Recent Advances in Monte Carlo Offline and High-Quality Real Time Rendering

Lecture #7: Wednesday, 27 September 2009
Lecturer: Ravi Ramamoorthi
Scribe: Terrence Zhao

1 Introduction

The first half of this lecture goes over where the previous left off. We discuss some ways in which to improve the efficiency of our renderings by employing smarter sampling techniques such as lightcuts for scenes with complex illumination and adaptive wavelet sampling to aim more samples at regions that need them more. In the second half we being talking about real-time techniques for rendering high quality images such as using pre-computation, real-time ray tracing, and the GPU's programmable capabilities.

2 Rendering Complex Illumination Scenes

Although many computer generated scenes often assume simple illumination models, illumination in the real world is often very complex. A scene can consist of light being emitted from objects of complex shapes with the light reflected in every direction.

2.1 Representing Our Light Sources

In order to unify our representation of illumination, all lights, whether they be indirect, area lights, HDR environment maps, or sun and sky lights can be replaced with many point lights. However, sampling from millions of point lights can become prohibitively expensive. We can avoid checking each and every light, making our algorithms more scalable, by realizing that at certain distances, groups of point lights can be approximated by a single point light. The key idea here is to build a hierarchy of lights (a tree) where the individual point lights are at the lowest level and form the leaves of the tree. Individual point lights that are in close proximity to each other can be group together and approximated as a single light source which becomes the parent of the grouped lights. This act of clustering lights can be repeated until all of the lights in the scene are grouped and approximated by a single light source, which acts as the root of the light tree. An example of a light tree is shown in Figure 1.

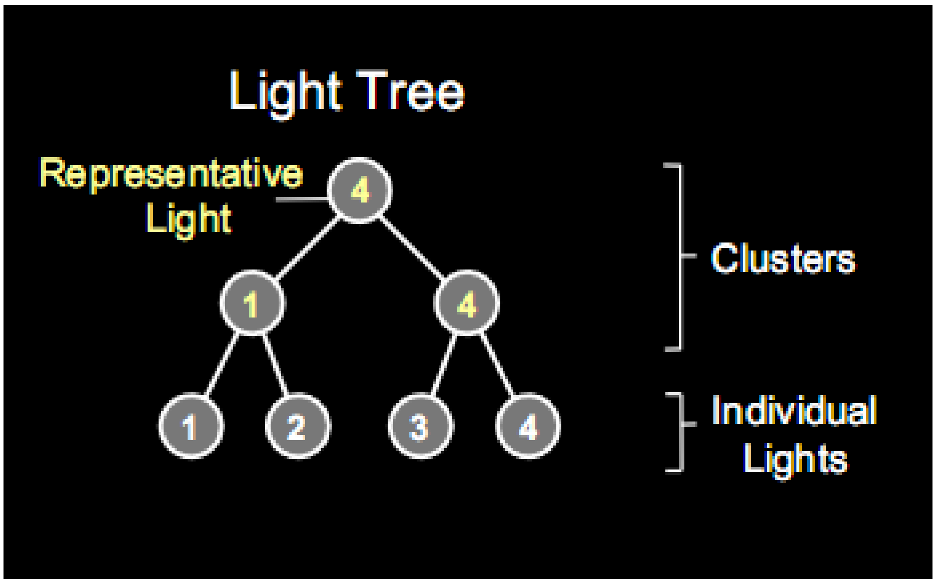


Figure 1: showing an example of a light tree. The individual lights make up the leaf nodes of the tree while their clusters are the parent nodes.

2.2 Light Cut

With this hierarchy of light representations, we can at any time choose to look at a group of lights as a single light source or look at them as individual lights. We can control the granularity of our light representations by choosing a depth in the light tree. For some lights, depending on how far the surface we are rendering is from them, we may like to choose lights higher up in the hierarchy in order to obtain a summary of the cluster of lights while for other lights we may choose the nodes deeper in the tree in order to acknowledge the individual point lights. Our selection of the nodes in the tree form a Lightcut. Examples of lightcuts are shown in Figure 2.

