HORIZON OCCLUSION CULLING FOR REAL-TIME RENDERING OF HIERARCHICAL TERRAINS USING PROGRAMMABLE GRAPHICS HARDWARE

Hacer Yalım

Department of Computer Engineering
Middle East Technical University
November 2003

Abstract

Terrain visualization is a difficult problem for applications requiring accurate images of large datasets at high frame rates, such as flight simulation and ground-based aircraft testing using synthetic sensor stimulation. The main problem is to maintain dynamic, view-dependent triangle meshes and texture maps that produce good images at the required frame rate. This report describes a technique to improve the horizon occlusion-culling algorithm for hierarchical terrains using programmable graphics hardware techniques so that the hierarchical terrains could be rendered in real-time efficiently. The occlusion horizon is a well-known algorithm. The main issue of this work is the adaptation of this algorithm to hierarchical terrains. The algorithm will be implemented with the usage of programmable graphics hardware to accelerate the rendering time. The chosen occlusion algorithm is simple to implement and requires minimal pre-processing and additional storage. The occlusion horizon is constructed as the terrain is traversed in an approximate front to back ordering. Regions of the terrain are compared to the horizon to determine when they are completely occluded from the viewpoint. Quad trees are selected to construct the hierarchical structure for the terrains. By this work, the improvement of rendering time using hardware programming techniques for occlusion culling on hierarchical terrains will be observed.

Table Of Contents

1 Introduction 1
2 Related Work 3
3 Terrain Rendering Algorithm 4
4 Horizon Occlusion Culling 4
4.1 Front-to-Back Ordering 5
4.2 Occluders and Occludees 5
4.3 Horizon Representation 6
5 Hardware Programming 7
5.1 Vertex Programming 7
6 Current State and Future Work 8
7 Conclusion 9
8 References 9

1 Introduction

Real-time navigation of detailed terrain model is important for many applications. Since the early days of computer graphics, the military has used flight simulators to train pilots. Flight simulators for commercial airlines soon followed. Simulator technology inevitably made its way into video games. Today real-time terrain navigation routinely used in other applications such as visualizing a proposed road construction project or data captured by a satellite. The quality of these applications often depends on the size and detail of the terrain models.

The huge size of terrain models presents several problems. Large models require a lot of memory. Models covering only a few square miles can consist of millions of polygons. Texture-maps for terrain models usually have a high resolution in order to provide sufficient detail at close proximity, which makes them very large. Often only part of the model can fit in memory at any time. Rendering large terrain models at real-time frame rates can be challenging. While rendering hardware has made great advances in recent years, even the best hardware can render only a few million polygons per second. Memory bandwidth limitations restrict the amount of texture that can be rendered to a few hundred megabytes per second. These limitations are not a problem when the viewer is close to the ground looking down because large portions of the model fall outside the view frustum and may be culled away. However, when the viewer is looking at the model from high altitude or is looking out horizontally at ground level, arbitrarily large portions of the model may be visible. Since it is currently not possible to render millions of polygons with many megabytes or even gigabytes of texture at high frame rates, rendering large terrain models in real-time requires specialized algorithms.

One solution is to render the terrain using an adaptive level of detail (LOD). Features in the distance do not need to be rendered with the same fidelity as those that are close to the viewer. A smaller, coarser representation may be used for distant features without any noticeable degradation of image quality. Hierarchical terrain representations such a quadtrees are especially well suited for adaptive LOD and have been used extensively in computer graphics and GIS systems. Quadtrees facilitate choosing a representation with an appropriate resolution for different parts of the model. Adaptive LOD can drop the number of polygons in a given frame from millions down to tens of thousands. Adaptive LOD yields similar exponential reductions in the amount of texture required for a given frame. This is because fewer, lower resolution texture maps can be used to cover large portions of the model.

While adaptive LOD dramatically reduces the polygon count for both high altitude views and horizontal views close to the ground, the polygon counts for horizontal views remain considerably higher than those for other views. This is a characteristic of terrain models. When viewed from above, all of the geometric detail becomes compressed in the viewing direction, so a coarser LOD may be used. Horizontal views require a finer resolution to capture the details of the terrain profiles, which leads to a higher polygon count. This is unfortunate because these views tend to be very common in 3D terrain visualization applications. However, the rough features of a terrain that require an increased polygon count also afford an opportunity for optimization, since they often occlude large portions of the terrain model. Occluded regions need not be drawn because they make no contribution to the final image. Figure 1 shows the reduction in rendered geometry due to occlusion culling.

