View-Dependent Selective Refinement for Fast Rendering

Kyle Brocklehurst
Department of Computer Science and Engineering
The Pennsylvania State University
kpb136@psu.edu

***Abstract*—Triangle meshes are used for surface representations of objects in almost all computer graphics software and hardware. The computational demand of rendering increasingly complex geometry has necessitated techniques to simplify or remove geometry from the rendering pipeline without negatively affecting the users visual experience. Here we present an approach using progressive meshes to build a very simplified version of a mesh, and we introduce a simple criteria that can be evaluated by a GPU during rasterization to determine areas to be refined, leading to a scalable representation of object surfaces that renders visually similar output at a greatly reduced polygon count.**

I. Introduction

The structuring and simplification of surface meshes is a well-explored area of research, but many methods can only compress a mesh to some finite number of levels of detail [1]. Traditional graphical applications switch between models with different levels of detail as object move closer or further from the camera. This works to reduce the total amount of geometry to be rendered, but can result in noticable visual artifacts as an object hits a distance threshold and suddenly changes sharply in detail.

We are intrigued by the work of Hoppe [2] as it allows for individual, targeted refinements to a compressed mesh and thus a near-continuous level of detail. This method allows for selective refinement, which means that it can be used to refine only areas of the mesh that meet some criteria. We propose a general and broadly applicable weighting function for vertex importance. This function is dependent only on object geometry and view location, information that is known to graphics hardware during rasterization. Thus, it would be appropriate for our method to be implemented on a GPU. This allows the GPU to refine the mesh only in areas of greatest importance under the current view. We show that a software implementation of our method can substantially reduce the overall polygon count of a surface while keeping detail high in the areas where it is needed from a specific camera.

II. Related Work

Schroeder et al. [1] present a method capable of removing 90% of the vertices from a surface mesh while retaining visually-pleasing similarity to the original quality. Their method, however, acts over the entire surface equally, and thus does not take advantage of the viewers position, which is commonly known in graphical applications.

Progressive meshes, introduced by Hoppe [2], introduce a representation of a mesh that allows for heavy simplification and retains the information needed to refine back to the original level of detail. Their method allows for either global or selective simplification / refinement of a mesh, and is also useful in that the mesh can be rendered as it is being transmitted to yield an effect of iterative refinement as transmission progresses. This method allows for selective refinement, but the author offers little discussion of what refinement criteria may be, noting that it is largely specific to particular applications. His work merely notes that a function call to an application with higher understanding of vertex importance could be used to specify, for each vertex, whether it should be simplified or refined.

III. Compressing the Mesh

Compression of the mesh is performed similarly to [2]. Compression can be accomplished solely by the edge collapse operation, illustrated in Fig. III. An edge collapse replaces an edge with a vertex, removing to triangles from the mesh. The information needed to invert the edge collapse and perfectly restore the mesh can be recorded when the collapse is performed and applied later as a vertex split operation.

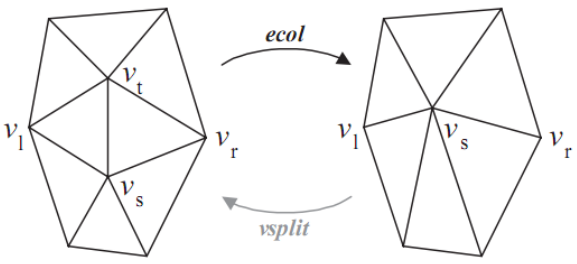


Fig. 1. The edge collapse and vertex split operations are the heart of Hoppe's method [2]. An edge collapse takes an edge (V_s to V_t above) and replace it with one vertex, removing two triangles in the process. A vertex split inverts one edge collapse, restoring the mesh perfectly to how it was prior to the collapse.

We represent a mesh with a vector V: a vector of vertex locations in 3D, S: a vector of colors (S_i is the color at V_i), K: the connectivity vector (in which each entry has 3 index values indicating the index of the vertices that create a triangle of the surface), and E: a vector of all edges on the surface.

Compression progresses by iteratively collapsing edges until we have reduced our mesh to a specified number of triangles. In each iteration, the edge collapse with lowest cost is performed. Cost is illustrated in Fig. III. It is formulated as $cost = (Gerr)^2 + (length)^2 + sqrt(length * \Delta color)$, where $\Delta color$ is the distance between the colors at $V1$ and $V2$ in RGB space. The $(Gerr)$ term is the distance from the vertex that would be created between $V1$ and $V2$ and the true closest point on the original mesh (shown in black); it prevents the compression from removing important geometric detail from the surface. The $sqrt(length * \Delta color)$ term prevents the compression from removing changes in color, or blurring the changes by stretching them over a triangle edge that is very long. The last term $(length)^2$ works to keep all edges of roughly equal size. This is described in [2] as a spring energy equation, but we find this formulation unneeded as the term need only depend on $(length)^2$ to keep compression from focusing too strongly in one area.

