Procedural dungeon generator.

Maurits Laanbroek
101110

A dissertation submitted in partial fulfillment of the requirements for the Bachelor Degree in 'Engineering in Game Architecture and Design'.

The Academy of Digital Entertainment.

Breda University of Applied Sciences

Supervisor's name:

Kim Goossens

Block C & D / 2013 – 2014

Date of submission

27-05-2014

INDEX

Introduction.

This paper goes through all the changes made to the dungeon generator that was the final result of my specialization project. It handles multiple parts that make up the generator and explains what the original part did, the goal for the rebuild based on analysis of the part and its interaction with the other parts. It also explains the experiments and changes made to reach the set goal. Outlining the process taken during the completion of the improved procedural dungeon generator.

Research question.

For my specialization I delivered a tool that contained the basis parts for the dungeon generator.
I propose to expend upon this project by optimizing the tool, so it can generate a random dungeon in a stable and fast manner.
The current tool is only able to create the outlines of rooms and corridors without connecting them together this makes it only usable for making an outline of a dungeon. The previously created tool can create a dungeon with up to 10 rooms between 3 to 5 minutes with a 50% chance the tool fails and crashes.

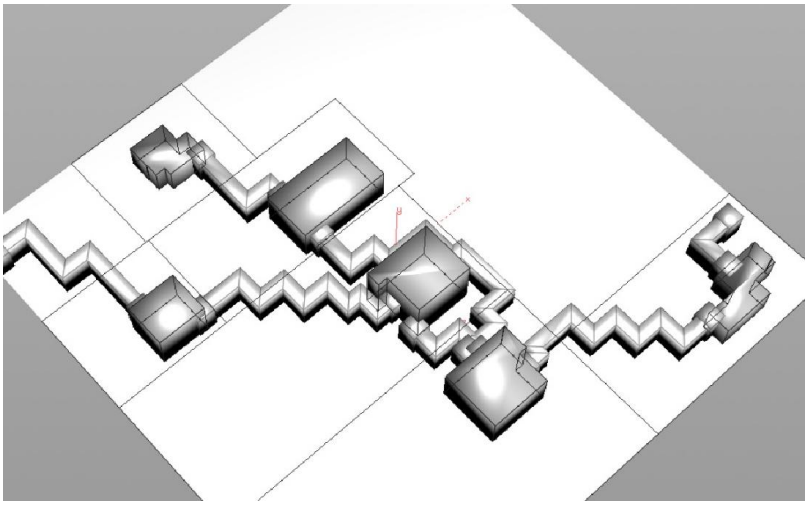


Figure 1: Result original tool

The main goal will be to increase stability, speed and usability of the previous tool. The new tool should be useful for quick dungeons prototyping and work in Unity with the help of the Houdini Engine (Houdini-Engine, n.d.) plug-in for Unity. Next to the creation of this tool I will keep a log of, recommendations based on analysis performed, different tests and experiments together with their results. For testing the produced result from the tool I propose to create a simple RPG in Unity for the testing the dungeon layout.

Component improvement logs.

The next part of the paper goes through all components one by one and explains the original abilities of each part, the expected goals bases on tests, experiments, made improvements and the final result.

Plane division.

Component: Plane division.

The plane division part is tasked with subdividing a grid or plane into smaller pieces. These polygons or cells can later be used to place rooms and calculate the path through the dungeon. The tool works by taking each polygon and randomly dividing it into two smaller polygons. These polygons are depending on the number of iterations are further divided into even smaller polygons. See Figure 2: Plane division workings.

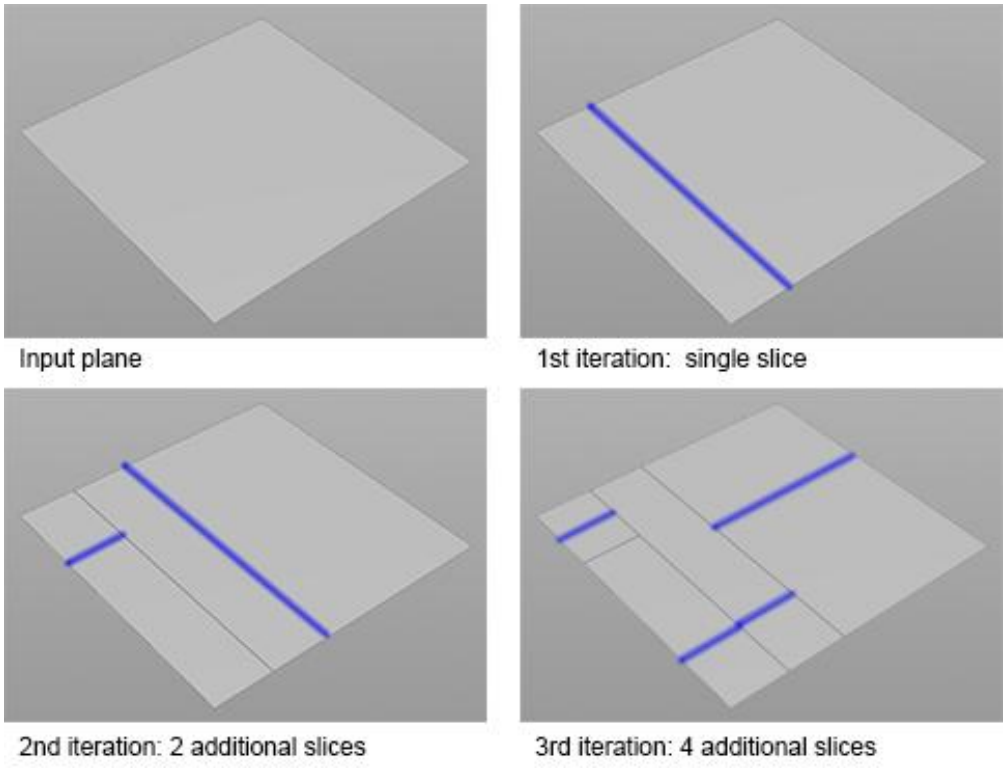


Figure 2: Plane division workings

Original component.

The starting component was able to split the grid into smaller cells similar to the final version but also gave the ability chance the position of every split. While this worked to a degree it was only useful for smaller number of divisions. With each iteration an extra subsdivision is created.

As the number of splits doubles on every iteration the interface would overflow with parameters to control every division.
To allow this type of control also requires a complex system with multiple forloops resulting in longer cooking times.

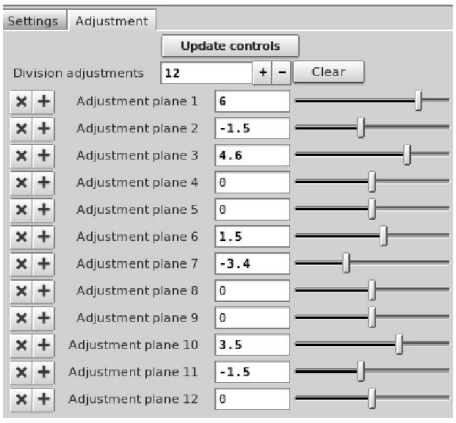


Figure 3: Old user interface

Set goals.

Remove of the ability for users to adjust individual splits and

decrease the cooking time with 25 to 45 percent. The main goal of these improvements is the creation of a tool that is also usable for different projects.

Test and improvements.

The first improvement was to remove all parts related to the ability to change the split position. This was done by removing two duel for loops from the process and remove the code used to generate the multiparms in the user interface.

The second iteration was to replace the Break node used to split polygons into a Clip node. This combined with the previous improvement resulted in an average cooking time reduction of 90%.

Third change was an addition to the interface allowing users to divide a group and not just on a whole object. While this increased the cooking time with 0.100 second it gives the tool more potential in the long run.

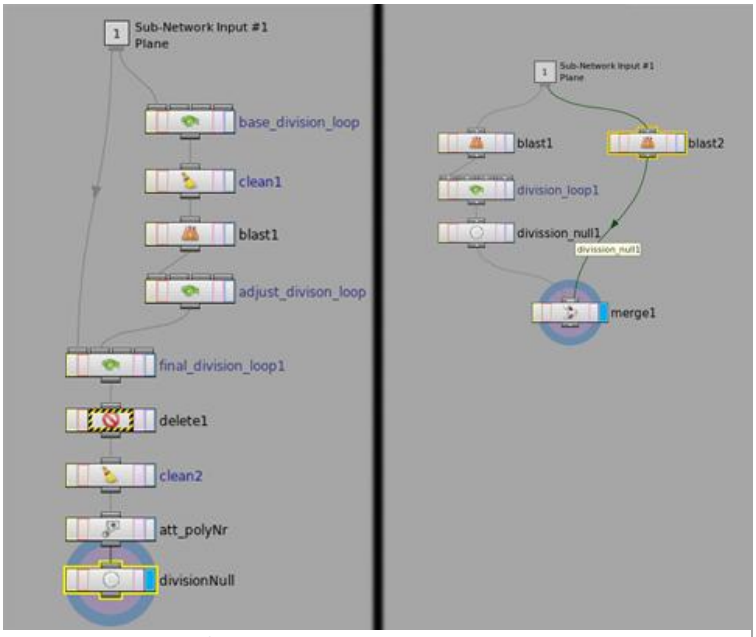


Figure 4: Division node tree

After feedback from testing and suggestions given several additional nodes where removed from the system. These where leftovers from the original tool, aiding the ability to chance split locations. Removing these resulted in a speed increase of 60% over the previous change.

