Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Procedural Generation of Videogame Environments

Jan Kutálek

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
May 2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kutálek** Jméno: **Jan** Osobní číslo: **483778**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Procedurální generování prostředí pro videohry

Název bakalářské práce anglicky:

Procedural generation of videogame environments

Pokyny pro vypracování:

Zmapujte metody procedurálního generování 3D scén. Soustřeďte se na techniky vhodné pro videohry. Vytipujte metodu vhodnou pro generování rozsáhlých prostředí typu bludiště. Metodu následně implementujte v programu Houdini. Vytvořte nejméně čtyři procedurálně generovaná prostředí. Vyhodnoťte implementaci z hlediska výpočetní náročnosti v závislosti na parametrech metody a velikosti generovaného prostředí. Vytvořte jednoduchou aplikaci (hru), která využije generovaná prostředí v herním enginu. Vygenerovaná prostředí vyhodnoťte z hlediska jejich struktury a vizuální kvality v rámci uživatelské studie.

Seznam doporučené literatury:

[1] Noor Shaker, Julian Togelius, Mark J. Nelson. Procedural Content Generation in Games. Springer International Publishing. 2016.
[2] Hendrikx, Mark et al. Procedural Content Generation for Games: A Survey. In: ACM Trans. Multimedia Comput. Commun. Appl. 9.February, pp. 1–22. 2013.
[3] Marco Niemann. Constructive Generation Methods for Dungeons. Seminar report, Munster University. 2015.
[4] Seth Teller, Carlo Séquin, Visibility Preprocessing for Interactive Walkthroughs, in Computer Graphics (Proc. Siggraph '91), 25:61-69.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Jiří Bittner, Ph.D., Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **11.02.2021** Termín odevzdání bakalářské práce: **21.05.2021**

Platnost zadání bakalářské práce: **30.09.2022**

doc. Ing. Jiří Bittner, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. | Datum převzetí zadání

Podpis studenta

Acknowledgements

I would like to thank my colleagues for sharing their experiences and hardships while writing their theses. I would like to thank two of my best friends for giving me their time when I needed a distraction or words of encouragement. From the bottom of my heart, I would like to thank my family, especially my mom and sister, for their mental support. Most of all, I would like to express my gratitude to my supervisor, doc. Ing. Jiří Bittner, Ph.D., for giving me his full support in my pursuit of having a video game related topic for my thesis, as well as for pointing me in the right direction when I needed, for sharing his understandings and wisdom, and for having patience with me and encouraging to push forward.

Declaration

I hereby declare that this thesis represents my own work.

Abstract

There are many features in games that take a lot of time and resources to develop. That is why this thesis is focused on procedurally generated content, precisely dungeon environments, and generates video game levels using binary space partitioning algorithm in Houdini Engine.

Keywords: BSP, Binary Space Partitioning, Dungeon Generator, Procedural Generation, Video Games, Houdini

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Praha 2, Karlovo náměstí 13, E-421

Abstrakt

V herním průmyslu je spousta věcí, které vyžadují hodně času a prostředků. Proto se tato prace soustředí na procedurální generování obsahu do video her, přesněji na generování herních úrovních typu bludiště, na které je použit algoritmus binárního rozdělování prostoru a je implementován v Houdini Enginu.

Klíčová slova: BSP, binární rozdělování prostoru, generátor bludišť, procedurální generování, video hry, Houdini

Překlad názvu: Procedurální generování prostředí pro videohry

Contents

Figures

Tables

Chapter 1

Introduction

Procedural content generation in games refers to the *creation of game content automatically using algorithms* [4]. Many kinds of content can be generated: levels, items, characters, textures, quests, music, and others. Each can play a crucial role in different kinds of game genres and situations. Well generated non-player characters and narrative can be a different and unique experience for every player, which can be, for example, seen in the *Nemesis System* in a game *Middle-Earth: Shadow of Mordor*. However, procedural generation is often used for game objects such as vegetation, terrain, and various level generators.

1.1 Benefits of Procedurally Generated Content

There are many reasons to use procedurally generated content, and it can save a lot of development time and resources. It is not even needed to use the generated content and can be just used as an inspiration. Randomization can also help with replayability - games such as *Minecraft*, seen in figure 1.1, and *Terraria* throw each player into different generated worlds. This brings everyone a completely unique experience and may make the player play again after finishing the game because they will play in a different world.

Figure 1.1: Procedurally generated world in *Minecraft*. Source Minecraft Fandom [8].

Nevertheless, in games, it is not usually desired to have the content generated totally randomized. Therefore it is aimed for algorithms to have parameters that can further adjust the results. Such parameters can be anything from a size of a car wheel to a number of different-looking planks in a wooden bridge. The number of *parameters* changes the level of control that the user has, and it is usually good to have at least a couple of them, but at the same time, too many of them can bring more complexity in the creation of desired generated object, and it can also be desired for the parameters to change only a local scope of their working area.

Replayability and procedural content go hand in hand with one specific genre - *rogue-like*, named after a game called *Rogue*, which was made back in the early-1980s. Rogue introduced a perma-death concept - once the player dies, he loses everything and has to start all over again. This required a lot of content to keep the player engaged as replaying the completely same game again would become dull real quickly, but at the same time, creating that much content with *perma-death* as one of the genre-defining concepts by hand would be either impossible or excessively time-consuming. That is where procedural generation turned out to be the best solution for both the developer and the player. See figure 1.2 for an example of a generated level in Rogue.

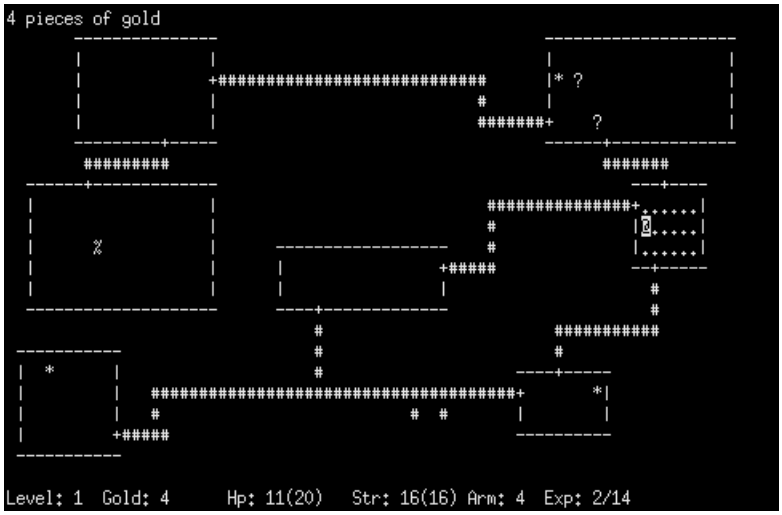


Figure 1.2: Procedurally generated map in *Rogue* (source [7]).

1.2 Dungeons

A dungeon in the real world is where prisoners were held captive, usually underground, and were present primarily in the medieval age associated with castles. However, a dungeon in games is referred to as a labyrinth area where the player enters at one point and dives in to explore various rooms, encounters enemies, and finds chests as a game reward. This idea probably originated from the famous tabletop game *Dungeons & Dragons*.

These dungeons are very popular in various games and game genres, and even *dungeon-crawler*, a subgenre of role-playing games, was created.

1.3 Games using Generated Content

Besides already mentioned games, there are many games that use procedural content. *Diablo* - a successful action role-playing game, uses generated dungeon and cave system, *Spelunky* - a popular indie 2D platformer, creates levels in a 2D world. *The Binding of Isaac*, yet another popular 2D indie game, but in a top-down view, is a rogue-like game that generates simple level layouts but changes the room's content. Furthermore, *Rimworld* is a sci-fi colony simulation game where in-game events are generated procedurally. Of course, there are many more, and it is impossible to mention them all.

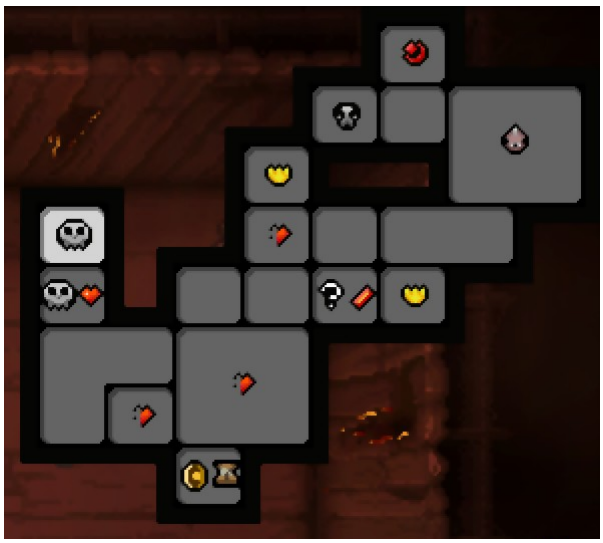


Figure 1.3: Procedurally generated map in *The Binding of Isaac*

1.4 Aim of this Thesis

The plan is to create a dungeon layout using one of the procedural algorithms - the Binary Space Partitioning, to change the way generated rooms are connected to achieve a more original layout with more user control, and finally, extruding the layout into 3D space. All of it will be implemented in Houdini, a modeling and animation software known for its procedural generation usage in the industry. This can provide both a useful set of tools and its own limitation and an extra challenge for the author to learn in new software.

Chapter 2

Procedural Generation of Dungeon Levels

There are many interesting ways of procedurally generating video game levels, and all of them have their advantages and disadvantages, as well as their use cases. This thesis picked just one of them to implement but will briefly mention others too. They are all talked about in more detail in *Procedural Content Generation in Games* [1].

2.1 Binary Space Partitioning

The thesis will solely focus on a *Binary Space Partitioning* (BSP) [14] algorithm, which is used to create random rooms for procedurally generated levels for games. BSP recursively divides space into two subsets. Each subset then can be represented in a *BSP tree* structure, where each node represents a subset before division and leaves represent the final space.

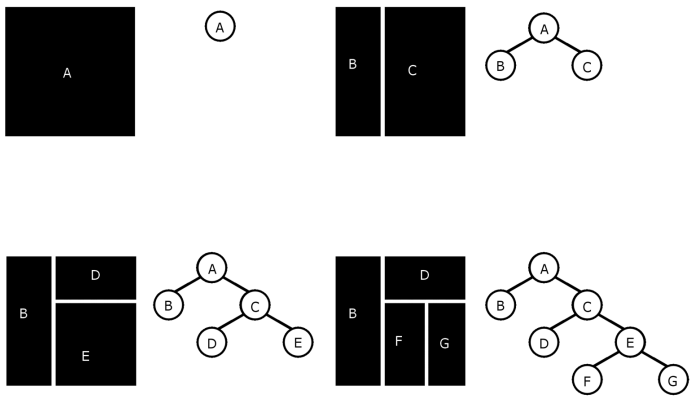


Figure 2.1: BSP Diagram

A straightforward way of making rooms out of the subsets is to take random 2 points in each cell, one top left and one down right, make a rectangle to represent a room. To then connect them, the BSP tree can be used to determine which room to connect by connecting every two leaf nodes with the same parent, making them a new subset area, and continuing the same

recursive algorithm by looking at the new areas as the new leave nodes as seen in figure 2.2. Connected corridors are then seen as part of the subset, and other corridors can be connected via them.

This approach guarantees that all rooms are connected but removes intuition and control over the look of the level; therefore, this method does not use the BSP tree to connect the rooms. Instead, this thesis uses the lines between rooms (white spaces in figure 2.1) to generate the main hallway and then connect the rooms all together.

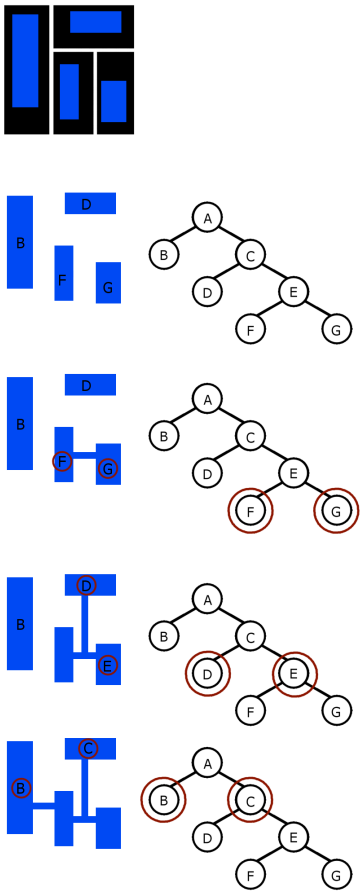


Figure 2.2: Levels from BSP

2.2 Agent-based Dungeon Growing

The agent-based approach uses a single agent that digs through a grid. It starts by digging a rectangle, then chooses a direction in which it continues to dig a corridor, and then it either continues digging the corridor or creating another room

The agent-based approach uses a single agent that digs through a grid. It starts by digging a rectangle, then chooses a direction in which it continues

to dig a corridor, and then it either continues digging the corridor or creating another room Agent-based algorithms usually create more chaotic layouts rather than systematic ones that the BSP does. The agent's AI can be either deterministic or stochastic, rooms can be allowed to intersect, and the agent can either have and use the knowledge of the already generated layout, or it can dig entirely blindly. These parameters are then used to produce different levels.

2.3 Cellular Automata

A cellular automaton uses a set of rules in an n-dimensional grid to change its states iteratively based on its neighbors. The grid is usually 1 or 2 dimensional. The most simple one is making a 2D layout with states either off or on. These states can be interpreted as space or a wall. The final look is dependent on the set of rules, number of iteration, and the state of the grid at the beginning but is usually used for cave layouts. However, a cellular automaton is not only limited to level generation; a popular example is *Game of Life* [10].

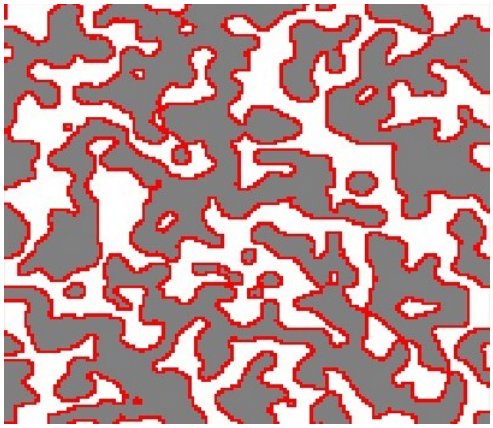


Figure 2.3: Cave generation using a Cellular automata algorithm (source [1]).

