Complexity Reduction of Catmull-Clark/Loop Subdivision Surfaces

Eskil Steenberg

The Interactive Institute, P.O. Box 24081, SE 104 50 Stockholm, Sweden
eskil.steenberg@interactiveinstitute.se

Abstract

By applying a filter this algorithm can reduce the number of polygons generated by subdividing a mesh dynamically. This algorithm is designed especially for real-time engines where the geometrical complexity is critical. It also avoids edge cracks and is generally more efficient than a general-purpose polygon reduction algorithm.

Keywords: Subdivision complexity reduction. Real-Time, Loop, Catmull-Clark, Hardware.

1. Introduction

While subdivision surfaces are quickly becoming the primitive of choice for modeling and animation, they have yet to become widely used in the real-time simulation and game communities. Even though subdivision surfaces by nature are recursively generated and therefore can be generated in different levels of detail (LOD) they are limited in the way they add complexity. For each generation of Catmull-Clark[1] or Loop[2] subdivision that is generated the amount of geometry quadruples making the LOD steps fairly large.

In theory an extended Catmull-Clark subdivision scheme with creases appears as the ultimate CGI primitive, it is compatible with polygons and NURBS, making it backwards compatible with old art work, it has been properly evaluated[3,4], it is supported in various commercial modeling and animation tools and it is a joy to model with.

Unfortunately our general conclusion is that Catmull-Clark subdivision surfaces are not useful in real time engines unless there is some geometry-reduction algorithm in place. The performance hit taken by the quadrupling face count for each iteration of subdivision can not be justified by the added smoothness of the surface.

The main problem is that when a mesh is subdivided all polygons are equally subdivided regardless of whether extra detail is needed at that particular location or not. The algorithm does not take into account proximity to the viewpoint, curvature, frustum clipping or polygon size, something that many other LOD algorithms consider [7,8,9,10,11,12]. This makes subdivision surfaces not too well suited for environments with large objects where local LOD is needed. This problem becomes even more apparent if the subdivision scheme includes some sort of crease algorithm delivering meshes with many flat areas that don't need to be divided [14].

It becomes apparent that what is needed is some type of local LOD. One possible approach would be to apply a standard polygon reduction algorithm on the subdivided mesh but this would not be very efficient. Since we have access to the control mesh we can analyze it rather than the subdivided mesh to get some higher level control of the surface complexity.

2. Understanding Optimization

In order to understand how to optimize an algorithm we need to understand how the mesh will be used. It is a common misconception that in a real time engine everything has to be computed for each frame, whereas in reality only a small portion of a scene changes for every frame making caching very rewarding. The highest priority is given to the actual drawing of the

frame. Second comes the animation that should be updated every frame, but in extreme circumstances it can be dropped to every second or third frame. The change of an objects LOD is more rare and only occurs when there is a significant change to the relation of the camera and the object. Topological changes to the control mesh may never occur and or occur very rarely.

This leads us to write a "kernel" in our engine that much like a multitasking operating system can prioritize and schedule tasks that needs to be computed. It can also maximize the use of the 3D hardware and multiple processors.

The biggest speed gain to be made on modern processors is to keep algorithms cache coherent, this means that an algorithm should not access memory randomly, but to work through it from one end to the other. This means that static lists are faster than linked list and that pointer references to other parts of memory are generally slow to follow. This makes many of the conventional mesh optimization algorithms very hard to implement since they are dependent on the ability to split and remove polygons and to compute each polygons relation to its neighbors.

2.1 The Geometry Pipeline

Our geometry pipeline will therefore be in four steps: The first step is the one that subdivides the control mesh in to a finer polygon mesh and creates relational data between the control vertices and the vertices on the subdivided surface. This part of the algorithm is in fact a conventional subdivision algorithm and will therefore not be discussed in this paper [1, 2].

The second stage analyses the mesh and creates a reduced version of it by applying a filter to it.

