PROCEDURAL TERRAIN
Jesse Levine and Brian Tam
Spring 2015
Abstract
We produce procedurally generated terrain consisting of geometry and textures. Geometry is created using the GPU at run-time. The textures are precomputed using the CPU and saved to disk for later access. A shader applies textures without the use of UVs to the geometry to create complex and interesting environments that are interactive in real-time.

1 Introduction
Unity3D is a flexible and powerful game engine that is widely used in the industry. It is used to create AAA titles, indie games, art experiences, educational programs, and much more for virtually all platforms. However, one big area of weakness is its native terrain tools. Unity's terrain tools only allow height map terrains with manually painted textures. This makes it impossible to do cliffs, bridges, or overhangs without funky hacks such as layering multiple terrains or creating the features in an outside 3D modeling program. However, these hacks are difficult to edit and usually generate other issues. Another key problem with Unity’s terrain tools is that if terrain features change, then a lot of texture work would have to be manually redone.

Our motivation for this project is to create fast and complex terrain geometry and textures without the tedious work of UV mapping. We also want to give the artist plenty of control in creating the textures and have it easy to make changes without losing previous work.

1.1 Related Work

[Freeman 2007] uses noise to generate height map terrain. Showcases early GPU blending and lighting. Terrain is generated cpu side.

[Andersson 2007] uses procedural parameters to generate masks to seamlessly blend different materials. They allow for real-time interactive mask generation based on player actions. They discuss about techniques to speed up the rendering by combining shader passes and using LOD.

[Geiss 2008] uses marching cubes to generate height density complex terrain. Uses the geometry shader to do most of the implementation on the gpu.

2. Procedural Geometry

2.1 Overview and Features
The geometry is generated at run time on the gpu. The geometry generation can be understood as a three part process:

1. Sample noise values per voxel point in a N x N x N coordinate space
2. Calculate surface normal
3. Run marching cubes using the voxel noise (density) values

The geometry generation is rather fast and a 64 x 64 x 64 chunk can be generated in under 20 milliseconds on modern graphics hardware. The current geometry generation implementation exclusively generates 64 x 64 x 64 chunks, however, it would be a trivial extension to increase this size. In short, the “Procedural Geometry” generated data contains information on vertices and their normal values.

The main technology utilized is Unity’s compute shader. The compute shader is a wrapper for DirectX 11’s Direct Compute Shader, therefore it only works on modern Windows systems. The power of the compute shader is that it allows for easy general purpose computing without shader hacks or having to move to OpenCL or CUDA.

2.2 Using noise to generate terrain
Using noise to generate terrain has been common practice in graphics. Perlin / Simplex noise are ideal for terrain generation due to their organic composition. Since noise generation is a task that lends itself to parallelization, it is speedy to generate on the gpu. However, this is a non-trivial task. Thankfully, there exist multiple open source implementations for gpu noise, most are based on Ken Perlin’s improved noise work. The generated geometry leverages both 2D and 3D noise functions.

The 3D noise is calculated based on world space and stored in a 2d image array for easy and rapid access. Thanks to open source work this process is entirely
handled by provided code and is black boxed in relation to the rest of the geometry. The noise is then sampled and used to assign a density value between -1 and 1 for all 8 of each voxel's corners. This data is stored as an array that can be passed to multiple compute shaders.

At this point a chunk of geometry can be created. However, the geometry here would not look much like terrain, but rather a cube with holes. In order to create a terrain that has a surface level with slopes and hills we need to filter our noise function. This is done using a secondary GPU based 2D noise function. The 2D noise is evaluated per voxel to determine how to cut the surface of the terrain out. Essentially, the 2D noise functions as a height map to clip the 3D noise. Finally, to make this process simple for testing a GUI control is added to change values on the noise.

