DEPTH TESTING

The depth buffer is a buffer that, just like the color buffer (that stores all the fragment colors: the visual output), stores information per fragment and has the same width and height as the color buffer. The depth buffer is automatically created by the windowing system and stores its depth values as 16, 24, or 32 bit floats. Most systems using a depth buffer with precision of 24 bits.
- **NOTE:** The depth buffer is also known as the z-buffer. This is because the "depth" of a fragment is relative to screen space (specifically, the projection-frustum's near and far planes) and the camera is always aligned with the z-axis.

When depth testing is enabled, OpenGL tests the depth value of a fragment against the content of the depth buffer.
- If the depth test passes, the fragment is rendered and the depth buffer is updated with the new depth value.
- If the depth test fails, the fragment is discarded.

Depth testing is done in screen space after the fragment shader has run (and after stencil testing, which is discussed in the next chapter).
- The screen space coordinates relate directly to the viewport defined by OpenGL's `glViewport` function. These coordinates can be accessed via GLSL's built-in `gl_FragCoord` variable in the fragment shader.
  - The x and y components of `gl_FragCoord` represent the fragment's screen space coordinates (with `(0, 0)` being the bottom-left corner).
  - The z component contains the depth value of a fragment, which is compared to the depth buffer's content.
- **NOTE:** Today, most GPUs support **early depth testing**, which allows the depth test to run <u>before</u> the fragment shader runs. Whenever it is clear a fragment isn't going to be visible (as in, it is behind other objects), we can prematurely discard the fragment.
  - This is useful because fragment shaders are expensive to run, so we should avoid running them where possible, which early depth testing helps a lot with.
  - However, if you want to use early depth testing for a fragment, you cannot edit the depth value of the fragment in the fragment shader; the changed depth value is simply ignored.

In 08 - Coordinate Systems, we briefly used `glEnable(GL_DEPTH_TEST)` and `glClear(GL_DEPTH_BUFFER_BIT)`.
- By default, depth testing is not enabled. So, we use `glEnable(GL_DEPTH_TEST)` to enable it. OpenGL now automatically stores the z-values of fragments in the depth buffer if they passed the depth test and discards fragments if they failed.
- If you have depth testing enabled, you should also clear the depth buffer before each frame using `GL_DEPTH_BUFFER_BIT`, otherwise you'll be stuck with the depth values from last frame.

There are some specific situations where you want to continue performing depth tests but <u>stop</u> writing to the depth buffer (e.g. nothing in the scene is moving, so the previous frame's depth buffer will be the same as the current frame's). You can accomplish this with `glDepthMask(GL_FALSE)`.

Depth Test Function

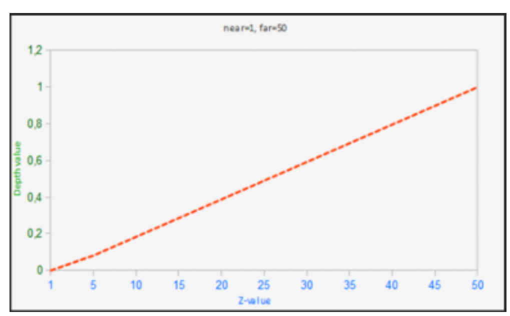
In OpenGL, you can modify the comparison operators used for depth tests. This allows you to control when OpenGL should pass or discard fragments and when to update the depth buffer.
- Example: `glDepthFunc(GL_LESS)`
  - GL_LESS is the default depth function for comparing depth values.
  - The comparison operators that this function accepts can be found here.

For this next part, render at least two objects in your scene so you can see the effect of changing the depth function.

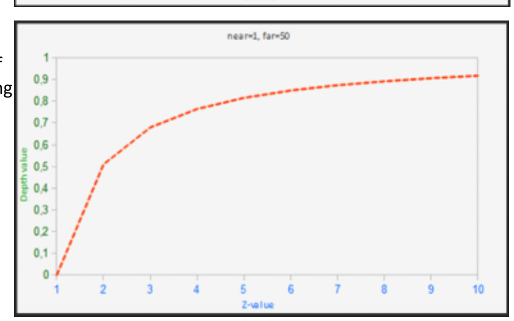
Change the depth function comparison operator to something other than `GL_LESS` *and study how it affects the objects in the scene.*
- The left image uses depth testing with the GL_ALWAYS comparison operator, which is effectively like not having depth testing enabled.
  - The fragments that are drawn last are rendered in front of the fragments drawn before.
- The right image uses depth testing with the GL_LESS comparison operator, which is the default for depth testing.
  - The fragments drawn closer to the camera are rendered in front of the fragments further away.

Depth Value Precision

The depth buffer contains depth values between 0.0 and 1.0, which are the normalized z-positions of the visible fragments relative to their position between the projection-frustum's near and far planes.
- Example: With a **linear depth buffer**, a depth value of 0.5 means the fragment lies in the center of the near and far clipping planes.
  - A linear transformation of the z-value to a depth value would be: $F_{depth} = \frac{z - near}{far - near}$
    - Where near and far are the boundaries of the visible frustum we set for the perspective projection matrix.



