4.12

View-Independent Progressive Meshing

Jan Svarovsky

A progressive mesh (PM) is a triangle-based mesh that is able to vary its level of detail in real-time, at the resolution of gaining or losing a couple of triangles at a time, while preserving its original shape as much as possible. It can be drawn at any detail level between the conventional mesh from which it was created and a lowest detail "base mesh" as defined by the detail reduction heuristic, which may be as small as no polygons at all.

Typically, these meshes are rendered at lower detail in the distance, so that more system resources are available to draw higher-resolution meshes in the foreground. The global detail level of the graphics engine can also be based on the power of the computer it is running on.

First, I will introduce progressive meshing, working through some of the arguments for and against different variations on the theme. Based on this discussion, I will describe an algorithm to convert conventional mesh data into progressive meshes, and some efficient and simple code to render these.

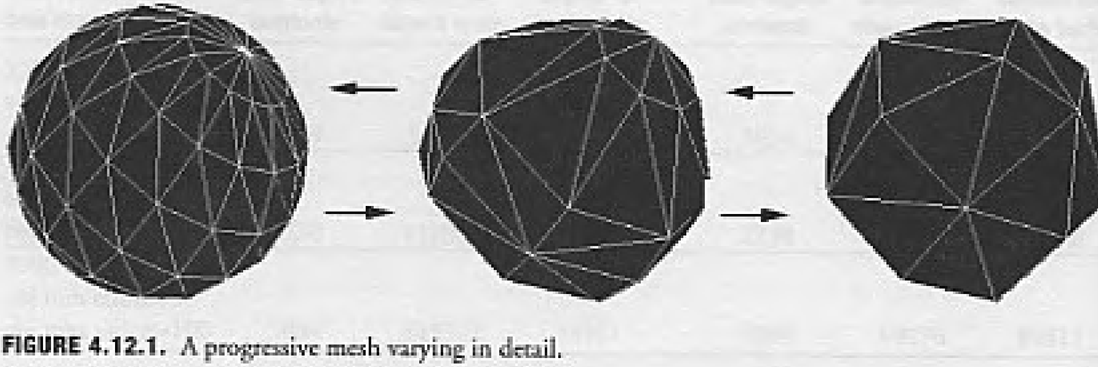

FIGURE 4.12.1. A progressive mesh varying in detail.

Progressive Mesh Overview

The basic principle can be simply described as taking a mesh, repeatedly deciding which is its least significant edge, and removing this edge by making the two vertex positions at its ends equal. This *edge collapse* operation typically makes two triangles sharing the edge redundant. Detail is put back into the mesh by reversing these collapses through *vertex splits*.

Much work has done by people and documented in the public domain, particularly [Hoppe96, Hoppe97, Hoppe98]. Figure 4.12.2 summarizes the unit-reversible operation and the common terminology.

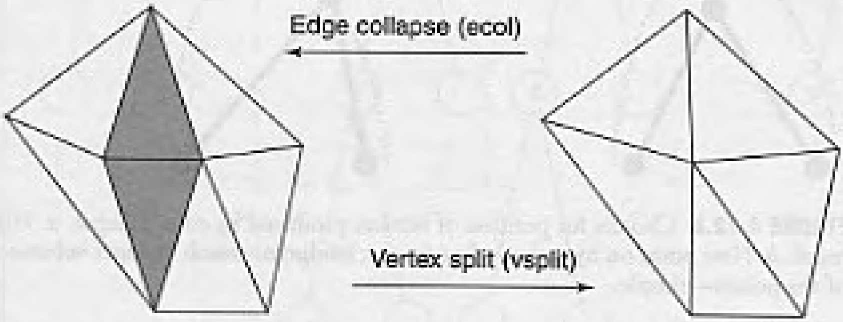

FIGURE 4.12.2 A single step of mesh refinement (vertex split) or reduction (edge collapse).

Variations on the Theme

Given this basic premise, there are various decisions that can be made about the finer implementation details. I will briefly touch on them here; see [Svarovsky99] for a more leisurely discussion.

When Vertices Collapse, Where Do They Collapse To?

