Marahel: A Language for Constructive Level Generation

Ahmed Khalifa and **Julian Togelius**
Tandon School of Engineering
New York University
Brooklyn, New York 11201
ahmed.khalifa@nyu.edu and julian@togelius.com

Abstract

Marahel is a language and framework for constructive generation of 2D tile-based game levels. It is developed with the dual aim of making it easier to build level generators for game developers, and to help solving the general level generation problem by creating a generator space that can be searched using evolution. We describe the different sections of the level generators, and show examples of generated maps from 5 different generators. We analyze their expressive range on three dimensions: percentage of empty space, number of isolated elements, and cell-wise entropy of empty space. The results show that generators that have starkly different output from each other can easily be defined in Marahel.

Introduction

Game level design is one of the most common domains for procedural content generation. There are many different methods for level generation, some ad-hoc and unique to particular games, others built on more principled algorithms and generalizable to great variety of games (Shaker, Togelius, and Nelson 2014).

Given the relatively bounded domain, it should be possible to apply algorithms across games, in order to compare them. However, doing so typically requires reimplementing each algorithm in the context of a particular game, such as Super Mario Bros (Horn et al. 2014). It should also in principle be possible to mix and match these algorithms so as to search (manually or automatically) the space of level generators for generators that deliver desired aesthetics or work with particular constraints. To make this possibility a reality, we need a unified framework for level generators.

In this work-in-progress paper, we present an early version of *Marahel*[1], a language and framework for constructive 2D tile-based level generation. The language is an attempt to formalize the principles behind a number of popular algorithms that can be used for constructive (i.e. not based on generate-and-test) level generation for tile-based games so that they can easily be recombined.

Any valid *Marahel* string constitutes a specification for a level generator, which when interpreted by the *Marahel* software can produce 2D tile-based levels. Not all valid *Marahel* scripts will produce usable levels for all games, because game mechanics play an important role in defining the space of plausible levels. For example: if the player is able to dig through walls, it is okay to have isolated areas. The user of *Marahel* (a human and/or an algorithm) must make sure that the script is not only a valid script but also a suitable one, i.e. it fits the requirements of the current game.

One of the key motivations for the development of *Marahel* is the General Video Game Level Generation challenge, which is to develop level generators that work for any game within a given domain (Khalifa et al. 2016). Another key motivation is to simplify the development process of level generators easily and make them accessible to developers of all stripes through an open API. A third motivation is to understand the design space of level generators through formalizing their design space.

Background

Methods for level generation can be divided into several categories. One common division is between search-based level generation, constraint-based level generation, and constructive level generation (Shaker, Togelius, and Nelson 2014).

In search-based level generation (Togelius et al. 2011), a search algorithm such as a genetic algorithm is utilized to find a level. The levels are tested using a fitness function that measures the quality of the levels. The fitness function can be anything from measuring the connectivity of the level to an AI agent playing the level and measuring its difficulty.

Constraint-based generators (Smith and Mateas 2011) use constraint based solvers to find a good map. In such methods, the user defines what are the feature required in the generated map and the solver tries to fit all those requirements.

Constructive level generation methods (Shaker et al. 2016) are widely used in the videogame industry due to many algorithms being very fast, and also relatively easy to implement and debug. These techniques has been used in videogames since the early days. For example, *Rogue*[2] generates a new dungeon for every playthrough. While constructive generation methods differ widely among each other, the defining feature of constructive generation is that

---

Copyright © 2017, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1] *Marahel* means *levels* in Arabic.

[2] https://en.wikipedia.org/wiki/Rogue_(video_game)

there is no re-generation of the output based on testing; generation happens only once. Due to this limitation, the algorithm should guarantee that the generated levels have the required features during construction. Not all algorithms can guarantee playable levels 100% of the time, so developers use repair techniques to fix the generated content. Since constructive algorithm are fast, developers sometimes wraps the algorithm in a generate and test algorithm where it keeps generating levels until a suitable one is found.

Depending on the requirements of the particular game and the desired type of results, different constructive methods are used such as template-based generation, binary space partitioning, cellular automata, diggers, and etc. Below we describe some constructive level generation techniques.