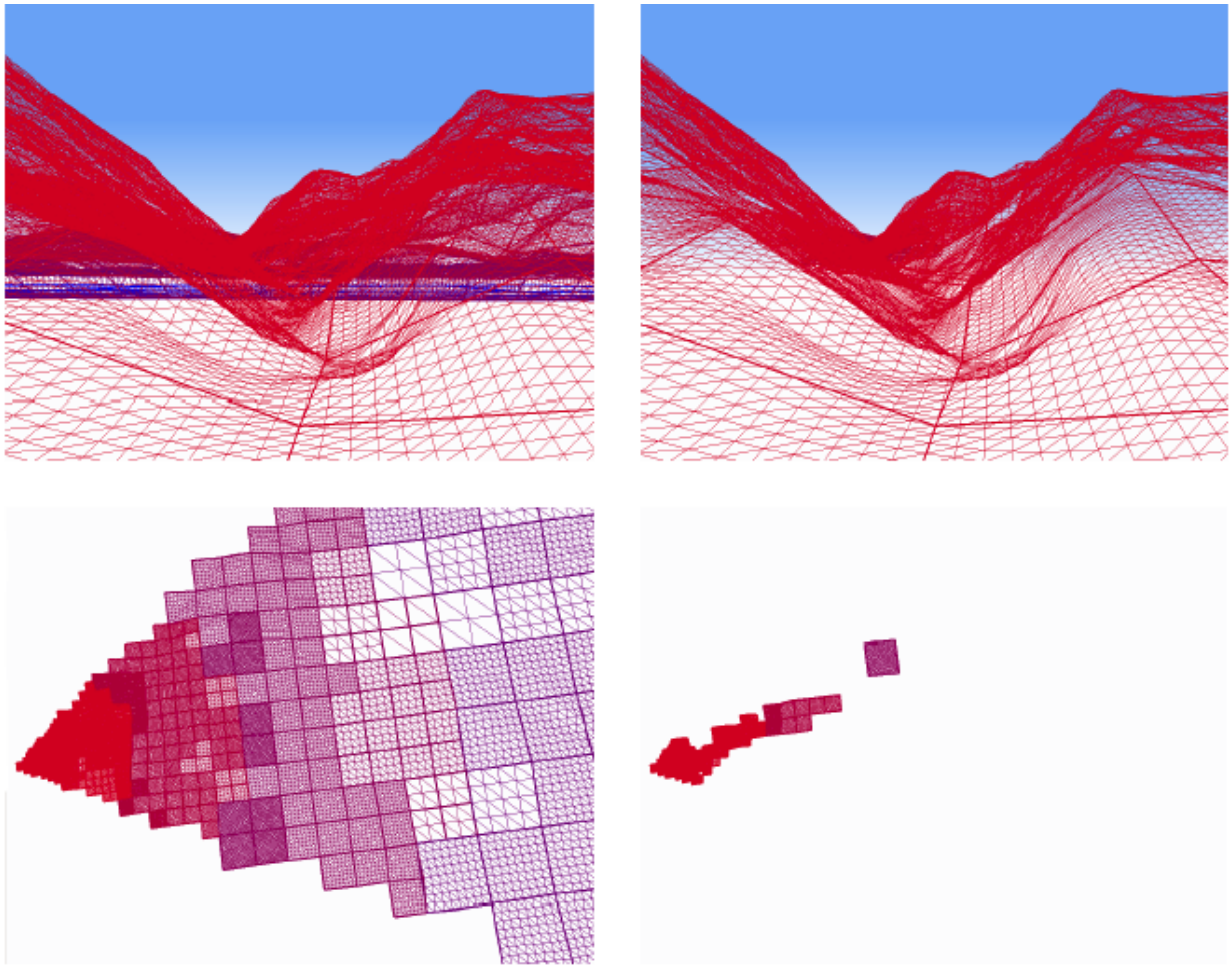


Figure 1: Top-left, view of a terrain without occlusion culling. Top-right, occlusion culling is enabled. The bottom row shows the terrain elements drawn for the corresponding views in the top row.

