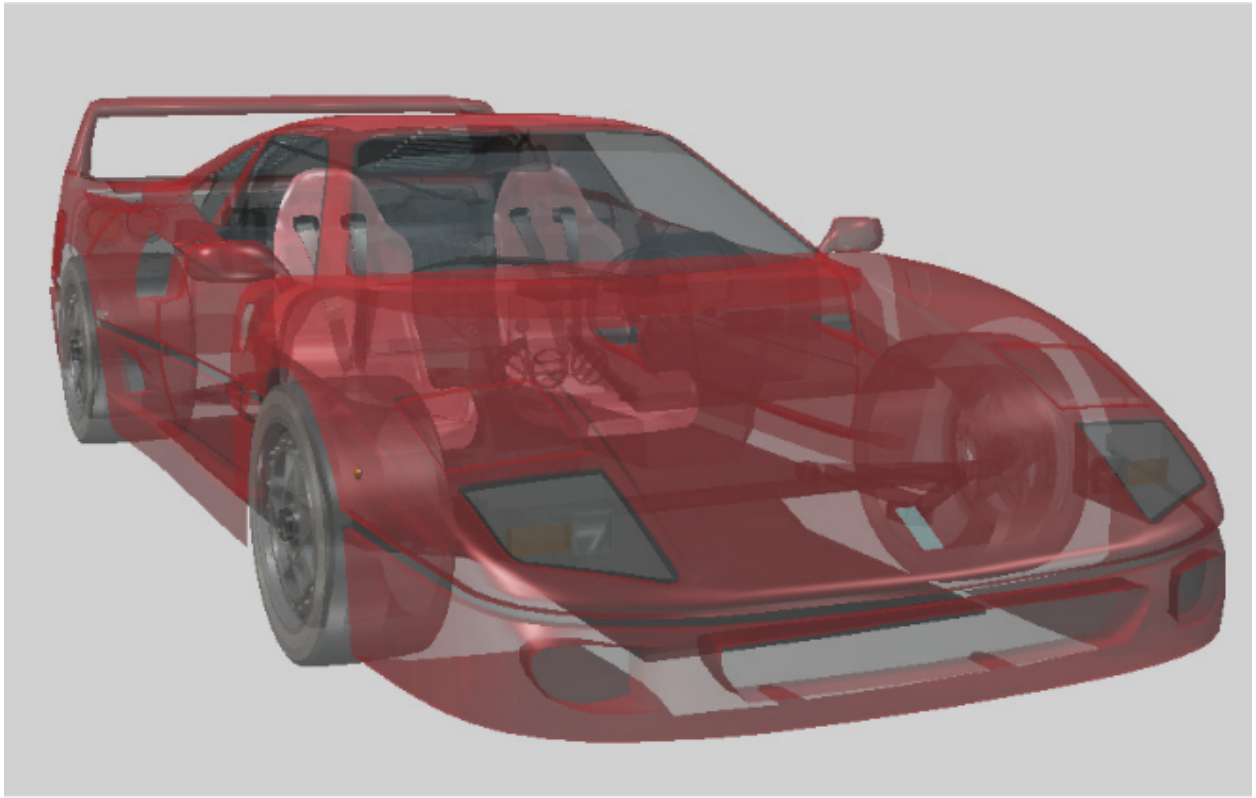
Angus Forbes
CS488
Brian Herman
Walter Dworak
November 8, 2014



Order Independent Transparency

Outline

1. Abstract
    - 1.1. What is order Independent Transparency
    - 1.2. Domains
    - 1.3. Relevance?
2. The Blending Problem
    - 2.1. Traditional Alpha Blending
    - 2.2. Non-Commutative Alpha Blending with Transparency
    - 2.3. Multiplicative Blending
    - 2.4. Additive Blending
3. Explanations of Previous Sorting Methods
    - 3.1. Face Sorting
    - 3.2. Trianangle Sort
    - 3.3. Depth Peeling
4. Problems not solved by previous methods(brian)
5. OIT Methods Overview
    - 5.1. General Form
        - 5.1.1. Meshkin Method

1. Abstract: What is Order Independent Transparency?

1.1. Order Independent Transparency, or OIT, describes techniques to quickly and correctly render one or multiple semi-transparent objects. These are in respect to the arrangement of overlapping faces. All these steps are done using the fragment shader. Previous methods used to solved this problem are subject to slowness or inaccuracy. These problems show up in complex models. Incorrect depth blending and incorrect rendering of curved geometry are common problems when using these old methods.

