Procedural Game Level Generation by Joining Geometry with Hand-Placed Connectors*

Rafael C. e Silva[1], Nuno Fachada[1,2], Nélio Códices[2,1], and Diogo de Andrade[1]

[1] Lusófona University, Campo Grande, 376, Lisbon, Portugal
castroesilva.rafael@gmail.com,
{nuno.fachada,nelio.codices,diogo.andrade}@ulusofona.pt
[2] Instituto Superior Técnico, Av. Rovisco Pais, 1, Lisbon, Portugal

Abstract. We present a method for procedural generation of 3D levels based on a system of connectors with pins and human-made pieces of geometry. The method avoids size limitations and layout constraints, while offering the level designer a high degree of control over the generated maps. The proposed approach was originally developed for and tested on a multiplayer shooter game, nonetheless being sufficiently general to be useful for other types of game, as demonstrated by a number of additional experiments. The method can be used as both a level design and level creation tool, opening the door for quality map creation in a fraction of the time of a fully human-based approach.

Keywords: Procedural content generation · Video games · 3D levels · Gridless generation · Mixed-initiative content creation

1 Introduction

In this paper we describe a method for procedural generation of 3D levels for video games. The proposed method uses a system of connectors with pins, similar in concept with a jigsaw piece or an electrical plug and outlet, to connect pieces of pre-made geometry for generating levels. This approach allows the level designer to avoid size limitations and layout constraints, such as in the case of grid-based layouts, while offering full control on the design of individual geometry components, including the placement and customization of associated connectors, using the chosen game engine's visual editor. Additionally, the generation algorithm is highly customizable, with several parameters for a designer to tinker with and control the properties and layout of the final level. This technique follows a mixed-initiative methodology, where both human and computer are actively involved in the level design process [8]. As such, while the levels are procedurally generated, the method still offers a degree of control to the level designer.

* Supported by Fundação para a Ciência e Tecnologia under Grant No.: UIDB/05380/2020 (HEI-Lab)

The proposed procedural content generation (PCG) method was initially developed for *Trinity*, a multiplayer third person shooter [13], developed as a semester project at Lusófona University's Bachelor in Videogames [5]. Given the frenetic and fast paced nature of the game, in which players have multiple options for mobility, the main goal was for the generated levels to encourage this type of gameplay, facilitating navigational flow.

The paper is organized as follows. In Section 2, we review related work concerning the use of PCG for level design, with special focus on the inspirations and references that lead to the core idea of our approach. In Section 3, we describe the proposed technique, namely the components that make up the system and the details of the PCG algorithm. Several case studies are presented in Section 4, including one where the method was used for generating levels for the *Trinity* videogame. Section 5 closes the paper, discussing potential improvements and offering some conclusions.

2 Background

Mixed-initiative content creation (MICC) [8], in which a mix of human input and computer-assisted PCG are used for game design, is a promising area in game development in general and level design in particular. Game level design and testing can be notoriously laborsome and time consuming [12], and with cost-effectiveness in mind, a number of level design tools, as well as full games, have been making use of computer-aided approaches. Tanagra is one such tool for 2D platformers [14], allowing human designers to partially specify a level's geometry and pacing, leaving up to the computer to fill in the gaps. Tanagra guarantees that generated levels are playable when human-defined specifications are valid.

Oblige is a MICC level generator for the DOOM family of games [2]. It allows the level designer to set a number of parameters, such as level size and approximate quantities of each type of level section (outdoors, caves, hallways, etc.), each type of monster, and of each type of power-up. Levels are created using shape grammars [10] on a grid-based layout, and are limited to a single floor – an inherent limitation of the DOOM family of games.

In the context of FPSEvolver, a Counter-Strike-like videogame, Ølsted et al. [11] proposed a novel grid-based interactive evolution approach for generating multiplayer maps according to the players' preferences. Players vote on a selection of evolving scenarios, with the goal of generating levels in accordance with what they consider to be a good map.

