

An Algorithm for Incremental Multi-Resolution Modeling

Dashamir Hoxha

April 11, 2003

Abstract

This article presents an algorithm for incremental multi-resolution modeling of 3D objects in computer graphics. This algorithm changes the resolution of the model incrementally by collapsing edges and triangles of the model to a single point. It shows how to do the collapse in such a way that the reverse process (called uncollapse) is possible and reverts the model exactly in the same state that it was before the collapse. Then it shows that the order in which the triangles and edges of a given model are collapsed is unique and it can be determined in a preprocessing stage. This preprocessing stage can build such a representation of the model that allows the rendering algorithm to collapse and uncollapse triangles and edges very efficiently. The article describes this representation of the model (called multi-resolution model) and shows how it can be saved in a file and loaded from it. It then shows that such a model is efficient in terms of memory. It finally describes how collapses and uncollapses are done on such a model, and shows that it is very efficient in terms of speed.

1 Introduction

In Computer Graphics, 3D objects are usually composed of many triangles which are rendered to display the object. Each triangle is composed of three vertexes, which are points in space, represented by coordinates (x, y, z) . Such a representation of an object is called a *model* of the object.

We can construct several models for the same object. E.g. to construct the model of a computer we can use 300 triangles or 500 triangles. The number of the triangles of a model is called the *resolution* of the model. Usually, the higher the resolution is (more triangles we use), the better is the quality of the 3D object and more computing time it takes to render the object on the screen. However, sometimes, more triangles doesn't mean better quality, e.g. we can model a cube using just 12 triangles (two for each face) or 100 triangles, but the quality will be the same. Or, if the object is far away from camera and its rendered image takes just a few pixels on the screen, then it doesn't matter whether the object is modeled using 100 triangles or 1000 triangles: the quality of the image will be the same but the rendering algorithm will take more time in the second case because it will have to process more triangles. When an object is near to the camera we would like to render it using a high resolution model, in order to have a good quality, however, when it is far from the camera we would like to use a low resolution model in order to save processing time.

To solve this problem, some rendering algorithms take the approach of pregenerating several models with different resolutions for an object, and then using a high resolution model when the object is near the camera, a lower resolution model when the object is a little bit farther, and so on. This representation of an object using several models with different resolutions is called *multi-resolution modeling*. While this approach may improve the rendering time of dynamic scenes, it has these drawbacks:

1. When the switch from one resolution to another happens, often it is visible, because the change of the resolution is large, e.g. from 1000 to 500, and this has a bad visual effect on the scene.
2. It increases the space that is required to store an object, because now we use several models instead of just one.

In order to eliminate the first problem above, some other algorithms take the approach of changing the resolution of an object incrementally, little by little. So, when the object starts moving away from the camera, the algorithm removes several triangles from the model time after time, and when the object moves nearer to the camera it starts adding them back again. The difficulty is how to choose which triangles to remove and how to remove them, so that the quality of the model is not damaged too much, and then how to put them back again in their previous places.

The algorithm presented in this article falls in the last category of algorithms. It saves the model of an object in such a way that makes possible the

incremental decrease and increase of the resolution. It also has a low space and time cost (complexity).

The algorithm decreases the resolution of the model by “collapsing” a triangle or an edge into a single point. It keeps a sorted list of edges and triangles of the model, which is sorted according to the *visual importance* of the triangles and edges, and always collapses the triangles and edges at the top of this list (which have the lowest visual importance). The visual importance of a triangle or an edge is defined as the error that will be caused to the model by collapsing this triangle or edge, and is calculated using some heuristic formulas.

2 Collapsing and Uncollapsing

2.1 The Data Structures

The algorithm makes use of some complicated data structures for keeping the model of the object, so it is necessary to have an overview of the data structures used, before trying to discuss the algorithm. See Figure 1.

