

REAL TIME RENDERING OF
SMALL EXPENSIVE ENVIRONMENTS

by

COLIN BRANCH

Advisor

DR. HALA ELAARAG

A senior research paper submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science
in the Department of Mathematics and Computer Science
in the College of Arts and Science
at Stetson University
DeLand, Florida

Spring Term
2004

Table of Contents

Abstract.....	5
1 Introduction.....	6
2 Expensive Small Environments.....	6
3 Implemented Features.....	8
3.1 Lighting Models.....	8
3.2 Normal Mapping.....	10
3.3 Shadows.....	12
3.3.1 Texture based Shadows.....	12
3.3.1.1 Light Mapping.....	13
3.3.1.2 Shadow Mapping.....	13
3.3.2 Geometry based Shadows.....	14
4 Optimization Methods.....	15
4.1 Progressive Mesh.....	16
4.2 Culling.....	17
4.2.1 Frustum Culling.....	17
4.2.2 Occlusion Culling.....	18
4.2.3 Light Culling.....	19
4.3 LOD.....	20
4.4 Vertex Arrays and VBO.....	21
5 Implementation.....	22
6 Testing.....	23
6.1 Hardware.....	23

6.2 Testing Procedure	23
6.3 Testing Results.....	24
6.3.1 Feature Comparison.....	24
6.3.1.1 Texturing.....	24
6.3.1.2 Lights	25
6.3.1.3 Shadows	25
6.3.1.4 Vertex Data	25
6.3.1.5 Comparison.....	26
6.3.2 Card Comparison.....	27
7 Conclusion	27
7.1 Further Work.....	28
References.....	29
Appendix A.....	30

Table of Tables

Table 1: Computer Components	23
Table 2 Testing Variables	24
Table 3: Relive Costs of Features	26
Table 4: Scene 1 FPS measurements On NVIDIA HW	AA
Table 5: Scene 1 FPS measurements On ATI HW	AA
Table 6: Comparison of ATI to Nvidia Performance	AA

Abstract

One of the major goals of computer graphics is the rendering of realistic environments in real-time. One approach to this task is to use the newest features of graphics hardware to simulate complex effects which increase realism.

This paper will propose a set of features which allows small environments to have increased realism. These features will be geared to work on consumer level graphics hardware and thus be not solely dependent on the CPU. Additionally the paper will present methods of increasing the speed of these environments so that they run in real time; this will be done by limiting the rendered data to approximate only data in the visible scene.

The implementation of these features allows for analysis of the performance cost and saving of each feature as well as comparison of the capabilities of modern graphics hardware. Both absolute and relative statistics on different hardware are determined.

1 Introduction

The evolution of computer simulated 3d environments could be summarized as massive expansion; the environments continually grow in size. A primary reason behind this growth is the rapid growth of hardware. The first generation of consumer 3d accelerators had modest limits on the number of triangles it could render. In 1996, the Voodoo chipset by 3Dfx could output approximately 1,000,000 triangles per second which translated in scene sizes of at most 16,000 triangles for an environment redrawn 60 times a second (hz). Less than 10 years latter the NVidia Geforce 5900 FX Ultra can output 112 million triangles per second, allowing 1.8 million triangles per scene for real-time display. Since early hardware was geometry limited, algorithms for real time rendering focused on limiting the number of triangles rendered by the graphics card. This focus was marginalized as hardware was designed to process larger and larger geometry loads.

Newer graphics hardware is not only designed for faster rendering speeds. One of the most powerful, complex, and slow feature is the ability to program the components that control the vertex and fragment processors. These programmable components allow for extremely detailed rendering at much higher costs compared to non-programmable or fixed pipelines. The programmability of the graphics card allows for greater flexibility when creating a scene.

2 Expensive Small Environments

An expensive environment is one which takes considerable amount of time to render. This terminology is relative to the time, hardware, and even the context of the environment. An expensive environment may be one that takes hours to render using a ray-tracing program, or

simply many milliseconds for a real-time application.

In either case the environment takes a comparably longer amount of time than is average or expected. A sufficient condition for an expensive environment is large size. An environment composed of a large amount of geometric data will take a large amount of time on any system. Once hardware and algorithms are developed to render the scene in a timely manner, the only requirement to make the scene expensive again is to further increase the size. A large amount of research on these types of environments has been conducted, such as the research at University of North Carolina using their power plant model. The solution to this type of problem is to find ways of limiting the geometry or simply making the size of geometry no longer large in comparison to the capabilities of hardware.

The research into rendering small environments is pertinent to large environments since large scenes are often transformed into small scenes before being rendered. Algorithms which incorporate structures such as an octree and binary space tree representations can generate a small environment from a much larger environment. The resulting small environment is often considered inexpensive in comparison to the larger one.

A different approach to creating an expensive environment is to make each primitive more expensive to render. Thus an expensive small environment is one where the cost of the rendering the geometry is the limit factor rather than the size of the geometry. Thus simply decreasing the geometry will have a lessened increase in performance.