While some work has been done to perform occlusion culling for terrains, most algorithms pre-compute visibility and require significant pre-processing and storage. In this report I explain an algorithm to compute occlusions in terrains on the fly. It is based on a well-known technique. With a height field, anything below the horizon line occluded. By tracking the horizon line of the height field as it is rendered in front-to-back order, self-occlusions within the height field may be detected. The horizon effectively fuses the occlusion of many individual parts of the terrain. Terrain elements that fall completely below the horizon are culled. The rest of the terrain elements are sent to the rendering pipeline and rendered using programmable hardware. Their information is used to update the visibility horizon as it sweeps through the scene. The improvement done in this work is the application of the occlusion horizon to hierarchical terrains using programmable graphics hardware. The algorithm requires minimal pre-processing and additional memory. Hardware programming techniques are used in the implementation.

After briefly discussing work related to this subject in Section 2, I give an overview of the terrain-rendering algorithm I use in Section 3. Section 4 explains the details of the occlusion-culling algorithm. In Section 5, I introduce the usage of the programmable graphics hardware for this work. Finally I make some concluding comments and discuss directions for future work.

2 Related Work

There is a vast amount of research in the area of visibility and occlusion culling. An excellent survey of visibility techniques is provided by Cohen-Or et al [7]. Visibility algorithms can determine visibility either in a precomputation step or on the fly. Precomputed visibility algorithms pre-process a scene to determine which objects are visible from points or regions in space. At run time, only the objects in the pre-calculated set of visible objects for the current viewpoint are drawn. This can result in enormous savings when the size of the set of visible objects is small compared to the number of objects in the scene. The major drawbacks of pre-processing algorithms are that they are limited to static scenes, they can require a lot of storage space to store the visibility information, and the pre-processing step can be very time consuming. These algorithms have been successfully applied to indoor scenes because these environments are static and the potentially visible sets are usually small [2,23]. Cell-based precomputed visibility algorithms have been applied to outdoor scenes [8,24]. Schaufler et al. [21] present a general algorithm based on voxelization of occluders, which can also be applied to terrains. Stewart [22] precomputes visibility for hierarchical terrains. A representation of the visibility horizon is created for each vertex of the terrain in a pre-processing step. This information is used at run-time to determine whether a particular vertex is occluded from the current viewpoint.

Another class of occlusion algorithms calculates occlusions on the fly. Green et al. [14] and Zhang et al. [25] use hierarchical data structures to calculate occlusions on the pixel level. Bartz et al. [3] use hierarchical spatial subdivision and OpenGL features to detect occlusions, thus taking advantage of graphics hardware. Hey et al. [15] use a low-resolution grid in combination with an occlusion buffer or z-buffer that is updated in a lazy manner order to reduce the cost of expensive pixel level queries. Coorg and Teller [10] classify large occluding polygons and use the support planes of these polygons to determine occlusion of other objects in the scene. Bitner et al. [4] propose a similar algorithm that uses a BSP data structure to merge occluders. Both of these methods require large polygons to be used as occluders. Since terrains typically consist of many small polygons, neither of the algorithms is suitable for terrain rendering.

Occlusion horizons can be used to render 2D-functions and height fields. Downs et al. [13] show how to use an occlusion horizon for urban environments. The proposed algorithm in this report uses an occlusion horizon and thus is similar in many ways to theirs. However, I use an occluder representation that is better suited for large terrains. Moreover, the horizon representation I use is much simpler.

Hierarchical terrain models may be built on top of a uniform rectangular grid in the form of a quad tree or a restricted quad tree, also called a bintree. When you go over the LOD Algorithms, you find out that there are mainly three different approaches [16][17][12]. In [16], Hugues Hoppe presents an algorithm based on *Progressive Meshes*, a relatively new technique for adding triangles to arbitrary meshes, as you need more detail. It is a good paper but a bit difficult. Moreover, it requires high memory requirements than the other two algorithms [12] and [17]. The second paper [2] Lindstrom et al. presents a *Quad Tree* structure that is used to represent a patch of landscape. A Quad Tree recursively tessellates the landscape creating an approximation of the Height Field. Quad Trees are very simple and efficient. Finally, in [12] Duchaineau et al. presents an algorithm (*Real-time Optimally Adapting Meshes*) based on a *Binary Triangle Tree* structure. Here each patch is a simple isosceles right triangle. Splitting the triangle from its apex to the middle of its hypotenuse

produces two new isosceles right triangles. The splitting is recursive and can be repeated on the children until the desired level of detail is reached. Hierarchical terrains may also be built upon irregular triangle meshes. Since more freedom can be used in selecting the location of vertices, it is often possible to represent a terrain with fewer triangles with these methods than with those based on regular grids. Examples of these types of terrains include Delaunay triangulations [11,19] and view dependent progressive meshes [16]. Terrains can also be rendered efficiently using voxels [9].