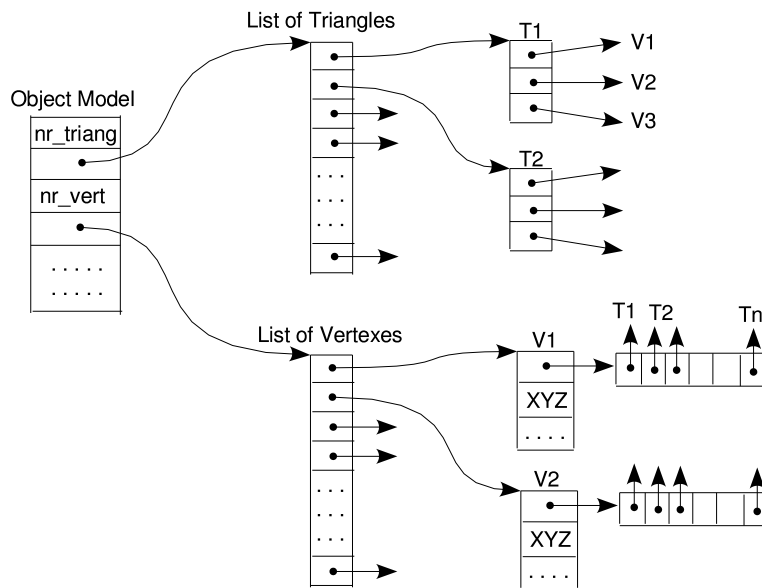


Figure 1: The Data Structure

A model structure keeps a list of triangles, a list of vertexes and some other things. The list of triangles has pointers to triangle structures and the list of vertexes has pointers to vertex structures. Each triangle has three

pointers to its three vertexes. Each vertex has a list with pointers to neighbor triangles, coordinates, etc.

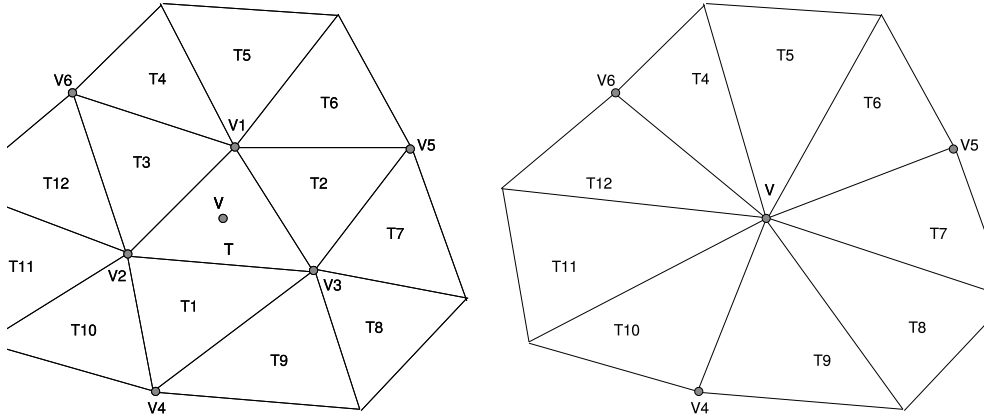


Figure 2: Collapsing a Triangle

In the Figure 2 is shown only that part of the model that is affected by the collapse of the triangle T , the other parts are not shown. The triangle T will be collapsed to a vertex V inside it. The triangles T_1, T_2, T_3 will also collapse. After the collapse, vertexes V_1, V_2, V_3 and triangles T, T_1, T_2, T_3 will not be part of the new model, a new vertex V will be part of the model and the triangles $T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}$ will be incident at the new vertex V (they will be the neighbors of V).

2.2 Collapsing a Triangle

The triangle that will be collapsed is in the top of the list of visual importances (it has the smallest visual importance). Assume that we have a pointer to the triangle that will be collapsed, how are we going to collapse it?

