Terrain LoD via ROAM

John Schwartz and Zachary Cross

1. Abstract

Rendering large terrain meshes can be difficult, as the level of detail involved is often too high in areas that we don't need it to be, and too low in areas where it would be better defined. However due to the regular grid-nature of height maps as opposed to traditional 3D meshes, some mathematical improvements can be made to this.

2. Related Work

Our work is based primarily off of the work done by Mark Duchanaieu et al on their paper entitled ROAMing Terrain: Real-time Optimally Adapting Meshes.

3. Data Structures

The ROAM algorithm requires a large data set to take values from in order to draw triangles at various levels of detail. For this, we chose to use a fractally generated terrain based on the Diamond Square algorithm. This allows us to generate a set of data as detailed as the user requests. The Diamond Square algorithm works by seeding four corner values, an amount of perturbation, and a roughness coefficient. Values are then chosen at the midpoints of the resulting squares and diamonds, and perturbed up or down. After each iteration, the roughness lowers this perturbation, creating either rough or smooth surface depending on the values.

Due to the nature of this algorithm, the highest or lowest point is almost always in the exact center. We used a novel approach of taking an input file of icon values (M representing a mountain, tall, rough, and high perturbation, etc) and sewed these together, averaging not only the height values, but also the roughness and perturbation values across the map. This results in smooth gradients and a lot of control for the user to create interesting landscapes relatively quickly.

The most important data structure for the ROAM algorithm, the Binary Triangle Tree, was introduced by Duchanaieu et al. The structure is able to store right isosceles triangles in a way that allows for easy refinement, while at the same time simplifying the process of preventing T-junctions in the resulting triangle mesh.

The Binary Triangle Tree consists of a triangle which keeps track of five other triangles: its left child, right child, left neighbor, right neighbor, and base neighbor. These five triangles contain all the information needed to create adaptively refined triangle meshes.

Together, the two children fill up the entirety of the area occupied by the parent triangle. The base neighbor is the triangle adjacent to the current triangle along their hypotenuses. With the hypotenuse on the bottom, the current triangle's left neighbor is on the left, and the right neighbor is on the right.

4. Method Overview

The main algorithm works as follows. First, a height field, stored as a nested vector, is stored in memory. Then the two base triangles that make up the square piece of terrain are initialized. Now, a series of splits are called, refining the original two triangles down until the given error metrics either tell the split function to stop, or the maximum number of binary triangle tree nodes are reached.

Triangle are prioritized via different error metrics, and then placed onto a priority queue. The top triangle on the priority queue is then checked to see if it is as refined as it can get. If so, it removed from the queue. Otherwise, split is called on the triangle before removing it form the queue.

We implemented three different metrics to judge whether or not a given triangle needed to be split. First, we calculated the difference between the interpolated height of the middle of the hypotenuse given by averaging the left and right vertices and the actual height value of the middle of the hypotenuse from the actual height field. Triangles with a larger difference (e.g. a larger error between the current level and the next level down) were given a higher priority value.

Second, we took into account a triangle's distance from the 'focus point.' The focus point is a point we placed on the terrain which simulated a potential camera. When used in practice the focus point would be the actual camera itself, but this way we were able to see the results of the algorithm from a distant viewpoint.

Finally, we implemented a visibility check. We made our focus point 'look' towards the center of the terrain. Any triangle whose center was not in the field of view was taken off of the split queue.

5. Results

Our implementation is able to create an optimal mesh given the metrics we programmed. On an IBM T60 laptop, the program drops down into 'interactive' frame rates at around two hundred fifty thousand triangles. With some more optimization, as well as possible inclusion of graphics hardware in the algorithm, our implementation should be able to retain real-time frame rates at even greater triangle count.

6. Conclusion

The ROAM algorithm is, at its core, a very simple, expandable algorithm. There are a multitude of different error metrics which may be added in to the basic set of code depending on the problem at hand.

Unfortunately, as it stands, the ROAM algorithm has fallen behind more current terrain LOD algorithms because it does not

take advantages of modern graphics hardware.

Also, our code does not allow for the number of triangles to be dynamically changed; it is hard coded into roam.cpp.

7. Notes

Our project is limited in many ways, a problem that we recognize, though have been unable to rectify as of yet. Our method does not take advantage of frame coherence in order to merge already-split triangles. Rather, our method recomputes the optimal triangulation for every frame. Nor does our method take advantage of full view frustum culling. We attempted to implement view frustum culling, but our implementation was not working as well as was to be expected, so it was removed. In its place, however, triangles in the view frustum are not split as readily as others.

John created the terrain generation algorithm, though he has not expanded it yet as that was deemed too low of a priority. Zachary started work on getting the basics of the ROAM algorithm set up, though John was the first to make serious headway on the algorithm. Zachary then cleaned up the code and reformatted the code into classes. At this point, both John and Zachary worked together to implement the rest of the algorithm. John then cleaned up the final code, while Zachary worked on this report.

8. Bibliography

Duchaineau, M., Wolinski, M., Sigeti, D., Miller, M., Aldrich, C., and Mineev-Weinstein, M. "ROAMing Terrain: Real-time Optimally Adapting Meshes", IEEE Visualization '97 Proceedings, 1997.

Bryan Turner, “Real-Time Dynamic Level of Detail Terrain Rendering with ROAM”, http://www.gamasutra.com/features/20000403/turner_01.htm.

Hoppe, H. "Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering" (http://www.research.microsoft.com/~hoppe)