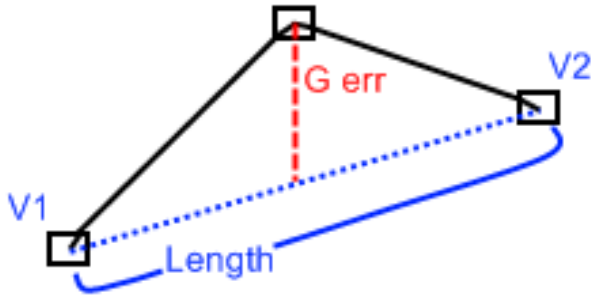
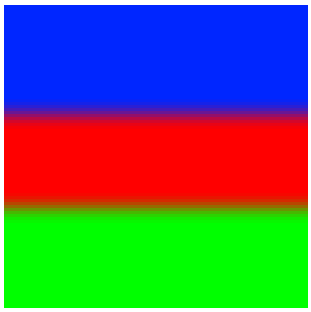


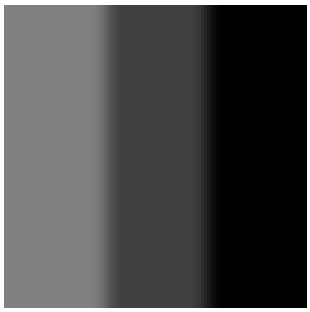
Fig. 2. The black squares represent actual values of the original mesh. $V1$ and $V2$ are vertices of an edge of a somewhat compressed mesh, such that the edge between them (blue line) is spanning multiple edges of the original mesh. The cost of collapsing this edge is dependent on the its length as well as on the distance from the midway point of the edge to the original mesh (red line). Cost is also dependent on the difference in color values at $V1$ and $V2$ (not illustrated).

Initially, cost is computed for all edges in E and E is sorted by cost. In each iteration, the lowest cost edge is collapsed and all affected edges (those that had been connecting to $V1$ or $V2$ as illustrated in Fig. III) have their cost recomputed. Then E is sorted again and the process repeats until the specified number of triangles have been removed. Figures III and III show the results of compressing a surface mesh to 25% of the original number of triangles.

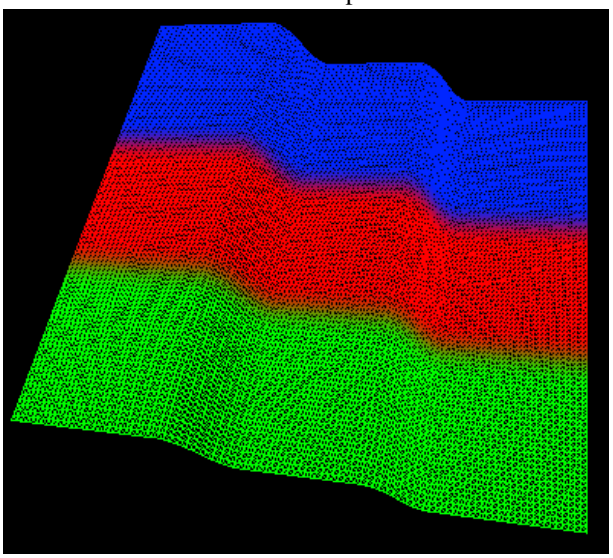
An interesting change that I make from [2] is that I do not delete any triangles during an edge collapse. I use an additional vector of boolean variable to note, for each triangle in K, whether or not that triangle should be displayed. Thus, I can simply turn triangles off as I collapse edges so that they can be turned back on during refinement, sparing the time to delete and reallocate the two triangles in each operation. This means that triangles are not removed from memory, so simplification of the surface does not reduce the amount of memory used to store the mesh, but this is of little concern as modern graphics hardware has an abundance of memory and the triangles that are turned off need not be rendered, and thus do not slow the rendering process.



