Rendering on the GPU

Tom Fili
Agenda

- Global Illumination using Radiosity
- Ray Tracing
- Global Illumination using Rasterization
- Photon Mapping
- Rendering with CUDA
Global Illumination using Radiosity

*Global Illumination using Progressive Refinement Radiosity* by Greg Coombe and Mark Harris (GPU GEMS 2: Chapter 39)

The radiosity energy is stored in texels, and fragment programs are used to do computation.
Global Illumination using Radiosity

- It breaks the scene into many small elements and calculates how much energy is transferred between the elements.

\[ F_{i\rightarrow j} = \frac{\cos \theta_i \cos \theta_j}{\pi r^2} V(i, j) \]

- Function of the distance and relative orientation.
- V is 0 if objects are occluded, 1 if they are fully visible.
Global Illumination using Radiosity

- Only works if objects are very small.
- To increase speed we use larger areas and approximate them with oriented discs.

\[ F_{i \rightarrow \Delta j} = \Delta j \frac{\cos \theta_i \cos \theta_j}{\pi r^2 + \Delta j} V(i, \Delta j) \]
Global Illumination using Radiosity

- The classic radiosity algorithm solve a large system of linear equations composed of the pairwise form factors.
- These equations describe the radiosity of an element as a function of the energy from every other element, weighted by their form factors and the element's reflectance, $r$.
- The classical linear system requires $O(N^2)$ storage, which is prohibitive for large scenes.
Progressive Refinement

Instead we use Progressive refinement.

Each element in the scene maintains two energy values: an *accumulated* energy value and *residual* (or "unshot") energy.

All energy values are set to 0 except the residual energy of light sources.
Progressive Refinement

To implement this on the GPU we use 2 textures (accumulated and residual) for each element.

We render from the POV of the shooter.

Then we iterate over receiving elements and test for visibility.

We then draw each visible element into the frame buffer and use a fragment program to compute the form factor.
Progressive Refinement

initialize shooter residual \( E \)
while not converged
{
    render scene from POV of shooter
    for each receiving element
    {
        if element is visible
        {
            compute form factor \( FF \)
            \( DE = r \times FF \times E \)
            add \( DE \) to residual texture
            add \( DE \) to radiosity texture
        }
    }
    shooter's residual \( E = 0 \)
    compute next shooter
}
Visibility

The visibility term of the form factor equation is usually computed using a hemicube.

- The scene is rendered onto the five faces of a cube map, which is then used to test visibility.

Instead, we can avoid rendering the scene five times by using a vertex program to project the vertices onto a hemisphere.

- The *hemispherical projection*, also known as a *stereographic projection*, allows us to compute the visibility in only one rendering pass.
- The objects must be tesselated at a higher level to conform to the hemisphere.
void hemiwarp(float4 Position: POSITION, // World Pos
uniform half4x4 ModelView, // Modelview Matrix
uniform half2 NearFar, // Near/Far planes
out float4 ProjPos: POSITION) // Projected Pos
{
    // transform the geometry to camera space
    half4 mpos = mul(ModelView, Position);
    // project to a point on a unit hemisphere
    half3 hemi_pt = normalize(mpos.xyz);
    // Compute (f-n), but let the hardware divide z by this
    // in the w component (so premultiply x and y)
    half f_minus_n = NearFar.y - NearFar.x;
    ProjPos.xy = hemi_pt.xy * f_minus_n;
    ProjPos.z = (-2.0 * mpos.z - NearFar.y - NearFar.x);
    ProjPos.w = f_minus_n;
}

bool Visible(half3 ProjPos, // camera-space pos
uniform fixed3 RecvID, // ID of receiver
sampler2D HemiItemBuffer )
{
    // Project the texel element onto the hemisphere
    half3 proj = normalize(ProjPos);
    // Vector is in [-1,1], scale to [0..1] for texture lookup
    proj.xy = proj.xy * 0.5 + 0.5;
    // Look up projected point in hemisphere item buffer
    fixed3 xtex = tex2D(HemiItemBuffer, proj.xy);
    // Compare the value in item buffer to the // ID of the fragment
    return all(xtex == RecvID);
}

Projection Vertex Program Visibility Test Fragment Program
Form Factor Computation

\[
F_{i \rightarrow j} = \Delta j \frac{\cos \theta_i \cos \theta_j}{\pi r^2 + \Delta j}
\]

```cpp
half3 FormFactorEnergy(
    half3 RecvPos,  // world-space position of this element
    uniform half3 ShootPos,  // world-space position of shooter
    half3 RecvNormal,  // world-space normal of this element

    uniform half3 ShootNormal,  // world-space normal of shooter
    uniform half3 ShootEnergy,  // energy from shooter residual texture
    uniform half ShootDArea,  // the delta area of the shooter

    uniform fixed3 RecvColor )  // the reflectivity of this element
{
    // a normalized vector from shooter to receiver
    half3 r = ShootPos - RecvPos;
    half distance2 = dot(r, r);
    r = normalize(r);

    // the angles of the receiver and the shooter from r
    half cosi = dot(RecvNormal, r);
    half cosj = -dot(ShootNormal, r);

    // compute the disc approximation form factor
    const half pi = 3.1415926535;
    half Fij = max(cosi * cosj, 0) / (pi * distance2 + ShootDArea);
    Fij *= Visible();  // returns visibility as 0 or 1

    // Modulate shooter's energy by the receiver's reflectivity
    // and the area of the shooter.
    half3 delta = ShootEnergy * RecvColor * ShootDArea * Fij;

    return delta;
}
```
Adaptive Subdivision