2.4 Other Level Generation Algorithms

Several other algorithms can be used for level creation, one of them being *Grammar-based dungeon generation*. An example of a grammar-based approach is Cite Engine, that creates cities of any complexity, see figure 2.4. It is also possible to use *Fractals* or *Noise* to generate a terrain environment. *Grammars*, more specifically *L-Systems*, can then be used to generate various vegetation for such areas or can be used to generate plants as a decoration for dungeon rooms.

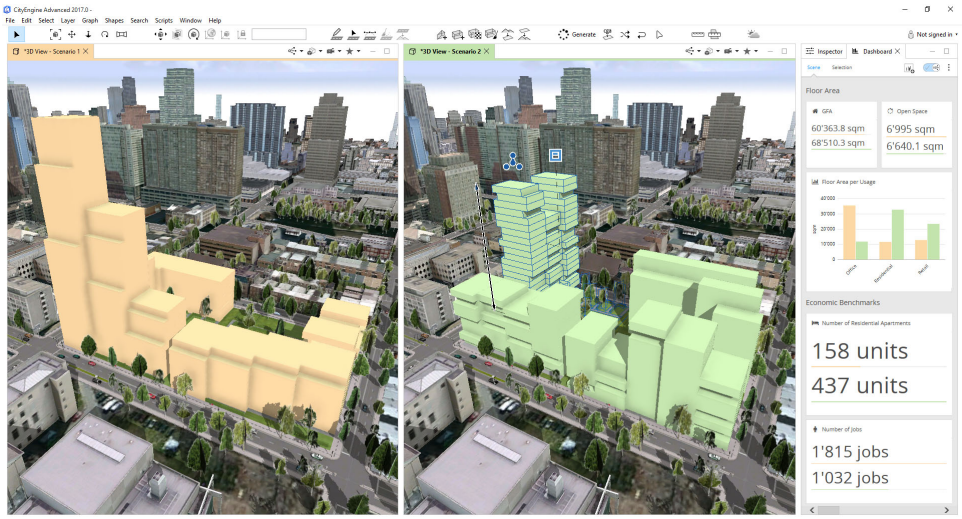


Figure 2.4: A city generation using City Engine. Source [9]

Chapter 3

Houdini

Houdini is a 3D animation software application developed by Toronto-based company SideFX. It is especially great for creating procedural content, which is why it has been chosen for this work.

3.1 Nodes

Houdini utilizes a node-based procedural workflow that makes it easy to work with. Each node represents an operation, and it is possible to traverse through each node to see its result, as well as it is possible to connect one node with several others as means of reusing older parts' results.

In figure 3.1, there are three nodes. The first *Grid* node creates a grid. The other two are Transform nodes that transform the grid. The former node scales the grid, and the latter translates it. In addition, the translating node is marked blue, which displays it, and the *Grid* node is marked pink, which displays it as a reference in a wireframe mode. The final view is in figure 3.2, reference grid on the bottom, the current scaled and translated grid on the top.

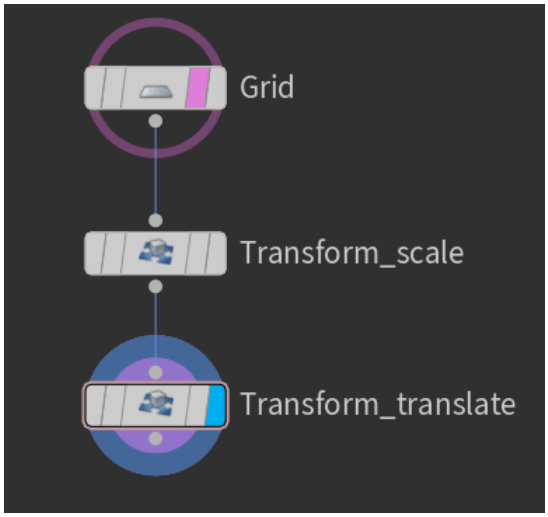


Figure 3.1: An example of 3 nodes wired together.

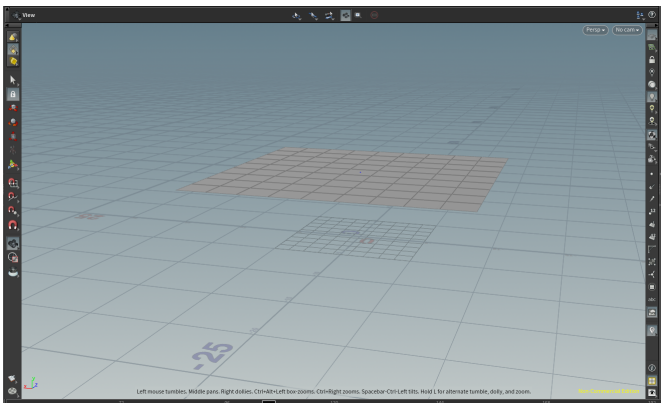


Figure 3.2: A visualized example of 3 nodes wired together in the viewport.

3.2 Geometry

Geometry in Houdini is represented through 3 main components - *Points*, *Vertices* and *Primitives*.

- A *point* is simply a point in space as defined by four numbers (X, Y, Z, W).
- A *vertex* is a reference to a point.
- *Primitives* use vertices to reference points. For example, the corners of a polygon, the center of a sphere, or a control vertex of a spline curve. Primitives can share points, while vertices are unique to a primitive [6].

3.3 Metadata

To work with the geometry procedurally, there is a need for various data to work with during the whole generating process. In Houdini, such data is called *attributes* and can be stored into the geometry - into the *points*, *vertices*, and *attributes*. In addition, it is also possible to store *attributes* into *Detail*, which is a storage area shared for the whole geometry.

There are several attribute types to work with. The most used ones are *int*, *float*, *string*, and *vector* of varying dimensions of 2, 3, and 4. In addition, it is also possible to work with arrays that are dynamic.

Examples of attributes can be *point* or *primitives' ID*, vertex position *P*, and of course, any user-defined such as *Area* that can be calculated using the *Measure node*.

3.4 Notable nodes

There are dozens and dozens of nodes in Houdini; luckily, it is not needed to know all of them. In this project, there is a total number of 26 nodes used. This section will briefly introduce them; they can all be found in figure 3.3.

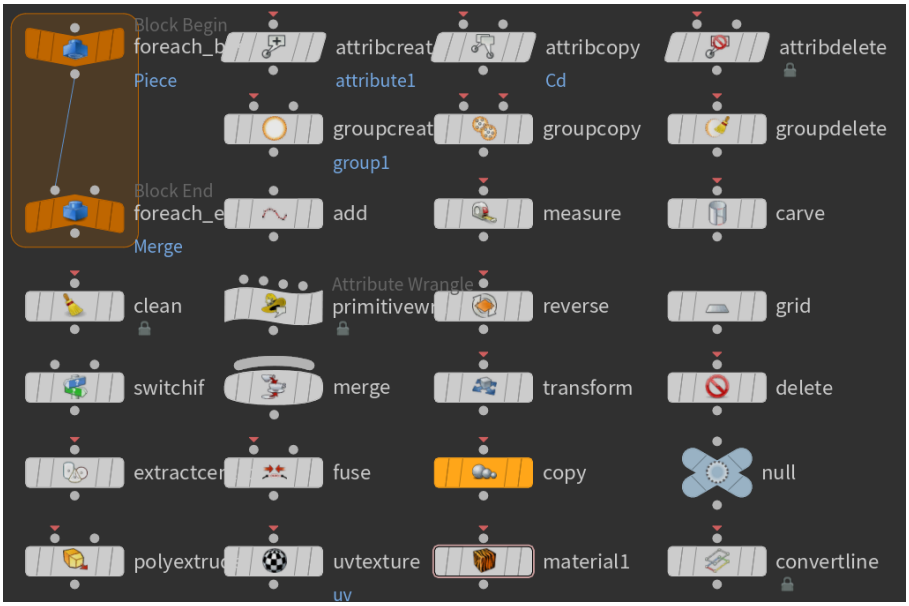


Figure 3.3: All nodes used in this thesis.

- For-Each

  For-Each is a node that allows us to iterate. It is composed of two subnodes - *For-Each_Begin* and *For-Each_End*. In addition, it is also possible to create a *For-Each_Metadata* subnode that stores the current iteration number. It is possible to iterate over few options: points, primitives, and by count. Moreover, there is a gathering method to be set - *Feedback Each Iteration* that outputs the last iteration of the loop.

- Attribute Create/Copy/Delete

  Creates/Copies/Deletes an attribute from geometry, the type of attributes can be selected. *Attribute Copy* copies attributes from one geometry to another.

- Group Create/Copy/Delete

  Analogical to the attribute variation, but instead groups the selected geometry.

- Add

  Adds *Points* or creates *Polygons* from selected points.

- Measure

  Measures the *area*, the *perimeter*, the *centroid* of the geometry, and few others too that were not used in this work.

- Carve

  Carve slices, cuts or extracts points or cross-sections from a primitive.

- Clean

  Clean node consolidates points, removes duplicates, removes unused points, or can reverse winding.

- Primitive Wrangle

  Runs a VEX snippet that can modify the geometry, add or change the attributes. It is iterated over primitives, but can the iteration can be changed to points, vertices, or even whole geometry, in which case it can iterate a defined number of times. The node uses a VEX language, which is very similar yet slightly different from the C language.

- Reverse

  Reverses the order of vertices in primitives.

- Grid

  Creates a grid geometry.

- Switch-If

  Has two inputs and passes one depending on a condition. There can be multiple conditions set and test various expressions, attributes value, or element count. If one of the inputs is not connected, empty geometry can pass through.

- Merge

  Merges any number of geometry inputs into a single geometry.

- Transform

  Transforms the geometry using translation, rotation scale, or shear. The pivot can be changed.

- Delete

  Deletes the specified geometry.

- Extract Centroid

  Extracts the centroid of the geometry.

- Fuse

  Fuses points in the geometry based on its distance.

- Copy and Transform

  Duplicates the geometry and transforms in the same fashion as the *Transform* node.

- Null

  Null node does nothing essentially but can be used as a place holder. It is possible to change the parameter interface of every node, including the null node. Therefore it can be used as a node that holds all the parameters that change the procedural generation.

- PolyExtrude

  Extrudes the geometry, either in the direction of primitive normals or point normals. Several outputs can be selected, the extruded sides, front, and bottom primitives.

- UV Texture

  Assigns uv texture coordinates to the geometry.

- Material

  Assigns material to the geometry.

- Convert Line

  Converts the geometry from faces to lines.

Chapter 4

BSP Implementation

This chapter takes the introduced BSP algorithm shown in figure 2.1 and explains the implementation in Houdini. It creates a square face and divides it into several subparts using an iterative version of BSP. The following sections discuss the *Set-Up*, the *Main Loop* with the partitioning process, and, finally creating room areas out of them.

4.1 Setip

As BSP operates on a rectangle, a square is used as the starting geometry. Moreover, a placeholder for parameters is needed. To create a square in Houdini, a *Grid* node is used with rows and columns set to 2. After that, a *Null* node is created separately; parameters will be added here when needed throughout the rest of the project.

4.2 Main loop

The core of the BSP algorithm is its recursive partitioning. As the author did not find a way to recursively connect nodes in Houdini, the main loop was adjusted into an iterative version using two for loops. The inner loop iterates over each primitive and divides it accordingly. The outer for each loop iterates n number of times where n is a large number.

As this loop would run forever, a stop condition is set. That is why an attribute *leaveLoop* is needed beforehand that will be read from. In the inner loop, then the division algorithm will be held, but to add more control over the result, a new parameter *minArea* is set. The division will be held only if the primitive's area is at least *minArea* large.

If all primitives' areas are less than *minArea* then the loop must end. That is why leaveLoop is set to `true` in the outer loop, and if any primitive is divided, it is set to `false`. The outer loop then ends if all primitives are

smaller than *minArea*.

Algorithm 1: Main Loop

```
1 bool leaveLoop = false;
2 for ( int i = 0; i < n and !leaveLoop; i++ ) {
3 |   leaveLoop = true;
4 |   foreach Face ∈ Faces do
5 |   |   if Face > minArea then
6 |   |   |   leaveLoop = false;
  |   |   |   // divide Face
```

This algorithm is replicated by first creating an *Attribute Create* node and renaming the attribute to *leaveLoop*, integer is used as there are no booleans in Houdini. Then making a *For-Each*, setting it to iterate *By Count* with a large enough number of iteration, 99999999 was used in the implementation. *Primitive Wrangle* node follows with a single line of code `setdetailattrib(0, "leaveLoop", 1);`. This will set the *leaveLoop* to 1. It continues with another *For-Each* node, this time set to iterate over primitives.

Now to check the primitive's area, a *Measure* node is added for the calculation. Then an *If-Switch* node is used to check the area condition. This node is then duplicated to create another branch, but this time with a negated condition.

One branch is left blank, as its only purpose is to not lose the primitive that is already done, but the splitting branch has one more *Primitive Wrangle* node that sets the *leaveLoop* to 0. Finally, both branches are merged together with a *Merge* node. The node system, together with the first set-up, can be seen in figure 4.1.