1.2. These objects are often used in photo realistic video games. These games have objects like glass, smoke, and hair. Engineers often use semi-transparent objects for Computer Aided Design(CAD) and visualizations. For example, the car you have seen on the front cover of this report. These visualizations commonly are used to show the interaction and orientation. The cooperation of multiple parts that would otherwise be impossible.

1.3. The ability to create these visualizations in the way described in this paper. Is the advent of exciting and recent changes and development in graphics hardware and software. The inclusion of more and larger buffers, as well as multi-render targets. Which are the primary hardware features allowing for this ability. Previous methods had to work in the lower resource footprint on the GPU because of the lack of these feautures. Thus this is a source of much of their drawbacks.

2. Alpha Blending Problem

2.1. Basic blending follows this basic equation. This equation determinines

$$RGB_d = A_s \times RGB_s + \left(1.0 - A_s\right) \times RGB_d$$

the resulting color of two RGB vectors. The mathematically commutative function. Takes in an alpha value and RGB vector from a source locations. Then, does an additive blend based on the destination RGB vector multiplied by one minus the source alpha. Then it is added to the product of the source alpha and source RGB vector. This, however, is order dependant and assumes sorted faces. The OpenGL reference makes not that the standard function for this form of blending is only effective on transparent objects when dealing with sorted, farthest to nearest, primitives. ???

2.2. Since we are order dependant the obvious solution is to modify the equation. To product the correct colors. As seen, the function is now

$$RGB_d = RGB_d + A_s \times \left(RGB_s - RGB_d\right)$$

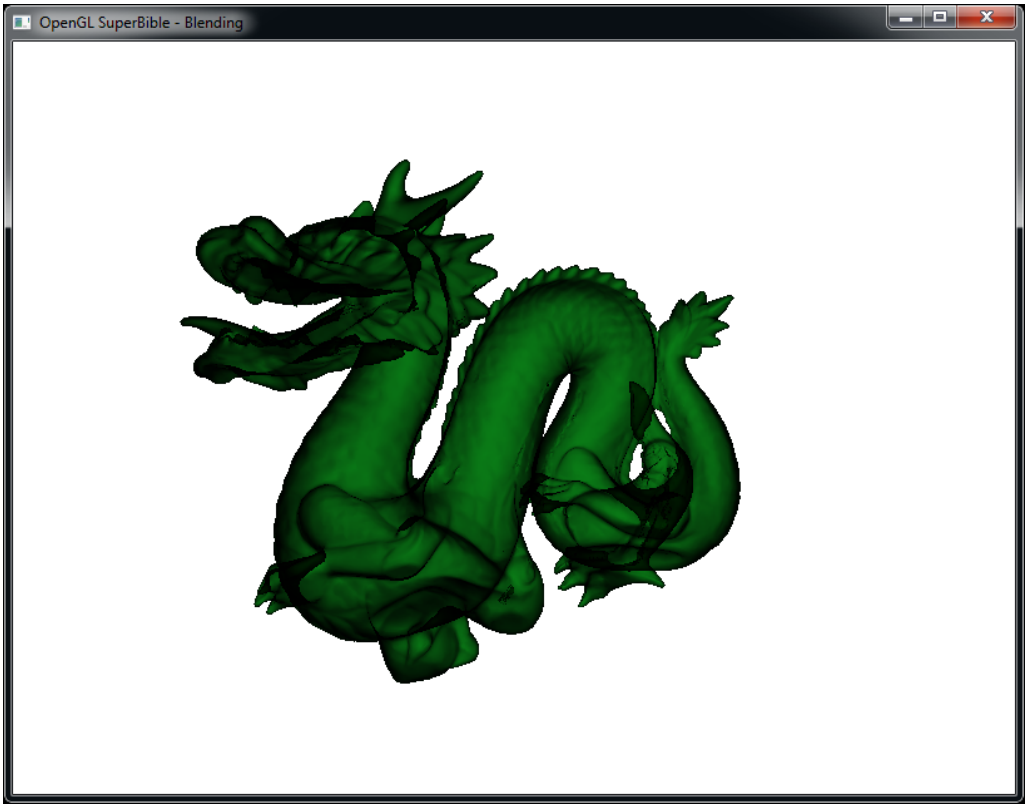
computed with a subtraction operation. Thus, breaking the commutative property. Since order matters some sorting method and is needed