The third stage takes the topology data, and the vertex relation data along with the positions of the control vertexes and computes the animation of the object. The relational data will be explained later in section four.

The fourth stage renders the mesh.

So to get a new image we only redo the last step, if we need to animate the object we redo the last two steps, if we need a new level of detail we re compute the three last steps and only in the rare occasion of a topological change do we need to recompute all of the steps including the first and most time consuming step.

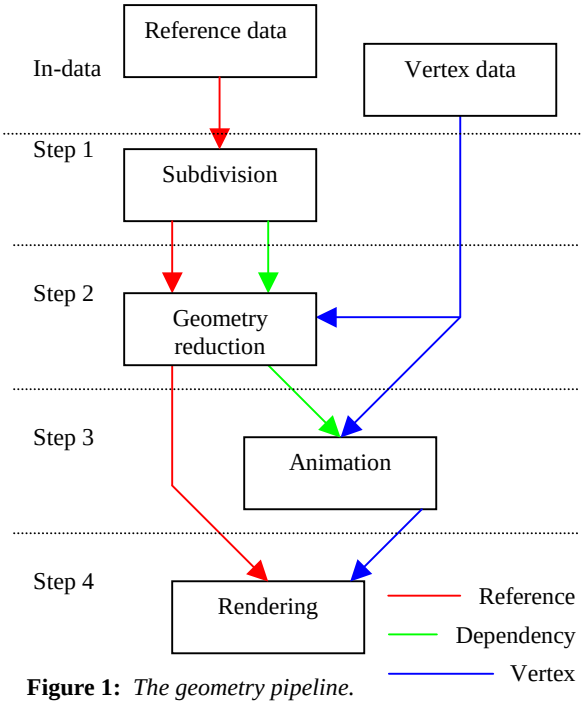


Figure 1: *The geometry pipeline.*

3. Tessellation Selection - Step II

The simple approach is of course to divide different polygons different number of times, giving you a flexible way of choosing where you want your complexity. Pixars Photo Realistic Renderman does this to create sub-pixel-sized quadrilaterals. Unfortunately cracks will appear in the edges between two differently tessellated polygons. In the case of a rendering architecture like Rendermans REYES[5] this is a limited problem since the cracks are very small, but in a real-time orientated engine the cracks are not acceptable due to the larger polygons. So our algorithm must be able to subdivide different edges differently in order to fit each polygon with its neighbors.

We choose to compute the level of tesselation per edge, to do this we use both the vertexes in the ends of the edge and one vertex in the middle of the edge. If we measure the distance between the edge point and the mid points between the two end vertexes and divide it by the length of the edge, we will obtain a value that represent the curvature of the edge (see Figure 2). We can also compute the distance between the edge point and the view position to create a view-dependent tessellation level. The great benefit with this is that since the tessellation selection algorithm uses only data

that is shared with the neighboring polygon there is no need to store the result of the computations to match it with the tessellation of the neighboring polygon. If the neighboring polygon uses the same input it will obtain the same result and will therefore be tessellated to match its neighboring polygon. This approach side-steps the process of having two neighboring polygons "agree" on the level of tessellation needed, a process that can be difficult to implement and not very cache friendly.

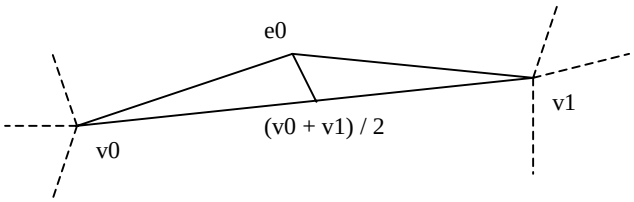


Figure 2: *The tessellation selection algorithm.*

3.1 The Tessellation Table

Our suggested approach, to first subdivide the mesh a given number of times and then apply a filter that picks out a different amount of detail in different edges so that each edge can be tessellated individually, makes it possible to precompute this filter in the form of a look up table that contains two arrays of data per polygon. One holds the indexes of the vertexes needed in the filtered polygon and the other holds the references describing how to bind those vertexes in to polygons. (See listing 1). This means that we need one table element, for each possible combination of how the edges in a polygon can be tessellated. In our particular implementation we have implemented two sets of tables, one for quadrillions and one for triangles since our engine is a Catmull-Clark / Loop hybrid. If you want to have a more general solution for n-sided polygons you can choose first to tessellate all polygons using one generation of Catmull-Clark to turn all polygons in to quadrilateral.

