Level Graph – Incremental Procedural Generation of Indoor Levels using Minimum Spanning Trees

Bartosz von Rymon Lipinski
Game Tech Lab
Nuremberg Institute of Technology
Nuremberg, Germany
bartosz.vonrymonlipinski@th-nuernberg.de

Simon Seibt
Game Tech Lab
Nuremberg Institute of Technology
Nuremberg, Germany
simon.seibt@th-nuernberg.de

Johannes Roth
Mimimi Games GmbH
Munich, Germany
j.roth@mimimi.games

Dominik Abé
Mimimi Games GmbH
Munich, Germany
d.abe@mimimi.games

***Abstract*—Procedural generation of virtual worlds is an important aspect of game development since decades, typically for increasing replayability or for speeding up the level creation process. However, the utilization of this potential has always been a great challenge due to the difficult controllability of the underlying algorithms or limitations to specific level geometries, like 2D regular structures. In this paper, we present a novel approach for semi-automatic generation of a wide variety of 2D/3D corridor and room systems. The underlying processing pipeline is based on a separation between a user-guided generation of a graph-based abstract level structure and a fully-automatic construction of the corresponding geometry using a pre-modeled component library. The core algorithm is built on the computation of an extended minimal spanning tree, which can be controlled by a set of intuitive vertex and edge parameters. First experimental results have shown that our incremental generation pipeline allows the efficient creation of complex indoor levels, minimizing limitations on level and game designers' creativity.**

Keywords—Procedural content generation, level design, game development, graph algorithm, geometry generation

I. INTRODUCTION

Video game worlds are becoming more and more complex, driven by advances in game engineering, high-performance graphics hardware and increasing demands on player experience [1]. As an example, Rockstar Games' „Red Dead Redemption 2" required about eight years of partly manual development time for the game content [2]. One consequence for game development companies is thus the growing challenge between meeting high-quality requirements and achieving reasonably short production times. This makes automation techniques for content creation more and more important. In this work, we therefore concentrate on procedural world generation and in particular on indoor levels, consisting of corridors and interconnected rooms. The first procedural level generation methods in games date back to the 1980s, but these first solutions were difficult to control for non-expert users [3, 4]. Incremental procedural improvements of the level design were not practically feasible, so that usually time-consuming manual post-processing was still necessary [5]. In addition, many of the techniques were limited to simple 2D mazes or planar dungeons, so that more complex or three-dimensional game worlds could only be implemented by complicated assembling of individually generated fragments, which sometimes led to restriction in creativity [6]. One example is Bethesda Game Studios' highly rated title "The Elder Scrolls IV: Oblivion", partly criticized for its repetitive and generic dungeon design [7].

Our approach implements an iterative procedural generation pipeline based on a graph data structure, called the *level graph*. It represents the abstract model of a static indoor level: Vertices correspond to positions of junctions and rooms, edges to connecting hallways. Generation takes places within a user-defined hull mesh, which can be any 3D model, e.g. from a simple bounding box for a rectangular maze to a closed terrain model that is supposed to contain a mine level with multiple floors. Using hull meshes makes it possible to take advantage of the "focus-in-context" principle: After each iteration the user gets the possibility to adjust the current result via regeneration, manual modification or to concentrate on another local area of the graph [8]. The structure of the level is defined by the execution of one or a series of the following main graph algorithms: 1. construction of a fully connected random base graph, 2. computation of a minimal spanning tree (MST), and 3. insertion of cycle-forming edges. Using such a multistep process allows to control the generation by a set of user-prioritized vertex and edge parameters that can be mapped to easily understandable, yet finely structured geometric properties. Examples are angles, lengths and variables that affect the distribution of individual graph elements. Parameter violations and mutual geometric intersections are mathematically modeled as edge weights and implicitly resolved in the computation of the MST, which gives us a basis for a level that is as violation-free as possible.

The final part involves the construction of the geometric representation of the level by converting the level graph into a polygonal model. This process requires a set of pre-modeled 3D components, each labeled e.g. as a room, corridor or door. It is executed fully automatically, including stitching of multiple corridor components to cover long hallways and constructing room entrances using Boolean operators from Constructive Solid Geometry modeling [9, 10].

After presentation of the related work in the following section, a more detailed description of our processing pipeline, including the corresponding generation parameters, can be found in section 3. The subsequent topics comprise the implementation of our prototype in the Unity 3D game engine and the presentation of qualitative and quantitative results in section 4 [11]. We conclude our paper with a final discussion in section 5, followed by an outlook on future work.

II. RELATED WORK

The techniques of procedural content generation in game development are mostly restricted to specific types of world elements. For example, previous approaches are used for the generation of landscapes, vegetation, game rules or mechanics [5]. Especially for the procedural generation of indoor maps

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

numerous different approaches already exist: Dormans and van der Linden et al. use generative graph grammar to create room and corridor systems [12–14]. Johnson et al. use cellular automata to generate infinite cave levels [15]. Search-based evolutionary algorithms for dungeon generation were also investigated by different researchers [16–20]. Roden and Parberry proposed a constrained-based pipeline for generation of underground levels using sub-graph topologies [21]. Hilliard et al. developed two other algorithms, they called *Span** and *Growth* [22]: The first one combines the minimal spanning tree computation with A* search. The second approach is based on simultaneous expansion of room and hallway elements. Both algorithms are only designed for array-based 2D-tile maps.

