Dynamic View Dependent Level of Detail

Nathan Berkey Yumi Song

ABSTRACT

We present a method for dynamically computing non-uniform resolution meshes for use in real time interaction. Our method allows for several levels of detail to coexist within a single mesh in different regions without discontinuities at the edges, and allows these regions to be computed in real time.

CR Categories and Subject Descriptors: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling – surfaces and object representations

Additional Keywords: mesh simplification, level of detail, shape interpolation, progressive transmission, geometry compression

1 INTRODUCTION

A common problem in computer graphics is the problem of picking a level of detail for a mesh that maintains quality of display while improving running time to allow for interactive display rates. Many solutions exist to solve this problem, typically defining some set of of meshes to be swapped between as an object's importance diminishes in the scene thus saving computational resources on objects which would improve the image quality little by using the higher level of detail. These solutions perform well in scenes composed of many small objects where many of these objects are in the background or take up small amounts of image space, allowing them to be swapped to low detail models.

Unfortunately these types of solutions do little to help when the scene is primarily one item or a few large items. In these situations much of the items may lie outside of the image or contribute little to the image while still taking up a significant portion of image space. In these situations we would like to be able to simplify the model to save resources on the unimportant regions, but are unable to with a uniform level of detail without sacrificing image quality.

We could think to solve this by dividing the mesh into discrete regions each of which could have its own level of detail, but this would require significant work to avoid discontinuities at the edges of each region as well as being generally unwieldy. What we want for these situations is a way to compute heterogeneous levels of detail over a single object without introducing seams in the mesh or requiring expensive global calculations while still taking into account the current view of the object.

2 RELATED WORK

We base our work on the progressive mesh (PM) framework proposed by Hoppe [1]. The PM representation as discussed in [1] already allows for selective refinement of a given region by processing only those vsplit records which affect a vertex which should be refined based on some runtime criteria. This does not, however, deal with the problem of re-simplifying areas that are no longer of importance without returning to the global optimization problem used to originally compute the PM representation (which is too expensive to do at runtime), and constructs its series of edge collapses without regard to the number of refinements needed to introduce a given vertex which results in an O(n) operation to refine a given area of the mesh.

Xia and Varshney [2] proposed a solution based in the PM framework by computing a merge tree rather than a linear sequence of edge collapses and introducing a notion of an edge's region of influence and collapsing based on shortest edge to help construct a balanced tree. They also encode dependencies for each transformation based on the neighbors of the parent vertex after each edge collapse in order to avoid the expensive checks to determine the legality of each transformation at run time, instead only checking the few dependencies.

Hoppe also proposed an extension in [3] at a similar time but independent of [2] that works on a general PM rather than a specially constructed one, and encodes dependencies based

on a subset of neighboring vertices and faces. This allowed them to maintain a smaller set of constraints than [2] and thus have as short or even shorter merge trees than [2] without restricting the construction of the PM.

Both papers use a range of scalar attributes at each vertex and its children to guide refinement, dynamically computing a set of active vertices and corresponding triangulation for each frame at run time. Our approach draws from both methods and will be discussed below.

3 ALGORITHM AND REPRESENTATION

3.1 Computing the Merge Tree

We use a merge tree structure much like that of [2] in our implementation. Technically it is not a single tree but a forest of trees rooted in the vertices of the lowest level of detail. We compute the tree iteratively starting from the original mesh. At each iteration we pick the shortest edge that it is legal to collapse, e, add a new vertex, v, to the mesh as the parent of the endpoints of e, delete the faces adjacent to e, and then update all triangles that include one of the endpoints to instead use v. We then mark every triangle so updated as being in the region of influence of a collapsed edge.

An edge is legal to collapse if no triangle in its region of influence is marked and no vertex is adjacent to the endpoints of e by more than one edge. The first condition helps to keep the merge tree balanced while the second ensures the mesh does not become non-manifold. If we find that there are no legal edge collapses remaining we unmark all triangles and try again, if we still cannot find a legal collapse we accept the current forest as the final state and end construction.

The selection of shortest edges can be done in O(n log(n)) time by using a heap with key updates, but we chose not to spend time on this as the tree construction is done offline and does not affect runtime performance.

