Surveying Real-Time Rendering Algorithms Beyond Programmable Shading

David Luebke
NVIDIA Research
The continuum “Beyond Programmable Shading”

“Just” programmable shading: DX, OGL
The continuum “Beyond Programmable Shading”

“Just” programmable shading: DX, OGL

“Pure” compute-based graphics: CUDA, OptiX
The continuum “Beyond Programmable Shading”

Interesting middle ground!

“Just” programmable shading: DX, OGL

“Pure” compute-based graphics: CUDA, OptiX
My focus will be on pointing out the patterns rather than explaining individual algorithms
I added Stochastic Transparency the morning of the SIGGRAPH course because it (a) fits the pattern of render-analyze-render, though it doesn’t require CUDA/DirectCompute to do so, and (b) it tackles directly one of the 5 major challenges that Johan highlighted.
Stochastic Transparency

Eric Enderton, Erik Sintorn, Peter Shirley, David Luebke, *2010 ACM Symposium on Interactive 3D Graphics & Games*
Stochastic Transparency

Screen-door transparency

The problem: order-independent transparency.
The proposed solution: stochastic transparency.
Key insight: turn transparent stuff into opaque stuff that only exists at some samples
This is basically screen door transparency
Based on one of the oldest and most reviled transparency algorithms: screen door transparency. (Image from Mark Kilgard SGI sample c. 1996.)
But if you do it multi-sampled, it starts to make sense. GPUs (starting with Reality Engine) have a mode for this, called Alpha to Coverage.
Alpha2coverage uses same masks for all pixels, easy to show this gives wrong result.
Solution: use random masks.

Tada! Unbiased estimator. No extra passes, no extra memory. And importantly, all cases are unified into a single algorithm → code is simple.
You can reduce noise by increasing samples. “The Elegance of Brute Force” (Kurt Akeley).
These are stratified
You can reduce noise by increasing samples. “The Elegance of Brute Force” (Kurt Akeley).
These are stratified
You can reduce noise by increasing samples. “The Elegance of Brute Force” (Kurt Akeley).

These are stratified
You can reduce noise by increasing samples. “The Elegance of Brute Force” (Kurt Akeley).

These are stratified.
You can reduce noise by increasing samples. “The Elegance of Brute Force” (Kurt Akeley).
These are stratified
You can reduce noise by increasing samples. “The Elegance of Brute Force” (Kurt Akeley).
These are stratified.
You can reduce noise by increasing samples. “The Elegance of Brute Force” (Kurt Akeley).

These are stratified.
Or instead of more samples you can use more algorithm:

render a pass, do some analysis (in this case, of depth and total alpha/pixel), and render another pass (fanciest algorithm requires 3 passes, see the paper)

I claimed “no extra passes, no extra memory”, now we’ve got a couple extra passes but it is still a fixed cost, unlike depth peeling or the A-buffer technique I’ll present later (but remember, those techniques get the right picture, no noise, assuming you take enough time/memory)
Engineer art, but you get the idea. Mix hair, translucent cloth, billboards, etc robustly, all with shadow maps.
Key insight: turn transparent geometry into opaque geometry. Now you can render everything you could have rendered non-transparently.

With enough samples, noise starts to go away (especially when animated)

Thus we make OIT into a brute-force algorithm, which Moore’s Law can eventually solve for us.
Sample Distribution Shadow Maps

Andrew Lauritzen, Marco Salvi, Aaron Lefohn,
*Advances in Real-Time Rendering in 3D Graphics and Games, SIGGRAPH 2010 Courses*


Thanks to Andrew for these slides!
The disconnect between light-space sampling and camera-space sampling means you can easily get serious aliasing even with reasonable tuning of parameters.

This is a cascaded shadow map (parallel-split shadow maps)

Significant aliasing problems both near the camera and far away. The PSSM tuneable parameter has been tweaked for this view to provide a reasonable solution.

Significantly worse results can occur in practice!
This is a cascaded shadow map (parallel-split shadow maps), which splits the view frustum along the Z axis into several segments. This technique helps but doesn’t eliminate the aliasing.

