Computer Graphics II
– Depth & Stencil Testing

Kai Lawonn

Introduction

- In the lecture about coordinate systems, we rendered a 3D box using the depth buffer to prevent faces rendering to the front while they're behind other faces
- Now focusing more on these depth values and the corresponding depth buffer (or z-buffer) and how it actually determines if a fragment is indeed behind other fragments

Introduction

- Depth-buffer is a buffer like the color buffer, instead of storing the fragment colors it stores the depth
- Depth per fragment with (usually) the same width and height as the color buffer
- It is automatically created by the windowing system and stores its depth values as 16, 24 or 32 bit floats (mostly a precision of 24 bits)
- When depth testing is enabled: OpenGL tests depth value of a fragment against the content of the depth buffer (depth test)
- If it passes → depth buffer is updated with new depth value, otherwise discard fragment

Introduction

- Depth testing is done in screen space after the fragment shader (and after the stencil test → later)
- The screen space coordinates relate directly to the viewport defined by OpenGL's glViewport function → accessed with GLSL's gl_FragCoord variable in the fragment shader
- gl_FragCoord.xy represent the fragment's screen-space coordinates (with (0,0) being the bottom-left corner)
- gl_FragCoord.z contains the actual depth value of the fragment → this value is compared to the depth buffer's content

Most GPUs support a hardware feature called *early depth testing* (OpenGL 4.2)

It allows the depth test to run before the fragment shader runs

If it is clear that a fragment is not visible (e.g., behind other objects) → prematurely discard

Fragment shaders usually expensive so avoid it running wherever we can

Restriction on the fragment shader for early depth testing: do not change the fragment's depth value (OpenGL won't be able to figure out the depth value before the fragment shader)

Depth Test

- Depth testing is disabled by default so to enable depth testing we need to enable it with the GL_DEPTH_TEST option:

```
glEnable(GL_DEPTH_TEST);
```

- If it is enabled OpenGL automatically stores fragments' z-values in the depth buffer if they passed the depth test and discards fragments if they failed the depth test accordingly

Depth Test

- If depth testing is enabled, the depth buffer should be cleared before each render iteration using the GL_DEPTH_BUFFER_BIT, otherwise we are stuck with the written depth values from last render iteration:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Depth Test

- If you want to perform the depth test on all fragments and discard them accordingly, but **not** update the depth buffer (read-only depth buffer), OpenGL allows disabling writing to the depth buffer by setting its depth mask to GL_FALSE

```
glDepthMask(GL_FALSE);
```

- Note that this only has effect if depth testing is enabled

Depth Test Function

- OpenGL allows modifying the comparison operators for the depth test → control when OpenGL should pass or discard fragments and when to update the depth buffer
- We can set the comparison operator (or depth function) by calling glDepthFunc:

```
glDepthFunc(GL_ALWAYS);
```

Depth Test Function

- The function accepts several comparison operators:

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.

Depth Test Function

- By default the depth function GL_LESS is used

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.

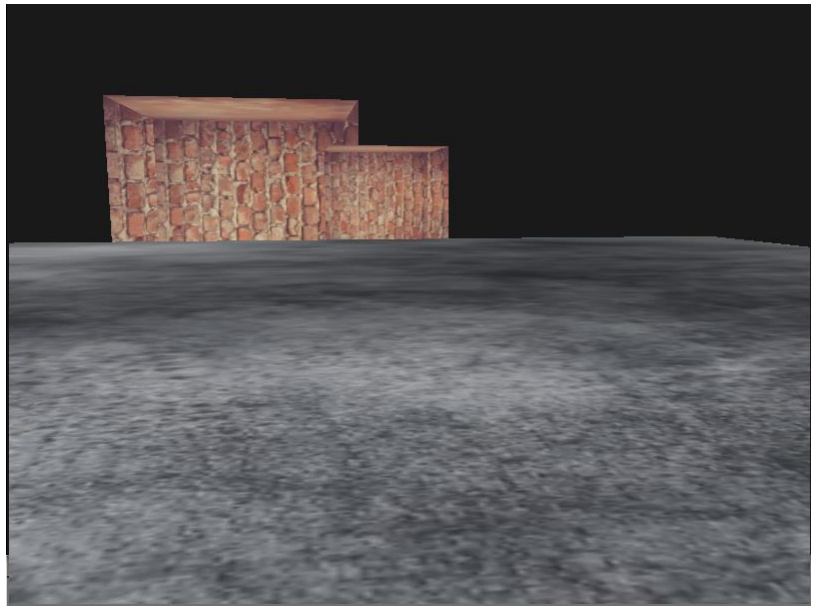
Depth Test Function

- Let's experiment with changing the depth function
- Use a setup with two textured cubes sitting on a textured floor (no lighting) → 4.advanced_opengl__1.1.depth_testing
- Change the depth function to GL_ALWAYS:

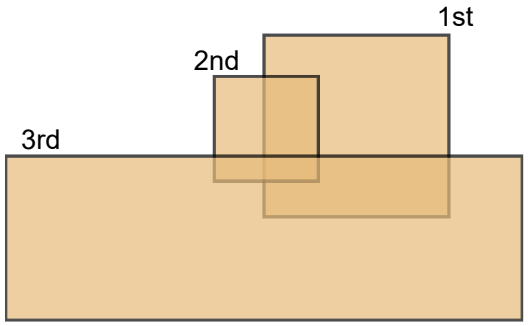
```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_ALWAYS);       // always pass the depth test
                              // (same effect as glDisable(GL_DEPTH_TEST))
```

