Procedural Mesh Generation of Organic Forms in Unity

Christopher Eicher, Jason Murray

# 1 Introduction

Procedural content generation is pervasive throughout games with vast, unending worlds to explore. Procedural mesh generation creates models. One example of PMG is terrain generation—terrain can be a large subdivided plane with a procedurally generated heightmap to control the height of the vertices in the plane.

Often procedural content generation is used to place objects in the world, constructing elements of the world like children do with toy bricks. Modern 3D games incorporate the idea of adding decals, the idea of stacking a texture onto something that already has a texture on it, and this is a very powerful tool because it lets you break up a surface that may have a repetitive texture on it. We aim to give you the ability to add additional shape on top of the shapes of your world using procedural mesh generation—to give you the tools to add extra touches of detail to your world.

## 1.1 Procedural vs. Random

Even in game development circles there is confusion about the difference between procedural generation and random generation. Randomly generated content isn’t completely random—there are rules in place—so all random content is procedural content. Procedural content isn’t necessarily different every time. It doesn’t always need a pseudo-random number generator, but even if it does, careful preservation of the seed means that you can generate the same world consistently with less data.

Fuel is an open world racing game released in 2009 on the Xbox 360, PlayStation 3 and PC, that used procedural generation to let players wander around 14,000 square kilometers of an open world that was not random but the same every time. The game was held entirely on a single disk, a feat not possible without procedural content generation.

## 1.2 Procedural Mesh Generation in Shipped Titles

Even games that don’t boast procedurally generated content may have procedurally generated meshes. One of the most common examples is terrain. Many games use terrain that is saved on disk as a grayscale texture, but ultimately the terrain must be rendered as triangles in 3D space. The process of taking a heightmap with a relatively small memory footprint and generating a terrain mesh is a simple form procedural mesh generation.

The character creation system in Bethesda games is another example of procedural mesh generation. At the beginning of the game the player is given several tabs of sliders that are used to change the shape and color of the details that make up the player avatar’s head. This is a more advanced version of mesh generation. On top of having custom made avatar faces, players can also equip helmets, which means the mesh generation system has to be capable of removing any hair, spikes, and cat ears that might be poking out of the top of the avatar’s head.

No discussion of procedural generation is complete without bringing up games like No Man’s Sky and Spore. These games procedurally generate entire planets and the creatures that inhabit them. Both games populate their world’s terrain with a mix of premade and generated assets. No Man’s Sky pieces together parts and skins to make their creatures. They “sew” these parts together in a way that allows them to reasonably animate and render the creatures without obvious seams.

Spore appears to use a skeleton system that allows you to scale and the bend the creature's body and then add arms and legs wherever you want. Then you can shape and scale the arms and legs however you want, and finally add details to the surface of the creature. This is particularly impressive because Spore will then animate your creature and most of the time it does so in a pretty reasonable way.

# 2 Transformations Review

Before we jump into the nitty gritty of our brand of mesh generation we’ll present a cursory review of transformations in 3D space. Position, Scale, and Rotation are the major elements that go into describing a mesh’s orientation and location in a game world.

Each of these elements can be represented in the form of a matrix that transforms a mesh’s “model space” representation into “world space”. Order is important, and in most 3D game pipelines they are applied in the following manner:

modelWorldTransformMat =

modelLocalTransformMat

\* positionMat \* rotationMat \* scaleMat

Transformation matrices are typically 4x4 to account for the fact that vectors shouldn’t be transformed in the same manner as points. To this end it’s important to construct transformed Vectors with w=0 and Points with w=1. It makes everything easier if you use vec4s to store these.

## 2.1 Rotation Considerations

As we’ll see later, this mesh generation technique relies on orienting rings and planes of vertices in space and connecting them together. Finding equations for rotation of an object in space over a period of time is a painful exercise most have never gone through. Instead of solving that problem once and for all, we resorted to finding equations for “facing direction” over time and generated sufficient rotation matrices using Unity’s Quaternion.LookRotation()function.

## 2.2 Successive Linear Transformations

Suppose you’re generating one mesh and wish to place other meshes or objects at a certain position and orientation relative to a location on your generated mesh. This can be done! Linear transformations can stack successively to transform a model from its local space to some position and orientation relative to another arbitrary space.

In the Vine example below vines are recursively generated by passing down transformation matrices that describe the base position of the vine:

secondaryVineWorldMat =

primaryVineSurfaceMat \* secondaryVineLocalMat

In this example the local secondary vine transformation is run through a transformation to place it on the surface of an earlier vine.