Significant aliasing problems both near the camera and far away. The PSSM tuneable parameter has been tweaked for this view to provide a reasonable solution. Significantly worse results can occur in practice!
This is what the camera frustum looks like from light space.
A simple frustum partition bounding box results in lots of wasted space. Large parts of the frustum are empty or occluded (this is typical).
Sample distribution shadow maps address both of these issues by analyzing the actual distribution of light space samples required to shadow the current view.

- Analyze the shadow sample distribution
  - Find tight Z min/max
  - Partition Z range logarithmically – fully automatic!

- Compute tight light-space bounds per partition
  - Axis-aligned bounding box on view samples
  - Greatly increases useful shadow resolution
Near/far has been tightened to cover only visible samples.
Note that near being tightened is actually the most important.
Tight light space bounds take advantage of camera space occlusion and produce excellent bounds for each partition.
Sample Distribution Shadow Maps

Scene from Left 4 Dead 2, courtesy of Valve Corporation

Results for SDSM – subpixel accuracy nearly everywhere
Flip back and forth to see differences.
Render to Data Structure

- Alias-Free Shadow Maps
- Append-Consume OIT
- GPU Progressive Photon Mapping
The slides and exact algorithm described here are from:

Erik Sintorn, Elmar Eisemann, Ulf Assarsson.


See related publications (Irregular Z-buffer and Alias-Free Shadow Maps) in speaker notes

The slides and exact algorithm described here are from:

Erik Sintorn, Elmar Eisemann, Ulf Assarsson.

**Sample Based Visibility for Soft Shadows using Alias-free Shadow Maps.**


That paper introduces a CUDA implementation of hard shadows, which I describe here, and then they go on to extend to soft shadows.

See also the original alias-free shadow maps paper by Aila and Laine, and the more hardware-oriented simultaneous work by Johnson et al:

**Alias-Free Shadow Maps**


**The Irregular Z-Buffer: Hardware Acceleration for Irregular Data Structures**

Gregory S. Johnson, Juhyun Lee, Christopher A. Burns and William R. Mark  
ACM Transactions on Graphics, October 2005  
Shadow Maps are a common technique for rendering shadows that was introduced by Williams in 1978. It has proven very efficient to implement on the GPU but can suffer severely from aliasing.

The two images here show a simple scene from the point of view of the Camera and from the light source.

The first step here is to render the depth of each fragment into a shadow map, so that when the whole scene is rendered, each texel contains the depth of the fragment closest to the light.

Then, when rendering the scene from the camera’s viewpoint, for each fragment we transform and project the fragments’ world coordinates onto the shadow map, and consider the fragment in shadow if its depth is larger than that in the map.

As I’m trying to show in this image, several of the screenspace fragments will end up in the same shadowmap texel, no matter how high a resolution we have on the shadow map, resulting in,

This sort of shadow aliasing.
Now I will explain how our alias free shadow maps algorithm works, I will start with an overview and then go into a little more detail.

After that I will talk about how this can be extended to render soft shadows.

First of all, for opaque objects, we only need to find shadow information for the fragments closest to the eye. So,

In a first step, we render the world coordinates of these fragments into a buffer.

We then, using CUDA, transform and project this array of view-samples into lightspace and reorder them into a compact array of lists, where each list contains the view-samples that would end up in a shadow map texel.
Then, we render the geometry from the light.

For each each fragment generated by a triangle, we look up the list of view-samples and check, for each sample, if the triangle occludes that sample.

The result for each sample in the list is written as a binary value to the corresponding bit in the output rendertarget.

With a current maximum of eight rendertargets, we are limited to lists with 1024 samples per pass, but in practice we have found 128 samples to be sufficient in almost all cases.
Then we just have to draw the actual shadows.

A fullscreen pass is performed “over” the image rendered from the cameras viewpoint, and for each pixel, we can look up whether the view-sample is shadowed or not, and draw shadow accordingly.
Fantastic example of render-to-alternate-data-structure pattern, in this case an A-buffer of fragments implemented as a linked list threaded through a DirectCompute UAV.

Thanks to Mike Houston and Justin Hensley who got me these slides, which I have adapted slightly to simplify the presentation. These were presented in greater detail in yesterday’s course.
The Problem with Transparency

With Sorting

Sorting is hard!