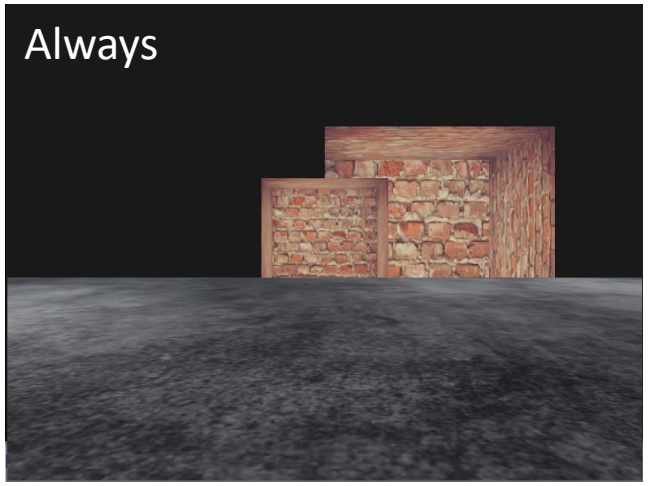
Depth Test Function

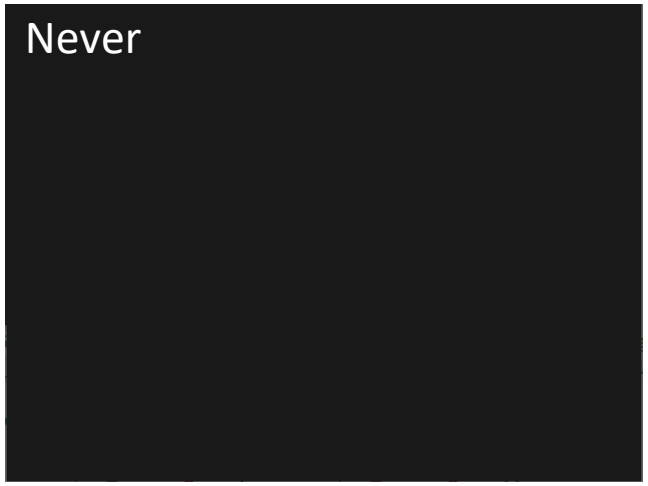
- Same behavior as didn’t enable depth testing
- Depth test always passes → fragments drawn last are rendered in front of the fragments that were drawn before
- Drawn the floor plane as last the plane’s fragments overwrite each of the container’s fragments