3.2 Choosing Dependencies

As in [2] we maintain a set of dependencies at each parent vertex encoding which vertices must be active in the mesh in order for that vertex to form or split to its children. This restricts the set of legal meshes from the tree and serves both to prevent illegal transformations by assuring the same neighbors as were present during tree construction and to prevent dramatic changes in resolution over small areas of the mesh by restricting the difference in levels in the tree between neighboring vertices.

We compute the dependency set by noting the neighbors of the parent vertex as it is formed by an edge collapse and storing a pointer to each in the parent vertex. Since we introduce each parent as a new vertex like the renaming in [3] we are able to check for existence in the mesh as well as adjacency by simply checking if each dependency vertex is active. This set of dependencies is more restrictive than the dependencies used for [3], but does not require keeping all faces from all levels of the tree around and also does not require keeping a list of active faces, significantly simplifying implementation of mesh operations.

3.3 Scalar Representation

Both [2] and [3] use a set of attributes at each vertex computed at the same time as the tree to guide where refinement is desired during runtime such as bounding spheres of the region of influence of a vertex and all its children, or bounding cones of a the normal and its children's normals. Our implementation uses only the bounding sphere of a vertex's region of influence, but could be extended to use any set of attributes desired, though the checks used on those attributes should be fast since they will be ran many times per frame.

In our implementation we follow the example of [3], and compute for each leaf vertex, v, calculate the radius of a sphere centered on v that bounds all neighbors of v. Then, during tree construction, for each parent vertex we compute the radius of a sphere centered on the vertex that bounds the bounding sphere of each of its children.

3.4 Selecting Refinement

At runtime we use a procedure, shouldRefine(v), that returns true if further detail is desired around vertex v. We use two tests to determine if v should refine. First, we check if v is near the camera. The further v is from the camera the smaller its impact on image space and thus the less detail needed in that area, so we allow these regions to simplify. Second, we check if the bounding sphere at v intersects the view frustum by computing the signed distance from v to each face of the frustum and comparing to the bounding radius. If the sphere lies outside the frustum then we allow it to simplify since it does not affect the image. Otherwise the vertex is close and in the frustum and thus should be refined.

3.5 Computing a Level of Detail

At each frame we perform a traversal of the active vertex list (initialized to all vertices in the original mesh) and decide if each vertex should be refined, simplified or left alone. If shouldRefine(v) returns true then we refine v, add each of its children to the end of the active list so they will be considered later, and remove v from the list. Otherwise if the parent of v should not refine we try to simplify v and have to reconsider a few vertices which we simply move to the end of the list. Otherwise we leave it alone. Some vertices will be considered multiple times due to being moved around in the list, but this is required in order to ensure that the mesh simplifies as much as possible while still refining everywhere we want it to.

The triangulation is maintained as it is in [2] by modifying the neighboring triangles each time a vertex splits or an edge collapses.

3.6 Refining or Simplifying a Vertex

The process for refining and simplifying vertices warrants some attention as it must both ensure that the mesh reaches the proper level of detail regardless of the order vertices are considered as well as maintaining a valid triangulation.

Simplifying a vertex is done by first checking all of it's dependencies. If any dependency is not in the active list we cannot simplify this vertex and we move on. Similarly if the other child of it's parent is not active it also cannot simplify since the edge that needs to be collapsed does not exist. Otherwise we get the edge between the vertex and its sibling and perform an edge collapse. We then need to reconsider all neighbors of the parent since they might need to simplify and the parents introduction may have satisfied a previously missing dependency. To do this we simply move them to the back of the active list so they will be considered later in the traversal. This saves us from having to manage everything recursively.

When we want to refine a vertex, v, we don't want to lose detail due to the restraints chosen so rather than checking dependencies, we scan them and if any are missing we find their closest active ancestor in the tree and recursively force them to refine until we introduce all the dependencies of v. We then perform a vertex split on v. To do this we know we need to introduce two triangles as well as split the existing neighbors between the two children, v1 and v2. To do this we store at v the vertices that make the third corner of the two triangles deleted during the formation of v, tv1 and tv2. We then introduce the two triangles (v1, v2, tv1) and (v2, v1, tv2). We also know that any time we find an edge containing tv1 or tv2 we've swapped which child we should be replacing v with. This lets us accurately divide the adjacent triangles between the children. We also then need to check if the children should refine, but rather than recursively do this we simply append them to the end of the active list for later consideration.

