Practical Order Independent Transparency

Johan Köhler, Treyarch

1 Introduction

Transparencies have always been a tricky problem for games, to render them correctly you have to draw them in front-to-back order from the view of the camera, traditionally this means to sort them when you do this you have to break apart material batches and sometimes even meshes, which affects performance. Order Independent Transparency (OIT) solves all these problems but simple implementations severely limit the number of layers of transparencies you can have. This document describes a method that combines OIT with software rasterized sprites via compute shaders, which we'll call "Compute Sprites". Most of our transparent layers were due to particles, the Compute Sprites efficiently take care of most of them, which allows for a practical implementation of OIT for the remaining mesh transparencies in the scene.

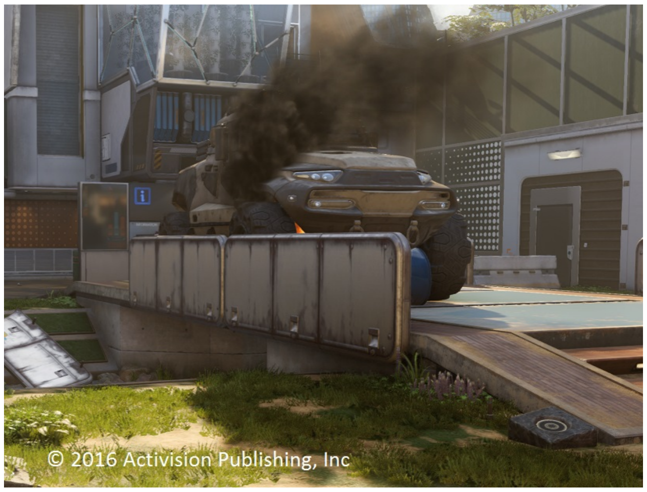

Figure 1: A scene with glass windows and smoke particles that cast shadows

2 Description

Our mesh OIT buffer is comprised of 9 layers packed in a texture array: we store color, alpha and depth. Pixel shaders write out the final color value using UAV's, without front-to-back sorting. The Compute Sprites particles are camera facing and are represented by a 2D position matrix and material settings like texture, diffuse color and so on. In a compute shader pass we read all OIT layers, sort them, merge with the generated Compute Sprites color values, blend with volumetric lighting and write the final blended color to the framebuffer. We implemented Compute Sprites to be able to efficiently perform OIT, but as a nice bonus they turn out to be also faster than regular particles as we don't pay bandwidth on overdraw. Note also that on the other hand, any implementation of Compute Sprites does also need to consider OIT for meshes, otherwise it wouldn't be possible to correctly sort these particles with the other transparencies in the scene.

3 Other features

Compute Sprites can Z feather with OIT surfaces, so fire particles near glass, for example, will not have a hard edge. When running out of OIT layers the transparency will write to a separate frame buffer, so a flamethrower that have particles incompatible (not camera facing) with Compute Sprites and use a lot of transparency layers won't stop rendering. Particles are lit correctly and won't "pop" as they enter shadows, will not be affected by lights behind walls and can also cast shadows, see example in Figure 1.

4 Implementation details

All the compute particles are rendered in one shader but each one might use different material textures: on PS4 and Xbox One we use bindless textures to handle this while on PC (DX11) we combine all material textures into a texture array. We calculate which particles are visible over the screen using a grid of 32x32 pixel tiles (performing depth tests per tile, each stores its farthest depth) and store the particle per tile using a bit array. Rasterization happens in 8x8 tiles (threadgroups) and we only load draw information (particle coordinates, materials) for the visible sprites in each tile. To use as much of the GPU's parallelism each shader wave loads up and processes sprites if any of its lanes use them, this allows us to use scalar loads on AMD (GCN) hardware speeding up rendering significantly. All particle lighting is done using a runtime generated lightmap where each visible particle has an

8x8 texel square allocated (see figure 4). The sprite and OIT merge shader uses uniform dynamic branching (using ballot) on PS4 and XBox One, so for example, if there are no OIT pixels used by any of a wave's pixels it branches to a code path that does not do any OIT sorting and merging. The Compute Sprites and OIT resolve shader also generates an Adaptive Transparency[1] estimate, which is used to properly composite volumetric effects (which are calculated in a "2d" raymarching shader, not using 3d volume textures).