2.3. Other functions are needed so it isn't tightly linked to fragment ordering. Plus being commutative.

2.4. Multiplicative Blending attempts to create a better approximation of translucency. This is done using a multiplication between two RBG

$$RGB_d = RGB_s \times RGB_d$$

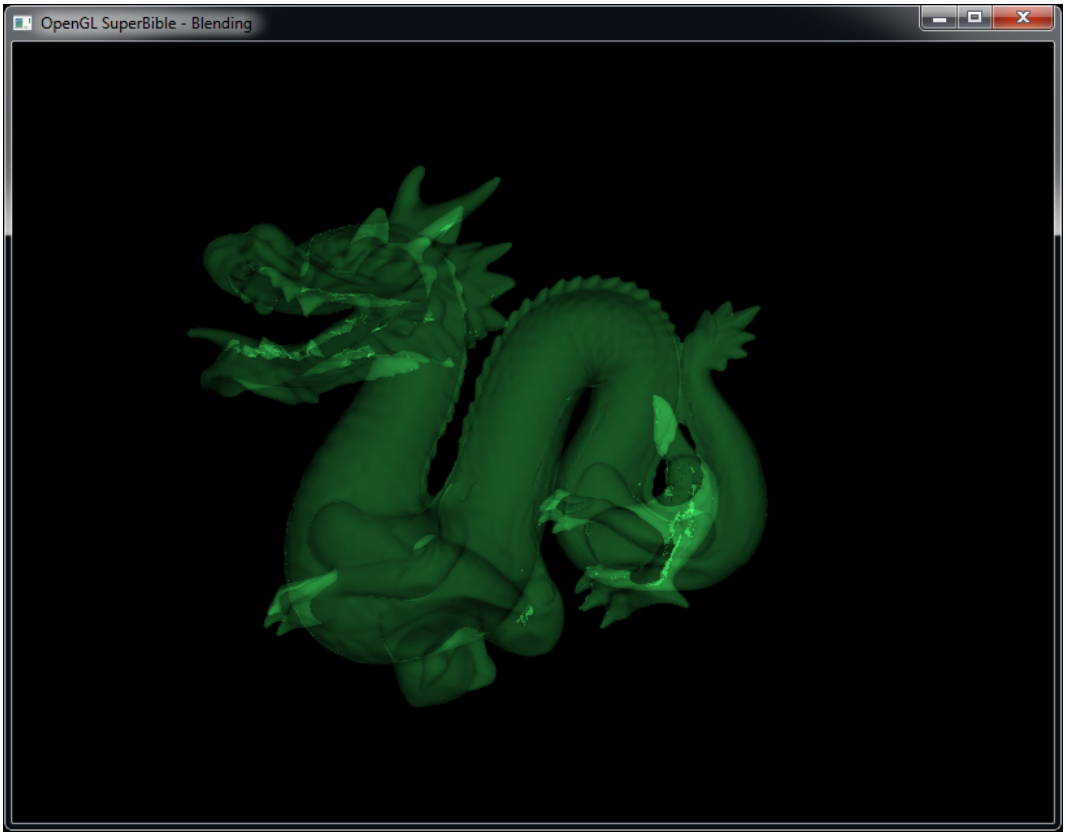
vectors. The glBlendFunc function is called and passed two arguments. The first is GL_ZERO and the second is the source color. All opaque geometry is rendered first with depth testing. And the subsequent tests are kept for rendering the translucent objects. This mimics the attenuation effect produced by light. That light passing through the object with a decent degree of accuracy. However, since this method attenuates whatever color is behind the current one. An incorrect output is easily possible. Despite this allowing the translucent objects to be rendered in any order. The example below requires the window to be cleared to white to produce this effect. And blending with a more complicated background becomes complicated quickly.