The last change was a test to see if the code language used to determine which way to split a polygon would influence the speed of the final tool. Based on performance testing the version using expressions seemed to be 0.100 second faster than the Python version that was used before.

Test settings: grid: size 250/250, row/col 15/15
Tool: seed 1, divisions 2, min size 2

	Base	Test 1	Test 2	Test 3	Test 4	Test 5
Overall time:	22.71	8.157	2.906	2.944	1.203	1.115
Dual loop 1: Division loop.	10.451	8.144	2.89	2.641	1.201	1.113
Dual loop 2: Adjustment loop.	6.283					
Dual loop3: Re slice loop.	5.102					
Most time used by:	Dual loop1: Break	Dual loop 1: Break	each	each	switch	switch
Node cooking time:	6.717	5.794	1.014	0.658	0.468	0.407

Table 1: Test and improvement time results division component.

The final result is a component that is 95% faster than the original component. A possible further addition could be the ability to slice vertical orientated polygons instead of only horizontal polygons, something that is currently not required in context of the generator.

Dungeon flow.

Component: Dungeon flow.

This component creates the main flow through a dungeon and adds data to the cells. For this it uses a user defined curve and the output plane from the previous component. The data added to the cells indicate what cells have a room, if a room is from the main path or side rooms (*Or the "roomLv" value where level 1 is the main path and higher levels are side rooms*.) but also the cell number. With this data the tool can link each occupied cells to other cells from a lower level. These links are used later by the Pathfinder to create corridors between rooms.

Original component.

The original component created a randomly shaped path using a simple line between the given start and end points and then use a Mountain node to add a random distortion to the curve. It could only handle a two-point spline as input. The same tool calculated if a cell was occupied with a room or not. This data was then used by the last part to generate connections between main rooms and side rooms. This was accomplished by connecting all side rooms to the closest main rooms and when that was not possible to the closest other side rooms.

Set goals.

Use a different method to create the shape of the main path giving it an additional dimension of deformation so it can not only go sideways but also back and forwards. The ability to add multipoint splines should also be possible allowing the users to shape the main path.

After working on the Path finder described later the need arose to divide this component up into smaller parts, this was to allow more interactions between different components. The new goal was to divide the component into four parts each with their own specific task.

1. Create a curve for the main path.
2. Add values to the cells.
3. Calculate cell occupation.
4. Link rooms to each other.

By splitting this component up into smaller parts, means results from other parts can be used as input for the new parts. For example rooms could be made using the result from the occupation part and then be used as input for the link creation between rooms. This way the size of a room can influence the number of other rooms than can connected to it.

Path creation.

The first objective was to enhance the path generation method improving the efficiency and path generated.

Test and improvements

The main focus was on experimenting with different ways of creating the path curve.
Three methods where conceived and tested, each based on one of the following principles.

- L-systems.

- Path finding with a grid as input.
- Path finding using a Voronoi mesh as input.

For the L-system the rules need to allow the possibility to create a curve that would look like a “?”. The rules needed to decrease the chance of going in the same direction over every generation while increasing the chance of making a turn in the other direction. While also allowing the possibility to get a 180 degree turn. Table 2: L system rules. Figure 5: dungeon path results

Start	FA				
	A=+(45)FB:0.5 A=-(45)FG:0.5				
Left	G=+(45)FH:0.5 G=-(45)FA:0.5 H=-(45)FG):0.6 H=+(45)FFI):0.4 I=+(45)FH):1 J=+(45)FJ):0.2 J=-(45)FI):0.8	Right		B=+(45)FC:0.5 B=-(45)FA:0.5 C=-(45)FB):0.6 C=+(45)FFD):0.4 D=+(45)FC):1 E=+(45)FE):0.2 E=-(45)FD)):0.8	

Table 2: L system rules

The resulting curve is then is scaled and rotated so the starting and end points of the curve line up with the start and end points of the input curve. This method allows for a path that goes both sideways but also back and forth. The flipside is it also has a chance of going outside of the level bounds defined by the input plane of the division component.

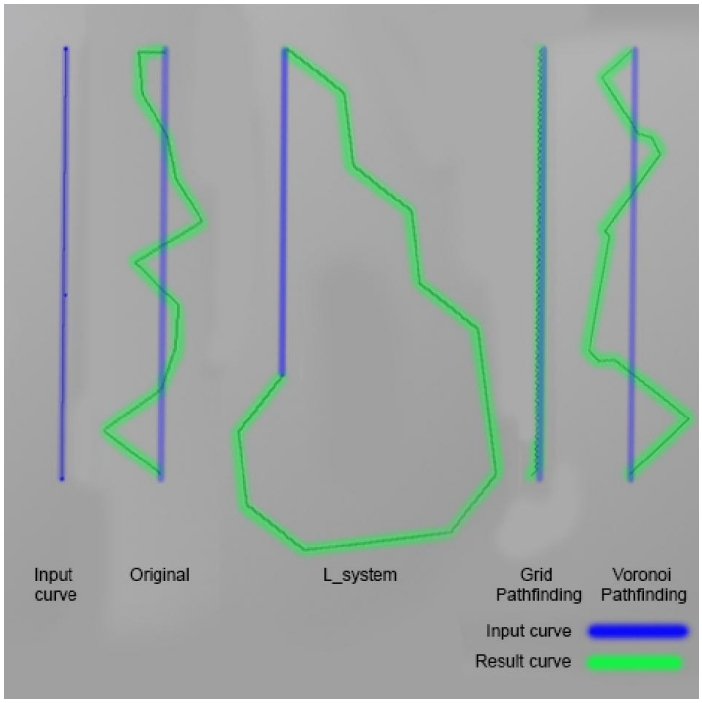


Figure 5: dungeon path results

Both Path finding orientated methods use the Find shortest path node to create a curve between the supplied input points. The first of the path finding methods would create, depending on the grid density an almost straight line between the start and end goal.
The second path finding methods used a Voronoi mesh and would create a curvier path between the start and end point. Here the size of the Voronoi cells also has influence on the result of the path, as the cells get smaller the resulting path become more direct. The resulting path then also has to be scaled and rotated so the points would correctly line up.

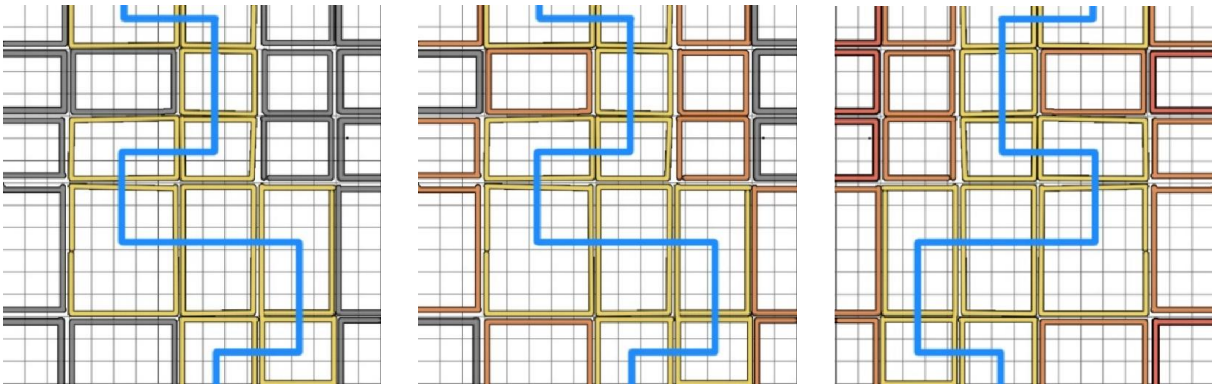
The final choice was to use the L-system, as it came down to the chance of a path not only going sideways towards a goal but also back and forth. Giving the tool a new dimension of movement over the original Mountain node. Figure 5: dungeon path results

The last addition was to add support for input curves with more than two points, allowing the user to predefine the main shape of the path. For this a simple Foreach node is used that goes over every edge in a given spline. The loop would create a curve using the L-system and fit it between the supplied edges until every edge has a new curve.
Additional performance has also been gain with the use of VEX code to replaced all previous used Python scripts.