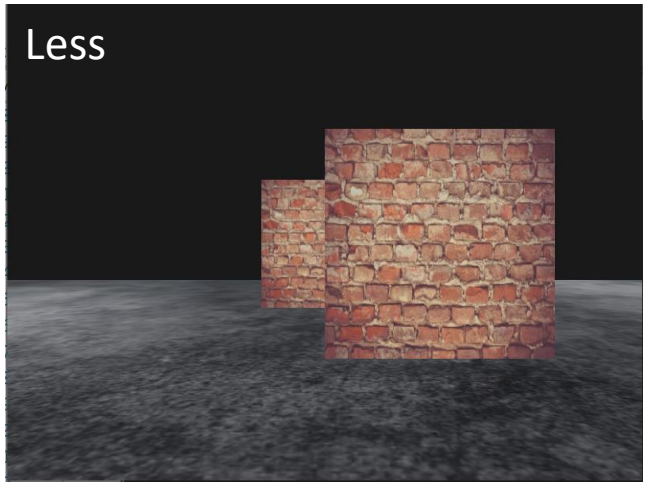


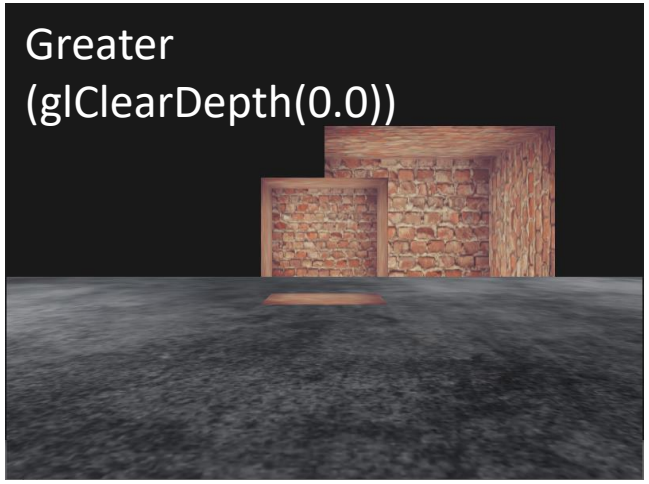
Depth Test Function

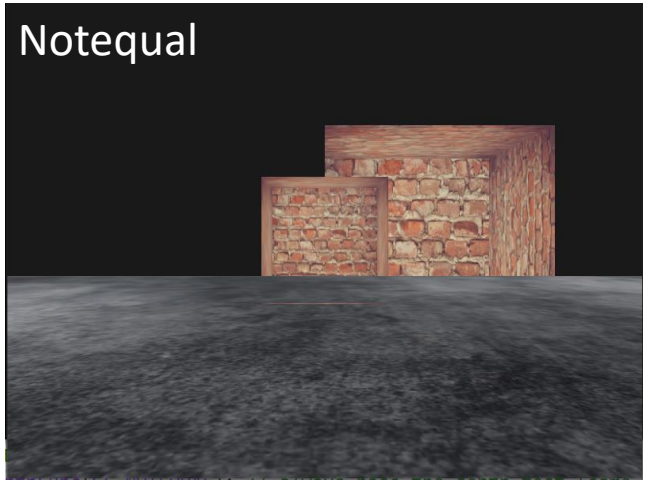












Depth Value Precision

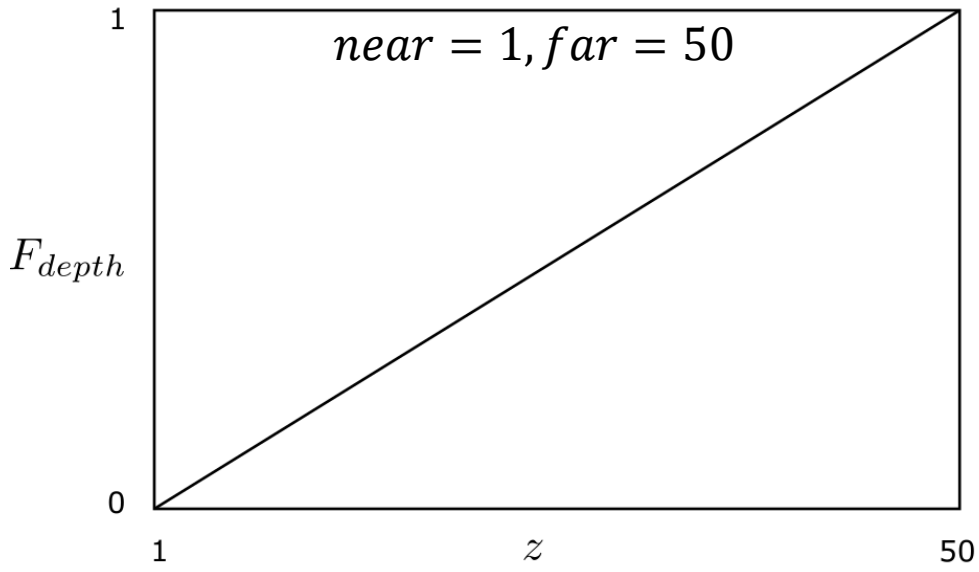
- The depth buffer contains depth values in $[0,1]$ and compares it with z-value of all the objects in the scene as seen from the viewer
- Z-values in view space is any value between the projection frustum's near and far value
- Need to transform these view-space z-values to the range of $[0,1]$ and one way is to linearly transform them:

$$F_{depth} = \frac{z - near}{far - near}$$

Depth Value Precision

$$F_{depth} = \frac{z - near}{far - near}$$

- Here *near* and *far* are the values used in the projection matrix to set the visible frustum (Lecture Coordinate Systems)
- Equation transforms a *z value* within the frustum to $[0,1]$
- The relation between the z-value and its corresponding depth value is presented in the following graph:



Depth Value Precision

$$F_{depth} = \frac{z - near}{far - near}$$

- Linear depth buffer like this is almost never used
- For correct projection properties a non-linear depth equation proportional to 1/z is used
- Enormous precision when z is small and much less precision when z is far away
- The linear equation doesn’t take this into account

Depth Value Precision

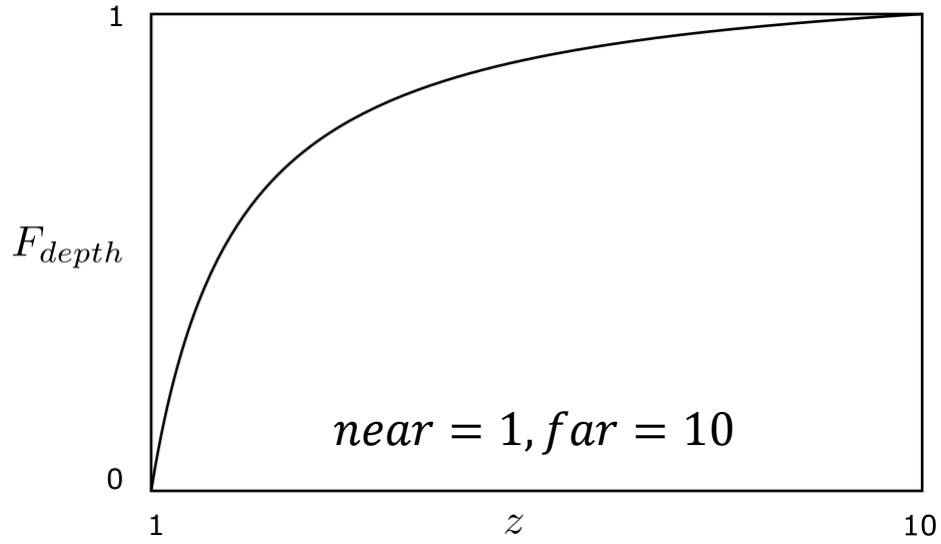
- Non-linear function is proportional to $1/z$, z in $[1,2]$ → between 1 and 0.5 (half of the precision a float provides us → enormous precision at small z-values)
- Z-values between 50.0 and 100.0 account for only 2% of the float's precision (what we want)
- Equation, that also takes near and far distances into account:

$$F_{depth} = \frac{1/z - 1/near}{1/far - 1/near}$$

Depth Value Precision

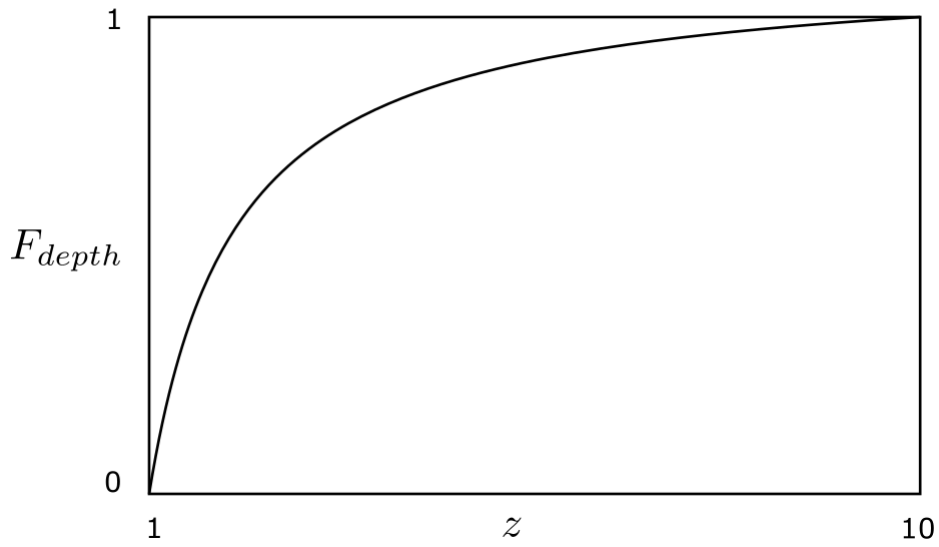
$$F_{depth} = \frac{1/z - 1/near}{1/far - 1/near}$$

- Note: The values in the depth buffer are not linear in screen-space (they are linear in view-space before the projection matrix is applied)
- A value of 0.5 in the depth buffer does not mean the object's z-values are halfway in the frustum (it is actually quite close to the near plane)
- Non-linear relation between the z-value and the resulting depth:



Depth Value Precision

- Depth values are greatly determined by the small z-values → enormous depth precision to the objects close by
- The equation to transform z-values (from the viewer's perspective) is embedded within the projection matrix
- The effect of this non-linear equation quickly becomes apparent when we try to visualize the depth buffer



Note that all equations give a depth value close to 0.0 when the object is close by and a depth value close to 1.0 when the object is close to the far plane.

