A Simple, Fast, and Effective Polygon Reduction Algorithm

by Stan Melax

If you're a game developer, there's a good chance that 3D polygonal models are part of your daily life and that you're familiar with concepts such as polygons per second, low-polygon modeling, and levels of detail. You probably also know that the objective of a polygon reduction algorithm is to take a high-detail model with many polygons and generate a version using fewer polygons that looks reasonably similar to the original. In addition to talking about what polygon reduction is and why it is useful, this article explains one method for achieving it. Before going any further, I suggest you download my application, BUNNYLOD.EXE, which demonstrates the technique that I'll explain. You can find it on the *Game Developer* web site.

Motivation

Before diving into a sexy 3D algorithm, you may be asking yourself if you really care. After all, there are commercial plug-ins and tools that reduce polygons for you. Nonetheless, there may be reasons why you want to implement your own reduction algorithm.

- The results of your polygon-reduction tool may not meet your specific needs, and you would like to build your own.
- Your current polygon-reduction tool may not produce the morph information that you require for smooth transitions between different levels of detail.
- You want to automate your production pipeline so that the artist has to create only one reasonably detailed model, and the game engine does the rest.
- You're creating a VRML browser, and you want to provide a menu option for reducing those huge VRML files placed on the Web by supercomputer users who didn't realize the frame rate would be slower on a home PC.
- Special effects in your game modify the geometry of objects, bumping up your polygon count and requiring a method by which your engine can quickly reduce polygon counts at run time.

Stan Melax is researching interactive 3D techniques and algorithms for his Ph.D. in computer science at the University of Alberta. He is also the Director of Technology at Bioware, where he had worked on SHATTERED STEEL and is now implementing cool stuff for their next 3D titles. He can be contacted via e-mail at melax@cs.ualberta.ca.

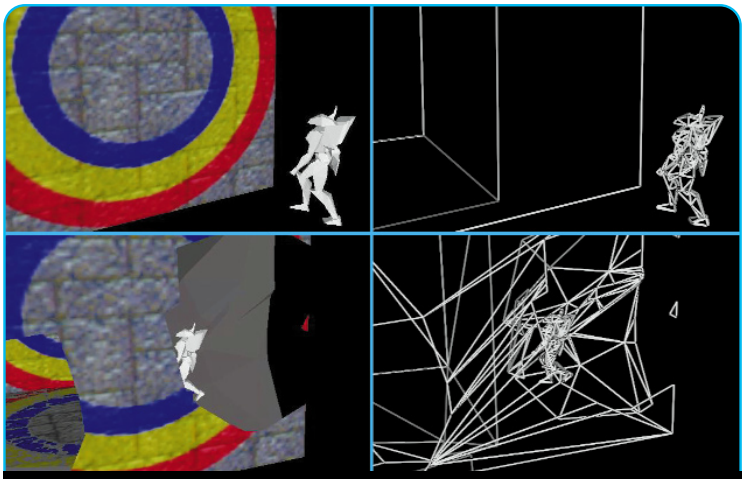


FIGURE 1. *Effect of Boolean operations on polygon count.*

Still not convinced? Figure 1 shows a concrete example of an instance in which a game engine requires polygon reduction capabilities.

At Bioware, I implemented real-time Boolean operations and used them in a game prototype that we developed to impress our publisher. Players could shoot and blast arbitrary chunks out of a solid object wherever they decided to point the gun. Modifying the game environment where the bullets impact produces much more stunning results than the typical "place pipe bomb here" technique, in which the game world only changes in a predetermined manner. Unfortunately, repeated use of Boolean operations performed on polygonal objects generates lots of additional triangles, as you can see in Figure 1. Many of these additional faces are small or splinter triangles that don't contribute to the visual quality of the game — they just slow it down. The situation demanded run-time polygon reduction, so I began my quest to find an algorithm that would do this efficiently.

Collapsing Edges

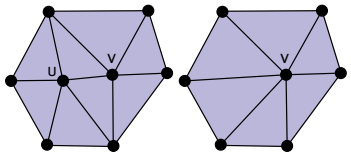
Rather than attacking this problem all by myself, I studied polygon reduction with some other people at the University of Alberta Graphics Lab. (It helps to work with a team in order to figure out how the different algorithms work and which technology is appropriate for which task.) A lot of research has gone into this subject recently, and most of the better techniques are variations of the progressive meshes algorithm by H. Hoppe (see "For Further Info"). These techniques reduce a model's complexity by repeated use of the simple edge collapse operation, shown in Figure 2.