Template-based level generation uses hand authored content to generate the level. The algorithm combines different authored pieces that fit together according to certain constraints. In some cases, the algorithm alters the generated level by adding noise to it. You can find this technique used in *Spelunky* (Yu 2016) and *Binding of Isaac* (McMillen 2011) among other games.

Binary space partitioning generates levels by partitioning the space, either vertically or horizontally. The algorithm partitions the space until it reaches a certain number of regions then it connects these regions with hallways. For example: Splitter (Leemoor 2013) a game created for 7 Day Roguelike (7DRL) competition uses binary space partitioning to generate the game map.

Cellular automata are a mathematical technique of great generality (Wolfram and others 1986) which can also be used for level generation (Johnson, Yannakakis, and Togelius 2010). For example, this technique can be used to generate maps that mimic natural caves. In this method, the map is filled randomly. Each location is then updated based on its surrounding values. As this method does not generate fully connected maps, an A^* algorithm can be used to connect the isolated areas, or by filling the smaller isolated areas by solid. For example: Galak Z (Aikman 2014) uses cellular automata to generate separate rooms. Tomb of Tomeria (Cook and Colton 2016) not only uses cellular automata to generate the whole level but also utilizes it as a game mechanic.

The various digger algorithms are agent-based methods. Typically, the map starts as all solid. The agent moves around randomly changing all the locations it passes over to empty which ensures the map's connectivity. At each time step, the agent has a probability to spawn a room (a random size area of empty locations). Nuclear Throne (Ismail 2013) is an example of a game that uses an agent-based algorithm to generate the map. Other techniques such as grammar based level generation (Van der Linden, Lopes, and Bidarra 2013; Dormans 2010) are out of the scope of the current version of *Marahel*.

As far as we know, there is no prior work in defining a level generator description language. However, game description languages are an active research topic. Game description languages are descriptive languages that can define a group of games. For example: there are game description language for board games (Love et al. 2008; Browne and Maire 2010), card games (Font et al. 2013), video games (Ebner et al. 2013), puzzle games (Lavelle 2013), strategy games (Mahlmann, Togelius, and Yannakakis 2011), first person dungeon crawlers (Farbs 2017), and etc. Having a specific language help to decrease the search space for new games making it easier to find good games such as Yavalath (Browne 2011). Having a level generation description language will be the first step towards formalizing the space of level generators that can be searched manually/automatically to find new techniques, have a deeper understanding about level design, and etc.

Marahel

Marahel approaches level generation as a description language that describes the steps of the generation process instead of the required level. By comparing level generation techniques to programming paradigms, we can see that techniques such as constraint-based generation follows a declarative programming paradigm, while other techniques such as constructive generation follows an imperative programming paradigm.

Answer Set Programming (Smith and Mateas 2011) can be described as a language that follows a declarative programming paradigm where the user define the features required in the output and the system finds a solution for it. Following this logic, *Marahel* can be described as a language that follows an imperative programming paradigm (like C++) where the user defines the steps required by the generator to change the current map.

A *Marahel* script constitutes a 2D tile-based level generator. Each script consists of 5 section: Metadata, Entities, Neighborhoods, Regions, and Explorers. The first 3 sections (Metadata, Entities and Neighborhoods) defines different data required during the generation process, while the rest (Regions and Explorers) defines the steps of the generation. Comparing these sections to an imperative programming languages, the first 3 sections will be similar to the input data and constant values required/used by the program, while the last 2 sections are the actual program itself.

The following five steps are taken by the *Marahel* when implementing a generator description:

1. Parse the first 3 sections (Metadata, Entities and Neighborhoods) and save them for later usage.
2. Define a 2D array of the dimension specified in the previous step and initialize it with "unknown".
3. Use the algorithm defined in the Regions section to divide the map into several areas.
4. Apply all the defined explorers sequentially to modify the 2D array based on their rules.
5. Return the 2D array to the user.

The *Marahel* language can be described as a context free grammar. Grammar 1 shows the full definition of the current version of *Marahel*. Terminals in *Marahel* are a list with the current supported features in the system. Adding a new terminal to the list extends *Marahel*'s capabilities. For example: if a new divider algorithm is required, we only need to add a new terminal to "<divider>".