Visualizing the Depth Buffer

Visualizing the Depth Buffer

- Built-in gl_FragCoord.z contains the depth value of that particular fragment
- To output this value as a color → return a fragment's depth-based color

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

Visualizing the Depth Buffer

- Run the program and you notice that everything seems white
- So why aren't any of the depth values closer to 0.0 and thus darker?

Visualizing the Depth Buffer

- Depth values in screen space are non-linear (high precision for small z; low precision for large z)
- Depth increases rapidly over distance → almost all vertices are close to 1.0
- Close to objects → colors getting darker (z-values becoming smaller)

Linearize the Depth Buffer

- Objects close by have a much larger effect on the depth value than objects far away
- We can transform the non-linear depth values to a linear one → have to reverse the process of projection:
  1. Re-transform depth values from $[0,1]$ to NDCs in $[-1,1]$ (clip space)
  2. Reverse the non-linear equation
  3. Apply this inversed equation to the resulting depth value →result is then a linear depth value

Linearize the Depth Buffer

1. Re-transform depth values from $[0,1]$ to NDCs in $[-1,1]$ (clip space)

```
float z = depth * 2.0 - 1.0; // back to NDC
```

Linearize the Depth Buffer

2. Reverse the non-linear equation

$$proj = \begin{pmatrix} \frac{2n}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2n}{top-bottom} & -\frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2 \cdot f \cdot n}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Linearize the Depth Buffer

2. Reverse the non-linear equation

$$proj = \begin{pmatrix} \frac{2n}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2n}{top-bottom} & -\frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2 \cdot f \cdot n}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$$(proj \cdot v).zw = \begin{pmatrix} -\frac{f+n}{f-n} \cdot v.z - \frac{2 \cdot f \cdot n}{f-n} \cdot v.w \\ -v.z \end{pmatrix}$$

Linearize the Depth Buffer

2. Reverse the non-linear equation

$$(proj \cdot v).zw = \begin{pmatrix} -\frac{f+n}{f-n} \cdot v.z - \frac{2 \cdot f \cdot n}{f-n} \cdot v.w \\ -v.z \end{pmatrix}$$

(gl_FragCoord.z=) $$z = \frac{-\frac{f+n}{f-n} \cdot v.z - \frac{2 \cdot f \cdot n}{f-n} \cdot v.w}{-v.z}$$

Linearize the Depth Buffer

2. Reverse the non-linear equation

$$z = \frac{-\frac{f+n}{f-n} \cdot v.z - \frac{2 \cdot f \cdot n}{f-n} \cdot v.w}{-v.z}$$

$$= \frac{f+n}{f-n} + \frac{2 \cdot f \cdot n}{v.z(f-n)} \qquad (v.w = 1)$$

$$v.z \cdot \left(z - \frac{f+n}{f-n} \right) = \frac{2 \cdot f \cdot n}{f-n}$$

$$v.z = \frac{\frac{2 \cdot f \cdot n}{f-n}}{\frac{z(f-n)-(f+n)}{f-n}}$$

Linearize the Depth Buffer

2. Reverse the non-linear equation

$$v.z = \frac{\frac{2 \cdot f \cdot n}{f-n}}{\frac{z(f-n)-(f+n)}{f-n}}$$

$$= \frac{2 \cdot f \cdot n}{z(f-n)-(f+n)}$$

Linearize the Depth Buffer

2. Reverse the non-linear equation

$$v.z = \frac{\frac{2 \cdot f \cdot n}{f-n}}{\frac{z(f-n)-(f+n)}{f-n}}$$

$$= \frac{2 \cdot f \cdot n}{z(f-n)-(f+n)}$$

$$\frac{NDC_Z(-n)}{n} = -1$$

$$\frac{NDC_Z(-f)}{f} = 1$$

Linearize the Depth Buffer

2. Reverse the non-linear equation

$$v.z = \frac{\frac{2 \cdot f \cdot n}{f-n}}{-\frac{z(f-n)-(f+n)}{f-n}}$$

$$= \frac{2 \cdot f \cdot n}{-(z(f-n)-(f+n))}$$

$$= \boxed{\frac{2 \cdot f \cdot n}{(f+n)-z(f-n)}}$$

$$\frac{NDC_Z(-n)}{n} = -1$$

$$\frac{NDC_Z(-f)}{f} = 1$$

Linearize the Depth Buffer

$$v.z = \frac{2 \cdot f \cdot n}{(f+n) - z(f-n)}$$

2. Reverse the non-linear equation

```
float z = (2.0 * far * near) / (far + near - z * (far - near));
```

Linearize the Depth Buffer

$$v.z = \frac{2 \cdot f \cdot n}{(f+n) - z(f-n)}$$

2. Reverse the non-linear equation

```
#version 330 core
out vec4 FragColor;

float near = 0.1;
float far = 100.0;
float LinearizeDepth(float depth)
{
    float z = depth * 2.0 - 1.0; // back to NDC
    return (2.0 * far * near) / (far + near - z * (far - near));
}

void main()
{
    float depth = LinearizeDepth(gl_FragCoord.z) / far; // /far -> [0.001,1]
    FragColor = vec4(vec3(depth), 1.0);
}
```

F5…

- … colors are mostly black, linear depth from near plane (0.1) to far plane (100) → far away

Z-Fighting

- Visual artifact occur when two planes/triangles are closely aligned → depth buffer not enough precision to figure out which in front
- Causes glitchy patterns: z-fighting