The resulting tool is a more specialized component that is able to take a curve and create a path between the start and end points, including any additional points in between.
The resulting path not only goes sideways but also back and forward. This path is used in the second new component to calculate the cell value.

Cell value.

The original component from which this was a part of only had two types of room's, main rooms and side rooms. The goal of this part is to help expend on this and give every room a value based on the degree of separation from a main room. Figure 6: Cell value.
The original concept for this was created during my specialization phase but never implemented.



Main path cells are selected. Room lv 1: yellow | Room lv 2: Orange | Room lv 3: red

Figure 6: Cell value.

The first step was to group all cells that form the main path together. For this the part uses the curve created by the previous component and the result from the division component as inputs. At the start all cells are given an initial "roomLv" value of -1. A Ray node is used to give each cell under the input curve points a roomLv value of 1. All the cells with the roomLv value of 1 are selected and grouped for later use.
The main process uses a double forloop where the first loop repeats an X amount of times based on the value given by the user. The second forloop that is inside the first one is used to select all neighboring polygons from the last created group (at start this is the main path group). The newly selected polygons roomLv values are check, if this is -1, a new value is given based on the current cycle of the first loop and the polygon is added to a group with polygons of the same value.
For the selecting of neighboring polygons, a Group node is used with the Boundary box selection method. The boundary box is given the same dimensions as polygon but multiplied by 1.1 so the neighboring points fall in the selection area.

Cell occupation.

The starting version gave each cell an equal chance of getting a room no matter the degrees of separation from the main cells. The suggested improvement is to increase the chance of rooms the closer a cell is to the main path. This way rooms clusters more towards the center and become more spares further outwards.

For this, the tool creates a unique percentage for each room level, based on the roomLv value and the number of room Levels.
The formula for this takes the base percentages and calculates the minimum and maximum percentages.

Maximum = Base percentages + fall off.
Minimum = Base percentages - fall off.

The different between these two is then determent and the result is divided by the number of cell levels resulting in the chance modifier.
With this the chance value of each cell Level can be calculated.

*Max value – (chance modifier * cell level);*

With this code the chance of a cell containing a room can be calculated based on the roomLv value.

The last improvement is to remove the possibility of rooms that are too small from being made. Checking the cell size does this and minimum room dimensions and then either allow or deny a room.

This results is a tool that increases the density of room the closer it get to the main path but also prevents rooms that are to small from being made.

Room connections.

The last new component created from the original Dungeon flow part takes all data from the other parts and uses it to links room together. The first version connected all side rooms to the closest main room until there were no free connections left and then would try to connect to the closest other side room. The planed improvement is to change this method to the one described during my specialization research. See Figure 7: Room linking. With this method a side room would get connected to the closed room of a lower level. When the first lower room has no empty connections the rooms from the same level are added into the process so a room can only connect to a lower level or same level rooms.

The number of connections a room can have is determent by two factors. One the size of a room and two the max number of connections allowed by the user.

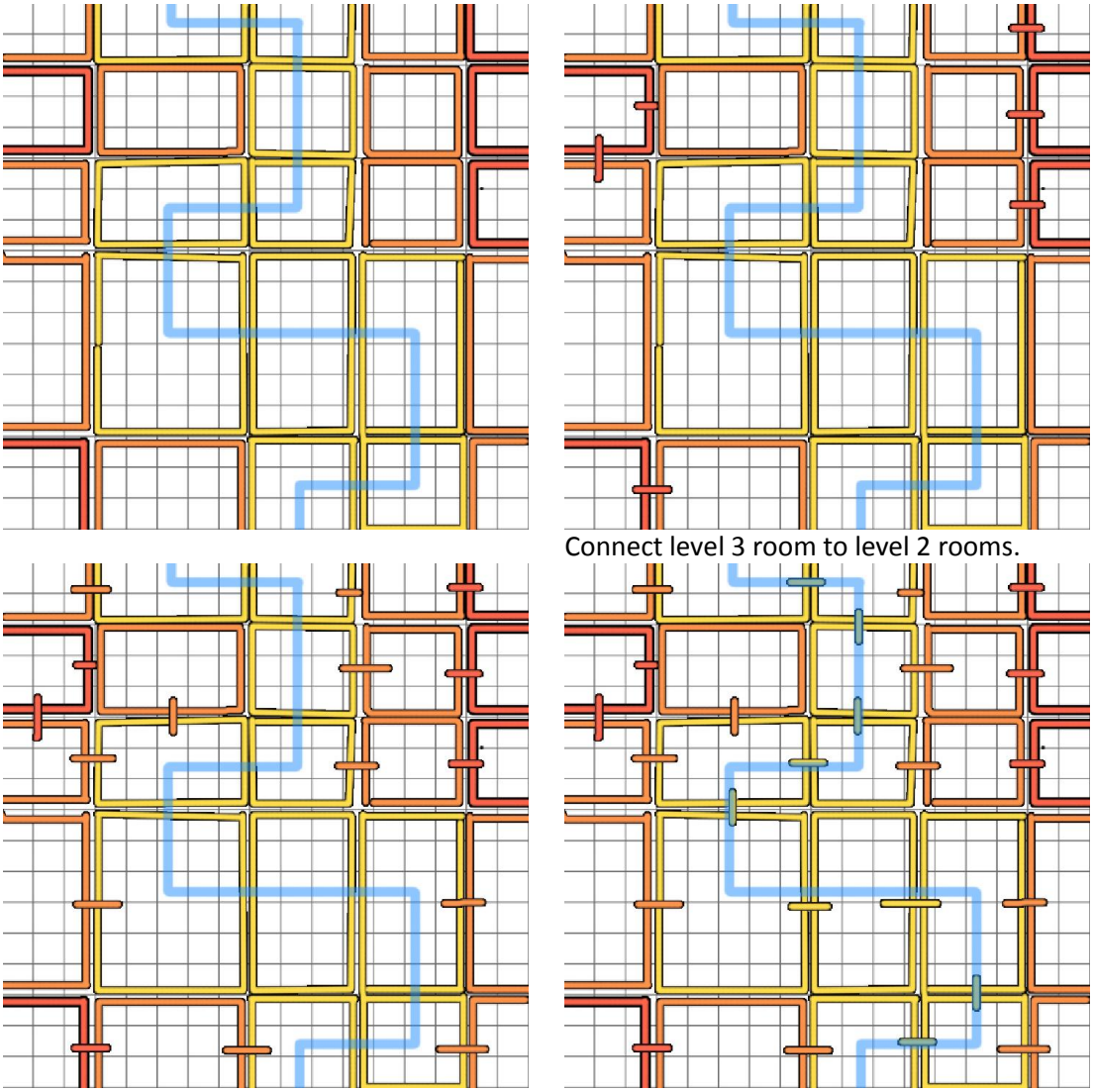


Figure 7: Room linking.

Test and improvements

The first test was to connect lower lv rooms to rooms of one level higher. This is accomplished by isolating the higher-level room and then using the Sort node to sort them based on the distance from the room currently being processed. The polygon number of the selected higher level room is then stored as a variable in the processed room. This approached had one possible problem.

If the higher level has more rooms then the one being worked on the remaining higher-level rooms would not get connected to a lower one. This could result in branches of the dungeon not connecting to the main path. Figure 8: Death branches

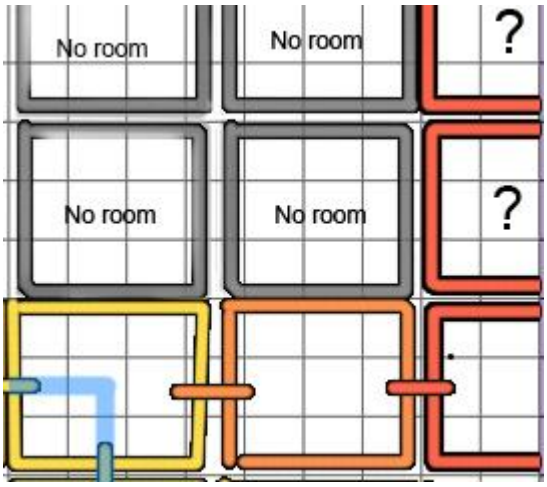


Figure 8: Death branches

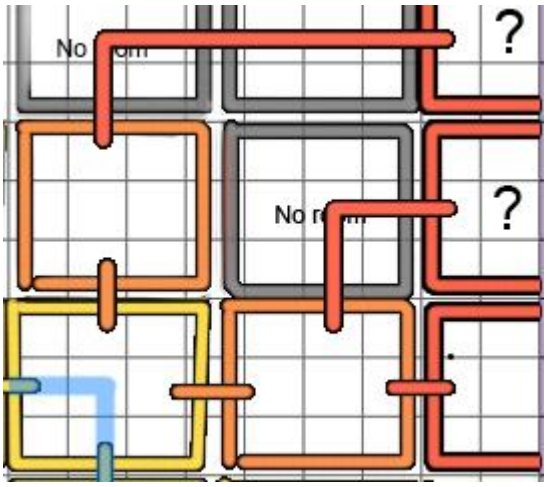


Figure 9: Connection to far away.