# 3 The Primitives – Planes and Bands

## 3.1 Conventions

All meshes we generate are built out of either bands or subdivided planes. Mesh generation requires finding a set of points to lay faces across. We create **Planes** that are subdivided into quads by generating positions of points via a function parameterized on a t value to give curvature or other effects. We separately create **Bands** (like wedding bands) by generating successive rings of vertices and connecting the rings with faces. An overview of how to do this is given in the following sections.

In code samples and equations, we make assumptions so that our descriptions are more concise. We assume the variable t in the context of a mesh generation for-loop is equal to i/(float)numLoopIterations. The variable tc is equal to 1-t, or “t-complement”. Both cover the range [0,1]. t increases linearly and tc decreases linearly.

At the lowest level of abstraction, we have helper functions for generating the rings and planes described in this section. Each helper function takes a Matrix4x4 as their first parameter. All rings and lattices are built in the X-Y plane and then transformed by this matrix back into model space. It’s easier to reason about the size of these primitive elements in terms of this local space rather than trying to figure out directions, positions, and lengths directly in model space. This is especially true as you start creating the finer details of a model. Using matrices to transform from local space to model space is further explored in section 4.

In Unity the elements of a mesh are straightforward and what one might expect. A mesh is composed of Points, Faces, and UVs. Points are stored in a Vector3 array. Faces are represented by an int array. The length of the face array is three times the number of faces and the elements in this array are indices into the array vertices. UVs are stored in an array of Vector2s with the same length as the Vertices array—they map one-to-one.

In our project we attach materials to the object ahead of time so that the code we write can assume that the mesh material is what we expect it to be. We now move on to the generation of basic planes and bands.

