BLENDING

Full transparent window   Partially transparent window

Blending in OpenGL is commonly known as the technique to implement **transparency** with objects.

- Transparency is all about objects (or parts of them) not having a solid color, but having a combination of colors from the object itself and any other object behind it, with varying intensity. Transparency allows us to see through objects.
- A colored glass window is a transparent object; the glass has a color of its own, but the resulting color contains the colors of all the objects behind the glass as well.
  - This is where the name blending comes from, since we *blend* several pixel colors (from different objects) to a single color.

Transparent objects can be completely transparent (letting all colors through) or partially transparent (letting colors through, but also some of its own colors).

- The amount of transparency of an object is defined by its alpha value (the 4th component of a color vector).
  - An alpha of 0.0 would give an object 100% transparency, making it appear invisible.
  - An alpha of 0.5 would give an object 50% transparency, blending the colors of the slightly transparent object with the colors of the objects behind it at a 1:1 ratio (50/50).
  - An alpha of 1.0 would give an object 0% transparency, making it opaque.

Discarding Fragments

Some effects don't care about partial transparency, and either want to show something or nothing at all.

- Think of a grass texture. The grass itself is opaque, but the background is fully transparent.

We want to discard the fragments that show the transparent parts of the texture, not storing that fragment into the color buffer.

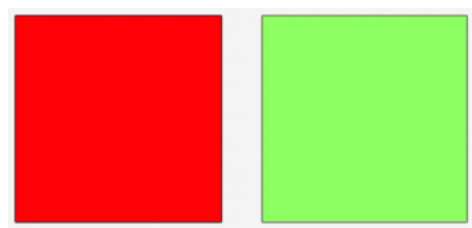
Each of the grass objects is rendered as a single quad with the grass texture attached to it. While not perfect, it is very efficient to draw thin objects as quads rather than complex 3D shapes.

- A quad is just a pair of triangles that form a square using four coplanar vertices.

Add grass quads into your scene at these positions and discard any fragments with an alpha value less than 0.1.

- **NOTE:** If the top of your grass quads appear to have some non-transparent fragments, it is not because of the texture itself, but because of how the texture is being wrapped.
  - When sampling textures at their borders, OpenGL interpolates the border values with the next repeated value of the texture (if you are wrapping the texture using the GL_REPEAT method). This is usually OK, but because we're using transparent values, the top of the texture image gets its transparent value interpolated with the bottom border's solid color value, which is what is causing those non-transparent fragments at the top of the quad to appear.
  - To fix this, simply set the texture wrapping method to GL_CLAMP_TO_EDGE. Do this whenever you use alpha textures that you don't want to repeat.
- Basic Transparency Rendering

Blending



(1.0, 0.0, 0.0, 1.0)   (0.0, 1.0, 0.0, 0.6)

$$\bar{C}_{result} = \begin{pmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 0.6 \end{pmatrix} * 0.6 + \begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 1.0 \end{pmatrix} * (1 - 0.6)$$

To render images with different levels of transparency, we have to enable blending, using `glEnable(GL_BLEND)`.
After blending is enabled, we need to tell OpenGL how it should blend.

Blending in OpenGL happens with the following equation: $\vec{C}_{result} = \vec{C}_{source} * F_{source} + \vec{C}_{destination} * F_{destination}$

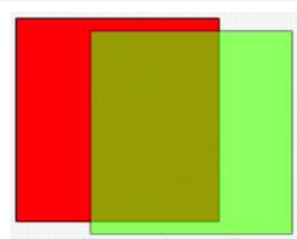
- $\vec{C}_{source}$: The source color vector. This is the color output of the fragment shader.
- $\vec{C}_{destination}$: The destination color vector. This is the color vector that is currently stored in the color buffer.
- F_{source}: The source factor value. Sets the impact of the alpha value on the source color.
- $F_{destination}$: The destination factor value. Sets the impact of the alpha value on the destination color.

After the fragment shader has run and all the tests have passed, this blend equation usedd with the fragment's color output and whatever is currently in the color buffer. The source and destination colors will be automatically set by OpenGL, but the source and destination facto can be set to a value of our choosing.

Take a look at the images to the right. We want to overlay the slightly transparent green square over the opaque red square.

- The red square will be the destination color (and thus should be the first color in the color buffer).
- F_{source} then becomes the alpha value of the source color vector, which is 0.6 (60% opaque).
- Since the green square is contributing 60% of its color to the final fragment color, the red square can only contribute 40%, so $F_{destination}$ then becomes 0.4 (40% opaque).