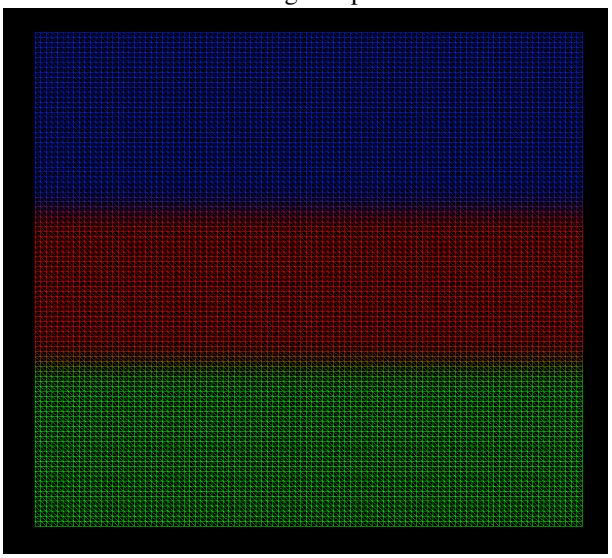
Colormap



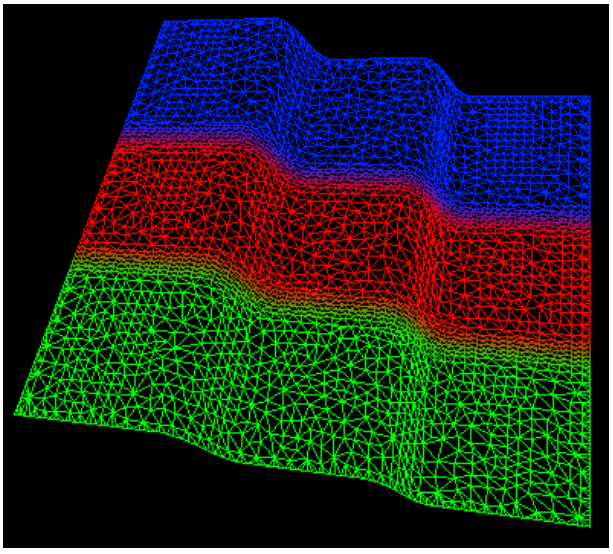
Heightmap



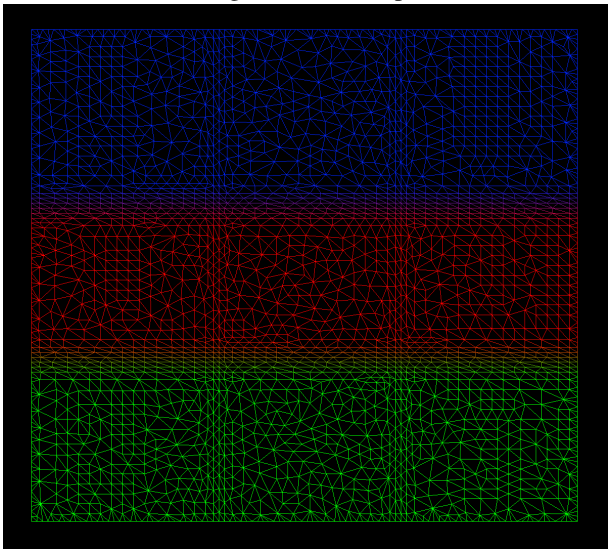
Original Detail



Original Detail (top)

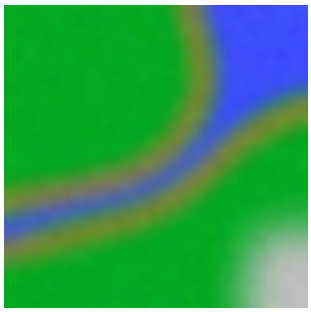


25% Detail

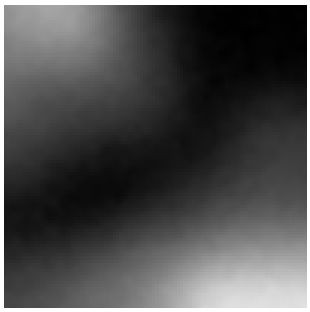


25% Detail (top)

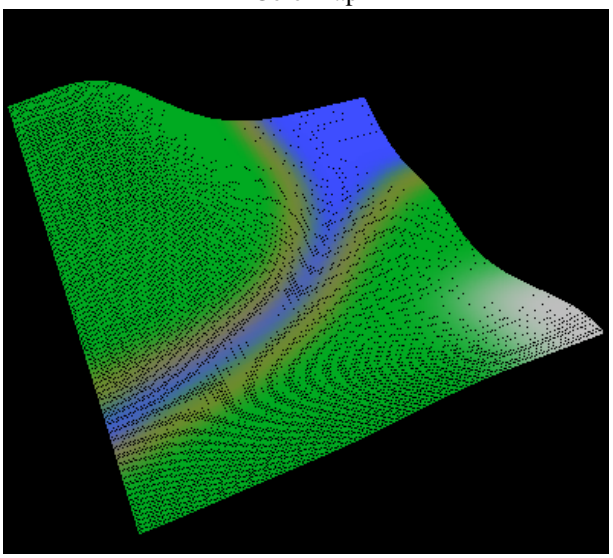
Fig. 3. A surface mesh of a terrain is created using the colormap and heightmap shown. At original detail, the mesh has 20,000 triangles. We have reduced this to 5,000 triangles with little decrease in quality. Not how detail is highest near changes in color or height.