Linden et al. surveys the current state of research regarding procedural generation of dungeons, identifying 3D content as the less investigated topic [6]. Only a few researchers address this challenge, but their solutions are ad-hoc, implemented only for a particular game [14, 21]. The survey also points out another shortcoming of most of the discussed methods: Typically, the parameters for controlling the procedural generation are hard to understand for non-expert users and require a deep technical understanding, like choosing the appropriate fitness function for evolutionary optimization [23]. Another recent survey, which however only focuses on 2D maps, can be found in the work of Monaghan [24].

The aforementioned approach of Roden and Parberry shares most similarities with our work [21]: Their generation process creates levels using also undirected graphs embedded in 3D space. Constraints on the graph topology are used to control the generation. In contrast to our approach, graph nodes are utilized to place complete, prefabricated geometry sections of the target level. Other differences that characterize our work are a wide variety of control parameters, a pipeline supporting incremental generation and the possibility to combine procedural content with manually placed geometry.

III. Procedural Level Generation Pipeline

In the following sections, we describe the individual steps for the computation of the level graph data structure, followed by the generation of the final geometric representation. Each of the steps can be repeated as often as desired. Thereby, it is possible to focus processing on specific parts of the structure using a bounding volume that is defined by a three-dimensional hull mesh. Graph elements can also be added, removed, connected or modified manually, e.g. by setting a vertex at a specific position in the level that must be reachable by the player in any case.

A. Generation of the Base Graph

The base graph is used to describe the first abstract structure of the indoor level, where vertices are later mapped to rooms and edges to corridors. The base graph does not yet represent the final appearance of the level: It is randomly connected and contains just all possible candidate edges, from which the definite ones are then extracted in the next pipeline step. Its vertex set is generated automatically using jittering [25]. For this purpose, the current hull mesh is overlaid with an axis-aligned 3D grid. Then, graph vertices are instantiated randomly within the jitter cells and clipped w.r.t. the hull surface (and manually placed 3D models, if available). This process is determined by the following key parameters, mainly for controlling the density and irregularity rooms or junctions of the level: (a) 3D resolution of the grid,

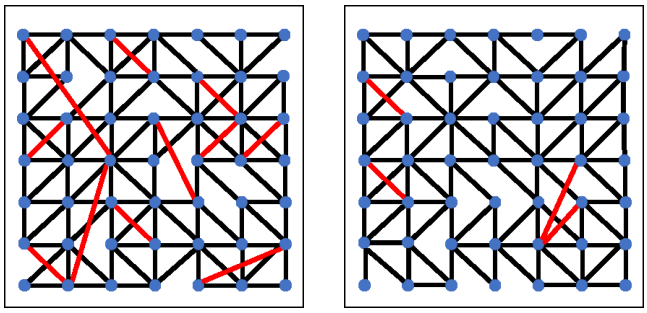

Fig. 1. Example of edge growing using depth-first (**left**) versus breath-first traversal (**right**). Intersecting edges (geometric violations) are red.

(b) probability for the instantiation of a vertex in a cell and (c) maximum vertex distance to the cell center. To create levels with a higher degree of irregularity, it is necessary to choose a lower grid resolution. However, in order to still achieve high vertex densities in such cases, the following two additional parameters are available to the level designer: (d) number range for instantiation of multiple vertices per cell and (e) maximum count of retries for failed clipping tests with the hull or user models.

The construction of the edge set is based on a growth algorithm, which starts from an arbitrary or manually defined seed vertex. It successively selects random vertices as targets for undirected edge connection from a nearest neighbor set. This set is recreated locally each time for the current source vertex. To avoid construction of equivalent edges, subsequent neighbor sets are built only from remaining graph vertices, i.e. that have no connection to the current source vertex. Growth is carried out using breath-first traversal [26]. This results in potentially fewer geometric intersections due to a more balanced edge expansion (Figure 1). To guarantee full graph connectivity, the algorithm only terminates when the set of remaining vertices becomes empty. A special case are vertices that have been placed manually on a 3D user model. All such vertices that are attached to the same model are interpreted as room entrances and assumed as implicitly interconnected (Figure 2, left). The *edge growth* process is ruled by the following parameters, primarily for controlling the density, spatial expansion and angular spreads of corridors of the later level: (a) minimal and maximal size for the nearest neighbor sets, (b) vertex degree range, (c) minimal and maximal angle between incident edges and (d) between edges and the 3D ground plane.

As an example, setting both angle interval limits that are listed under point (c) to 90 degrees and under (d) to 0 degrees will result in the generation of classic, i.e. planar and rectangular, mazes (Figure 2, right).