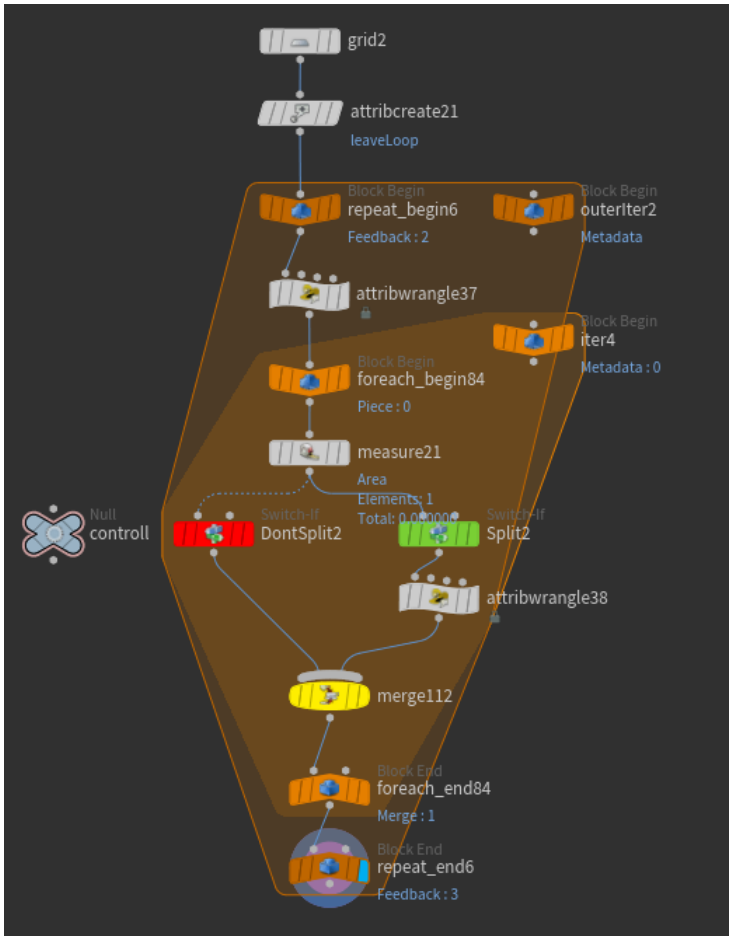


Figure 4.1: The main BSP Loop.

4.3 Face Division

With the pseudo-recursion working, it is division time. The division part can be broken down to separating edges, taking a parallel pair and cutting each edge from one pair. This creates two new pairs for the new primitives. As the vertices are actually the most important, there is no need to create the middle edge, and it is enough to take all the vertices, order them correctly and create new faces.

Furthermore, it is important to decide the cut direction. At first thought, it would be desired to be completely randomly chosen, but it will create layouts not good in terms of level design - some of the faces will be too long and narrow. That is why a condition check is added. If the shape is similar to a square, choose randomly; if not, cut the wider edge. The shape check is done through the edges' ratio. The minimum required ratio will also serve as a new parameter *Ratio*.

With the direction decided, cutting it remains. However, it may not be desired to slice the face too close to its vertices, as this would create too long

and narrow divisions again. Thus, 2 new parameters are added - *minCut* and *maxCut*, where: $0 < minCut < maxCut < 1$. These parameters then define the edge ratio cut limit and the cut position is randomly selected in the range $(minCut, maxCut)$.

Finally, to get different results, a seed will be defined as a parameter called *BSPSeed*. This seed is used in the random direction decision as well as the cut ratio. To avoid getting same outcomes, this seed is multiplied by the primitive's ID.

Algorithm 2: BSP Algorithm

```
1  bool leaveLoop = false;
2  for ( int i = 0; i < n and !leaveLoop; i = i++ ) {
3      leaveLoop = true;
4      foreach Face ∈ Faces do
5          if Face > minArea then
6              leaveLoop = false;
7              verticalLines[] = GetVerticalLines(Face);
8              horizontalLines[] = GetHorizonalLines(Face);
9              Line linesToCut[];
10             Line otherLines[];
11             float currentRatio = getRatio(verticalLines,
                horizontalLines);
12             if currentRatio > Ratio then
13                 if random(BSPSeed * Face.id) > 0.5 then
14                     linesToCut = verticalLines;
15                     otherLines = horizontalLines;
16                 else
17                     linesToCut = horizontalLines;
18                     otherLines = verticalLines;
19             else
20                 linesToCut = maxLen(horizontalLines, verticalLines);
                   // gets longer lines
21                 otherLines = minLen(horizontalLines, verticalLines);
                   // gets shorter lines
22             Line newLines[] = cutLines(linesToCut);
23             createNewFaces(newLines, otherLines);
24             delete(Face);
```

Transferring the algorithm into a node graph requires a bit more work and can look a bit confusing. Therefore the nodes will be colored at the end for better visualization. To start with edges separation, the primitive needs to be separated into lines. A carve node is used for this, following a set of *Delete* nodes as seen in figure 4.2. The two branches are symmetrical, just inverted the first selection. Starting with the first *Delete* node, it is set to delete by range and to delete every other primitive. Because the primitives are guaranteed to be order clock-wise, opposite lines are deleted. The node then branches again with two more *Delete* nodes. This time, one deletes the

0th primitive, and the other deletes the *1st* primitive. Each *Delete* node now holds one edge, which is equivalent to lines 6 and 7 in the BSP Algorithm [2].

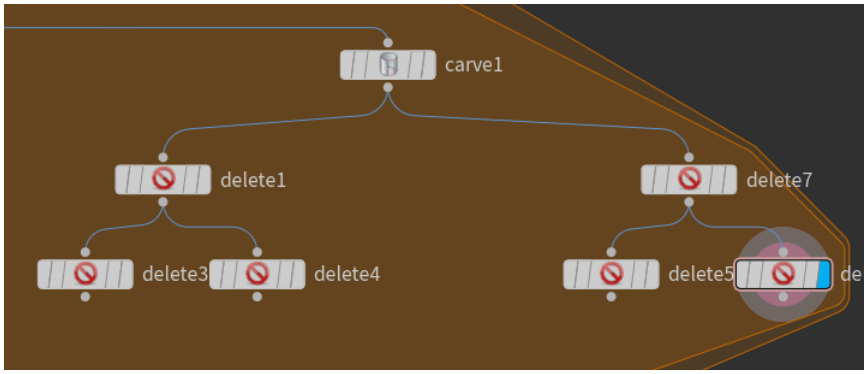


Figure 4.2: A node graph in Houdini for separating edges of a primitive

Following line 10, to get the current ratio, the length of the edges is calculated using two *Measure* nodes. The lengths are stored in attributes called *Width* and *Height*. As for the first if statement, several *If-Switch* nodes are used, four for each edge and four again for its negation. The condition looks as follows: `min(Width, Height) / (max(Width, Height)) > Ratio`. As it is possible to reference the test, only one *If-Switch* node has the condition, and others reference it. This is the first row with green nodes in figure 4.3. The following row is a set of two of the exact same node graph system, but this time with a different condition. The condition this time implements line number 12 in the BSP Algorithm. *BSPSeed* is referenced from the control node and Face.ID is referenced from the *For-Each* node's *Metadata* node, where the iteration number is stored.

The assigning part is done with *Merge* nodes, perpendicular pairs of edges are merged, which substitutes the lines 13, 14, 16, 17, 19, and 20. The four final merged nodes then represent two pairs of edges; the first two are *linesToCut*, the latter *otherLines*.

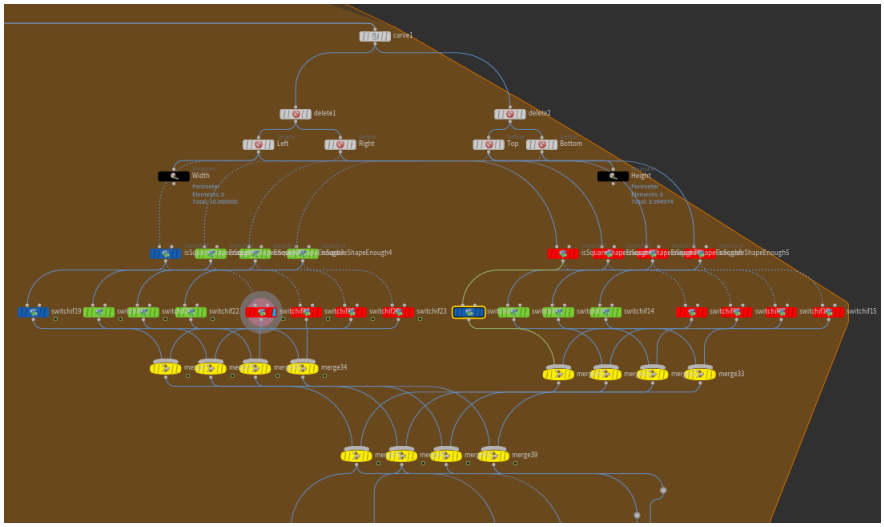


Figure 4.3: A node graph in Houdini of an *If-Switch* system for BSP algorithm

Each node from the first pair is connected into two *Carve* nodes. The first *Carve* node has its *First U* value set to a random value n between $minCut$ and $maxCut$, which are new parameters added to the control nodes and where $0 < minCut < maxCut < 1$. This shortens the line by the number n. The other connected *Carve* node has set the *Second U* to a value $1 - n$, as this shortens the edge in the opposite direction. The two new lines are visualized with a different color in figure 4.4

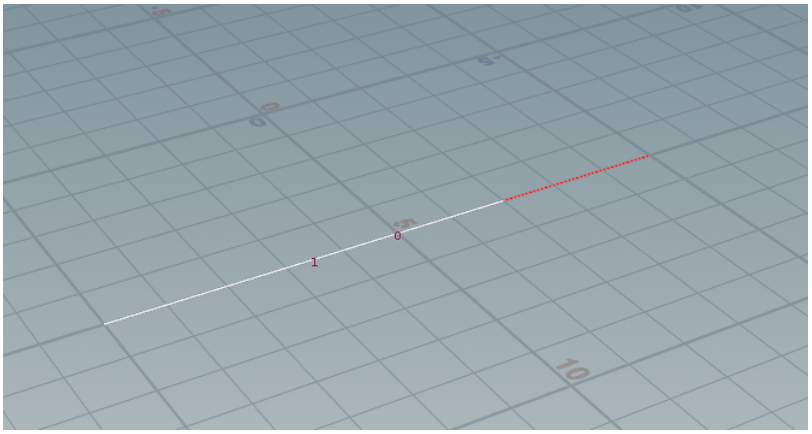


Figure 4.4: Cut primitive's line for BSP.

Finally, with the new edges created, they are merged together with the other previous lines to form two new faces. This is done with a combination of a *Merge* node, inputting respective lines anti-clockwise, using a *Clean* node to remove duplicate points, and creating the making a face with an *Add* node. As this is required for both of the two new faces, they are also merged together, cleaned of the duplicate points, and wired back into the for each loop. A test example result can be seen in figure 4.5 and the whole graph in Houdini in figure 4.6.

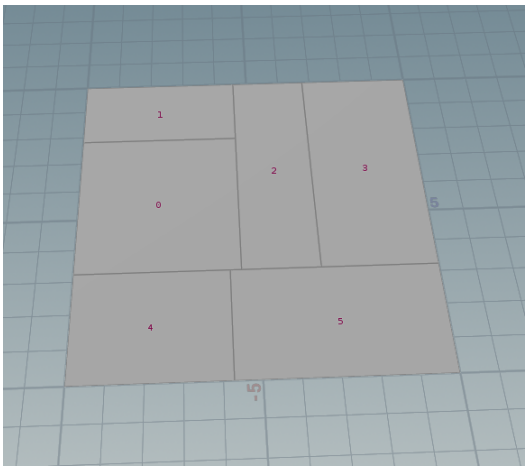


Figure 4.5: Example of BSP result.

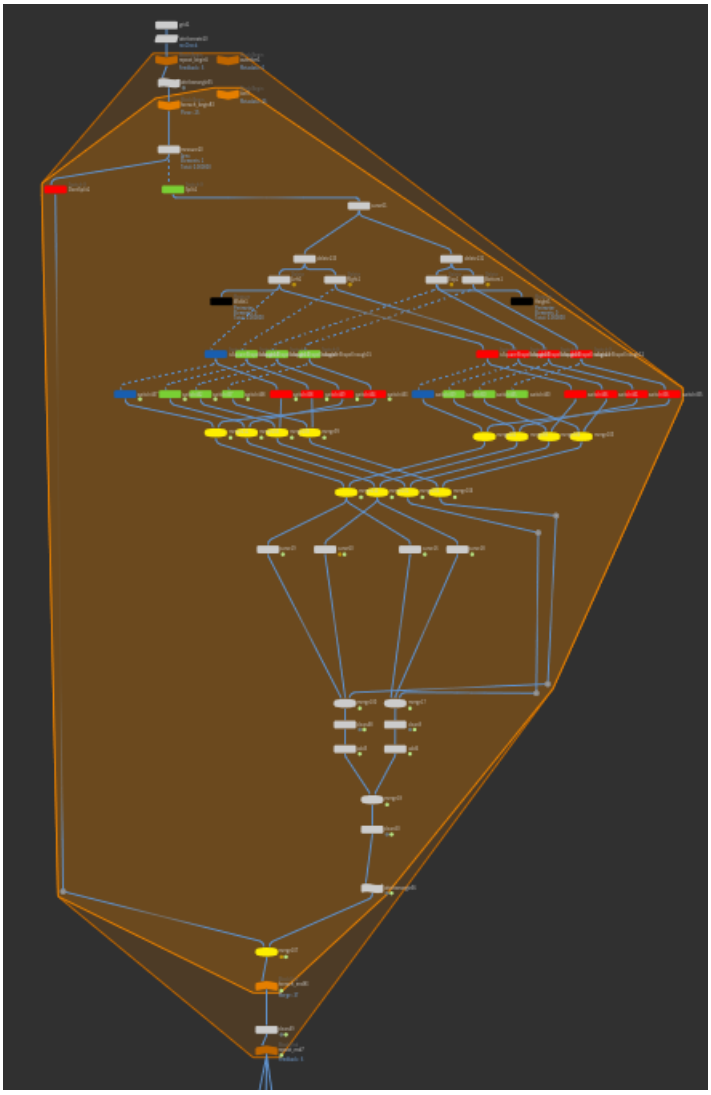


Figure 4.6: A whole node graph of a BSP algorithm in Houdini.

