PROBLEM SOLVING AND EXPLANATION OF TOOLS

THE IDEA:

The input is a curve of any type that defines the shape of the alley. This allows for a large variation of alleys and can create straight alleyways from more western cities like in the US or can create more curved alleys, even circular, like from many European cities. I also wanted to keep the buildings curve on the y axis allowing for adaptivity to more hilly terrain. It should follow the curve on the X and Z axis, however the buildings are straight so if there are only a few buildings, the curve will be less present. The generator will also randomize most settings that can be extremely tedious to modify such as Windows or roofs on a large amount of buildings. Within a few minutes, the artist will have several unique alleys.

Problems:

Problem 1: The first problem was figuring out how to create building shapes from a curve. The first idea was to copy shapes to points along the curve. I quickly ran into several problems with this solution. The issue with this was that depending on the amount of points on the curve, how straight the curve is and what the length of the curve is will very easily ruin the copy to points method. When thinking of solutions I brainstormed several ideas to try and make the copy to points slightly more dynamic and adaptive, but issues always came up.

Solution: The fixed solution that I came up with was to use the curve as the base for the geometry and build up from it, rather than trying to have it influence other geometry. When implemented properly, this allowed for very little error to occur when creating the geometry as it was created from the base curve. In order to implement this, I came up with the idea to create new points along the line that define where the building divisions are and differentiate them with an attribute from the original curve's points.

This took a small amount of vector math and VEX coding. The algorithm essentially follows every edge between each point and measures the distance to

add it up. It then divides the calculated distance by the amount of buildings to get the distance along the line where a new defining point needs to be added. It then runs through the points again (following the lines between points) and finds the spot along the line where it meets the distance requirement. In shorter words, it creates divisions in the line for the amount of buildings. (e.g. 3 buildings require 2 defining points along the line)

To add more variation I also implemented a random num to move these points randomly within a margin a back or forward. This creates more uneven divisions and thus different sized buildings

To make the buildings flat I wrote some VEX that scans all points between two defining points and places them along the vector between the two points on X and Z axis. This makes the buildings flat and rectangular, however maintains the height curve.

Finally, to add gaps between the buildings it separates each division into its own geometry and using a for loop it moves the end points of each division closer together.

Problem 2: The second problem I ran into was scaling. Since the input could be any curve - it could be extremely small or extremely large compared to real-world dimensions.

Solution: To combat this, I thought of an idea to have the buildings always scaled to real world-size. This size can be increased and decreased by the user but is always in meters and the buildings will always scale to that size non-relative to their initial size. This is done by taking the inputted value and dividing by the average length of the buildings. The number that is returned is the upscale value to get the buildings to the correct size. This was an extremely important problem to solve going forward as it allows for the details to be proportional to the average size of the buildings. (e.g a larger building should

have more windows and likely less obvious roof). It also makes it easier for the artist to export as they won't have to worry about scaling.

Problem 3: Adding details procedurally was a big learning process. I ran into several road blocks with this. In my idea, the user is able to modify each building's settings individually. I first worked on the roof, and created a for loop to iterate through each building and apply a random roof. My road block was not creating random or different roofs but how to allow the user to control that.

Solution: While looking through the Houdini Documentation I came across definitions for multi-parm folders. After reading a bit I discovered that this allows for the creation of multiple instances of a parameter. This was the answer to my problem but first I needed to figure out how to actually access an instance of a multi-parm from within a for loop. I knew how to create a direct channel reference to another parameter using ch() in HScript. When looking at different multiparm instances it seemed that each instance parameter's name included a string + an integer number representing the instance. This gave me the idea to find a string concatenation function that I could use to combine a name and the iteration of a for loop and then use that as a channel reference. After browsing the HScript documentation, I found a function called strcat() which concatenates two strings together. Using this, within a channel in the for loop, I placed a line of HScript that referenced a channel of two strings put together. It looks something like this:

```
ch(strcat( "../../nameOfParm", str(detail("../ForLoopMetadata", "iteration", 0) + 1)))
```

ch() -> is a channel reference function

strcat() -> is a string concatenation function

"../../nameOfParm" is a string representing the relative path to the parameter's name

str() -> converts to a string

detail() -> references a detail attribute from the current or another node

“../ForLoopMetadata” -> references the for loop’s metadata node

“Iteration” -> references the iteration attribute of the node which contains the current iteration of the loop (+ 1 as the multiparm starts from 1 and the for loop starts from 0)

This solution worked great to allow for the user to modify a certain iteration of the loop’s detail, however another problem occurred. (Problem 4)

Problem 4: Although the multi-parm idea worked properly, a major issue was that number of instances was not adapting dynamically to the amount of buildings. This meant that the user had to manually add the amount of instances that correlates to the amount of buildings. This raised many issues as the user could add too many, too little or easily remove an instance and lose data.

