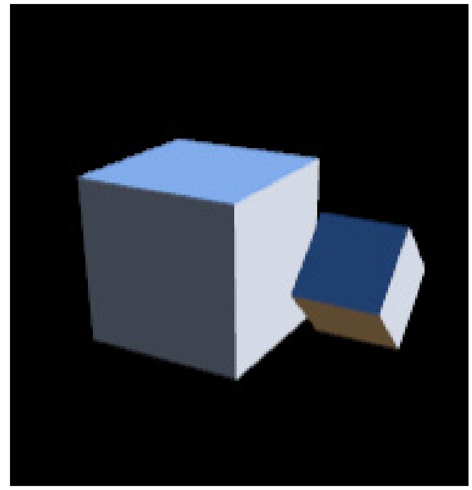
Kiwi Geometry Operations

Functional

Can pass functions as arguments of other functions. Here, we pass the makeCube() function as an argument to the "rotate" and "scale" functions.

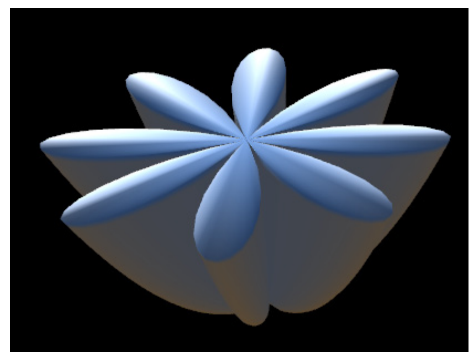
```
function r(obj)
  rotate PI/4 0 0
    obj()
function s(obj)
  scale 2 2 2
    obj()

function makeCube()
  cube 1

r(makeCube)
translate 2 0 0
  s(makeCube)
```



Lathing

I implemented an algorithm for generating parametric lathe meshes from equations. We iterate "y" from 0 to 1 and "t" from 0 to 2 pi and use the return argument of the function to generate the mesh.

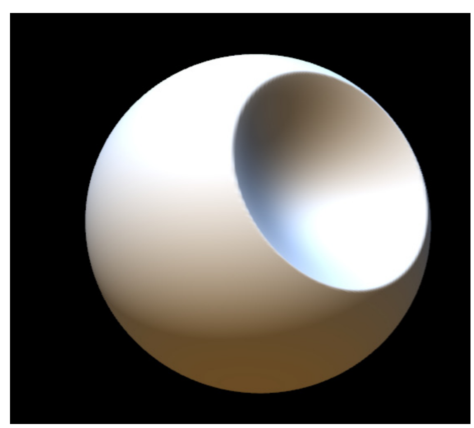
```
lathe 120 30
  return sqrt(y)*sin(4*t)
```



Displacement

Kiwi support arbitrary displacement functions that can operate on vertices or normals. This involves recalculating normals for the fragment shader.

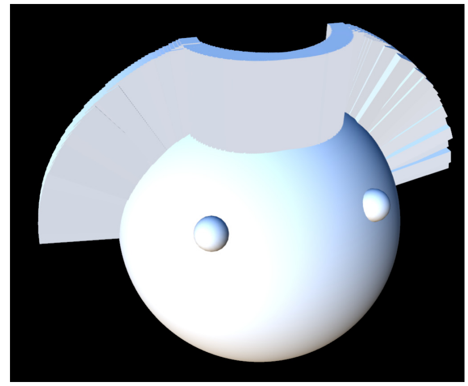
```
sphere 300
displace
  set r x*x+y*y
  if r<0.1
    set z  (0.4+6*r)*z
```



Attaching

Oftentimes, models are created by “attaching” primitives to one another. Instead of requiring the user to remember a complicated series of transforms, the “attach” function allows a user to specify a face on a model to attach to, adopting the local transforms of the face.

```
sphere 300
loop i -100 100
  set x (i/100)*0.5
  scale 0.05 0.5 0.05
    attach x 0 0 x 10 10
      cube 1
set x rand(0.2, 0.3)
set z rand(0.3, 0.4)
scale 0.1 0.1 0.1
  attach x 0 z x 10 z
    sphere 20
  attach -x 0 z 0-x 10 z
    sphere 20
```