4.4 Rooms

Rooms from the divided space are created by scaling down each room. The scale defines the width of the wall, but since this is something that can be helpful to have as a parameter, it is introduced as such as *WallWidth*. However, because scaling is a done with percentages, a formula to calculate the percentage is needed. Because edges of all faces are parallel with x and z-axis, scaling in each direction will be done. Since $x \cdot length = newLentht$, where x is wanted scale factor, and $newLength = length - 2 \cdot wallWidth$, because a wall is on both sides, then $x = \frac{length - 2 \cdot wallWidth}{length}$. The sides are to be scaled down using this equation with their respective line lengths.

Houdini implementation is pretty straightforward. It takes the divided space from the partitioning process and runs a for-each loop with the *For-Each* node. In this, it branches out into two directions. One is for calculating only, and the other uses a *Transform* node for scaling.

The first branch uses an *Extract Centroid* node to get the centroid for each face, as it would transform the face in respect to the zero vector otherwise, and then continues with a *Carve* node to get edges of the primitive, and then two series of double *Delete* nodes to separate a single edge in both directions, the edge isolation is analogical to the edge separation process in the previous section. Both perpendicular edges are then wired to a *Primitive Wrangle* node that is set to iterate just once.As this node can be wired with multiple nodes, it is used to calculate the length of both edges in a single node and correctly stored since it is also required to know what direction length is. For this *xLen* and *zLen* attributes are used, and a condition checks whether the edge's points have equal x or z coordinate. All can be seen in pseudocode 3.

Algorithm 3: Calculating primitive's dimension

```
1  float ϵ = 0.001;
2  float xLen;
3  float yLen;
4  vector edgeVertices1[] = getVertices(input[0]);
5  vector edgeVertices2[] = getVertices(input[1]);
6  if abs(edgeVertices1[0].x - edgeVertices1[1].x) < ϵ then
7  |   xLen = distance(edgeVertices2);
8  |   yLen = distance(edgeVertices1);
9  else
10 |   xLen = distance(edgeVertices1);
11 |   yLen = distance(edgeVertices2);
```

The other branch uses the *Transform* node to scale down the primitive using the formulas $x = \frac{xLen - 2 \cdot wallWidth}{xLen}$ and $z = \frac{zLen - 2 \cdot wallWidth}{zLen}$ respectively, and translating the pivot to the primitive's centroid. See the figure 4.7 for the cmoplete graph in Houdini.

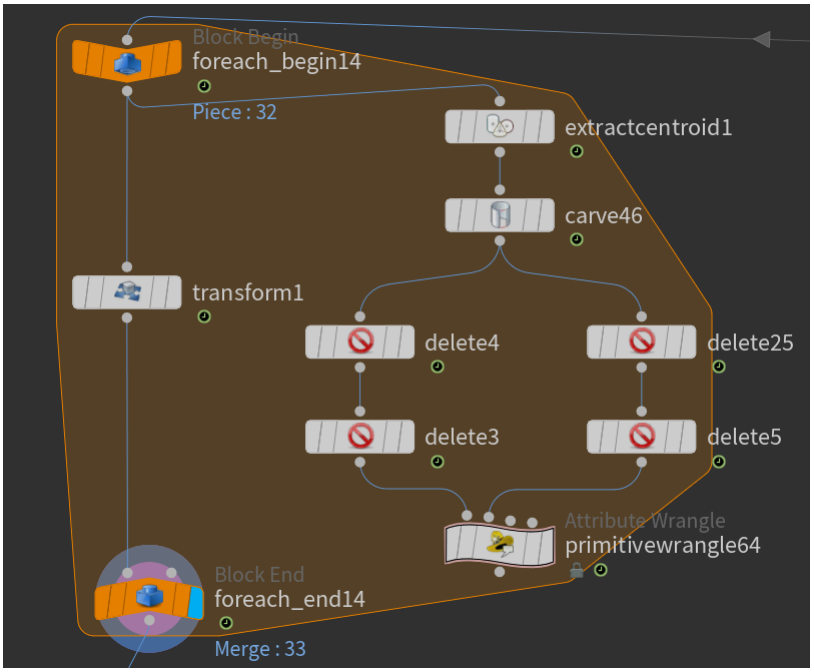


Figure 4.7: Node graph of rooms transform

Chapter 5

Creating Corridors

This chapter focuses on connecting all the rooms together. As discussed in chapter 2, there are many ways to connect rooms, a main hallway is used for this project. To create a main hallway, the edges of the divided space are used as a guideline and attached to the rooms. Attaching the hallway to the rooms requires calculating neighboring the neighboring first. Once it is obtained, the rooms are connected to the main hallway.

5.1 Lines and Neighbours

Getting the edges is very simple in Houdini, and a *Convert Line* node does all the work. The graph from figure 2 is wired into it. However, if an edge has three or more faces neighboring, then the edge is actually made of many sub-edges after converting to lines. In example 5.1, the faces A, B, and C all have one common edge. However, each face has its own individual edge - A has the red one, B has the green one, and C has the blue one. Therefore they will be unified. However, first, the vertices of the edges will be used to obtain the relationships between edges and faces.

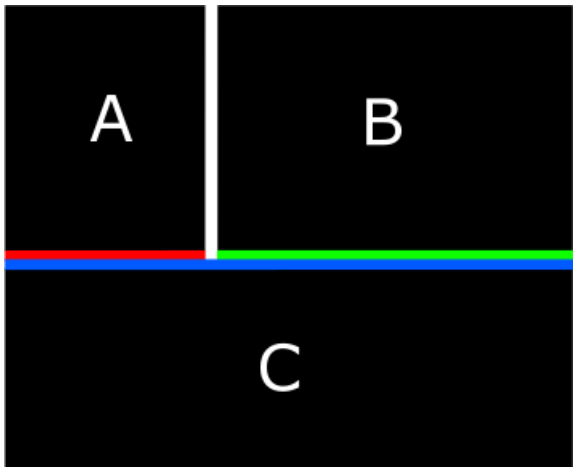


Figure 5.1: Node graph of rooms transform

5.1.1 Neighbors

To get each edge its neighboring faces, all vertices from the edge are checked with all faces. As there will never be an edge with more than two faces, it is used as an optimization.

Algorithm 4: Finding neighbouring faces for edges

```
1  foreach edge ∈ edges do
2      int commonVertices = 0;
3      foreach face ∈ faces do
4          foreach edgeVertex ∈ edge.vertices do
5              foreach faceVertex ∈ Edge.Vertices do
6                  if edgeVertex == faceVertex then
7                      commonVertices++;
8                      break;
9          if commonVertices == 2 then
10             edge.faces.push(face);
11     if edge.faces.size() == 2 then
12         break;
```

In Houdini algorithm 5 is written in VEX code using the *Primitive Wrangle* node, wired in both, the edges and the faces, and the faces are saved are then saved in the edges geometry using new primitive attribute *faces*.

5.1.2 Deleting Borders

Having the neighbours set, the edges are to be merged. However, few lines will be deleted as this thesis takes having a hallway around the outer rooms as undesirable. Therefore these lines will be deleted by checking their vertices coordinates.

The outer edges are deleted if they have their vertices on the original primitive square, which is checked by comparing all vertices' x or z components to the half of the original square's width. If both coordinates are, the vertex is deleted, as it is a corner; if only one coordinate is, it is marked as a border and later used for the edge deletion as well as for shortening the corridors in the following section.

Algorithms 5 and 6 are implemented using two *Primitive Wrangle* points, the former iterating over points and the latter iterating over primitives. The border property is stored in a point attribute.

Algorithm 5: Finding border points and deleting corners

```
1 float ϵ = 0.001;
2 foreach vertex ∈ vertices do
3     if abs(vertex.x ± width / 2) < ϵ and abs(vertex.z ± width / 2) <
       ϵ then
4         removeVertex(vertex);
5     else if abs(vertex.x ± width / 2) < ϵ or abs(vertex.z ± width /
       2) < ϵ then
6         vertex.border = true;
```

Algorithm 6: Deleting border edges

```
1 foreach edge ∈ edges do
2     bool delete = true;
3     foreach vertex ∈ edges.vertices do
4         if vertex.border == false then
5             delete = false;
6             break;
7     if delete == true then
8         deleteEdge(edge);
```

5.1.3 Lines Merging

For avoiding duplicates and optimizing extrusion later, all lines that are parallel and intersecting will be merged. The exception are edges that only share a single point, these lines are only touching, and if there were to be a cross - four different lines would share the single point, it would create a cross that in the future extrusion would create a walled-off cross. However, in it is guaranteed that all other edges merge because unless a point has four common faces, it is either a final edge or intersects with a different edge. Furthermore, this merging will be done recursively using a pseudo-recursive iterative version, with the same principles as in the BSP algorithm.

The implementation in Houdini is done with an *Attribute Create* node to create the *leaveLoop* check, following a *For-Each* node that serves as an infinite for loop with the *leaveLoop* set as a stop condition, and finally a *Primitive Wrangle* node for the code.

Algorithm 7: Merging lines

```
1  bool leaveLoop = false;
2  for ( int i = 0; i < n and !leaveLoop; i++ ) {
3      leaveLoop = true;
4      sort(edges);
       // sort by length
5      for ( int e = 0; e < edges.size(); e++ ) {
6          for ( int otherE = e; otherE < edges.size(); otherE++ ) {
7              if areIntersect(e, otherE) and areParallel(e, otherE) then
8                  mergeFaceAttribute(e, otherE);
9                  mergeEdges(e, otherE);
10                 remove(edges, otherE);
11                 otherE–;
12                 leaveLoop = false;
```

5.1.4 Floor of The Hallway

The floor is created by duplicating the lines and translating them in both perpendicular directions in the xz plane. As this requires information of line direction, which will be needed later too, it is added as a new attribute *orientation*, equal to either x or z. The width of the corridors is a new parameter *CorridorWidth*. Since the neighboring corridors would overlap because they are tangents to each other, each corridor is squished in the opposite direction by scaling down that way. This, however, creates an empty space when there is a corner of four faces, see figure 5.2. For this reason, for each point that is part of exactly four faces, a square patch is created in the middle.

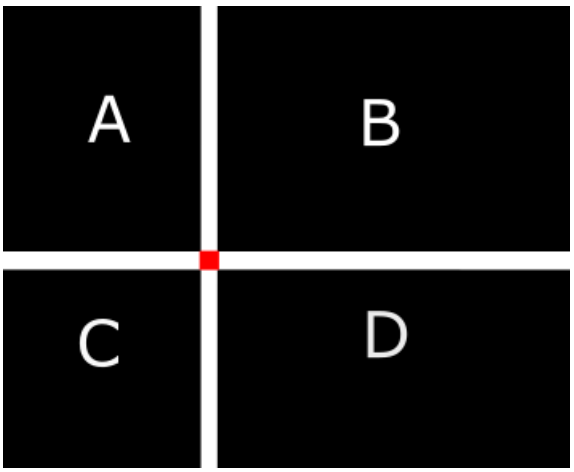


Figure 5.2: A red missing patch on a corner of four faces

Algorithm 8: Creating main hallway floors

```
1  float ϵ = 0.001 foreach edge ∈ edges do
2      Edge duplicatedEdge = EdgeCopy(edge);
3      string orientation;
4      if abs(edge.vertices[0].x - edge.vertices[1].x) < ϵ then
5          edge.vertices[0].x += CorridorWidth / 2;
6          edge.vertices[1].x += CorridorWidth / 2;
7          duplicatedEdge.vertices[0].x -= CorridorWidth / 2;
8          duplicatedEdge.vertices[1].x -= CorridorWidth / 2;
9      else
10         edge.vertices[0].z += CorridorWidth / 2;
11         edge.vertices[1].z += CorridorWidth / 2;
12         duplicatedEdge.vertices[0].z -= CorridorWidth / 2;
           duplicatedEdge.vertices[1].z -= CorridorWidth / 2;
13     vector faceVertices[];
14     push(faceVertices, edge);
15     push(faceVertices, duplicateEdge);
16     Face face = addFace(faceVertices);
```

This is implemented using a *Primitive Wrangle* node; however, there is one problem with the new primitive - the vertices are not guaranteed to be ordered right. That is why a new algorithm for ordering the vertices is introduced. If the face is convex, any point in the middle of the face can be taken as a pivot. Then any vertex is taken together with the pivot, and they make a pivot line. After that, take each of the other vertices, connect it with the pivot and calculate an angle between this line and the pivot line. See figure 5.3; there are points A, B, C, and D, a pivot P, and a red line as the pivot line. If α is the angle between the pivot line and the B-P line, β is the angle between the pivot line and the C-P line, then $\alpha < \beta$, analogically for all the other lines. Descending sort by angle provides anti-clockwise points.

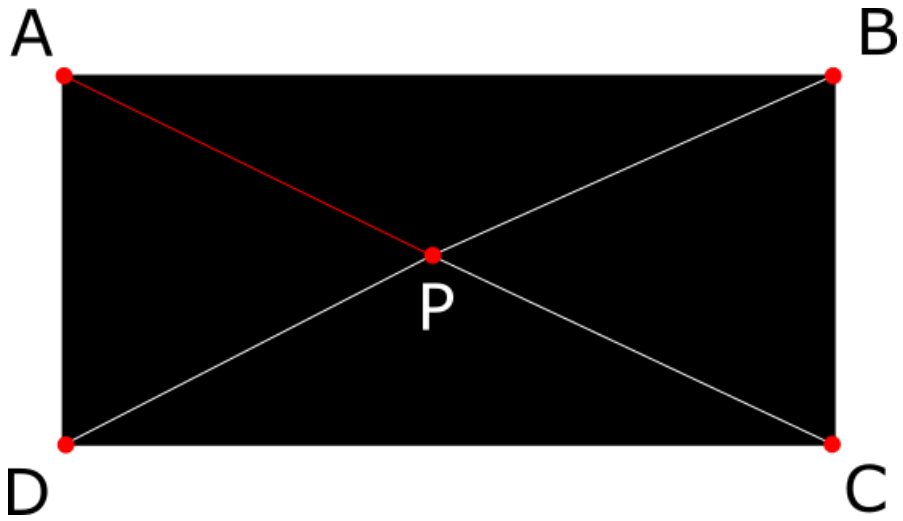


Figure 5.3: A rectangle with a pivot point P.