When two vertices collapse into one, there is a choice for where to put the vertex. It can be calculated to lie on the imaginary smooth surface that the polygon mesh is trying to represent. Alternatively, you can put the point just halfway in between the two it replaces, which you would think was cheaper, perhaps because you wouldn't have to store a pre-calculated new point. Last, you can just choose to preserve one or the other of the original vertices that are being collapsed (Figure 4.12.3).

The midpoint system has the disadvantage that convex objects become smaller as they lose detail. The clever precalculated point system takes up twice as much memory, or takes up extra CPU time calculating the new point online. Preserving one point or the other is the simplest, takes the least memory, and objects do not lose apparent vol-

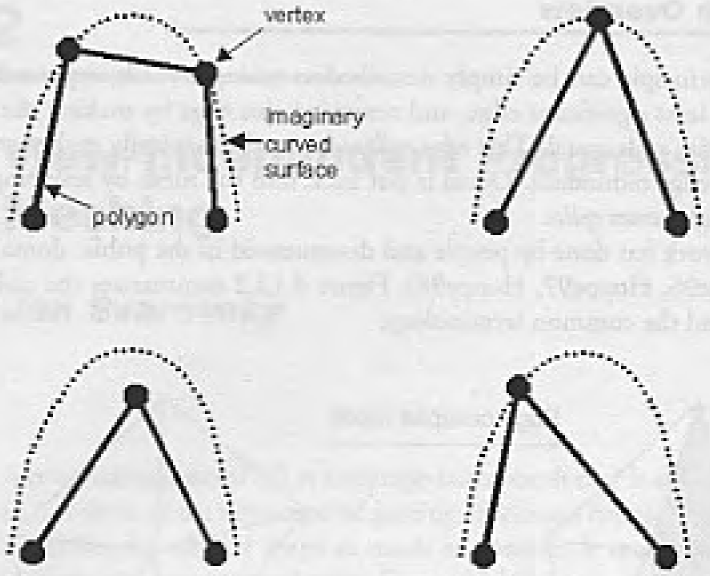

FIGURE 4.12.3 Choices for position of vertices produced by edge collapse. *a:* Higher detail mesh. *b:* New point on hypothetical surface. *c:* Midpoint- mesh changes volume. *d:* Pick one of the points—simple.

ume as drastically. It is often a good representation of the original shape, particularly when, for example, collapsing the corner of a cube-like object and a vertex somewhere along one of the faces of the cube shape. Though it lacks the flexibility of calculating a new point, its strong advantage is that it does not require real-time changes to the vertex data or the creation of new vertices. This is the system I will use here.

View-Independent vs. View-Dependent Rendering

Each sequence of vertex splits, starting with a vertex in the base mesh, can be visualized as a binary tree, each vertex splitting into two new ones (though, of course, in this system I just add one new vertex to an original one). The splits can either be left in their tree form or can be given some fixed order (Figure 4.12.4).

View-independent meshes use one fixed order for the edge collapses, which can therefore be calculated offline, and this tree representation can be thrown away. If you keep the tree form in some way, you can vary which nodes you expand. This effectively gives you more flexibility in positioning the dashed line [Hoppe97]. This view-dependent PM can be used to give more detail on parts of the mesh that are closer to the viewer, or on silhouette edges.

View-dependent PM (VDPM) is able to use triangle counts more effectively, because it has more flexibility in the choice of edge collapse order. In my opinion, however, this is never justified in modern systems because of the large gap in effi-

