly designed levels don't. To investigate this hypothesis, we played all levels of the publicly available VGDL games using *Adrienctx* and calculated the number of unique rules used; we then compared this to the number of unique rules used when playing randomly generated levels using the same algorithm. Figure 4 shows that well-designed levels tends to have a higher average of unique rules than random designed levels.

The fitness functions are treated as an average value of both the *Score Difference Fitness* and the *Unique Rule Fitness* as show in equation 1.

$$f_{feasible} = \frac{f_{score} + f_{rule}}{2} \quad (1)$$

where $f_{score}$ is the *Score Difference Fitness* and $f_{rule}$ is the *Unique Rule Fitness*. The Infeasible population is subject to seven different constraints:

- **Avatar Number:** Each level must have one avatar.
- **Sprite Number:** Each level must have at least one of each sprite which is not spawned by other sprites.
- **Goal Number:** The number of goal sprites must be greater than the limit of the termination rule associated with these sprites.
- **Cover Percentage:** Between 5% and 30% of level tiles must be covered by sprites.
- **Solution Length:** Levels must not be solved (by any of the players) in less than 200 steps.
- **Win:** The *Adrienctx* player must win the generated during evaluation.
- **Death:** The *DoNothing* player must not die for at least 40 steps over 50 different runs. Also it must not win after the same amount of time steps as *Adrienctx*.

These constraints might be used to disqualify some of submitted level generators if the competition attracts more entries than can be judged manually.

# 5. PILOT STUDY

We performed a small pilot study to test the performance of the proposed algorithms. The study used three different games, which are all VGDL remakes of (parts of) well-known games.

- **Frogs:** Port of *Frogger*. The aim of the game is to cross a street and reach the exit sign without getting hit by a car or drowning in the water.
- **PacMan:** Port of *Pac-Man*. The aim of the game is to collect all pills without getting caught by chasing ghosts. The player can eat one ghost each time he eats a power pill.
- **Zelda:** Port of the dungeon system of *The Legend of Zelda*. The aim is to get the key and get to the exit without getting killed by monsters. Points are scored by killing monsters using the sword.

The study did not incorporate any puzzle games because most existing agents do not play puzzle games well. Each generator supplies five levels for each game, and is allowed up to five hours to generated each level. Figure 5 shows an example level for *Zelda* from each generator.

The generated levels were tested by a group of human players. For that purpose, we created a program to help the players to understand the purpose of the study and get them familiar with the games. After that, the system picked two random levels from any of the generators and showed them to the human player. After playing both of these levels, the system showed a poll question as shown in Figure 6 where the player indicated which level they preferred, if both levels are equally preferred, or if none of them was preferred. We use preference indication (ranking) rather than a rating scheme such as Likert scales and rankings have been shown to be more consistent and reliable [16]. Rating systems violate two basic assumptions (ratings are ordinal data and they are not linear). On the other hand, ranking systems treat data as ordinal data while minimizing subjectivity bias. Data was collected from 25 players where each player played five pairs of games on average. The preference data was submitted automatically to a database.

Table 1 shows the results of our pilot study for all the three level generators. The *Equal* and *Neither* answer to the survey are discarded in this study, only the preference choices are taken into account. For this study we used three two-tailed binomial tests to test three null hypotheses about our generators, namely that there are no differences in preference between constructive and search-based, between random and search-based and between random and constructive.

From the preference numbers in the table its clear that Search-Based is better than both the Constructive and the Random, but the Constructive is voted far less than random which is not expected. The calculated p-values support this conclusion, as they reject only two null hypotheses (Search-Based and Random; Search-Based and Constructive) while showing no significant difference between the Constructive and Random level generators.

# 6. DISCUSSION

We had expected the search-based generator to produce levels that would be significantly better (more often preferred) than the Constructive generator; this was borne out in our pilot study. The main reason for this is likely the simulation-based evaluation function including constraints which ensured having a good playable level. In particular, using an altered version of *Adrienctx* ensured that the levels would not have too many enemies too close together, making the level playable for humans operating on human timescales.