2.3 Generating Normals
Because the voxel density process creates a buffer filled with noise values, we can interpolate these for smooth normals. Essentially all you need to do for normal is to take an approximation of the derivative of the noise function. Sampling three points around the desired point is generally good enough for smooth normals. After calculating, renormalizing the value is a must. One bug occurred when sampling the top of the voxels. The issue occurred because the voxel corners that are directly below voxels cropped by the 2D noise still sample the 3D noise for normals. This was fixed by creating a flag value in the 3D noise buffer that identified these surface corners. Next, in normal calculation the 2D noise was calculated at run time and used for the normal of these edge cases. Unity provides the capability to render a structured buffer to a texture. This can be helpful for sampling the normal during the actual geometry generation in marching cubes.

2.4 Marching Cubes on the GPU
Marching cubes on the GPU works almost identically to the traditional implementation of marching cubes. Paul Bourke provides a public domain implementation of marching cubes in C++. Since the C++ is not using many libraries it is straightforward to port over to the C# style syntax of Unity’s Compute shader. Only two significant changes are needed in the port. One, the precomputed arrays need to be stored in structured buffers. Thankfully Unity handles the heavy lifting with its library. Passing any array is as simple as using a set buffer call in C#. The second change is the compute shaders are general and do not enforce any organization for geometry information. This can be solved by using structs and structured buffers of type struct. For example, the vertex information is stored in a structured buffer of type vert as defined by a struct with a homogenous float4 position and a float3 normal. As long as each compute shader is explicitly told how to handle the data in the structured buffer this works. To get the end result back into the normal pipeline a CG shader is used. The shader is instructed how to parse out the vertex information to pass on.

2.5 Challenges
The geometry generation challenges were mostly around debugging and hacking compute shader code. There was a plethora of algorithmic resources available. Most information on the theory was available, the difficulty was in porting code to the compute shader. The compute shader writes nicer than traditional shaders, however, there is still no debugging. The performance is rewarding, especially for naturally parallel algorithms like marching cubes. One problem with marching cubes and noise is the floating island effect. This is when a piece of geometry appears to be floating and is disconnected from the rest of the geometry. This can mostly be mitigated by using lower detailed noise.

3 Procedural Textures
3.1 Overview & Features
The procedural texture system was created using a node-based system with artist control in mind. The artist has almost full control over the system to create the effects he or she envisions. The system also allows an easy method for the artist to take a texture at any node, edit it in an image editing program, and reintegrate it back into the node flow.

The procedural textures are generated through com-
Combining different Procedural Texture nodes in Unity’s Component system. There are two types of texture nodes, generators and adjustments. For this project, generators include Noise, Voronoi, Lines, Bricks, Plaid, and Pencils. Adjustments include Blend, Blur, GradientMap, HueSaturationBrightness, Invert, Levels, MakeItTile, RandomFlood, and ReplaceColor.

### 3.2 A Node-Based System

#### 3.2.1 Simple Nodes

**Base class** - The base ProceduralTexture class node defines variables and methods that are common in all derived nodes. This includes the texture file, resolution, pixel data, generating, previewing, refreshing, and saving to disk.

The textures are generated using the CPU and C# as it allowed for an easier access to individual pixel data and method of saving the resulting images to disk. **Extending the class** - The subclasses are split up into two categories: generators and adjustments. Generators do not take in a texture input and uses various different methods such as lines and noise to create output texture data. Adjustments take in one or more texture inputs and apply algorithms to change the pixel data for their output textures.

**Example of Generator**

![Randomly generated red-blue noise. Intermediate values are interpolated on a HSB scale.](image)

**Pseudocode**

```
Noise
Inputs: FromColor, ToColor, Resolution
Output: Texture of size Resolution that contains Noise
```

For every pixel:

1. Calculate Noise value using a seeded Random value
2. Lerp FromColor to ToColor based on Noise value using HSB (hue, saturation, brightness) scale
3. Set this pixel to lerped color
4. Apply pixel data to Texture

#### Example of Adjustment

**RandomFlood**

This algorithm replaces random areas of one color with another by flooding.

