Procedural 3D Environments

By: Devin Doane, Gregory Langenhorst, Jackson Rygel

Artificial Intelligence in Gaming - GMAP 368-001

Prof. Stefan Rank

12/10/19

Introduction

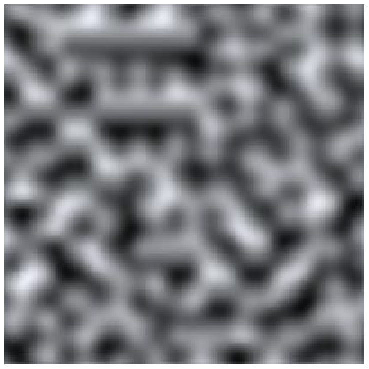
This paper is an in-depth explanation of a procedural level/terrain project created by a team of three (names are above) for an AI in Gaming class. The project was created over a span of three weeks. If you are interested in having the files for the project, please contact one of the team members using their emails on the last page of this paper. The goal of the project was to create a tool in Unity that would allow for randomly generated terrain to be randomly populated with modular assets. The tool would allow the user to manipulate multiple variables in order to create a randomly generated level tailored to their liking.

The project was split into three separate parts, one for each of the team members to work on. The first section of the project was landmass generation using Perlin noise. This section was taken on by Greg Langenhorst. The second section was a point selection system in order to randomly place objects onto the terrain. This section was headed by Jackson Rygel. Finally, the third section of the project was modular asset generation. This section was created by Devin Doane. Once all of the pieces of the project were created, we then combined all of the work into one Unity scene for our final build. A basic level where a first-person controller was dropped into a level can be found at: https://greglangenhorst.itch.io/procedural-first-person-prototype.

Landmass Generation (Greg)

The initial terrain generation is the first part of the project that runs and was the first part of the project that was created. After deciding on procedural level generation the team split up the work, and I was tasked with terrain generation. I searched around and landed on an interesting concept: Perlin noise map interpretation. Sebastian Lague's Procedural Landmass Generation is a great jumping off point for understanding the concept and using it in Unity ("Procedural Landmass Generation" video series, Sebastian Lague, youtube.com/watch?v=4RpVBYW1r5M).

So what is Perlin noise? It is a type of gradient noise created by Ken Perlin to make computer graphics seem more natural. Each pixel in Perlin noise is based on the pixel next to it, creating a black and white image that looks somewhat like smoke. ("Understanding Perlin Noise", https://flafla2.github.io/2014/08/09/perlinnoise.html)



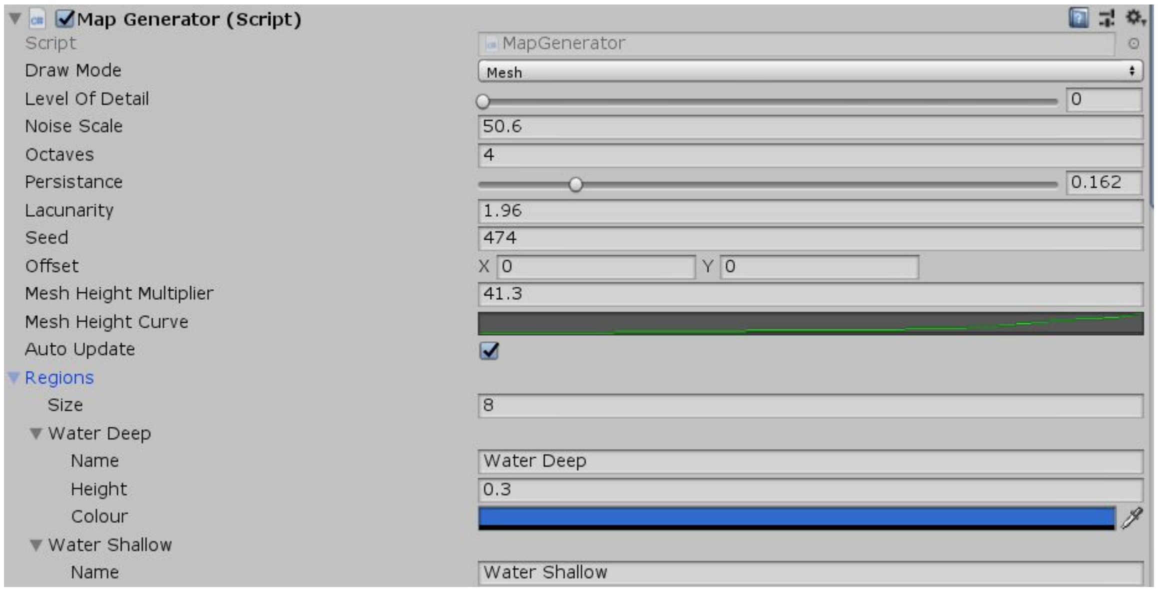
Perlin noise has a "natural" flow to it that can be interpreted to look like terrain. That terrain map can then be converted into a mesh based on variable parameters. The seed value is a value from -100,000 to 100,000 that randomizes the noise map. This seed value is passed to the other parts of the generation tool so that each instance is the same when given the same seed. Other values help choose which part of the noise maps are interpreted. The x and y offset change which part of the noise is sampled for interpretation, and the noise scale "zooms" in and out of the noise.