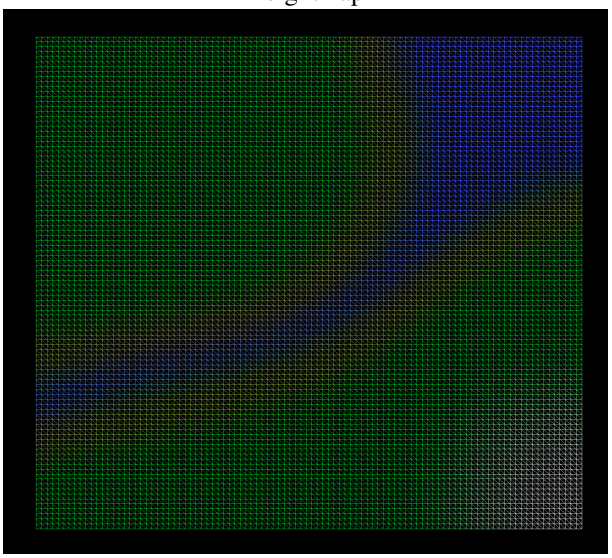
Colormap



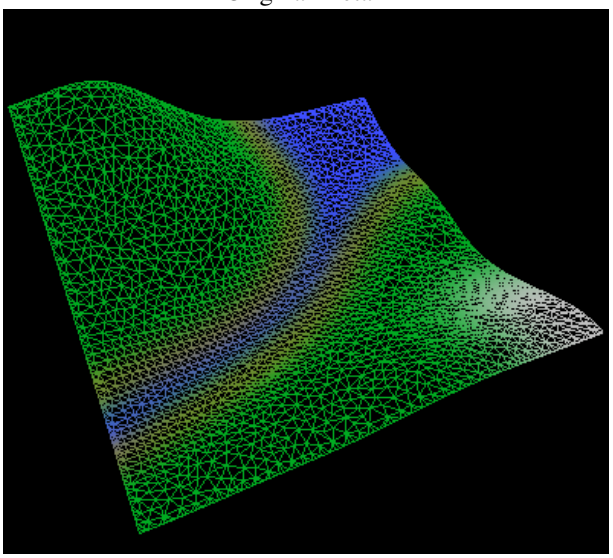
Heightmap



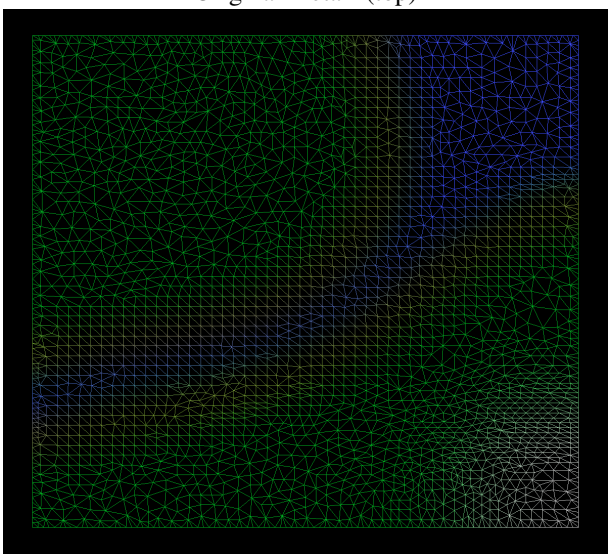
Original Detail



Original Detail (top)



25% Detail



25% Detail (top)

Fig. 4. A surface mesh of a terrain is created using the colormap and heightmap shown. At original detail, the mesh has 20,000 triangles. We have reduced this to 5,000 triangles with little decrease in quality. Not how detail is highest near changes in color or sharp changes in height.

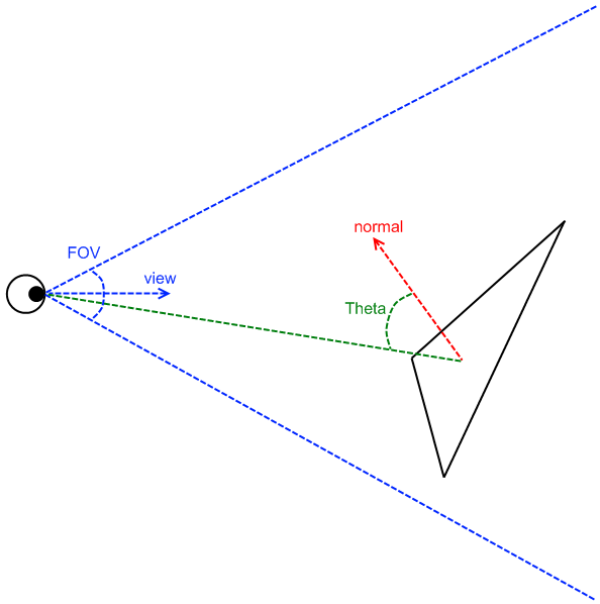


Fig. 5. The information that is known to the GPU during rasterization of a triangle includes the location, view direction, and FOV (anguar field of view) of the camera, the location of the triangles vertices, and the normal to the triangle's surface. The theta value shown is simply the angle between the normal and a vector from the eye to the triangle's center (α=0.5 β=0.5 in barycentric coordinates).

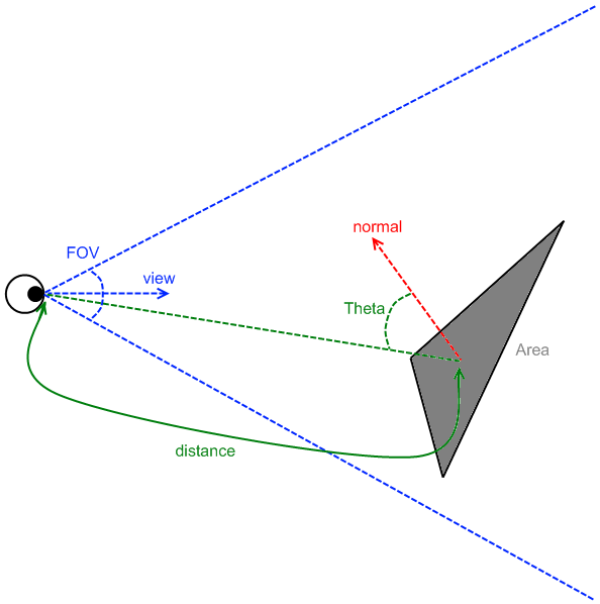


Fig. 6. In determining the importance of a triangle, we also utilize the triangles area and the distance from the eye to the triangle's center. Both of these values are calculable from the information presented in Fig. IV.

IV. Selective Refinement

We require a scheme to selectively refine the lattice to maximize quality of the image under the current view conditions while keeping the number of triangles being rendered low and the time required to render quick. Rendering progresses first through rasterization, a process using only the properties of the camera and the geometry of the objects to position each triangle in the image. One the geometry of the scene has been rasterized, the more time-consuming process of shading the geometry uses a multitude of other knowledge of the scene, such as the lights and their properties, reflection, transparency, or other effects such as fog to make a scene look as realistic as possible. The geometry of a scene is established during rasterization, the shading process cannot change it, merely change the colorization of what has been rasterized to yield a more convincing image. Thus, a selective refinement method that could intelligently decide what areas of the mesh need refinement to appear more accurate under the current view must be performed during rasterization, rather than shading, and thus must use only the information available during rasterization: the physical geometry of the triangles and the properties of the camera. This information is visualized in Fig. IV.

