Software Impacts 13 (2022) 100363

Contents lists available at ScienceDirect

Software Impacts

journal homepage: www.journals.elsevier.com/software-impacts

Original software publication

Unity Snappable Meshes

Nuno Fachada [a,*], Rafael C. e Silva [b], Diogo de Andrade [a], Nélio Códices [c]

[a] COPELABS, Lusófona University, Lisboa, Portugal
[b] ECATI, Lusófona University, Lisboa, Portugal
[c] INESC-ID, Instituto Superior Técnico, University of Lisbon, Lisboa, Portugal

ARTICLE INFO

Keywords:
Procedural content generation (PCG)
Computer games
Layout
Designer-centric methods
3D maps
Unity game engine

ABSTRACT

The Snappable Meshes algorithm procedurally generates 3D maps for computer games by iteratively selecting and linking pre-built map pieces via designer-specified connectors. In this paper we present an implementation of this algorithm in the Unity game engine, describing its architecture and discussing core implementational solutions. A number of examples illustrate the potential of the algorithm and the capabilities of the software. We assess the application's impact on past and ongoing research, and how it can be improved to support future research questions.

Code metadata

Current code version	v1.0.0
Permanent link to code/repository used for this code version	https://github.com/SoftwareImpacts/SIMPAC-2022-123
Permanent link to Reproducible Capsule	
Legal Code License	Apache License 2.0
Code versioning system used	Git
Software code languages, tools, and services used	C#
Compilation requirements, operating environments & dependencies	Unity (cross-platform game engine) ≥2020.3 LTS
If available Link to developer documentation/manual	https://github.com/VideojogosLusofona/snappable-meshes-pcg/wiki/User-guide
Support email for questions	nuno.fachada@ulusofona.pt

1. Introduction

This paper describes a Unity game engine [1] implementation of the Snappable Meshes technique for procedurally generating 3D maps for computer games [2,3]. The technique generates maps by iteratively selecting and connecting pre-modeled building blocks on designer-specified positions, while offering a number of parameters which allow users to manipulate the look and feel of the generated outputs. The concrete method for selecting the next building block is also given as a parameter for the generation algorithm, further promoting designer control over the created maps. The Snappable Meshes technique is able to produce sizeable maps in milliseconds, while avoiding size and layout constraints [3]. The Unity implementation presented here follows these conceptual guidelines, but also addresses a number of strictly implementational issues, such as geometry overlap detection, map validation through path finding, and reproducible large-scale experimentation.

This paper extends Refs. [2,3] by offering a detailed description of the software that supports them, and is organized as follows. Section 2 describes the Unity implementation of the Snappable Meshes technique, highlighting the software's architecture and how several implementational issues were solved. To better contextualize the reader on the type of outputs generated with the technique, a number of illustrative examples are presented in Section 3. The impact of this Unity implementation is discussed in Section 4, while its limitations and possible improvements are addressed in Section 5.

* Corresponding author.
E-mail addresses: nuno.fachada@ulusofona.pt (N. Fachada), castroesilva.rafael@gmail.com (R.C. e Silva), diogo.andrade@ulusofona.pt (D. de Andrade), nelio.codices@tecnico.ulisboa.pt (N. Códices).

https://doi.org/10.1016/j.simpa.2022.100363
Received 29 June 2022; Accepted 8 July 2022

2. Description

This section is organized into three distinct subsections. We start by summarizing the designer's workflow in Section 2.1. The software's architecture is presented in Section 2.2. Finally, a discussion of how several interesting implementational issues were solved takes place in Section 2.3.

2.1. Designer workflow

The Unity implementation of the Snappable Meshes technique makes use of Unity's editor tools to handle the input of the human designer, allowing designers and researchers to test the method. The implementation includes two preconfigured scenes (discussed in Section 3), allowing interested users to start experimenting immediately. The map generation workflow is divided into three steps:

1. Configuration of the generation process, during which the designer imports the building blocks to be used for map creation and specifies the algorithm's parameters. These building blocks, or map pieces, are in essence parameters of the generation algorithm.
2. Map generation and validation assessment, during which the created map and respective path finding validation results are presented.
3. Demo of an NPC traversing the map, which starts automatically when entering Unity's play mode, offering the designer a first-person perception of the generated map, as exemplified in Figs. 2(b), 2(d), 2(f), and 2(h).

This workflow can be repeated until the designer is satisfied with the results.