Grammar 1: *Marahel* language as context free grammar

⟨*script*⟩ ::= ⟨*metadata*⟩ ⟨*entities*⟩ ⟨*neighborhoods*⟩ ⟨*regions*⟩ ⟨*explorers*⟩

⟨*metadata*⟩ ::= ⟨*generalInfo*⟩ ⟨*metadata*⟩ | ⟨*generalInfo*⟩

⟨*generalInfo*⟩ ::= ’minDimension’
| ’maxDimension’ | ’dimension’

⟨*entities*⟩ ::= ’entityName’ ⟨*entities*⟩ | ’entityName’

⟨*neighborhoods*⟩ ::= ⟨*neighbor*⟩ ⟨*neighborhoods*⟩ | ϵ

⟨*neighbor*⟩ ::= ’neighborName’ ’relativePoints’

⟨*regions*⟩ ::= ’numOfRegions’ ⟨*divider*⟩

⟨*divider*⟩ ::= ’equal’ | ’bsp’ | ’sampling’

⟨*explorers*⟩ ::= ⟨*explorer*⟩ ⟨*explorers*⟩ | ϵ

⟨*explorer*⟩ ::= ⟨*appliedRegion*⟩ ⟨*generalParam*⟩ ⟨*expType*⟩ ⟨*rules*⟩

⟨*appliedRegion*⟩ ::= ’map’ | ’all’ | ’some’ | ’specific’

⟨*generalParam*⟩ ::= ⟨*param*⟩ ⟨*generalParam*⟩ | ϵ

⟨*param*⟩ ::= ’borderSize’ | ’borderHandling’
| ’replacingTech’ | ’iterations’

⟨*expType*⟩ ::= ’random’ | ’sequential’
| ’agent’ | ’connector’

⟨*rules*⟩ ::= ⟨*rule*⟩ ⟨*rules*⟩ | ⟨*rule*⟩

⟨*rule*⟩ ::= ⟨*conditions*⟩ ⟨*executers*⟩

⟨*conditions*⟩ ::= ⟨*cond*⟩ ⟨*conditions*⟩ | ϵ

⟨*executors*⟩ ::= ⟨*execut*⟩ ⟨*executors*⟩ | ⟨*execut*⟩

⟨*cond*⟩ ::= ⟨*bioperator*⟩ ⟨*estimator*⟩ ⟨*estimator*⟩
| ⟨*unioperator*⟩ ⟨*estimator*⟩
| ⟨*operator*⟩

⟨*estimator*⟩ ::= ’constant’ | ’random’ | ’noise’
| ’entityEstimator’ | ’neighborhoodEstimator’
| ’distanceEstimator’

⟨*bioperator*⟩ ::= ’equal’ | ’notEqual’ | ’greater’ | ’less’

⟨*unioperator*⟩ ::= ’isEven’ | ’isOdd’ | ’isSingular’

⟨*operator*⟩ ::= ’isConnected’

⟨*execut*⟩ ::= ’neighborhoodExecuter’

Listing 1 shows an example of a full *Marahel* script compatible with the current Javascript implementation[3]. This script generates dungeons that consist of 7 connected rooms of different size. Below we describe the different sections of a Marahel script.

Metadata

The Metadata section contains all the information that is related to the whole generation process. In the current implementation, *Marahel* supports only the minimum and the maximum dimensions of the generated map.

Listing 2 shows an example of metadata section. In this example, the level generator will always generate maps of size between “40x30” and “60x45”.

Listing 2: Example of the entities section.

```
metadata:{
    minDimension:"40x30",
    maxDimension:"60x45"
}
```

Entities

The Entities section contains a list of all the names of the entities that can appear in the final generated map, and is the “ontology” of the levels. Entities are the base unit of any generated level. A level is a 2D array of entities.

Listing 3 shows an example of the entities section. In this example, we have two different entities: “solid” and “empty”. This level generator is only able to generate maps that contains any of these entities.

Listing 3: Example of the entities section.

```
entities:[
  solid,
  empty
]
```

Neighborhoods

The neighborhoods section is a section that contains a list of different neighborhoods. A neighborhood is an entity that defines relations between multiple locations and a center one. Neighborhoods can be represented using various methods such as a list of points, a 2D arrays of numbers, etc. For example: “[(1,1), (0,-3)]” shows a list version of a neighborhood where (1,1) and (0,-3) are relative points. To calculate the relative locations from a certain point such as (2,2) using this neighborhood, you need to add each point separately to get (3,3) and (2,-1) as a result.

