Developing a Real-Time Renderer With Optimized Shadow Volumes

Mátyás Premecz (email: pmat@freemail.hu)
Department of Control Engineering and Information Technology,
Budapest University of Technolgy
Hungary

Abstract

This paper presents a collection of ideas that are needed for developing an efficient, general purpose, real-time rendering engine with recent hardware's possibilities taken into consideration. The principals of several visual effects seen in recent games (like tangent space Phong illumination, bump mapping, parallax mapping and shadow volumes) are discussed. A new way of generating shadow volumes is proposed that helps the reduction of render passes efficiently.

Keywords: lighting, culling, shadow volumes, reducing render passes, real-time rendering

1 Introduction

Today, many papers are presented that focus on various visual effect algorithms, but only a few discuss the possibilities and difficulties of implementing a renderer that combines them. This topic is very complex, because the structure of a renderer must be general, modular and upgradeable so it could efficiently render large scenes or be extended with new effects any time.

In this paper, we collect some of the most commonly used effects and try to combine them into a real-time renderer. Our main purpose is to create a shadowing and lighting mechanism that can handle multiple lights and dynamic shadows and unlike many shadow volume based renderers, handle more than one light in a pass, thus reducing the number of necessary draw calls. This improves performance for at least two reasons:

1. Better batching which is a key factor of speed today because recent GPUs are faster than CPUs. That causes that the frame-rate can easily become limited by the CPU overhead of the draw calls and state changes.
2. Reduction of the amount of redundant work that has to be done in every render pass (like texture look-ups, vertex transformations, normal computations, etc).

The basic idea of reducing render passes is to clip the shadow volume to the bounding box of the light source, and to use dynamic branching in the light's pixel shader. In addition, other optimization methods are presented to reduce fill-rate and unnecessary shader work. The renderer is discussed in details in Section 3. The various effects used in this paper are summarized in Section 2.

2 Description of the Effects Used

2.1. Phong Illumination

Phong illumination is one of the basic local illumination lighting models used in today's games. It is very well explained in [1], so only a brief explanation is given here.

In this model, real world lighting is approximated by three components: ambient, diffuse and specular. Their purpose is to mimic the following phenomena:

- The light that is reflected and scattered many times and comes from almost everywhere is modeled by the ambient component.
- The light source's direct light that hits a rough surface and gets scattered equally to every direction is modeled by the diffuse component.
- The light that gets reflected near to the ideal reflection direction (for example on metallic surfaces) is modeled by the specular component.

There is a variation of the Phong model that is important for us. If all the vectors used above are computed in tangent space, the lighting can be combined with further effects like bump mapping and parallax mapping. Tangent space (also known as texture space) is a coordinate system that is aligned by the surface. It is formed by three perpendicular vectors which can be easily pre-computed for any well-textured mesh. The vectors are:

- Tangent vector: it is parallel to the direction of increasing s or t on a textured surface

- Normal vector: it is perpendicular to the local surface.

- Bitangent vector (also known as the binormal – which is not correct): it is the cross product of the Tangent and the Normal vectors.

Further on this topic can be found in [1] and [2].

2.2. Bump Mapping

Bump mapping is a technique to improve visual complexity through texture mapping and per pixel lighting. The normal vector of the surface gets perturbed before the light calculations. Thus, high frequency detail can be added without the need of higher tessellation of the meshes. The perturbation is based on a texture map that usually contains the modified normals. This texture

map can be easily computed from a height map. Bump mapping gives very good results with Phong lighting and can be combined easily with parallax mapping.

2.3. Parallax Mapping

Parallax mapping is simple texture coordinate manipulation trick. When used with bump mapping, it comes almost for free, and improves visual quality very much. However, it is based on an assumption, thus can cause disturbing artifacts in some cases.

When this technique is used, not only the normal gets perturbed, but the texture coordinates used to index the color and the normal textures are modified as well according to the height of the actual point. As can be seen in **Figure** 1, the original T0 texture coordinates get substituted by T1, which is calculated from the direction of the tangent-space eye vector and the height value read from a texture at point T0. All the details can be found in [3].

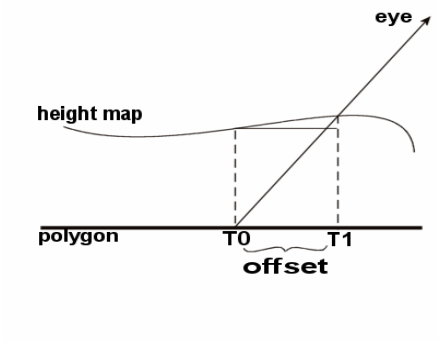

Figure 1: *The principles of parallax-mapping*

2.4. Shadow Volumes

Many game engines use shadow volumes, and in today's world of real-time soft shadows, the basic algorithm is still popular. The topic is very well documented, so only a short explanation is provided here. For further details, [4] and [5] are recommended readings, because they are very thorough discussions about ways of implementation and optimization.