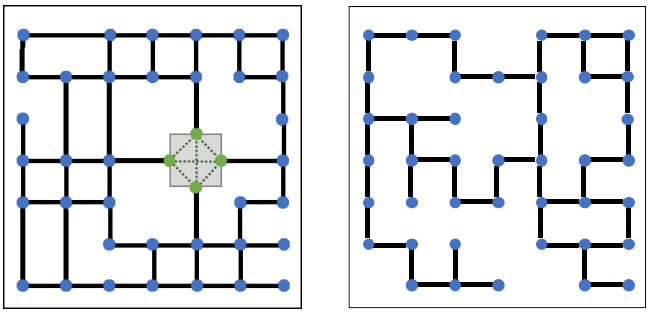

Fig. 2. **Left:** Green vertices represent implicitly interconnected room entrances of a user model (grey). **Right:** Example of a rectangular maze.

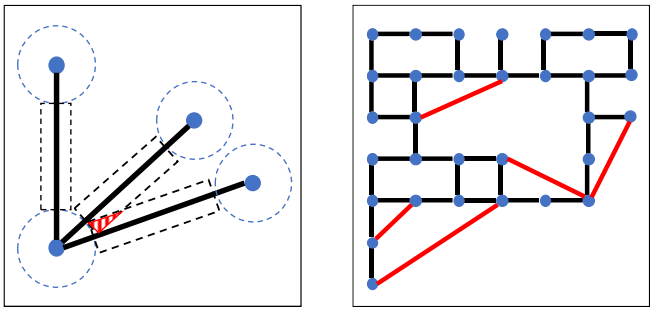

Fig. 3. **Left:** The red hatched area illustrates a geometric intersection of incident edges. **Right:** The lengths of the red edges exceed the maximally allowed relative deviation from the average length value.

B. Weighting and Computation of the Minimum Span Tree

The next pipeline stage involves the computation of a minimal spanning tree (MST) from the base graph. The MST is a cycle-free graph that connects all vertices. It can be interpreted as a *level skeleton*, more or less like a rig in skeletal animation, where each point can be reached via exactly one unique path [27]. MST algorithms require weighted edges and compute a tree with minimum total weight. In this work we exploit this fact by interpreting edge weights as violations of design preferences. Violations can be geometric intersections and unsatisfied parameter settings. To detect intersections, elements of the base graph are implicitly enclosed with simple proxy geometries: vertices with spheres and edges with cylinders, each with user-defined radii. All possible combinations of the following geometries are considered for edge intersection tests: vertices, other edges, current hull mesh and manually placed 3D models. Incident edges represent a special case: Here, overlaps are only considered outside the proxy geometry of the corresponding vertex, since otherwise intersections would always occur (Figure 3, left). Additionally, another parameter is introduced to control the MST generation: maximally allowed relative deviation of each edge length from the corresponding average value. This parameter can only be applied after the complete construction of the base edge set. It is used to determine the regularity of prospective corridor lengths (Figure 3, right).

Already during base graph generation, the number of violations is tried to be minimized as follows: On edge construction, randomly picked target vertices (from the current nearest neighbor set) are at first excluded in case of a violation. This vertex picking is repeated until all elements of the nearest neighbor set have been tested. Thereafter, if the number of constructed edges is less than the required degree minimum, edges with less violations are preferably included in the base graph. Obviously, violations cannot be completely avoided. Typical cases are geometric intersections caused by too high grid resolutions for vertex generation or too high vertex degree counts for edge construction. Another example refers to conflicting parameter settings, like wide angles between incident edges versus high vertex degrees.

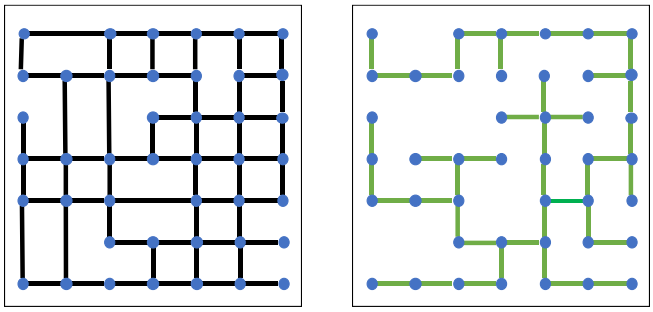

Fig. 4. **Left:** Example of a planar base graph (with rectangular edges) **Right:** A resulting minimum spanning tree.

The occurrence of violations can be minimized by executing the MST algorithm. In this context, the user can control this process by specifying preferences regarding the tolerance of individual violation types. This is accomplished by introducing numeric penalty factors for each violation category, i.e. for geometric intersections and for each base graph generation parameter. Then, prior to the computation of the MST, the final edge weights of the base graph are calculated using a linear combination of selective violation factors: Let $w_i \in \mathbb{R}$ be the weight for the i-th base graph edge, $f_t \in \mathbb{R}$ the penalty factor for violation type $t \in T$ and $s_i: t \to \{0,1\}$ the corresponding selection function, which assigns each violation type the value 0 or 1, depending on its occurrence for the current edge under consideration. Thus w_i is calculated with the following formula:

$$w_i = \sum_{t \epsilon T} s_i(t) \cdot f_t \qquad (1)$$

In our work we use Prim's algorithm to compute the MST from the weighted base graph [28]. Figure 4 shows an example result for a planar base graph.

