Particle Shadows & Cache-Efficient Post-Processing

Louis Bavoil & Jon Jansen
Developer Technology, NVIDIA
Agenda

1. Particle Shadows

2. Cache-Efficient Post-Processing
Part 1:
Particle Shadows
Particle Shadows

Assumption
Each particle transmits $(1-\alpha)$ of its incoming light intensity

Definition
Shadow cast by particles along a given light-ray segment
$= \text{Transmittance}$
$= (1-a_0)(1-a_1)\ldots(1-a_{N-1})$
“External Shadows”

Idea

Blend \((1-a_0)(1-a_1) \cdots (1-a_{N-1})\) to a \texttt{R8\_UNORM}

“Translucency Map” [Crytek 2011]

Pros

1. Compact memory footprint
2. Map rendered in one pass, order-independent
3. Fast shadow projection: \texttt{R8\_UNORM} bilinear fetch
Limitation: No self-shadowing

Screenshot from [Crytek 2011]
Wanted: Particle Self-Shadows

[Green 2012]  
[Jansen 2010]
Volumetric Self-Shadowing

Large body of research work

- Deep Shadow Maps [Lokovic 2000]
- Opacity Shadow Maps [Kim 2001] [NVIDIA 2005]
- Deep Opacity Maps [Yuksel 2008]
- Adaptive Volumetric Shadow Maps [Salvi 2010]
- **Fourier Opacity Mapping (FOM)** [Jansen 2010] (*)
- Extinction Transmittance Maps [Gautron 2011]
- Half-Angle Slicing [Green 2012] [Kniss 2003]

(*) Shipped in “Batman: Arkham Asylum” (PC)
Wanted: Scalability

Build on shadow mapping
   Extend existing opaque-shadow systems
   Support large scenes, multiple lights

Support large shadow depth ranges
   Do not get limited by MRTs
Wanted: Lots of Detail

Goal: reveal structural detail
Our Solution:
Particle Shadow Mapping
“Particle Shadow Map”

PSM = 3D Texture

Mapped into light space
xy/uv planes are always perpendicular to light rays

Store shadow per voxel
(transmittance through light ray up to that voxel)
PSM Algorithm

STEP 1: Clear PSM to 1.f everywhere
STEP 2: Voxelize particle transmittances to PSM
STEP 3: Propagate transmittances along rays through PSM
STEP 4: Sample transmittance from PSM when rendering scene
PSM Layout

3D Texture representing voxelized local transmittances
Storing FP32 transmittances would be overkill

voxels along a light ray
PSM Layout

Can pack 4 x 8-bit values into one 4x8_UNORM

e.g. $256^3$ PSM stored as 256x256x64 4x8_UNORM texture
Step 1: Clear PSM

Clear 3D Texture to 1.0 (no shadow)
Step 2: Voxelize Transmittances

light-facing particle transmittance = 0.5
Step 2: Voxelize Transmittances

**Geometry Shader**

with \([\text{maxvertexcount}(4)]\)

outputs \(\text{SV\_RenderTargetArrayIndex}\) *

* Works because shadow casters are particles. Hence the name “Particle Shadow Mapping”.

![Diagram showing layers and light Z with local transmittance and outputs SV_RenderTargetArrayIndex](image-url)
Step 2: Voxelize Transmittances

**GS** assigns particle to layer=2, channel=G

**PS** writes (1.f-alpha) to G, and 1.f to R,B,A

**OM** does **Multiplicative Blending**
Step 2: Voxelize Transmittances

light-facing particle transmittance = 0.2

Local Transmittance

layer 0 | layer 1 | layer 2 | layer 3 | light Z
Step 2: Voxelize Transmittances

Local Transmittance

1.0

layer 0  layer 1  layer 2  layer 3  light Z

0.5  0.2
Step 3: Propagate Transmittances

**Compute Shader**
with one thread per light ray
runs in-place, so space efficient
Step 4: Sample from PSM

Output from STEP 3

= Particle Shadow Map
= Per-Voxel Shadows

Shadow Evaluation

Cannot use a trilinear texture fetch due to RGBA packing
So perform 2 bilinear fetches & lerp between slices
PSM Practicality