In the current implementation, *Marahel* uses a 2D array of numbers to specify these relative points. Each neighborhood contains a name and a 2D array of numbers. The numbers indicate the relation between their locations in the matrix with respect to a certain location.

Listing 4 shows two neighborhoods: *all* and *plus*. Each *1* in the array tells the generator to use that location relative to

---

[3] https://github.com/amidos2006/Marahel

Listing 1: An example of a full generator.

```
{
  metadata:{
    minDimension:"40x30",
    maxDimension:"60x45"
  },
  entities:[
    "empty",
    "solid"
  ],
  neighborhoods:{
    all: ["111",
          "131",
          "111"],
    plus:["010",
          "121,
          "010"]
  },
  regions:{
    type:"bsp",
    numberOfRegions:7,
    parameters:{
      min:"8x8",
      max:"15x15"
    }
  },
  explorers:[
    {
      type:"sequential",
      region:{name:"map"},
      parameters: {iterations:1},
      rules:["self(any)->self(solid)"]
    },
    {
      type:"sequential",
      region:{name:"all", border:1},
      parameters: {iterations:1},
      rules:["self(any)->self(empty)"]
    },
    {
      type:"connector",
      region:{name:"map"},
      parameters:{
        type:"short",
        directions:["plus"],
        entities:["empty"]
      },
      rules:["self(solid)->self(empty)"
           ]
    }
  ]
}
```

Listing 4: Example of the neighborhoods section.

```
neighborhoods:{
  all:  ["111",
         "131",
         "111"],
  plus: ["010",
         "121",
         "010"]
}
```

the location of the value *2* or *3*. The value *3* is same as *2* but it tells the generator that this location is a relative location too. The *all* neighborhood can be represented as the following list of relative points "[(-1,-1), (-1,0), (-1,1), (0,1), (1,1), (1,0), (1,-1), (0,-1), (0,0)]" while *plus* neighborhood can be represented as "[(1,0), (-1,0), (0,1), (0,-1)]".

Regions

The Regions section defines the algorithm that is used to divide the map into several regions. Regions are portions of the generated map that are generated using the selected algorithm. In each *Marahel* script, The user selects the "divider algorithm" and the "number of regions". *Marahel* currently supports three different algorithms to generate rectangular regions:

- **Equal:** divides the map into equal sized portions using a grid then selects randomly some/all regions based on the required "number of regions".
- **Binary Space Partitioning:** divides the map into different size region by splitting each region either vertically or horizontally. The algorithm keeps splitting each region till the termination conditions are met. After that, it selects randomly some/all regions based on the required "number of regions".
- **Sampling:** adds regions to the generated map that do not intersect with the previous ones. The algorithm continues until the required "number of regions" is met.

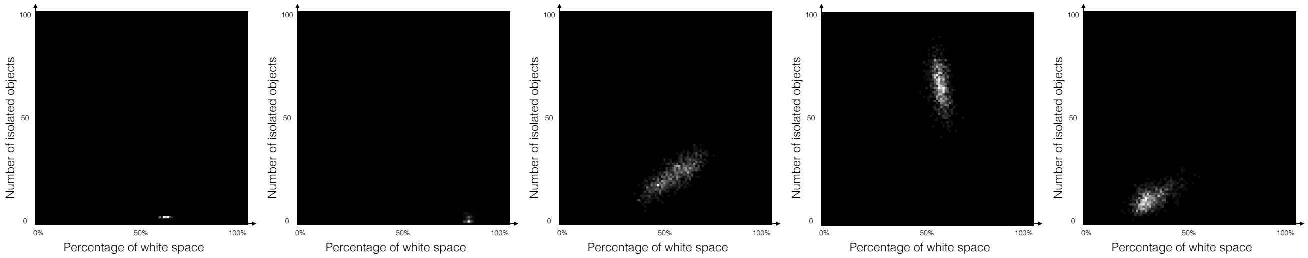
Listing 5 shows the regions section from a generator. The algorithm splits all the regions that are bigger than 15x15 while making sure the resulted regions are bigger than 8x8. In the end, the algorithm chooses 7 random regions from the output regions.