Effective rendering of expensive small environments has different concerns than those of geometry limited environments. In a geometry limited environment, the typical limit is imposed by both the memory bus over which geometry is sent to the video card and the speed that primitives can be generated on the video card.

Thus an approach to increase rendering speed is to cull out large amounts of geometry that doesn't contribute to the scene. In smaller environments it is harder to cull out large amounts of geometry since frequently it is not effective to cull out relatively small amounts of geometry. For example, it is feasible to cull out the entire object it is less feasible to cull out individual parts of a mesh especially if they only contain hundreds of primitives. This is because the cost of checking could be more than rendering the geometry.

Creation of an expensive small environment requires several features that raise the cost of rendering individual primitives. This paper will present techniques that when used together will create an expensive small environment. The goal of rendering this environment is to do it in real-time or redrawing the scene between 30 and 60 times a second. The techniques proposed to increase the cost of the environment include advanced lighting models, normal mapping, and dynamical shadows. Implementation of these features creates an expensive small environment which can then be tested for such information as the overall impact of each feature or the performance of a feature on particular hardware.

3 Implemented Features

This section covers the methods that raise the cost of rendering the environment. It will be of interest to see the performance hit accrued by each feature. Various parts of the video card are used for each effect and overloading one component will create a bottleneck and slow down performance in all others.

3.1 Lighting Models

With the introduction of programmable hardware came the liberation from a fixed

lighting algorithm. Since the beginning of computer graphics, many different equations for calculating the lighting of objects have been developed [2]. Some equations are better suited for hardware implementation due to the fact that their mathematics can be implemented in circuitry thus allowing for high performance. A very common lighting equation is called Phong lighting. Phong lighting, proposed in the 1970s by Bui-Tuong Phong, calculates the lighting of an object based on the light position, direction and the surface's position and normal. Phong lighting is the lighting model used by the two major graphics APIs: OpenGL and DirectX. Both libraries implement Phong lighting as a local lighting model, which handles each primitive as being independent of all other geometry. Thus there is no shadowing, reflection, refraction, or any other effects dependent on other geometry. These effects can be simulated using various techniques however they are mostly independent of the lighting model.

The programmable components on the graphics card are called vertex and fragment/pixel shader units. Programs written for these units are called shaders. These components replace the existing fixed equations so this allows shaders to be written for nearly any lighting model. Thus models such as Lambert, Blinn, Cook-Torrance, or Minnaert can be implemented and with good performance. Another feature which is obtained through the programmable hardware is the ability to calculate the lighting on a per-pixel basis [1]. Traditional hardware methods calculate lighting calculations on a per vertex basis. This creates artifacts since each lighting element is limited by the resolution of the geometry. With per-pixel lighting, a lower quality mesh can be rendered with far more realistic lighting especially specular highlights, which typically are highly effected by the size of geometry. The drawback to per-pixel lighting is that for each pixel drawn the lighting equations must be run. Thus the lighting model may be run for millions of pixels compared to thousands of vertices. Pixels in the scene can be drawn over with later

geometry. Thus frequently time is wasted on pixels that are not even visible in the final scene.

The implementation includes several lighting models that use both per-vertex and per-pixel calculation. All lighting was done using Vertex and Pixel. The shaders were programmed in Cg high level language developed by Nvidia. Currently they are all based on Phong lighting calculations. Implemented are standard material based rendering, texture based rendering and bumpmapped rendering. The implementation is flexible and allows for easy alteration and addition for more lighting models.

3.2 Normal Mapping

The first use of texture mapping was to wrap geometry with images which creates far more detailed objects [8]. This was proposed by Ed Catmull in 1974. The textures were used to store diffuse colors of planar surfaces (though the object did not need to be planar). The color of each pixel from the primitive was then retrieved from the texture and using perspective interpolation via per-vertex texture coordinates. It was proposed by Jim Blinn that textures could be used to store other information about the surface of an object such as its normal vectors. Thus normal or bump mapping was conceived. There is no real difference between normal mapping and bump mapping, only in perhaps the implied source of the normal maps. Bump mapping traditionally are based in combination with diffuse texture maps. First diffuse maps are converted to height maps through measure a relativistic height of each part of the diffuse map. Then these heights are converted into normal via tangent and binormal vector calculation:

Equation 1: Tangent, Binormal, Normal Calculations

$$\begin{aligned}\tau &= (1, 0, \text{map}[x+1, y] - \text{map}[x-1, y]) \\ \beta &= (0, 1, \text{map}[x, y+1] - \text{map}[x, y-1]) \\ N &= \tau \times \beta\end{aligned}$$

Normal mapping tends to refer to normal maps calculated from sampling another model. Thus there are two versions of the model, a high-polygon version and a simplified low polygon version. The high resolution version is sampled when generating a normal map for each polygon in the simplified version, which creates the appearance of higher geometry when rendering the low-res version. This method is quite effective in creating seemingly complex surfaces with very limited actual geometry.