C. Construction of the Level Graph and Merging

Due to its cycle-free property, the MST can already be interpreted as a labyrinth level. But in practice, dungeon-like game worlds often allow tours, offering the player alternative routes for level exploration. This limitation can be solved in the last pipeline stage by adding extra, hence *cycle-forming*, edges. Therefore, original base graph edges, which have been discarded during MST construction, can be reinserted into the graph. However, cycle-forming is now performed in a controlled manner: First, just a user-defined percentage of the base graph edges is considered for reinsertion. Secondly, edges with lower weights are selected with higher priority. And thirdly, edges with comparable weights can be picked randomly or based on a length prioritization (i.e. short versus long hallways), depending on the designer's preference.

The resulting cyclic graph can also contain edges with violations, just like the MST described in the previous section. So, in the final pipeline stage, the user can decide to neglect the graph connectivity criterion and instead discard violating edges (with non-zero weight) and associated incident vertices. This is implemented by inspecting vertices that belong to both, an edge i with weight $w_i = 0$, and another edge j with weight $w_j > 0$: If a subgraph, spanned by the violating edge j, contains too few non-violating edges, then all its vertices and edges are deleted (including edge j). The intensity of this *graph cleaning* process is determined by the following two user parameters: (a) threshold for the minimum number of non-violating subgraph edges and (b) minimum spanning edge weight for consideration (Figure 5).

The result is called the *level graph* and can either be used to generate the final geometric representation, or it can be connected to other previously generated level graphs. This connection, which we refer to as *graph merging*, can be used for facilitating procedural generation of complex indoor systems, consisting of multiple separately pre-generated and individually parameterized level structures. Therefore, all pipeline stages can be re-executed. However, graph merging excludes relationships within the same previously generated

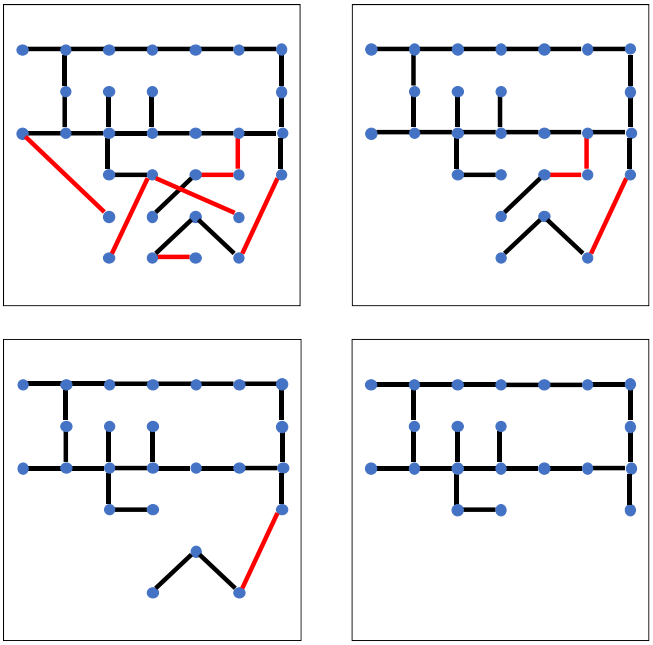

Fig. 5. **Top-Left**: Graph cleaning disabled. The other images show cleaned level graphs based on various thresholds for non-violating edges (T_{nv}): **Top-Right:** $T_{nv} = 0$, **Bottom-Left**: $T_{nv} = 1$, **Bottom-Right**: $T_{nv} = 2$.

vertex set. For example, the edge growth algorithm then only constructs connections between vertices of a newly-generated base graph and only between different pre-generated graphs. Figure 6 shows an example of a level structure that was created by merging two differently constructed graphs.

D. Conversion to Geometric Representation

The final indoor level is generated by conversion of the level graph data structure to a geometric representation. This process requires a set of pre-modeled 3D components: rooms (or junctions), dead ends, horizontal and vertical corridors (typically hallways and staircases), doors and gap fillers. Each of these components must be uniquely labeled, as it is used specifically during conversion: Graph vertices are mapped to rooms or dead ends, depending on whether the vertex degree is greater than one; edges to vertical or horizontal corridors, depending on the helix angle. Doors and gap fillers are used to join corridors and rooms. If there are multiple components with the same label (and of same size), then one is picked randomly for each conversion step.