Finally, I present the main references about programmable graphics hardware. Hewlett-Packard proposed an overview of VISUALIZE fx Graphics Accelerator Hardware [26] and an extension to OpenGL providing an occlusion query based on a flag [27].

3 Terrain Rendering Algorithm

I used a variation of quadtrees described by Cline and Egbert [6]. The advantage of their method is that it eliminates the noticeable 'pop' that occurs when the level of detail changes suddenly. The technique is called as 'q-morphing' (quadtree morphing) that perform a smooth morph between adjacent levels of detail.

The algorithm contains a quadtree with small grids called *gridlets* at the nodes. The resolution of each gridlet may change, however their spatial size is fixed. The quadtree is traversed top-down to decide which gridlet to draw. First, the visibility is checked for each gridlet against the view frustum. A gridlet lying totally outside the viewing frustum is discarded along with all its children. If projected screen radius of a gridlet falls below a user specified treshold, it is added to the drawing list. Otherwise, the traversal continues with its children. Once a gridlet is placed in the drawing list, a screen-space error metric and the local surface roughness are used to compute an LOD parameter describing how finely the gridlet should be decimated in order to satisfy user specified error bounds. The LOD parameters for both the current gridlet and its parent are combined in such a way that the parameter varies smoothly as position changes. The integer part of the parameter is used to pick a subdivision level. The fractional part is used to morph from one subdivision level to the next. Please refer [5] for the details of the calculation of LOD parameters.

4 Horizon Occlusion Culling

Gridlets are first tested against the view frustum. The gridlets that are inside the viewing frustum are then tested against the current visibility horizon. The horizon is constructed by the highest points in the screen space projection of the terrain, rendered thus far. The projection of the gridlet is checked against the horizon and if it falls below that horizon, it is guaranteed to be hidden and culled along with all its children. If any part of the gridlet lies above the horizon, the horizon is updated with the gridlet's occlusion profile and the gridlet is added to the drawing list.

Occlusion horizon algorithms, in general, consist of three main parts:

- Front to back ordering of the scene elements,
- Simplified representations of the scene elements for the calculations of the occlusions efficiently,
- A representation for the visibility horizon.

In the following sections, each of these components is described briefly.

4.1 Front-to-Back Ordering

The horizon occlusion-culling algorithm works only if the objects are rendered in front-to-back order. In my representation I have a hierarchical structure for the grids, which is a quadtree. Therefore, the correct traversal order is enough for front-to-back ordering. What is the correct traversal order then? A correct traversal order of the gridlet children only need to ensure that each child is rendered before its siblings that it may occlude. It depends on the slope of the line through the gridlet center from the viewing position. The traversal order is stored in a simple array for each quadrant.

4.2 Occluders and Occludees

The efficiency of the visibility test is very important for this application, because it is performed many times per frame. Hence, simple approximations to the terrain elements are more suitable for tests instead of the original terrain elements. There are two simplifications for a terrain element, *occludees* and *occluders.* The occludee is a conservative over-estimation of the extents of an object used for the visibility test. Over-estimation of the object ensures that when occludee is below horizon line, object itself is guaranteed to be below the horizon. However, for some cases, the occludee can be marked as visible while object is below the horizon. That time object is rendered unnecessarily, but this does not change the final image.
The occluder is the conservative under-estimation of the occlusion profile of the object, which is used to update the horizon.

For my implementation I used single line-segments as in [1] to approximate the edges of the gridlet as the occluder representation. Each line segment requires only two height values and only the top two or three of these segments have to be tested against per gridlet, depending on the orientation of the gridlet with respect to the viewer. The line that is generated with the edge points of the gridlet is shifted down so that, all edge points lie above it to ensure that the occluder edge is conservative. The edge points at the finest decimation level for the gridlet is used for the gridlet to guarantee that the occluder edge valid for all levels of detail. If finest decimation level is not chosen, then the conservative property of the occluder may not be satisfied, since morphing to a higher resolution may cause the edge to dip below the occluder line.