2.5. Additive blending works in the opposite order. Instead of removing color from the current frame buffer. Based on the multiplication by a new color

$$RGB_d = RGB_s + RGB_d$$

(attenuation), we will be adding the output of the shader to what is already in the frame buffer. This method is commonly used on particle effects and in this context the particle is adding its “energy” to the rendered scene. Below is an example using the same object. This is worth noting that like the previous method using a sync to a white background. This method is using a sync to black on the background since white color cannot be added to in this way and produce a visible object in the view. Its should also be noted that in the example below. The obscured faces of the semi-transparent model look washed out in



comparison to the previous example and is the reason this method is used on energy producing objects primarily.

3. Limitations of OIT
   3.1. If you have a scene with a large depth range. And it contains large clusters of different-colored transparent objects. And you are in a single rendering pass. In this situation you need to choose a depth weight function that will tell the difference between a cluster and clusters. Also Porter and Duff assume that partially covered locations are not “correlated”. OIT assumes that there are similar colors for surfaces and

distributed in a uniform way. Also OIT puts color precision second to accuracy. Accuracy is the most important thing in OIT.vi

4. Previous Methods that were used are face sorting, triangle sorting, and depth peeling. Face Sorting involved sorting every face of an object with respect to the transparency of the object. This involves sorting every face which is very processor intensive which is not feasible with reasonably large objects. Triangle Sorting is like face sorting but instead of using faces it uses triangles. The idea is very similar and has the same problems. Depth Peeling uses two z-buffers to render the object. It "peels" through the z depth. It is only useful if it is done multiple times. It creates a series of images which are like the layers of an object and it is all blended together to form a scene. All of the previous solutions were not accurate enough, created a large burden on the pipeline, needed the CPU for preprocessing, and view orientation dependant. Accuracy is very important for computer aided design because if a part is not modeled correctly the whole design could fail. Using the cpu for graphics processing will produce a significant loading time for anyone using the application. Finally, being view orientation dependant means that the camera is static. There are no camera transformations. In other words it is linearly dependant on the view matrix. All of these methods are use the CPU none of them use the fragment shader because of the way they are developed.

5. **The OIT Methods**

5.1. The high level overview of how OIT is implemented. The first step is to build an A-Buffer structure using the number of fragments per pixel. This generally takes the form on a linked-list where each node contains the RGB data, alpha, depth value, of each fragment and an index buffer containing the total number of fragments in that pixel. A full screen pass of the fragment shader will then sort the A-buffer and blend the fragments by their sorted index into a final result. Below is a code fragment showing the creation and sorting of a basic example. Ignoring the data types, it

```
layout (early_fragment_tests) in;

layout(rg32ui) uniform coherent uimageBuffer imgAbuffer;
layout(r32ui)  uniform coherent uimage2D imgCounter;
...

 uint oldCounter = imageAtomicAdd (imgCounter, coord, 1u);


 if ( oldCounter < K ){
    imageStore ( imgAbuffer, listPos (oldCounter),
                 fragment);
 }
```

can be seen that the basic implementation follows a traditional flow for iterating through the fragments, adding them to the structure, and then sorting them. Since this data is independent of the view, the model can

be reblended quickly as the view moves around the object since all the data in the sorted A-Buffer can be maintained as long as the model or models are not transformed.

5.1.1. Meshkin's Method is the first to implement order independent

$$C_f = \left(\sum_{i=1}^{n} C_i\right) + C_0\left(1 - \sum_{i=1}^{n} \alpha_i\right).$$

transparency was in 2007. This equation is represented in code below and commutes a weighted sum and the scene is setup in the same way as with the Additive Blending method covered previously. This basic function works well then the alpha is small and the colors similar due to how the results are sorted back to front and over compositing. When the alpha is large the intensity and colors may deviate greatly from the original surface and background colors.

```
drawOpaqueSurfaces();
copyColorBufferToTexture(C0Texture);
glDepthMask(GL_FALSE);
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);
bindFragmentShader("
...
uniform sampler2D C0Texture;
void main() {
...
vec3 C0 = texelFetch(C0Texture,
ivec2(gl_FragCoord.xy), 0).rgb;
gl_FragColor = vec4(Ci - ai * C0, 1.0);
}", C0Texture);
drawTransparentSurfaces();
```

5.1.2. Bavoil and Myer's method improves the approximation for

$$C_f = \frac{\sum_{i=1}^{n} C_i}{\sum_{i=1}^{n} \alpha_i} \cdot \left(1 - \left[1 - \frac{1}{n}\sum_{i=1}^{n} \alpha_i\right]^n\right) + C_0\left[1 - \frac{1}{n}\sum_{i=1}^{n} \alpha_i\right]^n$$