Looking at comercial games, roguelikes such as *Spelunky Classic* [17], *The Binding of Isaac: Rebirth* [9] and *Enter the Gungeon* [4] all use MICC. *Spelunky*, for example, uses premade room templates to fill out a grid. Rooms with specific characteristics such as top entrances and bottom pits are considered when generating levels in order to create a valid path for the player to traverse to the end [18,7].

The Binding of Isaac: Rebirth [9] (*Isaac*) uses MICC to create its map by connecting several rooms together [6,15]. These rooms are fit into a grid. However, each room may take more than one grid space, and each grid space it occupies can be connected to by other rooms occupying adjacent grid spaces. This allows for big rooms to connect to small rooms and vice-versa. Fig. 1 shows some screenshots of the "minimap" of a level displaying all the rooms discovered.

Fig. 1: Several minimaps of the full layout of a level in *The Binding of Isaac: Rebirth*.

Enter the Gungeon also features MICC, though it does not connect its rooms directly to one another as in *Isaac*, employing a more complex algorithm to obtain the desired layout. This algorithm uses nodes and connectors, placing different pre-made rooms as those nodes and afterwards creating corridors to join the rooms for the final map layout. Not only is the hand of the human designer present in the rooms, it is also visible in the overall layout of the level, as there seems to be a set of predefined general level layouts that make the algorithm place certain rooms in certain orders and block some paths [3].

3 Methods

The proposed approach falls under the MICC methodology, since the level designer controls the type of levels to be created, although these are procedurally generated. The generation process is highly configurable. For our prototype we used the Unity game engine [16] and leveraged its editor tools to handle the input of the human designer.

In broad terms, one or more **connectors** (each containing one or more **pins**) are placed in the geometry of a **level piece**, and then several level pieces are fed into the **generation manager**, composed by a number of generation parameters to be specified by the level designer. The generation manager, containing the level pieces and the generation parameters, then passes this data to the **generation algorithm**, which produces a fully playable map by connecting level pieces according to their connectors and the specified generation parameters. The level generation process is summarized in Fig. 2. Level pieces, connectors and pins

are detailed in Subsection 3.1. The generation manager and the generation parameters are discussed in Subsection 3.2. Finally, the generation algorithm is presented in Subsection 3.3.

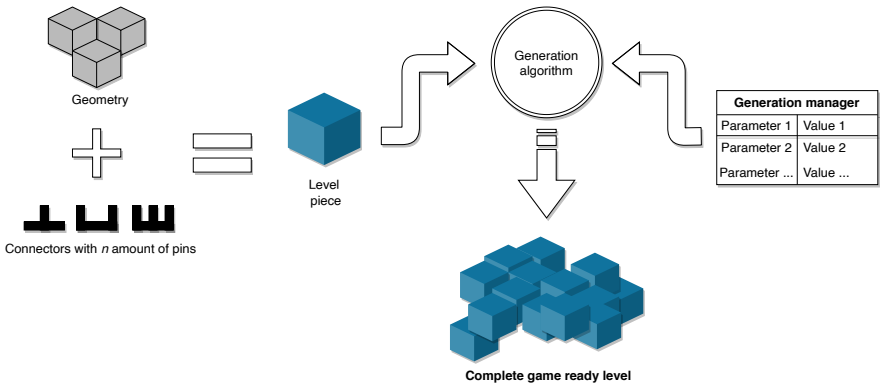
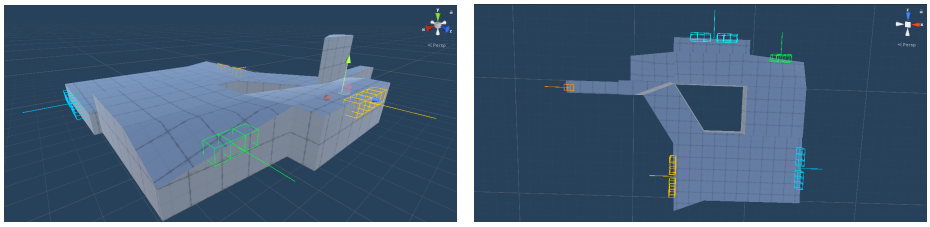


Fig. 2: Summary of the level generation process.