Listing 5: Example of the regions section using binary space partitioning.

```
regions:{
  type:"bsp",
  numberOfRegions:7,
  parameters:{min:"8x8",max:"15x15"}
}
```

Explorers

Explorers are the core of the generation process. Explorers use an algorithm to visit different tiles on a defined region



(a) (b) (c) (d) (e)

Figure 1: Expressive range of the different generators: (a) uniform map generator, (b) nonuniform map generator, (c) digger map generator, (d) cave map generator, and (e) mine map generator.

of the map. At each step of the algorithm, the explorer is at a certain location(s) where it will apply the defined rules. Explorers and rules together define how the system modifies the generated map. A *Marahel* script can have more than one explorer where they are applied sequentially. Explorers consist of 3 main parts:

- **Type and Parameters:** specifies the type of the explorer and its parameters. Different supported types will be discussed later in this section. Also, It specifies some general parameters such as “number of repetition” which allows the system to repeat this specific explorer any number of times.
- **Applied Region:** selects the area of the map that is affected by the explorer. The user can select either the whole map, all/some regions generated by the region divider, or manual defined regions. Any tile outside the applied region(s) won’t be affected by the explorer.
- **Rules:** is a list of conditional rules that change the generated map. *Marahel* goes over the list in order until the first rule is satisfied; that rule will then be applied.

$$rule\colon condition, ..., condition \rightarrow executer, ..., executer \tag{1}$$

Equation 1 shows the structure of rules in *Marahel*. Rules in *Marahel* consists of two sides: Left hand side and Right hand side. The left hand side is a group of conditions that need to be satisfied before applying the right hand side. If any condition fails, the rule fails.

$$condition\colon estimator < op > estimator \tag{2}$$

Conditions can be anything that returns either true or false. In the current implementation of *Marahel*, only comparative conditions exists (bi-operator conditions). Equation 2 shows the structure of the comparitive conditions. $<$op$>$ is either greater than ($>$), less than ($<$), equal ($==$), or not equal ($!=$). Estimators are functions that return a numerical value, it can be anything from a constant number to a complex equation. Estimators, in the current implementation of *Marahel*, are either a neighborhood estimator, a distance estimator, a number estimator, or an entity estimator.

The neighborhood estimator calculates the number of a certain entity/entities around the current location using the relative points defined by a specified neighborhood. The distance estimator calculates either the maximum, average, or minimum distance between the current location and a specified entity/entities. The number estimator is either a fixed number, a random number between 0 and 1, or a perlin noise value between 0 and 1 for the current location either in the applied region or in the whole map. The entity estimator gets the total number of a specified entity either in the applied region or in the whole map.

Executors are simpler than conditions. Executors modifies locations on the map relative to the current location. In the current implementation of *Marahel*, it supports one type of executor where it changes the current location and/or the surrounding locations (using the relative points of a specified neighborhood) to a certain entity. If the executor have a list of entities, it will pick one of them at random.

Marahel currently supports four different types of explorers that are described in details in the following text.

Random: is an explorer that visits the tiles in a random order. The user can control the number of tiles to visit using its parameters.

Listing 6 shows an example of an random generator. This generator picks 20 random location in the map and changes them to “enemy” entity only if they are “empty”.

Listing 6: Example of an automata generator.

```
{
  type:"random",
  region:{name:"map"},
  parameters:{numberOfTiles:20},
  rules:["self(empty) -> self(enemy)"]
}
```

Sequential: is an explorer that visits tiles in a sequential order. The user can define several parameters to control the explorer behavior such as, percentage of explored tiles as a value between 0 and 1, starting location as a value between 0 and 1, order of visiting tiles using neighborhood’s relative points.

Listing 7 shows an example of a sequential explorer. This explorer fills “all” the regions with “empty” entity while leaving 1 tile as border.