Sample occlusion profiles are given in Figure 2. As it is seen, there are some limitations for the occluder edges. They cannot catch for the occlusions caused by raised portions in the center of the gridlet and for the gridlet with curved edges.

Axis-aligned bounding boxes are good and practical approximations for the occludee representation. Only the top-most edges of the bounding boxes need to be compared to the horizon.

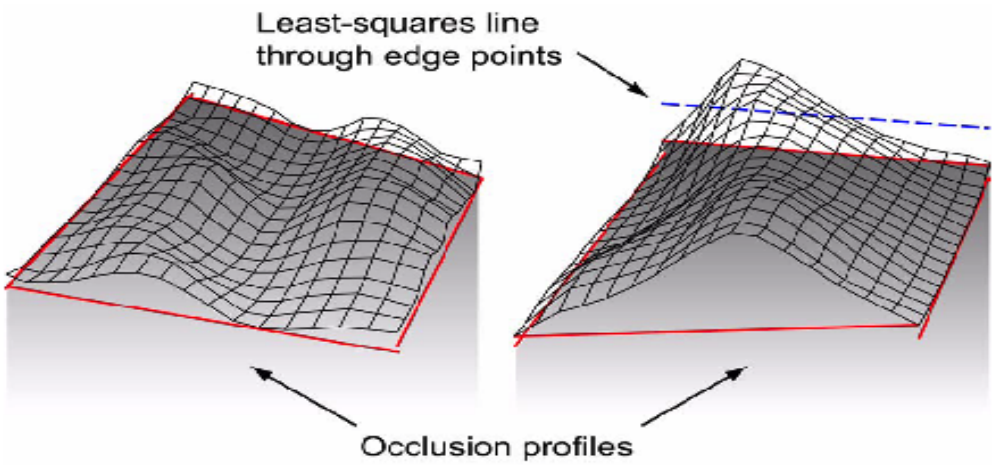


Figure 2: Gridlets with occluder edges and their occlusion profiles. The gridlet on the right is an example of a poor approximation of the actual gridlet edge by the occluder.

4.3 Horizon Representation

The occlusion horizon is an array of size of the resolution of the image. Each element of the array represents a vertical column of the image. Height of each column is stored in this array. A horizon update is performed by scan-converting the projected occluder edges into the array, recording the height of the edge at each column whenever it is greater than the column's current height. If the occludee edge is higher than the horizon at any point, than the scene object is marked as visible. An example for the visualization of the horizon occlusion culling is shown in Figure 3.

