Transparency

CS311: Computer Graphics
Final Project by Martin Green and Ritvik Kar

1.0 Introduction

Transparency is a core technique required for the application of scientific visualization. It becomes essential in the field of volumetric rendering, where rendering can accomplish both color and transparency. Accurate replication of transparency is a difficult job primarily because it is a computationally expensive process. Ray-tracing is the only method that provides us with realistic transparency but is time intensive. Therefore we often use rasterized, non-refractive transparency, where we only blend colors together and do not replicate how light interacts with the objects.

Real transparency is a subtractive-color process whereas transparency in graphics is a blending process. OpenGL's blending mechanism combines color present in the framebuffer with the color of the incoming fragment, storing the result back in the framebuffer.

2.0 Transparency in OpenGL

OpenGL does not implement transparency as a primitive but creates the effect with its blending feature. OpenGL enables blending as such:

```
    glEnable (GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

After enabling blending, incoming colors are blended with colors in the framebuffer to compute the final pixel color. The `glBlendFunc()` is what determines how the bending is going to be computed. The above input blends the incoming color with an alpha value and modifies the destination color with one minus the alpha value. The

RGBA color is in accordance with either a `glBlendFunc` or `glBlendFunci` call. The specifications for these blending functions are as follows:

```
     glBlendFunc (GLenum sfactor, GLenum dfactor);
     glBlendFunci (GLuint buf, GLenum sfactor, GLenum dfactor);
```

where,

`buf`: specifies the index of the draw buffer for which to set the blend function

`sfactor`: specifies how the red, green, blue, and alpha source blending factors are computed.

`dfactor`: specifies how the red, green, blue, and alpha destination blending factors are computed. [1]

`glBendFunc` activates a blending operation on all drawing buffers, whereas `glBendFunci` allows definition of operations for individual drawing buffers. OpenGL blending primarily utilizes the alpha channel, meaning opaqueness. Alpha ranges from 0 to 1, with 1 being fully opaque while 0 being fully transparent.

3.0 Types of Transparency and their Implementations

We will discuss three main types of rasterized transparency that are used today in various contexts and applications. They are screen-door transparency, alpha blending, and depth peeling.

3.1 Screen Door Transparency

The first, and most basic form of transparency is screen door, also known as stippling or dithering. Instead of altering a pixel by blending it with the fragment behind it, the transparent object is rendered using a binary, on/off system. Depending on the transparency-level of the object, an individual fragment is either rendered or left blank, allowing the object directly behind it to render its fragment at that pixel. For instance, if an object has 25% opacity, then one in every four pixels from it is rendered normally, leaving the rest of them clear to show the opaque object that the transparent one would otherwise be covering.

This system has a few main benefits. The foremost of which is speed; computers handle binary calculations extremely efficiently, for obvious reasons. The color of a pixel for one object is wholly independent of the color of any other pixel. Therefore, we do not need to do any dependent texture reads. We don't need to compute a blended color for a mesh fragment, much less worry about the layer-ordering or depth of any transparent fragment in relation to what is behind it.

