# Automated Video Game World Map Synthesis by Model-Based Techniques

Boqi Chen
Dylan Havelock
Connor Plante
Michael Sukkarieh
McGill University
Montreal, Quebec, Canada
firstname.lastname@mail.mcgill.ca

Oszkár Semeráth
Budapest University of Technology and Economics, Deptartment of Measurement and Information Systems
Budapest, Hungary
semerath@mit.bme.hu

Dániel Varró
McGill University
Montreal, Quebec, Canada
daniel.varro@mcgill.ca

## ABSTRACT

World maps contribute a significant part of the interactivity and entertainment to modern video games. While large-scale industrial world map generation tools exist, their use usually implies a substantial learning curve, and the cost of licences restricts the accessibility of these tools to individual game developers.
In this paper, we introduce a world map generator for Unity-based games that exploits model-based techniques. After the game-specific concepts of the world map are captured and turned into a metamodel, the world map generation problem is first formulated as a consistent graph generation problem solved by a state-of-the-art model generator. This graph model is subsequently refined into a concrete world within the Unity game engine by (1) mapping the abstract graph elements into Unity game objects and (2) creating a height map based on user-defined properties with the Perlin Noise technique. **Demonstration video:** https://youtu.be/03BbD61EKpk

## CCS CONCEPTS

• **Software and its engineering** → **Software design engineering**; • **Theory of computation** → *Automated reasoning*; • **Applied computing** → **Computer games**.

## KEYWORDS

world map generation, graph generation, video games, model-based engineering

**ACM Reference Format:**
Boqi Chen, Dylan Havelock, Connor Plante, Michael Sukkarieh, Oszkár Semeráth, and Dániel Varró. 2020. Automated Video Game World Map Synthesis by Model-Based Techniques. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3417990.3422001


*MODELS '20, October 18–23, 2020, Montreal, Canada*

ACM ISBN 978-1-4503-8135-2/20/10.
https://doi.org/10.1145/3417990.3422001

## 1 INTRODUCTION

*Motivation.* Virtual maps are a key constituent of modern video games, which present several square miles of complex terrain to the players[1], and the design of those maps takes a substantial undertaking in game development. To support map development, complex map editors are created with the game, which are domain-specific modeling environments mixing graphical modeling and programming or scripting languages. Moreover, procedurally generated content has become a highlighted feature of several games [17], which means that the application is able to automatically generate game maps before or during the game session in order to increase the size of game world and content and to enhance replayability. Some well-known generators are even exhibited as art pieces [1].

*Problem statement.* Creating maps is a challenging and cumbersome task with little support for automation. Existing map development tools [3, 18] have a steep learning curve with a complex development process, which can be prohibitive for small teams or individual (indie) developers. Additionally, their expensive licence is impractical for games with a smaller scope.

Each map should satisfy the domain-specific requirements of the game (i.e., each map should be playable). Additionally, map resources are expected to have a realistic and fair distribution. Existing solutions for automating map generation are frequently error-prone (thus frequently resulting in glitched or unplayable maps[2]), or they skip the generation of sophisticated features (which results in maps with significantly fewer features than manually created maps[3]).

*Objectives and contributions.* This paper aims to provide tool support to simplify and partially automate the creation of consistent video game maps using model-based techniques. In particular,

- We propose a declarative multi-step map generation approach to first derive an abstract graph structure and then the precise geometry of world maps.
- Our approach supports logic constraints to enforce the domain-specific game rules on the maps.
- We provide prototype implementation of the map generation approach, by integrating a consistent model generator [8] into Unity [13], the popular game development framework.

[1] Red Dead Redemption 2: 29mi$^2$, Grand Theft Auto V 31$^2$, The Witcher 3: 84 mi$^2$
[2] A sample bug repository for world generation can be found at: https://undertowgames.com/forum/search.php?keywords=map+generation&fid%5B0%5D=6.
[3] E.g. Brendan Caldwell: Broken Promises of No Man's Sky, Rock Paper Shotgun 2016, https://web.archive.org/web/20160818024314/https://www.rockpapershotgun.com/2016/08/17/broken-promises-of-no-mans-sky/

*Added value.* The proposed approach enables to automatically synthesize game world maps with structural guarantees. The proposed technique can also support the manual development of such maps by seamlessly integrating into the game development environment which can significantly reduce the development effort. We selected the Unity game engine as a target platform, since its free license is preferred by indie developers. With our prototype implementation, game developers can specify key map features in a generic, declarative way and define logic constraints that the solver ensures in all generated maps. Semantically, by generating game maps for different logic structure, the diversity of the game maps can be enforced [6, 7]. Our tool chain allows to use alternative components for graph generation or map visualization.