5 Timings

	Scene similar to Figure 1			**Particle heavy scene**		
	Disabled	OIT only	OIT + CS	Disabled	OIT only	OIT + CS
Transparent mesh rendering	0.40 ms	0.50 ms	0.50 ms	0.50 ms	0.60 ms	0.60 ms
Particle rendering	0.70 ms	1.00 ms	0.12 ms	2.20 ms	2.40 ms	0.06 ms
Particle lightmap		0.20 ms	0.20 ms		0.20 ms	0.20 ms
OIT resolve (+ CS) + AT		1.00 ms	1.20 ms		1.60 ms	2.70 ms
Total	1.10 ms	2.70 ms	2.02 ms	2.70 ms	4.80 ms	3.56 ms

The disabled mode can't render transparencies correctly, also volumetric compositing and lightmaps is not enabled.

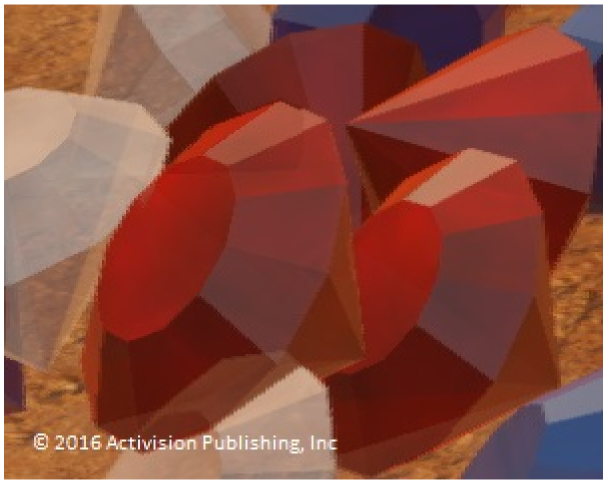

Figure 2: Intersecting transparent geometry.

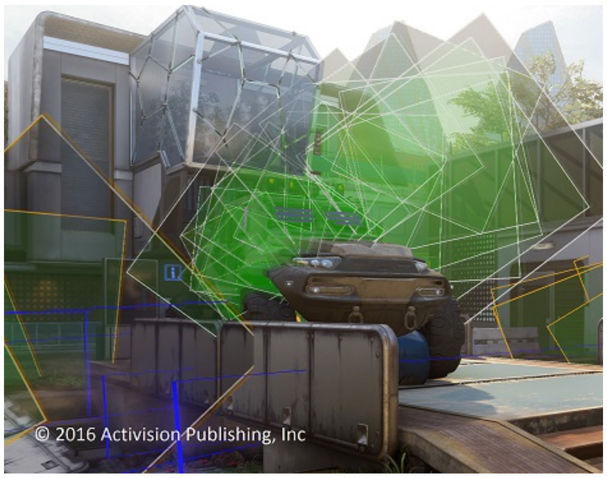

Figure 3: Particle overdraw debug mode, white and yellow are compute sprites, blue is OIT.

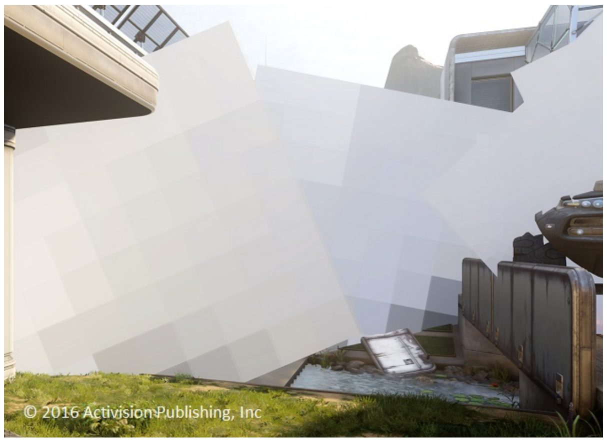

Figure 4: Realtime particle lightmap visualization.

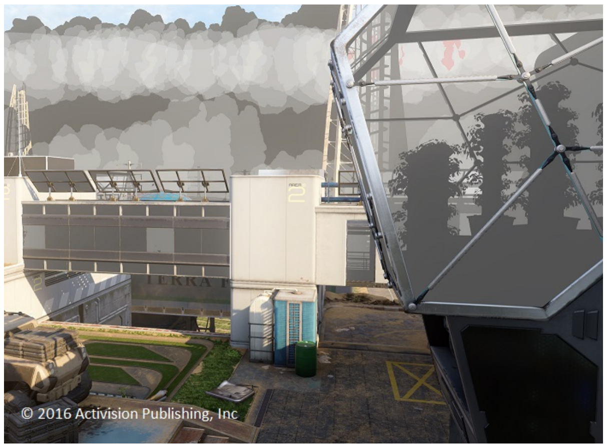

Figure 5: OIT overdraw debug mode, notice some red pixels in the top right where more than 9 layers are used.

[1]https://software.intel.com/sites/default/files/m/1/2/a/5/7/37944-adaptive_transparency_hpg11.pdf