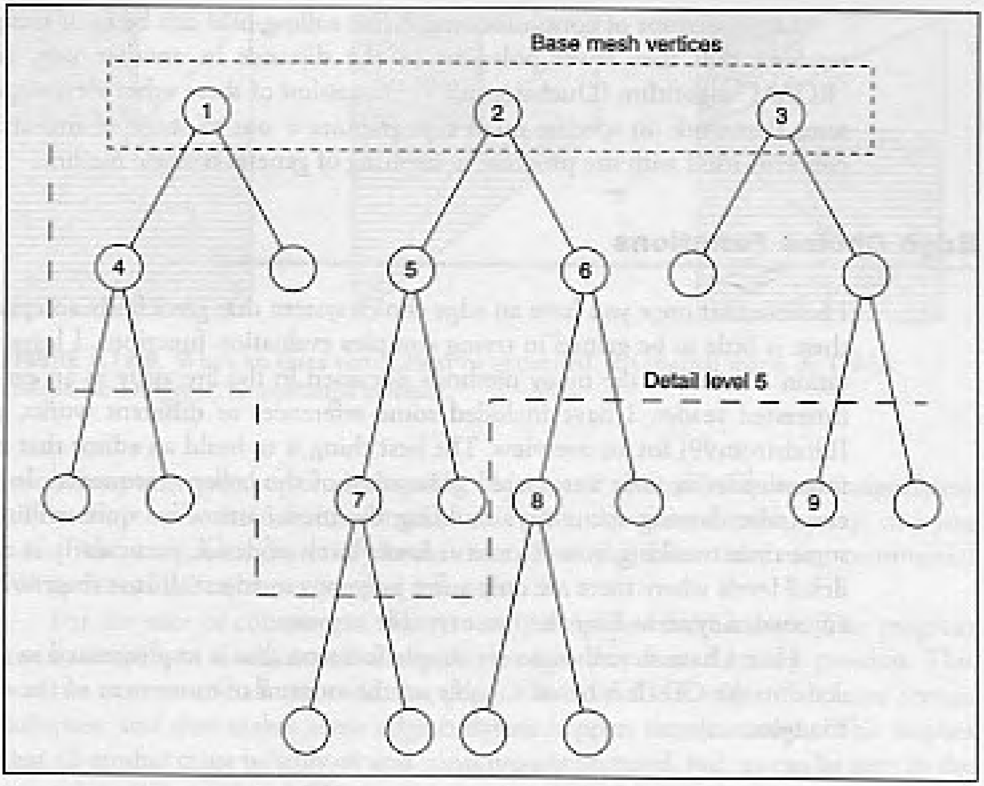

FIGURE 4.12.4 A forest of vertex split trees. The dashed line represents the vertices that will be used when rendering a mesh at detail level 5.

ciency between the two types of renderers. A VDPM renderer uses fewer triangles for a more visually pleasing scene, but this thriftiness is drowned out by the increased processor time that must be put into making more level of detail choices, and the data handling involved.

A view-independent PM's (VIPM) triangles and vertices in the mesh can be ordered such that the ones that disappear first are further toward the end of the list, and therefore are not traversed or "in the way" when the lower levels of detail are being used. This can also lead to interesting progressive file formats where the more you read, the higher detail mesh you get [Hoppe98].

Because there is only one collapse order, there is only one level of detail for the whole mesh. If you are close to one part of the mesh, and therefore want that part to be rendered at high detail, all the rest will have to be rendered at high detail, too. In practical game situations, however, a large object can be subdivided into independent (but possibly mutually intersecting) parts that can be rendered view-independently. Now that you have the pieces separate, you can assign some game code to them, so they become a bit more interactive, such as windows, antennae, radar dishes on a space station or individual huts, trees, and so on in a landscape.

Large sections of continuous mesh like rolling hills can be built using a custom renderer that uses fewer polygons in the distance in another way, such as the "ROAM" algorithm [Duchaineau97]. Discussion of these other view-dependent systems that work on specific mesh topographies is out of scope of this article, which concerns itself with the progressive meshing of general triangle meshes.

Edge Choice Functions

I believe that once you have an edge choice system that gives fairly acceptable results, there is little to be gained in trying complex evaluation functions. I leave implementation of some of the many methods discussed in the literature as an exercise to the interested reader. I have included some references to different works, particularly [Lindstrom99] for an overview. The best thing is to build an editor that allows artist intervention in your automated generation of the collapse sequence. In my experience, after having spent days building the mesh, artists are quite willing to spend some time tweaking how it looks at lower levels of detail, particularly at the very low detail levels where there are only a few polygons to adjust. It is at these low levels that automated systems have the most trouble anyway.

Here I have described a very simple function that is implemented in the example code on the CD. It is based roughly on the amount of movement of the surrounding triangles.