The process begins with the instantiation of a room or dead end model for each vertex. These objects are oriented to one

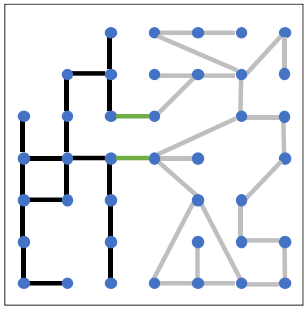

Fig. 6. Merging of two different level graphs (with black and grey vertices). Green edges illustrate new connections resulting from graph merging.

randomly selected incident graph edge. In case of a geometric room intersection, the algorithm tries to resolve it by selecting a smaller model, if available. The next step is the conversion of edges to corridor models. Depending on the length of the current edge, the following Constructive Solid Geometry (CSG) operations are executed: Trimming of too long corridor models or random stitching of several too short elements. And finally, CSG operations are also used to connect rooms and corridors, where door components are placed at the corresponding junctions and used to cut out room entrances. If the angular deviation between a room and a corridor is too large, then gap filler components are inserted in order to obtain a closed 3D model.

An overview of the entire incremental procedural generation pipeline, including base graph vertex generation, edge growth, graph weighting, MST computation, cycle-forming, graph cleaning, level graph merging and geometrical conversion is shown in Figure 7.

IV. Implementation and Results

Our procedural generation pipeline was implemented using the C# programming language as a plug-in for the Unity 3D game engine [11]. The current version of the graphical user interface is on the development stage of a scientific prototype, but already allows the parameterization and triggering of all described processes. By encapsulating all graph elements in corresponding game engine objects, all settings and the execution of corresponding algorithms can be accomplished with Unity's "Inspector Window". Furthermore, the "Scene View" is used for visualization of current generation results as well as for defining the hull mesh, manual manipulation of the graph structure and placement of user-defined models.

Figures 8 to 12 show our first experimental results: The generation of the 2D maze is based on 7 by 7 regularly and

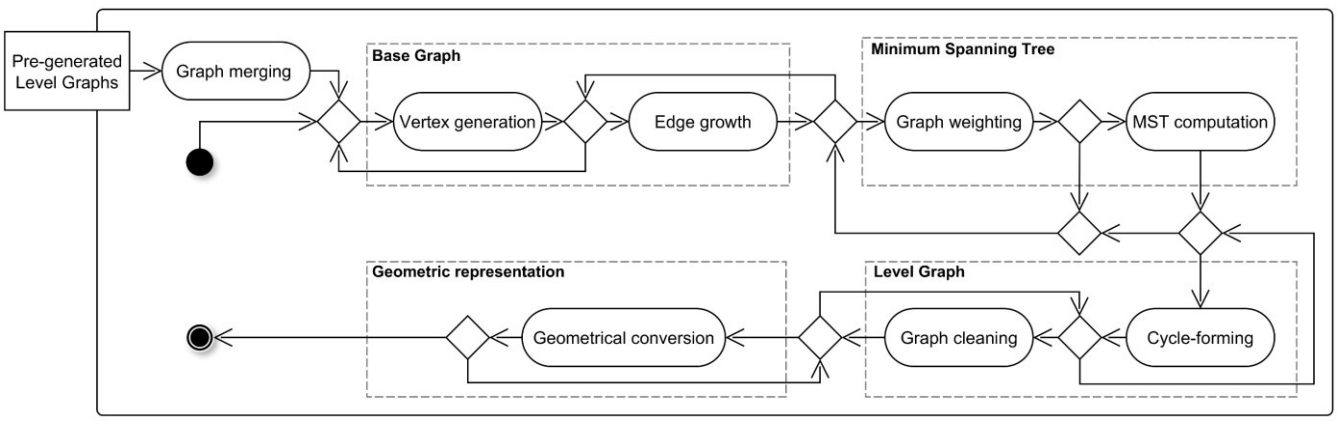

Fig. 7. Overview of our procedural level generation pipeline, including the main pipeline stages and corresponding processes (UML activity diagram).

TABLE I. BASE GRAPH COMPLEXITIES AND MEMORY USAGES

	Base graph		Memory usage *(in MB)*
	Nodes	***Edges***	
2D Maze	49	64	0.04
3D Maze	216	258	0.29
Dungeon	89	185	0.12
Space station	1056	2310	13.08

planarly arranged base graph vertices, edge growth with perpendicular corridors and insertion of 50% cycle-forming edges. The structure of the 3D cube is inspired by the "Borg cube", known from the TV series "Star Trek". It is supposed to illustrate the capability of our method to generate three-dimensional levels in a single step. In our case, it actually corresponds to a 3D version of the rectangular 2D maze. The dungeon represents a more practical level design example. It shows a dwarf mine in a fantasy setting, located inside a mountain. The black area was used to define the hull mesh for base graph generation. The grey round object is a manually placed 3D arena model that was also taken into account during level graph construction and considered e.g. in intersection testing. The different mine sections were generated iteratively with specific parameters. The resulting individual level graphs were connected to each other by graph merging to form the final level. A close-up of the dungeon (arena section) is shown in Figure 11. The last presented result is a space station. It is not intended to be a playable level, but rather an example for procedural generation of highly complex structures in order to test the computational potential of our method. Comparable to the previous case, multiple procedural generation iterations were performed, only this time using primitive bounding volume hull meshes (spheres and cuboids). Table I shows an overview of the base graph complexities used to generate the presented examples. Since minimal spanning trees (MSTs) and level graphs are respective subgraphs, their complexity values would not exceed the presented results. The estimates of memory consumption are based on an edge list representation. The total memory consumption of the final level is essentially determined by the geometric complexity of the underlying 3D component set.