## 3.2 Generating a Plane

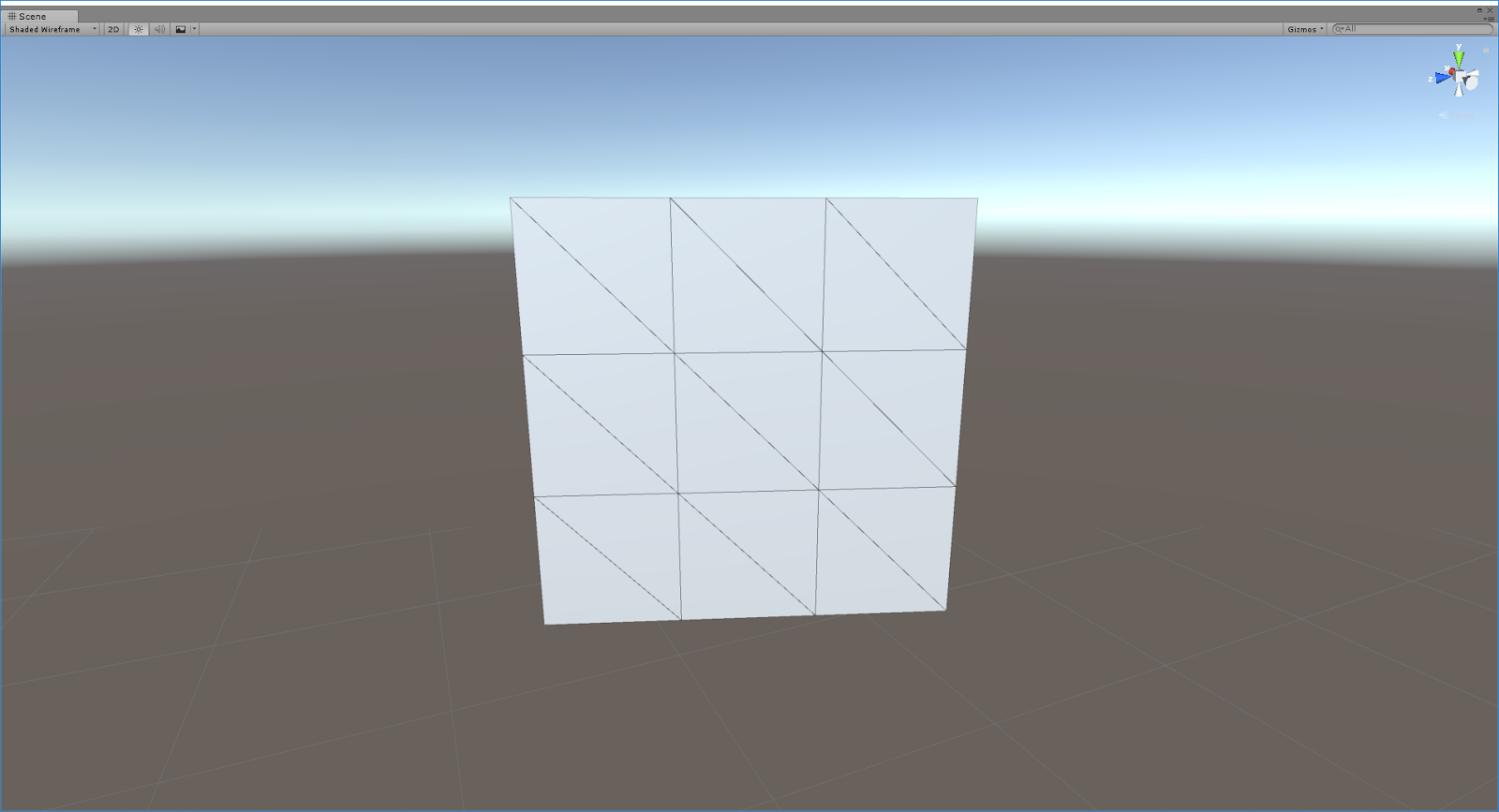


Figure 1 – A generated Plane

There are many ways to generate a plane, so it’s easy to see having several versions of this function rather than a catch all function that does everything. When making leaves and flowers we need to be able to rotate these planes around the center of their bottom edge, so we make lattices that span the Y-Axis from 0 to the height of the plane, and the X-Axis from -width / 2 to width / 2. But it’s easy to conceive of why you might want your plane to be rotated around its center, so you could build it to span the Y-Axis from -height / 2 to height / 2.

At the core of the function for plane generation is a double nested for-loop. The outer loop makes rows of points, and the inner loop makes the points the individual row. After each inner loop (except the first), we call another helper function called MakeStrip().

MakeStrip() takes rows of vertices and makes triangles out of them. It takes 4 parameters. The first two parameters are the indices of the first vertex in each row, and the third parameter is the number of vertices in each row.

MakeStrip() loops through the vertices and takes pairs of triangles that make up single quads. It does this by making a face out of 2 vertices from the first row and 1 from the second, then 1 from the first row and 2 from the second.

For a row of n vertices, the function will make n-1 quads. If you call MakeStrip() and the faces normals turn out pointing in the wrong direction, don’t change the function, go to where you called the function and swap the first and second parameters, the face normals will flip the other way.