3.1 Level Pieces, Connectors and Pins

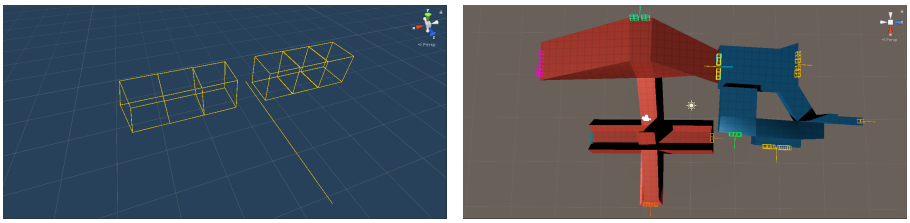
Level pieces are sets of geometry that include one or more connectors and any other elements the level designer wishes to spawn with the piece. These pieces are attached to each other by their connectors during the generation process. In Fig. 3, we see an example of how a level piece looks like in the Unity engine editor, where the connectors are visible. Note that for a piece to be usable by the generation algorithm it needs to have at least one associated connector.

Connectors are the components that determine where two level pieces will join. Each connector has a given number of pins which determine what other connectors it can connect with. The number of pins in a connector is illustrated by colored three-dimensional shapes, as shown in Fig. 4. Connectors are aware of their heading, represented in Fig. 4a by the straight line. Connectors can only form pairs with each other if they have an equal number of pins, as shown in Fig. 4b, or if the pin count difference is within a tolerance parameter set in the generation manager. When two connectors are matched, the piece being evaluated (*tentative piece*) will move so its connector and the connector of the selected placed piece (*guide piece*) are facing each other and at a distance defined by the designer in the generation manager (zero by default, connectors will overlap). Pieces can also overlap, but if any kind of geometry overlapping is undesired, the generation manager provides a clipping correction option to use physics simulation to push the pieces apart, as described in reference [1].



(a) Perspective view. (b) Top-Down view.

Fig. 3: A level piece with its various connectors visible.



(a) Isolated 6-pin connector with its heading represented by the straight line. (b) Two level pieces joined at their compatible connector.

Fig. 4: Connectors and geometry forming level pieces.

3.2 Generation Manager and Generation Parameters

The generator manager is where the designer can define the level piece prototypes to be used in the level, as well as specify the generation parameters, summarized in Table 1.

The `genMethod` parameter is crucial to the algorithm. There are four generation methods which produce different types of map, as shown in Fig. 5. These are as follows:

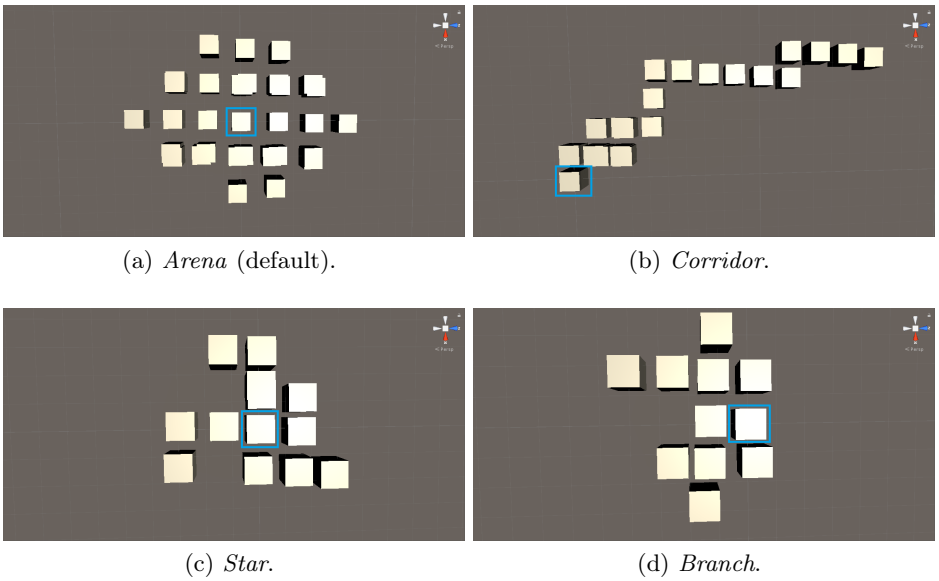
- The *arena* generation method (Fig. 5a) aims to create maps that sprawl in all directions, covering a large area with geometry.
- The *corridor* generation method (Fig. 5b) aims to create long, narrow levels where the geometry seemingly follows a line.
- The *star* generation method (Fig. 5c) is a mix of the arena and corridor generation methods, creating corridors sprawling from the starting piece and ending when that piece has no empty connectors.
- The *branch* generation method (Fig. 5d) creates branches in the same manner as the star method, however it does not return to the starting piece, choosing instead a previously placed piece to start a new branch, repeating this process however many times possible.