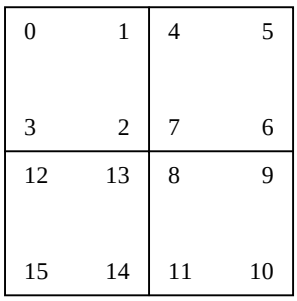


Figure 3 *Step I*

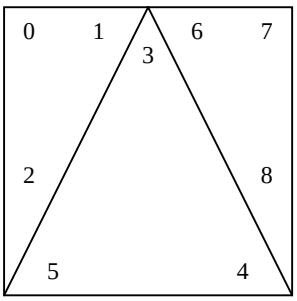


Figure 4 *Step II*

If we consider the reference data for the tessellated quad in fig 3 as an array of 16 elements, and we want to create the less tessellated polygon in fig 4. The new tessellated polygon will only need 5 out of the 9 vertexes. In order to acquire them we read the first vector in the look-up table element containing the entries 0, 1, 15, 10 and 5. We can then copy the vertex data referenced to in the original reference array elements 0, 1, 15, 10 and 5.

Now that we have a reduced vertex array we need a reduced reference. This reference can simply be copied from the look up table element s other vector in this case containing the data 0, 1, 2, 1, 3, 2, 1, 4 and 3. (See Listing 1).

One important criteria for this algorithm is that the subdivision algorithm is able to backtrack in a repeatable way how polygons were created and from what control polygons they origin. For each time we subdivide the polygons we get four times as many polygons, if we choose to put those four new polygons in order we can easily back track the origin of them. So if we for example choose to tessellate a subdivision mesh three times we will know that every 4^3:rd polygon entry will origin from the same control mesh polygon since each control polygon has been divided in to 4^3 (64) polygons. This makes it possible to make a table that can easily find the vertexes that are needed. In our implementation tables have been generated for up to six levels of subdivision taking almost a minute on a standard pc.

```
vertex_count = 0;
for(i = 0; i < control_polygon_count; i++)
{
  element = get_table_element();
  for(j = 0; j < element->reference_count; j++)
    reduced_reference[k++] = element-
>reference[j] +
    vertex_count;
  for(j = 0; j < element->vertex_count; j++)
  {
    vertex_index = original_reference[element->
    vertex_index[j] + i * pow(4, base_level +
1)];
    new_vertex[vertex_count].x =
     old_vertex[vertex_index].x;
    new_vertex[vertex_count].y =
     old_vertex[vertex_index].y;
    new_vertex[vertex_count].z =
     old_vertex[vertex_index].z;
    vertex_count++;
  }
}
```

Listing 1. *The inner loop that copies the subdivided data in to the reduced data.*

Exactly how the tables are generated is an implementation detail that will not be covered in this paper but it is important to note that the middle of the control polygons should be divided roughly the same number of times as the edges.

Since Catmull-Clark subdivision surfaces are based on quads you want your subdivision scheme to output quads while your 3D hardware expects triangles. The tessellation tables we have implemented outputs triangle but it is of course not necessary.

4. Animation - Step III

In the first step we do not handle the actual positions of any geometry but only how the vertexes in the subdivided mesh relates to the control mesh. So instead of computing the position of a vertex, we store references to the control vertexes that influence that vertex and how much they influence(see Table 1). This means that at any time we can re-compute the position of a vertex on the surface by weighting in the control vertices (see Listing 2). All animation is therefore preformed only on the control mesh that in turn influences the subdivided surface.