When rendering using a normal map, a requirement is that the lighting calculation be per-pixel since the normal is used in the calculation and it is sampled from the normal map [4]. The normal of the texture is used in combination with the vertex normal vectors.

There is an irregularity associated with normal mapping due to the fact that the normal vectors are not stored in object space but rather in an arbitrary space associated with how the texture coordinates, of the original model, are assigned. The solution to this irregularity is that the light coordinates need to be translated into a different space that allows the normal vectors to be used correctly. The most common space to use is tangent space, which requires that the tangent and binormal vector of each primitive also be calculated and used to translate the light into the same space as the texture.

Normal mapping is not dependent on any particular lighting model since it simply is a method of providing normal data on a per-pixel basis. Once the normal is acquired it is possible to use it to calculate the diffuse and or specular component using whatever light model is

desired.

The implementation includes normal mapping. Standard diffuse bump mapping was implemented which only calculates the diffuse component based on the normal map. Further bump map styles were programmed but were not used in testing. This includes specular bump mapping and offset bump mapping[9].

3.3 Shadows

Shadows are a very important portion of our visual experience. Many aspects of vision are influenced by shadow such as depth perception, edge detection, and object differentiation. Traditionally real-time rendered scenes have little to no shadows due to inherent complexities of shadows and the local lighting model does not allow shadows to be calculated with the lighting.

Shadows are caused by the absence of light from a specific area. The sharpness and darkness of a shadow is dependent on the distance the surface which the shadow on and the object that is casting the shadow. In a local lighting model each primitive is assumed to have light shining on it without being occluded by another object, thus excluding the concept of shadows. This makes it very difficult to implement actual shadows since there is no way to vary the light to a specific primitive.

Various techniques for simulating shadows have been developed. These techniques can be split into two groups: texture based shadows and geometry based shadows. Each technique allows for a different balance between adaptability, quality and speed.

3.3.1 Texture based Shadows

Texture based shadows are split into two main types: light mapping and shadow

mapping.

3.3.1.1 Light Mapping

Light mapping is the technique where the lighting for each primitive in a model is computed and saved in textures called light map. When rendering the model these light maps are added to the existing texture maps and draws giving the appearance of lighting and shadows. This technique is very inexpensive at render time since the complicated hence expensive work is done in preprocessing when creating the map. Also modern hardware has support to combine multiple texture maps together at high speed. A light map must be stored for each primitive; the amount of space taken up can be considerable for large models. Light maps are almost always static due to the high cost of generation and do not allow for dynamic lighting or shadowing. Maps can be combined with other shadowing techniques for added realism.

3.3.1.2 Shadow Mapping

Shadow mapping is a texture mapping technique which allows for dynamic shadows. For each light a shadow a texture is generated from the point of view of the light where only the distances are stored in a shadow map. Then when the objects that have shadow on them are drawn, possibly including the original object, the distance of each pixel is tested against the shadow map. If the distance of the pixel is farther than in the shadow map the object is in shadow and thus is drawn darker. While shadow mapping allows for dynamic lighting on any surface it does come at a cost, a new shadow map must be generated if either the light or the object moves.

Both texture techniques are affected by aliasing due to the relative low resolution of the

shadow maps and as the angle of incidence of the shadow increases the aliasing also increases. Thus individual texels of the texture can represent a large number of pixels on the screen. One solution that diminishes aliasing is to blend the texture which takes additional cost however it also adds one nice feature, soft shadows. This blending doesn't come for free, however it can be implemented completely in hardware which is faster than having to return the textures to system memory and use the CPU to blend them.

3.3.2 Geometry based Shadows

Geometry based shadows involve projecting geometry along vectors defined by the light from each vertex in the shadow caster. This technique is called Shadow Volumes since a mesh is generated that represents the volume of the shadow. This can be done on the graphics card via a vertex shader thus freeing the CPU from doing the calculations [6]. The stencil buffer is used to determine which pixels are shadowed and which are lit [7]. First the entire scene is rendered using only ambient lighting. Next the back faces of the shadow geometry are rendered with depth testing on but depth writing off and only rendering to the stencil buffer with at increment. Next the front faces are rendered with the stencil buffer decrementing. On new hardware it is possible to combine these two steps together by defining separate operations on the stencil buffer based the face orientation. Now the scene is rendered normally again with diffuse and specular lighting but only where the stencil buffer is positive. This process can be repeated for several lights requiring only clearing the stencil buffer. Special care needs to be taken to ensure only visible shadows are generated and the equation changes slightly if the viewer is inside a shadow volume.

The advantage of shadow volumes is that there is no aliasing since the shadows are

calculated on a per pixel basis. Generating the shadow volume is dependent on the complexity of the shadow caster however the number of shadow receivers is irrelevant only the resolution of the screen and thus the number of pixels drawn is an issue. The performance of shadow volumes is highly dependent on the fill rate of the graphics card since frequently shadow volumes require a large amount of stencil writing. Shadow volumes provide very sharp shadows however there is no easy way to add soft shadows. One way is to project several volumes at slight different sizes and blend them together to form a soft edge. This method is extremely expensive and not practical for real time rendering yet.