The values above change which part of the Perlin noise is sampled, but do not fundamentally change the noise in any way. In order to interpret Perlin noise in interesting ways that a game designer might wish to use for terrain maps, we need octaves. Octaves are a value that increases the “detail” of the noise, incorporating noise interpreted to have higher frequency and/or amplitude. As the octave integer increases, the frequency of the noise is increased by the set lacunarity value, and the amplitude is increased by the set persistance value. If a designer wishes to have simple hills and valleys without much variation, they may set the octave low and the lacunarity and persistance high. They may do the opposite if they want a more rugged terrain.

All of the changes to the noise are converted by a script into a mesh, which increases in height with the increase in the noise’s lighter values, and decreases with the darker values. This mesh is a plane that is stretched on generation to fit the noise’s data, and thus appears as a three-dimensional version of the interpreted noise. The material of the mesh is created, mesh triangle by mesh triangle, to match set color values that can be set in the Editor based on the height of the triangle. The result is a plane textured as if it has water, beaches, and mountains.

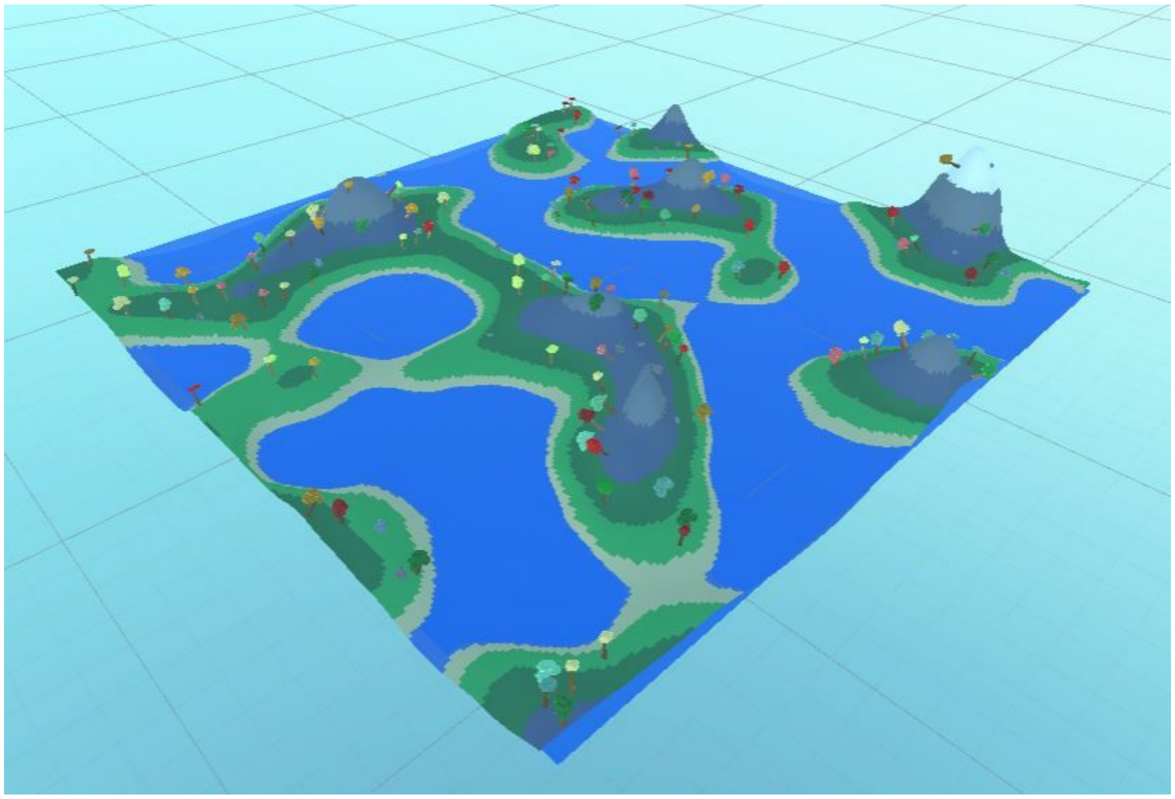
The final way the noise is turned into the terrain mesh is through manipulating it’s height. A mesh height multiplier factors into how high the mesh stretches upwards, and an Animation curve can be edited to affect how steep the slopes around the mesh are.

