5.10

Convincing-Looking Glass for Games

Gabor Nagy

This article presents a few extensions to the algorithms most widely used to render glass objects in real time.

Introduction

Rendering good-looking glass objects at interactive frame rates has long been a challenge. Until we have computer hardware that is fast enough for real-time ray tracing, we must compromise somewhere.

Transparent Objects

There are three main visual properties of glass. A glass object is usually:

- **Transparent.** It lets through some of the light hitting it, making objects behind it partly visible.
- **Refractive.** It refracts light going through it and distorts the environment that shows through.
- **Reflective.** It reflects some of the light hitting it, making the environment show on its surface.

This article mainly deals with the transparent and reflective properties of glass.

Rasterizer, Frame Buffer, Z-Buffer, and Pixel Blending

To draw a transparent object with today's 3D hardware, we usually use the feature called *pixel blending*, or simply *blending*. Pixel blending is implemented in the last stage of a rendering pipeline, in the *pixel renderer*, after rasterization.

The rasterizer does the conversion of a primitive (triangle, line, and so on) into pixels with X and Y screen coordinates and a depth (Z) value.

A simple pixel rendered with Z-buffering enabled will:

- Compute the Z (depth) value of a pixel to be drawn.
- Compare that value with the Z value stored at the corresponding position in the Z-buffer.
- If it is determined that the pixel to be drawn is closer to the viewer (it is in front of the object or objects already drawn at that location), it simply overwrites the pixel color and Z value in their respective buffers; if not, it does not change the frame buffer or Z-buffer at all.

In OpenGL, a smaller Z value means that a pixel is closer to the viewer. Before drawing a scene, the Z-buffer is initialized (cleared) to the maximum Z value at each pixel. This value depends on the bit-depth of the Z-buffer. Note that this value might be the exact opposite, depending on your 3D API and hardware.

Opaque Objects vs. Transparent Objects

Because the standard Z-buffer technique simply overwrites a pixel if it belongs to a surface that is closer to the viewer than the one already drawn, it is only capable of drawing perfectly opaque surfaces. To draw a transparent surface, instead of overwriting the pixel color in the frame buffer with the incoming (source) color, we need to somehow blend the two colors.

In OpenGL, we can use the blending function to perform this task. We can define the blending function by calling:

```
glBlendFunc(sfactor, dfactor);
```

The color to put in the frame buffer is usually determined by:

$$RGB_{result} = RGB_{source} * sfactor + RGB_{destination} * dfactor$$

where RGB_{result} represents the red, green, and blue components to be put in the frame buffer, RGB_{source} is the incoming pixel components, and $RGB_{destination}$ is the value already in the frame buffer at the corresponding pixel.

Depending on the OpenGL version, sfactor and dfactor can have many different predefined constants. For example:

```
glBlendFunc(GL_ONE, GL_ZERO);
```

does a simple overwrite, because:

$$RGB_{result} = RGB_{source} * 1 + RGB_{destination} * 0$$

For another example, if we want to add the current pixel color to the one already in the frame buffer, we can call:

```
glBlendFunc(GL_ONE, GL_ONE);
```

This gives us the following blending formula:

$$RGB_{result} = RGB_{source} * 1 + RGB_{destination} * 1$$

To make pixel blending work in OpenGL, we have to enable it by calling:

```
glEnable(GL_BLEND);
```

For a full description of pixel blending, please refer to your OpenGL manual [Woo97].

Since we have to consider the Z values, we refer to pixels as *RGBZ*.

When rendering a 3D scene with transparent objects, we can have one of the following cases. The pixel currently being rendered ($RGBZ_{source}$) belongs to either an opaque object or a transparent one and:

- Its Z value indicates that it is closer to the viewer than the corresponding pixel ($RGBZ_{destination}$) already in the frame buffer (it is in front of the object or objects already drawn at that location).
- It is further from the viewer than $RGBZ_{destination}$.
- It is at the same distance as $RGBZ_{destination}$.