Shadow volumes are implemented using vertex shaders. This method doesn't require calculating silhouette edges on the CPU however it requires new geometry to be generated. This can be done at load time and thus shadow generation isn't required each time a shadow is rendered. Also if indexed primitives are used no new vertex data is generated only a longer list of indices. Typically this extra shadow geometry is three times the size of the original model. Rendering shadows could be considerably more expensive than rendering the model. Only a portion of the shadow geometry is needed to form the volume. If the triangle is extruded equally among all its vertices it will have zero area. These zero area triangles can be culled out by the hardware thus decreasing the size of the geometry that is rasterized.

4 Optimization Methods

The features described above should provide sufficient cost to rendering to make the environment non-real-time if simply rendered without any optimization. Several methods of optimization are considered.

4.1 Progressive Mesh

Progressive Meshes are meshes that do not store the actual geometry of a model but rather a series of vertex splits that can be used to reconstruct the mesh. This allows a mesh be constructed with a specific number of vertices. The rationality behind progressive meshes is that they allow control over the size and complexity of a model. This technique was initially developed by Hugues Hoppe in 1996[5]. Various implementations of Progressive Meshes have been developed, but they all share the basic concept. Starting with a traditional Mesh, optimal edge collapses are calculated. Edge collapses are the opposite of vertex splits. The optimal edge collapse is determined by a cost or energy function. It calculates a value that measures the deviation of the new mesh formed by the collapse from the original. The edge collapse that causes the least amount of deviation is stored and the technique is continued until a lower limit of vertices is reached.

Progressive meshes are calculated on the CPU and thus the GPU doesn't affect the performance of this method. The result of progressive meshing is that it generates fewer polygons and once a specific mesh has been generated it can be reused for several frames. Thus overall the performance should be higher especially in cases where the performance of the environment is related to the amount of geometry.

Progressive meshes were not implemented since they would not have worked well with the shadow volume implementation. Since progressive meshes change the edges within the model it would likewise require that the shadow geometry which is generated per edge to change. This would either have to be done completely from scratch or a complex means of adding and removing indices based on the edge collapse. Combining progressive meshes with a more traditional silhouette detection might be a better combination.

4.2 Culling

Culling simply means to remove from a flock, and thus describes any technique which attempts to remove elements from being rendered. The goal of culling is to minimize the Possible Visible Set which initially is the entire environment. It is not hard to calculate the visibility of every individual item, however the cost of culling is far cheaper than the cost of rendering. Thus techniques for eliminating geometry organize sets of primitives using a simpler bounding geometry. For example calculating the bounding cube or sphere for each object and then using these approximations to see if the object is visible or not is much more effective and cost efficient.

4.2.1 Frustum Culling

Frustum culling involves testing the bounding volume of each object with the frustum of the view. A frustum is the pyramidal shape that is created from the connection of the near plane with the far plane.

The shape of the frustum is relative to the field of view and the view ratio. For orthogonal views the frustum is simply a rectangle. This type of testing is relatively inexpensive and culls out objects that are not present in the possible view. Thus to get optimal results, the bounding volumes used should be tight and accurately represent the volume of the object.

Simple frustum culling of all meshes was implemented. This cuts down on the geometry visible in the scene. Lights similarly were tested against the scene to see if they were visible. Additionally shadow culling was implemented by testing the direct on light verses the near plane. By extending this calculation to the frustum it would be possible to eliminate shadows

which would not possibly be visible.

4.2.2 Occlusion Culling

Not all objects in the frustum are visible since objects in the foreground occlude or cover objects in the background. Thus a different form of culling can test to see if objects are occluded by objects in front of them. There are geometric methods to determining if an object is occluded similar to frustum mapping by simply choosing objects that make good occluders and creating a volume from the object that extends away from the viewer. This is not technically simple since determining good occluders can be difficult as well as requiring additional tests for each occluder. A different method called hardware occlusion culling allows you to test objects against the current z-buffer and to retrieve how many pixels would be rendered if the object would be drawn. Thus an effective method for using this form of culling is to draw the scene from front to back and attempt to cull out farther objects. One benefit of hardware occlusion culling is it is asynchronous and thus while the GPU is testing the pixels the CPU is free to do other work.

Occlusion culling was not implemented for several reasons. The implementation thus far has been largely on the GPU and thus adding an additional set of operations may not increase performance. The benefit of occlusion culling is that it allows the CPU to do other work, while the GPU determines if the object is visible. An implementation of CPU based shadow silhouette calculation and progressive meshes combined with occlusion culling may result in a better mix of techniques to achieve higher performance. Finally the scenes used to test performance were designed so that each object is largely visible and is not entirely blocked by another object. Since the scene is small, each object is relatively close to the viewer. Thus each object

contributes to the ultimate view. A large environment where a far object contributes less to a scene would benefit more from occlusion culling.