2.2. Software architecture

Fig. 1 presents a simplified UML class diagram of the software's architecture, highlighting its main components. Here, `MonoBehaviour` and `ScriptableObject` are Unity-provided classes, essential for developing projects in this game engine, and which can be manipulated from Unity's editor. The former, `MonoBehaviour`, is the base class from which every game object[1]-attachable script derives. The latter, `ScriptableObject`, allows derived classes to serve as templates for persistent data-oriented game assets. These assets can also be handled from the editor—as well as from scripts—although they cannot be directly attached to game objects.

The `GenerationManager` class is at the core of the implementation. It holds the generation parameters and performs the map generation itself, implementing the Snappable Meshes algorithm. As can be observed in Fig. 1, the `GenerationManager` holds the collection of `MapPiece`—building blocks—to be used for map generation. An instance of this class is exposed to the user via Unity's editor, allowing them to choose and configure the algorithm's parameters, its building blocks, and the desired block selection method (these are further discussed in Section 2.3.3, namely the classes to the right and below `GenerationManager` in Fig. 1).

As shown in Fig. 1, each `MapPiece` can have zero or more connectors, although building blocks without connectors will not be very useful, as they cannot be snapped together with existing map pieces. In turn, a `Connector` instance can reference another connector during map generation. If so, it is linked with that other connector. Otherwise, it is free and may become linked before the map generation process terminates.

[1] Game objects, implemented in Unity's `GameObject` class, are entities that exist in Unity Scenes. They implement the Component design pattern [4], and among different types of component, they can contain `MonoBehaviour`-derived scripts.

The `NavBuilder` and `NavScanner` classes, bound to map navigation and validation, are analyzed in Section 2.3.4. Finally, the `Experimenter` class and `IExperiment` interface are related with reproducible experiments, and are further discussed in Section 2.3.5.

2.3. Implementation details

In this section we describe a number of implementation details which might be relevant for researchers or developers to better understand the code, either for improving/building on it, or with the purpose of reimplementing the Snappable Meshes technique in another framework.

2.3.1. Importing building blocks as prefabs

When the designer creates a map piece, it is necessary to specify its mesh, individual connectors (and their connection characteristics), and one or more colliders[2] if piece overlap—discussed next—is undesirable. Human-designed pieces are added to the generation pipeline as *prefabs*, Unity's implementation of the Prototype design pattern [4]. Therefore, the blocks with which the algorithm is parameterized are prototypes, while the pieces actually placed on the map are copies of the original blocks.

2.3.2. Avoiding geometry overlap

A crucial aspect of the Snappable Meshes technique is the capability of creating maps without overlapping building blocks. To guarantee this, when a map piece is selected for placement on the map, an optional verification—specified during the algorithm's parameterization—certifies that it does not intersect existing blocks for each of its possible connections [3].

Overlap verification requires that the building blocks contain one or more box colliders, i.e., rectangular cuboid-shaped bounding volumes approximately mirroring the block's shape. This allows Unity to quickly detect if the current block intersects with existing map pieces in a relatively precise way. Box colliders are used in this implementation since general convex mesh colliders in Unity are limited to 255 triangles and may display inaccurate behavior.

2.3.3. Block selection methods

The Strategy design pattern [4] was used to decouple the block selection methods from the main Snappable Meshes algorithm. Consequently, each selection method is implemented in its own class derived from the `AbstractSM` class, as shown in Fig. 1. Existing selection methods are enumerated through C#'s reflection system by searching for corresponding configuration classes, which extend `AbstractSMConfig`. An instance of the latter is then used by the `GenerationManager` to configure and instantiate the concrete selection method derived from `AbstractSM`. These relations are highlighted in Fig. 1.

The Unity implementation of Snappable Meshes is bundled with four block selection methods—*Arena*, *Corridor*, *Branch*, and *Star*—discussed in detail in Refs. [2,3]. The `GenerationManager` instance shown to the user through Unity's editor allows choosing the desired selection method via a dropdown list. In practice, the user chooses a concrete instance of `AbstractSMConfig`, the parameters of which are then exposed to allow them to configure the picked selection method.

Following this approach, the implementation of new selection methods becomes very simple, requiring only two new classes: one for the selection method itself, and another for configuring it.

[2] A collider is Unity's terminology for a bounding volume.