Algorithm 9: Ordering primitive's vertices

```
1  vector pivot = {0, 0, 0};
2  foreach vertex ∈ vertices do
3  |  pivot += vertex;
4  pivot /= vertices.size();
5  vector firstVertex = vertices[0];
6  firstVertex.angle = 0;
7  for ( int i = 1; i < vertices.size(); i++ ) {
8  |  vector v = vertices[i];
9  |  v.angle = angleBetween(firstVertex - pivot, v - pivot);
10 sort(vertices);
```

Taking all the corridor floors now, it is needed to shorten them in their respective directions. This is a slightly different variation of the graph in figure 4.7. It is expanded in the left branch by 2 *Switch-If* nodes, each having its own *Transform* node, and then merging together with a *Merge* node, see figure 5.4. The *Switch-If* nodes check the *orientation*. The *Transform* nodes scale down the corridor areas in their respective directions.

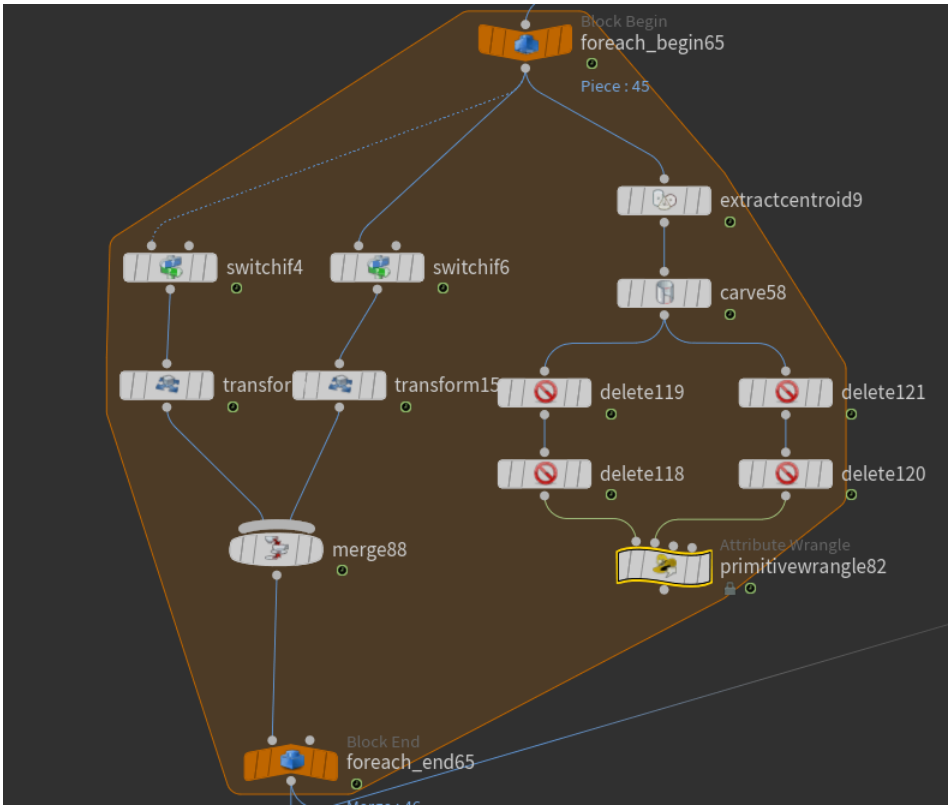


Figure 5.4: A node graph of shortening the hallway faces.

Finally, square patches are implemented by taking all points that belong to four primitives, making three duplicates of this point, and transforming

them to corners of a square. Because the patches are implemented separately in a different branch, all other points are deleted. A *Primitive Wrangle* node is used with the following algorithm 10.

Algorithm 10: Square patch creation

```
1  foreach vertex ∈ vertices do
2      if vertex.faces.count() != 4 then
3          deleteVertex(vertex);
4      else
5          vector patchVertices[];
6          vector vertex1 = vertex2 = vertex3 = vertex;
7          vertex.x += CorridorWidth / 2.0;
8          vertex.z += CorridorWidth / 2.0;
9          vertex1.x -= CorridorWidth / 2.0;
10         vertex1.z += CorridorWidth / 2.0;
11         vertex2.z -= CorridorWidth / 2.0;
12         vertex2.x -= CorridorWidth / 2.0;
13         vertex3.x += CorridorWidth / 2.0;
14         vertex3.z -= CorridorWidth / 2.0;
15         addFace(vertex, vertex1, vertex2, vertex3);
```

5.2 Connecting Rooms

Connecting rooms requires a few steps. First, for each hallway face, get its neighbor rooms, create connection corridors, add points to the corridors in places where the corridors connect, and finally, merge them all together. Adding points to the hallways is necessary for extrusion later; otherwise, there would be no holes in the geometry for the entrance.

5.2.1 Making Connections

To make the connection corridors, the algorithm goes as follows. For each corridor face, take all its rooms and create four new vertices for the connecting corridor. Then depending on the *orientation*, the x and z coordinates are assigned, but because it is completely analogical, only the x *orientation* is discussed. Find a corridor's edge that is close to the room, and a room's edge that is close to the corridor, as the new vertices will lie on them. Assign the z coordinate of the corridor's edge to the first two new vertices, and the z coordinate of the room's edge to the latter two coordinates. Because the connections will be perpendicular to the edges, find two x coordinates in the middle of from the corridor's edge and the room's edge vertices. This guarantees that choosing a value in between the two middle x coordinates

does not make a connection out of bounds. Then generate a random number in range

To make the connection corridors, the algorithm goes as follows. For each corridor face, take all its rooms and create four new vertices for the connecting corridor. Then depending on the *orientation*, the x and z coordinates are assigned, but because it is completely analogical, only the x orientation is discussed. Find a corridor's edge close to the room, and a room's edge that is close to the corridor, as the new vertices will lie on them. Assign the z coordinate of the corridor's edge to the first two new vertices and the z coordinate of the room's edge to the latter two coordinates. Because the connections will be perpendicular to the edges, find two x coordinates in the middle of the corridor's edge and the room's edge vertices. This guarantees that choosing a value in between the two middle x coordinates does not make a connection out of bounds. Then generate a random number in range $< minX + \frac{corridorWidth}{2}, \ maxX - \frac{corridorWidth}{2} >$ to get the x coordinate of the center of the new corridor and call it $randomX$. Add $randomX \pm \frac{corridorWidth}{2}$ respectively to the x coordinate of each of the new vertices to create a rectangle. Finally, order the vertices anti-clockwise and create a new primitive. Moreover, vertices that lie on the corridor edges are added to an array to add them to corridors' primitives in the following subsection; vertices that lie on the room edges are added now and sorted again too. The connections are visualized in figure 5.5 as red rectangles.

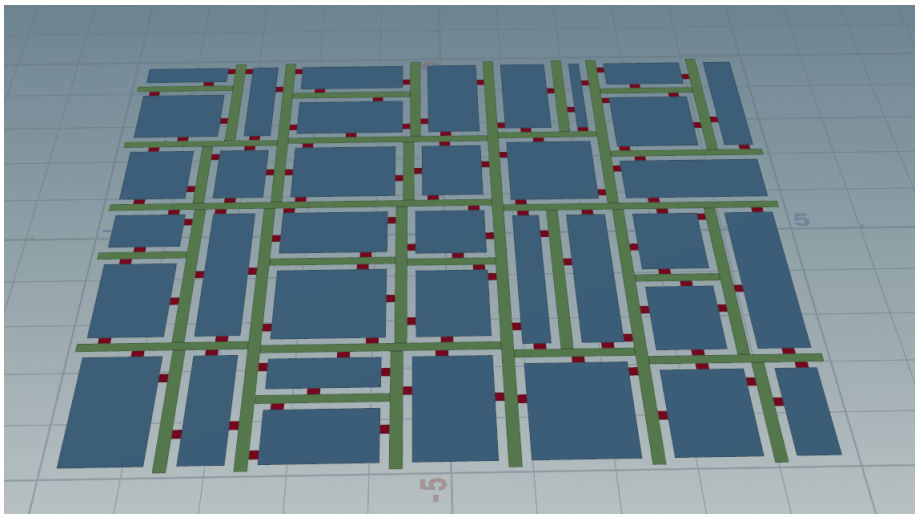


Figure 5.5: An example of generated hallway connected to rooms. Blue rectangles are the rooms, green ones are the hallway areas and the red rectangles are the connections.

Algorithm 11: Making Connections

```
1  foreach corridorFace ∈ corridorFaces do
2      vector verticesToAddToCorridor[];
3      foreach room ∈ corridorFace.rooms do
4          vector vertices[];
5          for ( int i = 0; i < 4; i++ ) {
6              push(vertices, {0,0,0});
7          if corridorFace.orientation == "x" then
8              edge corridorEdge =
                 getXEdgeFromFaceCloseToFace(corridorFace, room);
9              edge roomEdge = getXEdgeFromFaceCloseToFace(room,
                 corridorFace);
10             for ( int i = 0; i < 2; i++ ) {
11                 vertices[i].z = corridorEdge.vertices[0].z;
12             for ( int i = 2; i < 4; i++ ) {
13                 vertices[i].z = roomEdge.vertices[0].z;
14             float insideXCoords[] =
                 getInsideXCoordinates(corridorEdge.vertices,
                 roomEdge.vertices);
15             float maxX = min(insideZCoords);
16             float minX = max(insideZCoords);
17             float randomX = random(minX + corridorWidth/2,
                 maxX - corridorWidth/2);
18             for ( int i = 0; i < 4; i++ ) {
19                 vertices[i].x = randomX + (−1^i) · corridorWidth ;
20             face newFace = addFace(vertices);
21             newFace.orientation = "z";
22             for ( int i = 0; i < 2; i++ ) {
23                 push(verticesToAddToCorridor, vertices[i]);
24                 push(room.vertices, vertices[i + 2]);
25         else
               // Analogical but with x and z coordinates
                  exchanged
```

5.2.2 Adding Vertices to Hallways and Removing Borders

Because the points that lie on the borders of the original square create dead ends, they are to be removed. However, the hallways need extra points to maintain their rectangle shape, as they would collapse, as illustrated in figure 5.6 with red lines. For this reason a copy, of its corner is created and mirrored.

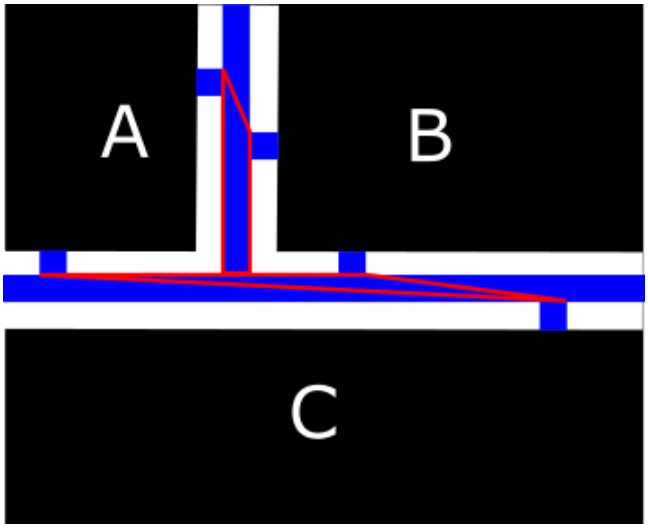


Figure 5.6: Diagram of a hallway after deleting its borders.

Since the points lie on axis x and z, finding the extra points to hold its structure is done by finding both the minimum and maximum value on either the coordinate x or z, depending on the orientation. Not always adding extra points is necessary, however, as seen the in figure 5.6. The hallway between rooms A and B is collapsed only on one side. The points will not be added if there are no border points deleted. The mirroring process is done by iterating over the original four points and finding a point that has the opposite coordinate different because there are only two options, overwriting it in such a way mirrors the point to the other side of the rectangle. Finally, the new point is added to the face.

Algorithm 12: Adding vertices from corridor parts to the main hallway.

```
25 if corridorFace.orientation == "x" then
26     vector minZ = {0, 0, minFloat};
27     vector maxZ = {0, 0, maxFloat};
28     foreach vertex ∈ corridorFace.vertices do
29         if vertex.z < minZ.z then
30             minZ = vertex.z;
31         if vertex.z > maxZ.z then
32             maxZ = vertex;
33     bool addMax = true;
34     bool addMin = true;
35     foreach vertex ∈ corridorFace.vertices do
36         if vertex.border == true then
37             remove(corridorFace.vertices, vertex);
38         else
39             if vertex.z > maxZ.z then
40                 addMax = false;
41             else if vertex.z < minZ.z then
42                 addMin = false;
43     foreach vertex ∈ corridorFace.vertices do
44         if abs(vertex.x - min.x) > ϵ // Is not equal
45          then
46             minZ.x = vertex.x;
47         if abs(vertex.x - maxZ.x) > ϵ // Is not equal
48          then
49             maxZ.x = vertex.x;
50     if addMax then
51         push(verticesToAddCorridor, maxZ);
52     if addMin then
53         push(verticesToAddCorridor, minZ);
54     push(corridorFace.vertices, verticesToAddCorridor);
55 else
       // Analogical but with x and z coordinates exchanged
```

5.2.3 Adding Vertices From Hallways To Other Hallways

Because the hallways need to be able to walk through between each other, they need a hole when they are touching; however, there are none at this moment, and upon extrusion, there would be walls in the way. Because of this, neighboring hallways add each other their respective points.

Algorithm 13: Adding vertices from hallway parts to other hallway parts.

```
1 foreach corridorFace ∈ corridorFaces do
2     foreach corridorFaceOther ∈ corridorFaces do
3         if corridorFace == corridorFaceOther or
           corridorFace.orientation ==
           corridorFaceOther.orientation then
4             continue;
5         edge corridorEdge[] =
           getEdgeFromFaceCloseToFace(corridorFace, room);
6         edge corridorEdgeOther[] =
           getEdgeFromFaceCloseToFace(room, corridorFace);
7         vector verticesToAdd[] =
           getOverLappingVertices(corridorEdge, corridorEdgeOther);
8         if verticesToAdd.size > 0 then
9             push(corridorFace, verticesToAdd);
```

To save computing time, algorithm 13 is divided into 2 *Primitive Wrangle* nodes in Houdini. The first seven lines implement one node from the original hallway primitives with four points each. The other has wired two inputs; the primary one is the hallway geometry with points in places of connectors to rooms. The secondary input brings the information about the new vertices and adds it to the hallways. In the end, the vertices in each primitive are sorted for anti-clockwise order.