We use the information available during rasterization to determine whether or not a vertex split should be performed on the vertices of each triangle. As each triangle is rasterized, if it should be refined, a vertex split is performed on every vertex of the triangle for which vertex split exists (only vertices that were created as a result of edge collapse can be split). When an edge is collapsed, an entry is created in a vector of vSplits, or vertex splits. A vSplit contains the index within V of the vertex that it splits, the indices within K of the two triangles that are turned back on after the split, and the information needed to reconnect neighboring triangles to the vertices at the end of the edge being recreated (the triangles neighboring V_s and V_t in Fig. III.

To determine whether or not to perform vSplits at the vertices of a triangle being rasterized, we first verify that the triangle is in view from the camera, if so, we formulate an equation for the importance of the triangle under the current view. We specify $importance = area/(distance * tan(FOV)) * max(-cos(\theta), sin(\theta))$, visualized in Fig. IV. Where $area$ is the area of the triangle. $1/(distance * tan(FOV))$ models the projected size that an object would have at the current distance. $max(-cos(\theta), sin(\theta))$ weights the importance of the angle between the triangle's normal vector and a vector from the eye to the center of the triangle. This value is highest when the triangle is either orthogonal to or tangent to the current view; we find these to be the most important cases as orhtogonal triangles occupy maximal screen space for their size, making refinement significant to quality, and triangles tangent to the current view make up much of the contour of the object, making refinement there also very important so the silhouette of the object is as accurate as possible. All triangles with an importance above a given threshold have vSplits performed on each of their vertices for which a vSplit exists. We find that a threshold of 0.3 produces pleasing results, but this is largely dependent on the needs of the system: a balancing act to keep quality high and polygon count low, which can be tuned by a user or learned over time by trying to acheive a consistent frame rate.

Our function of vertex importance results in selective refinement that is focused on areas in view of the camera, near the camera, and at the most important areas of the object. In Fig. IV it is clear that the refinement decreases with distance

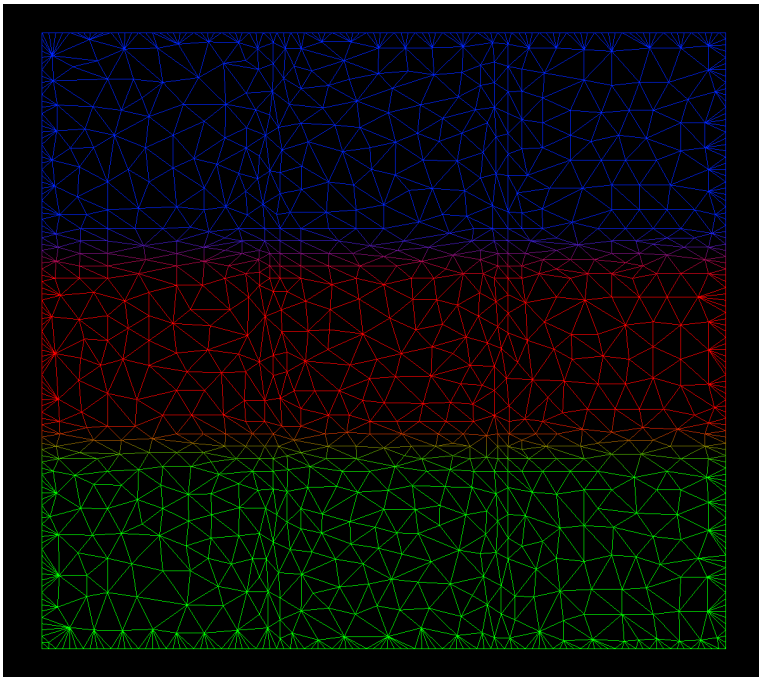
from the camera, and it is intuitive that it is less necessary in areas where perspective makes objects appear smaller. In Fig. IV, it can be seen that the refinement is more concentrated on the side of the white hill that directly faces the camera.

V. Results

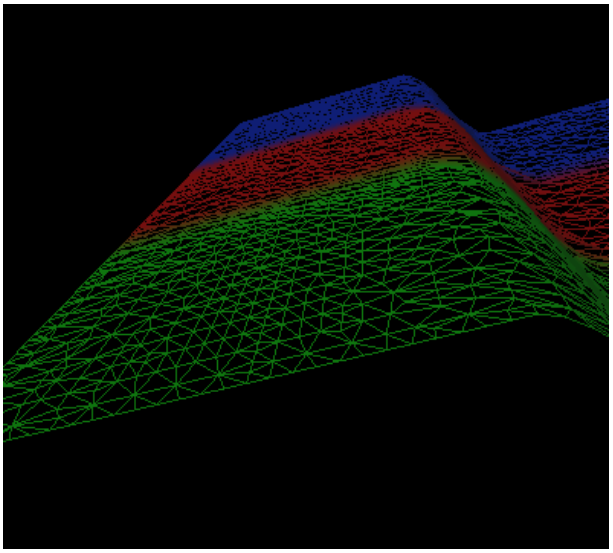
Our method is implemented in OpenGL using simplification of the mesh on the CPU to verify the level of simplification and selective refinement that could be acheived on a GPU. Currently, initial mesh simplification (to 10% the initial number of triangles) takes about 5 seconds for a mesh initially containing 5000 triangles and about 30 seconds for a mesh initially containing 20000 triangles. Selective refinement from a specified view takes less than a second. Actual speed and computational advantages could only be realized through GPU programming and, for some hardware, redesign to allow for the interruption of rasterization for vSplits. We demonstrate that our method is able to significantly simplify a mesh (easily to 10% the initial number of triangles), and that selective refinement yields significantly improved views from the current camera in Figures V and V.

