Blending

Jaanus Jaggo
The Roadmap

This week

- Construct geometry
  - Define transformations
  - Assign material properties

- Vertex Transformations

- Culling & Clipping
  - Determine front-facing triangles
  - Determine which vertices are visible

- Rasterization
  - Fill the triangle with fragments

- Fragment Shading
  - Calculate correct color values

- Visibility Tests
  - Blending
  - Is the fragment visible?

- Blend together multiple fragments
This lecture

● How to determine which pixel is visible?
● How to represent transparency?
● How to draw semitransparent pixel?
Back to projections

What does the projection matrix do?
Back to projections

What does the projection matrix do?
Projection matrix

$$\begin{pmatrix}
\frac{1}{\text{aspectRatio} \cdot \tan\left(\frac{\alpha}{2}\right)} & 0 & 0 & 0 \\
0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 \\
0 & 0 & \frac{\text{near} + \text{far}}{\text{near} - \text{far}} & -\frac{2 \cdot \text{far} \cdot \text{near}}{\text{near} - \text{far}} \\
0 & 0 & -1 & 0
\end{pmatrix}$$

With projection matrix we transform the z coordinate to range [-1, 1]
Projection matrix - z value

\[ z' = \frac{\text{far} + \text{near}}{\text{far} - \text{near}} + \frac{1}{z} \cdot \left( \frac{-2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}} \right) \]

How do we store this value?
Projection matrix - z value

\[
z' = \frac{\text{far} + \text{near}}{\text{far} - \text{near}} + \frac{1}{z} \cdot \left( -\frac{2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}} \right)
\]

How do we store this value?
We normalize it to range [0,1] by substituting:

\[
z'' = \frac{(z' + 1)}{2} \quad \Rightarrow \quad z' = \frac{\text{far} + \text{near}}{2 \cdot (\text{far} - \text{near})} + \frac{1}{z} \left( -\frac{\text{far} \cdot \text{near}}{\text{far} - \text{near}} \right) + \frac{1}{2}
\]

Finally we multiply it with the max value and round it to an integer.
Depth buffer

While objects are rendered one by one the complete image is stored in the framebuffer. **Framebuffer** is a collection of buffers including color buffer, depth buffer and some other buffers. **Depth buffer** is a 2D array with one integer value for each screen space pixel.
Depth buffer is not linear

But this is a linear representation:
Depth buffer

When new portion of geometry is processed how do we determine which pixels should be drawn to the framebuffer?
Depth testing

Depth test compares each pixel z value to the corresponding pixel z value in the framebuffer.

But what happens if both pixels have the same z value, and when does it happen?
Z-fighting

Example: https://www.youtube.com/watch?v=XjHt-4Z6Pwl&t=1m04s

How to avoid:

- Reducing view frustum depth
- Careful modelling of distant geometry
Some depth buffer applications

- Fog
- Depth of field
- Screen Space Ambient Occlusion (SSAO)
- Shadows
- Soft particles
Fog
Depth of field
Screen Space Ambient Occlusion

https://www.youtube.com/watch?v=-IFxjKT7MXA
Shadow projection
Soft particles

https://www.youtube.com/watch?v=ES0IY_e5Kd8
Color blending

Color blending is a way to mix \textbf{source} and \textbf{destination} colors together to produce third color.

Color:

\[ [R, G, B, A] \] \text{where} \ R, G, B, A \in [0..1].
Alpha

Color consists of R, G, B, A channels
At the fundamental level Alpha has no meaning
Most often it’s used to represent transparency.

Two ways:

- Conventional (straight) alpha
- Premultiplied alpha
Conventional alpha

Transparency is considered as:
● RGB specifies the color of the object
● Alpha specifies how solid it is
Conventional alpha

Blend equation:
Conventional alpha

Blend equation:

\[
\text{blend}(source, \ destination) = (source \cdot source_{alpha}) + (destination \cdot (1 - source_{alpha}))
\]

Problem: if we sample floating point precision we could encounter color bleeding issue:

https://www.youtube.com/watch?v=dU9AXzCabiM
Premultiplied alpha

Transparency is considered as:
- RGB specifies how much color the object contribute to the scene
- Alpha specifies how much it obscures whatever is behind it
Conventional alpha

Blend equation:
Conventional alpha

Blend equation:

\[
\text{blend}(source, destination) = (source \cdot 1) + (destination \cdot (1 - source_{\text{alpha}}))
\]

Advantages:

- no bleed issues
- blend function has one less multiplication
- Normal and additive blending can be done within the same batch
- Smooth transformation from normal to additive

[https://www.shadertoy.com/view/MdfGRX](https://www.shadertoy.com/view/MdfGRX)
Alpha conventions

The information whether an image has straight or premultiplied alpha is not stored in the file itself. Do not mix them together:

http://www.andersriggelsen.dk/glblendfunc.php
Blend function

Generalized formula:

\[
\text{blend}(\text{source}, \text{destination}) = (\text{source} \cdot \text{sourceBlendFactor}) \cdot \text{blendFunction} \cdot (\text{dest} \cdot \text{destinationBlendFactor})
\]

Conventional alpha blending:

- \text{blendFunction} =
- \text{sourceBlendFactor} =
- \text{destBlendFactor} =
Blend function

Generalized formula:

\[
\text{blend}(source, \; destination) = (source \cdot \text{sourceBlendFactor}) \; \text{blendFunction} \; (dest \cdot \text{destinationBlendFactor})
\]

Conventional alpha blending:

- blendFunction = GLFUNC_ADD
- sourceBlendFactor =
- destBlendFactor =
Blend function

Generalized formula:

\[ \text{blend}(\text{source, destination}) = (\text{source} \cdot \text{sourceBlendFactor}) \text{blendFunction} \ (\text{dest} \cdot \text{destinationBlendFactor}) \]

Conventional alpha blending:

- \text{blendFunction} = \text{GL\_FUNC\_ADD}
- \text{sourceBlendFactor} = \text{GL\_SRC\_ALPHA}
- \text{destBlendFactor} =
Blend function

Generalized formula:

\[ \text{blend}(\text{source}, \text{destination}) = (\text{source} \cdot \text{sourceBlendFactor}) \, \text{blendFunction} \, (\text{dest} \cdot \text{destinationBlendFactor}) \]

Conventional alpha blending:

- blendFunction = GL_FUNC_ADD
- sourceBlendFactor = GL_SRC_ALPHA
- destBlendFactor = GL_ONE_MINUS_SRC_ALPHA
Additive blending

- Behaves similarly to the light
- blend(source, dest) = (source * 1) + (dest * 1)
Multiplicative blending

- Useful for shadows
- \( \text{blend}(\text{source, dest}) = (\text{source} \times 0) + (\text{dest} \times \text{source}) \)
Try it yourself

http://www.andersriggelsen.dk/glblendfunc.php

Video recommendation for game designers:

https://www.youtube.com/watch?v=AJdEqssNZ-U
What did you learn today?
What more would you like to know?