Table 1: Generation manager global parameters.

Parameter	Description
`pieceList`	List of level piece prototypes to be used by the generator to create the level.
`useStarter`	Boolean. If `True`, the generator will select the first level piece from the `starterList` instead of the `pieceList`.
`starterList`	List of level piece prototypes to be used as the first level piece if `useStarter` is `True`.
`genMethod`	Selection of generation method. Available options are *arena* (default), *corridor*, *star* and *branch*.
`starterConTol`	The starting piece is selected among the set of pieces with most connectors (*arena* and *star* generation methods) or fewer connectors (*branch* and *corridor* generation methods), $n_{\max}$ or $n_{\min}$, respectively. The `starterConTol` parameter is an integer representing a tolerance, in number of connectors, from the piece(s) with most (or fewer) connectors, allowing pieces with as few as $n_{\max} - n_{\texttt{starterConTol}}$ (or as much as $n_{\min} + n_{\texttt{starterConTol}}$) connectors to be selected as the starting piece.
`maxPieces`	Integer. The maximum number of pieces the generator will place after placing the starting piece.
`pinTolerance`	Integer. The maximum allowed difference between pin counts in two connectors to allow them to be paired up.
`fixClipping`	Boolean. If `True`, the generator will use the available physics system to push overlapping pieces apart. Otherwise allows for overlapping geometry.
`pieceDistance`	Real value. Represents the spacing between two connected connectors. Works independently of `fixClipping`.

Table 2: Specific parameters for the *star* and *branch* generation methods.

Parameters	Description
`branchPieces`	Integer. Defines how many pieces will a branch contain in average.
`branchPiecesVar`	Integer. Represents the maximum variance from `branchPieces` in each branch.
`pieceSkip`	Integer. Defines how many pieces will the algorithm skip ahead from the starting piece when finding a piece to start a new branch. Only valid for the *branch* generation method.
`pieceSkipVar`	Integer. Represents the maximum variance from `pieceSkip` on each skip. Only valid for the *branch* generation method.



(a) *Arena* (default). (b) *Corridor*.

(c) *Star*. (d) *Branch*.

Fig. 5: Example outputs of the level generation methods. The blue square indicates the starting piece.

The different generation methods achieve their aims by guiding how the generation algorithm chooses the **starting piece** and the next **guide piece**, as discussed in the next subsection. Both the *star* and *branch* methods require a few additional parameters, described in Table 2, the purpose of which will become clear in the next subsection.

3.3 Generation Algorithm

The pseudo-code of the generation algorithm is presented in Algorithm 1. Details of how each of the algorithm steps works are given in the next subsections.

Selecting the starting piece The algorithm begins by selecting and placing the starting piece of the level in the game world. The method of selecting this piece depends on the generation method chosen. For the *arena* and *star* generation methods, the piece with most connectors is selected. Conversely, for the *corridor* and *branch* generation methods, the piece with fewer connectors is selected. If there are multiple pieces with the same highest/lowest number of connectors, one of them is picked at random. In any case, if the `useStarter` option is selected then it will override the generation method and instead choose a random piece from `starterList`.