Listing 7: Example of an sequential generator.

```
{
  type:"sequential",
  region:{name:"all", border:1},
  parameters:{iterations:1},
  rules:["self(any) -> self(empty)"]
}
```

Agent: is an agent based explorer. *Marahel* spawns multiple agents inside the applied regions that are updated step by step. At each time step, *Marahel* updates all the living agents based on their parameters then applies the rules to modify their current location. The user can define several parameters to control the agent's behavior such as, the number of agents, the number of steps needed to change direction, their lifespan, and the possible directions as a list of neighborhoods. Each agent selects a direction (relative point) randomly from the array of directions.

Listing 8 shows an example of an agent generator. This generator spawns 3 agents that change direction every 10 steps to a random direction picked from the "plus" neighborhood. These agents have a lifespan of 150 step. At each step, the agents spawn either a single empty entity (70% of the time) or a 3x3 area of empty entities.

Listing 8: Example of an agent generator.

```
{
  type:"agent",
  region:{name:"map"},
  parameters: {number:3, change:10,
      lifespan:150, directions:["plus"]}
      ,
  rules:[
    "self(any),random<0.7 -> self(empty
       )",
    "self(any) -> all(empty)"
  ]
}
```

Connector: is a special type of agent. Connector uses an A^* algorithm to explore tiles between the isolated areas inside the applied region. The user can control the behavior of the agent using a set of parameters such as, the names of connected entities, the allowed directions using a list of neighborhoods (where *Marahel* pick a random relative point and use it as a possible movement), and the type of connection. The type of connection specifies the goal of the agent (heuristic function for the A^* algorithm). The current implementation supports shortest connections (minimize the distance between unconnected areas), random connections (randomly connect unconnected areas), and hub connection (use one area as a central hub and connect it to all the other areas).

Listing 9 shows an example of a connector generator. This generator tries to make sure that all "empty" entities are connected using a "plus" neighborhood. Connections are selected based on the shortest distance. Each location on the connection path will be set to an "empty" entity.

Listing 9: Example of a connector generator.

```
{
  type:"connector",
  region:{name:"map"},
  parameters:{type:"short", directions:
      ["plus"], entities:["empty"]},
  rules:["self(any) -> self(empty)"]
}
```

Results

In this section, we analyze different map generators made for a top down roguelike game[4] such as Desktop Dungeon[5]. This analysis is done to show the ability of *Marahel* to describe different level generators with different characteristics.

We designed 5 different generators in the Marahel language, intended to generate dungeons in different styles[6]:

- **Uniform map generator:** divides the map using equal divider and fill each region with "empty" then connecting them.
- **Nonuniform map generator:** similar to the previous generator but it uses bsp divider instead of equal divider.
- **Digger map generator:** uses multiple agent explorer to generate the map by adding a single "empty" entity 70% of the time or a 3x3 "empty" entities the rest of the time.
- **Cave map generator:** uses sequential explorer with rules similar to cellular automata to generate cave-like maps.
- **Mine map generator:** is similar to the nonuniform map generator but it modifies each region using the cave map generator.

We generated 1000 maps from each of these generators. We calculated the percentage of "empty" entities (white space) and the number of isolated objects for each generated map. We used these two values to showcase the expressive range (Smith and Whitehead 2010) of each of these generators.

Figure 1a shows the expressive range of the uniform map generator. There is a small white area near the bottom of the y-axis which is similar to the expressive range of the nonuniform map generator in 1b. The reason is both of these generators generated same number of regions of either different sizes or different locations. This forces all the generated maps to have similar white space percentage and a small number of isolated areas. Figure 1c shows the expressive range of the digger map generator. The digger map generator have more isolated areas than the previous two generator. Figure 1d shows the expressive range of the cave map generator. This generator has the highest number of isolated

---

[4] It is a game genre that defines games similar to rogue.

[5] http://www.desktopdungeons.net

[6] Check http://akhalifa.com/marahel/paper/scripts.zip for their full description.