In this operation, two vertices *u* and *v* (the edge *uv*) are selected and one of them (*u*) is "moved" or "collapsed" onto the other (in this case, *v*). The following steps implement this operation:

1. Remove any triangles that have both *u* and *v* as vertices (that is, remove triangles on the edge *uv*).
2. Update the remaining triangles that use *u* as a vertex to use *v* instead.
3. Remove vertex *u*.

This process is repeated until the desired polygon count is reached. At each step, one vertex, two faces, and three edges are usually removed. Figure 3 shows a simple example.

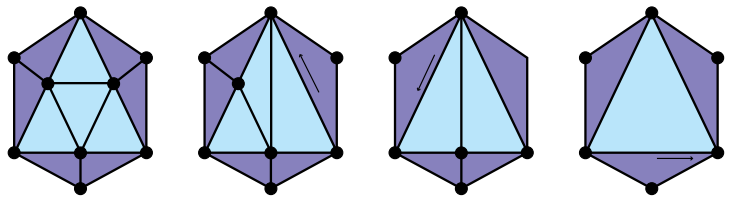
FIGURE 2. *Edge collapse.*



Selecting the Next Edge to Collapse

The trick to producing good low-polygon models is to select the edge that, when collapsed, will cause the smallest visual change to the model. Researchers have proposed various methods of determining the "minimal cost" edge to collapse at each step. Unfortunately, the best methods are very elaborate (as in, difficult to implement) and take too long to compute. Motivated to find a way to reduce polygons during run time in a game, I performed many experiments and eventually developed a simple and blazingly fast approach for this selection process that generates reasonably good low-polygon models.

FIGURE 3. *Polygon reduction via a sequence of edge collapses.*

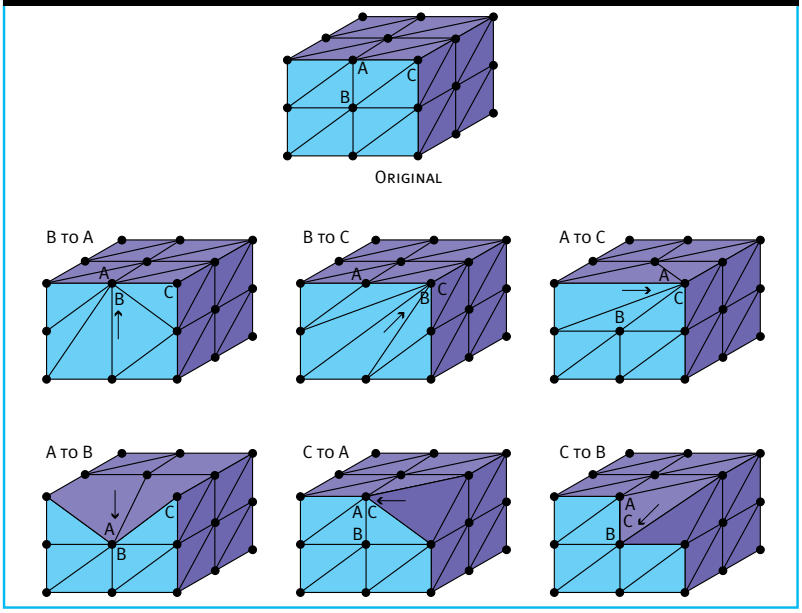


EQUATION 1. *The edge cost formula.*

$$\text{cost}(u,v) = \|u-v\| \times \max_{f \in Tu} \left\{ \min_{n \in Tuv} \left\{ \left(1 - f.normal \bullet n.normal\right) \div 2 \right\} \right\}$$

where *Tu* is the set of triangles that contain *u* and *Tuv* is the set of triangles that contain both *u* and *v*.