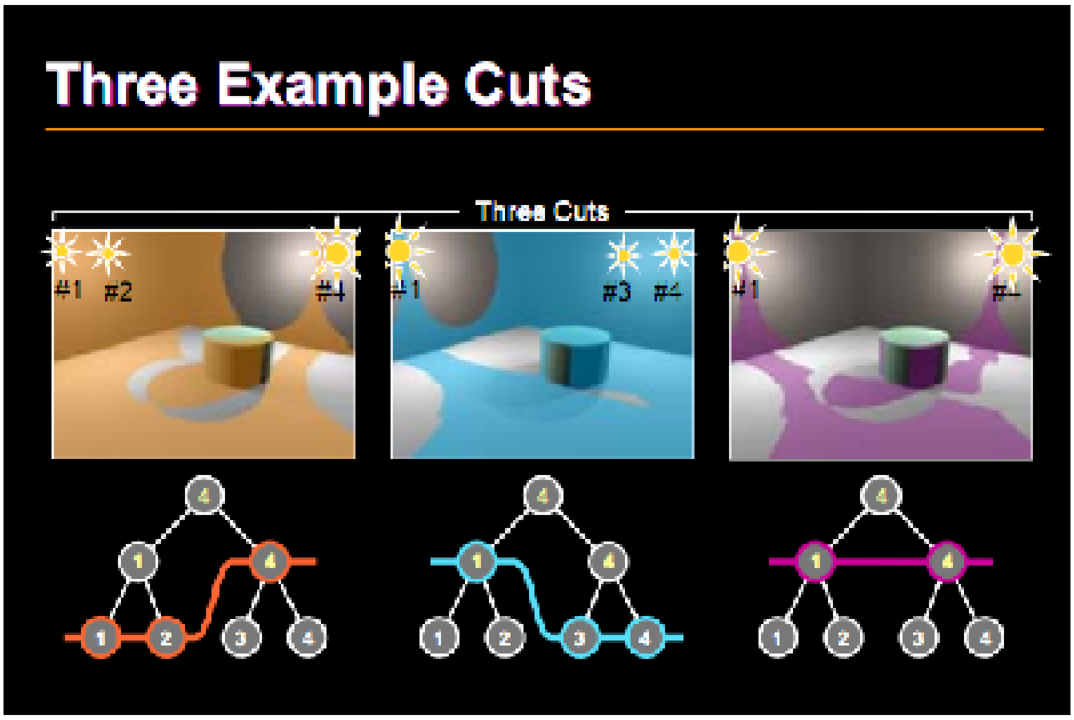


Figure 2: Examples of lightcuts. In the leftmost light cut, the individual light points #1 and #2 are left as individual point lights but #3 and #4 are clustered and represented by just #4. In the center lightcut, the #3 and #4 are left as individual point lights but #1 and #2 are represented by just #1. Finally, the lightcut at the right chooses to represent the four lights as two clusters with just #1 and #4.

3 Adaptive Wavelet Rendering

Regions with lower frequencies tend to be less exciting so less samples are needed to depict the area accurately. High frequency regions, however, have great variations in color and energy so more samples are needed to capture all the details. However, when implementing multi-dimensional effects such as motion blur, anti-aliasing, or depth of field, most adaptive algorithms look at either the rate of variation in an image or the rate of variation in other dimensions but not both. One insight that adaptive wavelet rendering presents is that areas with low frequency in the image but high frequency in other dimensions should require similar number of samples as areas with high-frequency in the image but low-frequency in other dimensions so it is important to take both into account. Adaptive wavelet rendering employs wavelet transformations in order to estimate the image in the wavelet domain in order to obtain a multi-scale view of the image. One wavelet transformation is the Haar Wavelet Decomposition.