# 2 BACKGROUND

## 2.1 Running Example

Our map generation approach is illustrated in the context of a typical game world generation problem. In our approach, we specify the requirements of the world map using three categories of rules:

(1) **Structural** rules capture the key concepts of a world map as well as their potential relations. For example, a map may have regions like sea, island, continent, city, lake and mountain, The "inside" relation denotes containment between regions, while the "next" relation links two cities.
(2) **Quantity** constraints restrict world map generation to contain a designated number of concepts, e.g. 2 islands, 1 continent, 1 lake, 2 cities and 3 mountains in our example.
(3) **Logic constraints** help to enforce the domain-specific game rules on map generation. For example: (i) If two cities are linked with a "next" relation, then they are connected with a path there is a path between that exclude water areas (like sea or lake). (ii) An island should contain at most one city.

While complex quantity and logic constraints are key requirements of a game map, traditional noise-based world map generators do not guarantee to respect such constraints, thus they will eventually be violated at some parts of the map.

## 2.2 Game engines

A game engine is an integrated suite of software tools used for developing games. Usually, it two main components. The *game editor* provides a graphical user interface (GUI) that the game developer interacts with to modify the contents of a game. Normally, the editor allows developers to view all game objects in a given level as well as a real-time preview of the level where the user can manipulate the game objects (with some extra context-specific features). The *run-time environment* is the core program that executes the game, which typically includes simulating physics, running the game logic, handling network communication and rendering frames.

Unity [13] is a widely used game engine which offers a full 3D engine and supports game development with C# scripts. Compared to other game engines, Unity is more lightweight, as such, it has a smaller learning curve. Since Unity became free for individual (indie) developers in 2009, it has grown to be one of the most popular game engines for indie game developers.

## 2.3 World map generation

*Game worlds.* A *game world* represents an artificial environment where the players interact with other features or agents (like each other) during game play. In our context, a game world defines the static environment (or the initial state of a dynamic environment) that player can interact with to achieve the game objectives. In this paper, we aim to aid the development of this design artifact.

*Content Generation.* Procedural Content Generation (PCG) uses algorithms to generate content that would otherwise be manually produced by game developers [11]. In case of *static PCGs*, generation takes place prior to the start of the game play, whereas for *Dynamic PCGs*, the game content is generated while the game is running. World map generation is a specific subclass of PCG where a part or the entire world map of the game is generated automatically from a set of configurations planned by level artists, who are in charge of design the game levels or worlds.

A common PCG technique used in world map generation to synthesize computer graphics with realistic appearance is *Perlin Noise* [5]. It differs from other noise algorithms in its intuitive parameterization and cohesive noise results instead of pure random. Our tool uses Perlin noise library within Unity to derive the concrete final visualization (rendering) of the world map.

## 2.4 Model-based techniques

*Graph models.* Abstract game concepts in a game level are captured in our approach by a graph-based model with nodes (objects) representing game concepts and edges (links) defining relations between nodes. The possible types of nodes and edges are captured by a respective *metamodel*, which is compliant in our approach with the popular Eclipse Modeling Framework (EMF). However, EMF is a Java-based technology, which is not directly usable in the C#-based Unity framework, thus certain EMF-based code generators had to be adapted or re-developed.

*Graph Model Generator.* An abstract world map representation will be synthesized by a graph model generator, which takes a metamodel, a set of well-formedness constraints and a configuration file as input, then derives an instance model corresponding to the metamodel while satisfying the constraints. In this work, we use the state-of-the-art VIATRA Solver [8] for graph model generation. VIATRA Solver starts from an initial partial model, which is gradually refined to a concrete model. The actual generation is carried out as a design space exploration with incremental constraint evaluation supported by VIATRA Query [15].

# 3 KEY FEATURES AND ARCHITECTURE

## 3.1 Overview

The game world generator is available as a plugin integrated with the Unity game engine. The user can easily configure the generation directly inside the Unity editor. The generation results are obtained as a Unity game object within a game scene, which enables the game developer to perform modifications to the auto-generated world or continue development using the generated result.