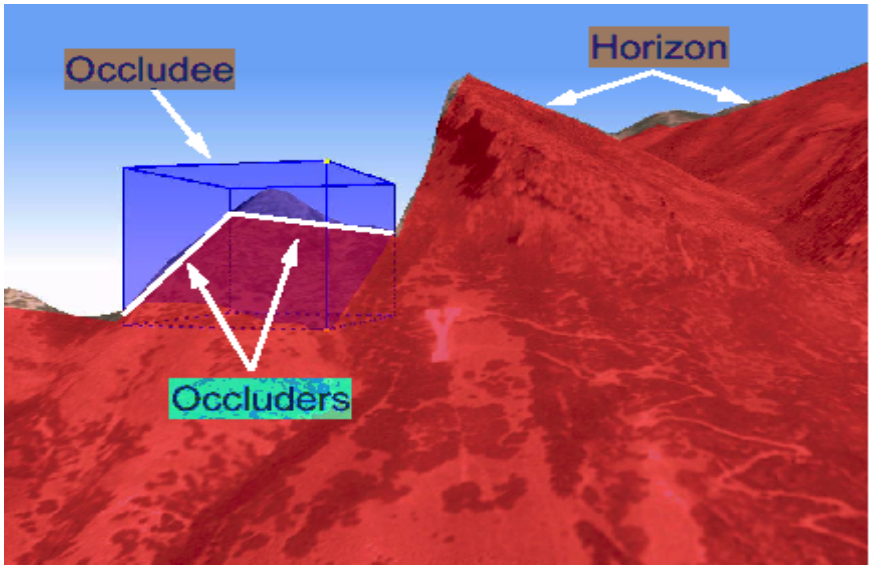


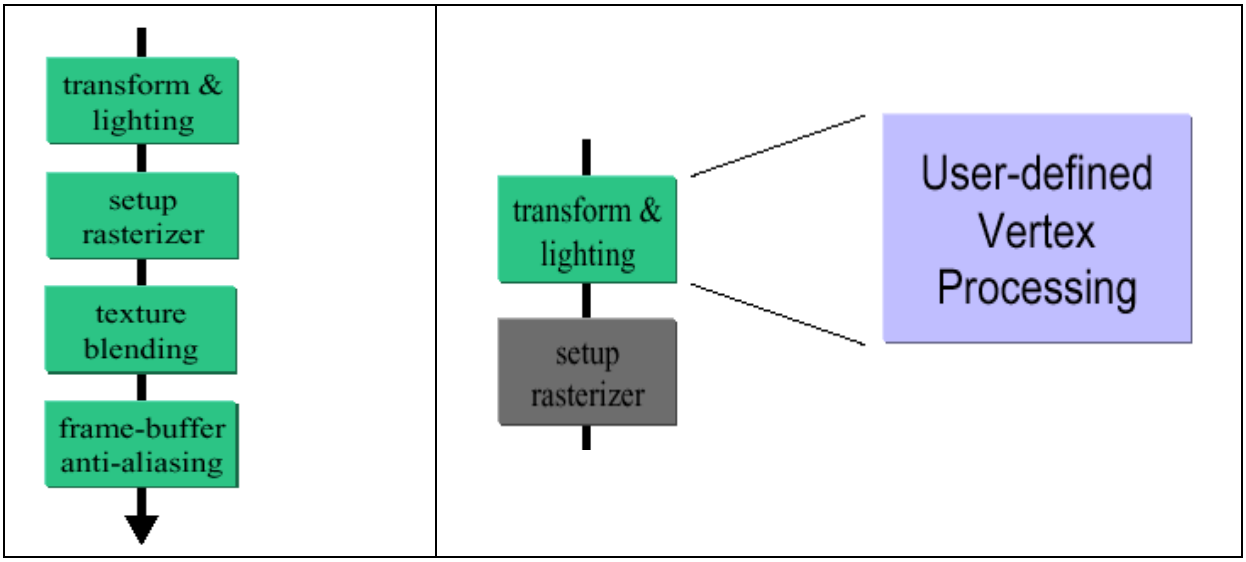
Figure 3: The red mask represents the current approximation of the visibility horizon updated with current gridlet's occlusion profile.

5 Hardware Programming

After the completion of the horizon occlusion culling, each triangle that is visible to the user is determined. At this stage we have the information about the vertices to be sent to the rendering pipeline. These information with other required parameters (i.e: texture coordinates, normals,..) are sent to the programmable graphics hardware and the resultant image is generated using Graphics Processing Unit (GPU). In this section I make an introduction to vertex programming. More detailed information can be obtained from [26],[27] and NVIDIA's home page (http://www.nvidia.com).

5.1 Vertex Programming

Traditional graphics pipeline consists of four sequential parts:



Each unit has specific functions and modes of operations. Vertex programming enables the graphics programmer to total control of vertex processing in the Transform and Lighting stage of the pipeline. It is a kind of customization of the vertex processing. Programmer can:

- Complete control of transform and lighting on hardware,
- Complex vertex operations accelerated in on hardware,
- Custom vertex lighting,
- Custom skinning and blending,
- Custom texture coordinate generation,
- Custom texture matrix operations,
- Custom vertex computations of your choice,

By this way, vertex computations are offloaded from the CPU and more time is left for the occlusion calculations of the next frame in CPU.

Vertex program uses an assembly language interface in T&L unit. All vertex maths is performed with the GPU instruction set. It reads an untransformed and unlit vertex and creates a transformed vertex with optionally creating lights for the vertex, texture coordinates, fog coordinates, etc.

Vertex program does not create or delete vertices (1 vertex in, 1 vertex out). Moreover, no topological information is provided (no edge, face or neighboring information). It is dynamically loadable and exposed through NV_ vertex_ program extension (Figure 4).