Everything above can be changed in the Editor. If a designer were to pick up and use our tool, they would not have to touch any of the scripts other than setting up a few Game Objects. Regions can have their colors changed within the Editor, seeds can be set, and the specific slopes a designer wants at certain points can be fine-tuned with the mesh height Animation curve.



✓ Map Generator (Script)	
Script	MapGenerator
Draw Mode	Mesh
Level Of Detail	0
Noise Scale	50.6
Octaves	4
Persistance	0.162
Lacunarity	1.96
Seed	474
Offset	X 0 Y 0
Mesh Height Multiplier	41.3
Mesh Height Curve	
Auto Update	✓
Regions	
Size	8
Water Deep	
Name	Water Deep
Height	0.3
Colour	🟦
Water Shallow	
Name	Water Shallow

Some planned parts of the terrain generation had to be cut for deadline reasons. One of those parts is chunk loading, the process by which several terrain maps are hidden or rendered based on player position. This is combined with level-of-detail switching (where the terrain meshes have more or less polygons based on proximity to the player) to create larger terrains that run smoothly on any PC. The complexity behind writing asset locations and keeping track of larger terrains was outside the scope of the project, but the basic of some of these concepts can be found online. These concepts can take what we created and evolve it into a larger generated world on par with games like Minecraft and even No Man's Sky ("Chunk Loading", Technical Minecraft Wikia, https://technical-minecraft.fandom.com/wiki/Chunk_Loading).



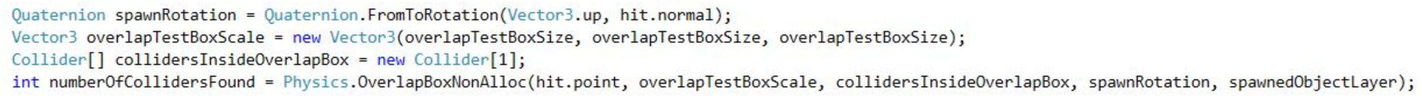
Point Selection (Jackson)

My part of the tool was to bridge the gap between the terrain mesh and the modular asset system. I needed a way to instantiate the assets on the mesh in a manner that felt normal i.e. correctly placed on the mesh. The assets needed to not intersect with one another and not instantiate in certain places such as on water. Trees tend to not grow on the top of a lake. While the initial idea was to use poisson disc sampling to find points, that was eventually scrapped due to time constraints. What I came up with is still fully functional and looks great. Some of the ideas I used came from Matt Mirrorfish's Procedural Generation series on Youtube, but changed to work with our changing terrain and assets ("Procedural Generation For Beginners: Randomize Object Placement", Matt MirrorFish, youtube.com/watch?v=tyS7WKf_dtk&list=PLuldlT8dkudoNONqbt8GDmMkoFbXfsv9m&index=2).

The first part of my point selection process is a script that chooses points randomly on a plane. The x and y values of the plane can be changed in the Unity Editor, but the basic idea is that the plane is above the terrain mesh. A number of points that can also be changed in the Editor are randomized based on the plane's area, and a GameObject is instantiated on each of those points. The points are simply randomized with Random.Range for the x and y values.

Each of the GameObjects created by the first script have the second script of the point selection process on them. This second script shoots raycasts down at the mesh. When these raycasts hit the terrain mesh, they return the rotation and position they hit the mesh at. At this point, trees and rocks could be instantiated, but they could still accidentally intersect one another. There is one more step in the process: boxcasts.