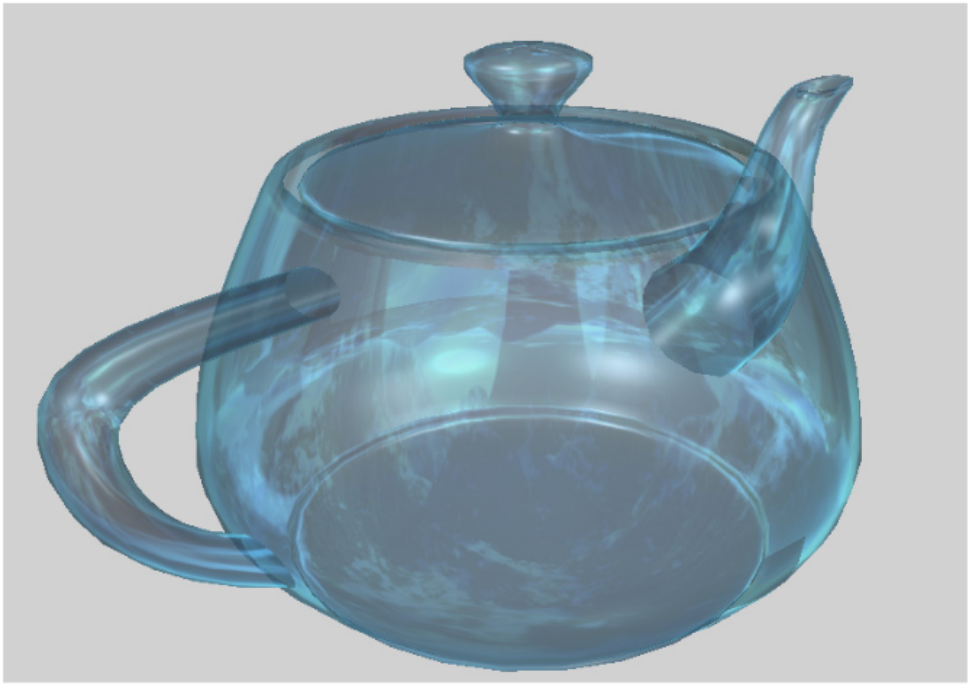
coverage and color using a weighted average operator. The code example below shows this implemented and is worth noting

includes the GL_ONE_MINUS_SRC_ALPHA as seen in the multiplicative blending example previously.

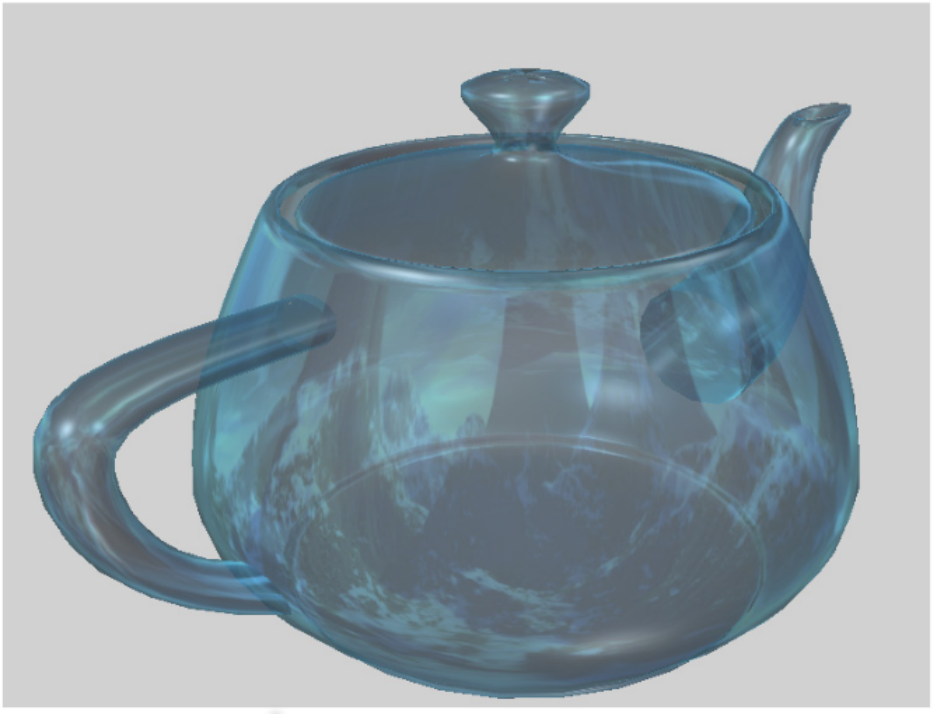
```
drawOpaqueSurfaces();
bindFramebuffer(accumTexture, countTexture);
glDepthMask(GL_FALSE);
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);
bindFragmentShader("...
gl_FragData[0] = vec4(Ci, ai);
gl_FragData[1] = vec4(1);
...}");
drawTransparentSurfaces();
unbindFramebuffer();
glBlendFunc(GL_ONE_MINUS_SRC_ALPHA,
GL_SRC_ALPHA);
bindFragmentShader("...
vec4 accum = texelFetch(accumTexture,
ivec2(gl_FragCoord.xy), 0);
float n = max(1.0, texelFetch(countTexture,
ivec2(gl_FragCoord.xy), 0).r);
gl_FragColor = vec4(accum.rgb / max(accum.a,
0.0001),
pow(max(0.0, 1.0 - accum.a / n), n));
...}", accumTexture, countTexture);
```

