High Performance Subdivision Surfaces

Chris Lomont, June 2007, www.lomont.org

Introduction

Subdivision surfaces are a method of representing smooth surfaces using a coarser polygon mesh, often used for storing and generating high detail geometry (usually dynamically) from low detail meshes coupled with various scalar maps. They have become popular in modeling and animation tools due to their ease of use, support for multi-resolution editing, ability to model arbitrary topology, and numerical robustness. This article presents enough detail about a specific subdivision method for direct implementation. The main result is a complete set of subdivision rules for geometry, texture, and other attributes. A second result is an overview of methods for fast generation and rendering.

Background

There are many types of subdivision schemes, with varying properties. Some of the properties are

- Mesh type – usually the mesh is made of either triangles or quads.
- Smoothness – this is the continuity[1] of the limit surface, and is usually denoted C^1, C^2…, etc., or G^1, G^2…., etc.
- Interpolating [Zorin96] or approximating – interpolating schemes go through the original data points, while approximating schemes may not.
- Support size – this is the amount of neighboring geometry affecting the final position of a given surface point.
- Split – some schemes work by replacing faces with more faces, others work by replacing vertices with new sets of vertices. A few more work by replacing the entire previous mesh, making a “new” mesh.

Table 1 lists common schemes and some data about them.

Method	Mesh	Smoothness[2]	Split	Scheme
Catmull-Clark	Quads	C^2	Face	Approximating
Doo-Sabin	Any	C^1	Vertex	Approximating
Loop	Triangles	C^2	Face	Approximating
Butterfly	Triangles	C^1	Face	Interpolating
Kobbelt	Quads	C^1	Face	Interpolating
Reif-Peters	Any	C^1	New	Approximating
Sqrt(3) (Kobbelt)	Triangles	C^2	Face	Approximating
Mideedge	Quads	C^1	Vertex	Approximating
Biquartic	Quads	C^2	Vertex	Approximating

Table 1- Subdivision Schemes

[1] Technically, this is true for all points except “a set of measure zero,” not discussed here. G^n is geometric continuity, meaning tangent directions line up, but may differ in magnitude. This is slightly weaker than C^n.
[2] Smoothness generally has one degree less of continuity at exceptional points.

Although this article focuses mainly on generating geometry and rendering issues, subdivision surfaces have many other uses including

- progressive meshes,
- mesh compression,
- multi-resolution mesh editing[3],
- surface and curve fitting[4],
- point set to mesh generation.

Most 3D animation and rendering packages support subdivision surfaces[5] as a primitive, although there is no standard type used throughout the industry. Catmull-Clark and Loop subdivision are the most commonly used.

A related topic is PN triangles [Vlachos00], which is a way to replace triangles at the rendering level with a smoother primitive. The basic idea is to quadratically interpolate surface normals, similar to Phong shading, and to use this cubically to interpolate new geometry. A good overview of subdivision is [Zorin00].

Usage

For a production tool chain for interactive games, one method to use subdivision surfaces would be to create art using high-resolution geometry and textures (and the geometry might be modeled in whatever subdivision flavors the tools support). The art is then exported as high-density polygon models and associated data. Tools then reduce the assets to a low poly count mesh with associated displaced subdivision maps, textures, and animation data. A subdivision kernel in the GPU dynamically converts assets back to needed poly counts at runtime based on speed, distance from camera, hardware support, etc. This allows different subdivision surfaces to be used in the asset creation and asset rendering stages, which has advantages.

A tool along these lines is ZBrush, which allows editing meshes using subdivision surfaces to add geometry at multiple resolution levels, and then converts the resulting high detail geometry to low detail meshes and displacement maps.

Choice of Subdivision Type

This article covers implementing Loop subdivision [Loop87]. Some reasons are that it is triangle based, making it (perhaps) easier to implement on GPUs, most artists and tools already work with triangle meshes, it is well studied, and it produces nice looking surfaces. Another common choice is Catmull-Clark subdivision [Catmull78], but being quad-based, seems less suitable for gaming. Pixar uses Catmull-Clark subdivision for animating characters. Many ideas presented in this article are applicable to quad based subdivision as well as other schemes.

---

[3] [Zorin97]
[4] [Lee98], [Levin99]
[5] See http://www.et.byu.edu/~csharp2/#A_SubD [as of 2007] for a partial list of toolsets supporting subdivision.

Loop Subdivision

Feature Options

A single iteration of the original Loop subdivision algorithm applied to a closed triangle mesh returns another closed triangle mesh with more faces. Repeated applications results in a smooth limit surface. Extensions to the original method are needed to model more features; a full-featured subdivision toolset includes

- Boundaries – allows non-closed meshes.
- Creases – allows sharp edges and surface ridges. Adding boundaries gives creases[6].
- Corners – useful for making pointed items.
- Semi-sharpness – modifies the basic rules for boundaries, creases, and corners to get varying degrees of sharpness.
- Colors and textures – easy extensions of the subdivision process needed for rendering and gaming.
- Exact positions – after a few subdivisions, vertices can be pushed to what would be their final position if the subdivision were carried out to the limit. This computation is not very expensive.
- Exact normals – computing exact normals for shading is not very expensive, and is less costly than face normal averaging.
- Displacement mapping – adds geometry to the subdivided surface, and is a very nice feature to have, but not implemented in this article. Instead see [Lee00] and [Bunnell05].
- Evaluation at arbitrary points – allows computing the limit surface at an arbitrary position on the surface [Stam99]. This is useful for raytracing or very detailed collision detection, but for game rendering is not likely to be needed.
- Prescribed normals – allows requesting specific normals on the limit surface at given vertices [Biermann06], and is useful for modeling. However it is more expensive to implement than what is presented below, and for this and space reasons details are omitted.
- Multi-resolution editing support – by storing all the levels of the subdivided mesh, a user can work on any level of the subdivision, making many editing features easier [Zorin97].
- Collision detection – needed for game dynamics; one method is in [DeRose98].
- Adaptive subdivision – subdivides parts of the mesh different amounts based on some metric, patching any holes formed in the process. Adaptive subdivision is useful to keep polygon counts low while still giving nice curves, silhouettes, and level of detail. Selecting where to subdivide the mesh is usually based on curvature.