Figure 1 provides an overview of the architecture of our tool. A level artist first uses the *Map Concept Editor* to define new region types to generate the structural relationship between regions and

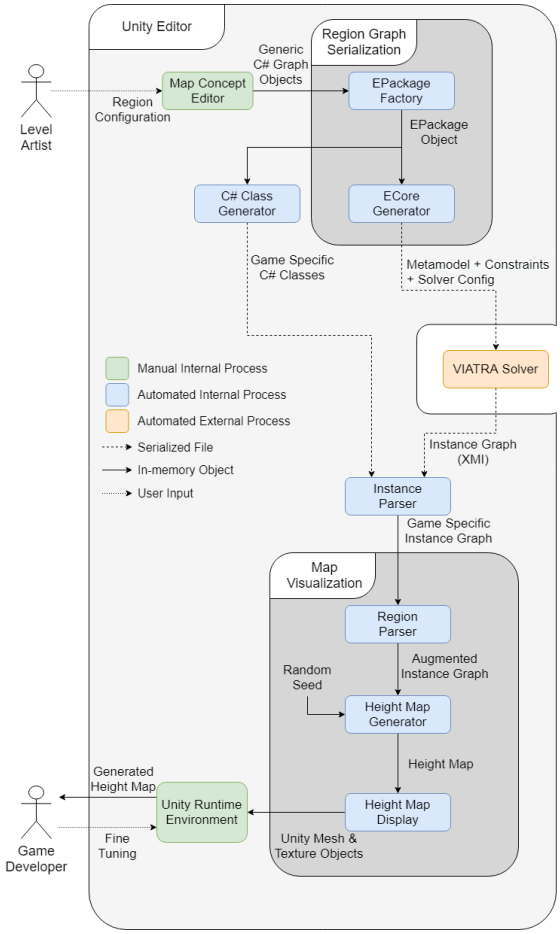


**Figure 1: Architecture of world map generation**

specific properties for each region type. Once all region configurations have been set, the generation can be started with one click of a button. The entire generation process is carried out in parallel with the main loop of the Unity Editor, thus the user can continue game development while the generation takes place. Once the generation is finished, the concrete world map will be a mesh game object, with visualization in the preview window of the Unity Editor. Game developers can easily copy the game object to save the generation result or fine-tune the resulting map to fit their needs.

The generation process consists of three main steps. First, the user-defined region configuration is translated and serialized for graph generation through the *Graph Region Serialization*. Then, the *VIATRA Solver* creates an instance graph model for the setting. This abstract graph is then post-processed by the *Map Visualization* component and finally rendered by Unity.

## 3.2 Map Concept Editor

The *Map Concept Editor* is the entry point of our tool for the level artist to set map-specific region configuration parameters prior to map generation. Figure 2 shows the region configuration for the running example to highlight the two main components of the Map

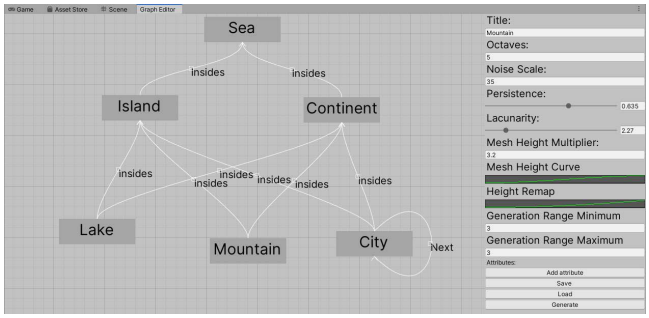


**Figure 2: The GUI of Map Concept Editor**

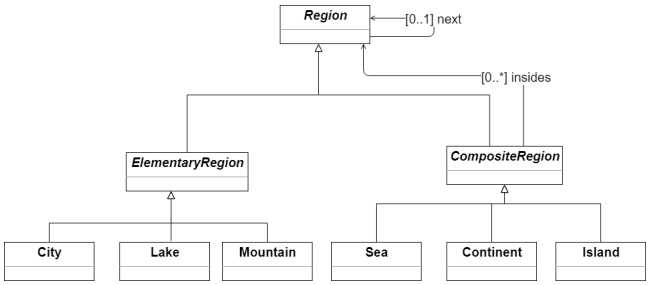


**Figure 3: A metamodel used for region graph generation**

Concept Editor, namely, the Region Editor (left) and the Region Property Editor (right).