The resulting color is stored in the color buffer, replacing the previous color.



(0.4, 0.6, 0.0, 0.76)

We tell OpenGL to use the equation using the `glBlendFunc` function.

- `glBlendFunc(GLenum sfactor, GLenum dfactor)` expects two parameters that set the option for source and destination factor.
- Common Factoring Options

To get the blending result of the two squares, we use `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`.

- **NOTE:** If we want to set different options for the RGB and alpha channel individually, we can do so using `glBlendFuncSeparate`.
  - `glBlendFuncSeparate` sets the RGB components as we've set them previously, but only lets the resulting alpha component be influenced by the source alpha's value.
  - Example: `glBlendFuncSeparate(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_ONE, GL_ZERO);`

Right now, the source and destination components are added together, but we could also subtract them by changing the blending mode. glBlendEquation(GLenum mode) allows us to set this operation and has 5 possible outcomes:

- GL_FUNC_ADD: Adds both colors to each other (default): $\vec{C}_{result} = Src + Dst$
- GL_FUNC_SUBTRACT: Substracts both colors from each other: $\vec{C}_{result} = Src - Dst$
- GL_FUNC_REVERSE_SUBTRACT: Subtracts both colors, but reverses order: $\vec{C}_{result} = Dst - Src$
- GL_MIN: Takes the component-wise minimum of both colors: $\vec{C}_{result} = \min(Src,\ Dst)$
- GL_MAX: Takes the component-wise maximum of both colors: $\vec{C}_{result} = \max(Src,\ Dst)$

Rendering Semi-Transparent Textures

Add the above semi-transparent window in place of each grass object grass in your scene.

- [Blending]()

You may notice that the transparent parts of some of the windows actually occlude the windows in the background.

- This is due to an issue with depth testing.
  - When writing to the depth buffer, the depth test does not check if the fragment has transparency or not, so the transparent parts are written to the depth buffer just like any other value. Therefore, some of the windows behind the one you look through might not be rendered when they should.
  - To make sure windows show the windows behind them, we have to draw the windows in the background first, which entails manuall y sorting the windows from furthest to nearest and draw them accordingly ourselves.
    - **NOTE:** With fully transparent objects, we can simply discard the fragments that are transparent rather than blending them, which avoids this issue.

Don't Break the Order

To make blending work for multiple objects, we have to draw the most distant object first and the closest object last. The normal non-blended objects can still be drawn as normal using the depth buffer, so they don't need to be sorted, but they do need to be drawn first before drawing the (sorted) transparent objects.

A general outline for drawing a scene with transparent and non-transparent objects is usually as follows:

1. Draw all opaque objects first.
2. Sort all the transparent objects.
3. Draw all the transparent objects in sorted order.

Create a map that stores the distance between the objects' position vectors and the camera's position vector, with the distances as the keys and the object positions as the values. Then, use that map to render (from furthest to closest) the window objects.

- [Ordered Drawing]()

Sorting objects in your scene can vary in difficulty. There are more advanced techniques, like **order independent transparency**, but these are out of the scope of this chapter.

Vegetation Positions

Tuesday, April 26, 2022 2:11 PM

```
vector<glm::vec3> vegetation;
vegetation.push_back(glm::vec3(-1.5f,  0.0f, -0.48f));
vegetation.push_back(glm::vec3( 1.5f,  0.0f,  0.51f));
vegetation.push_back(glm::vec3( 0.0f,  0.0f,  0.7f));
vegetation.push_back(glm::vec3(-0.3f,  0.0f, -2.3f));
vegetation.push_back(glm::vec3( 0.5f,  0.0f, -0.6f));
```

Basic Transparency Rendering

Tuesday, April 26, 2022 12:38 PM

main.cpp

```
float quadVertices[] = {
   // Positions          // Tex Coords
   -0.5f, -0.5f, 0.0f,      0.0f, 0.0f,       // bottom left
   -0.5f,  0.5f, 0.0f,      0.0f, 1.0f,       // top left
    0.5f,  0.5f, 0.0f,      1.0f, 1.0f,       // top right

   -0.5f, -0.5f, 0.0f,      0.0f, 0.0f,       // bottom left
    0.5f, -0.5f, 0.0f,      1.0f, 0.0f,       // bottom right
    0.5f,  0.5f, 0.0f,      1.0f, 1.0f    // top right
};

...

// ======================================== Grass VAO setup ======================================== \\
unsigned int grassVAO, grassVBO;
glGenVertexArrays(1, &grassVAO);
glGenBuffers(1, &grassVBO);

glBindVertexArray(grassVAO);
glBindBuffer(GL_ARRAY_BUFFER, grassVBO);

glBufferData(GL_ARRAY_BUFFER, sizeof(quadVertices), quadVertices, GL_STATIC_DRAW);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)0);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)(sizeof(float) * 3));

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);

glBindVertexArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Fragment Shader