FIGURE 4. *Good and bad edge collapses.*



Obviously, it makes sense to get rid of small details first. Note also that fewer polygons are needed to represent nearly coplanar surfaces while areas of high curvature need more polygons. Based on these heuristics, we define the cost of collapsing an edge as the length of the edge multiplied by a curvature term. The curvature term for collapsing an edge *uv* is determined by comparing dot products of face normals in order to find the triangle adjacent to *u* that faces furthest away from the other triangles that are along *uv*. Equation 1 shows the edge cost formula in more formal notation. The specific details can also be found in the source code (which you can dowlnoad from *Game Developer*'s web site).

You can see that this algorithm balances curvature and size when determining which edge to collapse. Note that the cost of collapsing vertex *u* to *v* may be different than the cost of collapsing *v* to *u*. Furthermore, the formula is effective for collapsing edges along a ridge. Although the ridge may be a sharp angle, it won't matter if it's running orthogonal to the edge. Figure 4 illustrates this concept. Clearly, vertex B, sitting in the middle of a flat region, can be collapsed to A or C. Corner vertex C should be left alone. It would be bad to move vertex A, sitting along the top ridge, onto interior vertex B. However, A could be moved (along the ridge) onto C without affecting the overall shape of the model.

If you're implementing your own reduction algorithm, you may wish to experiment with this equation in order to meet your needs. For example, in the case of an animating mesh, you might want to develop a formula that will look at more than just one keyframe when computing the cost of a potential edge collapse. If quality is more important to you than the reduction algorithm's execution time, then you should consider using Hoppe's energy function. We've added our own extensions to deal with texture coordinates, vertex normals, border edges, and surface discontinuities such as texture seams.

Results

The effectiveness of a polygon reduction algorithm is best demonstrated by showing a model before and after it has been simplified. Most research papers demonstrate their results using highly tessellated models in the neighborhood of 100,000 polygons, reducing them to 10,000 polygons. For 3D games, a more appropriate (and challenging) test of an algorithm is how it demonstrates its prowess by generating models that use only a few hundred polygons.

For instance, Figure 5 shows a bunny model taken from a VRML file created by Viewpoint Datalabs. The initial version (left) of the model contains 453 vertices and 902 polygons. Reductions to 200 (center) and 100 (right) vertices are shown. Hopefully, you'll agree that the models look reasonably good given the number of polygons used in each image. Figure 6 shows the consequences of not selecting the right edge to collapse at each step. In this case, edges were chosen randomly.

After completing animal testing, we began human clinical trials for the algorithm. Figure 7 shows three versions — at 4,858; 1,000; and 200 vertices — of a

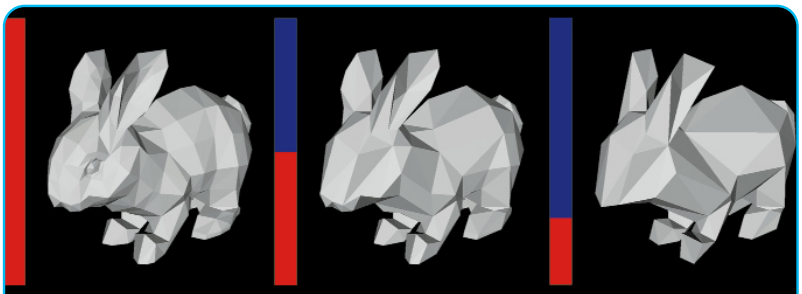


FIGURE 5. *Bunny model at (left to right) 453, 200, and 100 vertices.*

FIGURE 6. *Random edge selection (200 vertex version).*

female human model made by Bioware. (From Euler's formula, we know that the polygon counts are roughly double these numbers.) Once again, these images are shown with flat shading so you can see the difference in the meshes. When smooth shading and textures are applied, the differences are less apparent.

Practical Application

Our initial goal was modest: we wanted to find a way to get rid of a few excess polygons caused by too many Boolean operation effects. However, after developing the reduction algorithm and noticing better-than-expected results on actual models, we decided that the technique was good enough to generate the level of detail (LOD) models for the game engine. An improved version of this basic algorithm has since been incorporated into Bioware's 3D graphics engine, Omen. Now, for many game objects, our artists only have to create one detailed model. A preprocessing step does the polygon reduction. Then, when the frame rate falls below a predefined threshold or an object is to be rendered in the distance, a lower polygon version is used instead. Being able to make these choices at run time increases the scalability of a game. The game adapts itself to the horsepower of the system on which it's running.