A GameObject is once again instantiated on the rotations and positions found by the previous mechanisms. These GameObjects have box colliders on them, and they shoot boxcasts out a certain distance to see if they intersect or not. If a boxcast finds at least one other box collider, the instance deletes itself. The end result is GameObjects that do not instantiate a certain distance from one another ("Procedural Generation Basics: Prevent Overlap Spawns With Boxcasts In C# [Unity Tutorial]", Matt MirrorFish, youtube.com/watch?v=lFmKZskl45I&list=PLuldlT8dkudoNONqbt8GDmMkoFbXfsv9m&index=5).

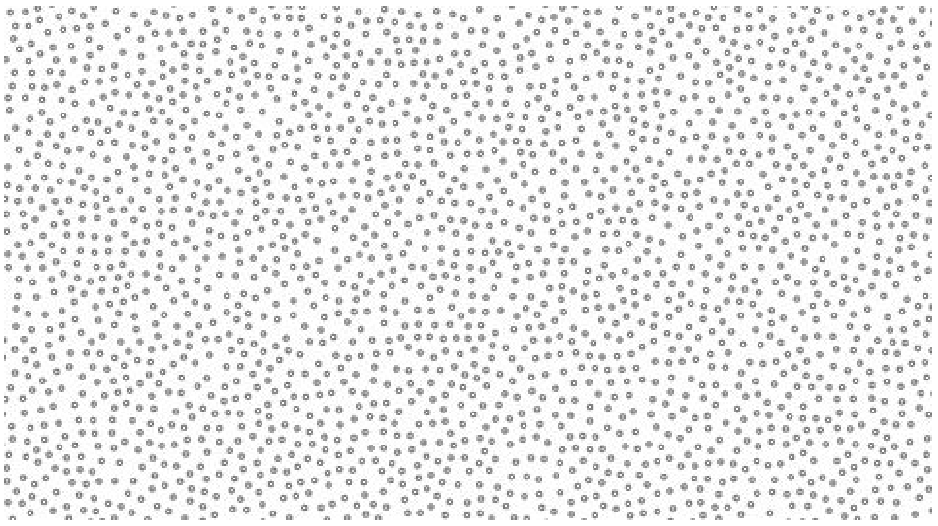


```
Quaternion spawnRotation = Quaternion.FromToRotation(Vector3.up, hit.normal);
Vector3 overlapTestBoxScale = new Vector3(overlapTestBoxSize, overlapTestBoxSize, overlapTestBoxSize);
Collider[] collidersInsideOverlapBox = new Collider[1];
int numberOfCollidersFound = Physics.OverlapBoxNonAlloc(hit.point, overlapTestBoxScale, collidersInsideOverlapBox, spawnRotation, spawnedObjectLayer);
```

These final GameObjects now call Devin's asset creation script, which populates the scene. The way the point selection works out is not optimized, but it certainly works. A lot had to be dropped from this step for the sake of time. One of the initial ideas was to interpret the Perlin noise and not have to deal with raycasts at all, but that turned out to be rather complex, and is not a common way to do random placement.

Another part of the point selection that was cut was checking to see if certain parameters are met for instantiation. Instead of writing the code to check if a tree is on water we had to clamp the raycast's length to be less than the length from the instancing plane to the water plane. A better way to do this would be to either check the RaycastHit's distance or check exactly which collider the raycast hit.

The largest element we had to omit for the sake of time was poisson disc sampling. This type of random point selection on a plane creates a more "natural" look to the randomization, something the entire project is chasing. Poisson disc sampling is a type of 2D random point selection where points have a set radius, or disc, that other points will not instantiate in. We ended up doing an odd sort of "inverted cube sampling", but poisson disc would have been preferred ("Poisson Disc Sampling", Jason Davies, www.jasondavies.com/poisson-disc/).


An example of poisson-disc sampling

While the end result of the point selection part of this project was not what we initially had in mind, what we came up with still works well and combines with the other parts of the project quite smoothly.

Modular Asset Generation (Devin)

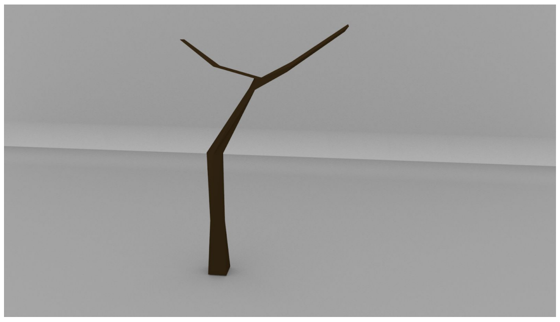
This section will go into further detail about the modular asset generation which was done by me (Devin). I started the research process by looking into a few different techniques of asset generation. These techniques were “L-systems” (Lindenmayer Systems), the process in which the developers of *No Man’s Sky* created their randomized alien lifeforms (un-named), and finally a technique that uses an authoring agent in order to randomly combine pre-made modular assets together.