Table II summarizes time measurements for the main pipeline stages, i.e. base graph generation (including vertex and edge sets), MST computation (with graph-weighting) and level graph construction (including graph cleaning and cycling-forming). All measurements were performed on a gaming laptop with an Intel Core i7-8750H CPU, Nvidia GTX 1070 Max-Q (8GB) graphics card and 32 GBs RAM, running the Microsoft Windows 10 operating system (version 1809). It becomes apparent that the performance of the level graph pipeline is basically adequate for practical use, and especially for iterative procedural generation. Typically, the measured time values are clearly below one second. An exception is the space station due to its artificially high complexity (and not yet implemented spatial acceleration data structures). The bottleneck in the current (non-optimized) implementation is the geometric conversion, and in particular the execution of

TABLE II. TIME MEASUREMENTS FOR THE MAIN PIPELINE STAGES

	Base graph	***MST***	***Level graph***
2D Maze	47 ms	21 ms	9 ms
3D Maze	574 ms	43 ms	21 ms
Dungeon	236 ms	33 ms	13 ms
Space station	43.75 s	381 ms	181 ms

TABLE III. TIME COMPLEXITIES OF USED GRAPH ALGORITHMS

	Time complexity
Vertex generation	$O(\|V\|)$
Weighted edge construction	$O(\|E\|^2)$
Prim's MST computation [29]	$O(\|E\| \cdot log\ \|V\|)$
Cycle-forming	$O(\|E\|)$
Graph cleaning	$O(\Delta(G) \cdot \|V\| \cdot \|E\|)$

the Constructive Solid Geometry (CSG) operations. The measured times range from about 3.5 minutes for the dungeon level to ca. 5 hours for the space station. Although CSG operations only need to be executed once after completion of the level design, the particular results are not yet satisfactory and require optimization. Table III gives an overview of the worst-case time complexities of used graph algorithms (for vertex set V, edge set E, maximum graph degree $\Delta(G)$).

V. CONCLUSIONS AND FUTURE WORK

Our level generation pipeline proves to be versatile and basically suitable for practical use. Level designers can extend their established working methods by utilizing the presented procedural techniques. This creates the potential to increase time efficiency and make the construction of indoor levels more convenient by exploiting iterativity and controlled randomness through re-running of steps and selecting the most promising results. The presented time measurements show that runtimes for generating the graph structures are small enough to implement our iterative level design approach. Similarly, the corresponding data structures require little memory resources, which are almost negligible in the context of the memory sizes required for 3D geometric models.

The elaborated parameters for controlling the procedural generation pipeline are basically intuitive for understanding. However, the currently implemented graphical user interface is rather more suitable for advanced users with a mathematical understanding of the pipeline stages. Here, it is necessary to design an interface for non-experts in such a way that graph parameters are mapped to simpler terms (e.g. "vertex degree" to "corridor density") or combined into easily understandable meta-parameters. Likewise, predefined restrictions regarding mutually exclusive parameter configurations, as well as an immediate visualization of corresponding violations, would be recommended for improvement. Future work also includes further development of our geometric conversion algorithms and in particular optimizations of the Constructive Solid Geometry operations, which can be a time-critical bottleneck for very complex levels. Examples are the support for more flexible and configurable 3D components and use of spatial acceleration techniques, respectively. Another useful future development step would be the procedural placement of game design elements, like enemies, non-player characters, pick-up and cover objects. Finally, we plan to carry out additional practical user tests, also to be able to conduct representative comparative studies for evaluating the efficiency of our procedural approach in relation to manual level design.

ACKNOWLEDGMENT

We would like to thank the Game Design faculty at the Mediadesign University of Applied Sciences in Munich, Germany, for supporting the implementation of our initial level graph software prototype. We would also like to thank Peter Eichinger from the University of Erlangen-Nuremberg, who contributed to the further development of our software.

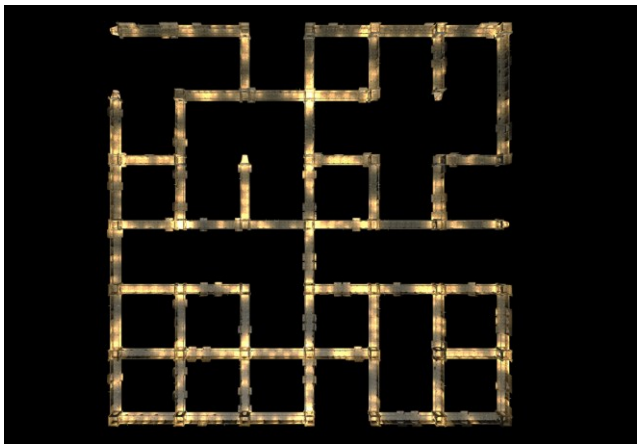

Fig. 8. 2D maze – Example of a planar, rectangular level.