The second test was similar to the first one but flipped the process around. Instead of going from low to high the process goes from the highest level to the lowest. So the chance of unconnected branches is removed and each room only has to store the address or polygon number of one neighbor. This offered one final problem, cells could be connected to a cell further away or even to ones on the other side of the dungeon. This would happen if the closest cells would have no free connections, the tool would then check the next closest point until it finds a room with a free connection. Figure 9: Connection to far away.

The last test was aimed at solving the problem of rooms being connected to rooms that are far away. The test would stay the same with one difference. As soon as the first cell of the lower level rooms has no connections the rooms of the same level would be added to the mix.
This increases the number of possible connections to pick from and removes the possibility of a room being connected to a room across the map. As rooms of the same level should almost always have all their connections open for use.

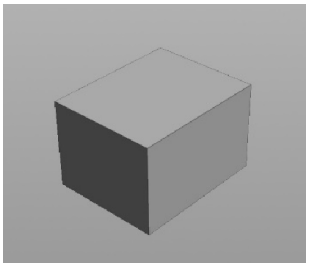
The completed component creates connections between rooms based on the roomLv value. This result in a better room branching then the original component could do.

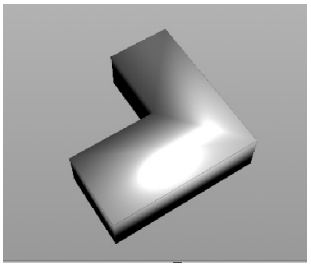
Dungeon path result.
The splitting up of the dungeon flow component into four parts helped create a more intelligent and powerful toolset. As the result of the first three parts could be used to create rooms before trying to connect them together. With the rooms created before the connections are made, the size of a room can be used to calculate the number of connections a room can have instead of using only the user set number of connections as limit. The splitting of the component also allows the use of individual parts in other projects and the possibility to give more visual feedback on each stage of the process.

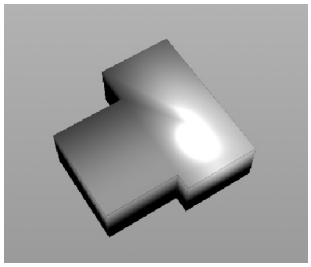
Room generation

Component: Room generation.
This component generates a single room and handles it shape, scale and positioning.

Original component.
The base component was able to create four different types of rooms and gives the room a random size independent of the size of a source cell. A big cell could get a small room and vice versa.
Figure 10: Room types


Square room


L shape room


T shape room

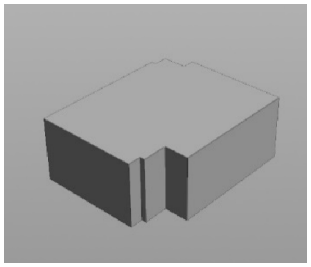

Random shape room

Figure 10: Room types

Set goals.
After a closer look at the research from my specialization the random room shape can be dropped. Secondly the size of a room should resemble the size of the original cell more closely. So a big cell has a big room and a small cell a small room.

Test and improvements
Removing the fourth room was no challenge as the node tree for this room only needed to be removed and the switch controlling the room type needed a small change.

The next improvement was to scale the rooms according to the cell size while staying within the set limits. For this a simple Transform node is used with expressions to normalize the scale to the cell size while clamping the goal size to fit between the users defined min and max size.

After some more testing the “T” shape room would often not be clearly defined and look like other types. Some tweaks in the code that controls the shape solved the problem so they are more clearly defined and stand out more from the other two types.

With the splitting of the dungeon flow component an additional change to the room node was required. So the size of a room would influence the number of connections it could support. As a room is created an outline curve of the room is created. This line is used to create the points used for connecting corridors. The outline is refined so it has points at a fixed intervals. This interval is determent by the width of the corridors. The last steps are too remove the corner points and then calculate the maximum number of connections a room can support. The maximum number of connections a room can have is calculated using these formula:

Number of points / 3.

This is done so as one corridor is made the two neighboring points can be removed preventing corridors from getting too close together in the end result. The generated line is used later for path finding and the connection limit is used by the room connection component. Figure 11: Corridor connection points.

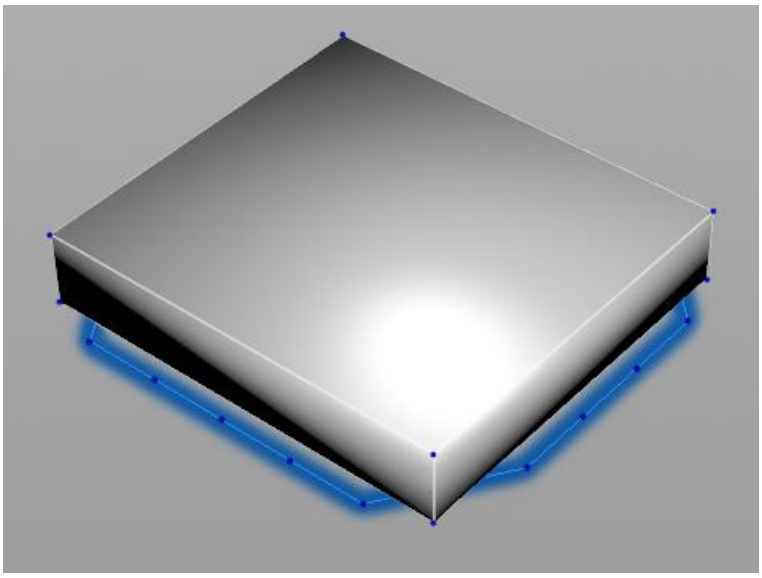


Figure 11: Corridor connection points.

The new room generator has only three types of rooms which are scaled to the size of a cell. It also calculates the maximum number of connection points a room has overall. While the changes are small they have a big impact in the final product.

Room Controller

Component: Room Controller.
This is one of the more technical nodes in the generator, it creates and manages the Room generator nodes for each cell with a room. This allows users to dive into the node tree and change the settings of each individual room, giving full control over each room to the user.

Original component.
The original component would create Room generation nodes for every room needed. If there were more nodes than room the extra nodes would be removed. Some of the time the tool would not properly update and show more or less rooms then required. The user then had to press a refresh button to recook the node and fix the problem. In some cases it would also cause a system crash forcing Houdini to quit.

Set goals.
Test if VEX can be used instead of Python for better performance, improve the stability and fix the problems causing the incorrect number of rooms to be made. If possible also order the nodes so that they won't create in a single line but are placed in a grid pattern for user friendliness.

Test and improvements.
After doing some research into VEX, the conclusion was that although faster it cannot be used to generate nodes forcing the usage of Python for this part.

The next step was to analyses the exiting code and refactor it for better performance and stability. Initial tests lead to conclusion that the original code was the most stable and the refresh button would remain

necessary. The analyses of the code lead to one of the following theories why the script would not work correctly.

1. That new nodes are added to fast to the Merge node so it can't properly rebuild.
2. That only the python node cooks and creates new nodes without the Merge node cooking.

To overcome this, several things where tried.

- Lock the merge node before creating new room nodes.
  Result: Houdini crash.
- Set the render flag to the python node, then create new nodes and set the flag back to the merge node.
  Result: Houdini crash.
- Remove all nodes including the merge node and then create new rooms. Then create a new merge node before connecting them.
  Result: Houdini crash.
- Using the "on input change" option under scrip tab of the property manager to force a recook when inputs are changed.
  Result: Error saying Cooking caused an error. No clear useful description given.
- Delaying actions using the Python sleep function.
  Result: Code would delay but also the creation of the nodes.

After some more trial and error the cause appeared to lay in the way Houdini handles code. When you normally create a script or class in for example Unity the moment a command to create an object is given the object get created. For example if I tell Unity to instantiate an object, the object get made before the next line in the code is processed. In Houdini if the same command is given it would store the command and continue with the code, only once the entire script has been processed the object gets created. This difference causes problems if you do not keep it into account when making a script.