The same cannot be said about the Constructive versus Random generator levels. The Random generator was pre-

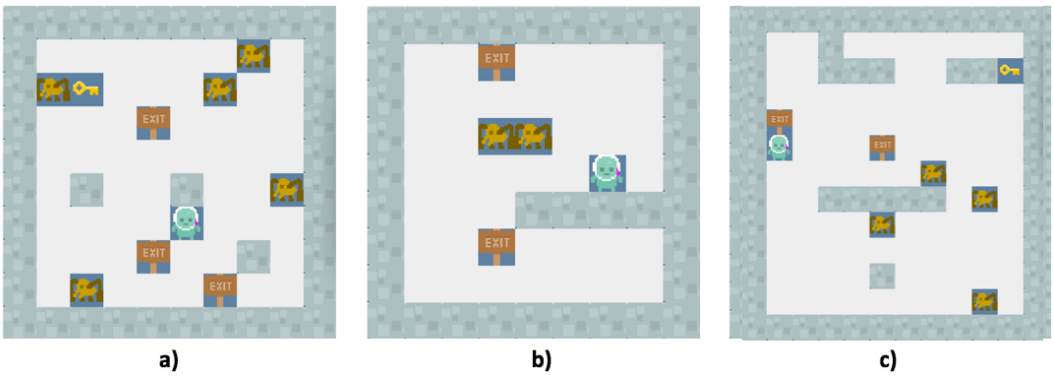


**Figure 5: Examples of generated levels from all the algorithms for Zelda: (a) Random Level Generator (b) Constructive Level Generator (c) Search-Based Level Generator**

| | Preferred | Non-preferred | Total | Binomial p-value |
|---|---|---|---|---|
| Search-Based vs Constructive | 23 | 12 | 35 | 0.0447 |
| Search-Based vs Random | 21 | 10 | 31 | 0.0354 |
| Constructive vs Random | 17 | 24 | 41 | 0.8945 |

**Table 1: This table shows players' preferences between different generators, aggregated over all three games.**

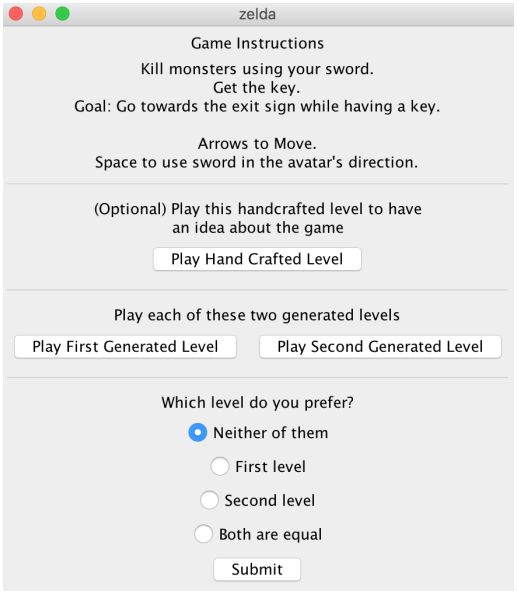


**Figure 6: Poll question comparing quality of the generated levels.**

ferred marginally more often than the Constructive generator. While the effect is not statistically significant, the likely main reason for any such effect is that the Constructive algorithm does not guarantee that each game object is generated at least once like in the case of the Random generator. For example the Zelda level in Figure 5 looks better than the random level but since it does not include a key, people tend to prefer the random generator over the Constructive generator. Not having this constraint gives more flexibility to Constructive generator over Random generator. For example: In almost all VGDL games there is multiple different kind of enemies where they are used to construct different levels. For example in Zelda, you can have slow enemies, normal enemies, and fast enemies. You don't need to place one of each type of enemy in the generated levels. Based on this fact, the Constructive generator tries to generate at least one sprite of each different category.

General Level Generation is a very hard problem to tackle. In order to generate playable levels for any game, we need a language to describe all different kind of games. The only languages that could represent all games would be Turing-complete programming languages. However, programming languages are not a good fit to describe games as they are simply too general and can be used to describe any kind of software (not only videogames); a random string in a standard programming language would not be likely to be a functioning game, or even a functioning program at all. This was a main reason for the development of Video Game Description Language (VGDL). VGDL is a description language that is readable, simple and can be used to describe a subset of different kind of videogames (real time 2D games). Although VGDL looks limited from outside, its can be used to describe different genres (such as puzzle, arcade, and shooter games). Even if the language is limited, the range of possible contributions is very large. For example Ludi language was developed by Browne and Marie [17] is only used to describe Combinatorial games, they managed to evolve very interest-

ing board game (Yavalath) which is ranked in the 100 top best abstract games by BoardGameGeek [18]. Following the same steps, the GVG-LG competition can be considered as a first step towards solving the general level generation problem. It can also be seen as a step towards solving the great general video game generation problem, as level generation is part of complete game generation.