Inputs: OldTexture, OldColor, NewColor, NumberOfSamples
Output: Texture

```
While NumberOfTries < NumberOfSamples * Constant && NumberOfSuccessfulFloods < NumberOfSamples
Get a random point in the OldTexture
If this point’s color matches OldColor
FloodFillArea with NewColor
```

![More examples of generators. From left to right, bricks, plaid, and colored pencils.](image)

**Figure 4:** More examples of generators. From left to right, bricks, plaid, and colored pencils.

**Example of Adjustment**

![Comparison of the texture before and after RandomFlood replacing 79 samples of white for red.](image)

**Figure 5:** Comparison of the texture before and after RandomFlood replacing 79 samples of white for red.

**Pseudocode**

```
RandomFlood
This algorithm replaces random areas of one color with another by flooding.

Inputs: OldTexture, OldColor, NewColor, NumberOfSamples
Output: Texture

While NumberOfTries < NumberOfSamples * Constant && NumberOfSuccessfulFloods < NumberOfSamples
Get a random point in the OldTexture
If this point’s color matches OldColor
FloodFillArea with NewColor
```
Apply pixel data to Texture

Notes:
The algorithm randomly picks points anywhere on the texture and can have trouble finding enough sample points. A constant multiplier with the number of desired samples determines how many total points to try. A constant of 1.5 is used in all examples. The FloodFillArea code is from [Bunny83].

3.2 Combining Nodes

Figure 6: Example outputs of the step-by-step nodes to create the rocky texture. First, a voronoi pattern is generated. Next, random flooding chooses colors from a gradient to fill in some rocks. Color replacement is used to fill in the remaining rocks and change the color of the space between rocks. Finally, a noise texture is blended to create the final output.

Nodes can be combined to as inputs to other nodes. Through the use of multiple generators and adjustments, layered and complex textures are created. In Figure 6, the step-by-step node sequence is detailed to create a rocky surface. Other textures resulting from combining nodes can be seen in Figure 7.

Figure 7: Dirt, grass, and snow textures used on the generated terrain.

3.2.3 Speed
Textures are generated on average within a few hundred milliseconds. More complex nodes such as Blur with a high radius and high number of iterations can take up to a few seconds. As textures can be generated before they are used for rendering in Unity, there was not a need for speed optimization.

4 Tri-Planar Shader
4.1 Overview & Features
The shader is written in Cg shader language with vertex and fragment components.

Pseudocode

Vertex shader:
For every vertex:
  Calculate world space position
  Calculate normal
  Calculate screen space position

Fragment shader:
  // Material Blending
  For each axis x, y, z:
    For each map at this axis:
      Sample texture map, using world space position as UV coordinates
      Calculate weight based on mesh features
      Blend current texture with previous texture based on weight
      Blend each axis texture based on normal direction
  // Lighting
  Get light direction from directional light
  Diffuse Reflection = Multiply light color with diffuse color based on fragment normal and light direction
  Ambient Lighting = Get ambient light from Unity
  Return result of Diffuse Reflection + Ambient Lighting

Terrain-Based Features
The shader uses other mesh data in addition to mesh normals to determine where to place textures. World position can be used for elevation to determine whether a point is near the sea level, tree line, or snowy mountain peaks. The slope of a point, based on the y-component of the normal, can determine areas where sand or gravel might settle.
Figure 8: The tri-planar shader applied to the Stanford bunny with dirt used for all three axes. An additional grass texture is applied from the y-axis depending on the slope at each point. The maximum slope for the grass is 0.7.

Customization
The shader in its current iteration supports textures for rock, dirt, grass, flowers, and snow and parameters for blending among them. Additional texture maps and parameters are easily added within the shader code.

4.2 Algorithm Details
Getting Data from the Generated Terrain
The shader takes in a uniform of type StructuredBuffer<Vert> which holds the data from the GPU-generated terrain. It uses each element of the buffer to calculate the world position, normal, and screen position of each vertex.

