Subdivision surfaces

A subdivision surface is a scheme for constructing a smooth surface from a grid of control points. The surface is formed by introducing new vertices along the edges and in the center of the faces of the original control grid, and then systematically deforming the shape by an averaging process applied to the vertices.

The image sequence above shows an initial control grid (the cube), and a series of rounds in a subdivision algorithm. In this example the subdivision algorithm is the Catmull-Clark algorithm, described below.

The net result of the subdivision is to smooth off the sharp corners and edges of the original control surface to make a surface that is almost everywhere smooth.

Subdivision algorithms are typically simple enough that the subdivisions can be computed in real time. This makes subdivision surfaces suitable for use in real-time design tools that allow users to manipulate the control grid and see instantly what effect the changes to the control grid will have on the resulting surface.

Advantages of subdivision surfaces

Subdivision surfaces were developed in part to overcome specific problems with Bezier patches and other patch-oriented solutions.

The first problem that patch-oriented solutions have is in constructing shapes with complex topology, such as shapes with holes in them.
In a patch-oriented solution such figures have to be constructed by splicing together many patches. With appropriate modifications, subdivision surface algorithms can make it easier to construct complex shapes from a relatively simple control grid.

The second problem with patch-oriented solutions crops up in the animation of complex figures. As we animate a complex figure constructed from patches, the seams between patches often have trouble. Gaps can open up during animation, or simply not look right.

Subdivision surfaces avoid these problems by allowing animators to construct a single, unified control grid for a complete, complex figure. This eliminates the problem with seams.

**Catmull-Clark subdivision**

There are many subdivision surface algorithms in common use. One of the more widely used schemes is due to Catmull and Clark.
Here is an outline of the steps in a single round of subdivision:

1. We construct a new vertex at the center of each face. The initial location of this vertex is in the geometric center of the face, determined by averaging together the locations of the vertices that make up the polygonal face.

   \[ v_f = \frac{1}{m_f} \sum_j v_j \]

2. We construct a new vertex for each edge in the grid. For an edge connecting vertices \( v_1 \) and \( v_2 \) and sitting between faces \( f_1 \) and \( f_2 \), we average together the new face vertices and the vertices that define the edge to make a new edge vertex.

   \[ v_e = \frac{1}{4} (v_1 + v_2 + v_{f_1} + v_{f_2}) \]

3. Finally, we move the original vertices all to new locations by averaging together the vertex itself, the other vertices it is connected to by edges, and the face vertices for the faces that come together at that vertex.

   \[ v_v = \frac{n_v - 2}{n_v} v + \frac{1}{n_v^2} \sum_j v_j + \frac{1}{n_v^2} \sum_j v_{f_j} \]

**Tweaking the algorithm to create creases**

One problem with the original Catmull-Clark scheme is that it causes all sharp edges in a control grid to "melt away" with iteration. A simple trick to prevent this is to tag certain edges in the original control grid as "sharp" edges. We then modify the update rules to cause vertices associated with sharp edges to move in a different way.

- For new vertices created on sharp edges, the new vertex is the average of the edge's endpoints (and does not move under the influence of nearby faces). The new sub-edges created by adding the new vertex are also tagged as sharp.

- We modify the update rule for vertices by taking into account how many sharp edges meet at that vertex.
  - If fewer than two sharp edges meet at this vertex, we use the original update rule above.
  - If exactly two sharp edges meet at the vertex, we use the crease vertex rule: the new vertex is a weighted average of the old vertex location (3/4) and of the two other endpoints of the incident creases (1/8).
  - If more than two sharp edges meet at a vertex, the vertex is considered a sharp corner and does not move.

Here is an illustration of this modified algorithm. The red edges in the original control grid have been tagged as sharp edges.
Further variations on this idea include the ability to give sharp edges a measure of sharpness, which forces the update algorithm to use a blend of the sharp and unsharp update rules. This makes it possible to create "semi-sharp" edges.

Displacement mapping

One problem with subdivision surfaces is that they frequently appear a little too smooth and "blobby." A simple fix for this problem is to use a technique called *displacement mapping*. In this technique we wrap a special texture around the object and use the texture to give the object a bumpy appearance. Unlike bump mapping, which uses a
texture to modify normals, displacement mapping uses the applied texture to displace the vertices in the subdivision surface. By applying small local displacements to the vertices, we can create bumps and spikes in the surface.

The images below illustrate how this works.

![Subdivision Surface Model](image1)

The figure on the bottom left is a subdivision surface model shown along with its containing control grid. The image in the upper right-hand corner is the displacement map. The image on the bottom right-hand side is the result of mapping the displacements onto the subdivision surface.

Displacement maps can be implemented in a straight-forward way in a vertex shader. This is effective, because subdivision surfaces can create large numbers of vertices for us to manipulate. If we don't have enough vertices to capture the fine details in a displacement map, we can simply run one or two more rounds of subdivision to generate a grid that is sufficiently fine.

Data structures for subdivision surfaces

A subdivision surface is more complex than a traditional mesh. In a traditional mesh, all we really need to store is a set of vertices that make up the mesh. In a subdivision surface, on the other hand, vertices will be arranged into faces and edges. To start to construct a data structure that will allow us to track all of the relevant components of a subdivision surface, we would have to follow an approach something like the following:

```cpp
class Mesh {
    struct face_t {
        Cvec<int, 4> vertex_; // this will be either a tri or a quad
        // (face_t::vertex[3] == -1 => this is a tri)
```
The structure shown here is the structure used for a Mesh class provided by the author on the textbook's web site.

The most obvious aspect of this structure is the fact that a subdivision surface is a mesh: it contains a vector vertex_ of vertices, and each vertex contains position and normal information, as is usual for a mesh. A subdivision surface goes beyond this, in that the vertices are then also organized into faces and edges. A face is a list of vertex numbers, giving the vertices that participate in this face, and a list of edge numbers that identifies the edges that participate in this face.

The one special aspect of this data structure is the edges. The most obvious way to represent an edge is to say that it connects two vertices. However, this raises what is essentially a search problem. The search problem comes about because the edge also joins two faces, and at various times we will need to know what those two faces are. One way we could discover this information is by brute force: we could simply search through all the faces and find the two faces that share the given pair of vertices. This is less than efficient, so we need to switch to an alternative representation. Instead, we will think of each edge as being composed of two "half edges." Each half edge belongs to a face, and runs to a particular vertex in that face. We can pack this information into a single integer by encoding the face number and the vertex number within that face into a single int by doing something like this:

```
code = (vertex_number << 28) | face_number;
```

Since the vertex numbers for a quadrilateral face will be 0, 1, 2, and 3, this computation is in no danger of overflowing an int. From the two half edge codes embedded in an edge we can determine what two faces the edge connects, and what two vertices the edge connects.

Once we have a data structure that makes it easy for us to iterate over the faces making new face vertices and iterate over the edges making new edge vertices, we can implement a subdivision algorithm.