4 RESULTS AND CONCLUSIONS

Our largest test case is a loop subdivided Stanford bunny with 160k faces, based on the 40k Stanford bunny provided for HW1 and 4. Points inside the view frustum are refined further and detail falls away outside the frustum. The view frustum is animated to move around the model in a circle in the xz-plane located at a height equivalent to the y value of the center of the bounding box of the model. Animation of the frustum around this specific model is given on a Lenovo T420 laptop with the following specs:

- Intel Core i5-2540M dual-core, 2.60 GHz, 3 MB cache
- 4 GB 1333 MHz DIMM RAM (1 open slot), 8 GB maximum
- NVIDIA Optimus technology and auto-switch between discrete and integrated graphics
- 1 GB Intel HD Graphics 3000 / NVIDIA NVS 4200M, PCI Express x16 graphics card

Frame rates are limited to 60 fps, and we can see in the table below how our selective simplification affects animation speed.

Model	Modifier	fps
160k bunny	Without simplification	4
	frustum 0.4	20
	frustum 0.2	40
	frustum 0.1	50
40k bunny	Without simplification	15
	frustum 0.4	60
	frustum 0.2	60
	frustum 0.1	60
200 bunny	Without simplification	60
	frustum 0.4	60
	frustum 0.2	60
	frustum 0.1	60

Table 1. Without our selective simplification, animation around the model averaged around 4 fps on the largest model. With our

selective simplification, that speed increased to 40 fps, with a frustum size set to 0.2. Decreasing the area of our frustum to a fourth(0.1) of the original area increases speed yet again to 50fps. Increasing the area of our frustum to four times the original area(0.4) decreases speed of visualization to 20fps. For bunnies with less faces, the change in frustum size does not affect speed greatly (if at all).