---

[6] Creases technically should have the techniques in [Biermann06] to prevent minor corner errors, but [Zorin00] claims these errors are visually minor. The corrections require more computation than what is presented and intended: a technique suited for real-time rendering.

Features are added to the mesh by tagging vertices, faces, and edges with parameters to direct the subdivision algorithm. Tag combination restrictions can be enforced in software to prevent degenerate cases if needed.

Geometry Creation

To implement boundaries, creases, corners, and semi-smooth features, each vertex and edge is tagged with a floating-point weight $0 \le w < \infty$. A weight of 0 denotes standard Loop subdivision, and ∞ denotes an infinitely sharp crease or boundary. Infinity need not be encoded in the data structures; since the weight is really a counter for the levels of subdivision affected. Any number larger than the highest level of subdivision performed will suffice. For example, 32767 should suffice, since it is unlikely that any mesh will be subdivided this many times.

Loop subdivision takes a mesh and creates a new mesh by splitting each old triangular face into four new faces. This is done in two steps. The first step inserts a new vertex on each existing edge, and the second step modifies old vertices (not those inserted on the edges).

Most of these geometry rules are from [Hoppe94a] and [Hoppe94b] with some ideas merged from [DeRose98] and [Schweitzer96].

Edges

The first step inserts a new vertex on each edge using a weighted sum of nearby vertices. Edge weights and the types of vertices at each endpoint of the edge categorize edges. Vertex categories are below.

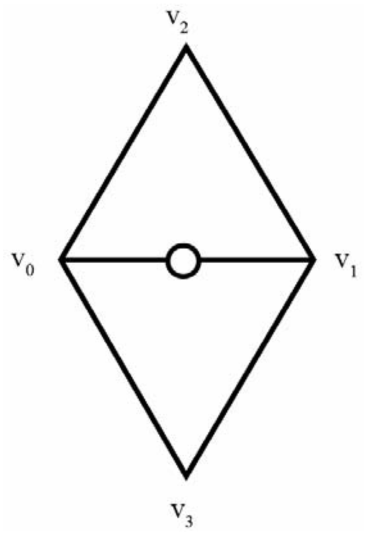

Figure 1 - Edge mask

Each (non-boundary) edge has two adjacent triangles; the new vertex has the position $\frac{3}{8}(v_0 + v_1) + \frac{1}{8}(v_2 + v_3)$ where v_0 and v_1 are the vertices on the edge to split, and the other two vertices are the remaining vertices on the two adjacent triangles. This is illustrated symbolically in Figure 1, where the circle denotes the new vertex on the edge between the triangles. The weights can be written $\left(\frac{3}{8}, \frac{3}{8}, \frac{1}{8}, \frac{1}{8}\right)$, where position j corresponds to vertex j (0-indexed).

	Dart	Regular crease	Non-regular crease	Corner
Dart	1	1	1	1
Regular crease	1	2	3	3
Non-regular crease	1	3	2	2
Corner	1	3	2	2

Table 2 - Edge mask selection

The weights used to create a new edge depend on the edge weight and the vertex types of the two vertices v_0 and v_1 defining the edge. Given the two vertices on an edge, Table 2 shows which type of weights to use to create the new edge vertex. Weights are

- Type 1 weights : $\left(\frac{3}{8},\frac{3}{8},\frac{1}{8},\frac{1}{8}\right)$.
- Type 2 weights : $\left(\frac{1}{2},\frac{1}{2},0,0\right)$.
- Type 3 weights : $\left(\frac{3}{8},\frac{5}{8},0,0\right)$, where the $\frac{3}{8}$ weight goes with the corner edge.

An edge is *smooth* if it has weight $w=0$. An edge is *sharp* if its weight is $w\geq 1$. If an edge has weight $0<w<1$ then the new vertex is linearly interpolated between the two cases $w=0$ and $w=1$, keeping the end vertex types fixed.

When an edge is split, each new edge gets weight $\tilde{w}=\max\{w-1,0\}$. This gives finer control over sharpness since the crease rules are applied a few levels, then the smooth rules, with possible interpolation on one step. An option for more control is to tag each end of an edge with a weight, giving two weights per edge, interpolate the new edges, and make the corresponding changes throughout.

Note in all cases the total weight sums to 1 (also true for vertex masks).

Vertices
The *type* of a vertex depends on the vertex weight and the types of incident edges.

A *smooth* vertex is one with zero incident sharp edges and weight 0. A *dart* vertex has one sharp incident edge and weight 0. A *crease* vertex has two sharp incident edges and weight 0. A *corner* vertex has >2 sharp incident edges or has weight $w\geq 1$. An interior crease vertex is *regular* if it if it has six neighbors and exactly two non-sharp edges on each side of the crease; a boundary crease vertex is *regular* if it has four neighbors. Otherwise crease and boundary vertices are *non-regular*. If an edge has weight $0<w<1$it suffices to call it smooth for vertex classification.

The second step of Loop subdivision modifies all the original vertices (not the vertices inserted on each edge in step one) using a weighted sum of the original vertex and all neighboring vertices.