Difficult Edges

It simplifies the algorithm to ban some special case edges. These cases stem from triangles sharing the same point in 3D space, but not sharing some other vertex data, such as vertex normals, texture type, or texture coordinates. It is an extra complication to have triangles pointing to *shared* texture coordinates and pointing to *shared* vertex positions. To avoid this, and to be friendlier toward current graphics hardware, our vertices contain all texture coordinates, normal and position information. This way, the mesh will contain multiple vertices in the same position but with different material information.

If an edge being removed contains these duplicate vertices, and therefore the triangles along the edge do not share vertices, it is handled as two edge collapses that happen simultaneously. The problem arises when a nearby triangle only refers to one of the ends of a collapsed edge, and this is the one that is to disappear (Figure 4.12.5).

An extra vertex could be created for this material, which would remain redundant until needed for lower levels of detail. This inefficiency is only slight because these vertices are created very infrequently. You can avoid this case by just banning these collapses. This will restrict the lowest polygon count that meshes will reduce to. Typically, when these edges are a significant percentage of the remaining edges, the polygon counts will be so low that the renderer call overhead means further detail loss

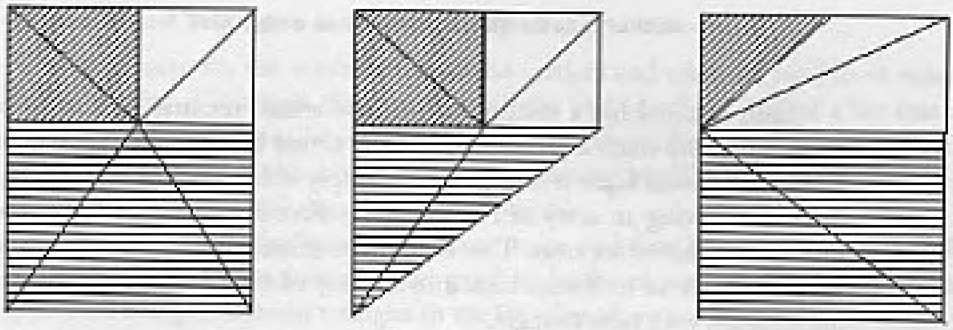

FIGURE 4.12.5 When an extra vertex must be generated. *a:* Original shape. *b:* Trivial collapse. *c:* Different, complicated collapse.

would not really speed up rendering. The objects will also probably be so insignificant (as determined by the decision function that made them low detail anyway) that you could just not draw them at all! At the time of writing this article, most commercial PM systems disallow these types of collapses.

For the sake of conciseness, the system is simplified further here. The program given contains no workarounds for multiple edges that share the same position. This way, we can remove all the code that checks for coincident edges, that bans certain collapses, and that makes some edge collapses happen simultaneously. This implies that all meshes must be smooth and continuously textured, but, as can be seen in the examples on the enclosed CD, careful construction of the meshes means many more general shapes are still possible.

Implementation

The Renderer

For the majority of frames, a mesh will not change in detail, so it is essential that the data structure being used to render from is as efficient as possible for the graphics system. Here we can arrange the data much like we would for a standard mesh renderer:

```
struct PMMesh
{
     int NumMaterials;
     struct PMMaterial *Materials;
};
```

The mesh is made of an array of materials:

```
struct PMMaterial
{
     PMTexture *Texture;
     struct PMVertex *Vertices;
     int *Indices;
```

```
        int NumVertices, NumIndices;
    };
```

Each material has a texture (or perhaps several textures in a multi-pass system, such as a bump map, a gloss map, and the actual base texture), and an array of vertices. It also owns some triangles, which simply index into the vertex array. Note that instead of having an array of PMTriangle's, there is an array of three times as many indices into the vertex array. This is done for efficiency in the EdgeCollapse structure later, and is trivial to change back into an array of triangles if you wish to record more information with each triangle.

```
    struct PMVertex
    {
        vector3 Position, Normal;
        float U, V;
    };
```

Each vertex contains position, lighting, and texture information. In this way, the materials are quite independent of each other, and the mesh looks like one continuous object because the positions of some of the vertices in different materials are the same.

Morph the Vertices or Pop?