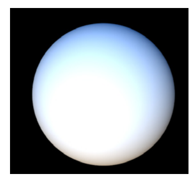


Kiwi Randomization

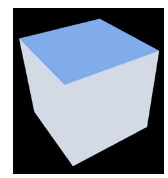
Choose

Users can define a “choose” block, where each “option” in the block can have a probability associated with it.

```
choose
  option 1
    cube 1
  option 1
    sphere 100
```



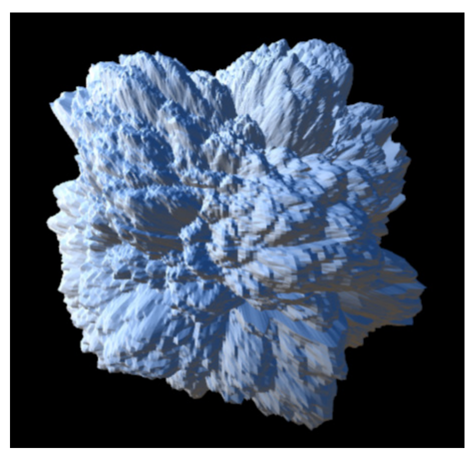
OR



Perlin Noise Displacement

I implemented a vertex shader that uses Perlin Noise to displace a mesh. This allows one to create interesting, randomize displacement maps.

```
sphere 300
displacen 10
```



Vertex Shader Displacement

One of the most powerful features of Kiwi is the ability to create GLSL vertex shaders directly in Kiwi. In games, meshes have to be low-polygon so that they can be rendered at an adequate framerate. Web-games in particular need to use extremely low-polygon meshes to guarantee performance on all computers. One technique that is used to simulate high-polygon meshes is to encode mesh detail in the vertex shader, and allow the GPU to apply displacement functions to the vertices.

To be able to accomplish this in Kiwi, I gave Kiwi the ability to compile directly into GLSL. This way, a game developer can use Kiwi to program a procedural vertex shader. For, example, the Kiwi code:

```
sphere 100
displacev
  loop i -10 10
    if i*x < 0.01
      set x 0.8*sin(PI*x)
```

compiles into the corresponding GLSL code:

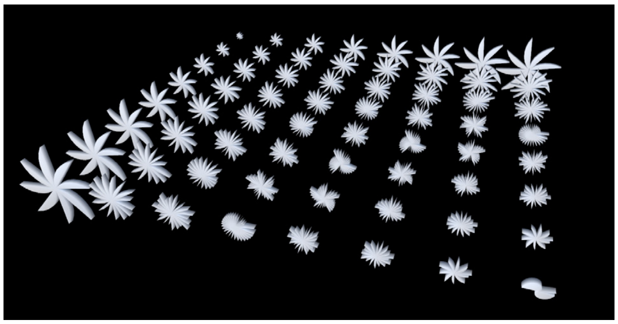
```
float PI = 3.141592653589793;
float EPS = 0.001;
for (float i = -10.; i < 10.; i++) {
      if (((i)*(x))<(0.01)) {
            x = ((0.8)*(sin((PI)*(x))));
      }
}
```

This code is automatically inserted into the appropriate mesh's vertex shader to transform vertex position. These vertex shaders can also be appended to one another in a sequence. Thus, a developer can encode an arbitrarily complex vertex shader in Kiwi.