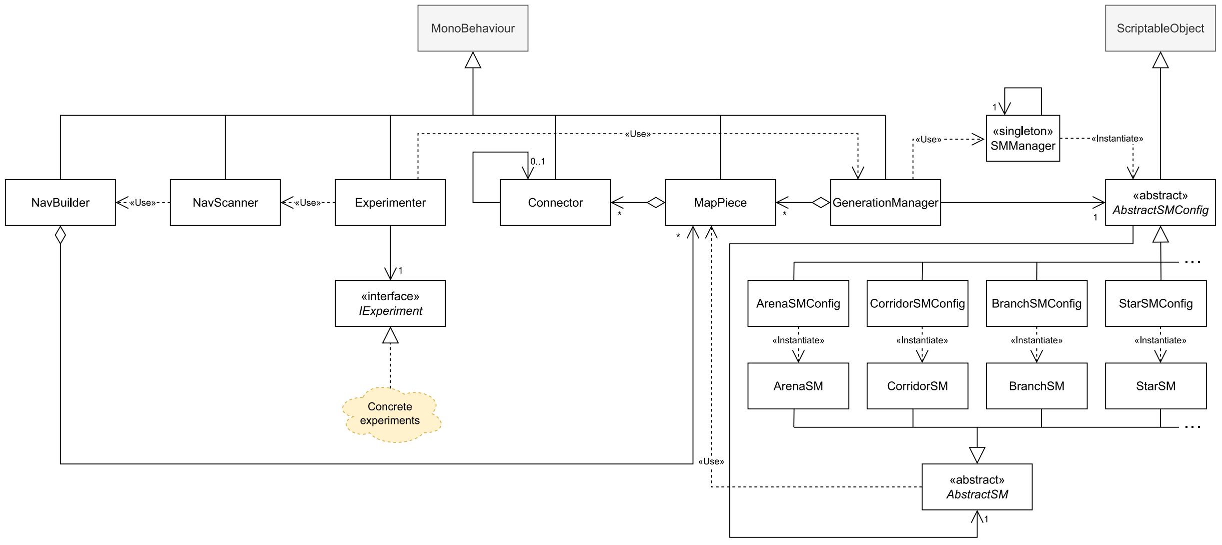

Fig. 1. Streamlined UML class diagram of the Snappable Meshes Unity implementation. Shaded blocks represent Unity classes.

2.3.4. *Map navigation, validation and demo*

Map validation occurs automatically after the generation process, and is carried out with Unity's built-in *navmesh* system. A navmesh—short for navigation mesh—is a mesh of convex polygons that define navigable areas on a map. These polygons can be considered nodes in a graph, with adjacent polygons forming valid paths, or in graph terminology, a link between nodes. Thus, a path finding algorithm such as A* can be used to determine if a path exists between any two nodes.

Unity's navmesh system allows the runtime creation of navmeshes on an existing map, and is used in this implementation for this purpose. After a navmesh is deployed for a generated map, a prespecified number of random navigation points is placed in the navmesh. Unity's path finding system is then used to determine if valid paths exist between each pair of navigation points. Several navigation metrics can be computed with this information, as discussed in Ref. [3]. The first-person demo mode uses valid paths between these navigation points to move the camera-carrying agent around the map.

Several classes are involved in this process, two of which, `NavBuilder` and `NavScanner`, are shown in Fig. 1. The former addresses the runtime creation of the navmesh over the generated map, while the latter is responsible for deploying the navigation points and calculating the different navigation metrics. `NavScanner` is exposed to the user via Unity's editor, allowing them to define the number of navigation points to deploy, the random number generation strategy (important for reproducible experimentation, discussed next), as well as a number of visual debugging options.

2.3.5. *Reproducible experimentation*

The Snappable Meshes implementation in Unity allows the user to configure and run reproducible experiments, essential in both academic and industry contexts. From a research point of view, reproducible experimentation allows the validation of the technique itself. In industry contexts, specific parameter sets can be assessed as adequate (or not) for deploying the map generator in production settings. The `Experimenter` class, shown in Fig. 1, is responsible for this process. It references an object implementing the `IExperiment` interface which exposes a list of parameter sets—either by containing or generating them—guiding the experiments to be performed.

The `Experimenter` class is exposed to the user via Unity's editor, allowing them to select the experiment to be performed. The parameter sets in the selected experiment are automatically associated with the parameters from the Snappable Meshes algorithm through reflection. After the experiment is executed, raw generation and validation times are saved to a CSV file, which can then be imported and analyzed using standard statistical or data science techniques, e.g. [5].

3. Illustrative examples

The Snappable Meshes implementation in Unity is bundled with two preconfigured scenes, *Benchmark* and *Artistic*, each with markedly dissimilar building blocks. Besides allowing interested users to get started quickly, these scenes exemplify some of the output diversity possible with the Snappable Meshes technique.

Fig. 2 displays four examples, one per row, where each column displays a different perspective: a general overview of the map is shown on the left, while a first-person perspective of the same map is presented on the right. The first two rows, Figs. 2(a)–2(b) and 2(c)–2(d), show two maps generated with the *Benchmark* scene. This scene contains "lego"-like pieces with contrasting colors, and is appropriate for designers to understand how the algorithm iteratively builds the maps. It is so named since it was used for benchmarking the technique with respect to generation and validation times, as well as overall navigability [3].