Essentially, shadow computation is a decision for every point: the point is in shadow with regard to a light source if there is no line of sight between that point and the light. This decision can be made with algorithmically generated helper geometry, called the shadow volume. The shadow volume of a triangle is a triangular frustum capped on top by a triangle itself and extending away from the light to infinity. The shadow volume of a compound object is the union of the shadow volumes of its component triangles. This volume contains all the points that lie in shadow of the given light source. The interior geometry of the volume can be removed which greatly simplifies it. After creating the shadow volume, the shadow determination can be reduced a simple point in polyhedra test which can be efficiently accelerated with the stencil buffer of current graphics hardware [5].

The algorithm in a nutshell is as follows:

1. Render the whole scene to depth buffer only.
2. Disable depth (and color) writes, enable stencil writes.
3. Clear stencil to 0, set stencil function to ALWAYS.
4. Render back faces of shadow volume. If depth test fails, increment stencil value, else does nothing.
5. Render front face of shadow volume. If depth test fails, decrement stencil value, else do nothing.
6. Render light only where the stencil buffer has a value of 0.

The algorithm has its pros and contras:

- The computed shadows are very accurate, self shadowing and point lights are handled well (in contrast to shadow mapping).
- The algorithm is rather CPU and GPU fill-rate consuming, though this can be reduced effectively.
- The generated shadows are always hard, though can be softened by more sophisticated algorithms.
- Only one light can be rendered in a pass, because only one shadow mask can be stored in the stencil buffer. This feature limits the number of displayable lights strongly, because each light needs a full render pass with all the transformations, state changes, texture look-ups and shader math repeated.

This very last problem is thoroughly discussed in Section 3.

3 The Renderer

3.1. Performance Considerations

Before our first attempt to combine the above effects into a complete system we have to consider the features of the hardware system we use. Typically a graphics workstation (that is a computer with a 3D accelerator unit) has two processor units today. This fact introduces the problem of synchronisation and task scheduling between the two components. These units differ in speed, capacity and capabilities, so care must be taken to what to do, where to do and when to do.

Recent GPUs are strongly pipelined systems where each stage needs the data from the previous part to do its job. That means that the whole system works only as fast as the slowest stage does. Thus, the best thing to do is to balance the workload between the stages, and focus optimization efforts on the slowest.

For these reasons, we review the structure of the composite CPU-GPU pipeline (all stages are on the GPU from stage 2):

1. CPU – All the game logic, physics, AI, I/O, sound and music is done here. Shadow volume generation and animation can be done here.
2. Geometry Storage – This is done usually in video memory, with the help of vertex buffers or display lists.
3. Texture Storage – Though not really a stage, it is mentioned here because it is usually very limited by memory bandwidth.
4. Geometry Processor – This is the stage, where coordinate transformations, animations and procedural geometry generation (like shadow volumes if not done on CPU) can take place. Vertex shaders run here.
5. Rasterizer – The stage where our primitives (points, lines, polygons) become fragmented to displayable pixels. We can only affect this stage indirectly.
6. Fragment Processor – This is where the actual colour of the pixels is computed. Pixel shaders run here. Texturing, filtering, blending, and numerous per pixel tests are done here. Usually this is the most overloaded stage in today's applications, due to the continuously extending capabilities of pixel shaders.
7. Frame buffer – The last stage. This is where the displayable picture is stored.

For further details and ideas on how to balance the pipeline, refer to [7].

3.2. The 'Brute Force' Method

After this short review, we work out the outlines of a simple renderer, without any optimizations. What we want: a renderer that can handle a scene with static and animated objects with different materials, textures with bump mapping and parallax mapping, shadows, multiple dynamic and local point light sources with Phong illumination. In this paper, we consider point lights only, but the discussed methods can be modified easily for other types, like spot or directional lights.

First, we need to store the objects and other entities somehow. There are sophisticated methods for this, like a scene-graph, but for us the use of dynamic lists is enough. However, if the scene is very complex, a powerful, large scale but not-so-precise culling algorithm is needed (like a binary space partitioning tree or a portal system) to prevent the processing of objects that are not visible. For now, it can be assumed that our renderer works with the output of such an algorithm. This means that the objects and lights in our lists are probably visible, but not necessarily. Our 'brute force' method will neglect these cases, and render everything.