Algorithm *collapse*

1. Find triangles T_1, T_2, T_3 (they are in the intersection of the neighbour lists of two of the vertexes V_1, V_2, V_3 , and they are different from T).
2. Create a new vertex V with appropriate coordinates. (This vertex should be positioned such that it minimizes the visual error of the new model with respect to the previous model.)

3. Add in the list of neighbor triangles of V all the triangles $T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}$. (They are all the neighbours of V_1, V_2 and V_3 , except for the triangles that are being collapsed: T, T_1, T_2 and T_3 .)
4. For each triangle in the above mentioned list (neighbour triangles of V) change one vertex of them so that it does not point to an old vertex (one of V_1, V_2, V_3) but to the new vertex V .
5. Find vertexes V_4, V_5 and V_6 . (They belong to triangles T_1, T_2 and T_3 , but are different from V_1, V_2 and V_3 .)
6. Remove T_1 from the list of triangles of vertex V_4 , T_2 from the list of V_5 , and T_3 from the list of V_6 .
7. Remove T, T_1, T_2, T_3 from the list of triangles of the model, recalculate visual importances of triangles T_4, T_5, \dots, T_{12} , and resort the list of triangles according to the new visual importances.
8. Now, if we delete these triangles, then this collapse would be irreversible, because we can't possibly restore the model exactly as it was before the collapse. Instead, let us keep a structure with undo (or uncollapse) info, which keeps pointers to V, T, T_1, T_2, T_3 . This structure can be pushed on top of an undo stack and can be used later to undo this collapse (or to do uncollapse).

2.3 Undoing a Collapse

During the uncollapse we restore the model to the exact state that it was before the collapse, by placing the triangles T, T_1, T_2, T_3 in the places that they were before. The triangle T already has all the information that it should have: it has pointers to the three vertexes V_1, V_2, V_3 , and they have pointers to their old neighbor triangles. Also, the triangles T_1, T_2, T_3 have the information about their vertexes. The problem is that triangles T_4, T_5, T_6 do not know the vertex V_1 , they know the vertex V (they don't have a pointer to V_1 but to V). Also, vertex V_5 has forgotten the triangle T_2 (it does not have it in the list of neighbor triangles), although T_2 has not forgotten V_5 . However these problems can be solved.

Algorithm *uncollapse*

1. Pop the head of the undo stack and get the pointers to V, T, T_1, T_2, T_3 .

2. For each triangle in the list of triangles of V_1 ,

if this triangle is different from T, T_1, T_2, T_3 , then

find the vertex that points to V and set it to point to V_1 (now T_4, T_5, T_6 have V_1 as their vertex, instead of V)

Do the same thing for V_2 and V_3 .

3. Add triangle T_1 in the list of neighbor triangles of V_4 , add T_2 to V_5 , add T_3 to V_6 .
4. Insert triangles T, T_1, T_2, T_3 in the list of triangles of the model, recalculate the visual importances of the triangles T_4, T_5, \dots, T_{12} , and resort the list of visual importances. (A good choice for implementing the list of visual importances is the *heap* data structure, since we have insertions and deletions in it and we want to keep it ordered all the time.)

2.4 Speed-Memory Trade-offs

In the undo stack we kept pointers to 5 objects: V, T, T_1, T_2, T_3 , in order to put everything back in the model. Actually we need only T and we can find the rest from it. T_1, T_2, T_3 can be found like this: look at the triangles around V_1 , around V_2 , around V_3 and choose those 3 that have 2 and only 2 common vertexes with T . A pointer to V can also be found, because V is the common vertex of more than two triangles which are neighbors of V_1, V_2, V_3 , but are different from T, T_1, T_2, T_3 .

So, if we keep just a pointer to T in the undo stack, it would be more efficient in terms of memory, but less efficient in terms of speed. Usually, in computer graphics we want to reduce calculations in order to gain speed, so we are willing to loose some space.

2.5 Collapsing and Uncollapsing Edges

Collapsing and uncollapsing an edge is very similar to collapsing and uncollapsing a triangle, as it can be seen in the Figure 3. In this case we would keep in the undo stack only V, T_1, T_2 .

