Adaptive Transparency

Marco Salvi
INTEL
About us

• Project developed with Jefferson Montgomery and Aaron Lefohn

• ART (Advanced Rendering Technology)
  – Real-time rendering and programming models research
Talk Outline

• Introduction
• Algorithm Description
• Live Demonstration
• Engine Integration & anti-aliasing
• Limitations
• Conclusions & Q&A
An (old) open problem

• “Order-dependent transparency has always been a big limitation for content creators & developers
  – Restrictive art pipeline: no glass houses
  – Even windows on cars & buildings can be painful
  – Restrictive interaction between objects”

• “Order-independent transparency is must going forward
  – Big challenge! Gradual process”

“Five Major Challenges in Interactive Rendering”, SIGGRAPH 2010
Johan Andersson
order-**dependent** transparency (alpha blending)

order-**independent** transparency (adaptive transparency)
A-buffer

1) Render fragments color and depth in per-pixel lists

2) Per-pixel sort and composite fragments to the frame buffer
A-buffer Limitations

- Poor & unstable performance, memory BW limited
  - BW scales quadratically with (per pixel) fragment count
- Unbounded memory requirements
Re-balancing BW usage

• Trade-off memory bandwidth for compute
  – Data compression?
    • E.g.: GPUs already do it with color/z compression

• No clear way to rebalance BW usage when sorting fragments
An Alternative Compositing Method..

• Re-factor the alpha-blending equation to avoid recursion and sorting*
  – Process terms in any order (iterative)
  – Introduce a “visibility function”

• \[
  \text{final	extunderscore color} = \sum c_i \alpha_i \text{vis}(z_i)
\]

* [Sintorn et al. 2009]
The visibility function (VF) models how light is absorbed as it travels through space.

VFVs are monotonically decreasing functions with values over the interval $[0, 1]$.

- For thin blockers the VF is a product of step functions.
Alternative A-buffer Implementation

• For each pixel..
  1) Read fragments from list and build a visibility function
  2) Read list again, evaluate \( \text{vis}() \) and composite fragments
    • Apply \( \sum c_i \alpha_i \text{vis}(z_i) \)

• Still BW limited, no performance improvement 😞
• Idea: Save BW by working with an approximate visibility function (AVF)
• Represent AVF as a sorted (front-to-back) sequence of nodes stored into a **fixed-size** array.

• To each **red** node corresponds a pair of values for depth and transmittance: \((d, t)\)
  – Don’t store first node, it always carries the values \((0, 1)\)
• To add a fragment $f$ to the AVF we multiply all nodes located behind it by $(1 - \alpha_f)$
• To compress visibility we remove the node that generates the smallest area variation
AVF Storage

• Store AVF in the frame buffer?
  – Data structure update cannot be mapped to DX11 blend modes
  – No Read/Modify/Write to the frame buffer from pixel shaders

• Store AVF in a Read/Write buffer (UAV)?
  – Cannot avoid data races as GPUs can concurrently shade two or more fragments that map to the same pixel (no atomicity)
    • Cannot map DX11 atomic ops to fragment insertion/compression ops
Adaptive Transparency

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
Live Demo

• Up to 40x faster than A-buffer
  – $O(N)$ vs. $O(N^2)$

• High image quality (no noise, popping, etc.)
  – Easy to trade-off IQ for performance by tuning node count

• Works on any type of transparent geometry
  – Foliage, particles, hair, glass, etc.
Results

SMOKE scene
10.6 MFragment
Max fragment per pixel: 312
21 ms (30x faster than A-buffer)

HAIR scene
15.0 MFragment
Max fragment per pixel: 663
48 ms (40x faster than A-buffer)

FOREST scene
6.0 MFragment
Max fragment per pixel: 45
8 ms (7x faster than A-buffer)
Integration with Forward Renderers

• Simply replace your typical alpha-blending pass with lists rendering plus AT compositing
  – Exploit Hi-Z to reject occluded transparent fragments
Integration with Deferred Renderers

• Memory & shading costs associated to defer shading over transparent layers are prohibitive
  – Not aware of any title deferring shading for transparent geometry, even without OIT

• Fall-back: use forward rendering like pass for transparent geometry
MSAA

• Store coverage mask in list nodes, composite and resolve at MSAA-sample frequency*
  – Slow, not accurate on intersections

• Do we need anti-aliasing on transparent geometry?
  – Not with proxy-geometry (foliage, particles, etc.)
  – MLAA can help in the remaining scenarios

* [Carpenter 1984; Yang et al. 2010]
AT Limitations

• Same unbounded memory requirements of A-buffer
  – Current APIs issue, could be addressed on future APIs
  – Poor performance vs. running out of memory
    • 10% of high-end GPU memory to render up to ~15M Fragments

• Screen-space tiling helps avoiding memory issues
  – Render geometry N times with scissoring
  – Performance impact (~50% fill rate reduction with 4 tiles)
    • Per-tile frustum culling could improve performance
Conclusions

• AT is a new high-performance OIT algorithm for DX11 class GPUs

• Excellent image quality
  – E.g.: enables nicely anti-aliased foliage without MSAA

• Easy to integrate with modern rendering engines
Acknowledgements

• Jefferson Montgomery, Aaron Lefohn and the rest of the Advanced Rendering Team at Intel.

• Special thanks to 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
Q&A

slides & demo source code: http://intel.ly/aoit_gdc

twitter: @marcosalvi

e-mail: marco.salvi@intel.com

blog: http://pixelstoomany.wordpress.com
Learn more about developing games & apps for MeeGo ... 

• Intel AppUp Application Lab sessions are scheduled March 3 at 12pm and 3:30pm across the street at the Marriott Marquis

• Space is limited, register today at intelapplicationlab.com and arrive 15-30 minutes before the session to check in and reserve a seat!

Talk about this on Twitter! #appup
Bibliography

To composite \( n \) fragments with color \((c_i, \alpha_i)\) and depth \( z_i \), we can use two equivalent OIT methods:

a. For each pixel **sort** and recursively composite fragments according to their distance from the viewer.

\[
f_n = \alpha_n c_n + (1 - \alpha_n) f_{n-1}
\]

b. For each pixel compute a visibility function, which can be used to composite fragments in any order.

\[
\sum_{i=1}^{n} \alpha_i c_i \text{vis}(z_i)
\]
Visibility Functions

• Visibility functions are monotonically decreasing functions with values over the interval $[0, 1]$.
  – For thin blockers the VF is a product of *step functions*.

\[
vis(z) = \prod_{z_i < z}(1 - \alpha_i)
\]