The Region Editor enables to define novel map concepts and various relations between them. The *structural* rules of the running example are defined with this editor. Map features are specified as region types connected by *insides* relation to describe the structure of the map. The *next* loop of a city captures the potential reachability relations between two cities. The region property editor enables to set properties for each region type. Such properties include important visualization such as *Octaves* and *Noise Scale* as well as configuring the number of region instances to generate (*Generation Range Max/Min*) to respect the *quantity* constraints.

## 3.3 Graph Generation

The graph generation step bridge between game world generation with a consistent graph solver. It is responsible for transforming the game world generation problem into a graph model generation problem, invoking the graph model generation and parse the resulting instance graph model into C# object representations.

*Metamodel.* Transforming from the game world generation problem to graph generation relies on mapping from region configurations to a metamodel (carried out by the *EPackage Factory* and *Ecore Generator*) to respect the *structural* rules (see Figure 3). The underlying metamodel contains fixed three abstract classes as superclasses of the game-specific classes: *Region*, *CompositeRegion* and *ElementaryRegion*. A region can have one or many next region with other regions. The concrete region classes inheriting the abstract classes are derived from the user inputs and created in the metamodel. The critical relationship in the metamodel is represented

as a composition from composite regions to other regions (*insides* relation). It describes the structural information of region instances that can be later translated to positional information during the visualization. The classes at the bottom of the metamodel are the map features defined for the running example. A map feature is classified as a composite region if it can contain at least one other feature; otherwise, it is an elementary region. For example, a sea is a composite region because it can contain continents and islands. However, a lake is an elementary region as no other regions can be contained in it.

*Constraints and configurations.* Since the metamodel only shows generic composition and associations (e.g. a region can be inside a composite region), the exact relationships (e.g. an island region should be in a sea region) and the *logic constraints* are expressed using the VIATRA query language [15]. For example, our map needs to satisfy the constraint that a sea region can only contain islands or continents. The corresponding constraint is formulated as follows:

```
pattern isIsland(region){ Island(region); }
pattern isContinent(region) { Continent(region); }
@Constraint pattern SeaInsideViolation(sea: Sea) {
    Sea.insides(sea,ins);
    neg find isIsland(ins);
    neg find isContinent(ins);
}
```

The first two patterns are helper patterns used in the actual constraint. The logic constraint is specified as a violation, that is, any graph containing an occurrence of this is inconsistent, i.e. if there is a region inside the sea and is neither an island or a continent. We provide a set of default definitions in *Ecore Parser* of the relation constraints such as *insides*, whereas a user can also write custom constraints directly using the query language.

Moreover, the *Ecore Parser* examines the tree structure of the region configuration, creates a starting instance graph with an instance of the root region as the container object. In our running example, the starting instance graph is a sea object. As further input of the generation, a configuration file is automatically produced when game world generation is initiated. This configuration file contains the references to the metamodel, the constraint specification and the seed instance model together with search parameters such as the required number of instances for each region type.

*Graph Generation and Parsing.* The graph generation step in our world map generator tool is invoked by creating a background JVM process which executes the *VIATRA Solver* by passing the inputs prescribed by the configuration file above. As such, Unity developers are not blocked during map generation. Once the graph generation is completed, each region in the generated instance graph is transformed into Unity objects of C# class generated by the *C# Class Generator*. The *Instance Parser* first transforms all the regions along the *insides* composition relation and then infers other relations. It creates a single region instance object of the root region that contains the entire representation of the instance graph, which is then used for synthesizing the concrete game world.

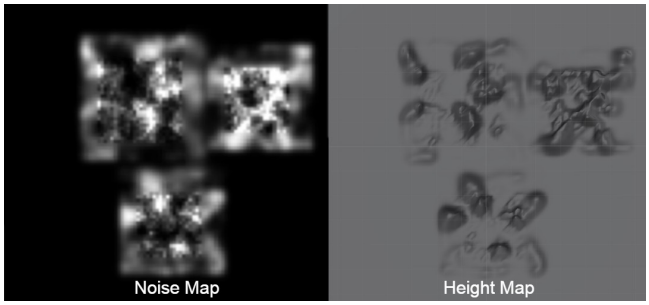


**Figure 4: An example of height map annotation**

## 3.4 Concrete World Map Synthesis

The abstract representation of the world map derived by the graph generation phase only contains structural information of the regions without precise positional information. It is the job of the map visualization phase to determine where these regions are actually located, the height map of each region, and how the regions should look like when rendered on the screen. The map visualization phase contains three components to achieve these tasks: the Region Parser, the Height Map Generator and Height Map Display.