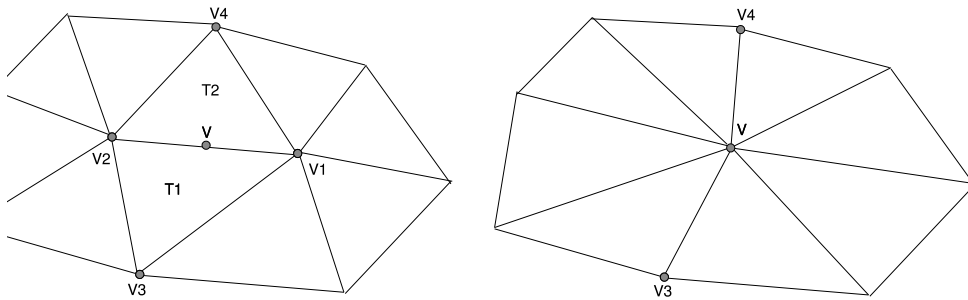


Figure 3: Collapsing an Edge

3 The Multi-Resolution Model

3.1 The Series of Models

Starting with an initial model M_0 , by collapsing a triangle or an edge we get another model M_1 which has a lower resolution. Repeating this process again we get another model M_2 with a lower resolution, and so on. Thus, we get a series of models: $M_0, M_1, M_2, \dots, M_{min}$, with decreasing resolutions:

$$Res(M_0) > Res(M_1) > Res(M_2) > \dots > Res(M_{min})$$

At the end of this series there is M_{min} , a minimal model, composed of only a few triangles. From any model in this series we can get the previous one by performing the uncollapse process (using data from the undo stack) because it is the reverse of the collapse process. The model M_0 is the most detailed one that we have, and M_{min} is the most simple. Lets call the process of removing triangles (getting from M_0 to M_1 , to M_2 , etc.) *simplification* and the reverse process *adding details*. When the model starts to move away from the camera, the rendering algorithm starts to simplify it, and when the model moves toward the camera, it adds details to it.

3.2 Preprocessing a Model

For any given model M_i in the series, the triangle or the edge with the lowest visual importance is the one that is collapsed next. The order in which the triangles or edges are collapsed depends on the heuristic that calculates the visual importance and is unique.

Since this order is unique, it can be found in a preprocessing stage and stored somehow together with the model. This would free the rendering

algorithm from calculating visual importances, from resorting triangles and edges according to their visual importances, etc. On the other hand, this means that we can use a complicated expression for calculating the visual importance. It doesn't matter how much processing time it takes, since it is done only once, in the preprocessing stage, and is not used during the rendering.

To find this order, the preprocessing algorithm can do a maximal simplification, i.e. collapse triangles and edges until we get M_{min} , the minimal model. The order in which the triangles and edges have been collapsed during this maximal simplification, is the order of collapsing that should be used in rendering time.