Implementation Details

This algorithm only works with triangles. Nothing is lost by this limitation; polygons with more sides are easily triangulated if necessary. In fact, many applications use triangles exclusively.

Most data structures for storing polygonal objects use a list of vertices and a list of triangles that contain indices into the vertex list. For example,

```
Vector vertices[];
class Triangle {
   int v[3];   // indices into vertex list
} triangles[];
```

The `Indexed Face Set` node data type used in VRML is another example of this type of data structure. When two

LISTING 1. *The enhanced data structure.*

```
class Triangle {
   public:
          Vertex *   vertex[3];// the 3 points that make this tri
          Vector     normal;   // orthogonal unit vector
                     Triangle(Vertex *v0,Vertex *v1,Vertex *v2);
                     ~Triangle();
          void       ComputeNormal();
          void       ReplaceVertex(Vertex *vold,Vertex *vnew);
          int        HasVertex(Vertex *v);
};
class Vertex {
   public:
          Vector     position; // location of this point
          int        id;       // place of vertex in original list
          List<Vertex *>       neighbor; // adjacent vertices
          List<Triangle *>     face;     // adjacent triangles
          float      cost;     // cached cost of collapsing edge
          Vertex *   collapse; // candidate vertex for collapse
                     Vertex(Vector v,int _id);
                     ~Vertex();
          void       RemoveIfNonNeighbor(Vertex *n);
};
List<Vertex *>       vertices;
List<Triangle *>     triangles;
```

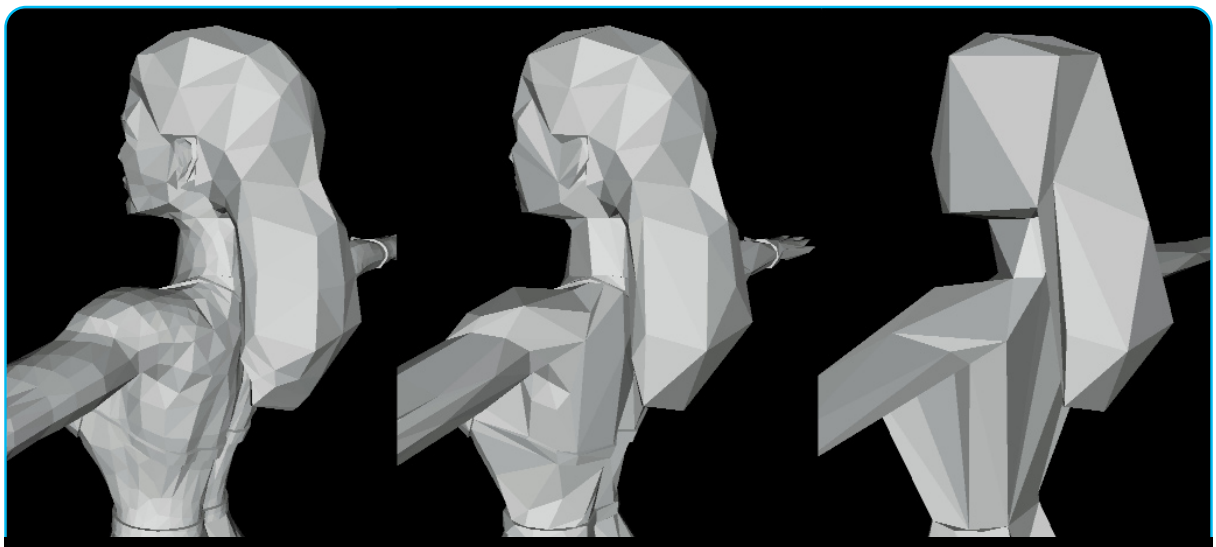


FIGURE 7. *Female human model showing 100 percent of the original polygons (left), 20 percent of the original polygons (center), and 4 percent of the original polygons (right).*

triangles on an object meet at the same vertex, they'll have the same index (so they share the same entry in the vertex list).

We've enhanced this data structure as required by our polygon reduction algorithm. One major improvement is that we now have access to more information than just which vertices each triangle uses — we also know which triangles each vertex bounds. Furthermore, for each vertex, we have direct access to its neighboring vertices (which gives us the edges). Listing 1 shows the enhanced data structure.