Drawing Opaque Objects

Let's examine what happens when we draw an opaque object. If $RGBZ_{source}$ is closer to the viewer than $RGBZ_{destination}$, we can simply overwrite the frame buffer with it. However, if $RGBZ_{source}$ is further than $RGBZ_{destination}$, we might still have to draw it if it is "behind" a transparent object! Usually, we can simply avoid this problem by drawing all the opaque objects first.

Drawing Transparent Objects

We use the value A_{source} (*Alpha* or *Opacity* value) to define how opaque the currently drawn pixel is. A_{source}=0.0 means that the pixel is completely transparent; 1.0 means it's completely opaque. If the pixel being drawn ($RGBZ_{source}$) is *in front of* the one in the frame buffer ($RGBZ_{destination}$), we need this formula to determine the resulting color:

Blend Formula A: $RGB_{result} = RGB_{source} * A_{source} + RGB_{destination} * (1.0 - A_{source})$

If $RGBZ_{source}$ is behind $RGBZ_{destination}$, we need this formula:

Blend Formula B: $RGB_{result} = RGB_{source} \cdot (1.0 - A_{destination}) + RGB_{destination} \cdot A_{destination}$

This formula uses the presence of alpha-bitplanes in the frame buffer to keep track of the opacity of each pixel; see [Woo97] for more details. We also have to update the alpha values of the pixels drawn.

Clearly, we have two different blending functions, or two different courses of action to take, depending on whether $RGBZ_{source}$ is in front of $RGBZ_{destination}$ or behind it. Since we can define only one blending function at a time, we have to find a workaround for this problem.

Depth Complexity

The core of the problem is *depth complexity*: the possibly multiple pixels (belonging to different primitives) occupying the same screen position but with different depth values.

A depth complexity of 1 means that there are no primitives overlapping on the screen. We would not even need Z-buffering in this case. When drawing a transparent primitive, we only have to blend it with the background color, using Blend Formula A.

A depth complexity of 2 means that the number of overlapping primitives at each pixel is 2. In this case, an opaque pixel either obscures the background or a transparent pixel or it is behind only *one* opaque or transparent pixel.

Note that the depth complexity of a 3D scene can change when the camera moves.

Fortunately, OpenGL gives us some control over how the Z-buffering is performed. Specifically, we can disable Z overwriting, so when a pixel is drawn, only the RGB values are changed in the frame buffer. Combined with some other features, this allows us to find solutions for most cases.

A Simple Solution

To draw opaque and transparent objects in the same image, we can take a not perfectly correct but simple approach:

- Clear the Z-buffer.
- Draw all the opaque primitives (triangles, lines, and the like) with Z-testing and Z-overwriting enabled.
- Draw transparent primitives with back-face culling enabled (to minimize depth complexity between transparent pixels), Z-overwriting *disabled*, and using Blend Formula A.

This method makes sure that opaque surfaces always obscure transparent ones and that opaque surfaces behind transparent ones show through. It also guarantees that two transparent surfaces always blend correctly if both surfaces have an alpha value of 0.5, because $0.5 = 1.0 - 0.5$, so Blend Formula A and Blend Formula B are equivalent. Therefore, it does not matter if the currently drawn transparent pixel is behind or in front of the one in the frame buffer. For alpha values other than 0.5 or more than two transparent objects behind each other, the results are not accurate but are still acceptable in many cases.

"Simple" Solution #2

This method is designed to solve the problems mentioned in the previous section by making sure that the currently drawn primitive is always in front of the one already in the frame buffer:

- Clear *Z*-buffer.
- Draw all the opaque primitives (triangles, lines, and so on).
- Sort all transparent primitives by depth and draw them in farthest-to-nearest order.

This way we need only Blend Formula A.

The major caveat of this approach is that the depth sorting might cause a significant performance hit, especially if there are many transparent objects in different hierarchy nodes. There are also cases in which a primitive is neither completely in front nor completely behind another (as in Figure 5.10.1), so we need a per-pixel depth sort, which would be extremely computationally expensive.

As long as the depth sorting works correctly, there is no limit to the depth complexity this method can handle. Note that with this method, we don't have to draw the opaque objects first, but doing so could simplify the process.