- We create smaller elements along areas that need more detail (e.g., Shadow edges).
- Reuse same algorithms except we compute visibility on the leaf nodes.
- We evaluate a gradient of the radiosity and if it's above a certain threshold we discard it.
- If we discard enough fragments then we subdivide the current node.
Performance

Can render a 10,000 element version of Cornell Box at 2 fps.

To get this we need to make some optimizations

- Use occlusion queries in visibility pass
- Shoot rays a lower resolution than the texture.
- Batch together multiple shooters.
- Use lower resolution textures to compute indirect lighting. Compute direct lighting separately and add in later.
Global Illumination using Radiosity
Ray Tracing


Shows how to design a streaming ray tracer that is designed to be run on parallel graphics hardware.
Streaming Ray Tracer

- Multi-pass algorithm
- Divides the scene into a uniform grid, which is represented by a 3D texture.
- Split the operation into 4 kernels executed as fragment programs.
- Uses the stencil buffer to keep track of which pass a ray is on.
Storage

- Grid Texture
  - 3D Texture
- Triangle List
  - 1D Texture
  - Single Channel
- Triangle-Vertex List
  - 1D Texture
  - 3 Channel (RGB)
Eye Ray Generator

Simplest of the kernels.

Given the camera parameters it generates a ray for each screen pixel.

A fragment program is invoked for each pixel which generates a ray.

Also tests rays against the scene’s bounding volume and terminates the ones outside the volume.
Traverser

- For each ray it steps through the grid.
- A pass is required for each step through the grid.
- If a voxel contains triangles, then the ray is marked to run the intersection kernel on triangles in that voxel.
- If not, then it continues stepping through the grid.
Intersector

- Tests the ray for intersection with all triangles within a voxel.
- A pass is required for each ray-triangle intersection test.
- If an intersection occurs then the ray is marked for execution in the shading stage.
- If not the ray continues in the traversal stage.
Intersection Shader (Pseudo)Code

float4 IntersectTriangle(float3 ro, float3 rd, int list pos, float4 h )
{
    float tri id = texture( list pos, tri list );
    float3 v0 = texture( tri id, v0 );
    float3 v1 = texture( tri id, v1 );
    float3 v2 = texture( tri id, v2 );
    float3 edge1 = v1 - v0;
    float3 edge2 = v2 - v0;
    float3 pvec = Cross( rd, edge2 );
    float det = Dot( edge1, pvec );
    float inv det = 1/det;
    float3 tvec = ro - v0;
    float u = Dot( tvec, pvec ) * inv det;
    float3 qvec = Cross( tvec, edge1 );
    float v = Dot( rd, qvec ) * inv det;
    float t = Dot( edge2, qvec ) * inv det;
    bool validhit = select( u >= 0.0f, true, false );
    validhit = select( v >= 0, validhit, false );
    validhit = select( u+v <= 1, validhit, false );
    validhit = select( t < h[0], validhit, false );
    validhit = select( t >= 0, validhit, false );
    t = select( validhit, t, h[0] );
    u = select( validhit, u, h[1] );
    v = select( validhit, v, h[2] );
    float id = select( validhit, tri id, h[3] );

    return float4( ft, u, v, idg );
}
Shader

- This adds the shading for the pixel.
- It also generates new rays and marks them for processing in a future rendering pass.
- Also gives new rays a weight so the color can be simply added.
Global Illumination using Rasterization

- High-Quality Global Illumination Rendering Using Rasterization by Toshiya Hachisuka (GPU GEMS 2: Chapter 38)
- Instead of adapting global illumination algorithms to the GPU, it makes use of the GPU’s rasterization hardware.
Two-pass methods

- First pass uses photon mapping or radiosity to compute a rough approximation of illumination.
- In the second pass, the first pass result is refined and rendered.
- The most common way to use the first pass is as a source of indirect illumination.
The process of final gathering is used to compute the amount of indirect light by shooting a large amount of rays. This can be the bottleneck.

Sampling and interpolation is used to speed it up. This can lead to rendering artifacts.
Final Gathering via Rasterization

- Precomputes directions and traces all of the rays at once using rasterization.
- This is done with a parallel projection of the scene along the current direction or the *global ray direction*. 