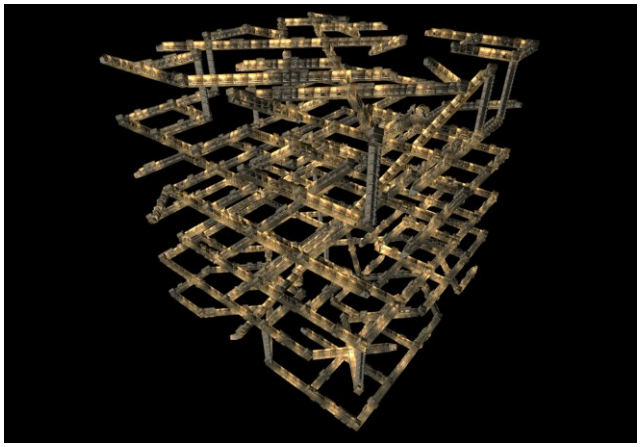

Fig. 9. 3D maze – Example for one-step 3D level generation.

Fig. 10. Dungeon – Practical level design example with multiple sections.

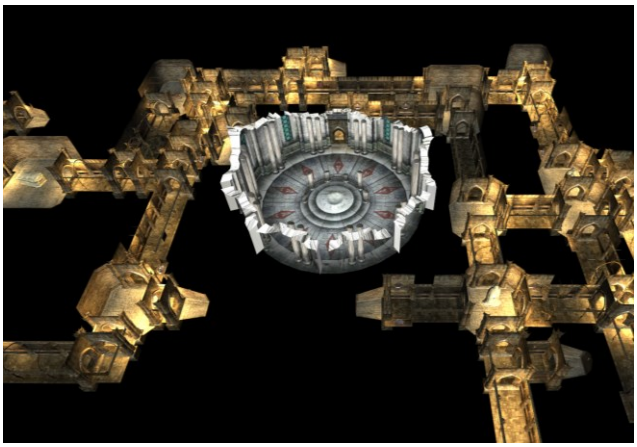

Fig. 11. Close-up of procedural dungeon (arena section).

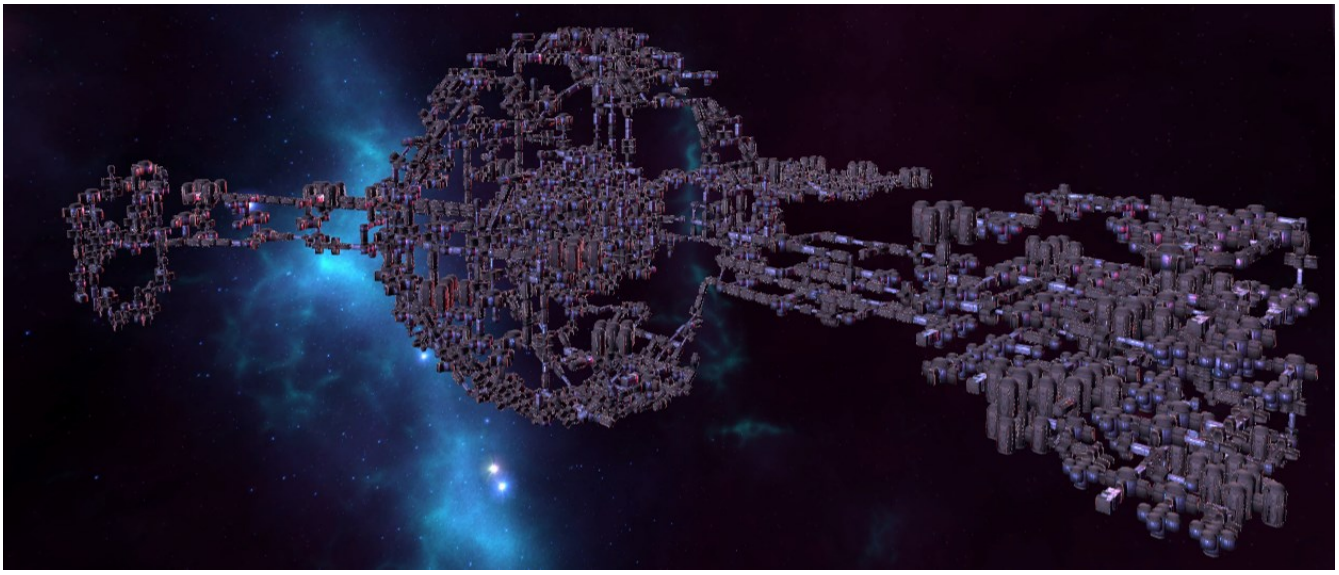

Fig. 12. Space station – Example for procedural generation of highly complex 3D structures (iterative construction using simple hull mesh geometries).