## 3.3 Making a Band

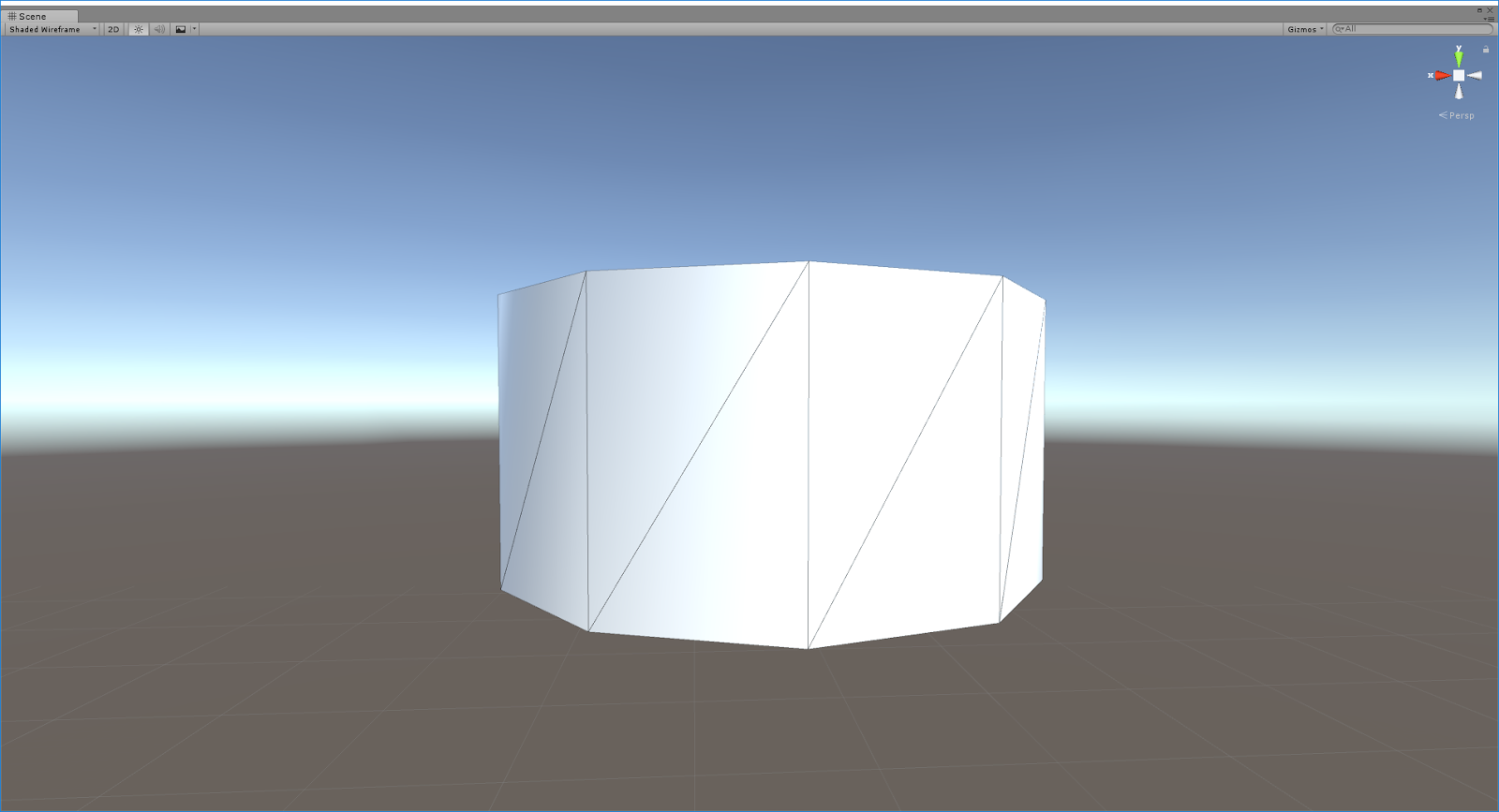


Figure 2 – A generated Band

We used two helper functions for making bands. The MakeRing() function takes a Matrix4x4 and transforms the rings it generates into model space. The rings are generated via a simple for-loop and for each iteration we take x = r\*sin(t\*2\*PI) and y = r\*cos(t\*2\*PI) to get positions in the X-Y plane.

The MakeBand() function works exactly like the MakeStrip() function except when you have n vertices you make n quads instead of n-1, and since you want to connect the last vertices in the ring the first vertices in the rings you need to modulo the index offset by the number of vertices in the ring.

The band function is good for making n-sided prisms, which done by calling the function with n-number of subdivisions. If you need a triangular prism you can tell it to use 3 subdivisions when making rings for your bands.

## 3.4 Making Fans and Radial Fans

Fans and radial fans are great for capping the ends of your 3D shapes. For a sphere, cone, or cylinder we start by generating the bands that make up the body of the shape. Then we manually add the tip vertex and call MakeRadialFan(). This function takes the index of a single vertex and the index that starts a ring of vertices and it makes triangles between successive pairs of points and the tip vertex. For the ends of triangular prisms and rectangular prisms it may look unusual—not that the player would notice, just you—to have an extra vertex in the cap of the prism, so just call MakeFan(), and pass it the first vert in the ring as the pivot vertex and the index of the second ring vertex for your second parameter.

You may notice that radial fans are all you need to make a cone, but if the cone is long you should make the cone out of several bands, as vertex based lighting will look better with more vertices.

# 4 Building Something More Complex

We’ve described the mechanics of building the smaller components involved in generating meshes. Now we move on to fleshing out more detailed shapes.

## 4.1 Local Spaces

As we take these primitives and piece them together we introduce the idea of local spaces. When discussing how we might start building something from scratch, we think, “what is the basic shape of this thing?” Once we determine where to start, we can begin to think in finer levels of detail of the mesh, and every time we go into a finer level of detail, it will be easier to think about the size, position, and orientation of additional primitives—not in model space, but relative to the surface of the 3D primitive we were just working in.

When making a tree, the base of the tree would be the trunk—a cylinder. When making a space ship it could be the fuselage. We start by building the base of the thing we are making, then we add details to it—branches on a tree or wings on spaceship.