A Slightly Different Approach

OpenGL lets us choose a *depth function* for *Z*-buffering. The depth function determines which *Z* values pass the *Z*-comparison. In OpenGL, it is usually *less-than*, which means that if a pixel's *Z* value is less than the value in the *Z*-buffer, the pixel is

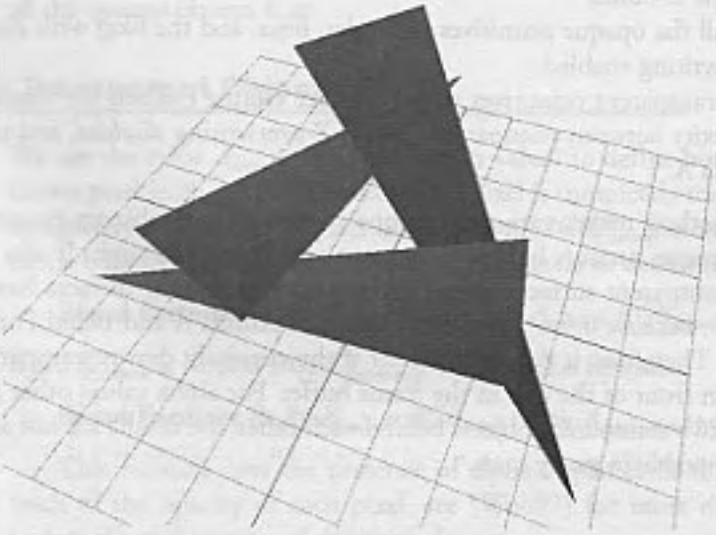

FIGURE 5.10.1. Indeterminate depth order of triangles.

drawn. This allows us to split the drawing of a primitive into two steps, using two different blending formulas. We need the presence of alpha values in the frame buffer for this method.

The process is shown below:

1. Clear *Z*-buffer.
2. Clear the alpha-buffer with value 0 (transparent).
3. Set *Z* function to *greater-than* (draw behind), use Blend Formula A, and draw the first transparent primitive with *Z* overwriting disabled. In addition, write the alpha value of the primitive into the frame buffer so that subsequent pixels behind it are blended correctly.
4. Set *Z* function to *less-than-or-equal* (draw in front), use Blend Formula A, and draw this primitive again with *Z* overwriting enabled. Write the alpha value of the primitive in the frame buffer so that subsequent pixels behind it are blended correctly.
5. Repeat the last two steps for each transparent primitive.
6. Set *Z* function to *greater-than* (draw behind), use Blend Formula B, and draw all opaque primitives with *Z* overwriting disabled. The alpha value to write in the frame buffer is 1.0 (or the maximum integer value).
7. Set *Z* function to *less-than-or-equal* (draw in front) and draw all opaque primitives with *Z* overwriting enabled and blending disabled.

Unfortunately, if more than one pixel is drawn in any given position of the frame buffer (with different depths), the opacity of this pixel can no longer be represented by a single value. It depends on the depth or how many pixels there are in front of the one being drawn. This is caused by the height-field-like nature of the *Z*-buffer: It can store only one depth value on a pixel, with subsequent pixels overwriting the old values. In other words, the *Z*-buffer has a fixed-depth complexity of 1.

Take the example in Figures 5.10.2a and 5.10.2b. Assuming that there are two surfaces drawn in the frame buffer:

- $Surface_1$ – *Alpha:* $A_1 = 0.5$
- $Surface_2$ – *Alpha:* $A_2 = 0.75$

If *P*, the pixel being drawn, is between $Surface_1$ and $Surface_2$, $A_{destination}$ is 0.5 (only $Surface_1$ is in front of *P*). However, if *P* is behind both $Surface_1$ and $Surface_2$, $A_{destination}$ is the *cumulative opacity* of $Surface_1$ and $Surface_2$, which is $A_1 * A_2 = 0.375$.

This approach is slightly more flexible than the one described in our simple solution, without having to depth-sort the transparent primitives, but it has more limitations and is more complicated than Solution #2. Furthermore, the frequent changing of the depth function at each primitive might cause a noticeable performance hit. A slight "tuning" of this method is recommended, depending on the application (especially regarding the modification of the alpha values in the frame buffer).