Solution: My original idea was to try and somehow use a channel reference in the multi-parm to set the instances. This did set the instances correctly, however when the instances were created/removed the nodes no longer had a proper reference to the parameters (for what I assume to be a baking reason) and thus it wasn’t properly editable. In order to solve this, I decided to use a little bit of python. I created a button that updated the instances of the multi-parm and cooked the nodes. Using parameter view-ability settings I made it so that when the button is clicked to create the details, the rest of the settings are hidden. This made it so that the artist cannot change the building amount if editing the details, preventing any linking issues between instances being added or removed accidentally.

The python function originally was a callback script. With some research into the Houdini Python API I discovered that when passing kwargs through a callback-function, it passes both the node and the parameter on which the function was called in a dictionary. Because of this, I pass kwargs through the call-back function of a button. I get the node from the dictionary and from the node, find the multi-parm parameter. Using the Houdini Python documentation I was able

to find methods that add and remove instances from a multi-parm object. I then then used hou.Parm.removeMultiParmInstance() method until there were no multi-parm instances and used hou.Parm.insertMultiParmInstance() to insert the necessary amount of multi-parm instances.

Problem 5: This issue that I ran into was also related to the multi-parm idea. I wanted to minimize the amount of parameters that the artist absolutely needed to modify in order to create a unique alley (they still CAN modify them though). In order for this to happen I needed the generator to automatically randomize different values to create different buildings and be able to easily change all of these with just one slider. Implementing this isn't hard without the multi-parm as I could have randomized attributes and used those attributes to drive the parameters, however if I wanted the user to be able to fine tune the settings individually I still needed the multi-parm.

Solution: My original idea was to use a script with the defaults for the multi-parm parameters however that didn't end up working. My second solution to this problem was to break down the steps of adding the details into 2 parts. The first part needed to calculated all of these randomized values and attributes, set the multi-parm instances, and then import all of the values into the multi-parm. Once the multi-parm is set, the details begin to actually get placed, and reference the multi-parm instances for the values to use. This means that by default, all of the multi-parm values will be set randomly, however the artist can then vary these values. When the artist varies the multi-parm values, since the only reading from these values is within the actual addition of these details, only part 2 of the section is cooked, and thus the details are cooked but the calculations and import is not cooked.

The calculation and import works by finding various random values for different settings and storing them in attributes. Once they are all calculated for a single building, a python SOP is cooked thats adds an instance and imports all the values to each parameter of that instance. The python works by creating a

dictionary with the name of the parameter for the key and the attribute's value for the value. It then loops through the dictionary and sets the parameter in the multi-parm to the value in the dictionary.

Problem 6:

My idea for the door was to have it placed somewhere on the first floor, but because of the fact that the bottom edge of the building could be curved this made it a challenge. In order to do this I had to brainstorm ideas on how to have the door be able to be moved from left to right, while adapting to the height of the curve as to not intersect with the ground.

Solution: I immediately began sketching out on paper to think about the problem. I broke down the problem into thinking about it with 2 lines. One is curved and represents the bottom edge of the building and one is straight and represents the bottom edge of the door. The straight line will always be within the domain of the curved line. The straight line must NOT have any point of the line that lies below the curved line. Breaking the problem down like this helped me think of a simple solution. Find all the points on the curved line that lie within the domain of the straight line. Also, add the start and end points of the domain as well. With all of these points, find the point that has the highest Y value (height) and that is the lowest possible height that the straight line can be without going under the curved line. Using that, I add a slight margin so it's not touching the line and now it works for no matter the placement of the straight line.