Z-Fighting

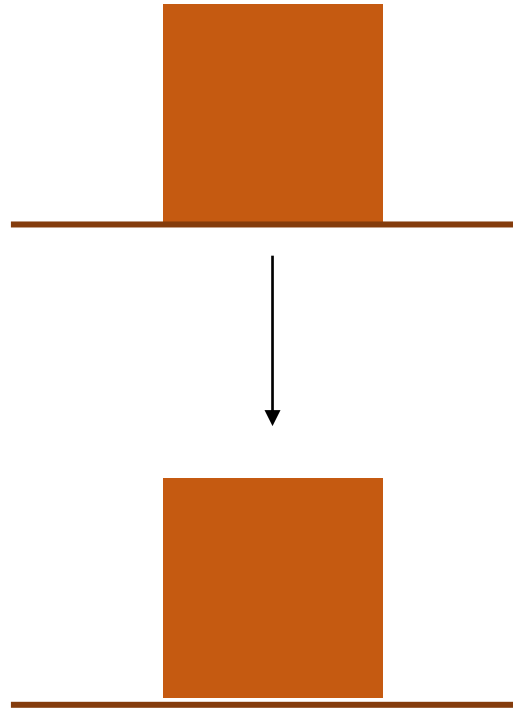
- Box placed at the exact height of the floor (bottom plane of the box is coplanar with the floor plane)
- Depth values of both planes are same → resulting depth test has no way of figuring out which is the right one
- Moving the camera inside one of the boxes yield the glitchy effect:

Z-Fighting

- Z-fighting is a common problem
- Generally stronger when objects are at a further distance (depth buffer has less precision at larger z-values)
- Z-fighting cannot be completely prevented
- A few tricks that will help to mitigate or completely prevent z-fighting

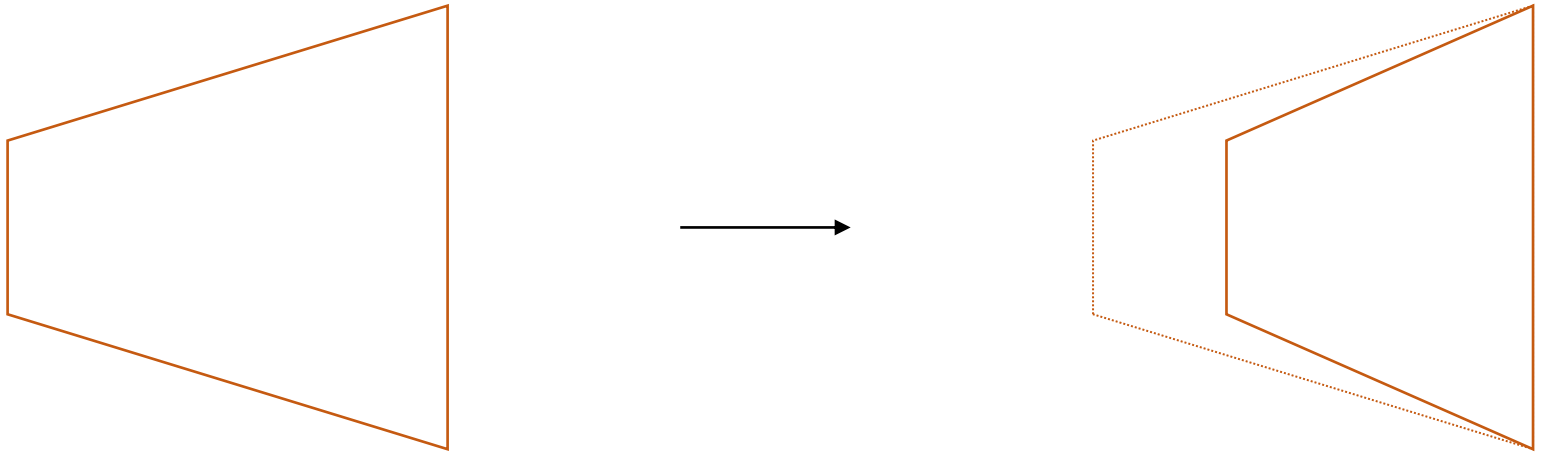
Prevent Z-Fighting

- Most important trick: *never place objects too close to each other (triangles closely overlap)*
- Small offset is hardly noticeable by a user → completely remove z-fighting
- In our case: move the boxes slightly in the positive y-direction (not be noticeable and completely reduce the z-fighting)
- Requires manual intervention of each of the objects and thorough testing



Prevent Z-Fighting

- Second trick: *set the near plane as far as possible*
- Precision is extremely large close to the near plane (moving near plane farther improves the precision over the frustum range)
- Setting the near plane too far causes clipping of near objects (matter of tweaking and experimentation)



Prevent Z-Fighting

- Another trick at the cost of some performance: *use a higher precision depth buffer*
- Most depth buffers have a precision of 24 bits, but most cards nowadays support 32 bits → increases the precision significantly
- At the cost of some performance we get much more depth precision, reducing z-fighting

Prevent Z-Fighting

- These techniques are the most common and easy-to-implement anti z-fighting techniques
- Other techniques require a lot more work and still will not completely disable z-fighting
- If you use the proper combination of these tricks you probably will not need to deal with z-fighting

Stencil Testing

Stencil Testing

- After the fragment shader processed the fragment, a stencil test is executed that has the possibility of discarding fragments
- Then the remaining fragments get passed to the depth test (possibly discard even more fragments)
- Stencil test is based on another buffer → stencil buffer
- It (usually) contains 8 bits per stencil value that (= 256 different stencil values per pixel/fragment)
- Can set these stencil values to preferable values to discard or keep fragments whenever a particular fragment has a certain stencil value

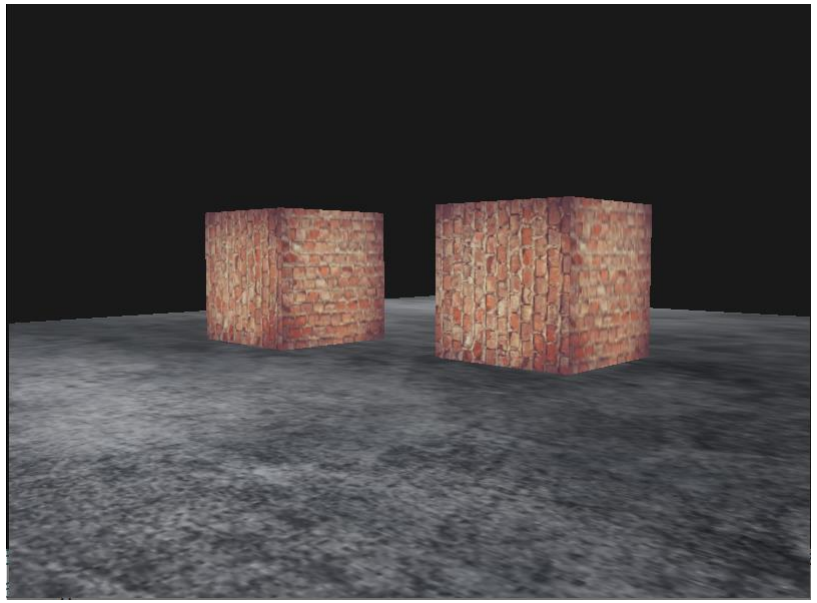
Each windowing library needs to set up a stencil buffer