The weighting is dependent on the number n of neighboring vertices. For smooth and dart vertices this is illustrated in Figure 2. The value of b is

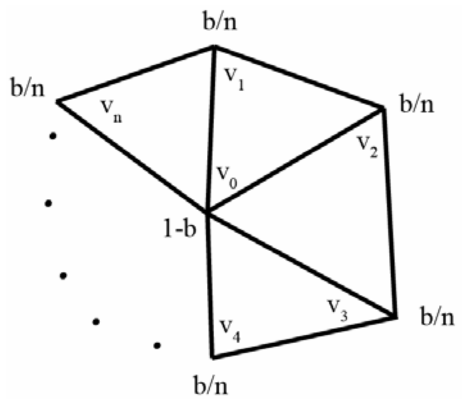

Figure 2 – Vertex mask

usually $b(n) = \frac{1}{64}\left(40 - \left\{3 + 2\cos\frac{2\pi}{n}\right\}^2\right)$, although other values are in the literature[7]. The old vertex is given weight $1 - b(n)$ and each old neighbor (not the vertices created in step one!) is given weight $b(n)/n$ to determine the new vertex position, which is then the weighted sum of all these vertices: $v_{new} = (1 - b(n)) \cdot v_{old} + \frac{b(n)}{n}\sum_{j=1}^{n} v_j$

For corner vertices, the vertex position does not move, so $v_{new} = v_{old}$.

For crease vertices, the new vertex is the sum of $\frac{3}{4}$ of the original vertex and $\frac{1}{8}$ of each of the two neighbors on the crease.

If a vertex has weight $0 < w < 1$ then the new vertex is linearly interpolated between the two cases $w = 0$ and $w = 1$. A new vertex also has a new weight $\tilde{w} = \max\{w - 1, 0\}$.

The final case is when a vertex has weight $0 < w_v < 1$ and some neighboring edge has weight $0 < w_e < 1$, leading to many possible combinations of interpolation. In this case evaluate each with weights 0 and weights 1, and interpolate on w_v, instead bilinearly interpolating the four cases the weights (0,0), (0,1), (1,0), (1,1).

Another option is to require integer weights, avoiding interpolation cases entirely at the loss of control on semi-sharp creases.

Displacement-mapped surfaces are implemented by moving the vertices as needed according to a displacement map. Vertices are also modified using [Biermann06] to implement prescribed normals, also splitting crease rules into convex and concave cases, avoiding certain degenerate cases.

Limit Positions

Vertices can be projected be projected to the position they would take if the surface were subdivided infinitely many times. This is often done after a few subdivision levels have been applied. This is optional and often doesn't modify the surface much.

Limit positions v^∞ are computed from a weighted sum of the current vertex v_0 and n neighbors v_j. Corner vertices stay fixed, that is $v^\infty = v_0$. Smooth vertices are projected using $v^\infty = \frac{3}{8b(n)(n+1)} v_0 + \frac{1}{n+1}\sum_{j=1}^{n} v_j$. A regular crease uses weights $\left(\frac{1}{6}, \frac{2}{3}, \frac{1}{6}\right)$ with v_0

[7] For example, [Warren95] proposes $b(n) = 3/(8n)$ for $n > 3$ and $b(3) = 3/16$, but this has unbounded curvature for a few valences.

getting $\frac{2}{3}$ and the two crease neighbors getting weight $\frac{1}{6}$, the rest of the neighbors get weight 0. Similarly non-regular creases use weights $\left(\frac{1}{5},\frac{3}{5},\frac{1}{5}\right)$.

Normals
True normals can be computed for each vertex, which should be done after computing limit positions for each final vertex. Surprisingly this is faster than computing approximate normals by averaging each adjacent face normal (partitioned to each side of a crease).

Computing two true tangents and taking a cross product computes true normals at each vertex.

For a smooth or dart vertex, the two tangents are $t_1 = \sum_{j=1}^{n} v_j \cos\left(\frac{2\pi * j}{n}\right)$ and $t_2 = \sum_{j=1}^{n} v_j \cos\left(\frac{2\pi * (j+1)}{n}\right)$.

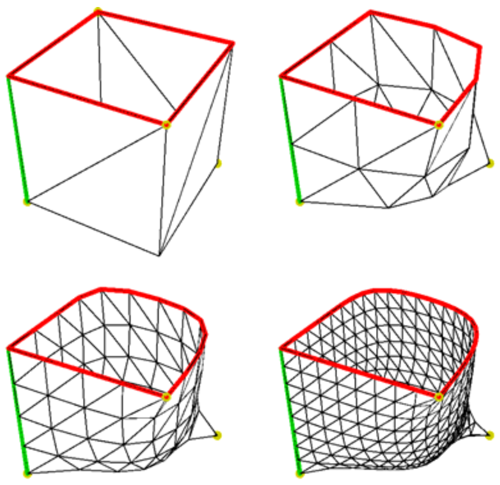
Figure 3 - Example geometry

Crease and boundary vertices require more work. Normals are not defined per vertex for corners, and must be done for each face. Tangents need computed for each side of the crease. Along a crease (or boundary) one tangent is -1 times one crease neighbor plus 1 times the other crease neighbor. The second tangent is more complicated to compute and is done as follows. Weights w_j are computed for each vertex, with $j = 0$ being the vertex where a normal is desired. Then the other indices are numbered $j = 1,2,...,n$ from one crease to another. The weights depend on the number of vertices and for each case are:

$$(w_0, w_1, w_2) = (-2,1,1)$$
$$(w_0, w_1, w_2, w_3) = (-1,0,10)$$
$$(w_0, w_1, w_2, w_3, w_4) = (-2,-1,2,2,-1)$$

$$w_0 = 0, w_1 = w_n = \sin(z),\ w_i = (2\cos(z) - 2)(\sin(i-1)z),\ z = \frac{\pi}{n-1} \text{ for } n \geq 5.$$

An example of the geometry this creates over 4 subdivision levels from a tagged cube with one face missing and marked as boundary is in Figure 3. Notice some corners have varying semi-sharpness.

Feature Implementation

Besides geometry, a full solution needs colors, textures, and other per-vertex or per-face information.