3.3 The Representation in Memory of a Preprocessed Model

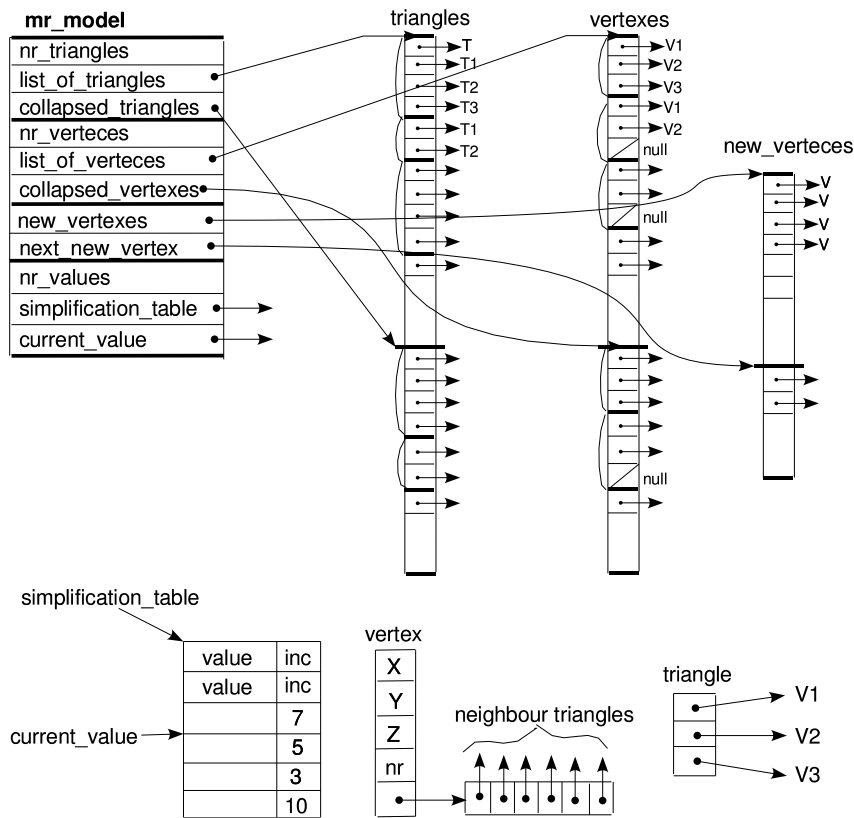


Figure 4: Representation in Memory of a MR-model

As seen in the Figure 4, the model is composed of a list of triangles, a list of vertexes, a list of "new vertexes" and a simplification table. The list of the triangles and the list of the vertexes is composed of two parts: the part that is still in the model, and the part that is already collapsed. The rendering algorithm uses only the triangles that are not collapsed to render the object. collapsed_triangles and collapsed_vertexes actually perform the role of the undo stack. The order in which the triangles are collapsed and uncollapsed is unique, so in the preprocessing stage, the triangles and vertexes have been sorted in this order. Then, when we want to make a collapse, we just decrement the pointers collapsed_triangles and collapsed_vertexes and increment the pointer next_new_vertex. When we want to uncollapse, we move the pointers in the reverse direction. The triangles in the list of triangles are organized in groups of 4 (for triangle collapse/uncollapse) or in groups of 2 (for edge collapse/uncollapse). The vertexes in the list of vertexes are always in groups of three, but for an edge collapse the third pointer is NULL (this is how the algorithm finds out whether the next collapse is an edge collapse or a triangle collapse).

A vertex structure has the coordinates (x, y, z) of the vertex, a list of neighbor triangles, and *nr* which is the number of the neighbor triangles. The triangles in the list of neighbor triangles are sorted according to the order in which they are removed from the list, with the first removed being the last in the list. This order is unique and they are arranged in this order by the preprocessing algorithm. When we have to remove a triangle from the list of the triangles we just decrement *nr* (to insert, just increment it).

The model contains also a simplification table. Each entry in this table has a float *value* and an increment. The *value* expresses somehow the degree of the simplification of the model M_i from the original model M_0 . It is stored in ascending order. The increment tells how many collapses must be done in the current model M_i in order to go to the next value in the table. So, this table does not necessarily contain an entry for each model in the series M_0, M_1, \dots, M_{min} , but only for some of them and the increment is the number of models between two selected models. This table is computed in the preprocessing stage by using some heuristics to evaluate the error caused by the simplification.

The rendering algorithm uses this table to decide how much to simplify the model (or how much detail to add). Taking into account the distance of the object from the camera, the resolution of the screen etc. it calculates, using some heuristics, what is the degree of the simplification allowed so that the quality of the rendered image is not damaged. Then, using the simplification table, it decides how much triangles to remove from the model (or to add to it), so that the resulting model has the required degree of

simplification.

This representation in memory contains all the models of the series M_0, M_1, \dots, M_{min} and the way how to switch from one of them to another, so let us call it *MR-model* (Multi Resolution model).