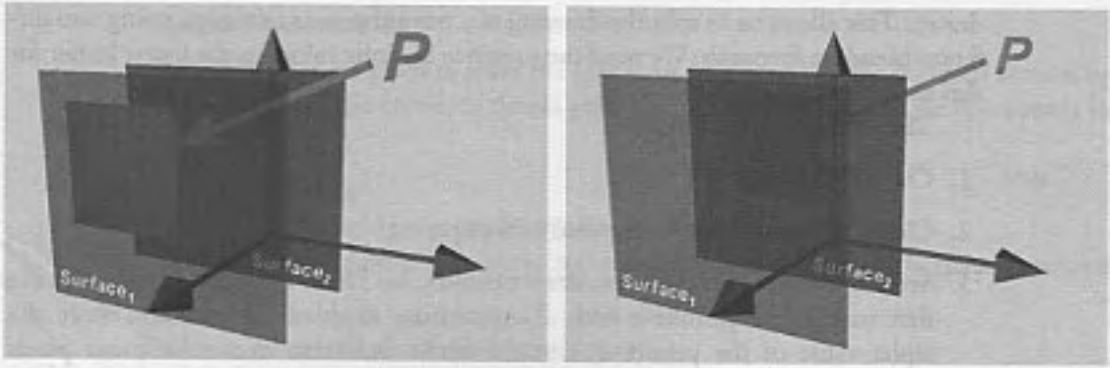

FIGURE 5.10.2. *a:* Effect of a single transparent surface on a pixel behind it. *b:* Cumulative effect of multiple transparent surfaces.

Non-Planar Glass Objects

If we look at a glass bottle or cup, we notice that it appears darker at the edges, where the surface normal starts to point away from the viewer. This is because light coming through the object is refracted at higher angles, so less of it reaches the viewer. We can simulate this effect by illuminating the object with a light source that is always at the same position as the camera (a "head-light"). Such a light source produces less illumination the more the surface normal points away from the viewer. A simple diffuse head-light is very easy to implement and is computationally inexpensive.

Reflections

For simulating reflections, we can use sphere- or cube-environment mapping. OpenGL supports the use of spherical environment maps (with fish-eye images as environment maps). After initializing a texture, we can enable sphere mapping with the following calls:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

There are many excellent articles on this subject, so please refer to them for further details (see References).

Colored Glass

Until now, we referred to the opacity of a surface as a single value. If a surface is behind a transparent one, the surface in front evenly decreased the R, G, and B color

components of the one behind. We can use different opacity values for the R, G, and B components to describe the pigment in a piece of colored glass. This might require the use of multiple drawing passes as described in the following sections.

Putting It All Together

Single-pass Rendering

We can render the glass object in a single pass with:

- An environment map applied as a 2D texture, the texture coordinates computed by a sphere-mapping algorithm
- "MODULATE" texturing algorithm and a head-light
- The proper pixel blending and Z testing set up to draw it as a transparent object (as described in our earlier solutions)

Multipass Rendering

To gain more control over the final appearance, we can perform two rendering passes:

1. Pixel blending and Z testing set up to draw it as a transparent object (as described previously). We can also apply lighting on the object to simulate a diffuse surface on the glass.
2. Render the reflections on top, using additive blending (as in the single-pass case).

With two passes, we can define both the opacity and the reflectivity of an object by changing the blending factors at each pass. We can also apply more complex formulas with multiple passes.

Implementation

For implementation details with OpenGL, please refer to the sample program and the comments in the source code on the included CD that accompanies this book.

To see what this technique looks like in action, take a look at Color Plates 8–11. These images were rendered on a Sony PlayStation 2.

References

[Greene86] Greene, Ned, "Environment Mapping and Other Applications of World Projections," *IEEE Computer Graphics and Applications,* volume 6. number 11, pp. 21–29 November 1986.

[Woo97] Woo, Mason, Neider, Jackie, and Davis, Tom, *OpenGL Programming Guide,* second edition, Addison-Wesley Developers Press, Silicon Graphics, 1997.