Obvious objection to PSM is space complexity e.g.
- $256 \times 256 \times 256 \times 8 \text{bits} = 16 \text{MB} (= 0.78\% \text{ of 2GB FB})$
- $512 \times 512 \times 512 \times 8 \text{bits} = 128 \text{MB} (= 6.25\% \text{ of 2GB FB})$

Arguably
- $256^3$ is feasible right now
- $512^2 \times 256 (= 64 \text{MB})$ could work as ‘extreme’ setting
## Comparison to External Shadows

<table>
<thead>
<tr>
<th></th>
<th>External Shadows [Crytek 2011]</th>
<th>PSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Render shadow map</td>
<td>RT=1x8bits</td>
<td>RT=1x32bits</td>
</tr>
<tr>
<td>Propagation</td>
<td>n/a</td>
<td>O(w x h x d)</td>
</tr>
<tr>
<td>Sample shadow map</td>
<td>1 texture lookup/sample</td>
<td>2 texture lookups/sample</td>
</tr>
<tr>
<td>Space complexity</td>
<td>O(w x h)</td>
<td>O(w x h x d)</td>
</tr>
</tbody>
</table>
## Comparison to Prior Art

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Render to shadow map</td>
<td>MRT=d×8bits</td>
<td>MRT=1×8bits</td>
<td>MRT=d×16bits</td>
<td>MRT=1×32bits</td>
</tr>
<tr>
<td>Render to shadow map RT changes</td>
<td>1</td>
<td>O(d)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Propagation</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>O(w × h × d)</td>
</tr>
<tr>
<td>Sample shadow map textures</td>
<td>O(d) fetches</td>
<td>1 fetches</td>
<td>O(d) fetches</td>
<td>2 fetches</td>
</tr>
<tr>
<td>Space complexity</td>
<td>O(w × h × d)</td>
<td>O(w × h)</td>
<td>O(w × h × d)</td>
<td>O(w × h × d)</td>
</tr>
</tbody>
</table>
PSM Performance

8K large particles
256^3 Particle Shadow Map

<table>
<thead>
<tr>
<th>PSM Generation</th>
<th>GPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM RT clear</td>
<td>0.01 ms</td>
</tr>
<tr>
<td>Render to PSM</td>
<td>0.23 ms</td>
</tr>
<tr>
<td>Propagation CS</td>
<td>0.33 ms</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>0.58 ms</strong></td>
</tr>
</tbody>
</table>

* Measured with D3D11 timestamp queries on GTX 680
Output of STEP 2:
Voxelized Local Transmittances
Coverage Optimization

**Goal:** in STEP 3, early exit for “empty light rays”

IDEA 1: slice 0 reserved for coverage

Does not work!
The additional rasterization into slice 0 doubles our fill workload, and therefore the execution time of the step
Coverage Optimization

Solution: Output particles to 2 D3D11 viewports

**GS output #0 → (Layer 0, Viewport 0)**
conservative coverage mask
[8x8 resolution]

**GS output #1 → (Layer >0, Viewport 1)**
entire PSM slice, as before
[256^2 resolution]
### Coverage Optimization

<table>
<thead>
<tr>
<th>PSM Generation</th>
<th>No Opt</th>
<th>Opt</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM RT clear</td>
<td>0.01 ms</td>
<td>0.01 ms</td>
<td>0%</td>
</tr>
<tr>
<td>Render to PSM</td>
<td>0.23 ms</td>
<td>0.26 ms</td>
<td>-11%</td>
</tr>
<tr>
<td>Propagation CS</td>
<td>0.33 ms</td>
<td>0.23 ms</td>
<td>43%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>0.58 ms</strong></td>
<td><strong>0.50 ms</strong></td>
<td><strong>16%</strong></td>
</tr>
</tbody>
</table>