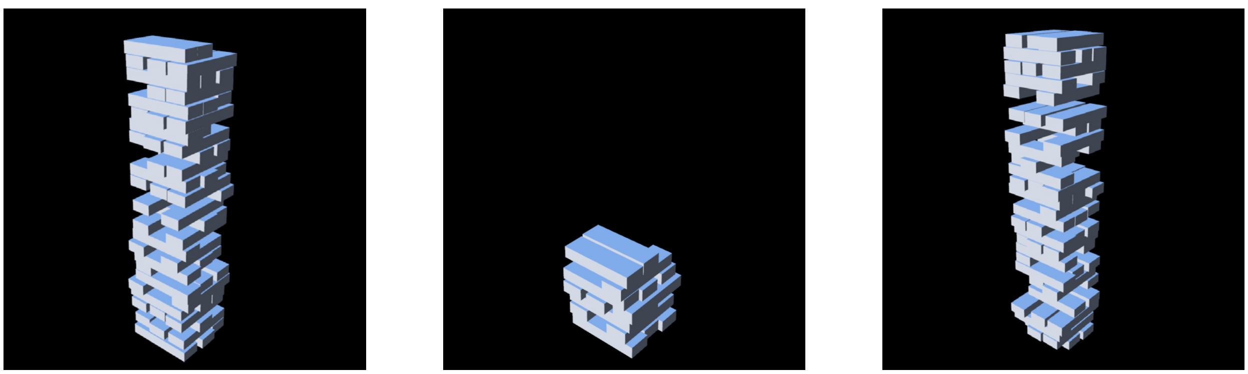
Complex Meshes

When combined, the geometry and randomness operations allow a user to create procedural meshes of arbitrary complexity. Here are some examples of meshes that I made, each of which takes only a few lines to code:

Flowers

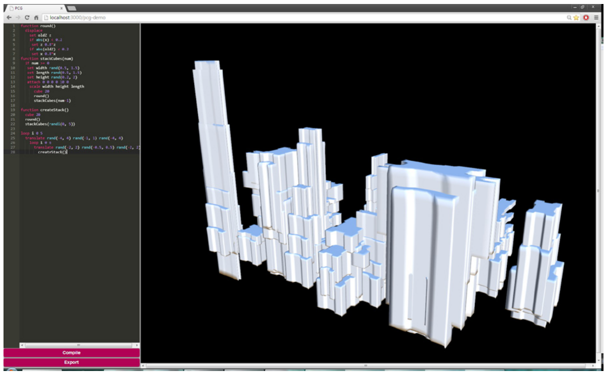


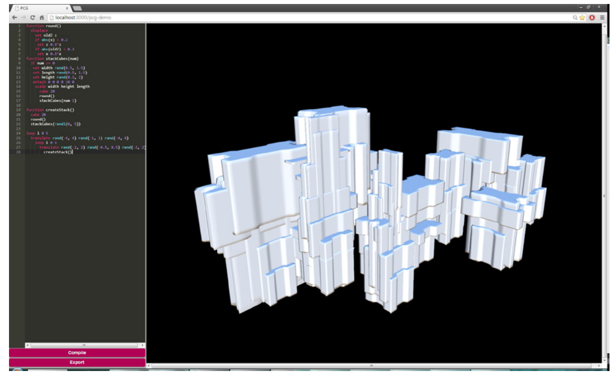
Jenga



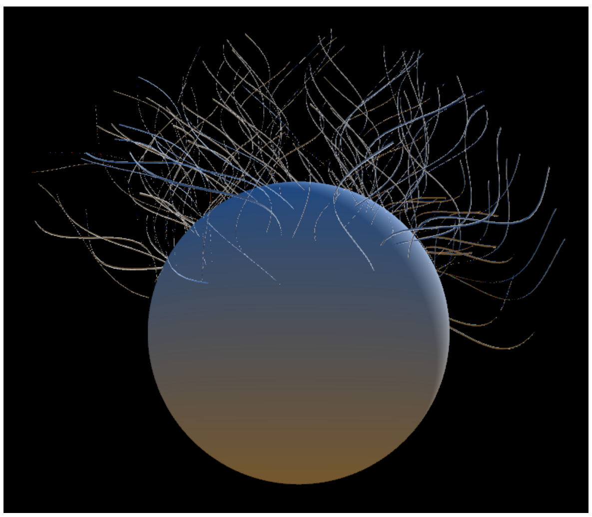
Christmas Tree

Buildings





Hair



Exportable

Models can be exported to .stl format and used in a variety of applications, such as game engines, 3d printers or 3d software packages. Figure 4 shows a rendering of the one of the Christmas trees.

Figure 4. A rendering of a Kiwi-generated Christmas tree.