4.2.3 Light Culling

Shadow volumes are expensive to calculate. Combine this with the cost of actually drawing the stencil shadows makes minimizing the number of shadows drawn necessary. Thus for each light it is essential to limit the number of objects that could possibly cast shadows. Based on the type of light it is possible to calculate bounding spheres for the possibility of shadows. Point lights emit light in all directions so there is no specific shape that limits the light volume however attenuation is frequently used and thus a sphere can be used to represent the bounds. Frequently a point light is really a hemisphere light which is a half sphere. Spot lights use a cone to bound their volume. Directional light has not bounding volume however frequently there is only one directional light, the sun which can be handled separately. Once each bounding volume is created then objects that are in the bounding volume could cast a shadow. The visible light set is simply the intersection of the lights volumes with the frustum. The intersection of the set of frustum objects and shadow objects is the set of all objects that will have a shadow if rendered. Additionally all shadows of objects not in the frustum must be tested to see if they are in the frustum themselves. The union of these two sets is the set of all possible shadows which should be significantly smaller than the total number of shadows.

Light culling was implemented for point lights. Each light has a bounding sphere which it tests against all the objects in the scene to see which objects it illuminates. This bounding is used for both lighting and shadowing purposes. Thus each light has its own list of objects to draw and cast shadows from.

4.3 LOD

One easy way to increase performance of a rendered scene is to use different versions of objects based on the amount of the image the object takes up. For example small objects that take up limited number of pixels do not need to be rendered using complex and expensive methods. Thus it would be prudent to substitute a lower quality version.

The level of detail or LOD of an object can simply be calculated by its distance from the viewer and its size. Once the LOD is calculated the appropriate object is drawn. There are various ways to eliminate discernable changes in LOD which are common when the change in quality is significant. In one method, the steps in quality are small enough that the change is never noticeable. Progressive meshes are based upon this principle so that the change is gradual enough that the viewer doesn't notice it. Other methods include blending multiple LODs together which requires the object to be drawn twice and decreasing the transparency of the old object until it is invisible.

There are several other areas of the environment that can be changed to create levels of detail. One is to implement simpler lighting models that can be selected for lower levels of detail. For example, small objects frequently do not exhibit specular highlights since they are relatively small and it isn't necessary to compute them. Another possibility would be to use only vertex lighting for objects. This may or may not result in a favorable increase in performance since, at a point the number of pixels will be less than the number of vertices and thus it would be faster to simply render using only per-pixel lighting. Another option would be to use equations that are faster but less accurate. Smaller objects would not need to cast shadows however the fact that an object is small in the view doesn't mean that the shadow will be small. Shadows

themselves are very large as they reach to infinity, and thus creating a LOD on a shadow is considerable harder than for other objects.

No LOD was implemented for several reasons. First it would be difficult to measure the performance of a technique if it was unclear how often it was used when rendering a particular scene. Second the concept of a small environment limits the locality of all the objects to be close to the viewer or at least close to each other. Thus there LOD should be roughly the same and possibly very high. From this it could be seen that varying LOD should not contribute significant improvement to the scene.

4.4 Vertex Arrays and VBO

Sending geometry data to the graphics card is of real importance. Some APIs allow for immediate mode where it the user sends a single primitive at a time to the video card. This is by far the slowest means. A faster method for sending primitive data is using vertex arrays or buffers. Basically placing all the geometry in a block of memory and sending that block at once. This provides for additional performance as the driver can do optimizations and it makes better use of the limited bandwidth between the computer and the video card. Modern video cards frequently have large amounts of fast memory stored on the card itself for textures and buffers since they require fast access. It is thus possible to store geometry and other data directly on the video card. This is achieved in OpenGL using a Vertex Buffer Object or VBO. By storing the data in a VBO it is can be static or dynamic and streamed from the computer to the video card. A static VBO works well with other GPU techniques since it cuts out the host system. It works well with shadows generated via a vertex shader but less well when using the CPU to create the shadow silhouette.

All meshes are stored using VBO for maximum performance. This limits the amount of data sent to the video card each frame as well as taking advantage of very fast memory.

5 Implementation

The implementation of these features was done in C++. All graphics were done using the OpenGL API. The extgl extension loading library was used to generate OpenGL extensions. Simple DirectMedia Layer (SDL) provides windowing, input, and image loading functionality. The implementation allows for cross platform support though all development was done in Windows using Visual Studio. Shaders were written in Cg from Nvidia and set to compile on the optimum settings for the video card. All scene models were created using Wings3D free modeler. All texture maps and normal maps were downloaded from the web or created in Adobe Photoshop. Total implementation time was approximately four months.