GLFW does this automatically

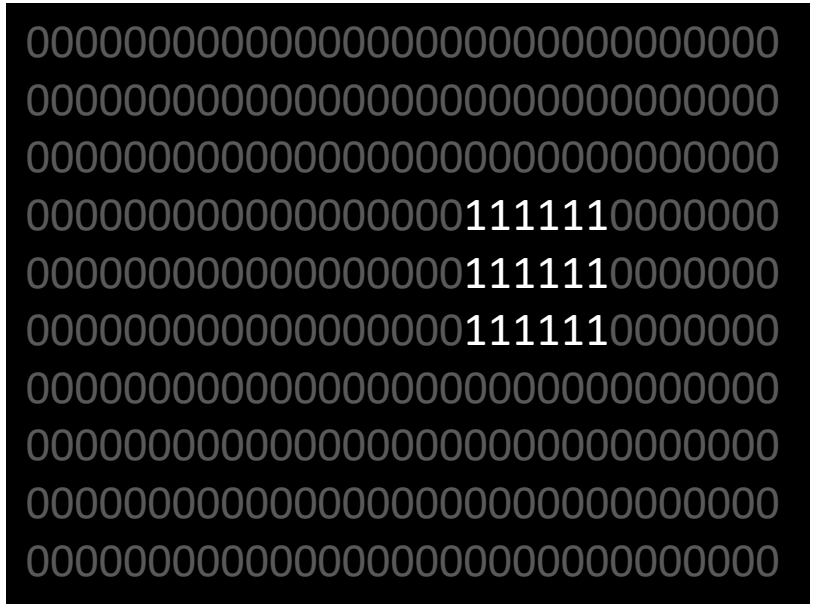
Other windowing libraries might not create a stencil library by default

Stencil Testing

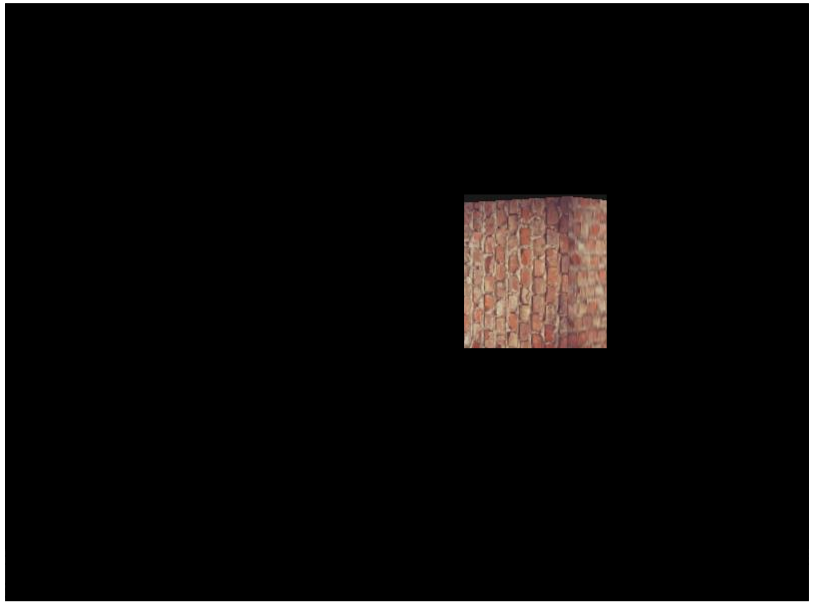
- A simple example of a stencil buffer (pixels not-to-scale):