VI. Conclusion

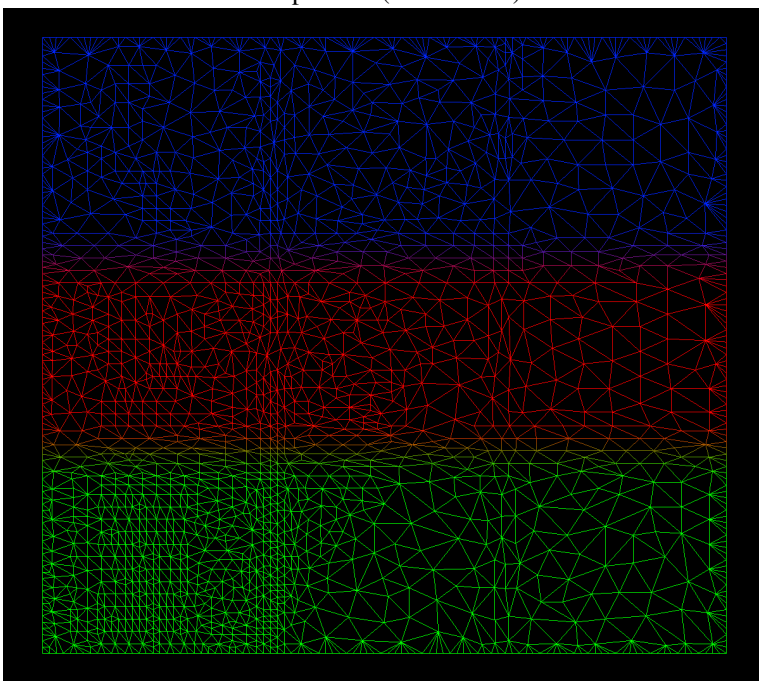
We present a modification of progressive meshes [2] built around the idea of improving rendering performance on a GPU. We introduce a modified version of edge collapse cost which proves effective in guiding significant simplication of an input mesh while depending only on information local to the edge being considered. We formulate a measure of a triangle's importance under a current view using only the triangle's geometry and the camera properties, thus making it appropriate for evaluation during rasterization, when vertex splits can be easily performed to enhance the surface where needed. We have implemented our method and shown that it produces pleasing results, adapting well to a variety of views and focusing refinement on areas where significant and noticeable detail is recovered.



Compressed (10% Detail)