Algorithm 1: Level generation.

```
1  startingPiece ← genMethod.SelectStartingPiece()
2  guidePiece ← startingPiece
3  for i ← 0 to maxPieces do
4      connection ← none
5      do
6          failCount ← 0
7          if i > 0 then
8              guidePiece ← genMethod.SelectGuidePiece()
9          do
10             tentativePiece ← piecesList.GetRandomItem()
11             connList ← guidePiece.GetConnectionsWith(tentativePiece)
12             if connList is not empty then
13                 connection ← connList.GetRandomItem()
14             else
15                 failCount ← failCount + 1
16         while connection is none and failCount < maxFails
17     while connection is none
18     newPiece ← tentativePiece.Clone()
19     guidePiece.Join(newPiece, connection)
```

Selecting the guide piece When the main loop of the algorithm begins, the starting piece is selected as the guide piece. In subsequent iterations, the generation method will influence this selection as follows:

- In the *arena* generation method, the algorithm checks if the current guide piece has unused connectors and if so, keeps it as the guide piece. Otherwise, the piece placed immediately after the current guide piece is selected as the next guide piece.
- In the *corridor* generation method, the guide piece in each iteration is always the most recently placed piece.
- In the *star* generation method, when a new piece is placed, a counter is incremented. Until the counter reaches `branchPieces` ± `branchPiecesVar`, that piece remains the guide piece. When the counter hits the limit, it resets and the starting piece is selected as the guide piece if it still has unused connectors. Generation ends when the starting piece no longer has unused connectors.
- In the *branch* generation method, when a new piece is placed, a counter is incremented. Until the counter reaches `branchPieces` ± `branchPiecesVar`, that piece remains the guide piece. When the counter hits the limit, it resets and, counting from the starting piece, a jump of `piecesSkip`±`piecesSkipVar` pieces is performed. The next guide piece is the piece where the jump "lands".

Selecting and evaluating a tentative piece A tentative piece is randomly selected from `pieceList`. All possible connector pairings between the guide piece and the tentative piece are evaluated. Valid pairings are stored in a temporary list. Then one of these valid pairings is selected at random. A connector pairing is considered valid if, and only if, the following conditions are true:

1. Both connectors are unused, thus available for pairing.
2. The pin counts of both connectors is the same or its difference is equal or less than `pinTolerance`.

Selecting more pieces and moving on If the previous step yielded a valid result, a new piece is created by cloning the tentative piece. The new piece is then placed in its correct position in the game world. If, however, no valid pairing was found in the previous step, the algorithm will keep the same guide piece and randomly select another tentative piece from `pieceList`. This process is repeated until a valid pairing is found or a limit of failed attempts is reached for the current guide piece. This process will continue until there are no more valid pieces or connectors to select in the game world, or the maximum number of pieces has been placed.

4 Case Studies

4.1 *Trinity* – a Third Person Multiplayer Shooter

The proposed PCG algorithm was used to create the levels – arenas – for *Trinity*, a competitive multiplayer game [13]. The competition between players takes the form of shooting projectiles at each other while dashing, jumping and running around the map to dodge incoming enemy projectiles. The game can be seen in Fig. 6.

The PCG approach to level design was chosen since it is our belief that having players learning and adapting to a new map every match would promote the frenetic nature of the game, while boosting its replay value. However, given the game's focus on player vs player, fully AI-controlled design of high quality maps would be difficult to achieve, hence the need for some level of input from human designers, paving the way for a MICC approach.

Trinity was developed with the Unity game engine, which offered a number of important features for the implementation of the proposed PCG algorithm, namely:

- Parameterization of the generation manager was done via the object inspector, as show in Fig. 7.
- The built-in physics engine performed the necessary piece adjustments required when the `fixClipping` parameter is activated.
- The *prefab* system allowed designers to create individual level piece prototypes, as well as simplifying their cloning and deployment in the levels.

Fig. 6: Screenshot of *Trinity* during a match.