# 7. CONCLUSION AND FUTURE WORK

This paper defined the general level generation problem and proposed the GVG-LG framework as a benchmark for it. This framework will be used for the Level Generation track of the GVG-AI competition. It also introduced three different level generators: Random, Constructive, and Search-Based, and a small pilot study that compares the levels produced by these generators.

We had conjectured that the search-based level generator would be better than the constructive level generator, which in turn would be better than the random level generator. Although the human players were unable to distinguish between the Constructive and Random generators, our Search-Based generator exceeded both of the Constructive and Random generator. It would be interesting to explore whether we could improve Constructive generator by adding some constrains to ensure playability.

Since this is a new field of study, there is not much previous work done on general level generation, which means there is much to explore. Our immediate priority is to improve our search-based level generator. In particular, we will seek to better understand the effect of each module and try to enhance our fitness evaluation so that it is more predictive of perceived level quality, which includes improving the automated player.

Beside doing different studies for the generators, we plan to test the GVG-LG framework with more people and get further feedback about the interface. This feedback will help in organizing the Level Generation track of the GVG-AI Competition.

## Acknowledgement

Thanks to Tiago Machado for help with running experiments and user study.

# 8. REFERENCES

[1] Shaker, N., Togelius, J., Nelson, M.J.: Procedural content generation in games: A textbook and an overview of current research. Procedural Content Generation in Games: A Textbook and an Overview of Current Research (2015)

[2] Geek, B.G.: Rogue (video game). http://www.boardgamegeek.com/abstracts/browse/boardgame?sort=rank&rankobjecttype=family&rankobjectid=4666&rank=113#113 Accessed: April 2, 2016.

[3] Liapis, A., Yannakakis, G.N., Togelius, J.: Sentient sketchbook: Computer-aided game level authoring. In: FDG. (2013) 213–220

[4] Levine, J., Congdon, C.B., Ebner, M., Kendall, G., Lucas, S.M., Miikkulainen, R., Schaul, T., Thompson, T., Lucas, S.M., Mateas, M., et al.: General video game playing. Artificial and Computational Intelligence in Games **6** (2013) 77–83

[5] Ebner, M., Levine, J., Lucas, S.M., Schaul, T., Thompson, T., Togelius, J.: Towards a video game description language. (2013)

[6] Schaul, T.: A video game description language for model-based or interactive learning. In: Computational Intelligence in Games, IEEE (2013) 1–8

[7] Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the aaai competition. AI magazine **26**(2) (2005)

[8] Khalifa, A., Fayek, M.: Automatic puzzle level generation: A general approach using a description language. In: Computational Creativity and Games Workshop. (2015)

[9] Lavelle, S.: Puzzle script. http://www.puzzlescript.net/ Accessed: November 4, 2015.

[10] Lim, C.U., Harrell, D.F.: An approach to general videogame evaluation and automatic generation using a description language. In: Computational Intelligence and Games (CIG), 2014 IEEE Conference on, IEEE (2014) 1–8

[11] Neufeld, X., Mostaghim, S., Perez-Liebana, D.: Procedural level generation with answer set programming for general video game playing. In: Computer Science and Electronic Engineering Conference, IEEE (2015)

[12] Perez, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S., Couëtoux, A., Lee, J., Lim, C.U., Thompson, T.: The 2014 general video game playing competition. IEEE Transactions on Computational Intelligence and AI in Games (2015)

[13] Shaker, N., Togelius, J., Yannakakis, G.N., Weber, B., Shimizu, T., Hashiyama, T., Sorenson, N., Pasquier, P., Mawhorter, P., Takahashi, G., et al.: The 2010 mario ai championship: Level generation track. Computational Intelligence and AI in Games, IEEE Transactions on **3**(4) (2011) 332–347

[14] Kimbrough, S.O., Koehler, G.J., Lu, M., Wood, D.H.: On a feasible-infeasible two-population (fi-2pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. European Journal of Operational Research **190**(2) (2008) 310–327

[15] Nielsen, T.S., Barros, G.A., Togelius, J., Nelson, M.J.: General video game evaluation using relative algorithm performance profiles. In: Applications of Evolutionary Computation. Springer (2015) 369–380

[16] Yannakakis, G.N., Martínez, H.P.: Ratings are overrated! Frontiers in ICT **2** (2015) 13

[17] Browne, C., Maire, F.: Evolutionary game design. Computational Intelligence and AI in Games, IEEE Transactions on **2**(1) (2010) 1–16

[18] Wikipedia: Yavalath (board game). https://en.wikipedia.org/wiki/Rogue_(video_game) Accessed: November 3, 2015.