Because of this difference, Houdini crashed allot as I wanted to do things to non-existing nodes. I learned three things from this.

1. *Houdini does not handle these kinds of mistakes with grease and instead just crashes without a useful debug message.*
2. *I need to keep in mind when working in Houdini that objects are made and destroyed after the script is done.*
3. *Houdini is not made for programing.*

With this insight the solution to the stability issues and wrong room amount was to separating the different stages of the process into separate nodes, this way all room nodes are removed by the first Python script. The second script then creates new nodes and the last one then connects them to the Merge node. This solution removed the need for a refresh button and stopped Houdini from crashing. While making the node creation script, additional lines were added to position new nodes in a grid pattern. Figure 12: Old and new room controller

The final problem that occurred, was that rooms would not update if the global room parameters where changed. This was because the Pyton scripts did not need to recook after these variables where

changed, as they did not use any of these variables. To fix this problem the first Python node takes all the parameters for the Room generator nodes and adds them up together as a single parameter. The script self then calls up the result but never uses it. When the room settings are changed the remove script recooks, forcing all the other Python nodes to recook also.

The resulting node generates a Room generator node for each room in the dungeon allowing users to dive in the node and tweak each room separately. The created nodes are ordered in rows of 10 and correctly update all the variables when the room settings are changed. Together with the better preforming components, the cause of the crashes is fixed and the code is less complex.

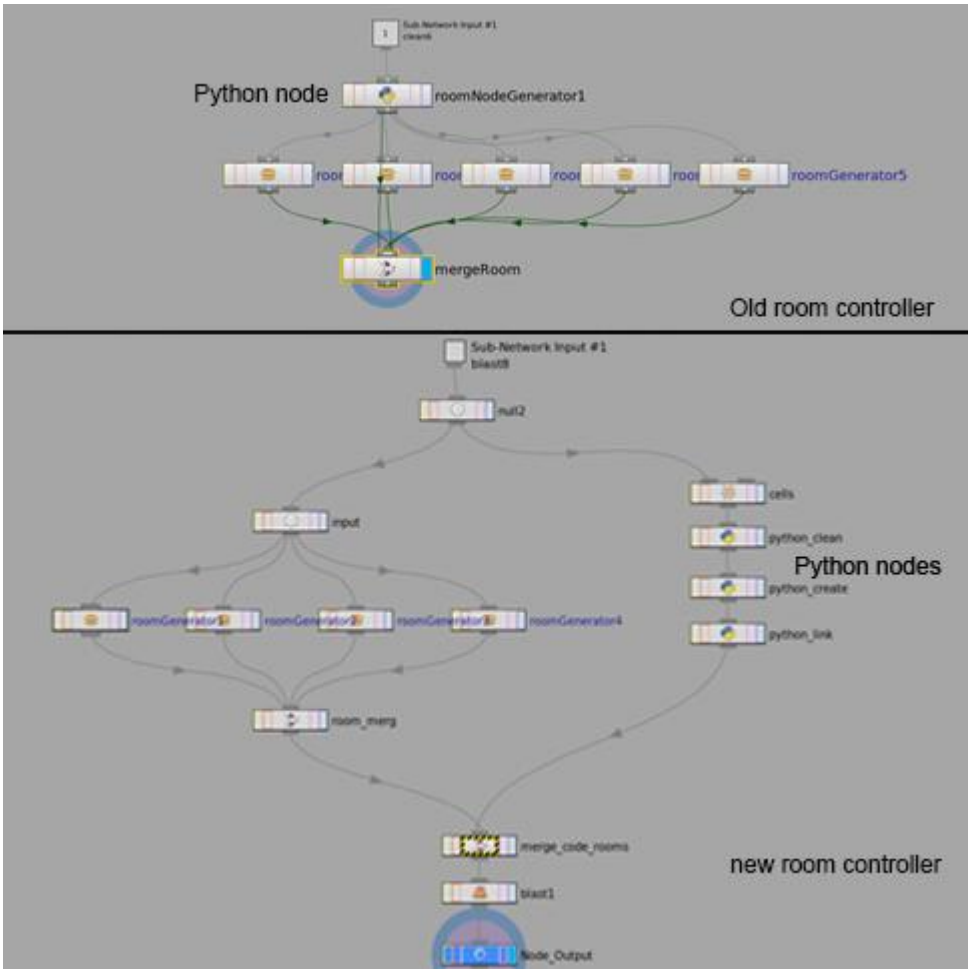


Figure 12: Old and new room controller

Corridor path finding.

Component: Corridor path finding.
This component takes the created rooms, the connection data and a grid for navigation then uses these to create the path for the corridors between rooms. It isolates each room in turn and create a path to the room it's connected too.

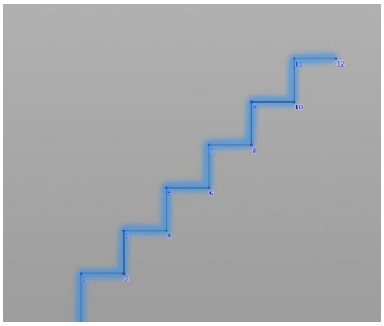
Original component.
The original component used a self-made A* path finding script to get the shortest path between point. It would isolate a room and the room it's connected to. Once isolated it selected the two points closest to the center of the other room and uses a path finder together with a navigation mesh to find the shortest path between the two points.
Once the path is found it's stored in a group, the connections points are removed from the room so they can't be used again and the component continues with the next room.

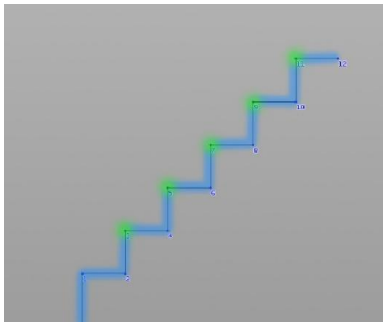
Set goals.
Replace the self-made path finding section with the new Find shortest path node from Houdini.
Give the user the option to refine the path removing any jagged diagonal lines while also removing the neighboring connection points from the rooms so corridors won't get too close to each other.

Test and improvements.
The first stage was to remove all part of the original A* path finding section and replace it with the build in version. The rebuild brought some flaws in the other components to the light, which resulted in the need to split and reorder components before continuing working on this part.

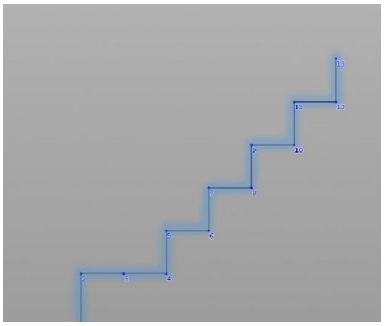
After redefining some part the next step was to remove the neighboring points of selected connection point, preventing corridors from being too close together. The first test used a VEX sop to select the neighbors and then remove them. The neighbors function (sidefx_help_files, 2014) of VEX in the Attribute wrangle caused allot of crashes when there were too few neighbors to select or the same one would be selected twice. This was solved by removing the VEX node and use a Blast node instead. As the line was already sorted by distance closest to the target room points 0,1,2 would always be neighbors and can be easily blasted away.
This showed me that I sometimes want to do thing in a more complex way then needed.

The following problem was to refine the result from the path finding node. Using an AtributeWrangle node to calculate the angle between points and then comparing the angles one by one to find specify patterns. If a set pattern was found the point would be moved based on the X and Z location of both neighbors decreasing the size of the diagonal line. The user could specify the number of times the script would execute until the path is optimized. Figure 13: Path refinement.

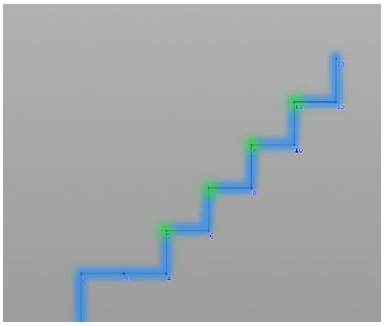




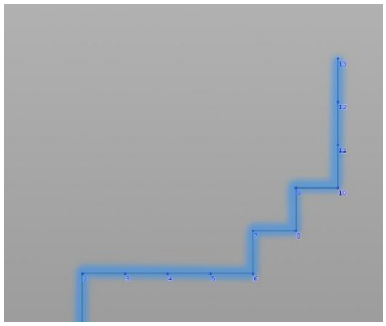
Select points after detecting patterns



Move points based on position neighbors



Select points after detecting patterns



Move points based on position neighbors

Figure 13: Path refinement.

Additional tweaks where made to improve the performance, for example moving the creation of the navigation grid to outside of the component. This way it would not be remade for every room and save time.

Mesh generation.