The main steps:

1. Render the whole scene with ambient light. Disable depth writes.
2. Take a light source that is not rendered yet. Compute all the shadows it generates, in our case generate shadow volumes and render the shadow mask into the stencil buffer with depth and color writes disabled.
3. Render the scene illuminated by that light source, with the use of bump mapping and parallax mapping with blending enabled and set to additive mode.
4. Go to 2 until all light sources are drawn.

This is all we need, if a scene is very simple, with very few light sources and shadow casters, but unfortunately, this is usually not the case, and the frame-rate can drop tremendously if the number of lights is increased. The truth is, many things are done unnecessary or multiple times. In the next subsection we try to find and eliminate them.

3.3. Finding Bottlenecks

Before the invention of various optimization techniques, it is necessary to analyse the workload of the graphics pipeline. This, of course, cannot be done without knowing the details of the actual application, but there are certain symptoms that typically occur. In most cases the overloaded stages are the CPU, the fragment processor and the frame buffer. The causes can be diverse, but the following ones are peculiar:

- The dynamic objects are animated on CPU, which is very time consuming because vertex positions, normal and tangent vectors, sometimes other attributes have to be recalculated every frame. Some of these tasks can be done on the GPU to relieve the CPU.
- Shadow volume generation can be computationally expensive, especially with multiple light sources and highly tessellated objects. Parts or whole of this task can be moved to the GPU, or alternative shadow generation methods can be applied.
- Texture fetching is fast until the textures are read coherently, which means that the neighbouring texels are read successively. This texture cache coherency can be broken with parallax mapping. Since parallax mapping must be done in every render pass, the only solution is to reduce the number of render passes.
- Render-state changes (like texture, shader of vertex buffer changes) can have large overhead on both the CPU and the GPU. A possible solution is to reduce state changes by the reordering of the objects by shaders, textures, etc. Another solution is the use of texture atlases.

- The frame-buffer can be a bottleneck due to the additive blending used to summarise the light sources' effects and the heavy use of stencil buffer.
- The fragment processor is the stage where pixel shaders run. Tangent-space Phong illumination with bump mapping and parallax mapping can be done fast assuming that only those parts of the frame-buffer is touched, that are affected by the current light source.

The last two problems are addressed in the next two subsections. In subsection 3.4., methods for speeding up one render pass are exposed. In subsection 3.5., the possibilities of reducing the number of render passes are discussed.

3.4. Speeding up Render Passes

The ideas in this subsection are based on the revelation that a point in a typical scene is lit by very few light sources regardless of the total number of lights. This implies that many of the objects are left unchanged in a typical lighting pass. In order to make use of this fact, we have to introduce the use of 'bounding geometries'. The bounding box of a model is a simple box that contains the whole model, and does not exceed its size too much. Even a light source can have a bounding box that contains the area that the light has an effect on. With the help of these bounding boxes, we can omit most of the redundant work.

We present three ideas, two of which reduce the frame-buffers usage, and one that reduce the necessary amount of animation and shadow calculations and the number of draw-calls.

1. All objects can be tested whether they intersect the light's bounding box or not. If not, all further work on that object can be skipped.

2. A shadow volume can be generated in a way that it does not exceed the light's bounding box much. This way, the stencil buffer's usage can be reduced and traded to additional CPU work. This is not always acceptable, but as shown in the next subsection, it can speed things up if combined with the trick proposed below.

3. When the shadow mask is drawn into the stencil buffer, the points that are not visible from the light source are discarded as they are not illuminated by that light. Actually, there are usually more points that are left unchanged, namely the ones that are further from the light than the radius of the light's affected sphere.

One way to mask out these points is doing a scissor test on the fragments. Scissor test is an easy way to prevent certain portions of the frame-buffer from refreshing. All we have to do is to draw a quad on the part of the screen that the light has an effect on. However, this approach has some drawbacks: many 3d accelerator cards slow down if scissor testing is used (some recent ATi models for example) and this test neglects the light source's in-depth position. This means that a part of the frame-buffer might be drawn even if the light does not affect any visible objects. **Figure 2** shows this situation. Notice that the whole bounding box is visible causing the rendering of a great part of the wall in the forefront that is not affected by the light. Another good example for this is a scene with a single object in the middle and a light that lies so far behind that its bounding box does not intersect the sphere. The bounding box of the light in the scissor buffer causes redundant work, because its in-depth position cannot be tested.