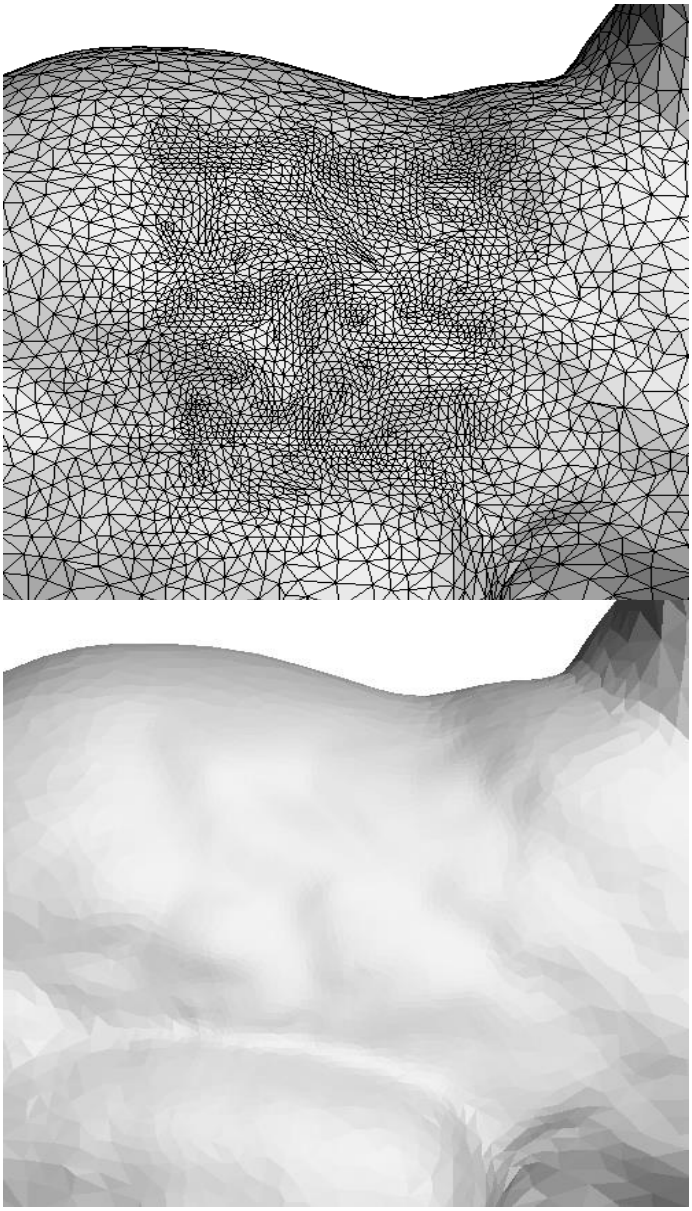


Figure 1. A close-up view of the wireframe of the bunny with frustum set to 0.2, and the same area viewed without the wireframe. Within the frustum the bunny is refined to the leaf nodes and we see vertices from higher levels of our forest the farther we get.

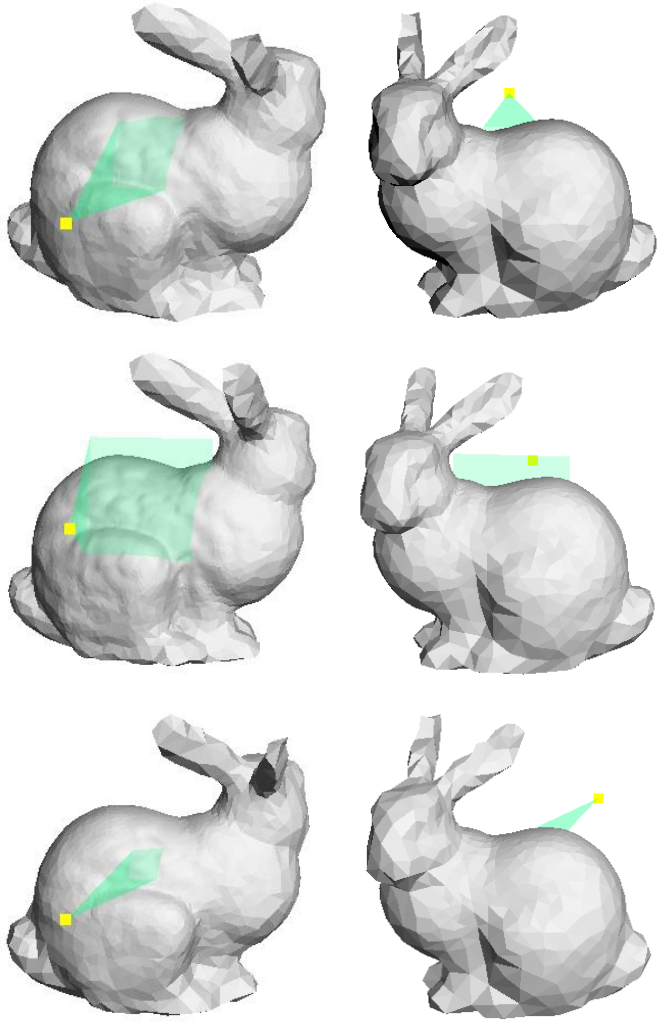


Figure 2. 160k face bunny with standard frustum of 0.2 (top), frustum of 0.4 and area four times as large (middle) and frustum of 0.1 and area a fourth as large (bottom).