No vertex morphing will be done in this implementation—vertices pop in and out of existence. This is cheaper and, in my experience with game teams, actually looks better than morphing. This surprising result is because, for a given polygon count, the mesh is as close as it can be to its proper shape, rather than being blended somewhere between the current shape and the next lower detail level. The pops in practice are less of a problem actually than the extra expense of doing morphing (particularly of having to edit vertex data).

Progressive Mesh Rendering Only Affects the Triangle Lists

Because an edge collapse preserves one out of the two vertices involved, this renderer modifies the triangle lists only, with no effect on the vertex data. This means that the vertex arrays can be left alone (and in some modern hardware, pre-processed into some more efficient format), and can also be shared between multiple instances of the same mesh. The triangle lists will be modified over time, and must be duplicated once for each active instance of each mesh.

This also means that vertex position modifiers, such as animation, can happen fairly independently of the progressive meshing, as long as you don't mind that the collapse order won't change even as the vertices move about against each other. The animation system only has to handle the fact that vertices can come and go, rather than be used continuously.

Lower Detail Triangles and Vertices First

A point of note for the renderer is that the vertices and triangles have been ordered offline so that it is always the triangles and the vertices at the end of a list that are made redundant by a collapse. The renderer will always be submitting triangle and vertex lists starting in the same place, just of varying lengths. Discussion of the data structure generation will show how this is possible.

This does mean that unless you create strips and fans of triangles in some other way, you will be always presenting the graphics hardware with an indexed list of triangles. Interestingly, adjacent triangles in the list often share vertices, which in many systems is as good as having triangle strips and fans. This is because at least pairs of triangles on either side of an edge collapse will be next to each other in the triangle list.

The Reversible Edge Collapse List

The other renderer data structure describes the reversible sequence of edge collapses that changes the level of detail of the mesh. Each edge collapse loses one vertex, one or more triangles, and changes which vertices some of the remaining triangles use. There is one edge collapse list for the whole object, though different individual collapses affect different materials. Alternatively, there could be a collapse list per material, but these would have to be tied together somehow so that the seams of the object don't come apart.

```
struct PMEdgeCollapse
{
    float Value;
    PMMaterial *Material;
    int NumIndicesToLose, NumVerticesToLose,
    NumIndicesToChange;
    int *IndexChanges;
    int CollapseTo;
};
```

The `CollapseTo` member says which vertex should replace all references to the vertex that is being lost off the end of the list. All these changes are stored in the IndexChanges array. This operation is simple to reverse for vertex splitting when the level of detail is being increased again.

When a collapse happens, some triangles disappear (`NumIndicesToLose`), one or more vertices may be made redundant (`NumVerticesToLose`—loss of some triangles may leave vertices completely unused), and some indices in remaining triangles will be changed (`NumIndicesToChange`). Of course, the reverse happens during a vertex split.

Because the materials are so independent, sometimes two edge collapses must be performed at once, to preserve the mesh seams as much as possible. This is when, as discussed earlier, the two edges actually are the same edge in space, so must collapse together even though they refer to different vertices. The engine must continually compare the value of the next edge collapse or vertex split that could be performed against the level of detail required from the mesh based on its position and other variables.

In the simple system presented here, these edges are not taken into account, but they can be done quite well by simply giving all edges in the same place the same priority, even though they are unconnected in the data structure.

Offline Calculation

Here I will assume that the mesh data has been loaded by some means into a friendly format. For the sake of readability, the algorithm will be the simplest rather than the most efficient, particularly since we are not so worried about the expense of offline calculations.

The procedure can be summarized as repeatedly deciding which is the next bit of detail to be lost, and removing it from the mesh, while generating the edge collapse data that will be needed by the renderer later. When it is decided that a vertex should be removed, all triangles that refer to it must be changed, and any triangles that are made degenerate swapped to the end of the list. Similarly, the vertex is moved to the end of the remaining vertex list.

Of course, swapping triangles and vertices to the ends of their lists changes all references to them, in other edge collapse structures as well as in the remaining mesh. There may be other code (such as an animation system that is about to use the same mesh) that needs to know about vertex reordering.

Suggested Offline Calculation Optimizations

Looking at the code, it is obvious that extra temporary connectivity information would be useful in the mesh. For example, the code often looks for "all triangles that reference this vertex" by brute force. Also, the code repeatedly searches through all the triangles for the next edge to collapse. Huge performance improvements are possible if you put the edge collapse candidates into a priority heap.