Chapter 6

Grouping, Extrusion, and Materials

The chapter uses Houdini's group system to group every part separately, which allows easier control over the model and uses the fact that when the model is exported to a different engine, for example, Unity 3D, the model's hierarchy uses the group system. Then everything is extruded to showcase the dungeon in 3D space. Finally, materials are assigned to each group for a better visual experience.

6.1 Groups

Taking all of the final nodes of each chapter - rooms, the hallway, and the connectors, a *Group Create* node is assigned with their individual group names - *Room Floor*, *Hallway Floor*, and *Connectors Floor*. The latter two can also both have the same corridor group.

Moreover, Houdini allows running a function call in the parameter text fields in any node. This is used to our advantage to mark each room separately. Using a *For-Each* node to iterate over each room, a *Group Create* node is used with "*Room* " as the group name, followed up with a function call that references the *For-Each*'s iteration number. This creates a group with a unique identifier in its name. This group can also be used to encode any data into the geometry, but at the bare minimum, it is very beneficial in games to distinguish between rooms. This can be used to tell where the player is currently to trigger events or to use pathfinding algorithms for non-player characters.

An example can be seen in figure 6.1. The floor of the level was exported, and in the hierarchy on the left, each part has its unique name, which is a union of all groups. The rooms were part of 3 groups - *Room Floor*, *floor*, and *Room 'ID'*, where ID is its unique identification number. As seen in the picture, only one room is selected when selecting one of the room objects in the hierarchy.

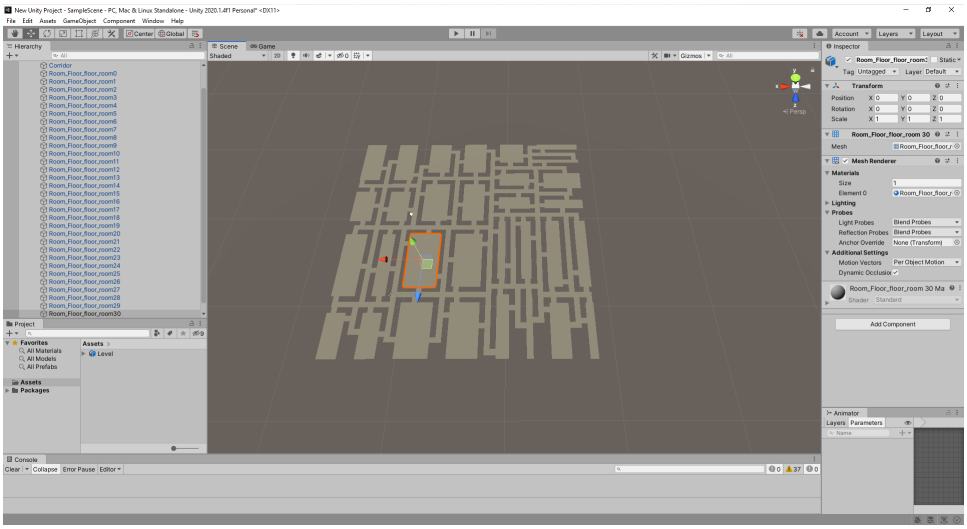


Figure 6.1: Example of room grouping in Unity 3D.

6.2 Extrusion

It is time to give the dungeon depth. Since it is a rectangular-shaped dungeon, two extrusions take place. The first extrusion gives the dungeon a height. It is trivially done with a *PolyExtrude* node. Moreover, a new parameter, *Height*, is introduced. However, the *PolyExtrude* node expects that the new primitives are an outside area of an object, and here they are walls facing inwards, which is why a *Reverse* node is used to reverse normals. In addition, only the wall faces need to be in the node; that is why the *Output Front* parameter is not no ticked.

The other extrusion uses the walls. Since their normals point outwards, the node extrudes that way as well because the extrusion follows normals. This creates width to the walls. In previous chapters, a *WallWidth* and *CorridorWidth* parameters were introduced; from them, the extrusion depth is calculated. Because the *WallWidth* defines the width and the *CorridorWidth* takes part of the depth, the *CorridorWidth* is subtracted at half of its value from the *WallWidth*, because only half of the width occupies the insides of the wall. Moreover, it is extruded at half of the value because the corridor is extruded from the other side. The formula is $\frac{WallWidth - \frac{CorridorWidth}{2}}{2}$.

Finally, a *Group Create* node is added for the walls at the end, and a *Wall* group is assigned. See figure 6.2 for the result.

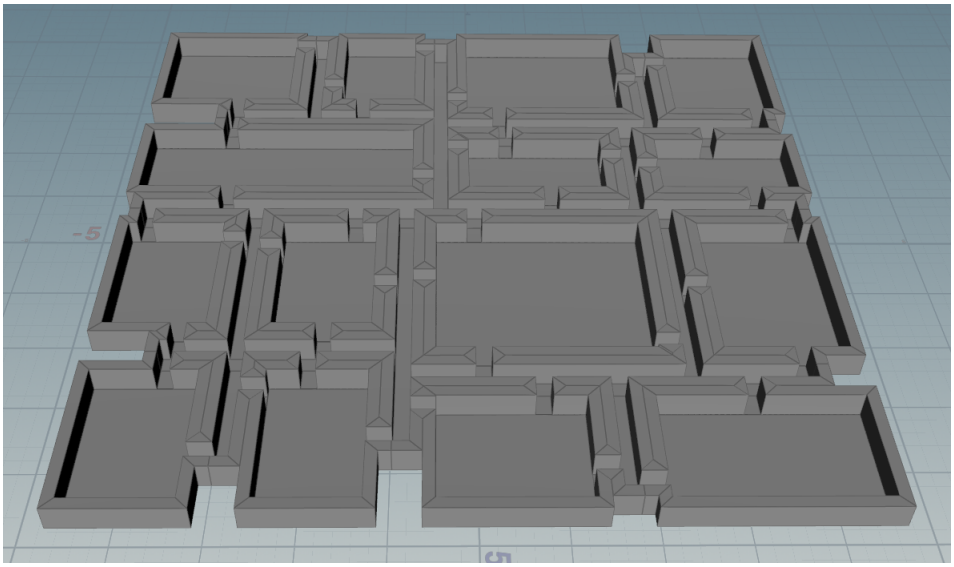


Figure 6.2: Example of an extruded level.

6.3 Materials

To give the dungeon more life, materials are given to each face. Moreover, each room is given a different random color for its texture. This has no other purpose than better visual clarity with more complex layouts and shows that it is very easy to access different rooms. With room types, each could be given a different texture and its properties could be changed. This would result in each room having a unique texture. However, this is outside of the scope of this thesis. Moreover, Houdini's materials have displacement textures; this thesis simplifies the project by removing all displacement textures.

Corridors use a Houdini built-in concrete material. First, all of the corridors are assigned a *UV Unwrap* node. Then a *UV Transform* node to scale the uv coordinates, and finally, a *Material* node.

In the *Material* node, concrete material is chosen, which was first created in the material workflow section of Houdini by selecting the built-in concrete material. The *UV Transform* node scale is set according to the material in relation to all other parameters as it sets the size of the material.

Rooms are all set analogically, but the *Material* node is in a *For-Each* node, and the material's color is changed randomly for each room, see figure 6.3 for the result.

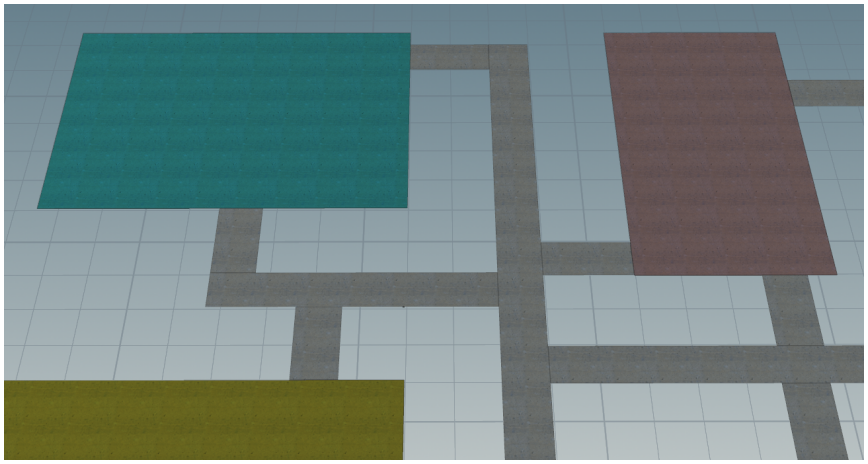


Figure 6.3: Example of textured floors.

Walls use a *UV Texture* node for unwrapping instead because each face will have its own set of uv parameters. Together with a *Material* node with a brick texture, they make brick walls in a single direction, see figure 6.4 for the result.

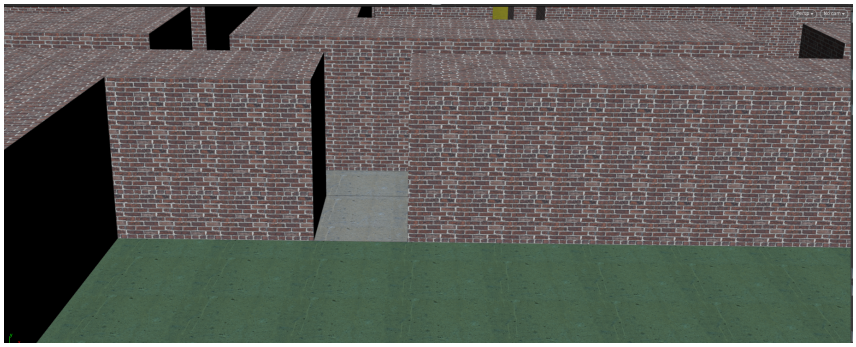


Figure 6.4: Example of textured walls.

Chapter 7

Making Doors

The dungeon's basic structure is finished, but to make it more likable and distinguish between rooms better, doors between rooms and corridors are added. This chapter discusses their creation and gives them materials as well. Because all doors are in the short connector corridors, their floors are worked with. Moreover, an extra line of code is added to the previous Making Connection algorithms 11 - since *orientation* was added to the primitive attributes, another more precise attribute is added - *direction*. *Direction* is a vector with values 1 or -1 in the x or z coordinates. It points towards to room it is connected to; in addition, a new group is added for the two points closer to the room called *doorFrame*. This marks down which points are the door frames made from.

7.1 Door Frames

Door frames play an important role for doors because they are being held by them. There are three parts to make, two vertical sides and one top part.

For each connector, two points pointing towards the room are taken. Then, they are duplicated, and a new parameter *doorFrameWidth* is introduced. The duplicated points are moved by *doorFrameWidth* in the opposite direction of the *direction* vector are added together as a new primitive. This also creates a base for the door in the next section.

Then, the base is extruded upwards by *Height*, creating a box with two holes, on top and the bottom. Using the *direction* vector to find which primitives face the room, the primitives are deleted, leaving only two side primitives, which are the sides of the door frame. Next, each of the primitives is extruded to make side blocks, serving as a door frame.

The top side is created by taking the top points from the side extrusion, adding them together as a primitive, and then extruding them down to form a block.

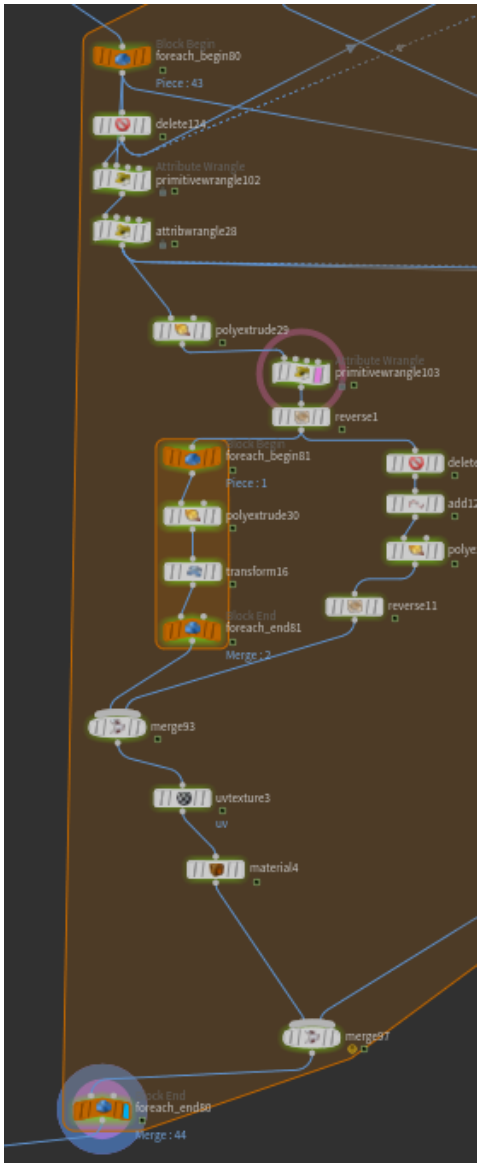


Figure 7.1: Houdini graph of door frame creating.

The Houdini implementation in figure 7.1 exactly follows the description above. It is placed in a *For-Each* node, followed by a delete node that deletes points that are not in a *doorFrame* group. Then a *Primitive Wrangle* node is used to duplicate the points and move them in the direction *direction* by *doorFrameWidth*. Moreover, the side points are grouped with their respective group names *side1*, and *side2*. The following *Primitive Wrangle* fixes the vertices order using the previous algorithm 9.

Following, a *PolyExtrude* node extrudes the primitive upwards with an option of only outputting the sides of the box. Then, a *Primitive Wrangle* node uses algorithm 14 to remove unwanted primitives. Because each side has a separate side group, if there are points from both groups in the face,

then the face aims alongside the corridor.