3.1 Haar Wavelet Decomposition

Suppose we have an array of pixel values as our image. We can use a wavelet transformation to bring these values into a frequency and spacial domain, similar to how the Fourier Transform that you may be familiar with takes signals into a frequency domain representation. The wavelet transform is done as follows:

Take the first two pairs of values in the array and compute the sum and difference, then divide both values by two. The first value (sum divided by two) in our new pair of values is the average of our original pair while the second value (difference divided by two) keeps track of the information we need to know how our average was created.

Repeat this for every subsequent pairs until we have all the sum and difference values. Now let's ignore the difference values for now and focus only on the sum values. We can repeat the above procedure on all of these sum values, averaging and differencing them. This creates a hierarchy until there is only one average and sum value. With each level of averaging that we do, the values become more and more similar and the frequency of our values decrease. Also, by keeping the difference values, we can recover our old values if we wish. By inspecting this representation of our pixel values, we can adaptively send more rays to regions we deem to have higher frequency in both the image and other dimensions.

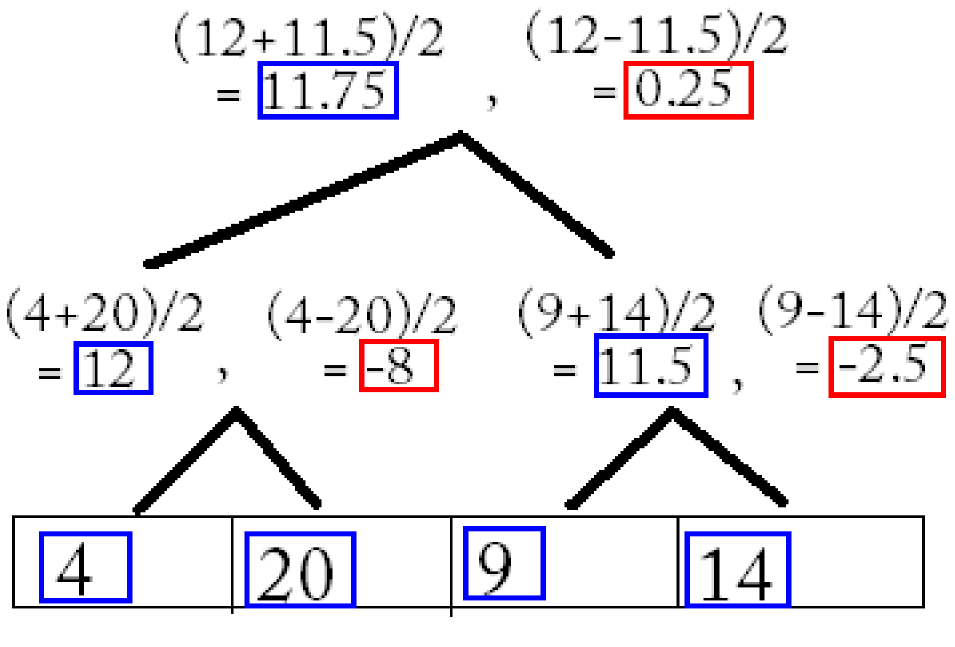


Figure 3: The Haar wavelet transform. The average values are in marked in blue while the difference values are marked in red.

4 High Quality Real-Time Rendering

Today, we have rendering techniques that can produce very photo-realistic and complex effects such as depth of field, area lighting, and global illumination. The effects of these advanced techniques can be seen in the film industry in the form of special effects and computer animation. Although the quality of these generated images can be so high that sometimes it is difficult to discern what is real and what is rendered, the algorithms that produce such images are extremely slow and can take hours or days to execute. If one wanted to render images in real-time, he or she would have to settle for images of much lower quality. With the vast increase in CPU and GPU power in the past few years, such interactive rendering has improved but the focus has primarily been on rendering increasingly complex geometry.