*Region Parser.* The *Region Parser* takes the instance graph produced by the Instance Parser and produces an augmented instance graph, which maintains the original region types and relationships but adds positional information onto each region instance. This process is achieved by performing a pre-order traversal of the instance graph where the container of a given region instance is always visited before visiting the region instance itself. In this manner, we determine the area taken up by a region instance in the map by randomly sampling a sub-area of the container area of that instance. In case of the root node, this area is the entire map.

*Height Map Generator.* Taking the augmented instance graph from the previous step, the *Height Map Generator* produces the height map of the level. The height map is a 2D array of real (float) numbers that describe the height of each point in the world. The final height map is produced by a pre-order traversal of the augmented instance graph where at each node, we generate the height map of the corresponding region instance. The exact heights are then generated using the Perlin Noise technique. By combining the parameters provided by the level artist at the right side of Figure 2. Each parameter will influence the resulting height map of the region instance. For instance, *Noise Scale* defines the range of the random variable the noise generation will sample, whereas *Height Remap* defines the curve when sampling the final height of each region. The height map of each region instance is then annotated onto the initial height map using the positional information of that node. A sample auto-generated noise and height map is shown in Figure 4, where height is color-coded in the noise map.

*Height Map Display.* This component takes the height map of the world and converts it into Unity-specific Mesh and Texture objects which are used to render the world. A *Mesh* object describes the geometry of the world and it is used by Unity to render the shape of the world. A *Texture* object represents the color of the world and it

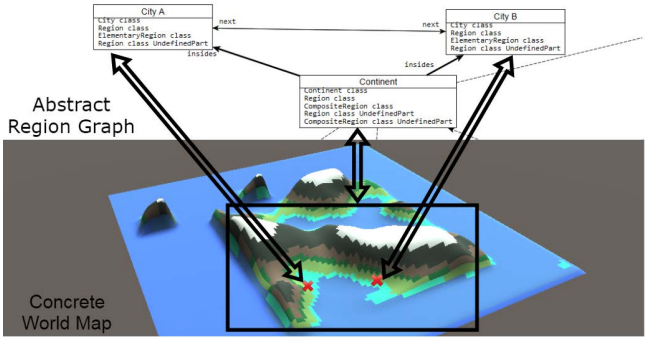


**Figure 5: Abstract region graph and concrete world map**

is used by Unity to visualize the final world. To determine the color of a point on the map, the game developer can specify a height-color mapping for each region with the objects generated in the scene. Figure 5 shows a sample output generated automatically by our tool for our running example. As an illustration, we also added a comparison with the corresponding auto-generated region graph. There, a continent contains two cities that are next to each other. The generated concrete map can then be fine-tuned or directly used for games by the game developers.

### 3.5 Related Work on Map Generation

Key research papers and tools related to grid-based map generation are overviewed in [16]. The main difference is that they are supporting discrete tile-based maps, while our multi-step approach transforms the generated graph into continuous geometry.

The conceptually closest approaches [10, 12] are based on *answer set programming* as a background technique to solve the logic problem of generating consistent maps. In contrast, our approach uses a graph-based background solver which can exclude isomorphic states [7] to enforce diversity requirements [6]. Other approaches are using optimization techniques (like *genetic algorithms* to place rooms [14]), or other techniques that semi-randomly create maze-like structures (like *cellular automatas* [4]). In those techniques, it is the responsibility of the developer to enforce the domain-specific requirements of the game on the maps.

Terrain geometry generation is typically carried out by repeated application of noise generation and region splitting [2, 9]: noise generation adds random features to the map (e.g., randomizes the heightmap of the terrain, randomly adds objects like trees into forest regions), and region selection splits the map into parts (e.g., based on the heightmap select parts of the map as forest). Those techniques are mainly focusing on the generation of realistic and aesthetic maps, consistency is typically a secondary requirement.