5.1.3. The main issue with these methods that has yet to be solved is the problem on unbounded complexity. Since this method is directly linked to hardware feature support, and buffers have obvious physical limitation, extreme complexity can make this method infeasible and require the end user to endure a less accurate rendering..

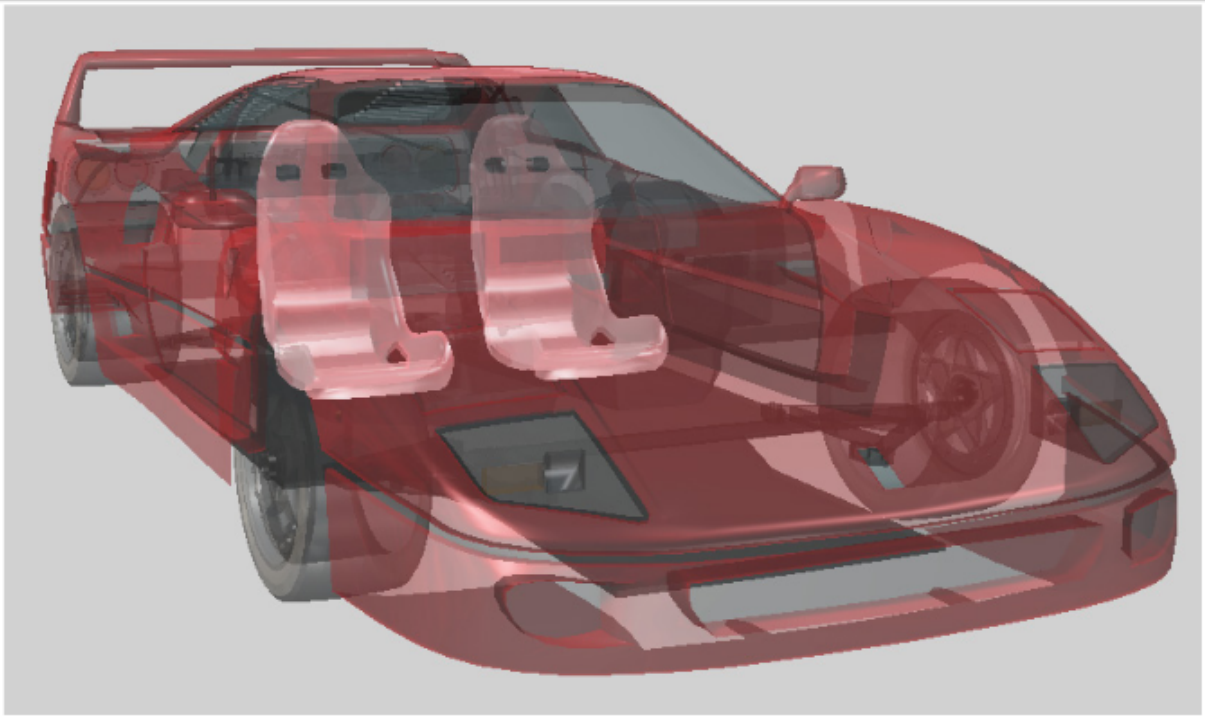
Examples Of Order Independent Transparency