Color buffer



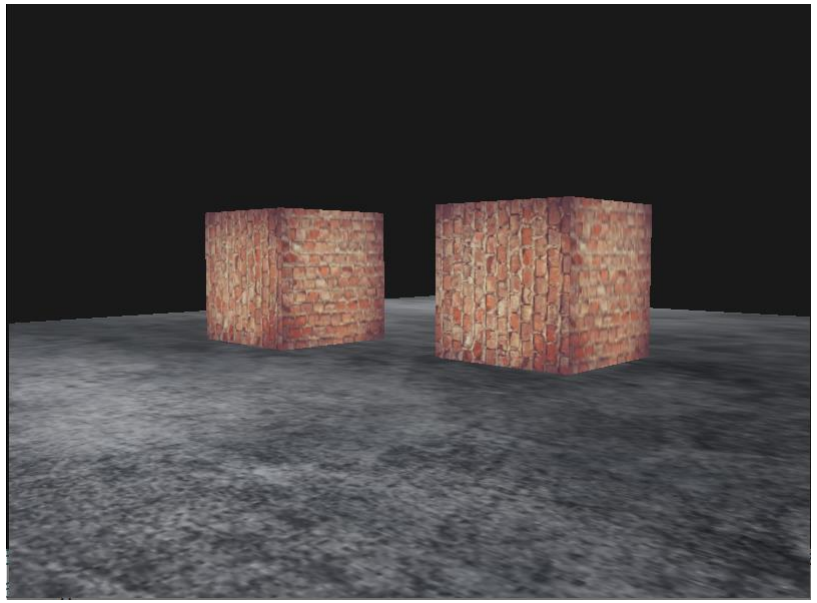
Stencil buffer



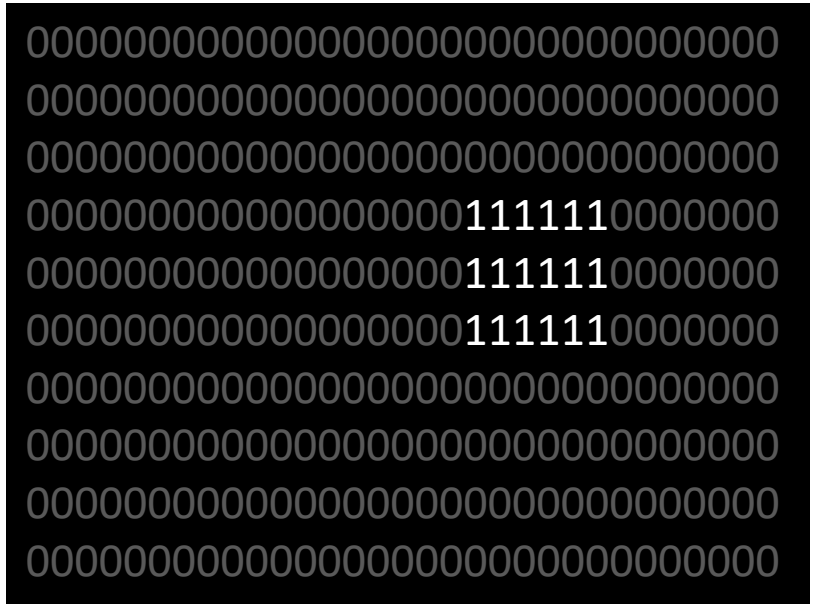
After stencil test

Stencil Testing

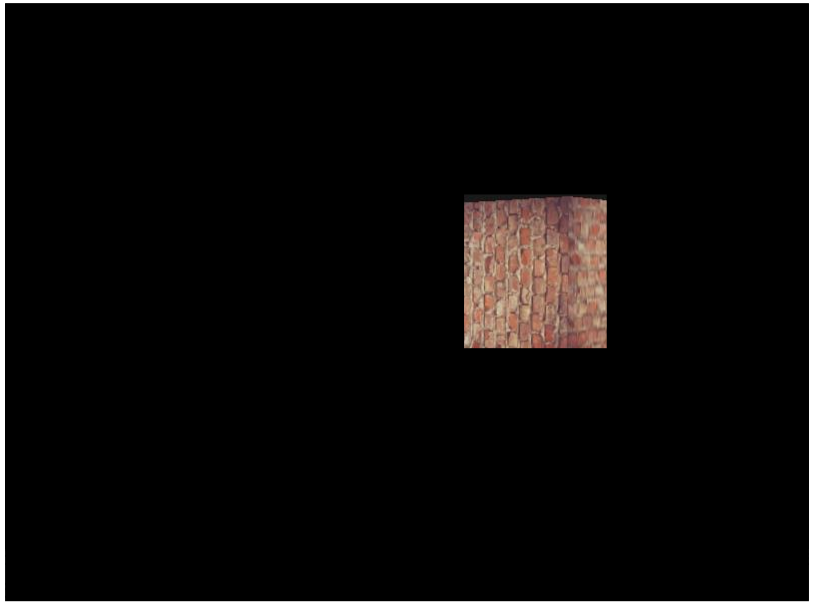
- Stencil buffer cleared with 0s, then rectangle of 1s is set
- The fragments of the scene are then only rendered (the others are discarded) wherever the stencil value of that fragment contains a 1



Color buffer



Stencil buffer



After stencil test

Stencil Testing

- Stencil buffer operations allow setting stencil buffer at specific values (wherever we are rendering fragments)
- Changing the content of the stencil buffer while rendering: we are *writing* to the stencil buffer
- Can *read* these values to discard or pass certain fragments, general outline is usually as follows:
  - Enable writing to the stencil buffer
  - Render objects, updating the content of the stencil buffer
  - Disable writing to the stencil buffer
  - Render (other) objects, this time discarding certain fragments based on the content of the stencil buffer

Stencil Testing

- By using the stencil buffer we can thus discard certain fragments based on the fragments of other drawn objects in the scene
- Enable stencil testing by enabling GL_STENCIL_TEST
- From that point on, all rendering calls will influence the stencil buffer:

```
glEnable(GL_STENCIL_TEST);
```

Stencil Testing

- Note to clear the stencil buffer each iteration (like color and depth):

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

Stencil Testing

- Like depth testing's glDepthMask function, there is an equivalent for the stencil buffer
- glStencilMask allows setting a bitmask that is ANDed with the stencil value
- By default, it is set to a bitmask of all 1s (unaffecting the output), but setting to 0x00 all the stencil values written to the buffer end up as 0s (equivalent to depth testing's glDepthMask(GL_FALSE)):

```
glStencilMask(0xFF);
glStencilMask(0x00);
```

Stencil Testing

- Mostly we will just writing 0x00 or 0xFF as the stencil mask, but it’s good to know there are options to set custom bit-masks

Stencil Testing

- Like depth testing, we can set when a stencil test should pass or fail and how it should affect the stencil buffer
- There are two functions we can use to configure stencil testing: glStencilFunc and glStencilOp

Stencil Testing

- glStencilFunc(GLenum func, GLint ref, GLuint mask) has three parameters:
  - func: sets the stencil test function; applied to the stored stencil value and the glStencilFunc's ref value: GL_NEVER, GL_LESS, GL_LEQUAL, GL_GREATER, GL_GEQUAL, GL_EQUAL, GL_NOTEQUAL and GL_ALWAYS
  - ref: specifies reference value for the stencil test; stencil buffer's content is compared to this value
  - mask: specifies a mask that is ANDed with reference value and stored stencil value before the test compares them (Initially set to all 1s)

Stencil Testing

- In the case of the simple stencil example from the beginning, the function would be set to:

```
glStencilFunc(GL_LEQUAL, 1, 0xFF);
```

- Tells OpenGL whenever the stencil value of a fragment is equal (GL_EQUAL) to the reference value 1 the fragment passes the test and is drawn, otherwise discarded
- glStencilFunc only describes what OpenGL should do with the content of the stencil buffer, not how we can actually update the buffer
- That is where glStencilOp comes in