The Three Planes
The tri-planar shader 'projects' textures onto a surface from three orthogonal planes: x, y, and z. Because of this, the geometry does not need UVs to determine texture mapping, but rather the world space position and normal is used. The world space position dictates which part of the tiling texture map is applied to a specific point while the normal dictates which plane or blend of planes should be projected to that point.

Figure 10: The tri-planar shader applied to a sphere. The normals on the sphere dictate which texture map to use for each point.

Masking
Masks are used to further decide which texture map to use on a terrain within the same axis plane.

Figure 11: On the left, the tri-planar shader applied to the Stanford bunny. Rocks are used for the x- and z-axes. Grass, dirt, and snow are used for the y-axis depending on elevation and slope. On the right, the mask used for blending dirt onto the bunny.

Lighting
The lighting is simple Lambertian shading using ambient light and a directional light as light sources per
fragment.
The ambient contribution multiples the ambient light with the texture component.
The directional light contribution multiplies the light’s color, the texture component, and the dot product between the fragment normal and the light direction. The ambient and directional light contributions are added together to get the final fragment color.

4.3 Bug & Fix
Screen-Space Shader
In an early iteration of the shader, the world position of a vertex was not explicitly defined. Rather, the shader used a precomputed ‘position’ variable to determine the UV coordinates. This position variable was actually the result of multiplying the ModelViewProjection matrix with the vertex’s local position, which is the vertex’s position in screen-space.

When using this value for the UV coordinates, the textures were applied screen-space: they were stuck as part of the screen. If the camera or model moved, rotated, or zoomed, then the textures would stay fixed relative to the screen. An example of this is found here. http://i.imgur.com/9BnrohI.gifv

5 Results
Textures are precomputed and stored. Geometry is generated per request and computes a 64 x 64 x 64 voxel chunk in approximately 20 milliseconds. Once generated, the terrain runs at 60 FPS. The rig used runs an Intel i7 2720QM @ 2.20 GHz x 4 with 8 GB of RAM @ 1333 MHz containing an NVIDIA 485M with 2GB of VRAM. See included images.

6 Limitations:
The largest limitation of the geometry is in the simplicity. The geometry only stores vertex and normal information, limiting the scope of texturing. Additionally, the geometry generates one chunk at a time. The current implementation does not support seamless addition of more chunks. The geometry is also voxel resolution limited, creating some non-organic polygonal when running marching cubes. The textures are precomputed and do not correct line width, and aliasing/stretching problems. Overall, the biggest limitation of this project is how procedural geometry is rendered by Unity. Unity renders procedural geometry outside its normal pipeline. This means that Unity mesh features will not work on the resulting terrain (SSAO for example). Instead all features must be implemented as a shader pass.

7 Future work
7.1 Procedural Geometry
There are a few possible extensions for the procedural geometry of varying complexity. Different lighting situations such as Ambient Occlusion could be added into both the geometry and the shader to create effects such as flowers not growing in areas of low light. LOD and the generation of multiple chunks could be used to create bigger maps for the user to explore. Subdivision on the GPU can be used to solve the cubiness issue resulting from the marching cubes algorithm. The geometry data could be ported from the GPU into Unity’s native Mesh data to allow for collision detection.

7.2 Procedural Textures
There are two major extensions for the procedural textures that are very apparent. First, the system can be ported into a visual graph-based system to make the process of creating and editing textures easier. This system would use a drag & drop interface to connect inputs and outputs of nodes. Second, the system currently only generates a color map. Other maps, such as normal, specular, metallic, and roughness would give the artist the ability to add more detail to his or her work.

8 Conclusions
Overall, the feasibility of procedural content is there and continues to show strength. Procedural technique may one day replace modelers for many aspects of reality. Moving forward, adding a special data-structure for seamless geometry would add interactivity. Additionally textures and geometry could leverage benefits from more information on geometry and displacement.

9 Division of Labor
Jesse Levine worked on geometry. Brian Tam worked on texturing. Both worked on deliverables and combining work.
10 References


Moderate Height Terrain

Grass and Mud globs
Smooth Canyon Walls

Stepped Terraces