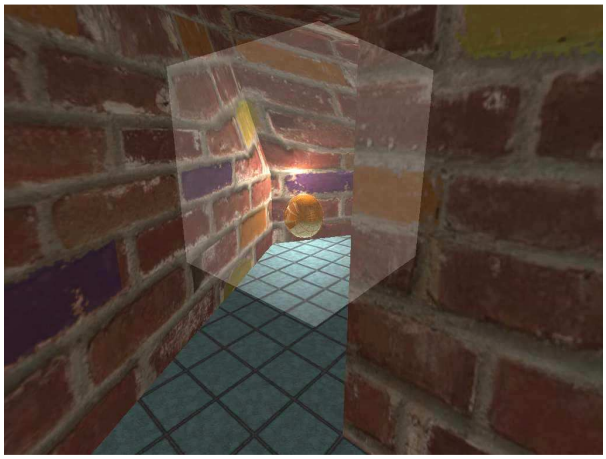

Figure 2: *the light's bounding box rendered into the scissor buffer without depth testing*

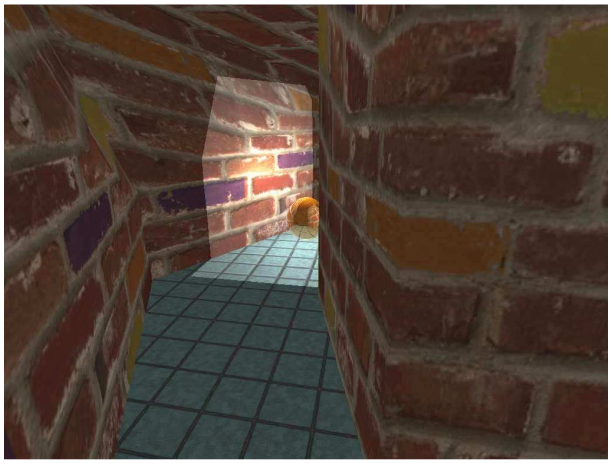

Figure 3: *the light's bounding box renderer into the stencil buffer with depth testing*

A solution to this problem is the use of the stencil buffer, because it is possible to affect the stencil buffer even if the depth-test fails and the result of the stencil test can differ depending on the depth-test. Thus, it is possible to localize the render pass to objects that truly intersect the light's bounding box. **Figure 3** shows such a situation. Notice that only the parts intersected by the bounding box are highlighted.

The steps to do this:

1. Render the scene with ambient color. Disable depth writes.
2. Clear stencil to 1 (notice it was 0 in subsection 2.4).
3. Render back faces of light's bounding box. If depth test fails, decrement stencil value (this implies that we are inside the bounding volume), else do nothing.
4. Render front face of light's bounding box. If depth test fails, increment stencil value (this implies that the bounding volume is completely behind the object and no drawing is required), else do nothing.
5. Render back faces of shadow volume. If depth test fails, increment stencil value, else do nothing.
6. Render front face of shadow volume. If depth test fails, decrement stencil value, else do nothing.
7. Render light only where the stencil buffer has a value of 0.

Note that this is the shadow volume algorithm already described, with two additional steps (3 and 4). With the use of these stencil culled lights, we can trade redundant pixel shader work to additional stencil work, which is usually a great win if the application's speed is limited by the fragment processor. Furthermore, this trick is an important element of the render pass reduction technique proposed in the next subsection.

3.5 Reducing the Number of Render Passes

The simplest method of reducing the number render passes is frustum culling. If the bounding box of a light source does not intersect the visible frustum (it is behind us for example), there is no need to draw the pass of that light. The following technique is more sophisticated, and should be used only in case of complex or moderately complex scenes, with lots of textures and long pixel shaders.

The main reason, why every light needs a render pass when using shadow volumes is that shadow volumes extend to infinity even if the light is located to a small area. So, in order to render multiple lights in one pass, the shadow volumes generated by each light should be restrained in a way that they cannot intersect each other. This is only possible if the lights we want to draw are not too near to each other. In other words, their bounding boxes do not have shared points. So the first task is to form groups of the lights. Lights in a group are all far enough from each other. The second task is more complex and computationally expensive.

The shadow volumes generated by a light source must be clipped that it has no parts out of the light's bounding box. This can be done by clipping the shadow volume to all six planes of the faces of that box.

With the bounding boxes not intersecting each other and the shadow volumes already clipped, it is possible to fill in the stencil buffer with a shadow mask that is correct for all the lights, because there is no part of the stencil that belongs to more than one light. This is achieved by combining the second and third trick from the previous subsection.

After the stencil is prepared, a pixel shader is needed that can handle multiple lights, and dynamically save the calculations of a light that is too far. This is possible with the dynamic branching of Shader Model 3.0.