Member functions such as `ReplaceVertex()` have been added to perform edge collapses during polygon reduction. Consistency of this data must be maintained as vertices and triangles are added, deleted, or replaced. The constructors, destructors, and member functions contain code to keep things in order. We cache face normals because they are frequently used by the edge selection formula. In order to save us the effort of recalculating these costs, the best edge and its cost is cached for each vertex. The implementation of the member functions is fairly straightforward, so I haven't included it in this article. If you're interested, simply examine this algorithm's source code on the *Game Developer* web site. Listing 2 contains the code for determining edge costs and doing the edge collapse operation.

Performing polygon reduction is easy given these functions. Simply initialize the vertex and triangle lists with the object's geometry, and then do something like this:

```
while(vertices.num > desired) {
    Vertex *mn = MinimumCostEdge();
    Collapse(mn,mn->collapse);
}
```

The demo, BUNNYLOD.EXE, doesn't use this simple loop. Instead it creates an additional data structure for the animation.

Making Better Use of the Data

Rather than throwing away information about triangles and vertices that have been removed, this information can be preserved so that a

LISTING 2. *Determining the edge costs and performing the edge collapse operation.*

```
float ComputeEdgeCollapseCost(Vertex *u,Vertex *v) {
   // if we collapse edge uv by moving u to v then how
   // much different will the model change, i.e. the "error".
   float edgelength = magnitude(v->position - u->position);
   float curvature=0;

   // find the "sides" triangles that are on the edge uv
   List<Triangle *> sides;
   for(i=0;i<u->face.num;i++) {
      if(u->face[i]->HasVertex(v)){
         sides.Add(u->face[i]);
      }
   }
   // use the triangle facing most away from the sides
   // to determine our curvature term
   for(i=0;i<u->face.num;i++) {
      float mincurv=1;
      for(int j=0;j < sides.num;j++) {
         // use dot product of face normals.
         float dotprod =
            u->face[i]->normal ^ sides[j]->normal;
         mincurv = min(mincurv,(1-dotprod)/2.0f);
      }
      curvature = max(curvature,mincurv);
   }
   return edgelength * curvature;
}

void ComputeEdgeCostAtVertex(Vertex *v) {
   if(v->neighbor.num==0) {
      v->collapse=NULL;
      v->cost=-0.01f;
      return;
   }
   v->cost = 1000000;
   v->collapse=NULL;
   // search all neighboring edges for "least cost" edge
   for(int i=0;i < v->neighbor.num;i++) {
      float c;
      c = ComputeEdgeCollapseCost(v,v->neighbor[i]);
      if(c < v->cost) {
         v->collapse=v-neighbor[i];
         v->cost=c;
      }
   }
}

void Collapse(Vertex *u,Vertex *v){
   // Collapse the edge uv by moving vertex u onto v
   if(!v) {
      // u is a vertex all by itself so just delete it
      delete u;
      return;
   }
   int i;
   List<Vertex *>tmp;
   // make tmp a list of all the neighbors of u
   for(i=0;i<u->neighbor.num;i++) {
      tmp.Add(u->neighbor[i]);
   }
   // delete triangles on edge uv:
   for(i=u->face.num-1;i>=0;i--) {
      if(u->face[i]->HasVertex(v)) {
         delete(u->face[i]);
      }
   }
   // update remaining triangles to have v instead of u
   for(i=u->face.num-1;i>=0;i--) {
      u->face[i]->ReplaceVertex(u,v);
   }
   delete u;
   // recompute the edge collapse costs in neighborhood
   for(i=0;i<tmp.num;i++) {
      ComputeEdgeCostAtVertex(tmp[i]);
   }
}
```

model at any specified number of vertices can be retrieved on demand without having to recompute the polygon reductions. This feature is easily implemented by storing the vertex to which each vertex is collapsed and sorting the vertices by the order in which they were collapsed.

The BUNNYLOD.EXE demo uses this method. Initially, the bunny is reduced from 450 to 0 vertices in approximately one second. Then, as the slider on the left animates the bunny, the model is rendered in increasing detail using the specified number of polygons. Another way to think of this animation is as a sequence of models for every number of vertices between 0 and the number in original model.