OpenGL has native support for screen door transparency through the `glPolygonStipple()` function. It takes in a 32-by-32 bitmap array of unsigned bytes, which describes the pattern for the dithering to represent. However, another simple option to create a dithering pattern is to use an additional texture on the object. It is easy to create unique dithering patterns through Adobe Photoshop by limiting the output colors for an image or gradient to only black or white. Then, we can read from that texture to determine if we will color the fragment or not. This could easily be used to define more complex dithering patterns to apply to entire meshes that aren't as easily specified through a single array.

Dithered transparency has a few obvious drawbacks. The most glaring of which is how pronounced the stipple pattern can be; it would be a clear eye-sore in the middle of a high quality graphics engine. Some of this can be mitigated if the pattern is at a high resolution - at a smaller scale, it is less noticeable. Additionally, if two transparent objects, particularly with the same transparency level, are in front of each other, one could occlude the other in an aliasing, stroboscopic effect. Similarly, an object moving or rotating would appear to be moving 'behind' the screen door, and the dithering pattern would appear motionless. That could be remedied by using the mesh-specific texture and the pattern would appear to move and rotate along with it just fine, even handling changing the projection type. However, that method would be slower, potentially defeating the purpose of using screen door transparency to begin with, because we are doing a dependent-texture read from the dithered texture to determine a fragment color.

That isn't all to say that this type of transparency is completely useless. As mentioned above, it is still very fast and allows for order-independent rendering, in fact some current game engines still use this technique. The DigitalRune engine is one such example. Its most common use is in game engines to fade out objects near the far-plane or to fade between Level-Of-Detail nodes as the camera moves closer. Volition even used it in *Saints Row: The Third* to render transparent materials and then apply a blur to the material to smooth out the details. Screen door transparency is also used for technical imaging; CAD programs, archaeologists, doctors, and biologists have used stippling to represent transparent or semitransparent objects without obscuring the image. It makes it easy to see opaque objects clearly while also making it obvious that there is something else in front of the object, such as a cover on a generator or a wing on an insect.

3.2 Alpha Blending

Alpha Blending is the process of combining a translucent foreground color with a background color, producing a new blended pixel color. The foreground's transparency ranges from 0 (completely transparent) to 1 (completely opaque). Our blended color will be calculated by weighing the foreground and background colors as such:

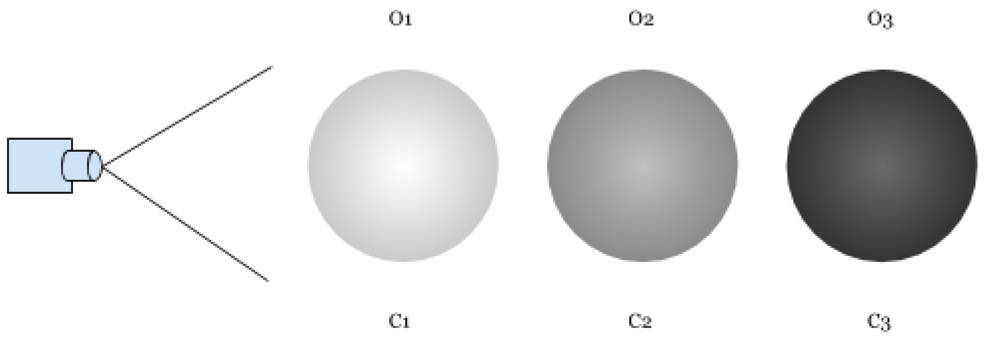
```
Destination(rgb)=(Source(rgb)*Source(a))
                 +(Destination(rgb)*(1-Source(a)));
```

This formula for blending sometimes causes the wrong color for the image, therefore a premultiplied alpha is used, and the blending equation is modified to:

```
Destination(rgb)=(Source(rgb)*1)
                 +(Destination(rgb)*(1-Source(a)));
```

Alpha blending comes with its own set of significant problems. For starters, for something to be transparent, we need to be able to view things behind it. As such, when drawing objects that have an alpha value less than 1, we want to disable the depth test. To make this process efficient, we first draw all opaque objects, disable the depth test using `glDisable(GL_DEPTH_TEST)`, and then draw all of our blended items. This works pretty well giving us transparent looking objects but with the depth test off, we're going to have extremely confusing visual artifacts.

Take for example three objects O1, O2, and O3 with colors C1, C2, and C3 arranged front to back from the viewer's point of view. Give the objects an alpha value of ½ each and the blend function is the one mentioned above.



When we start to draw, the frame buffer receives C1 as its color. The second object changes the calculation to ½C1+½C2 and the third makes it ¼C1 + ¼C2 + ½C3, taking into account the weights. The final color will show O3 to be the darkest color. Now if we were to reverse the order in which we draw the objects, the calculations would be ½C1 + ¼C2 + ¼C3, making O1 the darkest object.[2] This goes to prove that the order we provide the objects in can significantly change the effect of our transparency.

In order to achieve the effect we want, we need to sort the objects of our scene back to front, i.e. we draw the back-most object first and move forward from there. This is much like the painter's algorithm or z-buffering. Sorting becomes problematic when objects intersect each other. They are going to be parts where object A is closer than object B, and visa versa. Therefore just sorting the meshes isn't enough. This isn't only a problem with two objects but also the back faces of individual objects. We're going to have to sort every triangle in our scene, each time the frame is being rendered. This is an expensive process and even that won't be enough to get correct results all the time. If we have multiple transparent triangles intersecting each other, we need to sort one part of a triangle over the other and then bring in the remainder of the triangle. To be able to render this, we calculate the number of triangles that multiple intersecting triangles emit.

Alpha blending comes with a lot of choices. We could implement very basic alpha blending but that's not going to be versatile and is going to have various visual problems. We could sort at the triangle level and handle all the edge cases but that's more expensive in terms of efficiency. Even with our two pass method, drawing all opaque objects followed by all the transparent objects, we face significant problems. Alpha blending will work great for a pre-rendered scene but is either too inaccurate or too slow for real-time rendering.

3.3 Depth Peeling

Depth peeling is the third, final, and most complex, form of rasterized transparency we will talk about. For a transparent object with complex geometry, the object is essentially rerendered in multiple passes, with each pass 'peeling' away the surface of the object at or above the current fragment's depth in the depth buffer. The individual fragments at each pixel do have to be sorted for this method to work, but each triangle of a transparent object *doesn't* need to be sorted, like in alpha blending.

This technique was first described by Cass Everitt, an NVIDIA engineer working in OpenGL, in 2001. It was the first hardware-accelerated method for achieving 'Order-Independent Transparency,' meaning that the bottleneck of sorting triangles can be skipped, instead sorting fragments in each additional passes. To execute this algorithm, first render the scene normally, as if the object weren't transparent. The depth buffer will then store the depth of the first layer of the object. On the next pass, ignore any fragment that is at or in front of the current depth-buffer depth and render the next layer down, storing the new depth values of the object, which can be repeated as many times as necessary. The implementation is very similar to using shadow-mapping and the same basic pieces are used to achieve this effect. The geometry peeling is continued for as many passes as is specified by the developer. Depending on the complexity of the object, only a couple passes are needed to convincingly render an object. The paper by Everitt demonstrates the technique using a teapot, and three passes are sufficient to render it in high-enough fidelity.

Depth peeling *can* suffer from floating-point imprecision errors. On each pass, the depth of the object is re-interpolated. Theoretically, using the same interpolater should produce the same results, but that can't be relied upon and there are many fragments that are right on the edge between one layer and the next. In many cases, this could be a problem which would have to be addressed by the programmer through methods to round the values or by including "fudge-factors" on the depth values. However, graphics cards, starting with NVIDIA, naturally, have started to support methods to store depth values through texture maps, rather than the rasterizer, which very simply solves the floating point variance issues.

4.0 Ray Tracing

We will now shift gears and discuss a wholly different method of image-generation that is completely divorced from OpenGL and rasterized image techniques. Where rasterization is easy to make fast, but hard to make realistic, ray tracing is just the opposite. It is inherently slower than any of the rasterized techniques described above but, like everything else in graphics, what we lose in speed we gain in quality. Although not exact, the high level idea of ray-tracing is that it simulates individual waves of light coming from a light source and bouncing off an object, or it simulates the opposite: waves of light ending at the camera bouncing off in reverse (or, a combination of the two).