Face parameters like color and texture coordinates can be interpolated using the same subdivision methods when new vertices are added. A simple method is to interpolate by distance after the old vertices are modified, giving new values for the new faces. Many features can be subdivided per vertex except at exceptional places, like along an edge where texture coordinates form a seam. Some details for Catmull-Clark surfaces (but applicable to Loop surfaces) are in [DeRose98]. Basically per-vertex parameters are interpolated like vertex coordinates, so adding (u,v) texture coordinates is as simple as treating vertex points as (x,y,z,u,v) coordinates. Per-vertex textures don't allow easy handling of seams, in which case per-face texture coordinates are useful. However, all internal points on a subdivided face become per-vertex parameters.

Possible features to add but not covered in this article for lack of space are adaptive tessellation (where only part of the mesh is subdivided as needed for curvature, silhouettes, clipping, etc., and making sure cracks aren't introduced), and displaced subdivision surfaces (which adds geometry by using a "texture" map to offset generated vertices as they are computed). Adaptive tessellation is covered in [Bunnell05] and displaced subdivision surface are covered in [Bunnell05] and [Lee00].

Collision Detection

If prescribed normals are not implemented, then the surface has the convex hull property, that is, sits inside the convex hull of the bounding mesh. This can be used for coarse collision detection. More accurate (and more expensive) collision between subdivision surfaces is covered in [Wu04] using a novel "interval triangle" that tightly bounds the limit surface. [Severn06] efficiently computes the intersection of two subdivision surfaces at arbitrary resolutions. Collision detection will not be covered here further.

Next a data structure is presented that accommodates Loop subdivision on a triangle mesh supporting many features. An algorithm follows that performs one level of subdivision, returning a new mesh. This structure supports most of the features described above, and is extensible to many of the other features.

Subdivision Data Structure

There are many approaches in the literature for data structures used to store and manipulate subdivision surfaces including half-edge, winged-edge, hybrid, and grids [Müller00]. For Loop subdivision, the data structure should allow finding neighbor vertices and incident edges easily, and preserve this ability on each level of subdivision.

There are many factors in designing the data structure. Converting a mesh to a Loop subdivided mesh is the main goal, leading to certain structures, but other times the end purpose is GPU rendering, in which case optimizing data structures for this use makes sense. The approach presented here is somewhat of a hybrid, resulting in a data structure that ports easily to a GPU. A later section covers performance issues when moving to a GPU.

The following data structure is easy to read/write from files or elsewhere, fast to use internally, and does not use pointers. Avoiding pointers makes it easier to move to GPUs or languages not as pointer friendly as C/C++, and makes the memory footprint smaller than the above schemes, since instead of storing connectivity information explicitly it is deduced from index positions. This structure also makes sending meshes to a GPU easier since items are arranged into vertex-arrays, normal-arrays, etc., using indices to render polygons.

Data Structure

Three benefits for avoiding pointers are it makes this algorithm easier to port to other hardware, it uses less memory (useful for large mesh tools), and a third benefit is it is simple. See Figures 4 and 5 for insight. Extensions (not discussed further) allow storing all the levels of subdivision for multi-resolution editing.

The mesh and supported features are stored in various arrays. Each array is 0-based. There is one array for each of the following:

1. Vertices array VA – each vertex is a 3-tuple x, y, z of floats, and a float sharpness weight $0 \le w < \infty$, with 0 being smooth, and a half-edge index v_h of a half-edge ending on this vertex (for fast lookup later). If the vertex is a boundary then v_h is the boundary half-edge index ending on the vertex. Optional per-vertex color, texture, or index to a normal can also be stored.

2. Faces array FA – each face represents a triangle, stored as three indices v_0, v_1, v_2 into the vertex array. Also stored are three indices n_0, n_1, n_2 into the normal array NA, corresponding to the three vertex indices. Each face can optionally store color, texture, and other rendering information, per face or per vertex as desired. Faces are oriented clockwise or counter-clockwise as desired, but all must be oriented the same way.

3. Half-edge array HA – each face has three (half) edges in the half-edge array, stored in the same order. Thus face with index f and (ordered) vertex indices $\{v_0,v_1,v_2\}$ has ordered half-edges with indices 3f, 3f+1, and 3f+2, denoting half-edges from vertex v_0 to v_1, v_1 to v_2, and v_2 to v_0, respectively. Note that half-edges are thus directed edges, with the two half-edges of a pair having opposite directions. A half-edge entry is two values: an integer marking the pair half-edge index or a –1 if a boundary, and a floating-point sharpness weight $0 \le w < \infty$ denoting the crease value, 0 being smooth, and larger values denoting sharpness. Each matching half-edge pair must have the same crease values to avoid ambiguity. Note that given a half-edge index determines the corresponding face index, which in turn determines a start and end vertex for the directed half-edge.

4. Normals Array NA – Normals can be included into the scheme in numerous ways with varying tradeoffs. In order to handle creases, boundaries, and semi-sharp features cleanly, one normal per vertex per face is needed, but for many vertices (smooth, regular, etc.) only a single normal is needed. An array of normals accommodates this, each a unit vector and a weight $0 \le w < \infty$ telling how fast a vertex normal converges to a prescribed normal, with 0 meaning no prescribed normal. Normals are referenced by index, avoiding redundant stores.

Besides knowing the size of each array, the number of edges E (where a matching pair of half-edges or a boundary edge constitutes a single edge) is stored. This is not too costly to compute if the mesh has no boundary (E=# half-edges/2 = #faces*3/2), and can be computed otherwise by scanning the half-edge array and setting E=(size of HA+# of boundary edges in HA)/2.

Optionally info about vertex types (smooth, crease, etc.) may be stored on a vertex tag for speed. Other items may also be tagged similarly, but the algorithm below obviously needs modified to maintain the invariants across subdivisions.

Unneeded features can be dropped, such as three normals per face, prescribed normals[8], or semi-sharp creases, but this loses finer grained control.