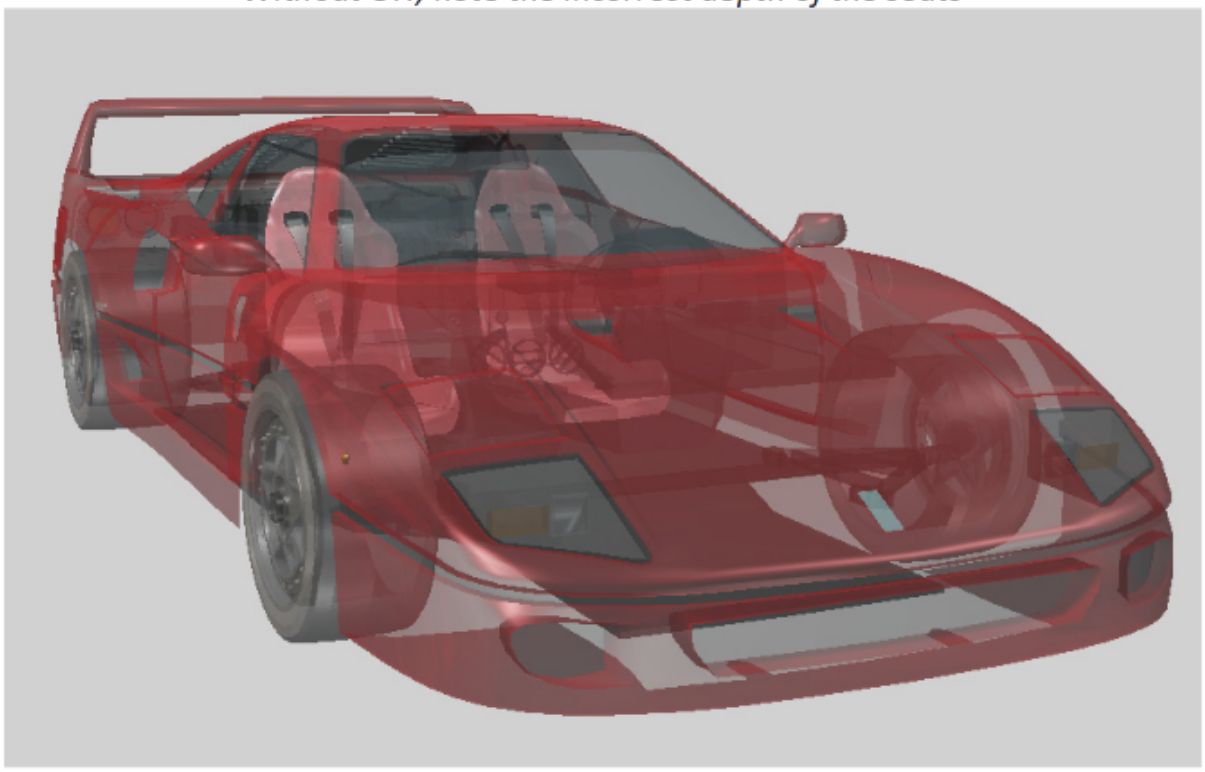
Without OIT



With OIT applied



Without OIT, note the incorrect depth of the seats



With OIT applied

Sources:

White Paper I Order Independent Transparency (OIT) in PTC Creo Parametric 2.0
http://www.amd.com/documents/creooit_white_paper.pdf

OpenGL SuperBible: Order Independent Transparency (2013)
http://www.openglsuperbible.com/2013/08/20/is-order-independent-transparency-really-necessary/

Journal of Computer Graphics Techniques Vol. 2, No. 2, 2013:
Weighted Blended Order-Independent Transparency
http://jcgt.org/published/0002/02/09/paper.pdf

Casual Effects March 9, 2014: Weighted, Blended Order-Independent Transparency
http://casual-effects.blogspot.com/2014/03/weighted-blended-order-independent.html

Order Independent Transparency in OpenGL 4.x
http://on-demand.gputechconf.com/gtc/2014/presentations/S4385-order-independent-transparency-opengl.pdf