This is significantly more computationally intense and most modern GPUs are not yet optimized for this type of calculation. There are some manufacturers who have begun producing ray tracing optimized embedded GPU chips, such as SiliconArts, but those are far from ubiquitous. That is why ray traced images are rarely rendered real time, instead being done ahead of time for still images, pre-rendered cut scenes, or videos. This type of offline rendering does have one major boon: this algorithm is well suited to run in parallel across multiple different machines in a render farm. Although still significantly slower, parallel computing does offset a large amount of the time required for large, high quality, complex scenes.

Obviously, the main advantage of ray tracing is the realism it provides. In modeling the way light waves behave and interact with objects, transparency naturally falls out of that. When a ray hits a semi-transparent red box, that red ray is simultaneously reflected back to the camera and continues to propagate through the box to interact with the next object the ray collides with. In addition to alpha blending-like transparency, the rays going through an object can self-intersect, such as in the case of a teapot. The color of a ray will become progressively darker the more layers of a single object it goes through, in the same way depth-peeling works.

Additionally, due to the meticulous nature of ray tracing, ray traced imagery can emulate effects that are nearly impossible to create through traditional rasterized means: namely, surface diffraction. Materials such as glass, water, or clear oil diffract light waves that pass through it - any high school physics class could explain that. The rays produced here can simply obey the same physics: when a ray hits a transparent object with an index of refraction that is greater than one, the ray can easily change orientation based on that index. Therefore, it is relatively straightforward to simulate glass and other complex materials or bodies that would be extremely difficult to reproduce through triangle geometry.

5.0 Conclusion

We started this project wanting to implement transparency into our graphics engine and it didn't take us too long to realize that transparency, if done properly, is an extremely complicated task to achieve. Over the course of this paper we have explored different methods of achieving transparency, how they're implemented and their drawbacks. Implementing transparency with rasterization proves to be especially difficult due to various issues relating to the way rasterization works.

OpenGL doesn’t implement transparency as a primitive but instead blends colors together. This blending process could be used in combination with the various techniques of transparency. Although each method comes with its drawbacks, we’ve seen how they can all be used efficiently for different purposes.

Various rasterization problems disappear as soon as we start to consider ray-tracing but we then face an excessive increase in computation time. Ray tracing based transparency, such as what can be found in the 3D modeling program Blender, produces accurate results and works well, but takes an impractical amount of time to render individual frames.

Most scenes can be rendered using a combination of ray-tracing and rasterization, depending on how detailed we want specific aspects of the scene to be. Although transparency as a concept seems pretty straightforward, the various issues that crop up make it an exciting, but complex topic to study.

6.0 Citations

Bailey, Mike. "OpenGL Transparency." *Oregon State University*. 31 Aug. 2016. Web. 15 Mar. 2017.

Cifariello, Francesco. "What Are Some Methods to Render Transparency in OpenGL." *Stack Exchange*. 09 Aug. 2015. Web. 15 Mar. 2017.

Cunningham, Steve. "Blending." *Springer Reference*. *Grinnell College*. Web. 15 Mar. 2017.

Everitt, Cass. "Interactive Order-Independent Transparency." *NVIDIA OpenGL Applications Engineering*. Web. 15 Mar. 2017.

"FAQ Transparency, Translucency, and Blending." *OpenGL.org*. The Khronos Group Inc. Web. 15 Mar. 2017.

Greenberg, Donald, and Douglas Scott Kay. "Transparency for Computer Synthesized Images." Cornell University. Web. 15 Mar. 2017.

Kircher, Scott. "Lighting & Simplifying Saints Row: The Third." Game Developers Conference. Moscone Center, San Francisco. Mar. 2012. Web. 15 Mar. 2017.

Kubisch, Christoph. "Order Independent Transparency In OpenGL 4.x." GPU Technology Conference. Silicon Valley. May 2014. Web. 15 Mar. 2017.

Lu, Aidong, Christopher J. Morris, Joe Taylor, David S. Ebert, Penny Rheingans, and Mark Hartner. "Illustrative Interactive Stipple Rendering." *IEEE Transactions on Visualization and Computer Graphics* 9.2 (2003): 127-38. Web.

McDonald, John. "Alpha Blending: To Pre or Not To Pre." *NVIDIA: Game Works*. Web. 15 Mar. 2017.

McNopper. "McNopper/OpenGL." *GitHub*. 31 May 2016. Web. 15 Mar. 2017.

MSFT, Shawn Hargreaves -. "Depth Sorting Alpha Blended Objects." *Shawn Hargreaves Blog*. Web. 15 Mar. 2017.

Mulder, Jurriaan D., Frans C.A. Groen, and Jarke J. Van Wijk. "Pixel Masks for Screen-Door Transparency." Web. 15 Mar. 2017.

"Order Independent Transparency." *OpenGL SuperBible*. 20 Aug. 2013. Web. 15 Mar. 2017.

"Screen-Door Transparency." *Screen-Door Transparency*. Web. 15 Mar. 2017.

"Transparency Sorting." *OpenGL Wiki*. OpenGL, 25 June 2015. Web. 15 Mar. 2017.

"Tutorial 10 : Transparency." *OpenGL Tutorials*. Web. 15 Mar. 2017.