In practice, though, a linear depth buffer (like is seen in the image to the right) is almost never used.
- This is because we want to prevent as much **z-fighting** (discussed later in this chapter) as possible for close objects, and essentially push that issue off onto objects which are further away.
- Because of the issue above, a non-linear depth equation that is proportional to `1/z` is used. The result is that we get enormous precision when z is small and much less precision when z is far away.
- Example: With a **non-linear depth buffer**, such as the one in the image under the linear depth buffer, a depth value of 0.5 corresponds to a fragment that lies exactly 1 unit away from the near clipping plane when the length of the clipping plane is 49.
  - The depth value of this fragment is 50 times larger than depth value of a fragment using a linear depth buffer, which means that depth values can be much more precise.
  - A non-linear transformation of the z-value to a depth value would then be: $F_{depth} = \frac{\frac{1}{z} - \frac{1}{near}}{\frac{1}{far} - \frac{1}{near}}$



The equation to transform z-values (from the viewer's perspective) to depth values is embedded within the projection matrix; when we transform vertex coordinates from view space to clip space to screen space, the non-linear equation is applied.

Visualizing the Depth Buffer

Output the depth value of each fragment as a color to display the depth values of all the fragments in the scene.

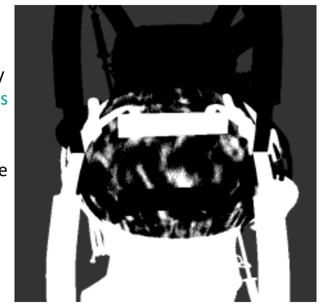
- Fragment Depth Value as Color

We can transform the non-linear depth values of a fragment back to a linear value. All we have to do is reverse the process of projection for the depth values alone. This is done by:
1. Re-transforming the depth values from the range `[0, 1]` to normalized device coordinates (NDC) in the range `[-1, 1]`.
2. Reversing the non-linear equation, $F_{depth} = \frac{\frac{1}{z} - \frac{1}{near}}{\frac{1}{far} - \frac{1}{near}}$, which is applied in the projection matrix.
3. Applying the inversed equation to the resulting depth value.

Change the depth values of all fragments to be calculated linearly.
- Linear Depth Testing

For more math-intensive knowledge on how the projection matrix works, go to: http://www.songho.ca/opengl/gl_projectionmatrix.html

Z-Fighting
Z-fighting occurs when two planes or triangles are so closely aligned to each other that the depth buffer does not have enough precision to figure out which one of the two shapes is in front of the other. The result is that the two shapes continually seem to switch order which causes glitchy, unappealing patterns.



Draw two objects with overlapping triangles, but with different colors, and observe how z-fighting affects the fragments.
- **NOTE:** If you draw two of the same object with the exact same configurations (model matrix, vertex shaders, etc.), you won't see any z-fighting. This is because after the depth values of the first object's fragments are written to the depth buffer, none of the fragments of the second object have a depth less than the respective fragment of the first object. And, since the fragments of both triangles of the objects have the exact same depth values, floating point precision won't cause the fragments to fight for precedence. Z-fighting only occurs when fragments of two triangles lie on similar enough planes that the precision of a floating point number is not accurate enough to determine its fragments' exact depths.

Prevent Z-Fighting
Z-fighting cannot be completely prevented, but there are some measures you can take to mitigate z-fighting in your scene.
1. The most important trick is to never place objects too close to each other in a way that some of their triangles closely overlap.
   - This may require manual intervention, which can be costly, depending on the size of the scene and the number of objects in it.
2. Another trick is to set the near plane as far as possible.
   - If more you move the near plane further away from the viewer, the precision of depth testing becomes significantly greater over the entire frustum range.
   - However, setting the near plane too far away from the viewer can cause clipping of near objects. So, this is something you'll have to experiment with.
3. Another great trick is to use a higher precision depth buffer.
   - Most depth buffers have a precision of 24 bits, but most GPUs nowadays support 32 bit depth buffers, which significantly increase precision.
   - However, this comes at the cost of performance.

There are many other anti-z-fighting tactics out there that are much more difficult to implement, but these three will be sufficient for most cases.

glDepthFunc Comparison Operators

Wednesday, April 20, 2022 4:04 PM

Function	Description
GL_ALWAYS	The depth test always passes.
GL_NEVER	The depth test never passes.
GL_LESS	Passes if the fragment's depth value is less than the stored depth value.
GL_EQUAL	Passes if the fragment's depth value is equal to the stored depth value.
GL_LEQUAL	Passes if the fragment's depth value is less than or equal to the stored depth value.
GL_GREATER	Passes if the fragment's depth value is greater than the stored depth value.
GL_NOTEQUAL	Passes if the fragment's depth value is not equal to the stored depth value.
GL_GEQUAL	Passes if the fragment's depth value is greater than or equal to the stored depth value.

Fragment Depth Value as Color

Wednesday, April 20, 2022 4:31 PM

<u>default.frag</u>

```
void main() {
    FragColor = vec4(vec3(gl_FragCoord.z), 1.0);
}
```

NOTE: If you remember, most depth values will be close to 1.0, with only the fragments within the first ~2% on the viewing frustum being under 0.5. Because of this, you must move your camera pretty close to and object in order to see a visible difference in the color of the fragments.

- If you want a more visible change in color, try making the near clipping plane further away so you can see more of the scene, and make the far clipping plane closer to the near clipping plane (handled in the projection matrix creation).

Linear Depth Testing

Wednesday, April 20, 2022 4:49 PM

<u>default.frag</u>

```
// Change these to match your near/far clipping planes
float near = 0.1;
float far = 100.0;

void main() {
    float z = (gl_FragCoord.z * 2.0) - 1.0; // Back to ndc
    float linearDepth = (2.0 * near * far) / (far + near - z * (far - near)); // Don't worry if you don't understand this

    FragColor = vec4(vec3(linearDepth / far), 1.0); // Dividing by far normalizes the values to the range [0.0, 1.0]
}
```

You should notice objects becoming whiter the further from the camera they are, and blacker the closer they come to the camera.