Road to Real-Time
Order-Independent Transparency

Marco Salvi

Beyond Programmable Shading Course, ACM SIGGRAPH 2011
Today's talk is about trying to understand how far off we are from having robust, high quality and fast OIT methods for real-time applications. We will begin by showing why there is a real need for better transparency methods and then we'll classify and review some of the most representative ones.

At the end of the talk we will review the current situation in order to try to understand what kind of next steps we should take.
“Reliance on a single program for rendering an entire scene is a poor strategy.”

“Separating the image into elements which can be independently rendered saves enormous time.”

"Compositing Digital Images", @SIGGRAPH 1984
Thomas Porter & Tom Duff

Beyond Programmable Shading Course, ACM SIGGRAPH 2011
This is a very good summary of the sort of problems we have to face while rendering transparent geometry with alpha-blending. AB is the still de-facto standard compositing technique in real-time applications, despite the fact it was introduced almost 30 years ago.

Since AB is an order-dependent technique every time we add a new transparent object to a scene we have to make sure it doesn’t intersect with other transparent objects or that is only visible from a certain direction, if we don’t want to incur in glaring image artifacts. So it’s not surprising that developers would love to abandon this method in favor of more robust techniques, if they were available.
A classic example of this problem is given by hair rendering. Even if sort every individual hair strand we still get images like this one on the left side of the slide while what we really want to render is something like what we have on the right side of the slide.
A classic example of this problem is given by hair rendering. Even if sort every individual hair strand we still get images like this one on the left side of the slide while what we really want to render is something like what we have on the right side of the slide.
This problem is well exemplified in current games by the space marine character archetype. A bandana is strategically placed on his head to conceal his hair.

Another way to address this problem from the content standpoint is to have characters with very short hair or completely bald characters.
This problem is well exemplified in current games by the space marine character archetype. A bandana is strategically placed on his head to conceal his hair. Another way to address this problem from the content standpoint is to have characters with very short hair or completely bald characters.
This problem is well exemplified in current games by the space marine character archetype. A bandana is strategically placed on his head to conceal his hair.

Another way to address this problem from the content standpoint is to have characters with very short hair or completely bald characters.
Incorrect compositing is not the only problem that we can solve with OIT techniques. Also objects like fences that are usually rendered with alpha-testing which are often prone to generate aliasing artifacts that can be rendered as semi-transparent objects with much better results.
The same technique can also be applied to foliage to get anti-aliased branches and leaves
If we look at this problem from an operational point of view, what does actually mean to correctly composite fragments?

How do we compute the single contribution of a transparent fragment to the final color of a pixel?

The visibility function is computed as a product of step functions, and represent the total transmittance between a transparent fragment and the viewer.
Assume thin transparent light blockers; the final pixel color it’s “just” a sum of the fragments’ color pre-multiplied by their alpha and visibility terms.
• High image quality
  – Accurately evaluate compositing equation
  – No major spatial and temporal artifacts

• High and stable performance
  – Low variability. Performance mostly independent from fragments ordering

• Bounded memory usage
  – No variable length data structures

• High image quality
  • No major spatial & temporal artifacts
    • Accurate evaluation of compositing equation

• High and predictable & stable performance
  • Weak or no dependencies on primitives ordering
    • Avoid methods that are based on sorting fragments

• Bounded memory usage
  • A must for real-time applications
OIT Algorithms Classification

- Solve compositing equation recursively
  - Composite fragment with result of previous composite operation

- Solve compositing equation by computing and evaluating (compressed) visibility functions

Recursive = implicit evaluation of visibility function
Other methods perform an explicit evaluation of the visibility function
Recursive Solvers

- Alpha Blending / Compositing
- Depth Peeling
- A-buffer
- $Z^3$ algorithm
Alpha Compositing [Porter and Duff 1984]
Alpha Compositing [Porter and Duff 1984]
Alpha Compositing [Porter and Duff 1984]

- Fast and stable/predictable performance
- No additional storage required
- But order-dependent..