A well-formed mesh requires a few rules. If a half-edge is not paired (it is on a boundary), it has pair index -1, and must have infinite crease weight. Otherwise the edge will shrink. Each half-edge of the same edge must have the same weight; otherwise the edges will subdivide differently, creating cracks.

File Format

Based on the above data structure, a file format is defined as an extension to the popular text based Wavefront *.OBJ format. An entry is a line of text, starting with a token denoting the line type followed by space-separated fields. Various pieces of data are stored to speed up loading so all items such as paired edges do not need recomputed each load. The format in order of file reading/writing is in Table 3.

Entry	Description
`#SubdivisionSurfL 0.1`	Denotes a non-standard OBJ file, versioned.
`si v f e n`	Optional Subdivision Info giving # of vertices, faces, edges, and normals. Allows pre-allocation of arrays.
`v x y z`	One entry per vertex with floating point position.
`f v1 v2 v3`	One entry per face with 1-based vertex indices, oriented.
`hd j wt`	Half-edge Data, one entry for each half-edge, in the order described by the faces, in half-edge order $v_0 \to v_1$, $v_1 \to v_2$, $v_2 \to v_0$. Each entry is a 1-based integer edge

[8] Prescribed normals are not implemented in this article.

	pair index `j` (or -1 for a boundary) and a floating-point weight `wt`.
`fc r1 g1 b1 a1 r2 g2 b2 a2 r3 g3 b3 a3`	Optional face colors, one per vertex, RGBA, [0,1] floats. Not allowed with per-vertex colors `vc`.
`vc r g b a`	Optional per-vertex color data, RGBA, [0,1] floats. Not allowed with per-face colors `fc`.
`ft u1 v1 u2 v2 u3 v3 texname`	Optional face textures with (u,v) floats in [0,1]. `texname` is application dependent.
`fn nx ny nz w`	Optional normal data, with weights for prescribed normals. 0 is default weight.
`fni n1 n2 n3`	Optional face normal indices into the normal table, one normal per vertex. Requires `fn` entries.
`vs wt`	Optional vertex sharpness, $[0,\infty)$, with 0 being smooth and default, one per vertex.

Table 3 – File format entries

Subdivision Algorithm Details

This is an outline of the Loop subdivision algorithm on the data structure above.

Let V = # old vertices, F = # old faces, H=3F=# of half-edges, and #E=number of edges = (H + #boundary edges)/2.

One level of subdivision consists of five steps:

1. Compute New Edge Vertices.
2. Update Original Vertices.
3. Split Faces.
4. Create New Half-edge Info.
5. Update Other Features.
6. Final Step

These are broken down as follows:

Compute New Edge Vertices

1. Since each existing vertex will soon be modified (and originals need to be kept around until all are done), and since new vertices are going to be added per edge, allocate an array NV for all new vertices of size (# old vertices + # edges). When creating new edge vertices the first V positions in the array are skipped so the original vertices can be placed back in the same positions as they are modified.
2. Allocate an array EM (edge map) of integers of size (# half-edges) to store indices mapping half-edges to new vertex indices. Initialize all to -1 to indicate half-edge not mapped yet.
3. For each half-edge h, if EM[h] = -1 then insert a vertex on the edge using the edge split rules. Store the new vertex in an unused slot in NV past the original V, and store the resulting NV index in EM[h]. If h_2=E[h] is not -1, h has a paired half-edge h_2, so store the NV index in EM[h_2] also.

Update Original Vertices

Now move each original vertex to a new position, putting the new vertex into the new vertex array NV, into the same order and position as before to make splitting faces easy. This is done using the vertex modification rules from before. During updating, reduce vertex weights by 1, clamping at 0. New vertices have weight 0. Each vertex stores a half-edge index v_h ending on the vertex, which is used to quickly walk neighboring vertices and determine edge types, as in Figure 4. Given a half-edge index h ending at the vertex, the joined neighbor vertex is VA[FA[Floor[h/3]].vertexIndex[h mod 3]]. Given e_A the next half-edge of interest is found by $e_B = EA[e_A]$ and $e_C = 3Floor\left[\frac{e_B}{3}\right] + ((e_B + 2) \bmod 3)$. With this information the edges and neighboring vertices can be queried rapidly.

The reason for requiring a boundary vertex to be tagged with an incoming crease is so traversal only needs to go in one direction, making the code simpler.

After all updates, change all vertices (new and old) to have a half-edge index of -1, denoting no incoming matching half-edge. These will be filled in during the face splitting.

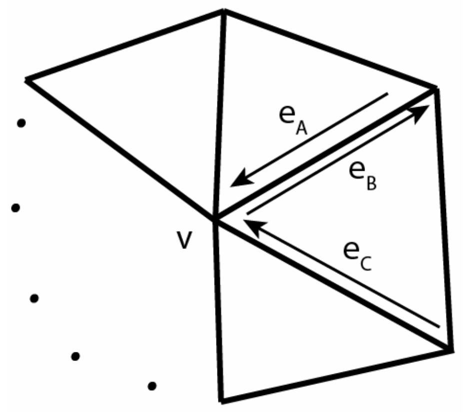

Figure 4 – Vertex neighbors

Split Faces

Each old face will become 4 new faces, split as in Figure 5. Figure 5 shows the original triangle with edge and face orientations, and how this maps to new edge and face orientations, along with the order 0,1,2,3 the new faces are stored.