Reference	Weight
23	0.4375
56	0.4375
34	0.0625
56	0.0625
45	0.0625
11	0.0625

Table 1. *The data for a vertex may look like this (this is the data produced by an edge vertex between two quadrillions).*

This idea of storing the relationship between control vertexes and surface vertexes has many advantages if implemented in hardware. First of all it is a very simple algorithm that can be used for a variety of surface types like Bezers, trimmed NURBS, Catmull-Clark, Loop, butterfly subdivision surfaces, edge collapse based polygon reduction algorithm [8,9]. It can also be used to create hardware-accelerated displacement mapping by weighting in normals. But the biggest gain is that it makes it possible to upload the relational data to the 3D hardware, and then when the animation occurs only send the new control vertexes. This will drastically reduce the bandwidth needed to perform complex animations. Sending only 500 control points over the bus can animate a 100.000 polygon character.

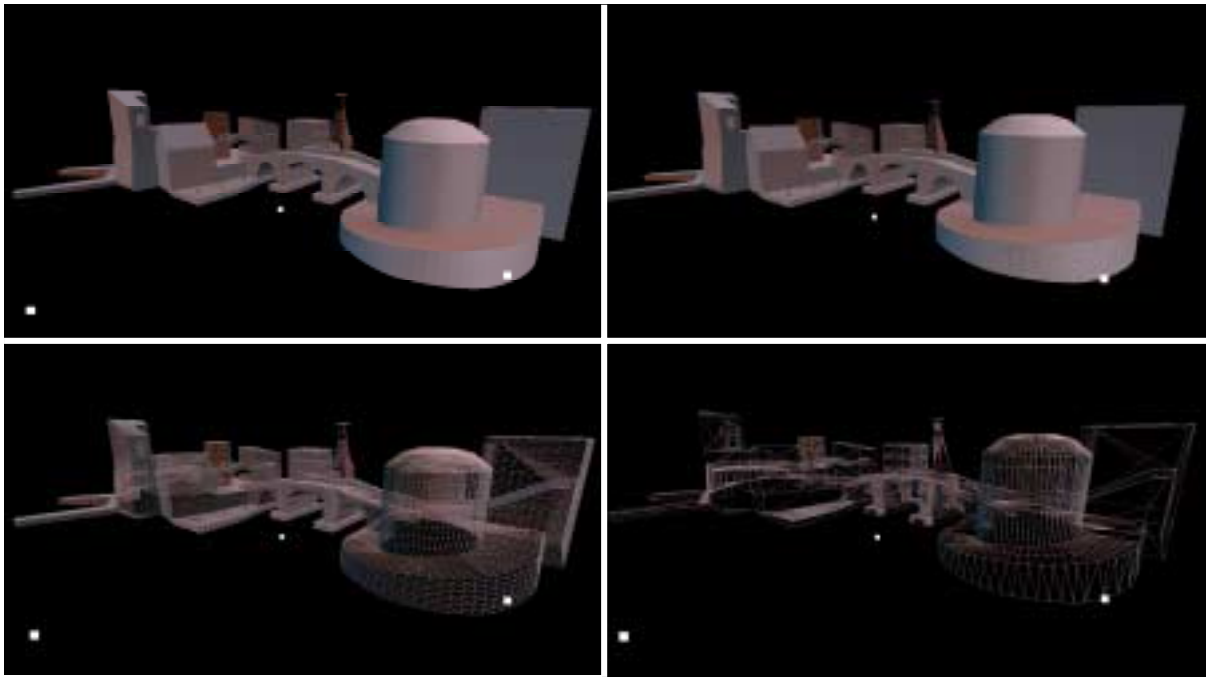
In order to test this we have implemented our own low-level experimental 3D API. This API called NGL features a stack based shader system and a programmable geometry pipeline.

```
for(i = 0; i < vertex_count; i++)
{
  x = 0;
  y = 0;
  z = 0;
  for(j = 0; j < *influence_list_length; j++)
  {
    index = *index_array++;
    value = *value_array++;
    x += value * control_vertex_array[index].x;
    y += value * control_vertex_array[index].y;
    z += value * control_vertex_array[index].z;
  }
  surface_vertex_array[i].x = x;
  surface_vertex_array[i].y = y;
  surface_vertex_array[i].z = z;
  influence_list_length++;
}
```