256^3 PSM, 8K large particles, GTX 680 timings
Particle Lighting with DX11

When rendering particles to scene color buffer
Can render particles with DX11 tessellation
And fetch shadow maps in DS instead (faster than PS)

See Bitsquid’s GDC’12 talk on “Practical Particle Lighting” [Persson 2012]
And NVIDIA’s “Opacity Mapping” DX11 Sample [Jansen 2011]
PSM Wrap Up

“Particle Shadow Mapping” (PSM)
Specialized OSM technique for particles shadows
Scattering particles to 3D-texture slices

D3D11 features used
- GS for particle expansion + voxelization + coverage opt
- CS for transmittance propagation
- DS for fetching the PSM faster than in PS
DEMO
Part 2: Cache-Efficient Post-Processing
Large, Sparse & Jittered Filters

SSAO

SSDO [Ritschel 2009]

SSR [Crytek 2011]

Goal: Generic approach to speedup such filters without sacrificing quality
Large, Sparse & Jittered Filters

Kernel size up to 512x512 texels
Large, **Sparse** & Jittered Filters

e.g. 8 samples in $256^2$ area

Difficult to accelerate with a Compute Shader
Large, Sparse & **Jittered** Filters

Adjacent pixels have different sampling patterns
Large, Sparse & **Jittered** Filters

Adjacent pixels have different sampling patterns
Fixed Sampling Pattern

Example kernel
Fixed Sampling Pattern

Now, for a pair of adjacent pixels executed in lock step

For each sample, adjacent pixels fetching adjacent texels

⇒ Good spatial locality 😊
Random Sampling Pattern

Randomizing the texture coordinates per pixel...

For each sample, adjacent pixels fetching far-apart texels

→ Poor spatial locality 😞
Jittered Sampling Pattern

Jitter each of the 4 samples within $\frac{1}{4}$th of kernel area

For each sample, **adjacent pixels** fetching **sectored texels**

→ Better spatial locality

... but as kernel size increases, sector size increases too 😞
Previous Art

1. Jittered sampling patterns
   Jitter within one sector

2. Mixed-resolution inputs
   Use full-res texture for center tap
   Use low-res texture for sparse samples

3. MIP-mapped inputs [McGuire 2012]

Still, remaining per-pixel jittering hurts per-sample locality
Assumption:

Interleaved Sampling Patterns

NxN sampling patterns interleaved on screen

Typical sampling strategy for SSAO, SSDO, SSR, etc.

Per-pixel jitter seed fetched from a tiled “jitter texture”
Approach

“individually render lower resolution images corresponding to the regular grids, and to then interleave the samples obtained this way by hand”

[Keller 2001]
Approach

“individually render lower resolution images corresponding to the regular grids, and to then interleave the samples obtained this way by hand”

[Keller 2001]
Approach

“individually render lower resolution images corresponding to the regular grids, and to then interleave the samples obtained this way by hand”

[Keller 2001]
Approach

“individually render lower resolution images corresponding to the regular grids, and to then interleave the samples obtained this way by hand”

[Keller 2001]
Our Solution:

“Interleaved Rendering”

Render each sampling pattern *separately*, using *downsampled* input textures
STEP 1: Deinterleave Input

Full-Resolution Input Texture

Width = W
Height = H

Half-Resolution 2D Texture Array

Width = iDivUp(W, 2)
Height = iDivUp(H, 2)

1 Draw call with 4xMRTs
STEP 2: Jitter-Free Sampling

Input: Texture Array A (slices 0,1,2,3)

Output: Texture Array B (slices 0,1,2,3)
STEP 2: Jitter-Free Sampling

1. Constant jitter value per draw call
   ➔ better per-sample locality

2. Low-res input texture per draw call
   ➔ less memory bandwidth needed
STEP 3: Interleave Results

1 Draw call

With 1 Tex2DArray fetch per pixel
4x4 Interleaving

4x4 jitter textures are commonly used for jittering large sparse filters

Can use a 4x4 interleaving pipeline

1. **Deinterleaving**: 2 Draw calls with 8xMRTs
2. **Sampling**: 16 Draw calls
3. **Interleaving**: 1 Draw call
Full-Res Jittered SSAO
1920x1200: 3.47 ms