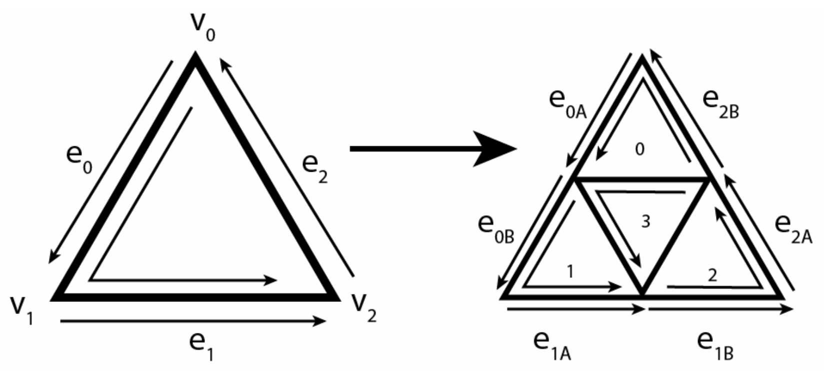

Figure 5 – Face splitting

1. Allocate an array NF for new faces of size 4F.

2. For each face f, with vertex indices v_0, v_1, v_2 lookup the three edge vertex indices as j_0=NV[3f+V], j_1=NV[3f+1+V], and j_2=NV[3f+2+V].

3. Split the faces in the order shown in Figure 5. To NF add faces $\{j_2, v_0, j_0\}, \{j_0, v_1, j_1\}, \{j_1, v_2, j_2\}, \{j_2, j_0, j_1\}$ at positions 4f, 4f+1, 4f+2, 4f+3. This order is important! Each parent half-edge e_k is conceptually split into two descendent half-edges e_{kA} followed by e_{kB}.

4. During the face split tag each vertex (which still has a -1 tag from the previous steps) with an incident half-edge index ending on the vertex, giving preference to an incoming boundary.

Create New Half-edge Info

This step could be merged with the Split Face step, but is separated for clarity. A new list of half-edges is needed, correctly paired and weighted. Create a new half-edge array NE of size 12*F (3 per new face, each old face becomes 4 new faces, giving 12).

Define a function `nIndex(j,type)` to compute new half-edge pair indices, where j is the old half-edge index, and type is 0=A or 1=B, denoting which part of the new half-edge is being matched. This function is

```
function nIndex( j, type)
   /* data table for index offsets - matches new half-edges */
   offsets[] = {3,1,6,4,0,7}
   /* original half-edge pair index */
   op = EV[ei]
   if (op == -1)
      return –1   /* boundary edge */
   /* new position of the split-edges for the face with pair op */
   bp = 12*Floor[op/3]
   /* return the matching new index */
   return bp + offsets[2*(op mod 3) + type]
```

The {3,1,6,4,0,7} array comes from matching half-edges to neighboring half-edges and is dependent on inserting items in arrays as indicated. For each original face index f, do the following:

1. Let b=12*f be the base half-edge index for a set of new half-edges, which will be stored in NE at the twelve indices b through b+11.

2. Store the 12 new half-edge pair indices at b,b+1,…,b+11 in the following order: $\{e_2B, e_0A, b+9, e_0B, e_1A, b+10, e_1B, e_2A, b+11, b+2, b+5, b+8\}$, where e_iT is `nIndex(ei,type)` with `ei` being the edge index and `type` = 0 for $T = A$ and `type` = 1 for $T = B$. These are grouped 3 per face in the order of faces created in Figure 5.

3. In the above 12 entries update the half-edge weights, with descendent half-edges getting the parent half-edge weights –1, clamped at 0. New half-edges with no parent get weight 0.

Update Other Features

Per-vertex colors and per-vertex texture coordinates can be updated during the edge vertex creation and during the vertex re-positioning steps by simple interpolation. Per face per vertex colors and textures coordinates can be interpolated in the steps above also, or can be done as a final step.

Displaced subdivision surface modifications can also be applied here by modifying the current vertex positions using a displacement map.

If the surface is about to be rendered, temporary normals can be computed using standard averaging techniques or by using the exact normals. Exact normals are more appropriate on a surface with vertices at limit points. Normals don't usually need computed until render time.

Final Step

Replace the arrays in the data structure with the new ones, and discard, store, or free the old ones as desired.

Finally, if the mesh is not going to be subdivided further, vertices can be pushed to limit positions, and true normals computed. In a rendering engine where the object is about to be drawn, this is an appropriate step.

Performance Issues

The preceding sections detailed a data structure and algorithm to implement Loop subdivision with extensions. An overview of hardware rendering techniques is below.

Performance Enhancements

There are many places to improve the performance of the algorithm itself, especially if all the features are not needed. If a simple, smooth, closed mesh is all that is needed, all the special cases can be removed, making subdivision very fast.

Implementation tips:

1. Use tables for the $b(n)$ based weights needed, the tangent weights, normal weights, limit position weights, as well as any other items. A given mesh has a maximum valence vertex and all new vertices have valence at most this, making tables feasible.

2. Make the half-edge array spaced out by four entries per face instead of three, allowing many divide by three and mod three operations to be replaced with shifts. This is the traditional space for speed tradeoff.

3. Most interior vertices will have valence 6 and be smooth, so make that code fast, with special cases for the other situations. Most boundary vertices will be regular

with valence 4. Most edges will be weight 0 and connect valence 6 smooth vertices.

4. Tag edges and vertices for whether they are smooth or need special case code, allowing faster decision instead of determining vertex and edge types by walking neighbors. Once a vertex or edge type is determined it is easy to tag descendents.

5. Pre-compute one level of subdivision to isolate the special case vertices, and then at runtime use a simpler version of the algorithm since there are fewer cases. This is a minor speed improvement, and is used for some hardware implementations.

6. A pointer based data structure like a half-edge structure can speed up subdivision at the cost of using more (and likely less contiguous) memory and making reading/writing harder. It is not clear which is really faster unless tests are done.

7. Move per vertex per face parameters to per-vertex when possible. For example, creases require per vertex per face normals, since neighboring faces require different normals along the crease, but once a face has been subdivided the interior smooth new vertices can (and should) all use per-vertex normals.

8. Do multiple subdivision steps at once if desired, storing only the resulting triangles, and not updating all the connectivity info. This is detailed below in the GPU section.