The next step was implementing it in VEX. It starts by getting the domain of the line (length on the X and Z axis) from an attribute as well as the location (between 0 and 1) that the left corner of the door should be at (imported from the multi-parm). Using these it finds the actual distance along the domain that the door should be placed: domain * location. It then loops through all the points until it find the two points in which the left corner of the door will be in-between. It locates where between those two points the left corner will be placed, and

that is the starting point of the door. It then travels forward along the domain for the remaining width of the door and records the highest possible Y value of all the points within that travelled distance (domain of straight line). Finally, it compares that with the start and end point to get the absolute highest point within the domain of the door's width. Then it adds a slight margin to that point + half the height of the door (because when copying it will go from the middle of the door, so the point needs to be moved up half of the height) and copies the door to that point.

I also made sure to constrain the possible domain of the curve to a bit less then the actual domain (essentially adding slight margins on the left and right side so that the door cannot be exactly on the edge).

I also allowed for a door Y location input which brings the door up and down from the bottom of the first floor to the top of the first floor.

Problem 7: During the window generator process I had very little problems. The only challenge came when I had to figure out the best way to export all the files so I can import them into the building generator.

Early on into creating the building assets, I realized that embedding the window generator directly into the alleyway generator would be too inefficient and so they must work alongside each other.

Realizing this, meant that I had to come up with an efficient pipeline method to export and import the custom windows.

Solution: My idea was to export the window and have the user manually import into the scene file as an input for the asset but I quickly realized this was both inefficient unnecessary for the user. I decided to use a library system, where the artist exports their files to a directory and then selects that directory in the alleyway generator. The alleyway generator will then scan and import all the assets to use for the buildings.

For the exporting from the window generator, I decided to do it via a call-back function in python. It gets different iterations of the model with higher quality export settings (LOD 0-4) and exports to a folder within the specified library. The user can also add in a name and the folder and FBX files will contain that name. The only issue I had was this was the data transfer, because the windows could have different depths and ratios. I needed a way to be able to get that data from the FBX file so I could import them properly. I figured the best way to do this was to use JSON. My python script grabs the needed attributes and stores them in a dictionary. Using json.dumps() the script puts them into a json object. Then it write the json object to a json file within the specified window directory. This could also help in the future, in case more features are added to allow for an easy export of attributes or data from one asset to another.

Problem 8: Another challenge I had, was with the importing of the windows into the alleyway generator. I wanted to be able to scan the library and import all the level of detail models while copying the json data into attributes.

My first implementation was poor as I decided to create and destroy nodes in order to add and remove the files and all of the data for the files. Although this is not a bad workflow for a batch import, it doesn't work for a digital asset, as the asset will be locked and thus nodes and parameters cannot be created nor destroyed. There was a possible a workaround that I brainstormed however that unlocks and locks the asset with python, modifying the nodes however I quickly realized that this was a dangerous workflow as this is not how the assets were meant to be used.

My second implementation was much better. Instead of creating nodes, I used a single node in a for loop that references the file path for the window. This file path changes for each iteration. This means that a different window is imported, however no nodes are created or destroyed.

In order to do this properly I had to use a fair bit of python. Within the file node there is a python script running to concatenate a multi-parm value with the directory file path and the selected level of detail in order to get the correct model.

I decided to create a hidden parameter (not in the multiparm) that is set to a menu. When I was looking at the menu user interface I realized that you can use a scripted menu. Within this script I call a function that I define in my python module and pass the directory into it. This function scans the directory, places all of the window names within the directory into a list. It sorts it, and returns the list. In order to make sure I was returning the correct list format I had to look at the Houdini documentation for scripted menus. In the documentation they say you can use a list but it will take the first item of the list as the menu value and the second value as the label, almost like a python dictionary. Because of this I had to put a number association before each label in the list before returning the list. —- (e.g. list = [‘0’, ‘name’, ‘1’, ‘name1’, ‘2’, ‘name2’)

The menu updates if you change the file path, add something to the directory or remove something from the directory.

Within the calculation portion of the asset, for the windows, I randomly select a number from the menu, get the label, and with the label and directory, retrieve the JSON metadata. I then am able to access the metadata in order to perform calculations for the window amounts.

In order to access the metadata later, I don’t want to have to re-read the JSON file since I already read it in the calculations. So, I created invisible parameters that I call container parameters, that are only there to store data, not to be edited by the user. This data is then accessed when the windows are actually added.