GPU time measured with non-blocking D3D11 timestamp queries on GTX 680
4x4-Interleaved SSAO
1920x1200: 1.74 ms [2.0x]

GPU time measured with non-blocking D3D11 timestamp queries on GTX 680
GPU time measured with non-blocking D3D11 timestamp queries on GTX 680

Full-Res Jittered SSAO
2560x1600: 9.25 ms
GPU time measured with non-blocking D3D11 timestamp queries on GTX 680

4x4-Interleaved SSAO
2560x1600: 3.14 ms [2.9x]
# 4x4-Interleaving Performance

<table>
<thead>
<tr>
<th>GPU Times (in ms) *</th>
<th>1920x1200</th>
<th>2560x1600</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEP 1: Z Deinterleaving</td>
<td>0.12</td>
<td>0.21</td>
</tr>
<tr>
<td>STEP 2: SSAO</td>
<td>1.50</td>
<td>2.69</td>
</tr>
<tr>
<td>STEP 3: AO Interleaving</td>
<td>0.12</td>
<td>0.24</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1.74</strong></td>
<td><strong>3.14</strong></td>
</tr>
</tbody>
</table>

* Measured with non-blocking D3D11 timestamp queries on GTX 680

**Input** = full-res R32F texture  
**Output** = full-res SSAO
Texture-Cache Hit Rates

Can query per-draw cache texture-cache hit rates via:
NVIDIA PerfKit
AMD GPUPerfStudio 2

Example GPU counters *
tex0_cache_sector_misses
tex0_cache_sector_queries

<table>
<thead>
<tr>
<th>Resolution</th>
<th>GPU Time</th>
<th>Hit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Interleaved</td>
<td>3.47 ms</td>
<td>38%</td>
</tr>
<tr>
<td>4x4-Interleaved</td>
<td>1.50 ms</td>
<td>67%</td>
</tr>
<tr>
<td>Gain</td>
<td>2.3x</td>
<td>1.8x</td>
</tr>
</tbody>
</table>

Texture-Cache Hit Rates

Can query per-draw cache texture-cache hit rates via:
- NVIDIA PerfKit
- AMD GPUPerfStudio 2

Example GPU counters *
- tex0_cache_sector_misses
- tex0_cache_sector_queries

<table>
<thead>
<tr>
<th>2560x1600</th>
<th>GPU Time</th>
<th>Hit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Interleaved</td>
<td>9.25 ms</td>
<td>32%</td>
</tr>
<tr>
<td>4x4-Interleaved</td>
<td>2.69 ms</td>
<td>62%</td>
</tr>
<tr>
<td>Gain</td>
<td>3.4x</td>
<td>1.9x</td>
</tr>
</tbody>
</table>

Example Sampling Pattern

With no Interleaved Rendering
With 2x2 Interleaved Rendering

Sample coords are snapped to half-res grid aligned with kernel center
With 4x4 Interleaved Rendering

Sample coords are snapped to \texttt{quarter-res grid} aligned with kernel center
With 4x4 Interleaved Rendering

Sample coords are snapped to quarter-res grid aligned with kernel center

Inner region may be sampled in additional pass with full-res input texture
Interleaved Rendering: Wrap Up

Improves performance
  Better sampling locality
  No jitter texture fetch anymore

Looks the same
  For large kernels (>16x16 full-res pixels)
  Missed details for small kernels may be added back

Used in shipping games
  ArcheAge Online (2013)
  The Secret World (2012)
4x4-Interleaved SSAO in Metro: Last Light (preview)

Image courtesy of 4A Games
Acknowledgments

NVIDIA
DevTech-Graphics
Miguel Sainz
Holger Gruen
Yury Uralsky
Alexander Kharlamov

Game Developers
Funcom
XL Games
4A Games
DICE
Crytek
Questions?

Louis Bavoil
lbavoil@nvidia.com
References


References