We make a matrix that performs a transformation from the local space of the surface of that object into model space and then call functions that take that matrix and use it to transform a local space primitive into model space. To make this transformation we first think, “where do we want the origin of this primitive to sit at?” Create a translation matrix to place it somewhere in your current local space. Then concatenate a rotation matrix that defines where the up of the new local space will be (in terms of the current local space). If you are coming out of the side of a cylinder, find a vector that represents the surface normal of the cylinder at that point and use a Unity’s Quaternion.LookAt function to generate the rotation matrix. If we also want to think in a different scale, this is the point at which we would come up with a new local scale matrix. When already working in a local space that is not model space we also need to make sure to apply the existing current LocalToModel matrix. Building the final local-to-model matrix might look something like:

Matrix4x4 newLocalToModel =

localToModel \* translation \* rotation \* scale;

Passing this matrix to your function that adds finer details makes it much easier to think about where things are in 3D space. This is analogous to thinking from the perspective of an ant walking up the trunk of a tree. A nail sticking straight out the tree doesn’t appear to stick out at 46 degrees in the X-Y plane, it simply points up from the perspective of the ant.

## 4.2 Parameterized Curves

The vine example shown later uses a combination of a helix shape and various other generated shapes placed elsewhere along the helix. Helices are a specific example of a more general set of shapes describable by parameterized curves; functions of a single variable that produce curves in 3D space:

simple\_helix(t) = <r\*cos(t), r\*sin(t), a\*t>

In our implementation we abstracted the ability to generate a mesh based on a parameterized curve. An algorithm was written to take position information from a parameterized curve and place connected rings according to the curve. This works fine for placing the rings but we also need a way to orient them properly. We can take the derivative of the parameterized position to produce a function that produces parameterized direction:

simple\_helix’(t) = <-r\*sin(t), r\*cos(t), a>

But sometimes we have a position function that isn’t easily differentiable. Other times we’re lazy. In either case, we can estimate the direction of a ring either by linearly subtracting the current ring’s position from the previous ring’s position, or by using more complex/precise discrete derivative computation methods.

With parameterized position and direction we can essentially generate any tube we desire. That’s neat! But we can do better. We can also provide a parameterized function for the radius of the mesh!

simple\_helix\_curve\_radius(t) = c\*(1-t)

Parameterizing the radius gives us significantly more power in defining the shape of our mesh, as opposed to only the discrete position and orientation. For example, the thorns on the vine below are generated using this method with a radius that decreases over time.

One of the great benefits of this approach is that you can put extra constants and terms into your parameterized function to create interesting effects and deformations. You can then expose these constants for artists and designers to tweak!