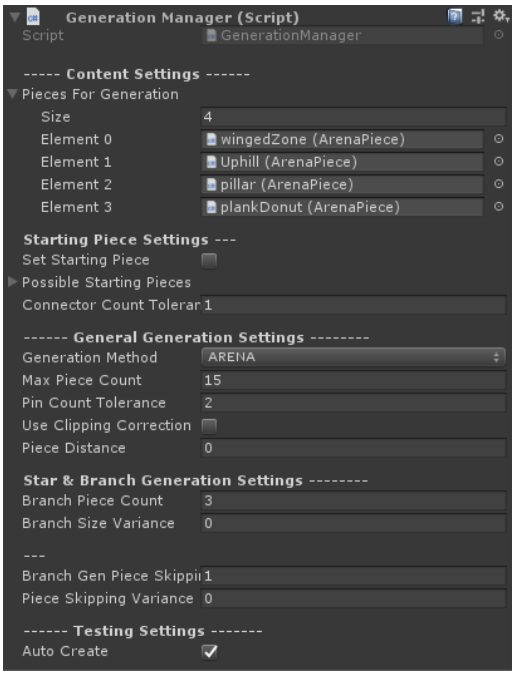


Fig. 7: Parameterization of the generation manager using Unity's inspector.

In the final game, arenas were always generated with the same parameters. However, the MICC approach allowed us to quickly iterate over a number of designs until we were satisfied with the style of arenas being generated. We opted for the *arena* generation method, since it creates large areas without holes where players could fall through – something which was undesirable for this particular game.

4.2 Other experiments

While developing the Unity implementation of the proposed PCG algorithm, we performed a number of experiments which provided us with a better understanding of the algorithms' capabilities, as well as of its limitations. Several levels generated during these experiments, and their relevant generator parameters, are shown in Fig. 8.

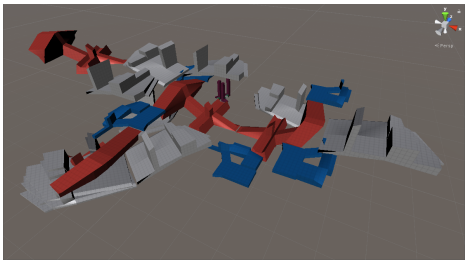
Figs. 8a–8d display experimental levels created with the different generation methods. In Fig. 8e, we used the `pieceDistance` parameter to create a level with "islands" and made the connectors visible in order to help visualize the connections.

Although our approach frees the designer of grid and space restrictions when creating level pieces, care must be taken in their design in order to maintain cohesion and predictability of the generated output, particularly in preventing extreme piece overlapping and looping. In fact, the design of the pieces factors heavily in how generator parameters are chosen. We found that the unique layouts of each generation method became somewhat less obvious as `maxPieces` pieces increased. Some loss of shape can be observed in Fig. 8f, which shows a level with 176 pieces generated with the *corridor* method. Nonetheless, the intended core layout is still noticeable, since the level presents the typical *corridor* "tips", indicating the beginning and end pieces of the generation. Compare, for example, with Fig. 8b.

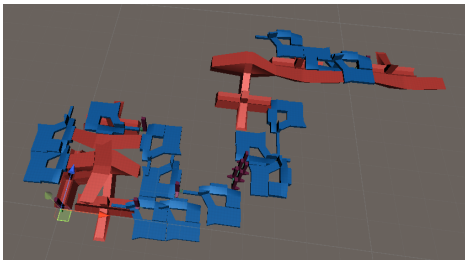
We also experimented with the algorithm in a way slightly different than originally intended. We created small level pieces, props and smaller obstacles, and then reserved all connectors with one pin to unite those small props with bigger pieces of geometry that would otherwise be "empty" without these details. An unpleasant side effect of this was that the levels are much smaller for the `maxPieces` amount given, and the pieces piled on each other. Some examples of this are shown in Figs. 8g and 8h. However, the general idea has great potential as it opens the door for the final design of the level pieces to also be procedurally generated.