Component: Mesh generation.

This component takes all the rooms and corridor curves and creates the final mesh.
It creates the corridor mesh, creates holes in the room for doors and links the meshes together.

Original component.

The original part was incomplete and could only create a separate mesh for the corridors but not connect them to the rooms. The final result was a dummy displaying the outline of the level.

Set goals.

To create a fully working component capable of taking the input and produce a playable level.
An extra addition would be the ability to give a thickness to walls, resulting in a more believable dungeon geometry.

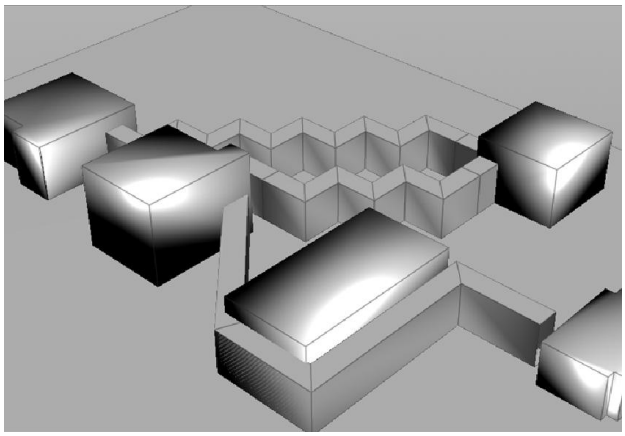


Figure 14: Original dungeon mesh

Test and improvements.

The first experiment was using a Copy and Box node to create boxes along the corridor splines. While the corridors would get a good mesh the end boxes would not line up correctly with the rooms. Figure 15: Corridor mesh problems. 50% of the time the box corridors would fall short of a room or have an offset to the side. The cause was a combination of the navigation grid being aligned incorrectly and the distance between the connections points and the rooms.

By increasing the gap between the connection points and the rooms, together with offsetting the navigation grid fixed the problem.

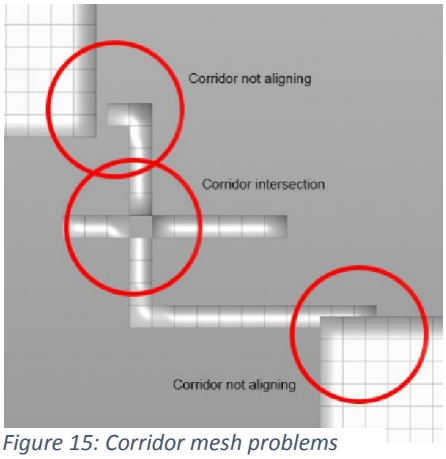


Figure 15: Corridor mesh problems

The following improvement test was to merge the corridor boxes together to generate a single corridor mesh followed up by connecting the corridors to the rooms. Using a Clean node to remove all overlapping polygons creating a single corridor. This did caused problems when two corridors would cross, the top and bottom polygons at the intersection would then also disappear leaving a hole. Figure 15: Corridor mesh problems. Also when a corridor ran next to a room the dividing wall would be removed resulting in holes and unwanted connections.

To solve this the corridors and doors needed to be created separately. By first creating boxes for each point in a corridor and then using a “Clean” node to remove all overlapping sides creating a usable corridor mesh.

Each door point would then get a box, which is then merged, to the room. The next step removes the door boxes using a group made earlier. This would leave a hole in the wall connecting the room to a corridor. (Figure 16: Corridor hole creation)The same process is also done the other way around removing the room from the door and also for the connection between corridors and doors. This created three different meshes one for the corridors, one for the doors and one for all the rooms. The resulting meshed have holes on the spot where there connect. Figure 17: Mesh components

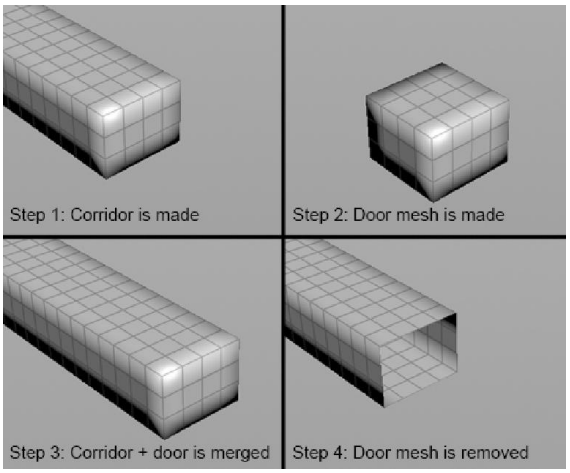


Figure 16: Corridor hole creation

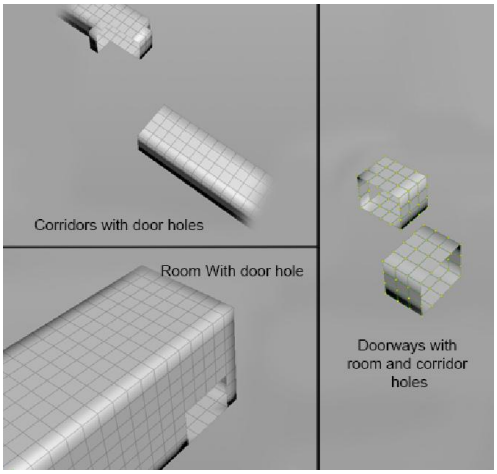
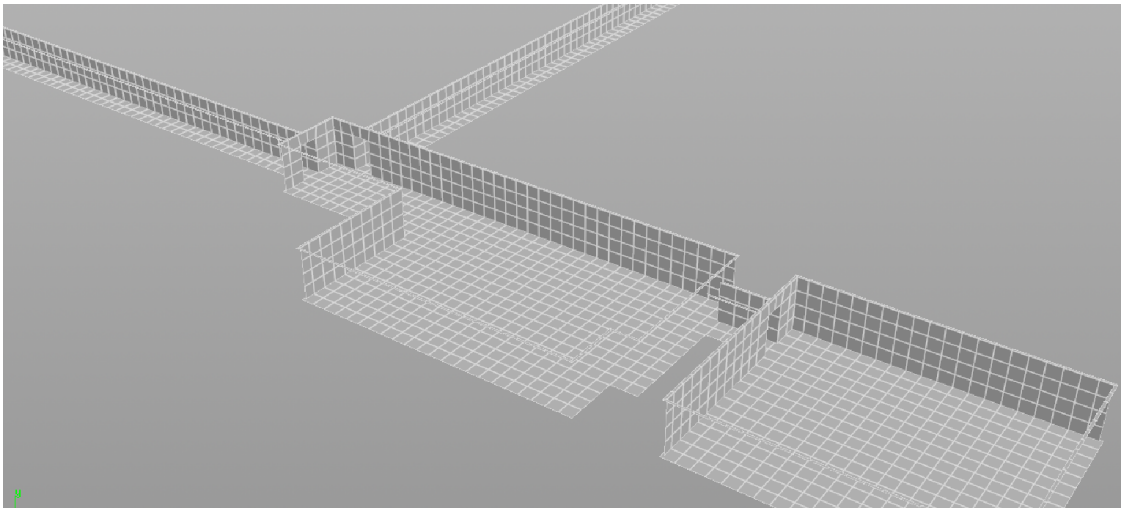


Figure 17: Mesh components

While this fixed the problem of connecting rooms and corridors without removing the wrong walls. It did not fix the overlapping corridor problem. This was done by merging all overlapping points from the corridor curves together to prevent two boxes from being made on the same spot.

For adding thickness to the walls all sides of the rooms and corridors where grouped and using a Polyextrude node indented to give them thickness. Other options tested where the Extrude and Peak node. Both fail either on creating straight walls or on creating corners that connect.

The new component creates the final dungeon mesh while also giving the connecting walls mess.



Houdini engine, Unity conversion.

Once the entire generator was working it could be converted for use in Unity (Unity3d, 2014).
As the Houdini engine is still in testing face, little documentation is available on the subject with the only sources available being videos and samples provided by Side effects (sidefx, 2014). For converting the tool to Unity, the best method was to do it one part at a time. The first component was the grid division, which worked as planed in Unity with a small difference in result. The resulting polygon plane was displayed as tri's but this had no further effect on the rest of the process, as it was more a visual difference.