Algorithm 14: Removing parts from a door frame geometry.

```
1  foreach face ∈ faces do
2      bool side1 = false;
3      bool side2 = false;
4      foreach point ∈ face.points do
5          if point.isInGroup("side1") then
6              side1 = true;
7          else if point.isInGroup("side2") then
8              side2 = true;
9      if group1 and group2 then
10         removeFace(face);
```

By default, the faces' normals point outwards after an extrusion; that is why there is a *Reverse* node to reverse the normals. Two branches follow. The left branch creates the sides, and the right branch creates the top of the door frame.

The left branch iterates over the two faces using a *For-Each* node, then extrudes them by *doorFrameWidth* with an *PolyExtrude* node. Because the normals point inside, the extrusion follows them. Finally, a *Transform* node makes it shorter from the top because that is where the top part will be.

The right branch uses *side1* and *side2* groups because the top points were not assigned with the groups. Then an *Add* node creates a primitive out of the four top points. Since the normal points upwards, an *Extrude* node extrudes in the negative value of *doorFrameWidth*. However, extruding in a negative direction makes all normals point inwards, which is why a *Reverse* node is used to flip the normals.

Finally, both branches are merged, and given a texture, and an iron texture was used.

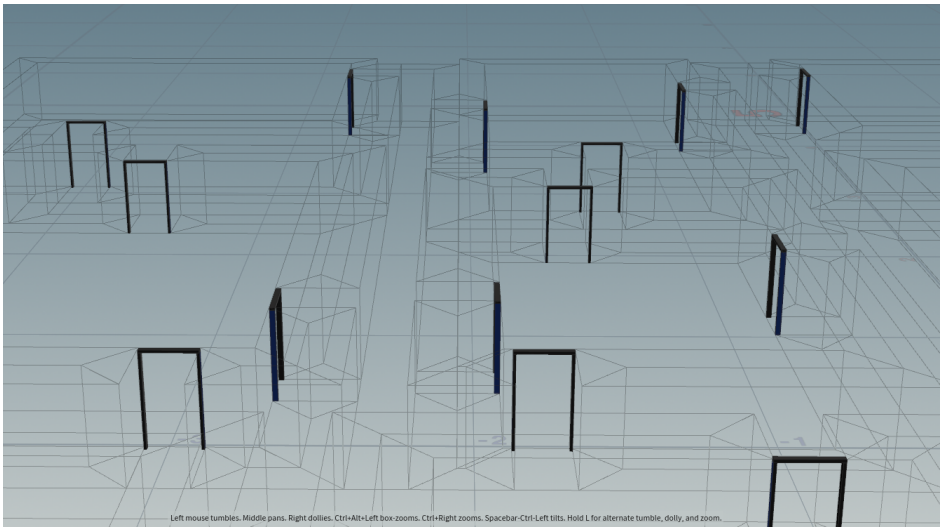


Figure 7.2: Door frames visualization.

7.2 Doors

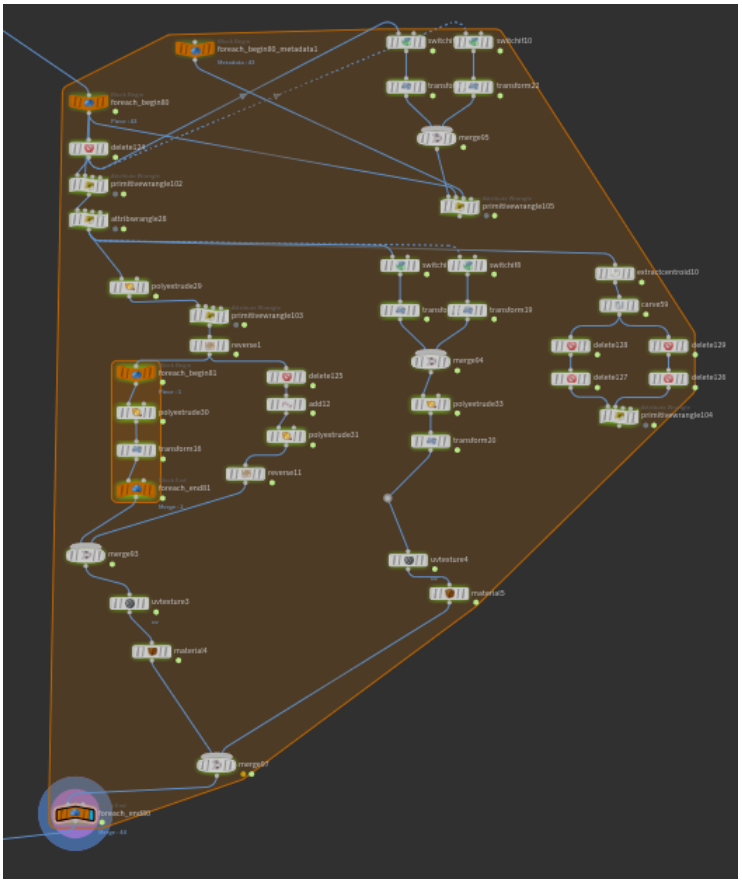


Figure 7.3: The complete Houdini node graph of door frame and door creation.

Making doors is pretty straightforward because this thesis uses a simple block as a door. However, to make it more procedural, there will be options to make the door opened or closed, as well as randomly choose an open angle, to make the scene more interesting.

First, the door is created from the leftover bottom face from the previous section and extruded. Then, whether the door is opened or not is generated, as well as the angle and its rotation pivot. Finally, the door is rotated and merged with the door frame.

The graph in figure 7.3 is quite large to digest at once, but the left branch was already covered in the previous section. The door is created in the second branch right from the door frame branch. The third branch is a supporting branch that was already seen before. It calculates the length of the edges in both, x and z direction, and it extracts the centroid too. And at last, the top branch is another supporting branch. It calculates the door angle, its pivot, and whether the door should be opened or closed.

The door in the second branch is created by first shrinking the bottom face prepared from the previous section. Because there is a door frame, the

door must get smaller by the *doorFrameWidth* amount. This is done by using a pair of *Switch-If* nodes to separate x and z direction doors. Then a *Transform* node is used to shrink the primitive down to fit inside the door frame. Then a merge node to merge both cases, and, finally, extruded with a *PolyExtrude* node upwards to create a door block.

The following transform node rotates the door, but it needs a few parameters defined in the top branch. The first two pairs of the top branch do the same thing as in the second down branch, but with a different input - it shrinks down two of the original points that lie on both the room and the corridor, to fit in the door frame. Now, one of the points is the pivot that the door rotates around. Algorithm 15 is used in the *Primitive Wrangle* node.

Algorithm 15: Getting a rotation pivot and direction for doors.

```
1  vector orientation = getOrientation();
2  vector vertices[] = getVertices();
3  vector pivot;
4  int direction;
5  floar r = rand(teration * openDoorSeed);
6  if openDoorPercentage > r then
7      r = rand(iteration * openDoorOrientationSeed);
8      if r > 0.5 then
9          pivot = vertices[0];
10     else
11         pivot = vertices[1];
12     if orientation.x > 0 then
13         if vertices[0].z > vertices[1].z then
14             if pivot == vertices[0] then
15                 direction = 1;
16             else
17                 direction = -1;
18         else
19             if pivot == vertices[0] then
20                 direction = -1;
21             else
22                 direction = 1;
23     else if orientation.x < 0 then
           // Analogical...
24     else if orientation.z < 0 then
           // Analogical...
25     else if orientation.z > 0 then
           // Analogical...
```

There are two new seed parameters introduced - *openDoorSeed*, and *openDoorOrientationSeed*. The *openDoorSeed* is for randomizing doors that are opened, which can be seen in line 6 together with a new parameter *open-*

DoorPecentage, which was created to open the desired number of doors in percentage. *OpenDoorOrientationSeed* randomizes the orientation of the doors. Doors will always open into the rooms, and the orientation picks which of the two points will serve as a pivot, which can be seen on lines 7-11.

Following lines 12-22, these calculate *direction*. If the *direction* is positive, the opening direction is clockwise; if the *direction* is negative, the opening direction is anti-clockwise. Since the door will always open into the rooms, the direction is calculated to open the door into the room. As an example, if the orientation of the connector face is positive om the x-axis, then the room is in a direction $(1, 0, 0)$ to the connector. This means that if the first vertex is the right vertex, and the second vertex is the left vertex (line 12), and the pivot is in the right vertex, then the door must rotate clockwise, that is why the direction is 1 (line 15), and if the pivot is the left vertex, then the door must rotate anti-clockwise, that is why the direction is -1 (line 17). Analogically for all other cases.

Moving back to the door rotation transform node, it just needs to be set right. Moreover, another three parameters will be introduced - *openDoorAngleMin*, *openDoorAngleMax*, and *openDoorAngleSeed*. The first two parameters speak for themselves; they decide the minimum and the maximum angle that the doors will be opened. The seed is used for the random generation, because sometimes when the doors are too closed together, they can collide, the number of new seed parameters is to prevent it. The pivot of the rotation is set from the previous algorithm and the rotation is done around the z-axis. The angle is a random number between *openDoorAngleMin* and *openDoorAngleMax*. This angle is multiplied by the *direction*, if the direction is -1, the rotation direction flips; if it is 1; it stays the same.

The last nodes are for assigning material for the door. Wood chip material was chosen for the doors because there was not any other material similar to wood. And at last, the door is merged with the door frame; see figure 7.4 to see the result.

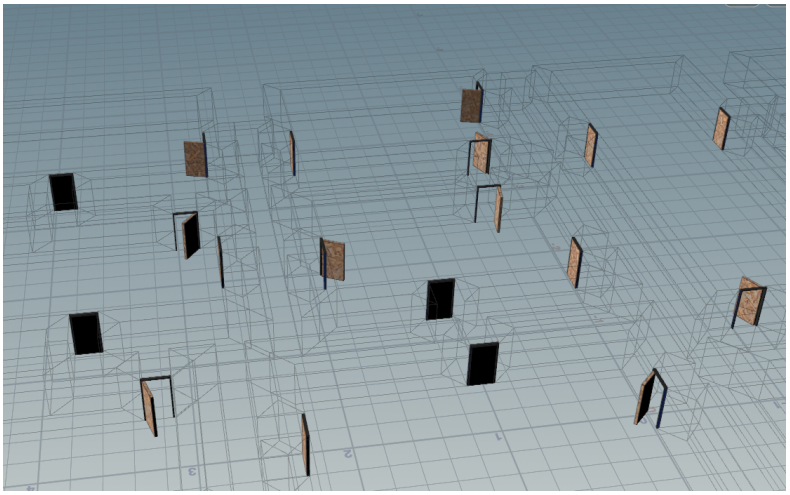


Figure 7.4: Doors with frames.

Chapter 8

Results

This chapter produces various *parameter* settings of the project to showcase and test the final results. The results will compare their look, practicality and time to create them. Each result has set their own following parameters: *Height*, *maxArea*, *minCut*, *maxCut*, *Ratio*, *CorridorWidth*, *WallWidth*, *DoorWidth*, and *DoorFameWidth*. Since, aside from the main seed, the other local seeds do not change the geometry in a way that would change the number of points or primitives, and therefore would not change the time as well, they will not be mentioned.

8.1 Tests

Tests of four varying parameters were run, all can be seen in the *Parameter Settings* table. The parameters were selected in such way that would make sense for the rooms. Because the *maxArea*, that decides the maximum size of a room, is the main parameter, that makes the dungeon more complex, the parameter graduates was picked significantly less than in the previous settings, to see the differences.

Parameter Settings				
Setting	1	2	3	4
maxArea	32.8	10	5.42	0.5
minCut	0.285	0.35	0.262	0.45
maxCut	0.561	0.55	0.629	0.6
Ratio	1	0.584	0.399	0.886
Height	0.2	0.4	0.16	0.1
CorridorWidth	0.15	0.3	0.1	0.05
WallWidth	0.4	0.4	0.016	0.1
DoorWidth	0.03	0.05	0.02	0.002
DoorFrameWidth	0.03	0.04	0.01	0.001

Table 8.1: Table of tested settings and their parameters

Setting 1				
Test	1	2	3	4
Rooms	5	5	4	4
Time(seconds)	0.205	0.274	0.206	0.211
Vertices	545	636	448	448
Primitives	606	582	403	403
Setting 2				
Test	1	2	3	4
Rooms	15	13	15	15
Time(seconds)	0.568	0.600	0.964	0.718
Vertices	2310	1996	2212	2132
Primitives	2083	1803	1990	2087
Setting 3				
Test	1	2	3	4
Rooms	28	28	28	25
Time(seconds)	1.618	1.237	1.459	0.977
Vertices	4408	4702	4576	4102
Primitives	3978	4260	4140	3712
Setting 4				
Test	1	2	3	4
Rooms	254	279	296	237
Time(seconds)	11.362	13.689	15.987	10.392
Vertices	48104	52947	56243	44726
Primitives	43669	48055	51056	40597

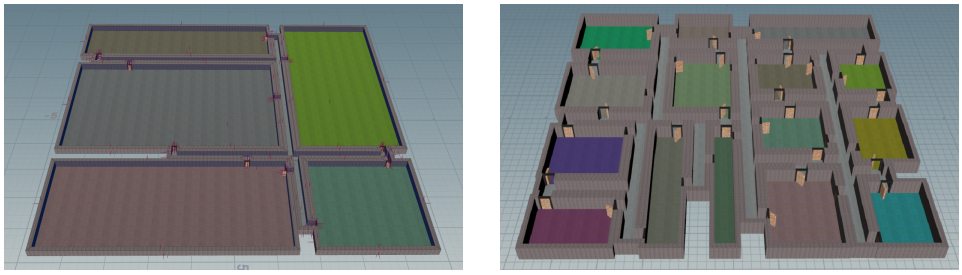
Table 8.2: Test settings and results.

With the *maxArea* growing smaller, more rooms are added, and so more time it takes to create. However, with a dungeon of a size of 200-300 rooms, 10-20 seconds is not much time, as in games, the dungeons are not usually that large.

