

Real Time Rendering of Expensive Small Environments

Colin Branch
Stetson University

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 create complex effects which increase realism.

The goal of this paper to propose a set of features which allows small environments to have comparable realism to much larger environments. 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 will allow for analysis of the performance cost and saving of each feature as well as comparison of the capabilities of modern graphics hardware. This should allow for performance measurements that have absolute and relative statistics on different hardware.

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 redrawing 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. The fact that early hardware was geometry limited,

algorithms for real time rendering focused on limiting the number of triangles sent to the graphics card for rendering. This focus was marginalized as hardware developed to process larger and larger amount geometry loads.

Features of newer hardware are not limited to faster rendering speeds. One of the most powerful, complex, and slow is the ability to program the components that control the vertices and fragments outputs. These programmable components allow for extremely detailed rendering at much higher costs compared to simpler fixed pipelines.

Since the added effects of these components add considerable amount of detail to a rendered image, it is desirable to be able to render scenes in real-time efficiently. Thus it becomes pertinent to

develop algorithms that attempt to limit the number of vertices and fragments that are processed by programmable components.

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 the norm. 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 all that is required 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 environment 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.

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 dynamically generated shadows. The methods used to increase the speed of rendering of this environment will be progressive meshing, hardware occlusion culling, level-of-detail, and multiple pass rendering. Implementation of these features should create 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 the fixed algorithm lighting models. 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 allow 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 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. Combined with the fact that many pixels will be drawn over with later geometry leads to the fact that time is wasted on elements that are not even visible.

The implementation will include at least one and possibly more lighting models that use both per-vertex and per-pixel calculation. Using per-pixel lighting should contribute significantly to the cost of rendering [3]. The performance will be highly dependent on the speed at which the video card can run vertex and fragment programs.

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 realized 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:

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

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 a problem 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 problem 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 what ever light model is desired.

The implementation will include normal mapping with some lighting models. Since normal mapping will require per-pixel lighting, only objects rendering using per-pixel lighting can use it. Also since additional materials and geometry data is required the cost to render normal mapping should be considerable.

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 which is exactly what a shadow is. 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 maps. When rendering the model these light maps are added to the existing texture maps and drawn giving the appearance of lighting and shadows. This technique is very inexpensive at render time since the hard work is done in preprocessing when creating the map. Since 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. However 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 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 represent large numbers 100 or more 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 not[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 an increment. Next the front faces are rendered with the stencil buffer decrementing. 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. Another possible method is to compute a shadow map for the entire screen using the stencil buffer and the depth buffer and then processes this texture using a blending method that takes the distance each shadowed pixel is from the light and blends accordingly. This method is also extremely slow, however once fragment shaders become faster, this method should be more successful.

Shadow volumes will be implemented using only vertex shaders. This method doesn't require calculating silhouette edges on the CPU however it requires the entire model to be rendered multiple times. Thus it would be prudent to use culling to determine if for the object is in the view and if the shadow of the object is in view. Also this method requires fast vertex processing on the video card.

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. Thus the following methods will be implemented to try and offset the cost and allow for real-time performance.

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 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.

The implementation should allow for much faster volumes shadow since a lower quality mesh could be used for the shadow itself.

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 progressive meshes and shadow meshes will be implemented. Using the bounding volume of the object and the light that the object is under the influence of, it is possible to generate a cheap bounding shadow volume and test to see if it is in the frustum. There is no easy way without to see if an object's shadow is visible, however by simply testing the geometry this way you can determine if the shadow could 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.

The implementation of the environment will have occlusion culling on expensive models. Models can be classified based on their size and composition, such as whether it has normal map. Some models can be assumed to be visible such as terrain and architecture. Thus by rendering them first, then sorting the remaining objects based on distance to the viewer it is possible to render these objects based on the following algorithm:

```
sortNearest(objects)
for( object = nearest; object <= farthest; object next)
{
    if occluded(object) > threshold
        render(object)
}
```

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 will be considered when rendering the environment. This technique should be successful however there is little literature on the subject and thus it's complexity to implement or its affect are unknown.

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 considerably harder than for other objects.

These techniques should maintain a high quality of visuals while decreasing rendering time. Varying the level of detail settings may allow for tweaking of performance as well as suitability for different hardware.

5 Testing

Testing will measure both the effects of the methods and the hardware that runs them.

5.1 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 5.1.

Table 5.1

	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

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.

5.2 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 that will be turned off will be:

Table 5.2

Methods	States
Normal Mapping	ON/OFF
Per-Pixel Lighting	ON/OFF
Frustum Culling	ON/OFF
Occlusion Culling	ON/OFF
Light Culling	ON/OFF
Shadow s	ON/OFF
LOD	CONSERVATIVE/LIBERAL/OFF

There are 6 binary states which

allows for 2 to the 6th power or 64 possible settings, multiplied by the three LOD settings gives a total of 192 possible different settings for rendering the environment. Using scripting, it will be possible to test each individual setting automatically and thus not require individual tests to be run manually.

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.

6 Conclusion

In this project the goal of the testing is to show the cost in terms of performance, positive or negative, of each method implemented. Certain predictions can be made based on knowledge of each method. A feature like per-pixel lighting will be dependent solely on the performance of the fragment capabilities of the graphics card. While other features such as shadow volumes are affected by nearly every feature of the card, such as the memory bandwidth, vertex and fragment pipelines, and the fill rate. The actual cost especially relative to each additional feature has not been analyzed in-depth. Application of this analysis would be able to measure the benefit of a feature based on its added realism at a specific cost which can be both absolute and relative to other features. One benefit of this project is it would test to see if these features can be used with current hardware in a real-time simulation. Assuming this is successful it leads to questions on how to improve realism. If this goal is not met, then it can lead to questions on how to improve performance. Thus in any case, useful insights into this area of graphics research can be gained.

References

1. Randima Fernando and Mark J. Kilgard, *The Cg Tutorial*, Addison-Wesley Pearson Education, 2003
2. Wolfgang Engel, *Implementing Lighting Models with HLSL*,
http://www.gamasutra.com/features/20030418/engel_01.shtml
3. Ronald Frazier, *Advanced Real-Time Per-Pixel Lighting in OpenGL*,
http://www.ronfrazier.net/apparition/index.asp?appmain=research/%20advanced_per_pixel_lighting.html
4. Razvan Surdylescu, *Cg Bump mapping*,
<http://gamedev.net/reference/articles/article1903.asp>
5. Hugues Hoppe, *Efficient Implementation of Progressive Meshes*,
<http://research.microsoft.com/~hoppe/efficientpm.pdf> January 1998
6. Razvan Surdylescu, *Cg Shadow Volumes*,
<http://www.gamedev.net/reference/articles/article1990.asp>
7. Cass Everitt and Mark J. Kilgard, *Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering*,
http://developer.nvidia.com/object/robust_shadow_volumes.html March 12, 2002
8. Mark J. Kilgard, *A Practical and Robust Bump-mapping Technique for Today's GPUs*,
<http://www.nvidia.com/attach/1512>, 2000