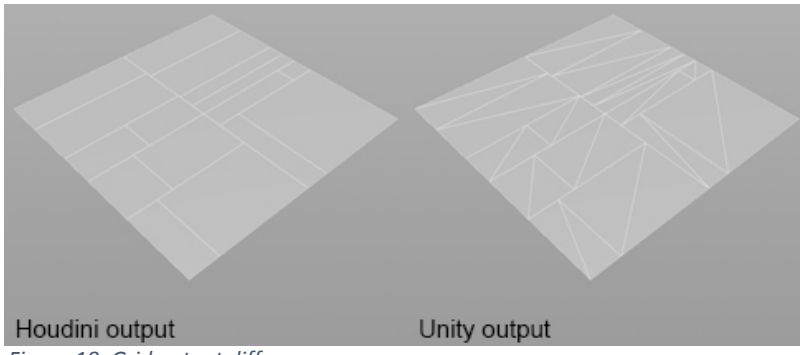


Figure 18: Grid output difference

The dungeon flow component that helps create the main path through a dungeon required more work and analyzing of samples from Side effects. This was needed to find out how to give the user the ability to manipulate a curve used as input for the component. Allowing the curve to be manipulated in Unity requires the curve to be added to the "editable nodes" bar under the basic settings of a digital asset.
The integration of the cell occupation part caused more problems as the python script used to calculate the room chance did not work in Unity. To overcome this the script had to be taken apart and recreated using expressions. This resulted in a more complex code as different parts of the code are performed in separate parameters. Having Python not working in Unity also meant that the current method for generating rooms would also not work. This was solved with the use of a Foreach node, this removed the ability to go in and change any of the rooms like the original concept, which was not feasible in Unity anyway so nothing was lost.

Implementation of the remaining parts gave no problems and everything worked as needed.
A problem arose during testing, as room meshes would disappear for no obvious reasons, while they would appear again when changing parameters. Cleaning up groups from the end mesh had also the same effect.

During testing to recreate the problem it only occurred when the poly count would get above 10,000. This combined with earlier observations that every group created in Houdini would result in a separate mesh in Unity lead to the conclusion that the number of polygons was the cause of the problem. Where before cleaning up the groups, the maximum number of polygons would be below the limit, removing all the groups and forcing everything into a single mesh resulted in the limit being reached and the meshes to disappear. By adding a function that groups meshes into smaller chunks solves the problem.

Expanding the tool.

To expand on the created generator and show some of the possibilities the decision was made to add several additional tools to the package. For example an automatic collision mesh so the levels becomes playable from the start. But also a tool that uses the data provided by the generator and places enemy's throughout the dungeon with increasing difficulty lev7els.

Collision mesh.

For the generator to be useful for use in Unity the dungeon should have a collision mesh, so a level can be made and tested. After going through videos and example projects the creation of a collision meshes comes down to creating a group named "collision_geo". The Engine takes this group and automatically adds a mesh collider to it. Baking the result does cause the mesh collider to lose its input mesh, but this can be fixed by dragging the correct mesh into the collider manually.

Player starting point and level goal placement.

The new part should allow the user to manually place the starting and ending points of a dungeon and also allow this to be done automatically. The second ability should be the possibility to add and instantiate Game Objects on these points. The first ability uses the first and last room as start and end points or a curve for manual input. In the second case the room closets to each point is used as start and end rooms. This is done in the main tool so the positions have effect on all others expansion parts.
The new asset needs to take these points and create placeholders or place Game Objects. For the placement of Game Objects some deconstruction and analysis was needed of the supplied samples. While it took some time to find out, the principle is easy.
Give the points an attribute value called "instance" with a reference to a Geometry node. Then load the points into an Instance node which than automatically creates the required objects on each point.

Recalculate roomLv value.

With the ability to manually define the start and end room the roomLv values also needs to update so they can be used later by other extensions. First a point is made for each cell with a room. These points contain all data needed from roomLv to address of the cell it's connected to. Using this data all points are connected to each other and used as navigation mesh for the path finding. Then to find the new main path a Find shortest path node is used to find the route between the points set by the user. The resulting points are given their new values based on point numbers. For this a VEX node checks the number of points between the current point and a main path point to get the new roomLv value.

Create a unique code for each room

adding a unique code to each room opens the possibility to calculate the position of each room in the dungeon and use this information for the placement of enemies or items.
The code is created by a VEX node that checks all points from lowest to highest room number and calculates the correct code for each room.
The code is created using the code of the room it's connected to and the number of other rooms that already checked the room before. The code for main path rooms is their room number.

For example an lv1 room has 2 lv2 rooms attached.

The room code for the lv1 room is "0" as it's the first room in the dungeon The number of room that have check this one is also "0" as no other room has check up on this room.

Then the code checks the first lv2 room, it takes the code from the lv1 room "0" and the number of other rooms that have check this node, which is also "0". With these values the code for this room is made. "0,0".

The value storing the number of checks in the lv1 room is then incremented by 1.
Then the second lv2 room is process.
It takes the code from room 1 and the number of rooms that checked this room and generates the room code "0,1".
The value storing the number of checks in the lv1 room is then incremented again by 1.
This process repeats for every room.

With this room code the distance of a room from the main path can be calculated and the number of side rooms. This is useful for placing monsters or items in a dungeon. As you can estimate their required value based on the room code. The new data is stored in a separate Geo object called DG_data which can be linked to extensions like and enemy placer.

Enemy placement.
This extra tool in the package is one that handles the population of the dungeon with enemies. For the placement of the monsters it uses the same methods used for placing the start and end points, with a small change. This change allows multiple difficulty levels in monsters bases on the location in the dungeon. The tool takes the DG_data as input to calculate what room gets witch tier of enemies.
The tool takes the number of rooms in the main path and divides it by the number of tiers. Using this value the tool can calculate what tier needs to be placed in what room. To calculate the monster tier for the side rooms the same initial value is used but with a multiplier so the difficult increases slower for side rooms compared to the main path.

Unity self allows for multiple Game Objects as input for instancing. This allows users not only the ability to place three tiers of enemy's but also multiple variations of enemies in a single tier. The Houdini engine randomly picks one of the options for each point.

Change room location and new rooms.
With the ability to use multiParms in Unity the option to tweak room locations is possible.
The user can select a room by number and apply the transform. In the component all cells are transformed to single points. These points have the same order as the rooms and contain all the needed data to select the correct room. By looping trough these cells the correct room can be isolated and the transform can be applied.

Adding new rooms requires a different method. For this an extra asset is needed that accepts multiple inputs and merges them all together like a reversed fork.

This allows users to add up to 10 rooms to the dungeon without adding congestion to the user interface of the base tool. A second new asset is made for the rooms. This contains a room generator and a simple polygon used for the dimensions. The user can add ten of these assets to the previous asset, which is in its turn linked to the main tool. The main tool also required some modifications for the path finding to include the new rooms. For every new room an unused cell is selected then the location is changed to match the location of the new room. At the same time a value is changed so the generator know this cell has a room and will be included into the later process.

Add corridors.
Changing corridor paths created by the tool is not possible as the engine does not allow dynamically created curves too be manipulated. Using the same technique as above new corridors can however be

added. This time a rebuild of the path-finding component was required to include the new curves. For this a second path finding part is added that takes the curves from the first pathfinder and add the new curves to create the final paths. The new part refines the original paths and then uses these points plus the points from the new curves to create the new corridors.

UV mapping and material applying.
To add materials to the mesh required research into adding material using the Engine and on UV mapping in Houdini. Adding a single material to the whole dungeon would not allow for the freedom to customize the dungeon as needed. For this the ability to apply unique materials to both rooms and corridors while also making a distinction between floors, walls and ceiling was needed. This is accomplish by creating a separate group for each of the parts and using a tool provided by Side effects and attribute value is added to each polygon that tell the Engine which material to add.
These materials need to be placed as a direct child of a folder called Resources else they can't be found by the engine.

To create the correct UV maps the Uvunwrap and Uvtransform nodes are used. This creates the required map while also scaling it to a size that allows for easy tiling of textures by the user.

The last three additions to the unity package are also incorporated into the Houdini version of the dungeon generator. Although in a different way that allows more control over the dungeon. For example the created corridors can be transformed while an infinite amount of new ones can be added.

Final result Unity.
The basic Unity tool combined with all the extra asses allows for almost the same range of abilities as the Houdini version. The tool allows the user to easily generate dungeons and test them. When combined with the extra tools it can also populate the dungeons with the starting point, end point and enemies or add additional rooms and corridors. Amongst the changes made to the Unity version is the option to select a stages of the process to see the result, which helps reduce the process time when tweaking settings.

Fail tests.

Multi floor dungeons.

The current tool creates single floor dungeons with this expansion users would get the ability to add additional floors. As this was an extra, I set a time limit of 2 weeks to get it working else it might take up to much time from more important things.

For the implementation of the new function a systematic modification of several nodes was needed. Some changes where relatively easy to implement, three components required a more extensive overhaul. These being the navigation mesh for the path finding component, the path finding component self and the dungeon mesh generator, which generates the final mesh for the dungeon gray box.