5 Conclusions and Future Work

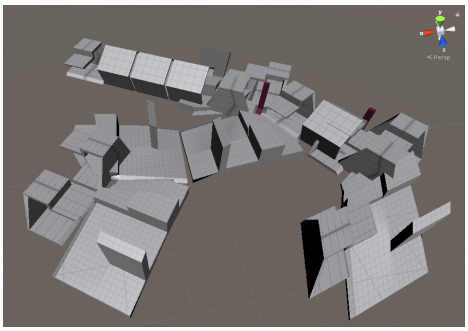
In this paper we presented a MICC-based PCG approach for creating levels primarily aimed at 3D competitive multiplayer games. The proposed method can be used as both a level design and level creation tool, allowing for fast iteration and speeding up development. While the human designer cannot directly manipulate



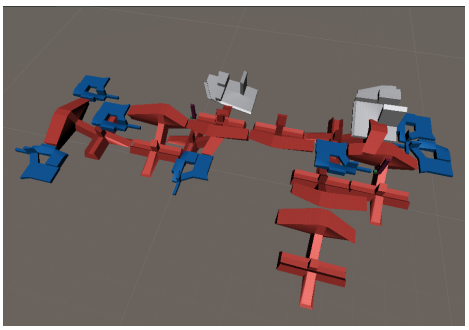
(a) `genMethod` = *arena*, `maxPieces` = 20.



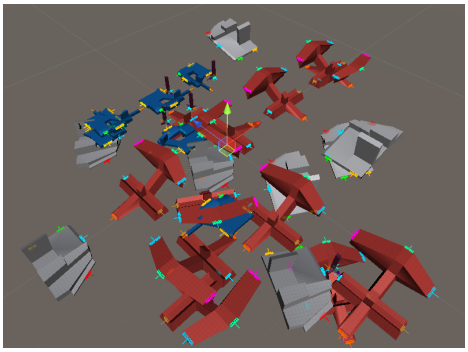
(b) `genMethod` = *corridor*, `maxPieces` = 30.



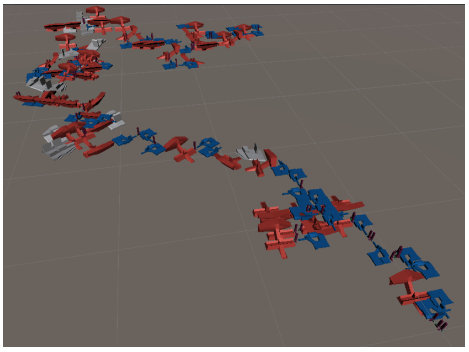
(c) `genMethod` = *star*, `branchPieces` = 8.



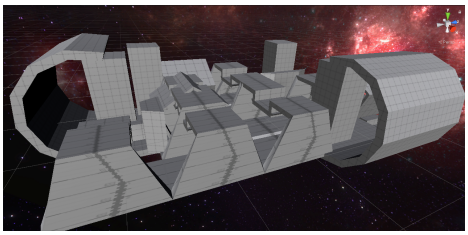
(d) `genMethod` = *branch*, `maxPieces` = 20, `branchPieces` = 8, `pieceSkip` = 1.



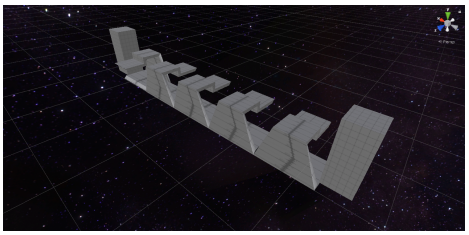
(e) `genMethod` = *arena*, `maxPieces` = 20, `pieceDistance` = 12. Connectors visible.



(f) `genMethod` = *corridor*, `maxPieces` = 176.



(g) `genMethod` = *arena*. One-pin connectors reserved for props and smaller obstacles.



(h) `genMethod` = *corridor*. One-pin connectors reserved for props and smaller obstacles.

Fig. 8: Several experimental levels. Only the more relevant generator parameters in each case are presented.

the final map layout, the different generation options on offer provide enough control for the designer to decide on the general characteristics of the generated level. Unlike several of the MICC-based approaches discussed in Section 2, our technique does not restrict map designs to a grid.