The edge collapse sequence could also be used for progressive transmission. Just as interlaced .GIF and .JPG pictures come over the Web in increasing detail, the vertices of an object can be broadcast in the reverse order from which they were collapsed. The receiving computer can display the model while it is reconstructed from the incoming data stream. This is a nice idea, but it's probably not relevant for game developers just yet.

An important component in many games is the LOD of models. A handful of models can be selected from the sequence generated by our algorithm to represent the object at various LODs. One problem with swapping models is that players often notice when this occurs (the phenomenon known as "popping"). A solution to the popping effect is to morph smoothly between the models. In order to morph between two models, the vertices of one model must be mapped onto the other. Fortunately, this information can be extracted from the edge collapse sequence. The BUNNYLOD.EXE demo also shows an example of morphing.

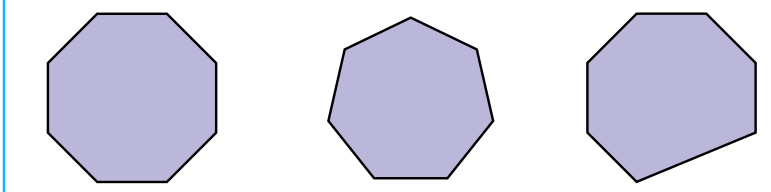
Alternatives to Edge Collapse Techniques

Polygon reduction algorithms aren't the only way to create a model with fewer faces. Artists will always be able to do a better job of representing a model using fewer polygons than any reduction algorithm. One reason is that algorithms have little or no higher-level understanding of the model. An artist, on the other hand, knows the object that he or she is creating (be it a rabbit, a chair, and so on) and can make careful aesthetic decisions as he or she manually reduces the face count. The human visual system is biased towards certain details, such as the eyes and mouth, and pays less attention to other details such as the collarbone or kneecaps. On the other hand, our simple algorithm merely compares a few dot products and edge lengths, and obviously doesn't have the intelligence to place automatically varying amounts of importance on different pieces to optimize for human perception. The advantage to using a polygon reduction algorithm is that it automates the process.

Another technique for doing LODs in a game is to represent an object's geometry using parametric surface patches, which are tessellated on the fly to the desired detail. Shiny's MESSIAH engine uses a similar approach. Certainly, these surface-based methods are preferable (and probably optimal too). Figure 8 illustrates the advantage using a 2D analogy. An octagon reduced by one edge is regenerated as a regular heptagon by the parametric approach. Collapsing an edge on the octagon produces non-regular results.

Unfortunately, using curved parametric surfaces isn't always appropriate. Some of the challenges include getting the object into this sort of representation and being able to generate polygons at render time so that adjacent surfaces fit together properly (without gaps or T-intersections). Furthermore, jagged objects aren't good candidates for use with curved surface patches because the number of surfaces would be no less than the number of polygons required. Polygon-based reduction methods are more generally useful, and work with typical models used these days.

FIGURE 8. *Comparison of techniques.*



While I hope that this information and the accompanying demonstration application that I've provided are useful, this article has not touched on issues such as dealing with texture coordinates, vertex normals, border edges, nonmanifold topology, texture seams, and so on. These subjects have been left as an exercise for the reader. Furthermore, many other variations and enhancements to this algorithm are worth exploring. One exciting topic is adaptive simplification, in which different parts of the same mesh are rendered at different levels of detail according to run-time parameters. This is especially useful for open terrain environments so that more detail can be used near the current viewpoint. ■

FOR FURTHER INFO

Polygon reduction has been a hot research topic lately, and most of the literature about it can be found in proceedings from academic computer graphics conferences. Some more places you can look:

- **Cohen, J., M. Olano, and D. Manocha. "Appearance-Preserving Simplification", SIGGRAPH '98.**
- **Hoppe, H. "Progressive Meshes," SIGGRAPH '96, pp. 99-108.**
- **Luebke, D. and C. Erikson. "View-Dependent Simplification of Arbitrary Polygonal Environments", SIGGRAPH '97, pp. 199-207.**
- **I have a demo on my university web site at http://www.cs.ualberta.ca/~melax/ polychop**
- **H. Hoppe, the Guru of polygon reduction, maintains a web site at http://research.microsoft.com/~hoppe/**