Finally, the complete renderer:

1. Render the whole scene with ambient light. Disable depth writes.
2. Choose all the lights that intersect the frustum.
3. Form groups of the chosen lights that do not intersect each other.
4. Take a group of light sources that is not rendered yet. Compute all the shadow volumes and clip them to the appropriate light's bounding box. Render the lights' bounding boxes and the shadow volumes into the stencil buffer with depth and color writes disabled.
5. Render the scene illuminated by that group of light sources, with the use of bump mapping and parallax mapping with blending enabled and set to additive mode. Do not render objects that are not illuminated by the actual group of lights. Use dynamic branching in the shader based on the light sources' distance.
6. Go to step 4 until all light sources are drawn.

4 Results

In this section, experimental results for the improved renderer are presented. The 'brute force' method is measured also for reference. The algorithms were tested in two scenes:

1. A moderately complex scene that consists of a torus shaped room, some 'statue-like thing' and a reflective sphere (**Figure 4**). The environment cube-map faces are refreshed with the same renderer and all the effects on. The cube-map size is 128. Three textures are used. There are 7 objects, 1422 vertices and 1368 faces. There are five moving light sources present, which barely intersect each other.

Table 1 shows the results of the Torus scene.

	Min. value	Average	Max. value
Optimized renderer	123	169	193
Reference	44	53	60

Table 1: *The measured frame-rates of the Torus scene*

2. A complex scene that consists of two rooms with several static objects and a reflective box (**Figure 5**). Seven textures and different material settings are used. There are 28 objects, 10992 vertices and 14288 faces.

There are five moving light sources with a continuously changing overlapping factor.

Table 2 shows the results of the Rooms scene.

	Min. value	Average	Max. value
Optimized renderer	69	101	210 *(in a corner where only one light is visible)*
Reference	34	47	71 *(as above)*

Table 2: *The measured frame-rates of the Rooms scene*

The tests were done on an AMD Athlon XP 2200+ CPU and an NVIDIA 6800GT video card, in 1024x768x32 resolution.

5 Conclusion

We have presented the concept of a simple, general purpose rendering system that handles multiple light sources, dynamic lights and shadows with Phong illumination. The measured frame-rates of the 'brute-force' approach shows that even the strongest cards get on the knees when treated incautiously.

The first rows in **Table 1** and **Table 2** show that the optimization methods proposed in this paper are proved to be efficient, though it is important to notice that there is no performance gain in the worst case, when all lights have an effect on every point on the screen.

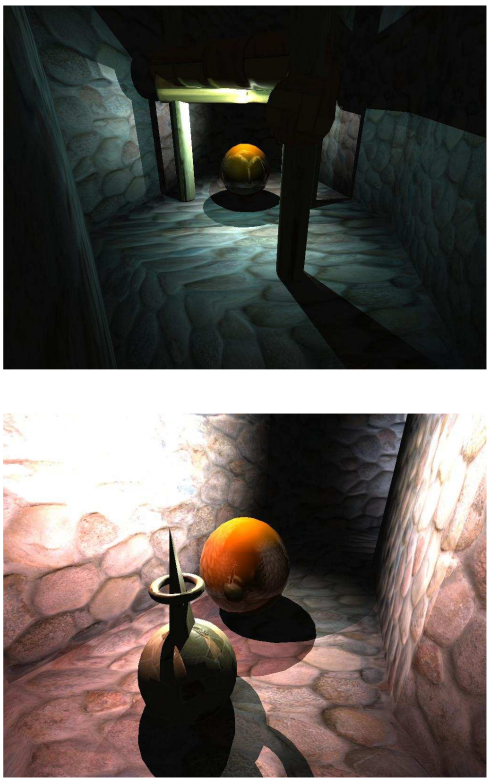

Figure 4: *Pictures from the Torus scene*

Figure 5: *Pictures from the Rooms scene*

References

[1] E. Persson (aka Humus): *Phong illumination*, http://www.humus.ca/, Feb 2003

[2] *Computing Tangent Space Basis Vectors for an Arbitrary Mesh*, http://www.terathon.com/code/tangent.html

[3] T. Welsh: *Parallax mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces*, Infiscape Corporation, Jan 2004

[4] M. McGuire, J. F. Hughes, K. T. Egan, M. J. Kilgard, C. Everitt: *Fast, Practical and Robust Shadows,* http://developer.nvidia.com, Nov 2003

[5] H. Y. Kwoon: *The Theory of Stencil Shadow Volumes,* http://www.gamedev.net/columns/hardcore/shadowvolume/

[6] F.C. Crow: *Shadow Algorithm for Computer Graphics,* In SIGGRAPH '77 Proceedings, 1977

[7] Ashu Rege, Clint Brewer: *Practical Performance Analysis and Tuning*, NVIDIA, GDC Presentation, 2004