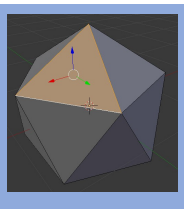
Efficiently procedurally generating polyhedrons

Richard Steele 1607539

When researching existing online technical portfolios, the increased strain upon the computer to render complex shapes using WebGL was very apparent. The aim of my WebGL portfolio is to be as lightweight as possible to give the best user experience.

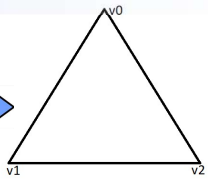
My objective includes procedurally generating polyhedron. However, storing and transmitting the needed buffer data can become increasingly expensive for more complex shapes. As the vertices and indices must be derived at some point, instead of predetermining these and transmitting them as part of the program, I have chosen to derive an algorithm to generate them at run time. An efficient solution will require avoiding repeated vertices for a compact vertex array, and a specific index array generator to avoid drawing unneeded triangles inside the polyhedron.

Generating new vertices and indices lists



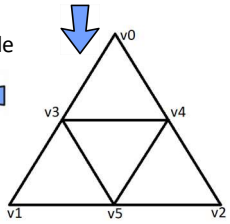
We start with a simple base shape. The indices and vertices can be generated through trigonometry, or at this level it is acceptable to supply them as literal values as the scope is limited and the amount of data will be small.

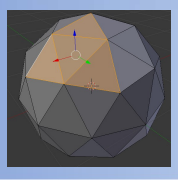
Consider each triangle face of the base shape in turn. For each edge, find the midpoint of the two consecutive vertices, and extrude to the desired radial distance from the centre.



The indices list to draw the subdivided triangle can be any variation of the four newly available triangles. This project chose to use as the **index ordering system**:

```
[v0,v3,v4, v1,v5,v3, v2,v4,v5, v5,v4,v3]
```



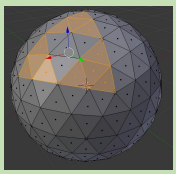


Vertex pruning

As the key focus of this project is **efficiency**, we must consider that newly derived vertices may have been previously derived and used by neighbouring triangles. So before adding these new vertices and indices to the respective lists, **we must check if the vertex already exists**. If so, then a reverse look-up will return the appropriate index for that vertex and we can discard the duplicated one. If not, then the vertex is added to the vertices list, and its index is its place in that list. At the time of pushing the new value onto the array then the index is the vertex's place in the array:

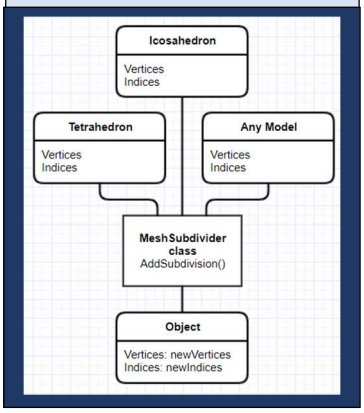
```
newIndex = vertexArray.length-1
```

After all duplicated vertices are discarded, we can use the above index ordering system to construct a temporary indices list to push onto the indices array.



This process can be repeated over many iterations to tend towards a sphere shape.

By abstracting this process into a separate class, any model with vertices and indices can be passed in to a subdivide function, and an object consisting of the new vertices and new indices will be



What worked well?

The initial plan was to generate a full list of dissected and extruded triangles, then prune that list of repeated vertices. An early optimisation was to incorporate this pruning into the triangle dissection with a reverse look up into the vertex array to return the index if the vertex already exists. Eliminating the need for a second loop through the values to check for repeated values.

Benchmark test

Method justification by time taken to complete five iterations on Lenovo X220 - i7 2620M 2.7GHz

Without vertex pruning:

2min 35sec

With vertex pruning:

42sec

Areas of improvement

While this approach takes advantage of a supplied set of initial vertices and indices, a completely procedurally generated polyhedron would derive the base shape as well.