Listing 2. *The inner loop that weights the control vertices to the surface vertices.*

5. Results / Future work

In our implementation we have seen vast improvement in efficiency especially on objects with few curved areas(See Figures 5-6). The largest shortcoming of this algorithm is that flat areas with curved edges tend to be over tessellated. This is something we intend to combat by having a separate set of look up tables for flat polygons. Another potential problem is that the algorithm can never reduce the mesh below its control mesh polygon count. Still it produces very good results especially with architectural objecs. The biggest features with this algorithm is of course that it is fast, contains no iterative searching and that all data can be stored in arrays since there are no insert/delete operations, something that 3D hardware APIs such as OpenGL like. The biggest problem is that the algorithm is so large and contains so many steps that it becomes very hard to implement and debug. In the future we would like to try to instead of generating a look-up table generate compliable code that would contain the tessellation information to see if there is a possible speed gain.



The original mesh subdivided 3 levels contains 158272 polygons. All images are flat shaded. The control mesh consists of roughly 1200 control polygons and has a number of creased features.

The same mesh in a reduced state only contains 11821 polygons, about 7.5% of the original geometry complexity. Note that flat areas have been reduced down to a minimum while the curved areas are still tessellated.

6. Acknowledgments

We would like to thank Emil Brink, Oskar Wahlberg, Mark Ollila, the Interactive Institute and everyone who has supported the Verse project and us.

7. References

[1] Catmull E. and J. Clark "Recursively Generated B-spline Surfaces on Arbitrary Topology Meshes". Computer Aided Design 1978

[2] C. Loop "Smooth Subdivision Surfaces Based on Triangles" Masters Thesis, University of Utah, 1987.

[3] J. Stam, "Exact Evaluation of Catmull-Clark Subdivision Surfaces at Arbitrary Parameter Values", SIGGRAPH'98 Proceedings, pages 395-404.

[4] J. Stam, "Evaluation of Loop Subdivision Surfaces", SIGGRAPH'99 Course Notes, 1999.

[5] R. L. Cook, L. Carpenter, and E. Catmull, "The Reyes Image Rendering Architecture" Computer Graphics, Volume 21, Number 4, July 1987.

[6] S. Junkins and A. Hux "subdividing Reality: Employing Subdivision Surface for Real-time Scalabel 3D" Game Developers Conference 2000, Proceedings page 287-300

[7] D. Luebke and C. Eickson. "View Dependent Simplification of Arbitrary Polygonal Enviorments" SIGGRAPH 1997, Proceedings page 199-207

[8] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle. "Mesh optimization" SIGGRAPH 1993 Proceedings pages 19-26.

[9] H. Hoppe. "Progressive meshes" SIGGRAPH 1996 Proceedings pages 99-108.

[10] H. Hoppe. "View-dependent Refinement of Progressive Meshes" SIGGRAPH 1997 Proceedings pages 189-198.

[11] S. Melax "Asimple, Fast and Efective Polygon Reduction Algorithm" Game Developer November 1998

[12] A. Certain, J. Popovic, T. DeRose T. Duchamp, D. Salesin and W. Stuetzle "interactive multiresolution Surface Viewing" SIGGRAPH 1996 Proceedings pages91-98

[13] T. DeRose, M. Kass and T. Truong "Subdivision Surfaces in the Making of GerisGame" SIGGRAPH 99 Subdivision course notes Chapter 10.

[14] H. Hoppe, T. DeRose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer and W. Stuetzle. "Piecewise Smooth Surface Reconstruction". SIGGRAPH 1994 Proceedings pages 295-302.