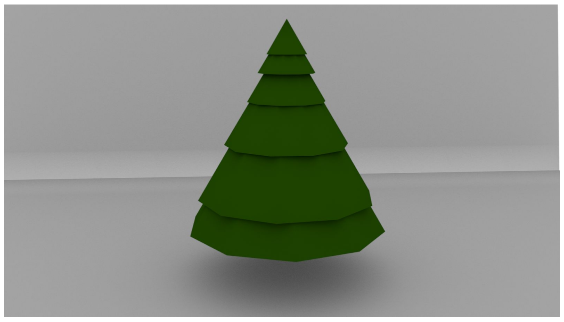
While looking at the L-system technique of random asset generation I came across a paper on the topic written by Paul Bourke (“L-System Manual”, http://paulbourke.net/fractals/lsys/). This paper helped me understand what exactly L-systems are, and how in depth this technique of asset generation is. L-systems are algorithms that are created in code by using an alphabet of symbols in order to create strings. These strings simulate the real-life process that trees, and plant life go through while growing their branches, and the branches off of those branches, etc. It is a very in-depth process, but it creates realistically grown organic assets like trees. After a few days of looking into this process, I realized that this technique would be too in depth, and require more time than the three weeks that I had would allow. So, I decided not to use this technique for `our project. However, if you are interested in these systems further, and would like a more visual representation of what L-systems can actually do in Unity, this video does just that, (“Procedural Plant Generation in Unity”, Kristin Stock, youtube.com/watch?v=wo3VGyF5z8Q&t=24s).

The second modular asset generation technique that I researched was the process in which the *No Man's Sky* developers used. I was not able to find too much information on this technique, however, I did come across this article on the topic ("Procedural Generation", Fandom, nomanssky.fandom.com/wiki/Procedural_generation). This article was really interesting to read about the general process the developers went through, however obviously they do not actually share their code, and/or specific details about the process. Due to the lack of information for this process of modular asset generation, I decided not to pursue this process for our project.

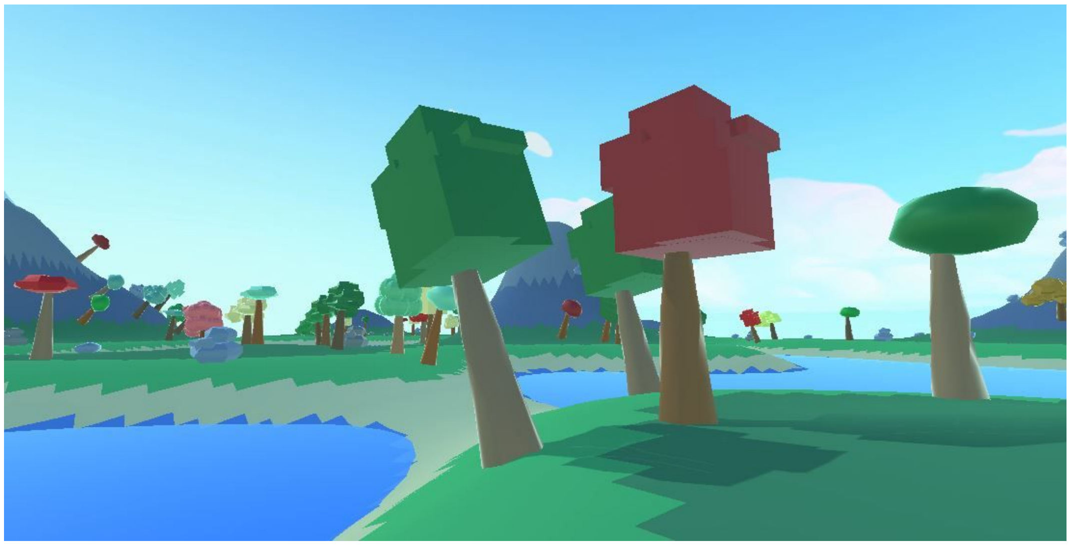
The final modular asset generation technique that I looked at was the authoring agent in order to combine pre-made assets. This is the process that I decided to do in order to create our randomly generated trees, and rocks. I mainly chose this process because there were multiple online tutorials that made the process easy to understand, and I would be able to use methods that I have used before in order to complete this technique. It also seemed the most feasible for a three-week project. The two tutorials that I followed throughout the creation process were, ("Scriptable Objects in Unity", Brackeys, youtube.com/watch?v=aPXvoWVabPY&feature=emb_logo), and ("Random Seed Tutorial", Kurt Hollowell, youtube.com/watch?v=eggw0fUn6Zk&feature=emb_logo).