The two bottom rows, Figs. 2(e)–2(f) and 2(g)–2(h), present maps created with the *Artistic* scene. The map pieces included with this scene are considerably different from those in the *Benchmark* scene, yielding cleaner maps with seamless connection interfaces, closer to what one would find in a game development context.

Each of the example maps in Fig. 2 was created with a different implementation-independent block selection method, namely *branch* 2(a)–2(b), *star* 2(c)–2(d), *arena* 2(e)–2(f), and *corridor* 2(g)–2(h). For more information regarding these selection methods, please refer to Refs. [2,3].

4. Impact

The Snappable Meshes algorithm was originally developed for a multiplayer shooter game in the context of a semester project at Lusófona University's Bachelor in Videogames—an industry-focused, interdisciplinary game development degree [6], in which 3D rendering is an important component of the overall teaching strategy [7]. The algorithm favored the game's replayability, demanding that players adapt to a brand new map on every match. Since the algorithm was shown to generate diverse map types (i.e., not specific to the game in question), a preliminary version was first presented in a conference [2]. The technique was later formalized, thoroughly evaluated, and contextualized within the academic and industrial state of the art in procedural level generation [3]. The standalone Unity implementation was first presented in this second publication, since it supported the exhaustive benchmarking and evaluation of the Snappable Meshes