3.4 Storing the MR-Model in a File

The preprocessing algorithm takes a model in a raw format (or any other format) and after processing it constructs a *MR-model*. Then it has to save it somehow in a file so that the rendering algorithm can load it easily.

A *MR-model* can be saved like this:

- the number of triangles (unsigned integer)
- the number of vertexes (unsigned integer)
- a list of all triangles, ordered in the collapsing order:
 - for each triangle are saved 3 pointers to vertexes, but since it is meaningless to save pointers in a file, indexes are saved instead, i.e. the index of the vertex in the list of vertexes:
 - vertex_1 (unsigned integer)
 - vertex_2 (unsigned integer)
 - vertex_3 (unsigned integer)
- a list of all vertexes, not in any particular order:
 - for each vertex are saved:
 - coordinate x (double)
 - coordinate y (double)
 - coordinate z (double)
 - a list of neighbor triangles:
 - the index of each neighbor triangle
- a list of vertex indexes, ordered in the collapsing order, (containing also NULL values, denoted e.g. by -1)
- a list of vertex indexes, ordered in the collapsing order
- simplification table:
 - number of entries in the table (unsigned integer)

- for each entry of the table:
 - simplification_value (float)
 - number_of_collapses (unsigned byte)

The program that loads the model from the MRM format, after getting the number of the triangles and the number of the vertexes, can allocate memory for each triangle and each vertex, so that all the triangles are next to each other and all the vertexes are next to each other in memory. Then, when it gets the index of a triangle or the index of a vertex, it can convert it easily to a pointer to the corresponding triangle or vertex.

3.5 Space Complexity of the *MR-model*

Let's try to calculate approximately the size of a *MR-model*, depending on the number of the triangles of the original model. Let's denote with T the number of triangles of the original model, V the number of the vertexes of the original model, C number of collapses (triangles and edges), and N the average number of the neighbor triangles of a vertex.

If we do only triangle collapses then $C \approx T/4$ (because in each collapse we remove 4 triangles from the object). If we do only edge collapses, then $C \approx T/2$. In the worst case, $C = T/2$.

N is ≥ 3 and usually is not very large; let's assume $N = 5$. If we sum up the number of neighbor triangles for each vertex, this sum would be $3T$, because we would have counted each triangle 3 times, for each vertex that it has. This sum can also be approximated to NV or $5V$. So, we have $5V \approx 3T \Rightarrow V \approx \frac{3}{5}T$.

Now let's calculate the size of the *MR-model*:

1. memory occupied by a triangle is $3 * 4B$ (3 pointers, 4 bytes each),
memory occupied by all triangles is $(3 * 4B)T = 12T$ B
2. for each collapse we create a new vertex, so the number of all vertexes is $(V + C)$; memory occupied by all vertexes is:
 $(V + C)(3 * 8B + 1B + 4B + 4NB) \approx 49(V + C)$ B
3. memory occupied by the list of vertexes is $4B * (V + C + C)$ (4B is the size of a pointer, $V + C$ is the number of all vertexes, but we can also have some NULL pointers, whose number is equal to the number of the edge collapses, which can be from 0 to C ; in the worst case it is C)
4. memory occupied by the list of the new vertexes is $4B * C$ (the number of new vertexes is equal to the number of collapses)

- The size of the simplification table is at most C (when all increment values are 1), so its size is $(8B + 1B) * C = 9C$ B (8B is the size of a float and 1B is the size of an unsigned byte)

In the worst case, the total size of the *MR-model* would be:

$$(12T + 49(V + C) + 4(V + 2C) + 4C + 9C) \text{ B}$$

By replacing $C = T/2$ and $V = \frac{3}{5}T$, we get approximately **90T** B.

Let's compare it with the size of a raw data format: for each triangle we need $9 * 8B = 72B$ (9 is the number of coordinates and 8B is the size of double), so in total it takes **72T** B.