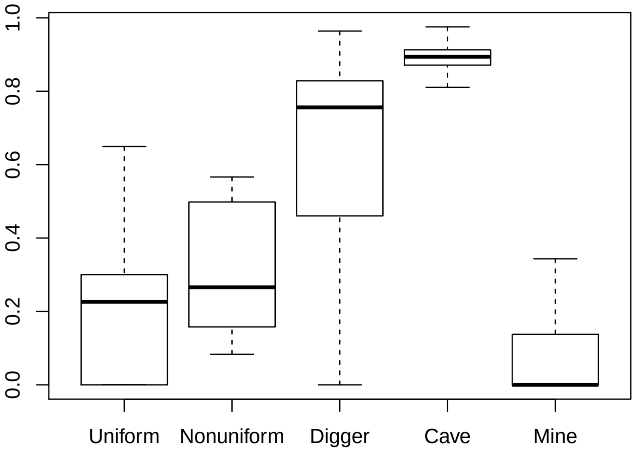


Figure 2: The entropy of empty entity over the generated maps. The black lines shows the median of the entropy for the generated maps. The box shows the standard deviation, while the horizontal lines shows the maximum and minimum values.

regions. Figure 1e shows the expressive range of the mine map generator. The mine generator have a small white space percentage and a small number of isolated areas.

Another metric, we used is measuring the cell-wise entropy of empty entity over the generated map. Each generated map is divided into 25 regions (5x5) where the entropy of the empty entity (white space) in that portion is calculated. We take the average over all 25 regions. The entropy of the current generated map is calculated using equation 3.

$$\overline{H(X)} = \frac{1}{n}\sum_{i=1}^{n}\sum_{x\in X} -P_i(x)\log P_i(x) \tag{3}$$

where n is the number of regions (25 in our case), $P_i(x)$ is the probability of empty/solid entity in the region i.

Figure 2 shows the distribution of the entropy over a 1000 generated maps from the 5 different generators. The cave map generator has a high entropy with a small standard deviation which indicates that its generated maps have an empty entity percentage around 50% in each region. On the other hand, the equal, binary space partitioning, and mine map generators have the lowest entropy values with a higher standard deviation. This low entropy reflect the presence of big areas of all empty or all solid regions. The digger generator has a high entropy values due to the high stochasticity in the agents that digs the map.

Figure 3 shows 4 generated maps using more than two entities. These maps feature 3 new entities: player, enemy, and treasure. The green dot is the "player" entity and it is spawned only in locations that are surrounded with "empty" entities from all the directions. Red dots are "enemy" entities and they are generated at any location in the map but with a higher chance to select locations that block hallways. Yellow dots are "treasure" entities, they are generated only at corners and hallways.

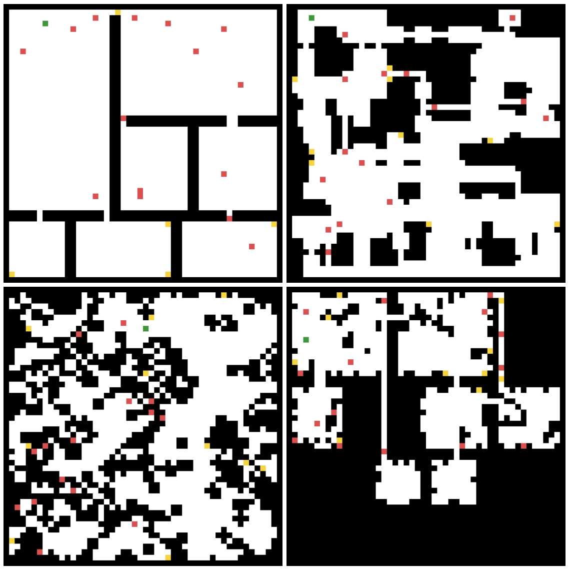


Figure 3: Different generated maps with more entities. The green dot is the player, red dots are enemies, and yellow dots are treasure chests. The used generators, in order from top left to bottom right, are: nonuniform map generator, digger map generator, cave map generator, and mine map generator.

Conclusions

This paper introduced *Marahel*, a description language for 2D tile-based constructive level generators. We used *Marahel* to define five generators and plotted their expressive range and their entropy. The results shows how different these generators are from each other in terms of isolated areas, open space percentage, and open space entropy.

We believe the results show clearly that *Marahel* can be used to describe generators with very different generation styles, both qualitatively and quantitatively. While *Marahel* can also be used to generate maps with more entities than just solid and empty, for example potions, traps, treasures and etc, such generation is not discussed in this initial paper.

