Mathematical Mesh Generation

Jón Steinn Elíasson[*]
Reykjavík University

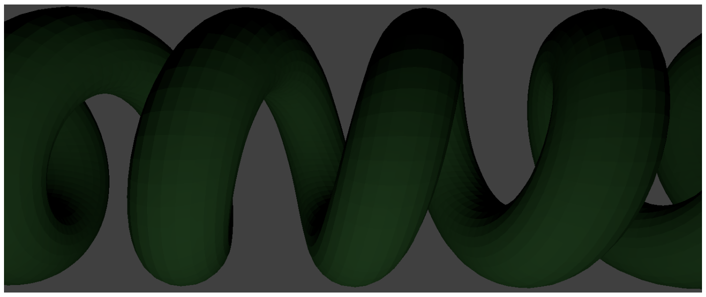

Figure 1: *A spiral mesh, rendered in Blender*

Abstract

With mathematical functions we can map two dimensional planes into three dimensional surfaces to create some beautiful geometries. I created a web application, using Flask and WebGL, to create, preview and export the resulting three dimensional surfaces of such mappings. Thereupon they can be imported to any system supporting the format for further use.

Keywords: Mesh, Functions

1 Motivation

Prior year's particle exporters sparked the idea of some kind of export system but I did want to create something different. I thought it would be a fun project to create objects with mathematics. It would allow countless possibilities for the user to create various things.

The most obvious use for such a system is to visualize mathematical functions where we can roam freely and explore the result of the mapping. Many game engines offer predefined primitive objects like spheres and we can add further objects to that, like toruses, cones or pyramids. It can also be used to generate terrain like surfaces.

2 Related work

Many of the computer algebra systems out there support 3D plots such as Wolfram Mathematica and SageMath. They are usually somewhat limited when exploring these objects and often only allow rotation and zooming and you mostly feel like you have a 3D object on a paper rather than being in 3D space with the object. I have not found one that support mesh exporting since that is usually not what they are intended for.

3 Approach

3.1 User interface

The interface is written in javascript and html. It offers the user three ways of creating meshes using various parameters, view what he has created and download it. I originally intended to create the project in Unity but did not find a suitable way to convert input into functions. Creating a web application was mostly just a means to an end and as little effort as possible was placed into that part of the project.

3.2 Preview rendering

I used a WebGL library called Three.js to render a preview of any object created. The scene is potentially kept raw and only includes the object it self, which is loaded from the `.obj` file created, a camera and lights. There is an ambient light and a point light that follows the camera. To explore this primitive world, there are key events to translate and rotate the camera.

3.3 Calculations

There are two types of mappings available to generate meshes. The method of generating a mesh works mostly the same for all mappings.

- A mapping from $\mathbb{R}^2 \rightarrow \mathbb{R}^3$ where 2D points of a grid in the xz-plane are mapped to height point. We can either map with a defined function or using noise.
- A mapping from $\mathbb{R}^2 \rightarrow \mathbb{R}^3$ where 2D points of a grid (uv-plane, unrelated to our space) are mapped to a 3D point.

Here we will go through the process of generating a mesh using a function from $\mathbb{R}^2$ to $\mathbb{R}^3$. We begin by parsing the input according to our supported syntax and convert it to a lambda function which is used to map our 2D points to a height point

If the user asks for a $n_x \times n_z$ grid with $x_a \leq x \leq x_b$ and $z_a \leq z \leq z_b$, then each cell is of size $\Delta x \cdot \Delta z$ where $\Delta x = \frac{x_b - x_a}{n_x}$ and $\Delta z = \frac{z_b - z_a}{n_z}$. We take each cell in the grid and find the height for all of its vertices.

Each cell is then split into two triangles, (031) and (023) as shown in figure 3. For the triangle (031) we use the cross product of the vectors (31) and (30) as normal and the cross product of (02) and (03) for (023).

All of these results are written to an `.obj` file along with texture coordinates which are by default, just a scaling of our domain into the texture coordinate plane.

[*]e-mail:jonsteinn@gmail.com

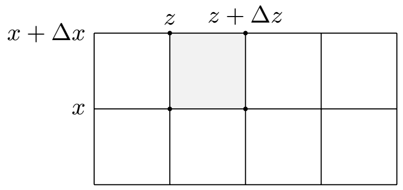

Figure 2: *A single cell in a grid*

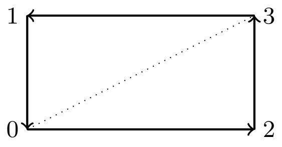

Figure 3: *Vertices, triangles and vectors*

The only difference for the noise mesh is that the height map is a random noise function and for the parametric surfaces, we have three functions and must map all three coordinates of each point.