I started the creation process by modeling multiple, modular assets that would work well when combined together. I modeled in Autodesk Maya*,* and created three tree trunks, six leaf types, 3 rock types, and finally a lily-pad. While modeling I made sure that the models were simple, not just in order to fit our project's aesthetic, but also, in order to make sure that each asset piece would fit together with any other asset piece without looking unnatural. The assets were relatively low-poly in order to make sure that there would be no performance issues in Unity when generating from dozens to hundreds of them into the scene.

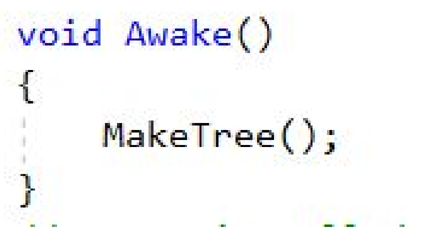




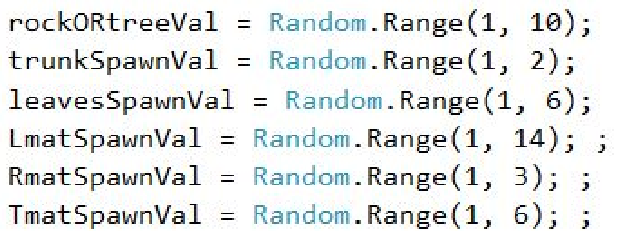
After the models were completed, I exported them as FBX files in order to import them into Unity.



Once all of the models were imported into Unity, I made each separate piece into its own prefab. I then wrote a script in C# that assigned a random number to each of my asset prefabs (right figure). After that, I made another script that on "awake" of the scene, would generate a new prefab based on either one of the randomly selected asset numbers, or a combination of two of the randomly selected numbers (left figure).

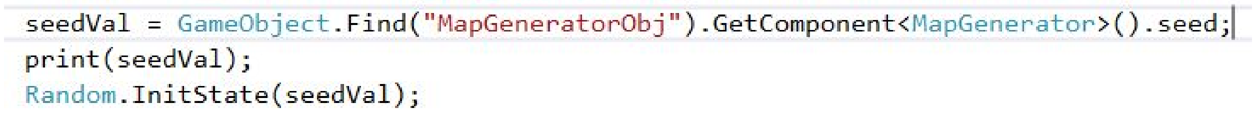


```
void Awake()
{
    MakeTree();
}
```



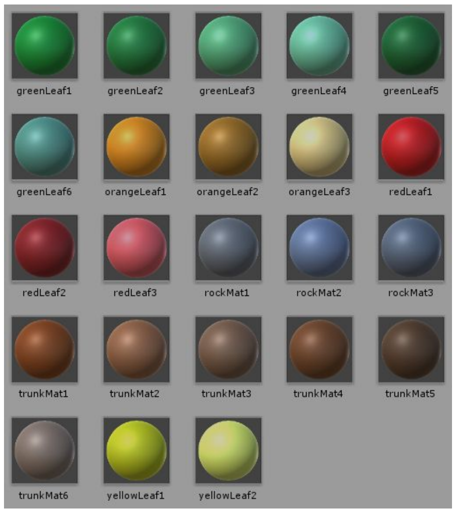
```
rockORtreeVal = Random.Range(1, 10);
trunkSpawnVal = Random.Range(1, 2);
leavesSpawnVal = Random.Range(1, 6);
LmatSpawnVal = Random.Range(1, 14); ;
RmatSpawnVal = Random.Range(1, 3); ;
TmatSpawnVal = Random.Range(1, 6); ;
```