9. Implement adaptive subdivision – having fewer triangles to split after a few steps will speed things up a lot (but will break the simple algorithm operating on the data structure presented).

GPU Subdivision and Rendering

The original plan to present a state-of-the art GPU subdivision renderer was dropped because reviewing the literature demonstrated how fast such articles become obsolete. Instead the focus is putting in one place unified rules for a subdivision scheme. This will assist future hardware and software implementations, making this article useful for a longer period.

For GPU rendering, here is a chronological review of several papers, most of which can be found on the internet. The papers are roughly evenly divided between Catmull-Clark methods, Loop methods, and universal methods.

1. [Pulli96] presents an efficient Loop rendering method. It works by grouping triangles into pairs during a precomputation phase, effectively passing squares and a 1-neighborhood to a rendering function, which then renders the two triangles to an arbitrary subdivision depth.
2. [Bischoff00] presents a very memory efficient and fast Loop rendering solution. The main concept is using multivariate forward differencing to generate triangles

several subdivision levels deep without having to generate the intermediate levels. Rendering is done patch by patch.

3. [Müller00] presents an extension to [Pulli96], and details a triangle paring algorithm and a sliding window method. Details are also presented for adaptive subdivision and crack prevention.
4. [Leeson02] covers a few subdivision methods, and gives an overview of some rendering tips such as hierarchical backface culling.
5. [Bolz02] implements Catmull-Clark subdivision, using a static array to hold the results. The methods are good for SIMD implementation.
6. [Bolz03] implement Catmull-Clark subdivision on a GPU, with special attention to avoiding cracks and pixel dropout caused through floating point errors.
7. [Boubekeur05] presents a general method useful for rendering many types of subdivision surfaces. The main idea is to implement a "refinement pattern" on the GPU, and then each triangle or other primitive passed to the GPU is refined using the pattern.
8. [Bunnell05] and [Shiue05] both implement Catmull-Clark subdivision on a GPU, with ample details.

Fast Subdivision Surface Rendering

A fast subdivision routine suited for a GPU is based on the following observation. For each triangle, upon subdividing, new items (vertices, edges, faces, colors, etc.) are a *linear* combination of a 1-neighborhood of the triangle. Second subdivision items are then linear combinations of first subdivided items, hence a linear combination of the original neighborhood. This is exploited in various ways in the preceding references, and will be explained for a simple case.

A *patch* is single triangle T and the surrounding triangles (those which influence descendent triangles from the triangle T). See Figure 6 for a patch illustration; on the left T is shaded and a 1-neighborhood is included. The second part shows T subdivided once, and a new 1-neighborhood (without all edge lines drawn) is shown.

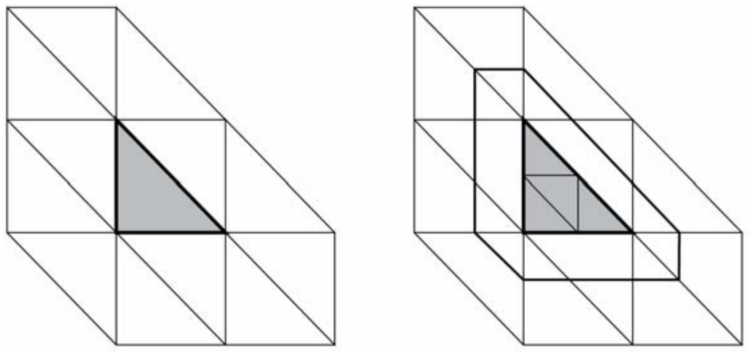
Figure 6 – Subdividing a patch

Assume for the moment there are no creases or boundaries (which can be added back in later). All the subdivision levels beneath T can be generated from linear combinations of existing vertices, so for each level of desired subdivision a mask can be computed in terms of neighboring vertices that outputs all the triangles descended from T, *without needing to compute intermediate levels.* Also connectivity information does not need computed or stored – all that is desired are the vertices of the faces, which naturally fall into a grid arrangement and are suitable for GPU rendering.

Mesh precomputation gathers needed data for each patch, stored per triangle. At render time, a subdivision level is selected, and each patch is passed to a GPU kernel which takes the low resolution triangle, creates subdivided triangles in one pass, and then renders the resulting triangles. In order to incorporate all the features from the article different kernels should be implemented. Alternatively preprocessing could simplify the numbers of cases, resulting in less GPU kernel variations.

A final point is this method may result in pixel dropout or cracks, since neighboring triangles may be evaluated using floating point operations in different orders. This is addressed in [Bolz03] for Catmull-Clark surfaces.

Code

At the time of this writing, the author has an old OpenGL Loop subdivision surface rendering demo on his website. Also there is a Mathematica based prototype of the algorithm and data structure presented. Unfortunately at the time of writing the author does not have a free version of the subdivision algorithm, but hopefully this will change soon. Check www.lomont.org for updates.

Conclusion

This article showed details of how to implement Loop subdivision surfaces with additional features and provides a starting point for the literature on subdivision surfaces. Geometry features such as creases, boundaries, semi-sharp items, and normals were covered, as well as surface tags like colors and textures. Future directions would be to add displaced subdivision surfaces and adaptive subdivision to the algorithm.

References

[Biermann06] Biermann, Henning, Adi Levin, and Denis Zorin, “Piecewise Smooth Subdivision Surfaces with Normal Control,” Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, pp. 113-120, 2006. http://mrl.nyu.edu/publications/piecewise-smooth.

[Bischoff00] Bischoff, Stephan, Leif Kobbelt, and Hans-Peter Seidel, “Towards hardware Implementation of Loop Subdivision,” Eurographics SIGGRAPH Graphics Hardware Workshop 2000 Proceedings, http://www-i8.informatik.rwth-aachen.de/.