\[ C_0 = \alpha_0 c_0 \]
\[ C_n = \alpha_n c_n + (1 - \alpha_n) C_{n-1} \]
More modern approaches to depth peeling can process multiple layers in one pass. Despite these advancements DP is not a frequently used technique.
A-buffer requires sorting fragments per-pixel. Once fragments are sorted we composite them using alpha-blending/compositing.
A-buffer requires sorting fragments per-pixel. Once fragments are sorted we composite them using alpha-blending/compositing.
A-buffer requires sorting fragments per-pixel. Once fragments are sorted we composite them using alpha-blending/compositing.
A-buffer Limitations

- Poor & unstable performance, typically memory bandwidth limited
- Unbounded memory requirements

Left: image rendered with Stochastic Transparency (24 samples per pixel, we can still notice some noise) vs Right: scene rendered via A-buffer (some smoke is missing due to lack of memory)
The Z³ algorithm [Jouppi and Chang 1999]

- Bounded A-buffer
  - Up to N fragments per pixel (sorted)

- Merge fragments to keep pixel memory footprint constant
  - Distance & coverage based compression metric
VFIs are monotonically decreasing functions with values over the interval \([0, 1]\).
Visibility Based Solvers

- Fragment Parallel Compositing
- Occupancy Maps
- Opacity Shadow Methods
- Stochastic Transparency
- Adaptive Transparency
• Compute visibility via parallel segmented scan
  – Load-balance across irregular number of fragments per pixel
• Evaluate compositing equation via parallel reduction

\[
pixel\ color = \color{red} + \color{blue} + \color{green} + \ldots
\]
• Assume all transp. geom. has the same alpha
  – Not applicable to many objects (smoke, foliage, etc.)

• Bit-field represents step functions of same height and spaced at regular depth
  – A slab map allows to re-modulate steps height on a given depth region

• Use visibility representation for OIT & shadows
Opacity Mapping Methods

- Replace visibility with opacity: \(-\frac{d}{dz}\left[\ln(\text{vis}(z))\right]\)
  
  - Less well-behaved function (no monotonicity, no [0,1] bounds)

- Basis function
  
  - Piece-wise constant intervals: Opacity Shadow Maps [Kim et al. 2001]
    
    - Warp OSMs first layer: Deep Opacity Shadow Maps [Yüksel et al. 2008]
  
  - Trigonometric: Fourier Opacity Mapping [Jansen et al. 2010]
• Fixed length visibility representation
  – Regular steps at irregular locations
  – A2C & z-test on MSAA samples
    • Compression by removing random step
    • Efficient use GPU fixed function HW
• Fast but “noisy” with complex objs
  – Can require many samples per-pixel
Adaptive Transparency [Salvi et al. 2011]

• Derived from volumetric shadow method [Salvi et al. 2010]
  – Optimized for thin objs and OIT

• Store up to N steps per pixel
  – Arbitrary location and height
  – Area based compression metric

We compute the smallest area variation and we remove the node that generates it. We never remove the front-most node, as it can give very important visual cues (closest transparent fragment to the viewer).
We compute the smallest area variation and we remove the node that generates it.
We never remove the front-most node, as it can give very important visual cues (closest transparent fragment to the viewer)
We compute the smallest area variation and we remove the node that generates it.
We never remove the front-most node, as it can give very important visual cues (closest transparent fragment to the viewer)
We compute the smallest area variation and we remove the node that generates it.
We never remove the front-most node, as it can give very important visual cues (closest transparent fragment to the viewer)
We compute the smallest area variation and we remove the node that generates it. We never remove the front-most node, as it can give very important visual cues (closest transparent fragment to the viewer).
We compute the smallest area variation and we remove the node that generates it. We never remove the front-most node, as it can give very important visual cues (closest transparent fragment to the viewer).
We compute the smallest area variation and we remove the node that generates it.
We never remove the front-most node, as it can give very important visual cues (closest transparent fragment to the viewer)
We compute the smallest area variation and we remove the node that generates it. We never remove the front-most node, as it can give very important visual cues (closest transparent fragment to the viewer).
We compute the smallest area variation and we remove the node that generates it.

We never remove the front-most node, as it can give very important visual cues (closest transparent fragment to the viewer)
We compute the smallest area variation and we remove the node that generates it. We never remove the front-most node, as it can give very important visual cues (closest transparent fragment to the viewer).
We compute the smallest area variation and we remove the node that generates it.
We never remove the front-most node, as it can give very important visual cues (closest transparent fragment to the viewer)
Adaptive Transparency [Salvi et al. 2011]

SMOKE scene
21 ms - 10.6 MFragment
Max fragment per pixel: 312
30x faster than A-buffer
2.5x faster than Stoc. Transp.