3.4 Texture coordinates

Since we are just generating a file and not directly using a rendering pipeline (other than the preview one), there are limited options for textures. I spent quite some time experimenting with texture coordinates to deal with the difference of triangle areas. By generating enough triangles you shouldn't see any stretched texture on the object but that isn't a good way to avoid the problem since the detailed meshes can be expensive.

Near the end of the project, I got the idea to calculate the length of each line (both horizontal and vertical) in the grid after it had been mapped to a 3D surface and use them to get better texture coordinates. Two such grid lines can be seen highlighted in the grid in figure 4. For each point (n, k) on the grid, we take the texture

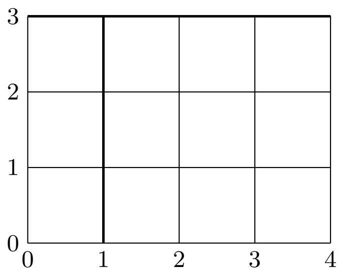

Figure 4: *Grid lines*

coordinate of $(n-1, k)$ and add to it

$$\frac{\text{dist}((n, f(n,k), k), (n-1, f(n-1,k), k))}{\text{length of } z = k \text{ after being mapped}}$$

which gives us the u coordinate for (n, k). The v coordinate is obtained similarly. An example texture map of this procedure can be seen in figure 5.

4 Results

All the maps work as intended but they are generated based on input so there are of course ways to get an incorrect visualization of a function, for example by using a small grid (in terms of cells) for a rapidly changing function on a large intervals. Figure 6 shows an example from the preview scene.

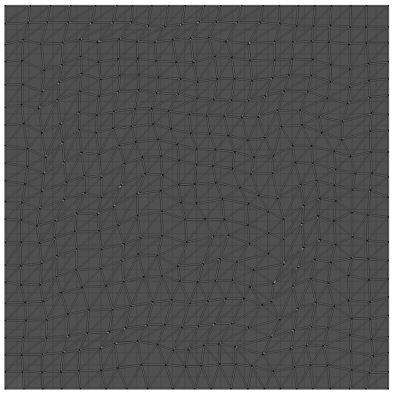

Figure 5: *Length based texture coordinates*

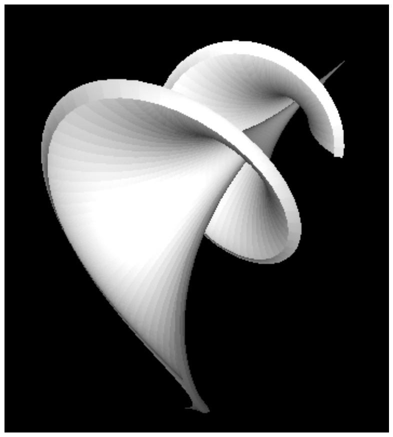

Figure 6: *Dini's surface*

We can download the mesh in Waterfront's `.obj` file format and import it to Blender or anything that supports the it. To use it in Unity for example, the file needs to be converted to a `.fbx` format and there are numerous converters available.

The length based texture coordinates can bee seen in figure 7. They avoid triangles with stretched textures and play out quite nicely.

There is however a flaw in this method that can occur. In texture coordinates, each cell can end up being mapped to a concave polygon rather than a convex which means that the triangles we split the cell into could potentially overlap. This is shown in figure 8.

An approach that might fix this is to split the cell from (1) to (2) when this occurs. We can guarantee that (1) and (0) are to the left of (3) and (2) respectively, and that (1) and (3) are above (0) and (2). I think each cell should always be able to pick either a split from (03) or (12) that does not go outside the cell but that would make the file generating a lot more complicated and is left for future work.

5 Future work

The code needs a lot of clean up, both performance and style wise and if this were to be a web application, there are some additional security issues that need to be handled.

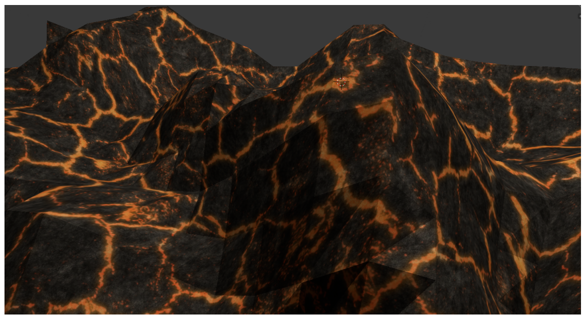

Figure 7: *Length based texture mapping*

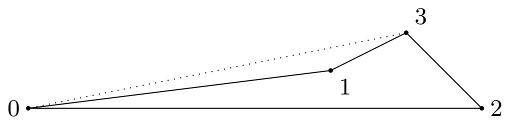

Figure 8: *Flaw in length based texture coordinates*

Having some shaders available for the WebGL preview would be a nice touch or even support uploading them.

The length based coordinates are a nice idea and would work if we were drawing rectangles rather than triangles. Is this approach possible with triangles? That is the next step of this project.