Professional and Descriptive Article Title (benis)

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 terrain must be rendered as triangles in 3D space. The process of taking a heightmap with a relatively small memory footprint and generating a 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 procedural mesh generation. On top of having custom made avatar faces, they can also equip helmets, which means they must be able to remove any hair, cat ears, and spikes that might be poking out of the top of the avatar’s head.

We can’t talk about procedural generation without talking about games like No Man’s Sky and Spore. These games generate entire planets and the creatures that inhabit them. Both 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 must be able to “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:

recursiveVineWorldPosMat =

prevVineSurfaceTransMat \* recursiveVineLocalMat

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

# 3 The Primitives – Planes and Bands

# 4 Building Something More Complex

## 4.n-1 Small Steps (Simple Shapes)

## 4.n 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!

## 4.n+1 The Vine

# 9 References

## 9.6 Online Documents:

[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).