The scene is created from an external data file which allows for reuse of models, materials, and lighting models within the scene. The basic structure of the rendering framework is a node-based scene graph. Each node knows how to render and update itself and its children. For example a light node renders the shadows and its lamination of all its child meshes. The rendering is done using multiple passes to create the final image of the scene. The first pass renders the depth buffer and ambient light. Each sequential pass is done per light, and is split into two phases: rendering the shadow volume to the stencil buffer and then rendering the lighting effects to the scene additively which results in blending the lights together to make the final image. The material associated with each object determines how the lighting phase is drawn. Each material stores colors and texture maps. Each object also has a render style to determine what lighting model to use.

6 Testing

6.1 Hardware

The testing will be performed on nearly two identical machines. The only difference will be the video card which is of the same class only from two different chipset developers.

The specifications of the systems can be found in table 1. The difference in graphics card should also allow of an in-depth analysis of the differences between the leading graphics card capabilities, strengths and weaknesses.

Table 1: Computer Components

	Manufacturer	Model
MB	MSI	N-force 2
CPU	AMD	Athlon XP 2700+
RAM	Corsair	PC-3200 DDR CAS 2
GPU	Nvidia	GeforceFX 5900
	ATI	Radeon 9800
HD	Matrox	ATA-133 40GB

Due to technical problems with one of the machines only one computer was used. The video cards were alternated for comparison. The results thus benefit for identical software configurations.

6.2 Testing Procedure

The testing procedure will involve measuring the frames per second (fps) of scenes averaged over a period of time. For each scene, various features will be turned off and the resulting fps will be measured. Features states are in Table 2. Also variable optimization such as culling will be turned off to fully test the cost of the features.

Table 2 Testing Variables

Method	States
Texturing	Bump/Texture/None
Lights	Number
Shadows	On/Off
VertexData	Array/VOB

Each feature measures specific aspects of the video card. By measuring the differences between scores on the ATI and NVIDIA cards, a detail analysis of the strengths, weaknesses and overall capabilities of the cards can be attained. All testing will be done in Windows 2000.

6.3 Testing Results

6.3.1 Feature Comparison

Nearly all features were found to be scale linearly. Thus each has a linear coefficient. This coefficient can be calculated on a per scene basis but has relatively little use in generality. Typically the coefficient is due to the particular geometry, light placement, screen resolution, computer hardware, and the operating system. These coefficients vary from execution to execution of the program and thus are only useful in relative observations.

6.3.1.1 Texturing

Texturing is primarily a function of the pixel capabilities of the video card. Thus changing the texturing will be based on the texture fetch capabilities. Since bump mapping uses a more considerable pixel shader it also tests the capabilities of the pixel shaders.

The difference in time per frame of texturing varies at a constant rate. This is due to the multi-pass rendering of the scene. Since the z-buffer is rendered before the lighting phase, only each lit pixel is rendered at most once. Thus there is a constant number of pixels rendered per

frame in the lit stage since only the closest pixels are calculated.

6.3.1.2 Lights

The number of lights has a large effect on the performance of the rendering. Each light will render each object that is visible in its scene. The tighter the light radius the smaller the number of objects that needs to be rendered for each light and vice versa. When combined with texturing this can result in a large slowdown as the number of times each pixel is rendered is increased.

6.3.1.3 Shadows

Shadow volume performance is based on fill rate to the stencil buffer and the vertex shaders used to extend the volumes. It is solely dependent on the size of the geometry and independent texturing. Each light will render the shadow geometry of each object within its radius. Thus worst case is that each light will render the shadow geometry of each object. Thus the timing for the shadow is highly dependent on the number of lights in the scene and the number of objects within the lights radius. The performance of shadows is linear in terms of the number of lights rendered with slight variation due to the number of objects within the light spheres. The constant in the relation is the size of the shadow volume geometry. The orientations of the light and the shadow have no effect on the performance of the vertex shader and only minimal impact on the fill rate of the shadow.

6.3.1.4 Vertex Data

The use of VBO has a significant effect on the performance of the rendering. Since each model needs to be rendered several times. Large performance gains were seen when the scene

was rendered once. Lesser gains were seen when more lights and shadows were enabled. Optimization or caching done by the video driver would account for this result.

6.3.1.5 Comparison

By far the most expensive feature was bump mapping. However this feature has only a single relative cost to the rendering. Due to multi-pass rendering this means that the lighting level pixels are only drawn once per pixel location in the buffer. Increasing the number of lights or the geometry doesn't increase cost greatly. The same rule applies to standard texturing. Shadows while relatively cheap individually increase in cost per light. This is due to the fact that for each light the larger shadow geometry for each object is rendered. Thus using five lights the cost of shadows is roughly 65% slower than not using shadows. This feature is more expensive than bump mapping in terms of average frames per second. The worst case performance-wise for shadowing assumes the entire scene is the light radius. By not rendering some shadows performance can be improved.