## 4 CONCLUSION AND FUTURE WORK

In this paper, we proposed an integrated game world map generation tool chain to synthesize maps respecting game-specific constraints as requirements. We integrated a graph generator to derive logic structure of consistent game worlds. Then, we visualized those structures to concrete game worlds by synthesizing height maps with Perlin noise. These auto-generated game worlds can be manually post-processed by game developers inside Unity. The GitHub repository of the generator can be found at: https://github.com/20001LastOrder/Map-Generation

As future work, we aim to combine our approach with modern noise-based map generators to improve the quality of concrete maps. Moreover, integrating a numerical solver for visualization may deduce better positional information of each game concept concerning the structural information provided by the graph.

*Actknowledgements.* This paper is partially supported by the NSERC RGPIN-04573-16 project and the BME-Artificial Intelligence FIKP grant of EMMI (BME FIKP-MI/SC).

## REFERENCES

[1] Tarn Adams and Zach Adams. 2011. Dwarf Fortress. http://www.bay12games.com/dwarves/ The Museum of Modern Art, Department of Architecture and Design, Object number 1748.2012, https://www.moma.org/collection/works/164920.
[2] Jonathon Doran and Ian Parberry. 2010. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games* 2, 2 (2010), 111–119.
[3] World Machine Software LLC. 2020. World Machine. https://www.world-machine.com/
[4] Andrew Pech, Philip Hingston, Martin Masek, and Chiou Peng Lam. 2015. Evolving Cellular Automata for Maze Generation. In *Artificial Life and Computational Intelligence*, Stephan K. Chalup, Alan D. Blair, and Marcus Randall (Eds.). Springer International Publishing, Cham, 112–124.
[5] Ken Perlin. 1985. An image synthesizer. *ACM Siggraph Computer Graphics* 19, 3 (1985), 287–296.
[6] Mike Preuss, Antonios Liapis, and Julian Togelius. 2014. Searching for good and diverse game levels. In *2014 IEEE Conference on Computational Intelligence and Games*. IEEE, 1–8.
[7] Oszkár Semeráth, Rebeka Farkas, Gábor Bergmann, and Dániel Varró. 2020. Diversity of graph models and graph generators in mutation testing. *International Journal on Software Tools for Technology Transfer* 22, 1 (2020), 57–78.
[8] O. Semeráth, A. A. Babikian, S. Pilarski, and D. Varró. 2019. VIATRA Solver: A Framework for the Automated Generation of Consistent Domain-Specific Models. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 43–46.
[9] Ruben Michaël Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. 2011. A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics* 35, 2 (2011), 352–363.
[10] Anthony J Smith and Joanna J Bryson. 2014. A logical approach to building dungeons: Answer set programming for hierarchical procedural content generation in roguelike games. In *Proceedings of the 50th Anniversary Convention of the AISB*.
[11] Gillian Smith. 2015. An Analog History of Procedural Content Generation. In *Proceedings of the 10th International Conference on the Foundations of Digital Games*, José Pablo Zagal, Esther MacCallum-Stewart, and Julian Togelius (Eds.). Society for the Advancement of the Science of Digital Games.
[12] Thomas Smith, Julian Padget, and Andrew Vidler. 2018. Graph-Based Generation of Action-Adventure Dungeon Levels Using Answer Set Programming. In *Proceedings of the 13th International Conference on the Foundations of Digital Games (FDG '18)*. ACM, New York, NY, USA, Article 52, 10 pages.
[13] Unity Technologies. 2020. Unity. https://unity.com/
[14] Julian Togelius, Mike Preuss, and Georgios N. Yannakakis. 2010. Towards Multiobjective Procedural Map Generation. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games* (Monterey, California) *(PCGames '10)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages.
[15] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. 2016. Road to a Reactive and Incremental Model Transformation Platform: Three Generations of the VIATRA Framework. *Softw. Syst. Model.* 15, 3 (July 2016), 609–629.
[16] Breno MF Viana and Selan R dos Santos. 2019. A Survey of Procedural Dungeon Generation. In *2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE, 29–38.
[17] Wikipedia contributors. 2020. List of games using procedural generation. https://en.wikipedia.org/wiki/List_of_games_using_procedural_generation
[18] Ken Xu and Damian Campeanuy. 2014. Houdini Engine: Evolution towards a Procedural Pipeline. In *Proceedings of the Fourth Symposium on Digital Production* (Vancouver, British Columbia, Canada) *(DigiPro '14)*. Association for Computing Machinery, New York, NY, USA, 13–18.