As can be seen, the *MR-model* does not take too much space even in the worst case, although it keeps all the M_0, M_1, \dots, M_{min} series of models and in the first sight it would appear that it is very expensive in terms of memory. Sometimes (or most of the times) it may take even less space than the raw data format, and this should not come as a surprise.

4 The Algorithm for Simplifying and Adding Details to *MR-Model*

The algorithm describes how to do a collapse (edge or triangle) and an un-collapse. It is expressed in C pseudo-code. Look at Figure 2, Figure 3 and Figure 4, in order to follow it easily.

4.1 Algorithm *collapse*

```
function collapse()
{
    //get V
    next_new_vertex++;
    V = * next_new_vertex;

    //get V1, V2, V3
    collapsed_vertexes -= 3;
    V1 = * collapsed_vertexes;
    V2 = * (collapsed_vertexes + 1);
    V3 = * (collapsed_vertexes + 2);

    //do an edge collapse or a triangle collapse
    if (V3==NULL) edge_collapse(V,V1,V2);
}
```

```

    else triangle_collapse(V,V1,V2,V3);
}

function edge_collapse(V, V1, V2)
{
    //get T1, T2
    collapsed_triangles -= 2;
    T1 = * collapsed_triangles;
    T2 = * (collapsed_triangles + 1);

    //for each triangle in the list of neighbor triangles of V,
    //change one vertex of them so that
    //it does not point to V1 or V2, but to V
    . . . . .

    //find V3, V4
    . . . . .

    //remove T1 from V3 and T2 from V4
    V3->nr_neighbors--;
    V4->nr_neighbors--;
}

function triangle_collapse(V, V1, V2, V3)
{
    //get T, T1, T2, T3
    collapsed_triangles -= 4;
    T = * collapsed_triangles;
    T1 = * (collapsed_triangles + 1);
    T2 = * (collapsed_triangles + 2);
    T3 = * (collapsed_triangles + 3);

    //for each triangle in the list of neighbor triangles of V,
    //change one vertex of them so that
    //it does not point to V1, V2 or V3, but to V
    . . . . .

    //find V4, V5, V6
    . . . . .

    //remove T1 from V4, T2 from V5, T3 from V6

```

```

V4->nr_neighbors--;
V5->nr_neighbors--;
V6->nr_neighbors--;
}

```

4.2 Algorithm *uncollapse*

```

function uncollapse()
{
    //get V
    V = * next_new_vertex;
    next_new_vertex--;

    //get V1, V2, V3
    V1 = * collapsed_vertexes;
    V2 = * (collapsed_vertexes + 1);
    V3 = * (collapsed_vertexes + 2);
    collapsed_vertexes += 3;

    //do an edge uncollapse or a triangle uncollapse
    if (V3==NULL) edge_uncollapse(V,V1,V2);
    else triangle_uncollapse(V,V1,V2,V3);
}

```

```

function edge_uncollapse(V, V1, V2)
{
    //get T1 and T2
    T1 = * collapsed_triangles;
    T2 = * (collapsed_triangles + 1);
    collapsed_triangles += 2;

    //for each triangle in the list of V1
    //that has a pointer to V,
    //change it to point to V1;
    //do the same for V2;
    . . . . .

    //find V3, V4
    . . . . .

    //add T1 to V3 and T2 to V4

```

```

    V3->nr_neighbors++;
    V4->nr_neighbors++;
}

function triangle_uncollapse(V, V1, V2, V3)
{
    //get T, T1, T2, T3
    T = * collapsed_triangles;
    T1 = * (collapsed_triangles + 1);
    T2 = * (collapsed_triangles + 2);
    T3 = * (collapsed_triangles + 3);
    collapsed_triangles += 4;

    //for each triangle in the list of triangles of V1
    //that has a pointer to V,
    //change it to point to V1;
    //do the same for V2 and V3;
    . . . . .

    //find V4, V5, V6
    . . . . .

    //add T1 to V4, T2 to V5, T3 to V6
    V4->nr_neighbors++;
    V5->nr_neighbors++;
    V6->nr_neighbors++;
}