Edge Selection Improvements

The most effective edge selection improvement is to make edges that affect discontinuities in the mesh less likely to collapse. This makes many more mesh shapes possible, and also allows objects to be subdivided further into subobjects. Each mesh subdivision gives an extra degree of freedom in level of detail choice—see the previous discussion about making huts and trees separate from the landscape mesh underneath them. See Figure 4.12.6.

A Further Variation on Progressive Meshing

Instead of being able to change level of detail at the resolution of one vertex at a time, you could just store several pre-calculated index lists at various resolutions. The changes between these levels of detail of course will be more obvious. This system

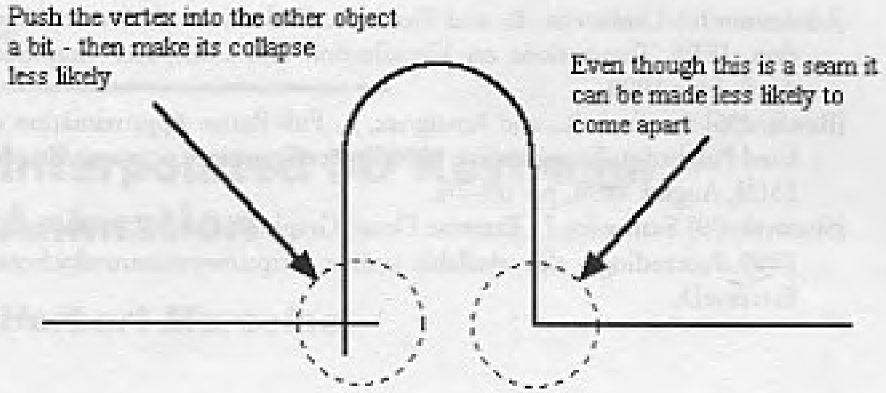
Push the vertex into the other object a bit - then make its collapse less likely
Even though this is a seam it can be made less likely to come apart

FIGURE 4.12.6 What was one mesh becomes two subtly intersecting ones, or just two coincident ones.

becomes more useful if the polygon counts or frame rates are so high that the popping is not a problem.

An advantage is that you can throw away the edge collapse list, which is actually quite a large data structure, certainly comparable to the extra index lists you are storing in this new method. You also lose the collapsing/splitting code, and you don't need a separate index list for each active instance of each object. The renderer becomes much simpler—you are back to a normal mesh renderer, but just with code to select which index list to use for each object at a given moment.

Source Code

Code for the progressive mesh generator and the renderer is contained on the CD. It is written in a general manner with little system dependency.

References

[Duchaineau97] Duchaineau, M. et al, ROAMing Terrain: Real-time Optimally Adapting Meshes, 1997, available online: http://www.llnl.gov/graphics/ROAM/roam.pdf.

[Garland97] Garland, M., and Heckbert, P.S., Surface Simplification Using Quadric Error Metrics, Siggraph 1997 Proceedings, pp. 209–216, August 1997.

[Hoppe96] Hoppe, H., Progressive Meshes, Siggraph 1996 Proceedings, pp. 99–108, August 1996.

[Hoppe97] Hoppe, H., View-dependent refinement of Progressive Meshes, Siggraph 1997 Proceedings, pp. 99–108, August 1997.

[Hoppe98] Hoppe, H., Efficient implementation of progressive meshes, Computers & Graphics, Vol. 22(1), pp. 27–36, 1998.

[Lindstrom99] Lindstrom, P., and Turk, G., Evaluation of Memoryless Simplification, IEEE Transactions on Visualization and Computer Graphics, Vol.5(2), April–June 1999.

[Ronfard96] Ronfard, R., and Rossignac, J., Full-Range Approximation of Triangulated Polyhedra. Eurographics 1996 Proceedings, in Computer Graphics Forum, 15(3), August 1996, pp. 67–76.

[Svarovsky99] Svarovsky, J., Extreme Detail Graphics, Game Developer's Conference 1999 Proceedings, also available online: http://www.svarovsky.freeserve.co.uk/ExtremeD.