REFERENCES

[1] M. Hendrikx, S. Meijer, J. van der Velden, and A. Iosup, "Procedural Content Generation for Games: A Survey," *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 9, no. 1, 1–22, 2013.

[2] B. Lister, *Red Dead Redemption 2 Required 8 Years And A Massive Team To Develop.* [Online] Available: https://gamerant.com/red-dead-redemption-2-8-years/. Accessed on: Mar. 18 2019.

[3] I. Bell and D. Braben, *Elite.* [Online] Available: http://www.iancgbell.clara.net/elite/. Accessed on: Mar. 21 2019.

[4] M. Toy, G. Wichman, K. Arnold, and J. Lane, *Rogue*, 1980.

[5] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games*. Cham, s.l.: Springer International Publishing, 2016.

[6] R. van der Linden, R. Lopes, and R. Bidarra, "Procedural Generation of Dungeons," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 1, pp. 78–89, 2014.

[7] Bethesda Game Studios, *The Elder Scrolls IV: Oblivion*: Bethesda Softworks, 2006.

[8] S. K. Card, J. D. Mackinlay, and B. Shneiderman, Eds., *Readings in Information Visualization: Using Vision to Think*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc, 1999.

[9] S. Krishnan and D. Manocha, "An Efficient Surface Intersection Algorithm Based on Lower-dimensional Formulation," *ACM Transactions on Graphics*, vol. 16, no. 1, pp. 74–106, 1997.

[10] A. M. Pereira, M. C. Arruda, A. C. D. Miranda, W. W. Lira, and L. F. Martha, "Boolean Operations on Multi-region Solids for Mesh Generation," *Engineering with Computers*, vol. 28, no. 3, pp. 225–239, 2012.

[11] Unity Technologies, *Unity 3D.* [Online] Available: https://unity.com/de. Accessed on: Mar. 20 2019.

[12] J. Dormans, "Level Design As Model Transformation: A Strategy for Automated Content Generation," in *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, 2011, 2:1 - 2:8.

[13] J. Dormans, "Adventures in Level Design," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, Monterey, California, 2010, pp. 1–8.

[14] R. van der Linden, R. Lopes, and R. Bidarra, "Designing Procedurally Generated Levels," in *Proceedings of the 2nd Workshop on Artificial Intelligence in the Game Design Process*, 2013.

[15] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular Automata for Real-time Generation of Infinite Cave Levels," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 2010, 1–4.

[16] K. Hartsook, A. Zook, S. Das, and M. O. Riedl, "Toward Supporting Stories with Procedurally Generated Game Worlds," in *IEEE Conference on Computational Intelligence and Games*, Seoul, Korea (South), 2011, pp. 297–304.

[17] V. Valtchanov and J. A. Brown, "Evolving Dungeon Crawler Levels with Relative Placement," in *Proceedings of the 5th International C* Conference on Computer Science and Software Engineering*, Montreal, Quebec, Canada, 2012, pp. 27–35.

[18] D. Ashlock, C. Lee, and C. McGuinness, "Search-Based Procedural Generation of Maze-Like Levels," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 260–273, 2011.

[19] C. McGuinness and D. Ashlock, "Decomposing the Level Generation Problem with Tiles," in *IEEE Congress on Evolutionary Computation*, 2011, pp. 849–856.

[20] A. Liapis, G. N. Yannakakis, and J. Togelius, "Sentient Sketchbook: Computer-Aided Game Level Authoring," in *Proceedings of the 8th International Conference on Foundations of Digital Games*, 2013, pp. 213–220.

[21] T. Roden and I. Parberry, "From Artistry to Automation: A Structured Methodology for Procedural Content Creation," in *Lecture Notes in Computer Science*, vol. 3166, *Entertainment Computing*, M. Rauterberg, Ed., Berlin, Heidelberg: Springer, 2004, pp. 151–156.

[22] N. Hilliard, J. Salis, and H. ELAarag, "Algorithms for procedural dungeon generation," *Journal of Computing Sciences in Colleges*, vol. 33, no. 1, pp. 166–174, 2017.

[23] A. Baldwin, S. Dahlskog, J. M. Font, and J. Holmberg, "Mixed-Initiative Procedural Generation of Dungeons Using Game Design Patterns," in *IEEE Conference on Computational Intelligence and Games*, 2017, pp. 25–32.

[24] Z. Monaghan, "Comparing Procedural Content Generation Algorithms for Creating Levels in Video Games," Technological University Dublin, School of Computing, 2019.

[25] J. F. Thompson, Ed., *Handbook of Grid Generation*. Boca Raton: CRC Press, 1999.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms,* 3rd ed. Cambridge, Mass.: MIT Press, 2009.

[27] A. Beane, *3D Animation Essentials*. Hoboken: John Wiley & Sons, 2012.

[28] R. C. Prim, "Shortest Connection Networks and Some Generalizations," *Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[29] S. Pettie and V. Ramachandran, "An Optimal Minimum Spanning Tree Algorithm," *Journal of the ACM*, vol. 49, no. 1, pp. 16–34, 2002.