```

4.3 Time Complexity

As it can be observed in the algorithm, to perform a collapse or an uncollapse it takes executing a certain number of statements, which is very small and most of them are just increments or decrements. There is no loop, no relation with the number of triangles (resolution of the model, size of the model), no relation with the degree of the simplification, etc. So, the complexity of the algorithm has to be $O(1)$, constant. Or we can even say that the complexity of the algorithm is $O(0)$, if by convention we accept this to mean *very little time*, or *almost no time*.

5 Related Work

This work was intended to be an improvement of the algorithm presented in the following article, although later it developed to be a different algorithm:

- Veysi Isler, Rynson Lau, and Mark Green, “Real-time Multi-resolution Modeling for Complex Virtual Environments,” Proceedings of ACM VRST, pp. 11-20, July 1996.

Another algorithm which is very similar to the one presented here is *Progressive Meshes* of Hugues Hoppe:

- Hugues Hoppe, “Progressive Meshes,” ACM SIGGRAPH 1996, pages 99-108. <http://citeseer.nj.nec.com/hoppe96progressive.html>

The difference between the algorithms is that MRM uses both triangle collapses and edge collapses. Another important difference is that, while PM saves the information that is needed to inverse the collapse, MRM keeps the collapsed structures themselves, which in turn have all the information needed to reverse the collapse. Both approaches have their advantages, e.g. while PM can conserve disk space by using special techniques to compress the uncollapse information, MRM is faster because it doesn't need to calculate anything in order to do a collapse or an uncollapse. Some of the features that are supported by PM, like geomorphs, selective refinement and progressive transmission, can also be supported by MRM with small modifications.

Other related articles are:

1. Stan Melax, “A Simple, Fast and Effective Polygon Reduction Algorithm”, Game Developer Magazine, November '98. <http://www.melax.com/polychop/gdm>
2. Raimund Leitner, “Hierarchical Dynamic Simplification for Interactive Visualization of Complex Scenes.” <http://www.cg.tuwien.ac.at/studentwork/CESCG/CESCG-2001/RLeitner/hds-web.html>

6 Conclusion and Future Work

The algorithm presented in this article showed how to collapse triangles and edges so that the uncollapse is possible. It also pointed out that the order of the collapsing of edges and triangles is unique and so it can be determined in a preprocessing stage, in order to free the rendering algorithm from such a burden. The preprocessing algorithm can build such a model of the object

that makes it possible for the rendering algorithm to change its resolution on the fly very easily and efficiently. This model, called *MR-model* (multi-resolution model) is also efficient in terms of space.

The preprocessing algorithm finds the order in which triangles and edges should be collapsed by keeping them sorted in the order of their visual importances. The algorithm collapses to a new vertex the edge or triangle with the lowest visual importance. A good heuristic or formula for calculating visual importances of edges and triangles and a good formula for calculating the coordinates of the new vertex are important for preserving the quality of the original model as much as possible while it is being simplified. Finding them is something to be done in the future.

The preprocessing algorithm also builds a simplification table by calculating somehow the degree of change of the simplified model from the original one. The rendering algorithm then uses this table and a heuristic to determine how much the model can be simplified without causing a loss of the quality of the rendered image. Finding such heuristics is very important because after all this is the goal of multi-resolution modeling: simplifying the model in order to save processing time, but not as much as to damage the quality of the rendered image. This is also a work to be done in the future.

Acknowledgments

I wish to thank my ex-teacher, Dr. Veysi İşler, which made me interested and involved me in the topic of multi-resolution modeling; without him this algorithm would not have existed. I also wish to thank the INIMA staff for the conditions that they created for me to write the article, and my friend, Artan Simeqi, for the review that he did to the article and for his comments.