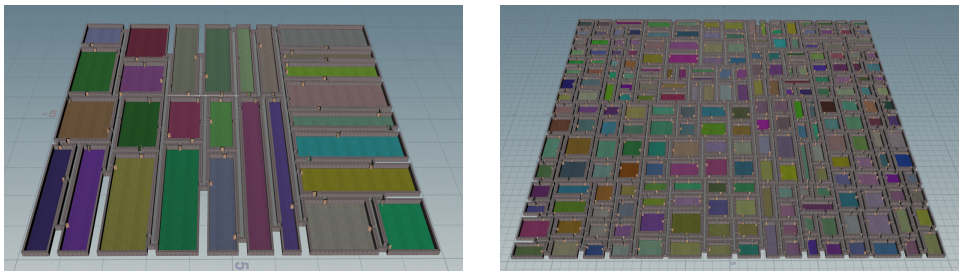
Settings 2 and 3 have their *Ratio* set to a lower value, and the *minCut* and *maxCut* deviate more from the other test, which in theory makes possible for the dungeon have more narrow rooms, which can be seen in figure 8.1.

Setting 1 has more of a square shape setting with an exception of *minCut*, which deviates significantly. Because the *maxCut* is set close to 0.5, this can create dungeons that have more square shapes in one corner, and more narrow rooms in the direction of the other corner, see 8.1.

Setting 4 was set to produce more square shapes. *minCut* and *maxCut* do not deviate much from 0.5, and the *Ratio* is close to 1 as well. This produces more regular pattern of rooms mostly of a square shape.



Setting 1 Setting 2



Setting 3 Setting 4

Figure 8.1: One example of each test settings

8.2 A Stress Test

Using the parameters from Settings 4 from the previous section with an exception of setting the *maxArea* to 0.05, a stress test was run. It run for 10 hours and 42 minutes but still did not finish, and the test had to be stopped. Most of the algorithms run for less than a minute; However, two algorithms were an exception.

The first one being the Merging lines algorithm 7. It run for 2 hours and 34 minutes. Since the algorithm runs over all edges and then over the remaining, the complexity is $O(n \cdot (n-1)) \in n^2$, and it runs recursively. However, each recursive iteration skips many operations after the first iteration, but it still is a considerable amount of condition check. At this stage, there were a total of 55441 edges before merging and 17274 edges after merging.

The other one was the algorithm for adding vertices from hallway parts to other hallway parts 13. It run for a total of 7 hours and 54 minutes, and it was in this part where it was stopped, so it would run for longer. Since it runs over all corridor primitives in two nested loops, the complexity is $O(n^2)$. Moreover, each face has four points, and there are more demanding geometry operations, from finding close edges to finding overlaps. Meanwhile, the previous merging lines algorithm only checks prepared edges already. Because there were a total of 17274 edges after merging, there were 17274 faces, which is a total of $4 \cdot 17274$ edges.

However, creating just the BSP areas is not that demanding because each algorithm took under a minute. The *minCut*, and *maxCut* were locally

Parameter settings for a stress test	
maxArea	0.05
minCut	0.45
maxCut	0.6
Ratio	0.886
Height	0.1
CorridorWidth	0.05
WallWidth	0.1
DoorWidth	0.002
DoorFrameWidth	0.001
Results	
Rooms	16348
Time	10h 42m

Table 8.3: Stress test parameters and results

modified for this test to percentages, as for a dungeon of this size, it would be impossible to find correct parameters with no dynamic restrains. There are 16348 rooms; see figure 8.2.

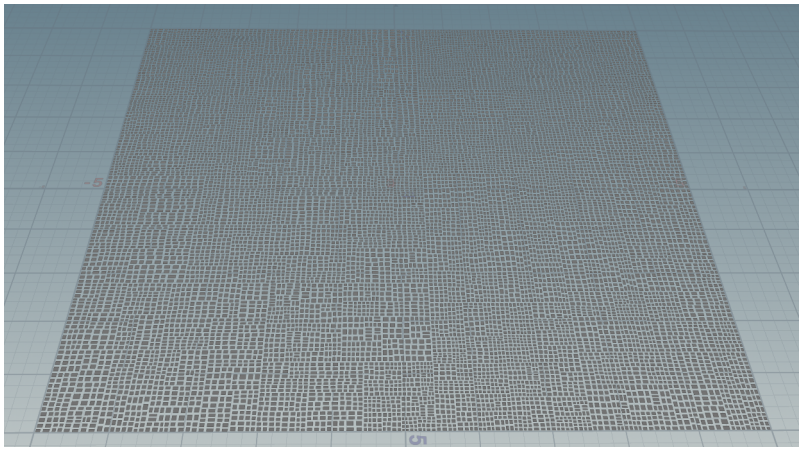


Figure 8.2: Stress test room layout.

8.3 Usable Limit

The stress test pushed the algorithms to their limits. However, this section was focused on finding a limit for generation that could be used in games. I tested many variation and setting *maxArea* as low as possible but many limitation appeared. The algorithms were breaking in the extreme case. I suspect float values with rounding errors being partly at fault, together with no dynamic restrains. However, the geometry gets demanding on hardware

and the scene camera gets hard to control as well. Parameters in table 8.4 were used for the successful test:

Limit Test	
maxArea	0.15
minCut	0.4
maxCut	0.6
Ratio	0.9
Height	0.05
CorridorWidth	0.04
WallWidth	0.7
DoorWidth	0.002
DoorFrameWidth	0.001
Results	
Rooms	870
Time(seconds)	72.876
Vertices	168619
Primitives	153049

Table 8.4: Limit test parameters and results.

A dungeon with 870 rooms was generated in a relatively short time, which means that in practice, it would be possible to use it to generate content with levels with up to 1000 rooms. However, it is very, very rare for games to use such large-level layouts, so these limitations are irrelevant. Moreover, the algorithm still takes most of the time in the same algorithms as the stress test, and the following algorithms for door generation, where the stress test did not get, took very little time in comparison and can be neglected. The result is in figure 8.3

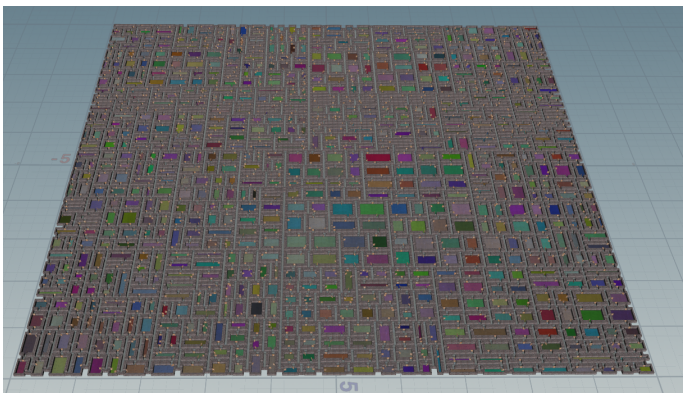


Figure 8.3: The biggest generated dungeon.

8.4 Unity 3D Test

A dungeon with the parameters in table 8.5 was created to test it out in Unity 3D. However, it was exported without materials, because a paid Houdini license is needed. In addition, a small application was written in Unity 3D - a ball that can wander around the dungeon, see figure 8.4.

Parameter Settings for Unity 3D Test	
Setting	Unity Test
maxArea	3.5
minCut	0.45
maxCut	0.6
Ratio	0.886
Height	0.31
CorridorWidth	0.2
WallWidth	0.3
DoorWidth	0.03
DoorFrameWidth	0.02
DoorFrameWidth	0.02

Table 8.5: Parameter settings for Unity 3D test.

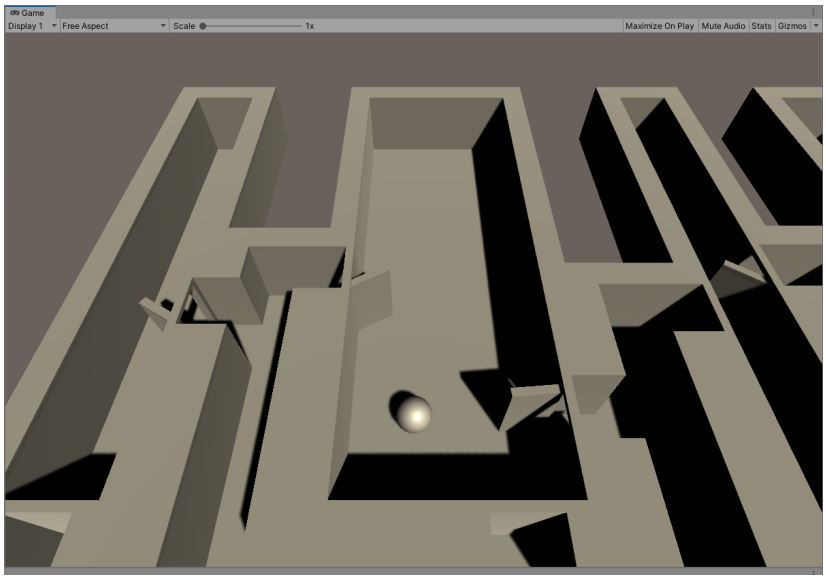


Figure 8.4: A small game with a ball wandering around a generated dungeon.

The game only consists of two scripts, one for moving the ball, the latter for camera following the ball. However, it is very trivial to use the dungeon in the game as it is only required to set up colliders and it is ready.

Chapter 9

Conclusion

The thesis succeeded in generating dungeon levels that could be used in a video game. However, there were a few limitations in Houdini that slowed down the implementation process, and more could have been achieved. The final chapter goes through the implementation one more time to summarize and review it.

9.1 What Was Achieved

There are many achievements, the most important one being the overall generation of a dungeon. However, there is much more to it. Simple dungeons can be created in just a few seconds, and the more complicated one still under a minute. A stress test was held, too, together with its usable limit testing.

A simple application, that used a generated dungeon, tested the level as well.

Furthermore, a custom corridor idea was implemented and doors with door frames as a bonus to the implementation. The levels are also exportable to game engines, and the rooms' information is stored in the hierarchy of the geometry of the dungeon.

Many algorithms could be written in a single *Primitive Wrangle* node or built up from various Houdini's nodes. In this project, both approaches were used for different algorithms.

Moreover, there is a total of 14 settable parameters that change the result to make a unique layout. These parameters include settings such as the maximum area of dungeons' rooms, a width of walls, height, percentage of the opened doors, and more.

There is room to expand. Because of Houdini's node based system, it is very simple to add features, as shown by adding doors, any other asset could be added similarly.

9.2 Flaws and Possible Improvements

The biggest limitation is probably the complexity of the algorithm, which adds vertices from hallway faces to other hallway faces because it does not

allow overly massive layouts. It does not have to be that big of an issue since games do not usually use such levels.

Parameters lack restrains. Currently, it can be difficult to change parameters because they are depended on each other. For example, a corridor width can be set higher than a wall width. This makes the wall overflow. Moreover, the corridors require more parameters because the door is always as big as the corridor. This limits the settings significantly because it can make unpleasant-looking wide doors when experimenting with very wide corridors, and height has to be also set accordingly because then the door can look too wide.

The project generates only a single floor of a level. For the future, it could be interesting to make multiple floors on top of each other, connected, for example, by a ladder.

Everything connects with everything. At the moment, the corridor always goes everywhere, but it could be expanded by counting how many connections in a single room is and limit its number to achieve more corridor variety.

The level layout can also use a more complex extrusion algorithm. The defined space areas could be potentially used for a cave system if it was not just a block room.

Appendix A

Electronic appendix content

The Houdini project file is included in a *src* folder, and all images are included in the *images* folder. Moreover, the latex files are in the *latex* folder and the specification is located in the *specification.pdf* file.

Appendix B

User manual

There are three nodes in the scene, a camera, a light, and a geometry node. The generation is done under the geometry node. To modify the parameters, they can be set in a *control* node at the very top above the top *For-Each* loop.

The main seed is linked to an animation frame; to get a different result, move to a different frame. By default, the frame control is located at the bottom of the Houdini window.

To export the geometry, there is a *File* node at the very bottom, but it is deactivated. To activate it, press the yellow button on the node. It is set to save the geometry as a *object.obj* file to Desktop, but it can be changed in the node's interface. Unfortunately, the fbx format is not supported in a free version, and the materials are not exported with obj files. The export is done upon rendering the scene in the *File* node; it is done by clicking the blue button on the node.

To see the current result, the last *Output* node must be selected for a render, but any part of the implementation can be seen by rendering in different nodes.

Bibliography

[1] Noor Shaker, Julian Togelius, Mark J. Nelson. *Procedural Content Generation in Games. Springer International* Publishing. 2016.

[2] Hendrikx, Mark et al. *Procedural Content Generation for Games: A Survey. In: ACM Trans.* Multimedia Comput. Commun. Appl. 9.February, pp. 1–22. 2013.

[3] Marco Niemann. *Constructive Generation Methods for Dungeons. Seminar report*, Munster Univesity. 2015.

[4] Togelius Julian, Kastbjerg Emil, Schedl David, Yannakakis, Georgios. *What is Procedural Content Generation? Mario on the borderline*, 2011/01/01

[5] Jiří Velebil. *Abstraktní a konkrétní lineární algebra*, České vysoké učení technické v Praze, 2020

[6] Points and vertices in Houdini `https://www.sidefx.com/docs/houdini/model/points.html`

[7] Rogue Wikipedia `https://en.wikipedia.org/wiki/Roguelike`

[8] Minecraft Fandom Wiki `https://minecraft.fandom.com/wiki/World_type`

[9] City Engine `https://www.esri.com/arcgis-blog/products/3d-gis/3d-gis/cityengine-2017-highlight-reel/`

[10] Andrew Adamatzky. *Game of Life Cellular Automata*, University of the West of England, Bristol, 2010

[11] Stefan Greuter, Jeremy Parker, Nigel Stewart, Geoff Leach. *Procedural modeling of cities*, Association for Computing Machinery, 2001

[12] Yoav I. H. Parish, Pascal Müller. *Procedural Generation of Dungeons*, Computer graphics and interactive techniques in Australasia and South East Asia, 2003

[13] Seth J. Teller, Carlo H. Sequin. *Visibility Preprocessing For Interactive Walkthroughs*, University of California, Berkeley, 2000

[14] Henry Fuchs, Zvi M. Kedem, Bruce F. Naylor. *On visible surface generation by a priori tree structures*, New York, 1980