4.1 New Trend

Today the focus has shifted towards the quality of photo-realistic rendering and away from the quantity of polygons and textures. Three ways of rendering high quality images in real-time are techniques involving pre-computation, real-time ray tracing, and using the programmability of the GPU. In this lecture we mainly talk about GPU programming.

4.1.1 Pre-computation Methods

In pre-computation techniques, one tries to hold certain variables constant and then pre-compute values to be displayed later. In the example with the purple teapot in Figure 4, the camera and position of the geometry are kept static while the position of the light source is moved around. The effects of the light's position are pre-computed. Such techniques for real-time rendering involve sophisticated representations and algorithms.

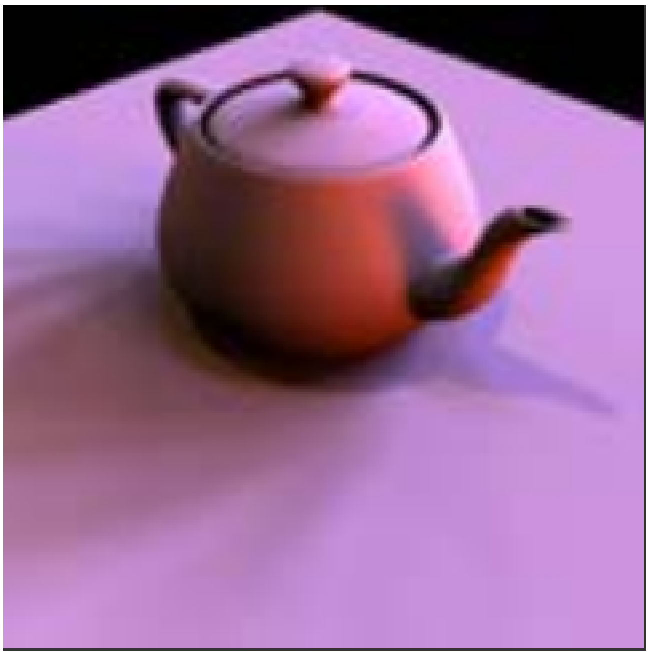


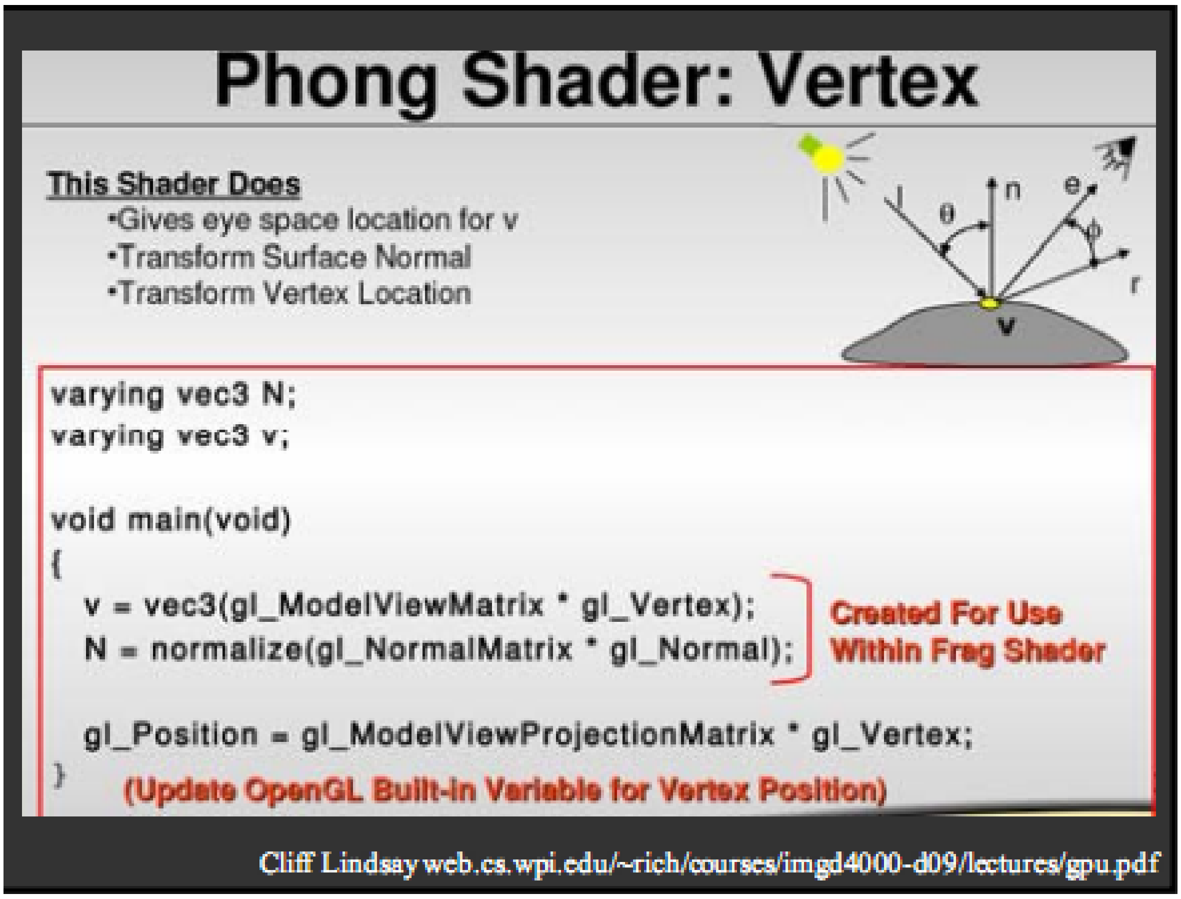
Figure 4: The effects of the light's position on the teapot are pre-computed.