Path finding mesh.

This was initially a plane with holes on for each room so corridors would go around the rooms.
The newer version required diagonal lines connecting two floors together and these connections had the following requirements.

1. Line up with points from both floors.
2. Change length depending on the floor height.
3. Not intersect with rooms when the final mesh in generated.

And if possible:

4. Prevent corridors from creating a “Z” shape. Where a path would go up along the diagonal and then over the same axis go back resulting in a corridor that would interest itself.
   Figure 19: Path self intersection (Z shape)

The fourth would be in the end be handled by the Path finding component for performance and design reasons.

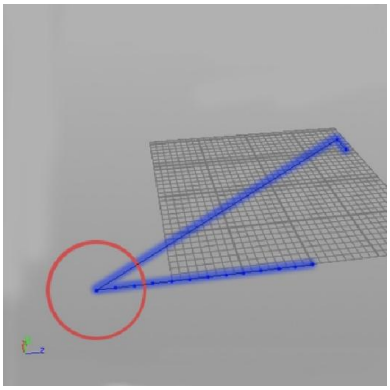


Figure 19: Path self intersection (Z shape)

The first iteration created a polygon plane for every floor and a diagonal line in four directions for every point on the polygon plane. These diagonal lines are used by the path finder to move between floors. While initially working it was slow to process. To get the lines the correct length the tool was given a fixed angle of around 35 degrees which is then used to calculate the correct length of the lines.
To make sure the diagonals would connect with points from the next grid a Fuse node is used with a wider consolidation range than normal connecting diagonal lines with the grid.

Diagonal lines or ramps would collide with rooms and other corridors when creating the final mesh. To prevent this an experiment was to remove points from just above and below the ramps to prevent “Z” shape paths leaving holes in the navigation mesh.
The second experiment was removed diagonals below the rooms. To select the diagonals points rooms where duplicated and resized to match the height of a corridor. The new rooms where then placed below their counterpart and used to select and remove diagonal lines. At the same time the grid density for the diagonals was reduced by 50% for performance.

While removing points from above and below the ramp to prevent “Z” shapes would not be optimal because of performance. It also cause problems with the path finding preventing the path finding component from creating straight paths, because of the many holes in the navigation mesh.
The last two changes did their work in preventing unwanted intersections and raise performance. So only the second modification was kept and the first one was removed.

The third series of changes would be to the grid used and the position of the diagonals. The navigation mesh got changed from a polygon grid to one created from lines. This allows the removal of points or edges from the grid without creating tri’s what would result in diagonal paths. The diagonals were also given an offset to prevent ramps from intersecting with each other. The second change made was a reduction in grid density for the diagonal lines to prevent ramp from intersecting.

Final changes were made to improve performance and mainly involve in removing or relocating nodes.

Path finding component.
The original component worked with the new navigation mesh in creating paths between rooms. The main reason for changing this was to prevent incorrect paths from being made like a “Z” shapes path. This part required more experimentations then any simple modifications.

The first test used VEX code to detect problems and then move points to the side of the ramp far enough so corridors wouldn’t intersect with the ramps. While it worked points would always be moved towards the same direction based on their directional axis. The final version needed to have more intelligence so points would be moved towards the side closest to the target, instead of always to the same side. This required a more complex version of the code and the same result could possibly be achieved using normal nodes. To prevent relying on complex codes to solve the problem I opted to go for a solution using nodes.

The second test takes the result from the Find shortest path node and then isolates any ramps present. On the top and bottom of the diagonal three additional points would be placed, one in front and two to the side. Figure 20: additional points
From these three points the points closest to the target room would be selected and used as target point for the second Find shortest path node. Finally the new points together with the ramps and target connection points would be fed into a new Find shortest path node to create a new path without unwanted routs.

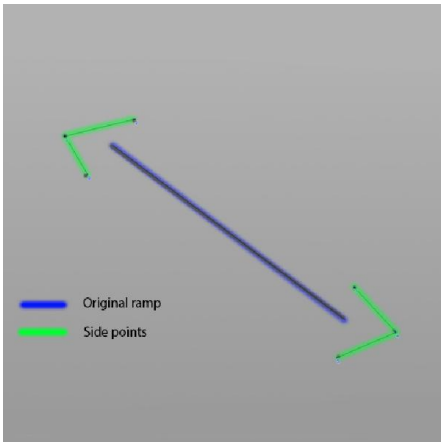


Figure 20: additional points.

This increased the complexity of the component but solved most of the problems, except for corridors that would get too close to a ramps resulting in corridors clipping with the roof or floor of a ramp

The final change was a combination of a part from the navigation mesh and the separation of the ramps from the last test. After a paths is created the used points and edges are removed from the navigation mesh which would be used again for the later iterations. When a ramp is detected it would again be isolated and used to selected specific points around it from the navigation mesh. These points are removed and the new mesh is used to create the final path. This solution removed almost all problems from the system and was less complex then earlier attempts.

At the end several small changes were made to improve performance and create a more consistent result. While working on the path finding component I gained a better understand of the node, what the possibilities are and how to use it, but also how to manipulate a navigation mesh if needed.

Dungeon mesh generator.
The final component is responsible for the generation of the meshes for the corridors and connections to the rooms.

The main modifications required here was a method to generate the meshes for the diagonal ramps. Where now square planes are used for flat corridors this same method won't not work for ramps.

The first task was to isolate the ramps and then generate the mesh. For this the same isolation method is used as in the Pathfinder component. Horizontal lines are placed on the start and end points of a ramp and using a Loft node connected. For a better fit with the corridors the ramps are reduced in width based on the size of the corridor so the top and bottom line up with the corridor polygons while keeping the width of the corridor intact. On the original top and bottom points planes are placed on to which the ramp are attached. Finally all parts of the hallways are re-connected, cleaned, extruded to the correct height and merged with the rooms.

While working some small changes needed to be made to prevent ramps and corridors from clipping on some points. To solve this an isolated ramps gets used to select intersecting corridors polygons and removed before the corridors are extruded.

The final component works 80% of the time, but due to the set time limit it had to be dropped. It did help me expand my knowledge of path finding in Houdini and point manipulation using VEX amongst others.

Known issues.

Here are the known issues that might occur when using any of the assets. Most should be fixed but are still mentioned just to be sure.

When adding thickness to the walls a corner wall might be extruded further than intended. Creating a long extended piece of wall that can reach far outside of the bounds of the dungeons.

When two rooms overlap with a difference in height the upper corners where one room transfers to the second room might display deformed polygons after excluding the walls.

Corridoro_conections might keep cooking and cause Houdini to stop responding. This happens when an incorrect number is enters in the Forloop. It's try's to connect rooms to rooms that don't exist. There are measures in place to prevent this but somehow they don't always work.

Known issues in the Unity toolset are also handled in the Unity_DG_help file located in the Unity project folder under "Help files".

Here is a short list with problems.
DG_data object can lose its content when linked to either the player or enemy instancer when one of these gets "Rebuild". To solve this unlink the main generator from the instancers and rebuild all three of them.
This also counts for other linked components in Unity.

When placing extra rooms and they don't get connectect with a corridor the navigation grid size needs to be changed.
When dungeon part show a magenta colour after aplying materials it means that either the name of the material is wrong or the material is not placed as a direct child of a folder called Resources.

Conclusion

The final consist of a Houdini tool that allows for the creation and modification of dungeons and a Houdini engine/Unity tool set that allows for almost the same functionality. The Unity tool set also has additional components that demonstrate the additional possibilities like populating the dungeon.
Most of the processes gave little problems except for a few parts. The usage of Python and VEX proved hard to debug errors as Houdini would often crash. A second major setback was failing add adding the option for multiple floors to the tool. The component was almost working but due to time constraints could not be finished. I'm happy with the end product as it stands, It is the tool I proposed to make for my socialization project, and when given more time could expand and improve it further. Finally I like to thank Kim Goossens for his supervisor Achraf Cherabi, Jeffrey Vermeer, Robert van Duursen for their feedback and help with testing.

References

Houdini-Engine. (n.d.). *Houdini engine*. Retrieved from Houdini engine: http://www.sidefx.com/index.php?option=com_content&task=view&id=2525&Itemid=66

sidefx. (2014, 05 19). *sidefx*. Retrieved from sidefx.com: http://www.sidefx.com/index.php?option=com_content&task=blogcategory&id=232&Itemid=393

sidefx_help_files. (2014, 05 21). *sidefx help files*. Retrieved from http://www.sidefx.com/docs/houdini13.0/vex/functions/neighbours

Unity3d. (2014, 06 02). *http://unity3d.com/*. Retrieved from http://unity3d.com/

Tables and Figures