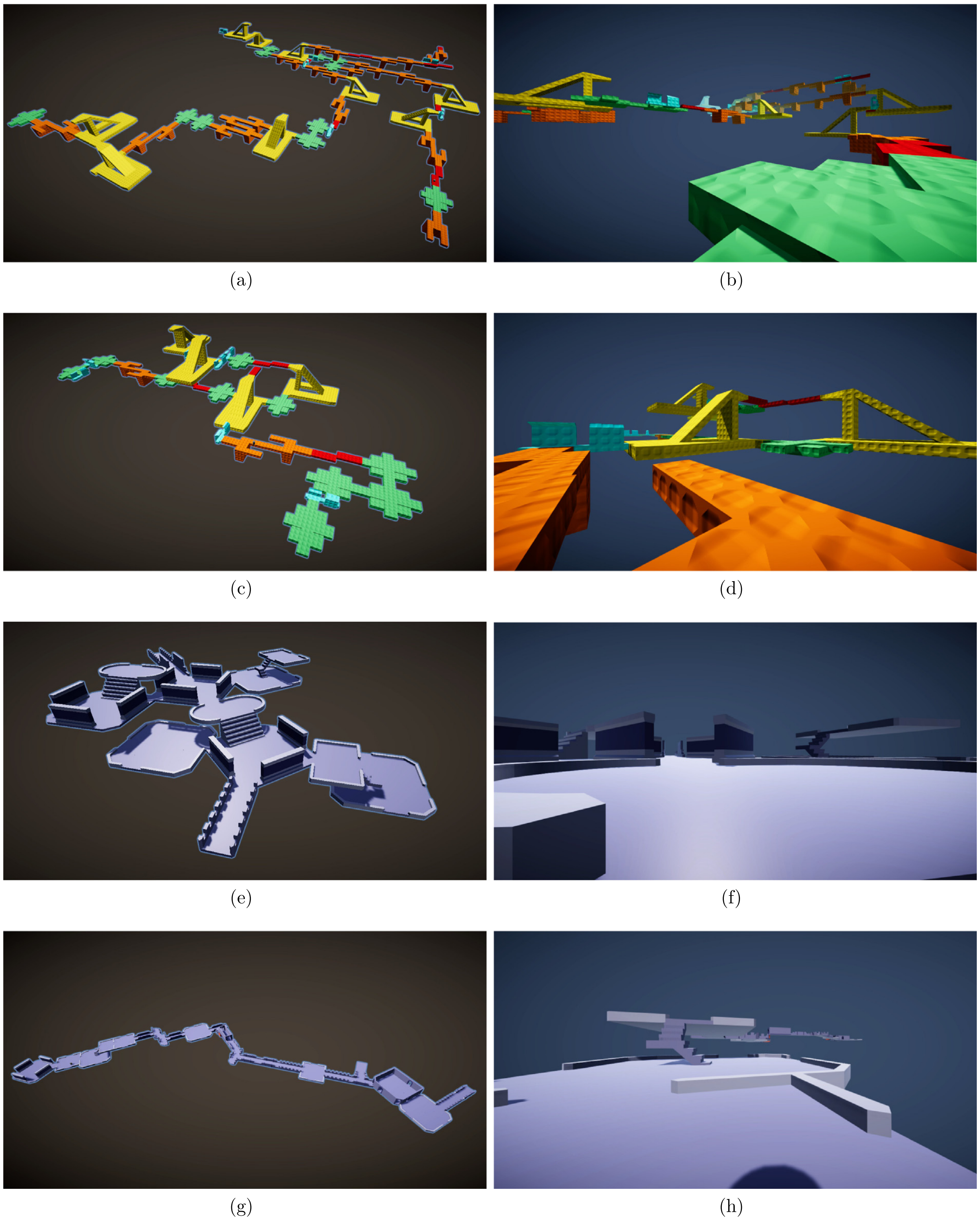

Fig. 2. Four illustrative maps generated with the Snappable Meshes algorithm. Each row contains two perspectives of an example map; more specifically, figures on the left column display an overview of the respective map, while figures on the right show a frame from the first-person demo running on that same map. Parameterization overview: (a)–(b) map generated in the *Benchmark* scene with the *branch* selection method; (c)–(d) map created in the *Benchmark* scene with the *star* selection method; (e)–(f) map generated under the *Artistic* scene with the *arena* selection method; (g)–(h) map generated in the *Artistic* scene using the *corridor* selection method.

technique [3]. Finally, the application's architecture and several implementational aspects—especially in the context of its architectural design—are described in the current paper.

The Unity implementation of the Snappable Meshes algorithm is currently aiding the pursuit of additional research questions. In particular, we are studying connector auto-generation given an arbitrary block/map piece, and improving collision detection by using a hierarchical bounding box tree, which is also automatically generated from the geometry. Although the algorithm allows generating maps in a fraction of the time of a fully human-based approach, it still demands considerable manual work by the designer, and our current research would allow for faster iteration times with individual block design.

There is also preliminary work on adding support for map loops, and extending the algorithm to dynamically change the geometry to close unused connectors, instead of just having a drop out of the current play area as in the current implementation.

5. Limitations and possible improvements

In addition to the limitations discussed in the previous section, namely the requirement for non-trivial human labor and lack of support for map loops—which are currently being addressed—we believe there are two other issues with the implementation that could be targeted for improvement.

The first issue concerns the way navigability validation is performed, i.e., by deploying a predefined amount of navigation points and then verifying their connectivity. In the current version, these points are randomly placed in the runtime-generated navmesh. Some of these points may be placed on rooftops or other areas not intended for navigation. Consequently, if this approach is followed as-is for indoor maps with hollow building blocks, invalid paths outside the intended play area may be generated. Therefore, individual building blocks may require additional metadata specifying valid movement zones. An interesting improvement to the application would be the automatic generation of this metadata, although this is not currently being researched.

The second issue is merely implementation-dependent, and is also related with navigation. More specifically, the current version does not support jumps, requiring map pieces to be linked to accept a direct path between them. While this has not limited our research with the Snappable Meshes technique thus far, it can hinder experimentation by interested readers, and is therefore an implementational aspect that could be improved.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was supported by the Fundação para a Ciência e a Tecnologia under Grant UIDB/04111/2020 (COPELABS).

References

[1] Unity Technologies, Unity®, 2022, https://unity.com/.

[2] R.C. e Silva, N. Fachada, N. Códices, D. de Andrade, Procedural game level generation by joining geometry with hand-placed connectors, in: Proceedings of Videojogos 2020-12th International Videogame Sciences and Arts Conference, SPCV, 2020, pp. 80–93.

[3] R.C. Silva, N. Fachada, D. De Andrade, N. Códices, Procedural generation of 3D maps with snappable meshes, IEEE Access 10 (2022) 43093–43111, http://dx.doi.org/10.1109/ACCESS.2022.3168832.

[4] R. Nystrom, Game programming patterns, Genever Benning, 2014, https://gameprogrammingpatterns.com/.

[5] N. Fachada, Snappable meshes performance dataset, 2022, Jan., http://dx.doi.org/10.5281/zenodo.5851209.

[6] N. Fachada, N. Códices, Top-down design of a CS curriculum for a computer games BA, in: Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE '20, ACM, New York, NY, USA, 2020, pp. 300–306, http://dx.doi.org/10.1145/3341525.3387378.

[7] D. de Andrade, N. Fachada, PyXYZ: an educational 3D wireframe engine in Python, in: Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE '21, ACM, New York, NY, USA, 2021, pp. 519–525, http://dx.doi.org/10.1145/3430665.3456345.