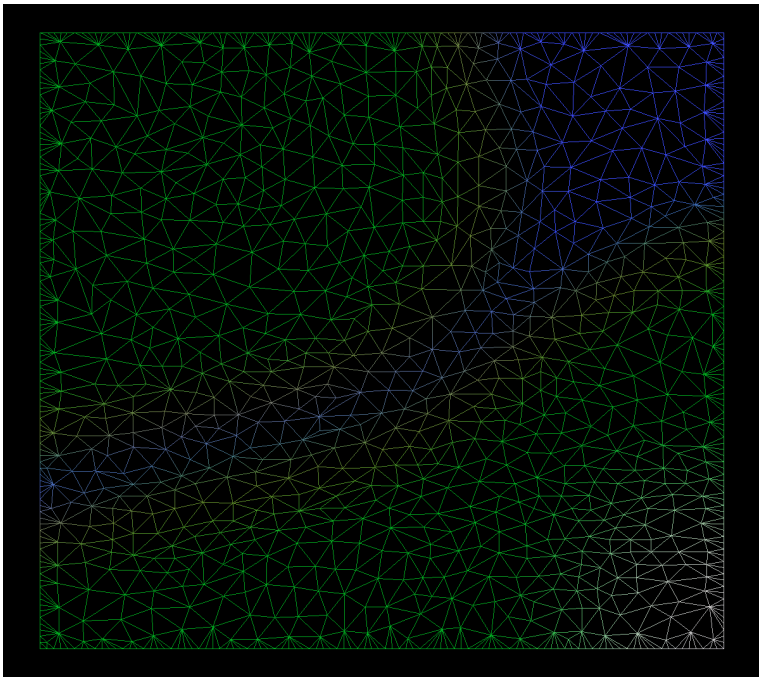


Current View

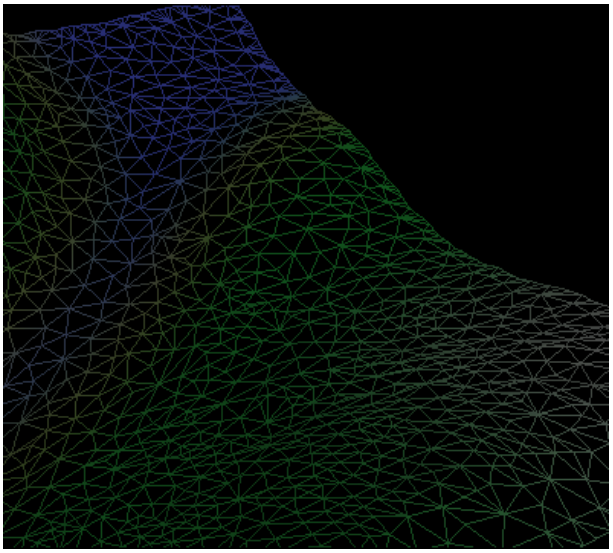


Selectively Refined

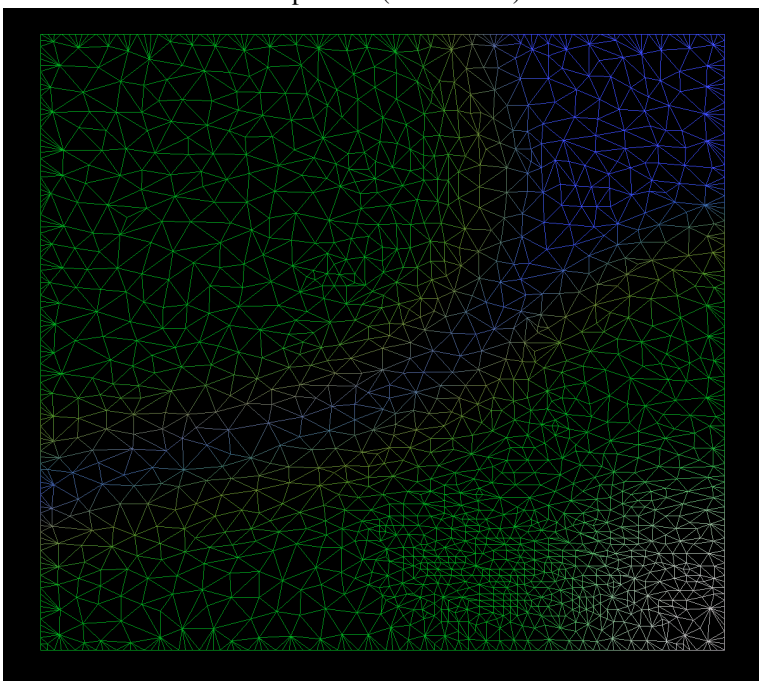
Fig. 7. We compress the surface from the colormap and heightmap of Fig. III to 10% of the original numer of triangles, then selectively refice from the given view. Note how detail is only increased in view of the camera and is most increased nearer the camera.



Compressed (10% Detail)



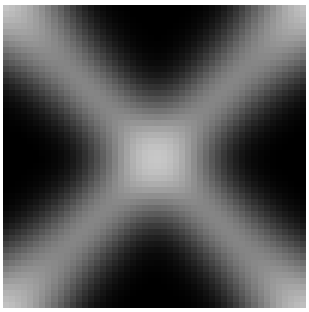
Current View



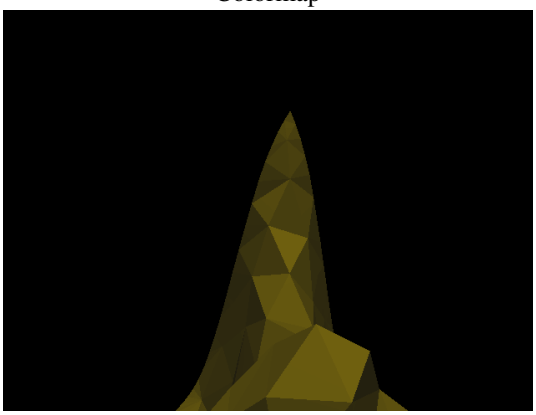
Selectively Refined

Fig. 8. We compress the surface from the colormap and heightmap of Fig. III to 10% of the original numer of triangles, then selectively refice from the given view. Note how detail is only increased in view of the camera and is most increased in important areas nearer the camera.