[Bunnell05] Bunnell, Michael, “Adaptive Tesselation of Subdivision Surfaces with Displacement Mapping,” GPU Gems 2, 2005, pp. 109- 122.

[Bolz02] Bolz, Jeffery, and Peter Schröder, “Rapid Evaluation of Catmull-Clark Subdivision Surfaces,” Web3d 2002 Symposium, http://www.multires.caltech.edu/pubs/fastsubd.pdf.

[Bolz03] Bolz, Jeffery, and Peter Schröder, “Evaluation of Subdivision Surfaces on Programmable Graphics hardware,” http://www.multires.caltech.edu/pubs/GPUSubD.pdf

[Boubekeur05] Boubekeur, Tamy, and Christophe Schlick, “Generic Mesh Refinement on GPU,” ACM SIGGRAPH/Eurographics Graphics Hardware, 2005, http://iparla.labri.fr/publications/2005/BS05/.

[Catmull78] Catmull, E., and J. Clark, “Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes,” Computer Aided Design 10, 6(1978), pp. 350-355.

[DeRose98] DeRose, Tony, Michael Kass, Tien Truong, “Subdivision Surfaces in Character Animation,” International Conference on Computer Graphics and Interactive Techniques , SIGGRAPH 1998, pp 85-94, http://www.cs.rutgers.edu/~decarlo/readings/derose98.pdf.

[Hoppe94a] Hoppe, Huges, “Surface Reconstruction from Unorganized Points,” PhD Thesis, University of Washington, 1994, http://research.microsoft.com/~hoppe/.

[Hoppe94b] Hoppe, Huges, Tony DeRose, Tom DuChamp, et. al., “Piecewise Smooth Surface Reconstruction,” Computer Graphics, SIGGRAPH 94 Proceedings, 1994, pp. 295-302, http://research.microsoft.com/~hoppe/.

[Lee00] Lee, Aaron, Henry Moreton, and Huges Hoppe, “Displaced Subdivision Surfaces,” SIGGRAPH 2000, pp. 95-94, http://research.microsoft.com/~hoppe/.

[Lee98] Lee, Aaron, Win Sweldens, et. at., “MAPS: Multiresolution Adaptive Parameterization of Surfaces,” Proceedings of SIGGRAPH 1998, http://www.multires.caltech.edu/pubs/.

[Leeson02] Leeson, William, “Subdivision Surfaces for Character Animation,” Games Programming Gems 3, 2003, pp. 372-383.

[Levin99] Levin, Adi, “Interpolating Nets of Curves By Smooth Subdivision Surfaces,” Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, 1999, http://www.math.tau.ac.il/~levin/adi/paper7.htm.

[Loop87] Loop, Charles, “Smooth Subdivision Surfaces Based on Triangles,” Master’s Thesis, University of Utah, Dept. of Mathematics, 1987, http://research.microsoft.com/~cloop/thesis.pdf.

[Müller00] Müeller, Kerstin, and Sven Havemann, “Subdivision Surface Tesselation on the Fly using a Versatile Mesh Data Structure,” Comput. Graph. Forum, Vol. 19, No. 3, 2000, available from http://citeseer.ist.psu.edu/.

[Pulli96] Pulli, Kari, and Mark Segal, “Fast Rendering of Subdivision Surfaces,” roceedings of 7th Eurographics Workshop on Rendering, pp. 61-70, 282, Porto, Portugal, June 1996, http://graphics.stanford.edu/~kapu/.

[Schweitzer96] Schweitzer, J. E., “Analysis and Application of Subdivision Surfaces,” PhD Thesis, University of Wahsington, Seattle, 1996, available from http://citeseer.ist.psu.edu/.

[Severn06] Severn, Aaron, and Faramarz Samavati, “Fast Intersections for Subdivision Surfaces,” In International Conference on Computational Science and its Applications, 2006, http://pages.cpsc.ucalgary.ca/~samavati/papers/FastIntersections.pdf.

[Shiue05] Shiue, L.-J., Ian Jones, and Jörg Peters, “A Realtime GPU Subdivision Kernel,” ACM SIGGRAPH Computer Graphics Proceedings, 2005, http://www.cise.ufl.edu/research/SurfLab/papers/.

[Stam99] Stam, Jos, "Evaluation of Loop Subdivision Surfaces", SIGGRAPH 99 Course Notes, 1999, http://www.dgp.toronto.edu/people/stam/.

[Vlachos00] Vlachos, Alex, Jörg Peters, Chas Boyd, and Jason Mitchell, “Curved PN Triangles,” ID3G 2001, http://www.cise.ufl.edu/research/SurfLab/papers/.

[Warren95] Warren, J., “Subdivision Methods for Geometric Design,” Unpublished manuscript, November 1995.

[Wu04] Wu, Xiaobin, and Jörg Peters, “Interference Detection for Subdivision Surfaces,” EUROGRAPHICS, 2004. Vol. 23, 3, http://www.cise.ufl.edu/research/SurfLab/papers/.

[Zorin96] Zorin, Denis, Peter Schröder, Wim Sweldens, “Interpolating Subdivision for Meshes with Arbitrary Topology,” Proceedings of SIGGRAPH 1996, ACM SIGGRAPH, 1996, pp. 189-192, http://www.multires.caltech.edu/pubs/interpolation.pdf.

[Zorin97] Zorin, Denis, Peter Schröder, Wim Sweldens, “Interactive Multi-resolution Mesh Editing,” CS-TR-97-06, Department of Computer Science, Caltech, January 1997, http://graphics.stanford.edu/~dzorin/multires/meshed/.

[Zorin00] Zorin, Denis, and Peter Schroeder, “Subdivision for Modeling and Animation,” Technical Report, ACM SIGGRAPH Course Notes 2000, http://mrl.nyu.edu/~dzorin/sig00course/.

TODO – make sure all mesh items updated and finished
TODO – remove - END OF FILE