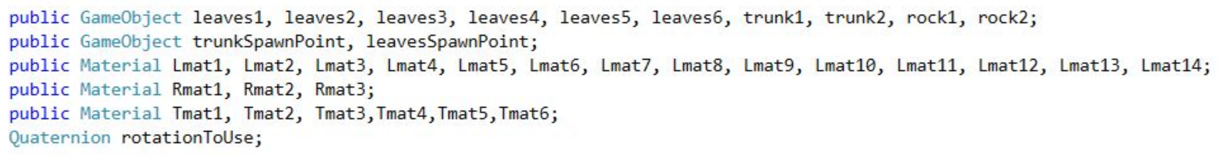
With the modular assets being combined together into their own placeable prefabs, I then wrote a script that took Greg's seed value for the scene, and baked 'Random.Range', and thus my generated objects to that seed. This would allow the user to change seeds, but still have the specific objects, and placement of those objects the same on each seed (code seen below).



```
seedVal = GameObject.Find("MapGeneratorObj").GetComponent<MapGenerator>().seed;
print(seedVal);
Random.InitState(seedVal);
```

For example, the layout of my objects would be the same for seed value "10" no matter what, even if the seed was changed. Finally, with the new prefab created, my modular assets were ready to be instantiated into the scene through Jackson's point selection tool. After all of that, I created multiple color materials for each asset piece (ex: greens, browns, etc.), and then created a script that would randomly assign one of those materials to the assets when generated, again, connected to the seed value.





```
public GameObject leaves1, leaves2, leaves3, leaves4, leaves5, leaves6, trunk1, trunk2, rock1, rock2;
public GameObject trunkSpawnPoint, leavesSpawnPoint;
public Material Lmat1, Lmat2, Lmat3, Lmat4, Lmat5, Lmat6, Lmat7, Lmat8, Lmat9, Lmat10, Lmat11, Lmat12, Lmat13, Lmat14;
public Material Rmat1, Rmat2, Rmat3;
public Material Tmat1, Tmat2, Tmat3,Tmat4,Tmat5,Tmat6;
Quaternion rotationToUse;
```

The only issue that I ran into was the fact that my lily-pad model had to be cut from the project. This was due to the way Jackson's point selection placement of my objects was working. We could not get just lily-pads to spawn on the water without anything else, so we just decided to not spawn anything on the water as a temporary fix.

In Conclusion

Overall the team is proud of the final product that we were able to come to in such a short amount of time. Everything works well together, and each section was separate enough that we could work separately, but still have the same final product. We hope that you have learned something just as we have. If you have any questions about the project, wish to view the project, or would otherwise like to contact the team please feel free (contacts below).

Devin Doane: dmd439@drexel.edu, or ddoane.com

Greg Langenhorst: ghl27@drexel.edu, or greglangenhorst.com

Jackson Rygel: jrr337@drexel.edu, or artstation.com/jacksonrygielgeneralist

Works Cited

1. “Understanding Perlin Noise”, https://flafla2.github.io/2014/08/09/perlinnoise.html
2. “Procedural Landmass Generation” video series, Sebastian Lague, youtube.com/watch?v=4RpVBYW1r5M
3. “Chunk Loading”, Technical Minecraft Wikia, https://technical-minecraft.fandom.com/wiki/Chunk_Loading
4. “Procedural Generation For Beginners: Randomize Object Placement”, Matt MirrorFish, youtube.com/watch?v=tyS7WKf_dtk&list=PLuldlT8dkudoNONqbt8GDmMkoFbXfsv9m&index=2
5. “Procedural Generation Basics: Prevent Overlap Spawns With Boxcasts In C# [Unity Tutorial]”, Matt MirrorFish, youtube.com/watch?v=lFmKZskl45I&list=PLuldlT8dkudoNONqbt8GDmMkoFbXfsv9m&index=5
6. “Poisson Disc Sampling”, Jason Davies, https://www.jasondavies.com/poisson-disc/
7. L-System Manual, Paul Bourke, http://paulbourke.net/fractals/lsys/
8. Procedural Plant Generation in Unity, Kristin Stock, youtube.com/watch?v=wo3VGyF5z8Q&t=24s
9. Procedural Generation, Fandom, nomanssky.fandom.com/wiki/Procedural_generation
10. Scriptable Object in Unity, Brackeys, youtube.com/watch?v=aPXvoWVabPY&feature=emb_logo

11. Random Seed Tutorial, Kurt Hollowell,

    youtube.com/watch?v=eggw0fUn6Zk&feature=emb_logo