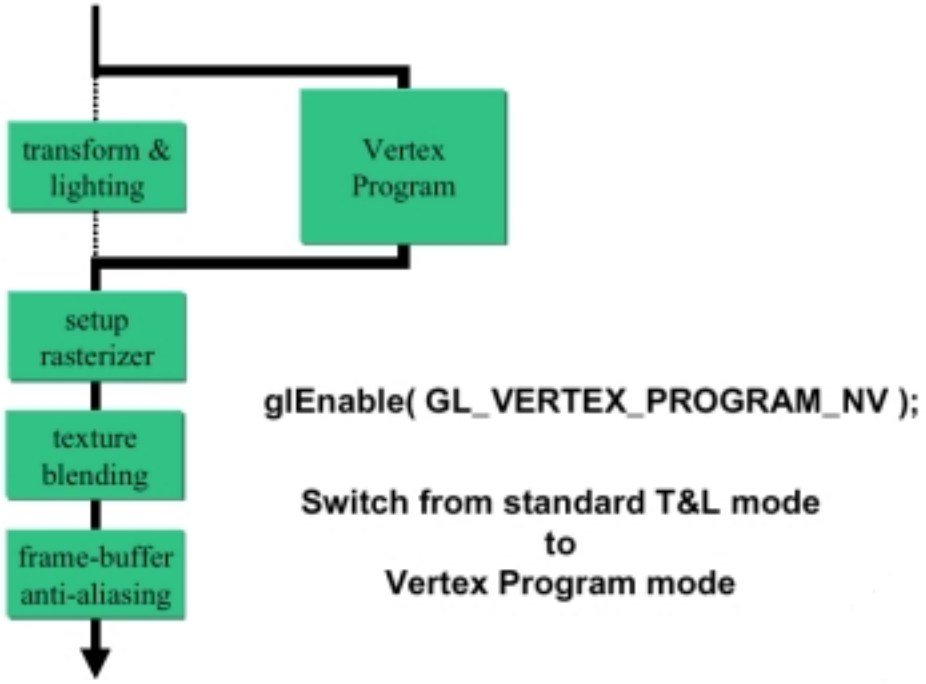


Figure 4: This figure shows the place of the vertex program and how it is enabled.

Vertex programs are arrays of Glubytes (“strings”). They are created and managed similar to texture objects and invoked when *glVertex* is used.

This section is a brief introduction to vertex programming, for the programming details refer to [26].

6 Current State and Future Work

Up until now, preliminary knowledge about hierarchical terrains and occlusion horizon culling algorithm is obtained and implementation of the horizon occlusion-culling algorithm for hierarchical (quadtree based) terrains is completed. In the current state, I am working on the hardware programming techniques and making research about the improvements that can be achieved in rendering times using programmable graphics hardware.

After the completion of the hardware programming part, I will compare the rendering times of horizon occlusion culling algorithm implemented with hardware programming and the one implemented without hardware programming techniques. In addition to this, the algorithm may be adopted to render the urban environments with some small adaptations to the data structures to include buildings. I will improve Downs [13] work for this adaptation using hardware.

7 Conclusion

Finding efficient rendering techniques for terrains is an important issue for the simulations using huge datasets to display environment. In the current state of the art with the improvement on the graphics cards, the need for realistic rendering of the scene increased.However, even high memory and speed of new graphic cards cannot afford the demands for rendering huge size of terrain polygons. Therefore, some simplifications over the terrain without disturbing the realistic looking are necessary. Level of detail and occlusion culling algorithms are very suited to find solutions to these kinds of problems.

The main issue on this project is the adaptation of horizon occlusion culling algorithm to hierarchical terrains. By this way, the polygon count sent to the rendering pipeline is reduced by the level of detail control and the determination of occluded regions of the terrain. The navigation over the terrain is enabled in real-time with minimal preprocessing and memory. The effect of rendering using hardware programming techniques will be seen after the completion of the project. I expect the rendering times will be decreased significantly after then.

8 References

[1] L. Brandon,E.Parris, Horizon Occlusion Culling for Real-time Rendering of Hierarchical Terrains, Bringham Young University, 2001.

[2] J. Airey, J. Rohlf, and F. Brooks Jr. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. *Symposium on Interactive 3D Graphics 1990,* pp. 41-50, 199*0.*

[3] D. Bartz, M. Meibner and T. Hüttner. OpenGL-assisted Occlusion Culling for Large Polygonal Models. *Computers & Graphics,* vol. 23, pp. 667-679, 1999.

[4] J. Bittner, V. Havran, and P. Slavik. Hierarchical Visibility Culling with Occlusion Trees. *Proceedings of Computer-Graphics International '9*8, pp. 207-219, June 1998.

[5] D. Cline and P. Egbert. Interactive Display of Very Large Tex-tures. *Proceedings of IEEE Visualization '9*8, pp 343-350, October 1998.