![Set of Rays in Conventional Final Gathering](image1)

![Set of Rays in Our Method](image2)
Depth Peeling

- Each depth layer is a subsection of the scene.
- Shoot a ray in the opposite direction of the global ray direction.
- This can be achieved by rendering multiple times using a greater than depth test.
Depth Peeling

- Step through the depth layers, computing the indirect illumination until no fragments are rendered.
- Repeat with another global ray direction until the number of samplings is sufficient.
This method only computes indirect illumination. The first rendering pass can be done with any CPU or GPU method that computes the irradiance distribution.
- They suggest Grid Photon Mapping.
We use this in the final gathering pass.
Direct illumination must be computed with a real-time shadowing technique.
- They suggest shadow mapping and stencil shadows.
Direct and indirect illumination are summed before the final rendering.
Performance

It's hard to compare performance because the algorithms are very different.

Performance is similar to CPU based sampling/interpolation methods.

Performance is much faster than a CPU method that would sample all pixels.
Global Illumination using
Rasterization
Photon Mapping

Photon Tracing

- Each pass of the photon tracing reads from the previous frame.
- At each surface interaction a photon is written to the texture and another is emitted.
- The initial frame has the photons on the light sources and their random directions.
- The direction of each photon bounce are computed from a random number texture.
Photon Map Data Structure

The original photon map algorithm uses a balanced k-d tree for locating the nearest photons.

This structure makes it possible to quickly locate the nearest photons at any point.

It requires random access writes to construct efficiently.

This can be slow on the GPU.

Instead we use a uniform grid for storing the photons.

- Bitonic Merge Sort – Fragment program
- Stencil Routing – Vertex program
We can index the photons by grid cell and sort them by cell.

Then find the index of the first photon in each cell using a binary search.

Bitonic Merge Sort is a parallel sorting algorithm that takes $O(\log^2 n)$ steps.

It can be implemented as a fragment program with each rendering pass being one stage of the sort.
Bitonic Merge Sort

3
7
4
8
6
2
1
5

8x monotonic lists: (3) (7) (4) (8) (6) (2) (1) (5)
4x bitonic lists: (3,7) (4,8) (6,2) (1,5)
Bitonic Merge Sort

Sort the bitonic lists
Bitonic Merge Sort

4x monotonic lists:  (3,7) (8,4) (2,6) (5,1)
2x bitonic lists:  (3,7,8,4) (2,6,5,1)
Bitonic Merge Sort

Sort the bitonic lists
Bitonic Merge Sort

Sort the bitonic lists
Sort the bitonic lists
Bitonic Merge Sort

2x monotonic lists:  (3,4,7,8) (6,5,2,1)
1x bitonic list: (3,4,7,8, 6,5,2,1)
Sort the bitonic list
Sort the bitonic list
Sort the bitonic list
Bitonic Merge Sort

<table>
<thead>
<tr>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Sort the bitonic list
Bitonic Merge Sort

<table>
<thead>
<tr>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Sort the bitonic list
Bitonic Merge Sort

<table>
<thead>
<tr>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Done!
Binary search can be used to locate the contiguous block of photons occupying a given grid cell.

We compute an array of the indices of the first photon in every cell.

- If no photon is found for a cell, the first photon in the next grid cell is located.

The simple fragment program implementation of binary search requires $O(\log n)$ photon lookups.

All of the photon lookups can be unrolled into a single rendering pass.
Fragment Program Method

Uniform Grid: 0 0 3 3 3 3 6 6 8 9 9 ...

Photon List: 1 2 7 5 4 6 0 3 8 9 ...

Photon Position: ...

Photon Power: ...

Photon Direction: ...
Vertex Program Method

Since the Bitonic Merge Sort can add many rendering passes, it may not be useful for interactive rendering.

You can use a Stencil Routing to route photons to each grid cell in one rendering pass.

Each grid cell covers a $m \times m$ set of pixels.

Draw a point with a point size of $m$ and then use the stencil buffer to send the photon to the correct fragment.
Vertex Program Method

(a) p0 → Vertex Program → p0

(b) p0 → Vertex Program → Stencil → p0

(c) p1 → Vertex Program → Stencil → p1 p0
Vertex Program Method

There are two drawbacks to this method:

- We must read from a photon texture which requires a readback.
- We allocate a fixed amount of memory so we must redistribute the power for cells with greater than $m^2$ photons and space is wasted if there is less.
Radiance Estimate

- We accumulate a radiance value based on predefined number of nearest photons.
- We search all photons in the cell.
  - If the photon is in the search range then we add it.
  - If not, then we ignore it unless we don’t have enough photons. Then we add it and expand the range.
Rendering

Use a stochastic ray tracer written using a fragment program to output a texture with all the hit points, normals, and colors for a given ray depth.

This texture is used as input to several additional fragment programs.

- One program computes the direct illumination using one or more shadow rays to estimate the visibility of the light sources.
- One that invokes the ray tracer to compute reflections and refractions.
- One to compute the radiance.
CUDA Rendering

All of these rendering techniques can be done with CUDA.

They are simpler to implement because you don’t have to store everything in textures and you can use shared memory.
CUDA Rendering Demo
References

- GPU Gems 2 – Chapters 38 & 39
- Jon Olick Video
  - [http://www.youtube.com/watch?v=VpEpAFGpIvnI](http://www.youtube.com/watch?v=VpEpAFGpIvnI)
- CUDA Voxel Demo