4.1.2 Interactive Ray Tracing

Through vast improvements in CPU power, parallel programming, and leverage of caches, ray tracing can now be done in real-time. Ray tracing has many advances in that it is relatively easy to render complex scenes through the use of acceleration structures to speed up intersection tests. Complex materials, shadows, and other effects come for free while other effects such as global illumination are easy to add. However, global illumination and complex lighting are very expensive.

4.1.3 Programming the GPU

The current hardware pipeline can be broken down into two smaller pipelines: the geometry or vertex pipeline and the pixel or fragment pipeline. The vertex pipeline processes the vertices of the geometry while the fragment pipeline acts as the inner-loop to process the pixels inside the geometry. Both of these pipelines are programmable so one can write his or her own custom shaders, texture mappers, or signal processing routines. Shown below are the vertex and fragment portions for a phong shader.



Phong Shader: Vertex

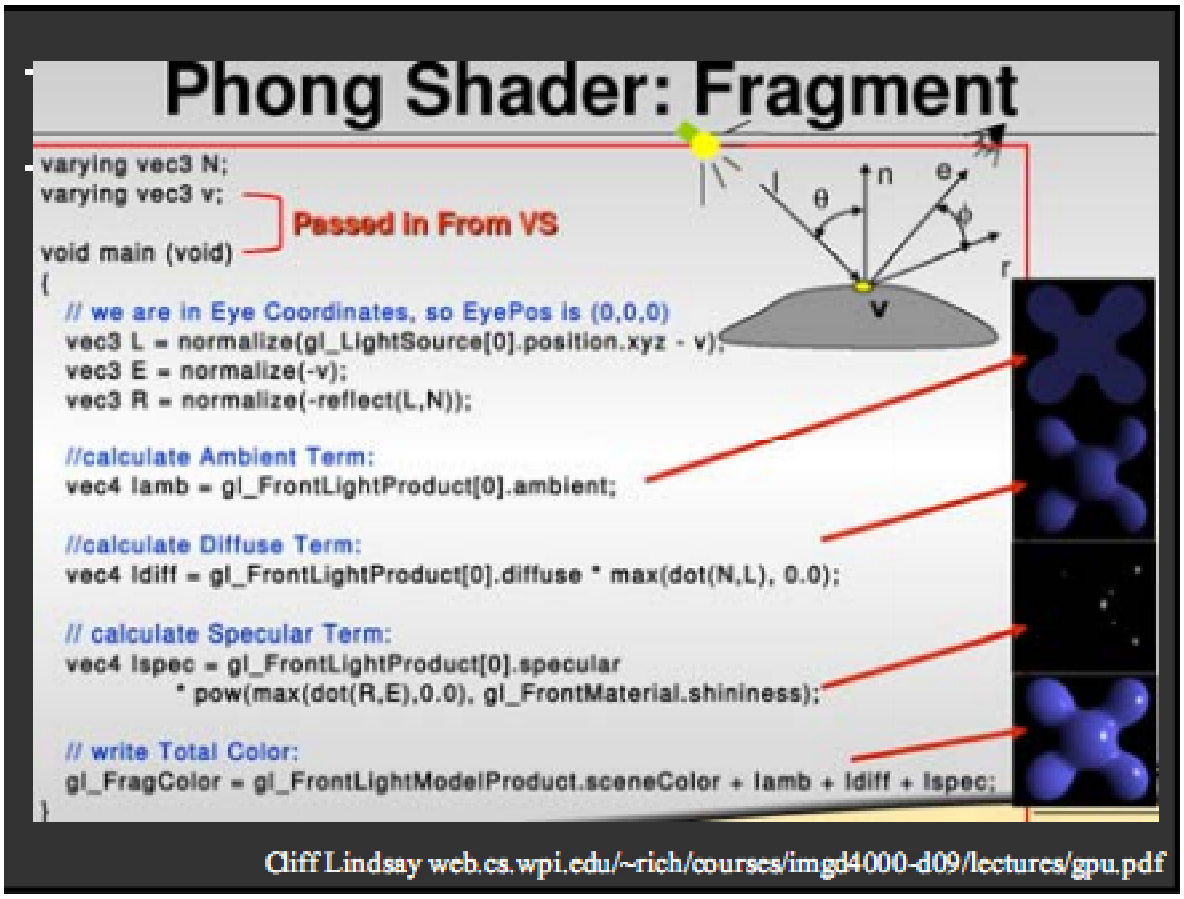
This Shader Does
- Gives eye space location for v
- Transform Surface Normal