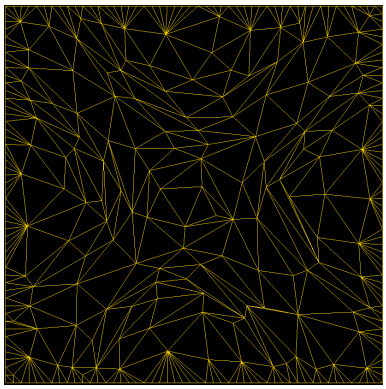
Colormap



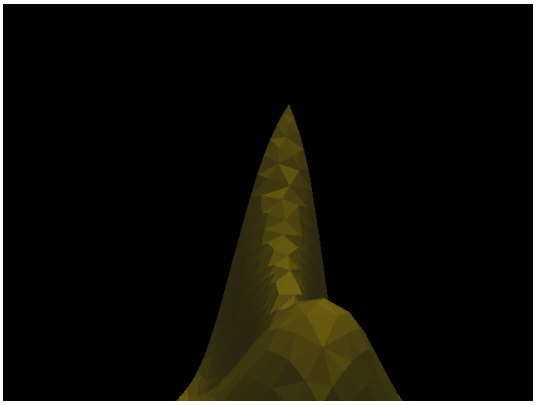
Heightmap



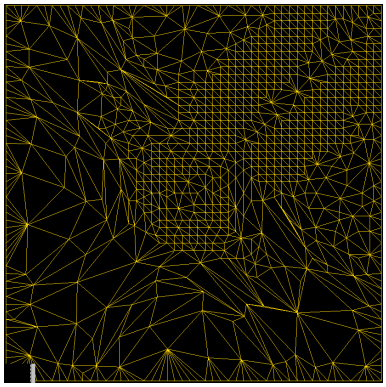
10% Detail



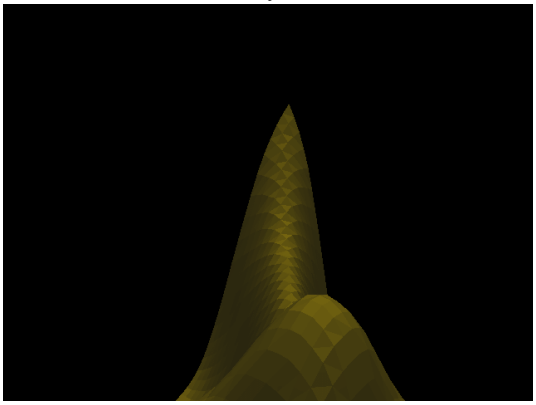
10% Detail (top)



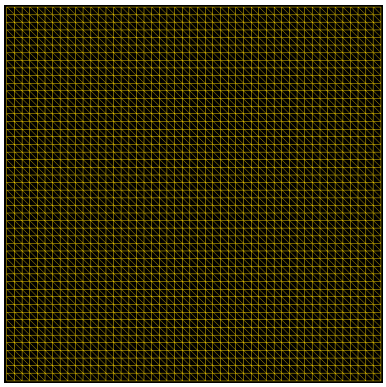
Selectively Refined



Selectively Refined (top)



Original Detail

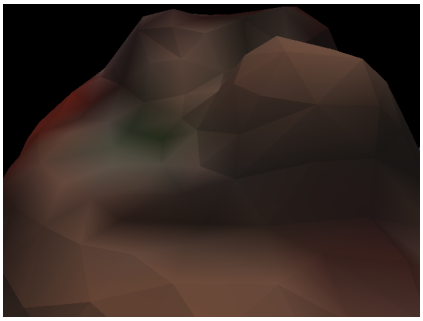


Original Detail (top)

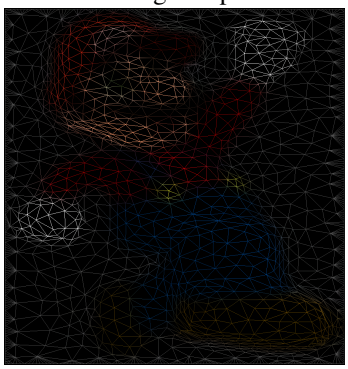
Fig. 9. We show the original mesh, compressed mesh at 10% the number of triangles, and selectively refined mesh for one view of the colormap and heightmap shown.

Colormap

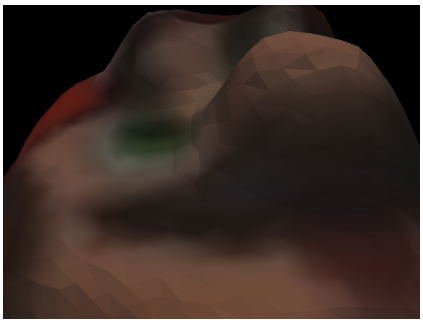
Heightmap



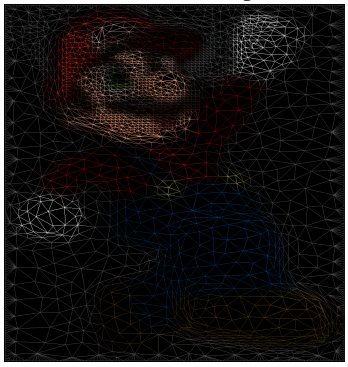
10% Detail



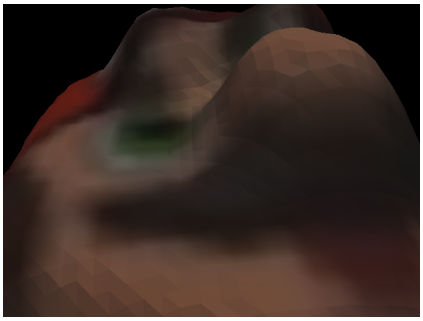
10% Detail (top)



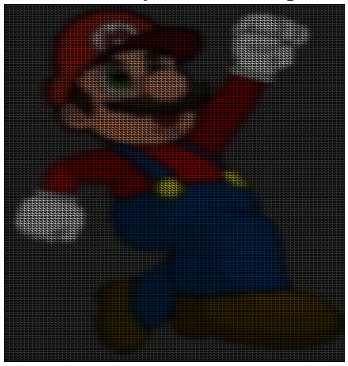
Selectively Refined



Selectively Refined (top)



Original Detail



Original Detail (top)

Fig. 10. We show the original mesh, compressed mesh at 10% the number of triangles, and selectively refined mesh for one view of the colormap and heightmap shown.

REFERENCES

[1] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen, “Decimation of triangle meshes,” in *ACM SIGGRAPH*, 1992, pp. 65–70.
[2] H. Hoppe, “Progressive meshes,” in *ACM SIGGRAPH*, August 1996, pp. 99–108.