[6] D. Cline and P. Egbert. Terrain Decimation through Quadtree Morphing. *IEEE Transactions on Visualization and Computer Graphic*s, vol. 7 (1), pp. 62-69, 2001.

[7] D. Cohen-Or, Y. Chrysanthou, and C. T. Silva. A Survey of Visibility for Walkthrough Applications. *Proceedings of EUROGRAPHICS '0*0, *Course Notes,* 2000.

[8] D. Cohen-Or, G. Fibich, D. Halperin, and E. Zadicario. Conservative Visibility and Strong Occlusion for Viewspace Partitioning of Densely Occluded Scenes. *Computer Graphics Foru*m, vol. 17(3), pp. 243-253, 1998.

[9] D. Cohen-Or, E. Rich, U. Lerner, V. Shenkar. A Real-Time Photo-Realistic Visual Flythrough. *IEEE Transactions on Visualization and Computer Graphics,* vol. 2(3), pp. 255-265, 1996.

[10] S. Coorg and S. Teller. Real-time Occlusion Culling for Models with Large Occluders. *1997 Symposium on Interactive 3D Graphics*, pp. 83-9*0*, April 1997.

[11] L. De Floriani and E. Puppo. Constrained Delaunay Triangulation for Multiresolution Surface Description. *Proceedings of Ninth IEEE International Conference on Pattern Recognitio*n, pp. 566-569, 1988.

[12] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, M.B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. *Proceedings of the Conference on Visualization '9*7, pp.81-88. Oct. 1997.

[13] L. Downs, T. Möller, and C. Sequin. Occlusion Horizons for Driving Through Urban Scenery. *2001 Symposium on Interactive 3D Graphic*s, pp. 121-124, 256, March 2001.

[14] N. Greene, M. Kass, and G. Miller. Hierarchical Z-Buffer Visibility. *Computer Graphics (Proceedings of SIGGRAPH '93*), pp. 231-238, 1993.

[15] H. Hey, R. Tobler, and W. Purgathofer. Real-Time Occlusion Culling with a Lazy Occlusion Grid. *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*. pp. 217-222, 2001.

[16] H. Hoppe. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. *IEEE Visualization '9*8, pp. 35-42, Oct. 1998.

[17] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner. Real-time, Continuous Level of Detail Rendering of Heightfields. *Computer Graphics (Proceedings of SIGGRAPH '96*), pp. 109-118, 1996.

[18] R. Pajarola. Large Scale Terrain Visualization Using The Restricted Quadtree Triangulation. *Proceedings of IEEE Visu-alization '9*8, pp. 19-26, 1998.

[19] B. Rabinovich and C. Gotsman. Visualization of Large Terrains in Resource-Limited Computing Environments. *IEEE Transactions on Visulaization and Computer Graphic*s, pp. 95-102, 1997.

[20] S. Röttger, W. Heidrich, P. Slussallek, and H.-P. Seidel. Real-Time Generation of Continuous Levels of Detail for Height Fields. *Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualizatio*n, pp. 315–322, Feb. 1998.

[21] G. Schaufler, J. Dorsey, X. Decoret, and F. Sillion. Conservative Volumetric Visibility with Occluder Fusion. *Computer Graphics (Proceedings of SIGGRAPH 2000*), pp. 229-238, July 2000.

[22] J. Stewart. Hierarchical Visibility in Terrains. *Eurographics Workshop on Rendering '97*, pp. 217-228, 1997.

[23] S. Teller and C. Sequin. Visibility Preprocessing for Interactive Walkthroughs. *Computer Graphics (Proceedings of SIG-GRAPH '91*), pp. 61-69, 1991.

[24] B. Zaugg and P. Egbert. Voxel Column Culling: Occlusion Culling for Large Terrain Models. *VisSym 2001-Eurographics/ IEEE Symposium on Visualizatio*n, pp. 85-93, May 2001.

[25] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility Culling using Hierarchical Occlusion Maps. *Computer Graphics (Proceedings of SIGGRAPH '97)*, pp. 77-88, 1997.

[26] N. Scott, D. Olsen, and E. Gannett. An Overview of the VISUALIZE fx Graphics Accelerator Hardware. *The Hewlett-Packard Journa*l, (May):28–34, 1998.

[27] Hewlett-Packard. Occlusion Test, Preliminary. http://www.opengl.org/Developers/Documentation/Version1.2/ HPspecs/occlusion test.txt, 1997.