Table 3: Relative Costs of Features

Feature	Relative Cost	
Shadows	13%	* number of lights
Lights	10%	per light
Bump map	55%	
Texturing	40%	
VBO	140%	* lights

The number of lights has a direct cost on rendering with each additional light slowing the scene by 10%. Though culling can improve this value by ensuring the minimum amount of lights is rendered at any one time. Vertex buffer objects increase the performance of the scene by around 140% however each individual light decreases this by 7%. So that with 5 lights there is only a small gain through use of VBO.

6.3.2 Card Comparison

The two video cards chosen are competing products of the same class. Both are considered the top tier products by their respective companies for the Q4 2003 to Q1 2004. The Nvidia GeForceFx 5900 processor is timed at 450 MHz with 425 MHz memory. The ATI Radeon 9800 processor is timed at 380 MHz and 340 MHz memory. Thus on paper the Nvidia card appears superior to ATI however in testing this is not the case. Features requiring the programmable pipeline were all slightly faster in the ATI card. The tests where the Nvidia hardware performed better included shadows and multiple lights. From this it can be concluded that the stencil operations used in the Nvidia card are faster than ATI's card. Since ATI performed better than the Nvidia card despite slower memory it can be assumed that the memory on the card is not the bottleneck. The bottleneck must be somewhere within the pipeline though probably not the pixel fill rate as the Nvidia has a faster core speed and performs well on the stencil operations. Thus the remaining components that are most likely the issue are the programmable vertex and pixel processing units. ATI's bottleneck most likely is in fill rate since it didn't do stencil operations as fast as Nvidia's card. Thus a faster speed memory and higher clock rate would improve performance.

7 Conclusion

The analysis of these features shows their relative costs. The application of this analysis allows measurement of the benefit of a particular feature at a cost which can be both absolute and relative to other features. At the current stage small and modest sized environments can be rendered in real time using these techniques. However it should be noted that the hardware used is considered high end and is not in the majority of desktop computers. Thus further

optimization would be necessary for real time rendering on such systems. However by good scaling of a large environment to a small environment these would not be necessary.

7.1 Further Work

Further work can be done to optimize individual features. Shadows can be optimized using scissor testing, which limits the size of the shadow volume. Pixel and Vertex Shaders can be optimized and written in assembly which could be faster than a high level language version.

Other techniques could also be implemented to extend the testing. Shadow maps which include features such as soft shadows could be implemented. More CPU based calculations could be used for example silhouette detection combined with Progressive meshes and occlusion culling which might result in faster shadow rendering. Other lighting models could be used which better simulate real world lighting. Multiple lights to be rendered in a single pass increasing speed when dealing with multiple lights. Different types of lights such as spot or area lights would allow for tighter bounding volumes for lit objects and shadows. Ultimately this implementation would allow for a base system to provide further testing in the topic of rendering expensive small environments in real time.

References

- [ENG03] Engel, Wolfgang Implementing Lighting Models with HLSL,
http://www.gamasutra.com/features/20030418/engel_01.shtml
- [EV02] Everitt, Cass and Kilgard, Mark J. Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering,
http://developer.nvidia.com/object/robust_shadow_volumes.html March 12, 2002
- [FER03] Fernando, Randima and Kilgard Mark J., The Cg Tutorial, Addison-Wesley Pearson Education, 2003
- [FRAZ00] Frazier, Ronald Advanced Real-Time Per-Pixel Lighting in OpenGL,
http://www.ronfrazier.net/apparition/index.asp?appmain=research/%20advanced_per_pixel_lighting.html
- [HOP98] Hoppe, Hugues Efficient Implementation of Progressive Meshes,
<http://research.microsoft.com/~hoppe/efficientpm.pdf> January 1998
- [KIL00] Kilgard, Mark J. A Practical and Robust Bump-mapping Technique for Today's GPUs, <http://www.nvidia.com/attach/1512>, 2000
- [SUR03] Surdylescu, Razvan Cg Bump mapping,
<http://gamedev.net/reference/articles/article1903.asp>
- [SUR03] Surdylescu, Razvan Cg Shadow Volumes,
<http://www.gamedev.net/reference/articles/article1990.asp>
- [WEL04] Welsh, Terry Parallax Mapping with Offset Limiting A Per-Pixel Approximation of Uneven Surfaces, http://www.infiscape.com/doc/parallax_mapping.pdf. January 18, 2004

Appendix A

Scene	
Name	TreeScene
Size(Triangles)	16280
Size(Bytes)	2735040
Textures	4

Settings	Texturing	Bumpmap	Textured	None
VBO = Yes	Avg FPS	37.86	47.93	56.75
Lights = 5	Min FPS	33	43	52
Shadows = Yes	Max FPS	42	43	61

Settings	Texturing	Bumpmap	Textured	None
VBO = Yes	Avg	135.61	160.3	213.2
Lights = 1	Min	123	147	192
Shadows = Yes	Max	157	184	237

Settings	Texturing	Bumpmap	Textured	None
VBO = Yes	Avg	62.7	96.64	141.89
Lights = 5	Min	55	88	140
Shadows = No	Max	69	105	144