For future work, we want to integrate our work with Danesh (Cook, Gow, and Colton 2016) to allow for an easier visualization for the generators and their expressive range. This will help the users of the system to easily debug their generators. We aim to have a user interface to make it easier to game/level designers to write *Marahel* scripts. We will also create an implementation of *Marahel* in Java, and possibly in C# and Python, to complement the current JavaScript implementation. The Java implementation will be made to interface with the General Video Game AI framework, and be included with the GVGAI Level Generation Track software. This work is the first step towards finding a general level generator. One of the core ideas is to use a genetic evolution to search the space of generators defined by *Marahel* that fits specific games.

References

Aikman, Z. 2014. Generating procedural dungeons in galak z. `https://www.youtube.com/watch?v=ySTpjT6JYFU`. [Online; accessed 28-July-2017].

Browne, C., and Maire, F. 2010. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*.

Browne, C. 2011. Yavalath. `http://www.cameronius.com/games/yavalath/`. [Online; accessed 26-July-2017].

Cook, M., and Colton, S. 2016. Towards procedural generation as gameplay: Clay and tombs of tomeria.

Cook, M.; Gow, J.; and Colton, S. 2016. Danesh: Helping bridge the gap between procedural generators and their output. In *PCG Workshop*. ACM.

Dormans, J. 2010. Adventures in level design: generating missions and spaces for action adventure games. In *PCG Workshop*. ACM.

Ebner, M.; Levine, J.; Lucas, S. M.; Schaul, T.; Thompson, T.; and Togelius, J. 2013. Towards a video game description language.

Farbs. 2017. Dungeon script. `http://dungeonscript.farbs.org/`. [Online; accessed 26-July-2017].

Font, J. M.; Mahlmann, T.; Manrique, D.; and Togelius, J. 2013. A card game description language. In *European Conference on the Applications of Evolutionary Computation*. Springer.

Horn, B.; Dahlskog, S.; Shaker, N.; Smith, G.; and Togelius, J. 2014. A comparative evaluation of procedural level generators in the mario ai framework. In *Foundation of Digital Games*. ACM.

Ismail, R. 2013. Random level generation in wasteland kings. `http://rami-ismail.squarespace.com/blog/2013/04/02/random-level-generation-in-wasteland-kings`. [Online; accessed 23-May-2017].

Johnson, L.; Yannakakis, G. N.; and Togelius, J. 2010. Cellular automata for real-time generation of infinite cave levels. In *PCG Workshop*. ACM.

Khalifa, A.; Perez-Liebana, D.; Lucas, S. M.; and Togelius, J. 2016. General video game level generation. In *GECCO Conference*. ACM.

Lavelle, S. 2013. Puzzlescript! `http://www.puzzlescript.net/`. [Online; accessed 26-July-2017].

Leemoor, G. 2013. 7drl: Splitter: Bsp dungeons. `https://pangoempire.wordpress.com/2013/03/09/7drl-splitter-bsp-dungeons/`. [Online; accessed 28-July-2017].

Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2008. General game playing: Game description language specification.

Mahlmann, T.; Togelius, J.; and Yannakakis, G. N. 2011. Towards procedural strategy game generation: Evolving complementary unit types. In *European Conference on the Applications of Evolutionary Computation*. Springer.

McMillen, E. 2011. Binding of Isaac Gameplay Explained. `http://edmundmcmillen.blogspot.com/2011/09/binding-of-isaac-gameplay-explained.html`. [Online; accessed 23-May-2017].

Shaker, N.; Liapis, A.; Togelius, J.; Lopes, R.; and Bidarra, R. 2016. Constructive generation methods for dungeons and levels. In *Procedural Content Generation in Games*. Springer.

Shaker, N.; Togelius, J.; and Nelson, M. 2014. *Procedural Content Generation In Games*. Springer.

Smith, A. M., and Mateas, M. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*.

Smith, G., and Whitehead, J. 2010. Analyzing the expressive range of a level generator. In *PCG Workshop*. ACM.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*.

Van der Linden, R.; Lopes, R.; and Bidarra, R. 2013. Designing procedurally generated levels. In *Artificial Intelligence in the Game Design Process Workshop*. AAAI.

Wolfram, S., et al. 1986. *Theory and applications of cellular automata*. World scientific Singapore.

Yu, D. 2016. *Spelunky*. Boss Fight Books.