```
...

void main() {
    vec4 texColor = texture(texture1, TexCoords);
    // if the fragment has a transparency value greater than 90%, we discard it.
    if (texColor.a < 0.1) {
        discard; // GLSL function that discards the current fragment
    }
    FragColor = texColor;
}
```

Make sure you're loading the texture image using the RGBA format, like so:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
```

Render Loop

```
...

grassShader.use();

glUniformMatrix4fv(glGetUniformLocation(grassShader.id, "view"), 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(glGetUniformLocation(grassShader.id, "projection"), 1, GL_FALSE, glm::value_ptr(projection));
glUniform1i(glGetUniformLocation(grassShader.id, "texture1"), 0);

glBindVertexArray(vegetationVAO);
glBindTexture(GL_TEXTURE_2D, grassTexture);
for(unsigned int i = 0; i < vegetation.size(); i++) {
    model = glm::mat4(1.0f);
    model = glm::translate(model, vegetation[i]);
```

```
    shader.setMat4("model", model);
    glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

Common Factoring Options

Tuesday, April 26, 2022 3:33 PM

Option	Value
GL_ZERO	Factor is equal to 0.
GL_ONE	Factor is equal to 1.
GL_SRC_COLOR	Factor is equal to the source color vector $\bar{C}_{source}$.
GL_ONE_MINUS_SRC_COLOR	Factor is equal to 1 minus the source color vector: $1 - \bar{C}_{source}$.
GL_DST_COLOR	Factor is equal to the destination color vector $\bar{C}_{destination}$
GL_ONE_MINUS_DST_COLOR	Factor is equal to 1 minus the destination color vector: $1 - \bar{C}_{destination}$.
GL_SRC_ALPHA	Factor is equal to the $alpha$ component of the source color vector $\bar{C}_{source}$.
GL_ONE_MINUS_SRC_ALPHA	Factor is equal to $1 - alpha$ of the source color vector $\bar{C}_{source}$.
GL_DST_ALPHA	Factor is equal to the $alpha$ component of the destination color vector $\bar{C}_{destination}$.
GL_ONE_MINUS_DST_ALPHA	Factor is equal to $1 - alpha$ of the destination color vector $\bar{C}_{destination}$.
GL_CONSTANT_COLOR	Factor is equal to the constant color vector $\bar{C}_{constant}$.
GL_ONE_MINUS_CONSTANT_COLOR	Factor is equal to 1 - the constant color vector $\bar{C}_{constant}$.
GL_CONSTANT_ALPHA	Factor is equal to the $alpha$ component of the constant color vector $\bar{C}_{constant}$.
GL_ONE_MINUS_CONSTANT_ALPHA	Factor is equal to $1 - alpha$ of the constant color vector $\bar{C}_{constant}$.

NOTE: $\vec{C}_{constant}$, the constant color vector, can be separately set via the `glBlendColor` function.

Blending

Tuesday, April 26, 2022 4:14 PM

main()

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

Fragment Shader

```
#version 330 core

uniform sampler2D texture1;

in vec2 TexCoords;

out vec4 FragColor;

void main() {
    FragColor = texture(texture1, TexCoords);
}
```

Make sure you load the window texture in place of the grass texture. Everything else from the grass rendering code should work with this.

Ordered Drawing

Tuesday, April 26, 2022 4:58 PM

Render Loop

```
// Stores the distance between the camera and window as the key of a map, with the position of the window as the value.
// Maps automatically sort by the key
std::map<float, glm::vec3> sortedWindows;
for (int i = 0; i < windows.size(); ++i) {
   float distance = glm::length(camera.position - windows[i]);
   sortedWindows[distance] = windows[i];
}

// Iterates in reverse over the sortedWindows map, drawing the objects that are further away first.
for (std::map<float, glm::vec3>::reverse_iterator revItr = sortedWindows.rbegin(); revItr != sortedWindows.rend(); ++revItr) {
   model = glm::mat4(1.0f);
   model = glm::translate(model, revItr->second);
   glUniformMatrix4fv(glGetUniformLocation(windowShader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));

   glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

NOTE: This is a fairly restrictive and inefficient solution to this problem. However, it is easy to understand and works with what we have, and that is why it is used here.