Settings	Texturing	Bumpmap	Textured	None
VBO = Yes	Avg	201.55	260.95	441.27
Lights = 1	Min	175	230	427
Shadows = No	Max	226	294	457

Settings	Texturing	Bumpmap	Textured	None
VBO = No	Avg	29.64		
Lights = 5	Min	28		
Shadows = Yes	Max	32		

Settings	Texturing	Bumpmap	Textured	None
VBO = No	Avg	102.25		
Lights = 1	Min	97		
Shadows = Yes	Max	111		

Settings	Texturing	Bumpmap	Textured	None
VBO = No	Avg	44.55		
Lights = 5	Min	41		
Shadows = No	Max	47		

Settings	Texturing	Bumpmap	Textured	None
VBO = No	Avg	134.59		
Lights = 1	Min	124		
Shadows = Yes	Max	144		

Table 4: Scene 1 FPS measurements On NVIDIA HW

Scene	
Name	TreeScene
Size(Triangles)	16280
Size(Bytes)	2735040
Textures	4

Settings	Texturing	Bumpmap	Textured	None
VBO = Yes	Avg FPS	35.13	45.95	57.5
Lights = 5	Min FPS	30	42	53
Shadows = Yes	Max FPS	41	47	61

Settings	Texturing	Bumpmap	Textured	None
VBO = Yes	Avg	138.51	166.2	216.5
Lights = 1	Min	128	157	201
Shadows = Yes	Max	164	194	254

Settings	Texturing	Bumpmap	Textured	None
VBO = Yes	Avg	85.4	110.36	185.23
Lights = 5	Min	77	102	160
Shadows = No	Max	89	120	195

Settings	Texturing	Bumpmap	Textured	None
VBO = Yes	Avg	240.5	330.67	550
Lights = 1	Min	221	312	460
Shadows = No	Max	259	340	563

Settings	Texturing	Bumpmap	Textured	None
VBO = No	Avg	27.59		
Lights = 5	Min	25		
Shadows = Yes	Max	31		

Settings	Texturing	Bumpmap	Textured	None
VBO = No	Avg	103.5		
Lights = 1	Min	97		
Shadows = Yes	Max	105		

Settings	Texturing	Bumpmap	Textured	None
VBO = No	Avg	42.2		
Lights = 5	Min	39		
Shadows = No	Max	45		

Settings	Texturing	Bumpmap	Textured	None
VBO = No	Avg	140.25		
Lights = 1	Min	132		
Shadows = Yes	Max	147		

Table 5: Scene 1 FPS measurements On ATI HW

Scene	
Name	TreeScene
Size(Triangles)	16280
Size(Bytes)	2735040
Textures	4

Settings	Texturing	Bumpmap	Textured	None
VBO = Yes	ATI : Nvidia	0.93 : 1	0.96 : 1	1 : 0.99
Lights = 5	ATI / Nvidia	0.93	0.96	1.01
Shadows = Yes	ATI - Nvidia	-2.73	-1.98	0.75

Settings	Texturing	Bumpmap	Textured	None
VBO = Yes	ATI : Nvidia	1 : 0.98	1 : 0.96	1 : 0.98
Lights = 1	ATI / Nvidia	1.02	1.04	1.02
Shadows = Yes	ATI - Nvidia	2.9	5.9	3.3

Settings	Texturing	Bumpmap	Textured	None
VBO = Yes	ATI : Nvidia	1 : 0.73	1 : 0.88	1 : 0.77
Lights = 5	ATI / Nvidia	1.36	1.14	1.31
Shadows = No	ATI - Nvidia	22.7	13.72	43.34

Settings	Texturing	Bumpmap	Textured	None
VBO = Yes	ATI : Nvidia	1 : 0.84	1 : 0.79	1 : 0.8
Lights = 1	ATI / Nvidia	1.19	1.27	1.25
Shadows = No	ATI - Nvidia	38.95	69.72	108.73

Settings	Texturing	Bumpmap	Textured	None
VBO = No	ATI : Nvidia	0.93 : 1		
Lights = 5	ATI / Nvidia	0.93		
Shadows = Yes	ATI - Nvidia	-2.05		

Settings	Texturing	Bumpmap	Textured	None
VBO = No	ATI : Nvidia	0.99 : 1		
Lights = 1	ATI / Nvidia	0.99		
Shadows = Yes	ATI - Nvidia	-1.25		

Settings	Texturing	Bumpmap	Textured	None
VBO = No	ATI : Nvidia	0.95 : 1		
Lights = 5	ATI / Nvidia	0.95		
Shadows = No	ATI - Nvidia	-2.35		

Settings	Texturing	Bumpmap	Textured	None
VBO = No	ATI : Nvidia	1 : 0.96		
Lights = 1	ATI / Nvidia	1.04		
Shadows = Yes	ATI - Nvidia	5.66		

Table 6: Comparison of ATI to Nvidia Performance