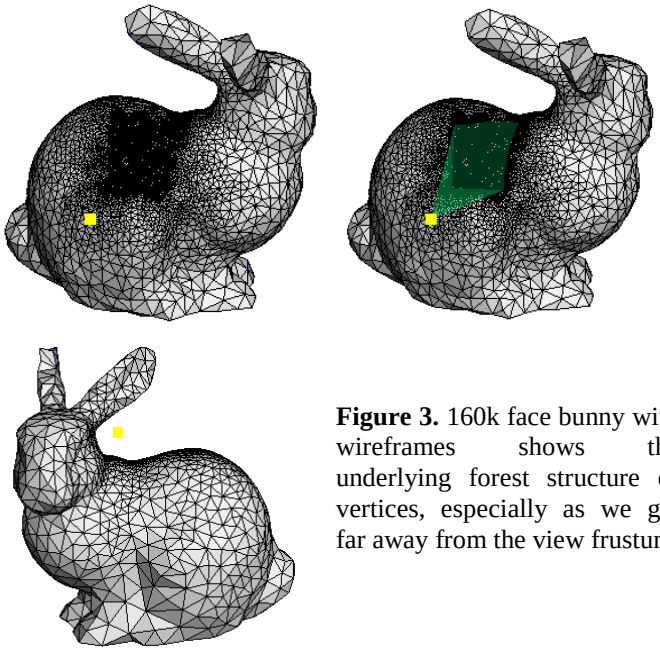
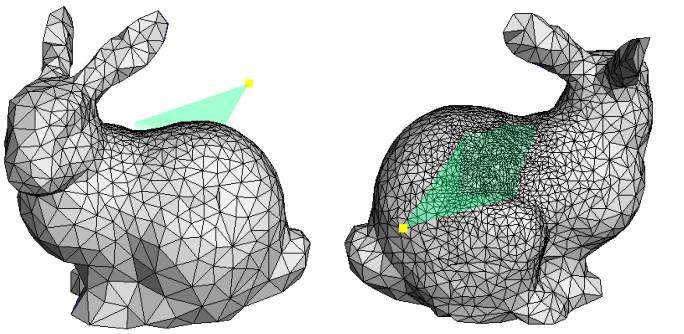


Figure 3. 160k face bunny with wireframes shows the underlying forest structure of vertices, especially as we get far away from the view frustum



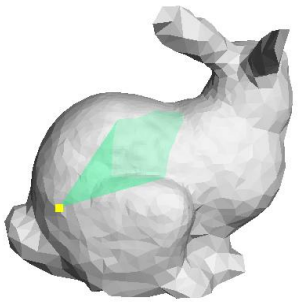


Figure 4. The 40k face bunny has very visible indications that selective simplification would greatly increase the simplicity of the model.

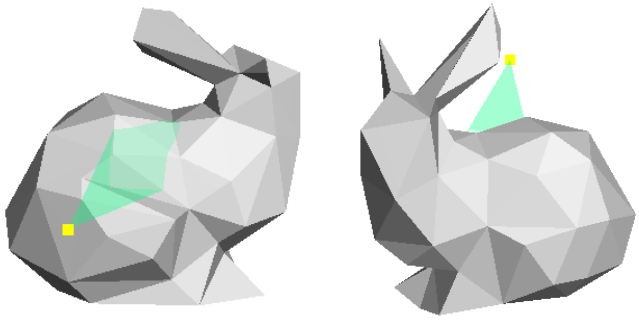


Figure 5. We note that the 200 face bunny has very little if any visible indication that it is affected by selective simplification. This is noted in the results where simplification did not give us any increases in fps.

5. LIMITATIONS AND FUTURE WORK

During early implementation we discovered a bug when trying to build the tree on open meshes in which not all edges to child vertices were properly moved to the parent and would cause segmentation faults. After significant time trying to hunt down the source of the issue we decided development time was best spent elsewhere. As a result our implementation only works on closed meshes. This is not a limitation of the system, but rather of our specific implementation.

We do, however, over-constrain the tree resulting in suboptimal simplification. Moving to a less constrained tree as in [3] would improve on this significantly. We also do not do the calculations during tree construction required to prevent self intersection so at times (especially in thin areas of the mesh) the mesh will self intersect, though it does not do this when in the view frustum as self-intersections only appear in the lower levels of detail.

We would also like to improve the performance of tree construction to allow for larger test cases. While it does not impact the final runtime it poses a significant barrier to introducing new models and test cases.

6. REFERENCES

[1] Hoppe, H., Progressive meshes, Computer Graphics, (SIGGRAPH ‘96 Proceedings), 1996, pp. 99-108.

[2] Xia, J. And Varshney, A., Dynamic View-Dependent Simplification of Polygonal Models, In Visualization ‘96 Proceedings, IEEE 1996, pp. 327-334.

[3] Hoppe, H., View-Dependent Refinement of Progressive Meshes, Computer Graphics (SIGGRAPH ‘97 Proceedings), 1997, pp. 189-198.

7. WORK BREAKDOWN

Nathan worked on initial tree construction and basic setup of the data structure. I worked on coding a file format (so the tree did not have to be built every time we wished to test selective simplification) and placement of modeled view point and frustum. The rest was worked on together. This project took approximately 40 man hours in total for coding and testing.