Skeleton hidden

No Sorting

Arm appears in front of body
Solution: Sort Fragments Per Pixel

• Store linked list of fragments per pixel
  – Each list record stores color, alpha, depth
  – Thread linked lists through DirectCompute UAV

• A “resolve” pass sorts then blends fragments
## Create Linked List

### Start Offset Buffer

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Fragment and Link Buffer

- The table represents a linked list where each cell contains a fragment or link.
- The buffer is organized in rows and columns, with each cell indicating the link or fragment.
- The diagram illustrates the visual representation of the linked list.
Create Linked List

-1 -1 -1 -1 -1 -1
-1 0 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1

0.87
-1
Create Linked List

-1 -1 -1 -1 -1 -1
-1  0 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1
-1 -1 -1  1  2 -1
-1 -1 -1 -1 -1 -1

0.87 0.89 0.90
-1 -1 -1
Create Linked List

-1 -1 -1 -1 -1 -1

1 5 4 -1 -1 -1

-1 -1 -1 -1 -1 -1

-1 -1 -1 -1 -1 -1

-1 -1 -1 1 2 -1

-1 -1 -1 -1 -1 -1

0.87 0.89 0.90 0.65 0.65 0.71

-1 -1 -1 0 -1 3

-1 -1 -1 -1 -1 -1
### Render Fragments

**Walk the list and store in temporary array**

<table>
<thead>
<tr>
<th></th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0.87</th>
<th>0.89</th>
<th>0.90</th>
<th>0.65</th>
<th>0.65</th>
<th>0.71</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>3</td>
</tr>
</tbody>
</table>

- 0.71
- 0.65
- 0.87
Render Fragments

Sort temporary array
Blend colors and write out

-1  -1  -1  -1  -1  -1
-1   5   4  -1  -1  -1
-1  -1  -1  -1  -1  -1
-1  -1  -1  -1  -1  -1
-1  -1  -1   1   2  -1
-1  -1  -1  -1  -1  -1

0.87  0.89  0.90  0.65  0.65  0.71
-1  -1  -1   0  -1   3

0.65  0.71  0.87
This is very cool still-unpublished work by Toshiya Hachisuka & Henrik Wann Jensen, UC San Diego. Many thanks to them for permission to give an early preview of their work to the SIGGRAPH audience!
The focus of this application is to implement a new global illumination algorithm called progressive photon mapping. The idea is to accumulate multiple small photon maps such that we can obtain the correct, unbiased solution in the limit. Each iteration of progressive photon mapping performs eye ray tracing, which finds eye ray hit points, and photon tracing, which accumulates photons into the eye ray hit points.
The main issue of implementing progressive photon mapping on GPUs is construction of photon maps. Since progressive photon mapping needs to construct a new photon map from scratch in each pass, we need to do this construction fast on GPUs to achieve an efficient implementation.

As a solution to this issue, Hachisuka developed a new hashing scheme that efficiently maps on data-parallel computation on GPUs. The idea is in fact very simple. Whenever a hash collision occurs, instead of keeping a list of photons as in CPU implementations, we just keep a single photon stochastically. For example, suppose that we have the photons 1, 2, and 4 that are mapped to the same hash index. A standard hashing constructs a list of photons, whereas the stochastic hashing only keeps one of the photons that are mapped to the same index and scale its flux based on the selection probability.

This simple modification enables efficient data-parallel construction of photon maps while maintaining the algorithm convergent. Moreover, since we are guaranteed to have a single photon per cell at most regardless of hash collisions, it ensures uniform work distribution at retrieval time. Efficient range queries necessary for photon density estimation can be done in the same way.
In a practical implementation, we use two buffers to store data of a stochastic-hashed grid: one that stores indices to the photon data and one that stores the number of collisions.

After tracing photons, we compute hash values of photon positions, and perform scattered writes using hashed values as destinations. The scattered writes to the hash entries just simply overwrites without caring any hash collision, and the scattered writes to the hash counters uses atomic increment operations to count the number of hash collisions.