- Transform Vertex Location

```
varying vec3 N;
varying vec3 v;

void main(void)
{
   v = vec3(gl_ModelViewMatrix * gl_Vertex);
   N = normalize(gl_NormalMatrix * gl_Normal);

   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Created For Use Within Frag Shader

(Update OpenGL Built-in Variable for Vertex Position)

Cliff Lindsay web.cs.wpi.edu/~rich/courses/imgd4000-d09/lectures/gpu.pdf



Phong Shader: Fragment

```
varying vec3 N;
varying vec3 v;

void main (void)
{
  // we are in Eye Coordinates, so EyePos is (0,0,0)
  vec3 L = normalize(gl_LightSource[0].position.xyz - v);
  vec3 E = normalize(-v);
  vec3 R = normalize(-reflect(L,N));

  //calculate Ambient Term:
  vec4 Iamb = gl_FrontLightProduct[0].ambient;

  //calculate Diffuse Term:
  vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);

  // calculate Specular Term:
  vec4 Ispec = gl_FrontLightProduct[0].specular
               * pow(max(dot(R,E),0.0), gl_FrontMaterial.shininess);

  // write Total Color:
  gl_FragColor = gl_FrontLightModelProduct.sceneColor + Iamb + Idiff + Ispec;
}
```

Passed In From VS

Cliff Lindsay web.cs.wpi.edu/~rich/courses/imgd4000-d09/lectures/gpu.pdf