The only editable parameter related to the window file reading is a dropdown menu that can change the window. This dropdown menu runs the same script

as the invisible python one but is within a multi-parm and thus is not accessible when there are no multi-parm instances which is why I use an invisible menu that is not within the multi-parm. The multi-parm menu however contains a python call-back script. The script takes the currently set label, finds the folder where the label is stored, oads the json file, reads the data, and sets the container parameters. This is repeated in this script in case the user changes the window type. If the user does this, the container data needs to change as well.

Problem 9:

Another major problem I had was about the placement of windows on the buildings. Because the window size never changes but the building size does, there needs to be a limit for the amount of window columns that you could set on a building. The hurdle was trying to make this dynamic. If the building is quite large there should be a large number of window columns available to toggle ON/OFF, however if the building is much smaller, it should not allow you to create the same amount of window columns.

Solution: My solution to this problem required python again. Python is a very powerful tool for technical artists as I clearly found myself using it a lot.

Because the columns are an embedded multi-parm, I had a integer slider that controlled the amount of instances for the column. To clamp the columns to the max amount of columns possible, was actually done within the creation of the details. There is a python SOP that scans the width of the building and width of the selected window and checks if the number that the user changed the slider too is greater than the number it calculated. If it is, it clamps that parameter down, making it impossible to go over that limit.

IMPROVEMENTS:

I think there is a lot that I can improve on with this tool

Firstly, I didn't get as far as I wanted to take this visually, however these tools could be easily reworked or updated in order for this to happen.

The first issue is that when the depth of buildings is increased and there is a curve in the buildings, they will potentially intersect into each other. This is because the buildings extrude out by their own normals and don't take into account the normals of the building next to it. My idea for a solution to this problem would be to create some VEX that checks the direction of the extrusion normal of each division point. For example, it would check the left point of one building with the right point of the building next to it. It would compare the extrusion direction * the magnitude that is set via the depth parameter. It would use vector math to figure out the shortest distance between the two line segments. Using this it can tell whether the lines intersect or are closer than the specified gap size. If this is the case they need to be spread apart. In order to do that, it must find the line of each buildings edge, in vector form, and move the direction of the extrusion vectors along those lines until the gap is equal to the specified gap. It would then need to check each building to make sure it didn't intersect with any extrusion vectors on it's own building. Lastly it would have to clamp the distance of the building in case the extrusion vectors are moved to the point where the left side of the building intersects with the right side meaning that the building cannot possible be built.

The second improvement is quite minor, but currently the UVS are not scaled properly. The UVS are unwrapped nicely, however the UVS for the roof and the UVS for the bevels are not proportionate to the UVS of the Floors. This can be easily solved using a UV layout node and laying out each item on an individual UDIM TILE. The UDIM workflow would also allow for higher quality textures as each island could fill up much more space. I would layout each floor UV randomly on the UV without rotation (to vary the texture that is applied)

Similar to the UV improvement would be the addition of procedural materials. Using Houdini's shader network I could create a brick texture, that is effected by

some tightly packed noise and connect it to displacement. I could then set the color of the bricks and grooves and add some dirt with color, noise and ambient occlusion. Done right, this texture could be seamless and then moving around UVS would not be a problem and could create a lot of variation. This is just one example of a procedural shader that could be created. I would also like to build a concrete one and a wood one for the windows, doors, roofs and divisions. These would be shown in a separate tab in the UI once the artist is satisfied with the details of the buildings, and using a multi-parm each individual building could modify a specific shader. On export, the generator would bake the textures for each building and then when imported into other software, they could be applied.

Another improvement could be the addition of multiple paths or a 3d landscape mesh essentially becoming a city generator on a very large scale. It would take in a height map and use that to drive the height scale in certain areas. Paths would be drawn along the map for which the asset would adapt to, and a height map would be painted. In addition to a height map, other maps could be imported for specific details. For example, an artist could select different assets and create a template. Then when the map is imported, a certain colour on the map could represent the occurrence of that specific asset template, like skyscrapers or townhouses allowing for different sections of styles within the map.

Asset Addition: Balconies, road and sidewalks, roof generator, door generator, varying styles of buildings (not just rectangular), asset placement on the sides and back of buildings.

Scattering tool + more asset generators like lamp posts, trash cans, trash bags,

The window generator could be improved with more styles and better UVs.