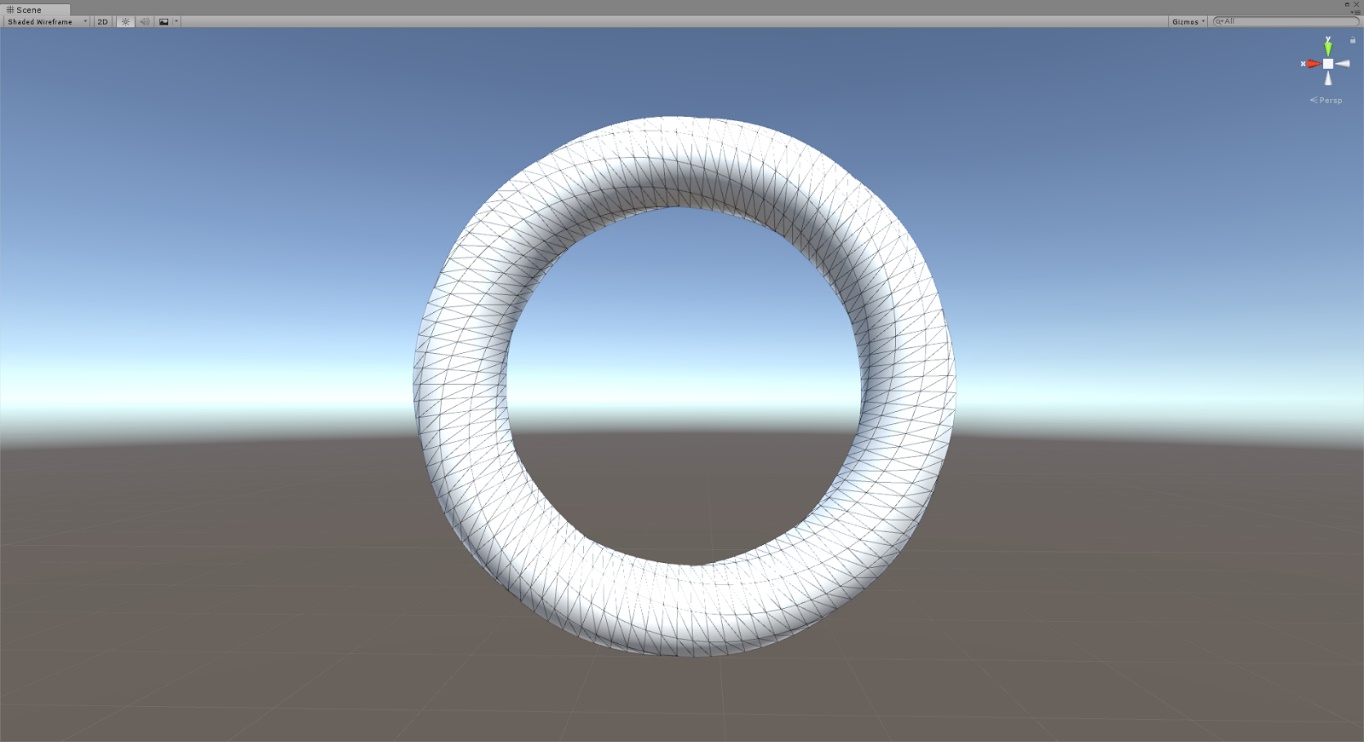


Figure 3 – A Torus generated using the parameterized curve technique

## 4.3 The Vine



Figure 4 – A vine generated using a variety of techniques described in this paper

In the case of a vine, we started with a simple helix curve. The helix intersects with the ground at a shallow angle instead of sticking straight up out of the ground. To make the curve move straight up we initially multiplied the Z and X components by t—this makes the rings point straight up initially and causes the helix to look like it’s wrapping around the surface of an upside-down cone. We later add more terms multiplicatively to the X and Z functions. These functions are arbitrary and fine-tuned to give the shape we want—they make the helix look less “upside-cone shaped” by making the radius increase non-linearly. The functions themselves don’t matter except to that we did have to factor changes in these functions into changes in their derivatives to ensure accurate ring orientation vectors.

To spawn details on the surface of the vine we would call a properly seeded random for each ring of the helix and then again for each vertex in the ring. The inner most loop (for each vertex) would have a (t \* 2 \* pi ) value you that we could plug into sine and cosine to get the up position in local space for the origin of our new local space. As an added bonus, this vector was also the up-direction of the local space.

## 4.4 Adding Non-Mesh Details

Originally more detail was added to the vine by calling the MakeCone()function or calling the vine function again to create a recursive vine. Later we used separate GameObjects for wicked curved thorns and some neat flowers. These were not directly a part of this GameObject’s mesh, so there was one last step needed to position these correctly. The objects have their own distinct transformation that represents their position and orientation in world space, so we had to transform them by our own LocalToWorld matrix. This would apply any translation, rotation, or scale applied to our model, placing the object right where we expect to be. If you forget this step you will know because all of your spawned objects will be sitting around the level’s origin instead of the model where they’re expected to be.

# 5 Conclusion

The methods we covered are a set of fundamental tools that can be used to make your world look much more natural and varied.

We covered bending curves through 3D space and generating geometry around them. We covered construction of cylinders and other primitives and how to extrude them through space. We covered thinking in successively deeper local spaces and how this aids in making more detailed geometry.

Further improvements--wandering primitives across surfaces to create detailed vines. Detailed exhaust pipes can add varied style to steampunk environments, even steampunk creatures. Deformed and detailed generated cones can add demonic asymmetric horns to creatures from below. Architecture can be generated for entire civilizations using meta-parameters that give your cities the appearance of a shared culture.

Having explored the math and strategies to work at the lowest levels of mesh generation, generating the extra detail to give your worlds a more handcrafted look is a matter of having the right eye.

# 9 References

[Suridge 13]. 2013. Modelling by numbers: Part One A. Blog Post, Gamasutra. <https://www.gamasutra.com/blogs/JayelindaSuridge/20130903/199457/Modelling_by_numbers_Part_One_A.php> (accessed October 26, 2017).