HAIR scene
48 ms - 15.0 MFragment
Max fragment per pixel: 663
40x faster than A-buffer
2x faster than Stoc. Transp.

FOREST scene
8 ms - 6.0 MFragment
Max fragment per pixel: 45
7x faster than A-buffer
2x faster than Stoc. Transp.

Beyond Programmable Shading Course, ACM SIGGRAPH 2011
• Higher quality than other “lossy” OIT methods
  - Works on any transparent object type (foliage, smoke, glass, etc.)

• Designed to run in fixed memory
  - Prototype uses variable memory data struct due to 3D APIs limitations

• High and scalable performance
  - Easy to trade-off IQ for performance and storage by tuning per-pixel step/node count
What’s Next?

- AT & ST are good examples of new algorithms close to ideal OIT method for real-time apps
  - ST main limitation is noise / lower quality per vis. rep. storage
  - AT main limitation is current variable memory implementation
- We are rapidly converging towards entirely practical and robust OIT methods for real-time rendering!
Open Problems

• Deferred shading and OIT methods
  – Many methods are trivial to extend to DSers..
  – ..but awfully inefficient..

• Some interesting work left to do in this area
Acknowledgements

- Special thanks to Jefferson Montgomery and Aaron Lefohn

- Craig Kolb, Matt Pharr, Charles Lingle and Elliot Garbus for supporting this work.

- Jason Mitchell and Wade Schinn at Valve Software for the assets from Left-for-Dead-2

- Cem Yuksel (Cornell University) for the hair model

twitter:  @marcosalvi
e-mail: marco.salvi@intel.com
blog:  http://pixelstoomany.wordpress.com
Bibliography


• **Idea:** Save bandwidth by working with an approximate visibility function

![Graph showing the relationship between transmission and distance from the viewer.](image)

0 1

Distance from viewer (depth)

200+ steps
**Idea:** Save bandwidth by working with an approximate visibility function

---

Transmittance

Distance from viewer (depth)

200+ steps

---

Transmittance

Distance from viewer (depth)

Beyond Programmable Shading Course, ACM SIGGRAPH 2011
**Idea:** Save bandwidth by working with an approximate visibility function.

![Graphs showing transmission over distance from viewer (depth)]
• **Idea:** Save bandwidth by working with an approximate visibility function.

![Graph showing transmission vs. distance from viewer (depth)]

- Green line: 200+ steps
- Orange line: 32 steps

---

Beyond Programmable Shading Course, ACM SIGGRAPH 2011
**Idea:** Save bandwidth by working with an approximate visibility function.

- **200+ steps**
- **32 steps**

Beyond Programmable Shading Course, ACM SIGGRAPH 2011
Cannot map DX11 atomic ops to fragment insertion/compression ops
Cannot avoid data races as GPUs can concurrently shade two or more fragments that map to the same pixel (no atomicity)
We avoid those issues by giving up a bounded memory implementation (proof-of-concept implementation)

---

**AT Proof-of-Concept Implementation**

1) Render transparent fragments to per-pixel lists
   - Same as A-buffer implementation

2) For each pixel: build an approximate visibility function and use it to composite all transparent fragments
   - Full-screen pass guarantees atomicity
Bandwidth Requirements

![Graph showing Bandwidth Requirements](image)

- AT8 MSAA 1x
- AT8 MSAA 4x
- ST16 MSAA 1x
- ST16 MSAA 4x
- VRB MSAA 1x
- VRB MSAA 4x

MBytes / frame vs MFragment / frame

63/66
Beyond Programmable Shading Course, ACM SIGGRAPH 2011
AT Future Work

• Investigate bounded memory implementations
  — Per-pixel locks? New frame-buffer format?

• Better visibility data compression
  — Reduce MSAA impact on memory requirements
Max depth complexity = 120
Transparency nodes = 8
Average depth complexity = 4
• Capture fragments in FIFO
  – Enable efficient multi-pass shading
• Require global sort
  – A-buffer requires local sort
• FIFO can overflow
  – Similar to A-buffer