When we perform rendering using photons, we first use the same hash function to compute the hashed-value of each eye ray hit positions. We then read photon data from the hashed entries using the hashed value, and multiply the corresponding number of collisions to compensate hash collisions. Since there is no synchronization step (except for atomic increments?), every operation can be done in a data-parallel fashion using modern programming languages on GPUs.
Here we compare path tracing and progressive photon mapping on GPUs (results on Radeon HD 4850). The top row shows the results using path tracing and the bottom row shows the results using progressive photon mapping. Both algorithms use the same ray tracing core. Since these scenes include complex light paths, results of path tracing are completely unacceptable even after 10min. In the same rendering time, progressive photon mapping can robustly deal with those paths. Note that, even with more sophisticated algorithms such as bidirectional path tracing, these scenes are very difficult to render, and progressive photon mapping is currently considered to be the only algorithm that can handle these scenes.
Rasterization as spatialized compute

- Image Space Photon Mapping
- Ambient Occlusion Volumes
- Stochastic Rasterization
I highlighted this last year as a good example of hybrid graphics+compute; I bring it up again more briefly today as a nice example of the rasterization-for-spatial-invocation-of-compute pattern.
Image Space Photon Mapping

First bounce: *bounce map*

Final bounce: *photon volumes*
ISPM Radiance Estimate

• Traditional photon mapping: *gather*
  • Per pixel
  • $k$-NN search in $k$-d tree
  • World-space (3D)

• Image space photon mapping: *scatter*
  • Per photon
  • Hardware rasterization using photon volumes
  • Image space (2D)
Similar to shadow volumes, fog volumes, light volumes

There’s a family of nice technique called instant radiosity that scatter “virtual point lights”. Those are samples of OUTGOING radiance that affect the entire scene. Photons are INCOMING radiance and only affect a small region (the photon volume).

There are other techniques that perform 2D splatting of illumination—but that assumes that everything in the scene is perfectly Lambertian.
Making photon volumes too small gives you spotty coverage
Making photon volumes too big gives you smooth effects but blows out the fine details, e.g. within the caustic.
Adaptively sized photon volumes capture fine detail in areas with many photons (high probability) while using large photon volumes to capture smooth indirect diffuse effects (low probability)
Two independent papers simultaneously developed the following idea and presented them at back-to-back conferences this year.

Both papers use the idea of ambient occlusion volumes, though Laine/Karras paper doesn’t call them that. They differ in the actual AO computation but that doesn’t matter for my purposes.

My slides are mostly taken from Samuli’s presentation at EGSR.
Ambient Occlusion in Theory

- Occlusion of incoming ambient light

Light from these directions reaches the surface

Light from these directions does not reach the surface
What It Looks Like
Ray Casts for Ambient Occlusion

- Cast a number of rays from the point to be shaded
- Determine occlusion distance, apply falloff, sum
Occlusion of a Triangle

- Combine four precomputed bitwise \textit{occlusion masks}
Rasterize Triangle Regions of Influence
Bounding the Region of Influence

Multiple choices for bounding volume
• Large triangles: construct hexagonal prism
• Small triangles: construct hemispherical billboard
This slide is from Morgan’s HPG paper and shows the actual volumes he uses.
One last brief look at a technique that uses rasterization to invoke computation only on regions that need it. Unlike the others this one does not assume a deferred shading pipeline.

http://research.nvidia.com/publication/real-time-stochastic-rasterization-conventional-gpu-architectures
Basic idea is to rasterize the convex hull of a motion-blurred triangle, and perform exact intersection between the ray and the time-continuous triangle in the fragment shader.
Worth noting that most but not all of the algorithms presented here rely on deferred shading – a definite pattern in Beyond Programmable Shading!
For example, the diffusion-based in Metro 2033, based on Lefohn’s work, fits pattern 1


Sungkil Lee’s real-time lens blur effects paper from Tuesday fits pattern 2

Real-Time Lens Blur Effects and Focus Control, Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. SIGGRAPH 2010
Acknowledgements

Many thanks for use of their slides & images:

Samuli Laine  Tero Karras  Morgan McGuire
Eric Enderton  Justin Hensley  Jason Yang
Mike Houston  Aaron Lefohn  Andrew Lauritzen
Sungkil Lee  Toshiya Hachisuka  Metro 2033
THANK YOU

dluebke@nvidia.com