The case studies discussed in Sec. 4, in particular the *Trinity* videogame, demonstrated that the algorithm is able to achieve its intended goals, even though much of its potential is yet to be fully explored. Nonetheless, there is room for improvements. For example, another layer of connector combination constraints could allow the algorithm to combine not only full level pieces, but also props, obstacles and even simple cosmetic changes to those same pieces. Pursuing this idea further, if the algorithm made use of multiple generation passes, it could potentially create levels with several floors, adding verticality to the generated maps. Finally, a more sophisticated way to correct overlapping geometry would also be desirable.

In any case, we believe the proposed algorithm, in its current form, has the potential to be a useful tool for game developers and game designers to create quality levels in a fraction of the time it would take them with a fully human-based approach.

References

1. Adonaac, A.: Procedural dungeon generation algorithm. Gamasutra (Mar 2015), https://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php, last accessed on 23/07/2020
2. Apted, A.: Oblige level maker 7.70. Sourceforge (Oct 2017), http://oblige.sourceforge.net/, last accessed on 19/08/2020
3. Boris: Dungeon generation in Enter The Gungeon. Boris The Brave personal blog (Jul 2019), https://www.boristhebrave.com/2019/07/28/dungeon-generation-in-enter-the-gungeon/, last accessed on 29/08/2020
4. Dodge Roll: Enter the Gungeon. Devolver Digital (Apr 2016), https://www.dodgeroll.com/gungeon/
5. Fachada, N., Códices, N.: 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. pp. 300–306. ITiCSE '20, ACM, New York, NY, USA (Jun 2020). https://doi.org/10.1145/3341525.3387378, https://doi.org/10.1145/3341525.3387378
6. Himsl, F.: Binding of Isaac: Room generation explained! Gamesquid – YouTube (Apr 2020), https://www.youtube.com/watch?v=1-HIA6-LBJc, last accessed on 29/08/2020
7. Kazemi, D.: Spelunky generator lessons. Tiny Subversions (Oct 2013), http://tinysubversions.com/spelunkyGen, last accessed on 23/07/2020
8. Liapis, A., Smith, G., Shaker, N.: Mixed-initiative content creation. In: Procedural content generation in games, pp. 195–214. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-42716-4_11
9. McMillen, E.: The Binding of Isaac: Rebirth. Nicalis, Inc. (Nov 2014), https://www.nicalis.com/

10. Millington, I.: AI for Games. CRC Press, Boca Raton, FL, USA, third edn. (Mar 2019). https://doi.org/10.1201/9781351053303, https://doi.org/10.1201/9781351053303
11. Ølsted, P.T., Ma, B., Risi, S.: Interactive evolution of levels for a competitive multiplayer FPS. In: 2015 IEEE Congress on Evolutionary Computation (CEC). pp. 1527–1534. IEEE (May 2015). https://doi.org/10.1109/CEC.2015.7257069
12. Shaker, N., Togelius, J., Nelson, M.J.: Procedural content generation in games. Springer, Cham, Switzerland (2016). https://doi.org/10.1007/978-3-319-42716-4
13. e Silva, R.C., Feliciano, H., Fernandes, P.: Trinity. Okapi Games, GitHub (Jul 2020), https://github.com/RafaelCS-Aula/rockpaper_djd, last accessed on 23/07/2020
14. Smith, G., Whitehead, J., Mateas, M.: Tanagra: A mixed-initiative level design tool. In: Proceedings of the Fifth International Conference on the Foundations of Digital Games. pp. 209–216. FDG '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1822348.1822376
15. Trashboyjet: The Binding of Isaac: Rebirth (windows). The Cutting Room Wiki (Jul 2020), https://tcrf.net/The_Binding_of_Isaac:_Rebirth_(Windows), last accessed on 01/09/2020
16. Unity Technologies: Unity®(2020), https://unity.com/
17. Yu, D.: Spelunky Classic. Mossmouth, LLC (Dec 2009), https://www.spelunkyworld.com/original.html
18. Yu, D.: Spelunky: Boss Fight Books# 11. Boss Fight Books (Mar 2016)