Stencil Testing

- glStencilOp(GLenum sfail, GLenum dpfail, GLenum dppass) contains three options:
  - sfail: action to take if the stencil test fails
  - dpfail: action to take if the stencil test passes, but the depth test fails
  - dppass: action to take if both the stencil and the depth test pass

Stencil Testing

- Then for each of the options you can take any of the following actions:

Action	Description
GL_KEEP	Currently stored stencil value is kept
GL_ZERO	Stencil value is set to 0
GL_REPLACE	Stencil value is replaced with reference value set with glStencilFunc
GL_INCR	Stencil value is increased by 1 if it is lower than the maximum value
GL_INCR_WRAP	Same as GL_INCR, but wraps it back to 0 if maximum value is exceeded
GL_DECR	Stencil value is decreased by 1 if it is higher than the minimum value
GL_DECR_WRAP	Same as GL_DECR, but wraps it to the maximum value if lower than 0
GL_INVERT	Bitwise inverts the current stencil buffer value

Stencil Testing

- By default glStencilOp is set to (GL_KEEP, GL_KEEP, GL_KEEP) so independent of any test, the stencil buffer keeps its values → does not update the stencil buffer
- Want to write to the stencil buffer you need to specify at least one different action for any of the options
- glStencilFunc and glStencilOp allows to precisely specify when and how to update the stencil buffer (can also specify when the stencil test should pass or not e.g. when fragments should be discarded)

Object Outlining

- As an example, to demonstrate the stencil test, we implement a particular useful feature: the object outlining

Object Outlining

- Useful effect for selecting objects
- The routine for outlining your objects is as follows:

1. Set the stencil func to GL_ALWAYS before drawing the (to be outlined) objects, updating the stencil buffer with 1s wherever the objects' fragments are rendered
2. Render the objects
3. Disable stencil writing and depth testing
4. Scale each of the objects by a small amount
5. Use a different fragment shader that outputs a single (border) color
6. Draw the objects again, but only if their fragments' stencil values are not equal to 1
7. Enable stencil writing and depth testing again

Object Outlining

- This sets the content of the stencil buffer to 1s for each of the object's fragments
- The borders, are basically scaled-up versions of the objects
- Wherever the stencil test passes, the scaled-up version is drawn which is around the borders of the object
- Discarding all the fragments of the scaled-up versions that are part of the original objects' fragments using the stencil buffer

Object Outlining

- First, create a very basic fragment shader that outputs a border color
- Simply set a color value and call the shader shaderSingleColor:

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(0.04, 0.28, 0.26, 1.0);
}
```

Object Outlining

- Add object outlining to the two boxes only (leave the floor out)
- Draw the floor first, then the two boxes (while writing to the stencil buffer), then draw the scaled-up boxes(while discarding the fragments that write over the previously drawn container fragments)

Object Outlining

- First, want to enable stencil testing and set the actions to take whenever any of the tests succeed or fail:

```
glEnable(GL_STENCIL_TEST);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
```

- If any of the tests fail do nothing, we simply keep the currently stored value that is in the stencil buffer
- If both the stencil test and the depth test succeed however, we want to replace the stored stencil value with the reference value set via glStencilFunc which we later set to 1

Object Outlining

- Clear the stencil buffer to 0s and for the boxes update the stencil buffer to 1 for each fragment drawn:

```
glStencilFunc(GL_ALWAYS, 1, 0xFF);
glStencilMask(0xFF);
shader.use();
DrawBoxes();
```

- With GL_ALWAYS, we make sure that each of the boxes' fragments update the stencil buffer with a stencil value of 1
- Because the fragments always pass the stencil test, the stencil buffer is updated with the reference value wherever we've drawn them

Object Outlining

- Stencil buffer is updated with 1s where the boxes were drawn
- Now draw the upscaled boxes, but disabling writing to the stencil buffer:

```
glStencilFunc(GL_NOTEQUAL, 1, 0xFF);
glStencilMask(0x00);
glDisable(GL_DEPTH_TEST);
shaderSingleColor.use();
DrawScaledBoxes();
```

- Set GL_NOTEQUAL to ensure that only drawing parts of the boxes not equal 1 are drawn (parts outside the previously drawn boxes)
- Disable depth testing → scaled up containers e.g. the borders do not get overwritten by the floor (enable afterwards)

Object Outlining

- Object outlining routine:

```
float currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;

processInput(window);

glClearColor(1.f, 1.f, 1.f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

glm::mat4 model = glm::mat4(1.0f);
glm::mat4 view = camera.GetViewMatrix();
glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)SCR_WIDTH /
(float)SCR_HEIGHT, 0.1f, 100.0f);
```

Object Outlining

- Object outlining routine:

```
// draw floor as normal, but don't write the floor to the stencil buffer
glStencilMask(0x00);
shader.use();
shader.setMat4("view", view);
shader.setMat4("projection", projection);
shader.setMat4("model", glm::mat4(1.0f));
glBindVertexArray(planeVAO);
glBindTexture(GL_TEXTURE_2D, floorTexture);
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindVertexArray(0);
```

Object Outlining

- Object outlining routine:

```
// 1st. render pass, draw objects as normal, writing to the stencil buffer
glStencilFunc(GL_ALWAYS, 1, 0xFF);
glStencilMask(0xFF);
// cubes
glBindVertexArray(cubeVAO);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, cubeTexture);
model = glm::translate(model, glm::vec3(-1.0f, 0.0f, -1.0f));
shader.setMat4("model", model);
glDrawArrays(GL_TRIANGLES, 0, 36);
model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(2.0f, 0.0f, 0.0f));
shader.setMat4("model", model);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Object Outlining

- Object outlining routine:

```
// 2nd.: draw scaled versions of the boxes, disabling stencil writing.
glStencilFunc(GL_NOTEQUAL, 1, 0xFF);
glStencilMask(0x00);
glDisable(GL_DEPTH_TEST);
shaderSingleColor.use();
shaderSingleColor.setMat4("view", view);
shaderSingleColor.setMat4("projection", projection);
glBindVertexArray(cubeVAO);
```

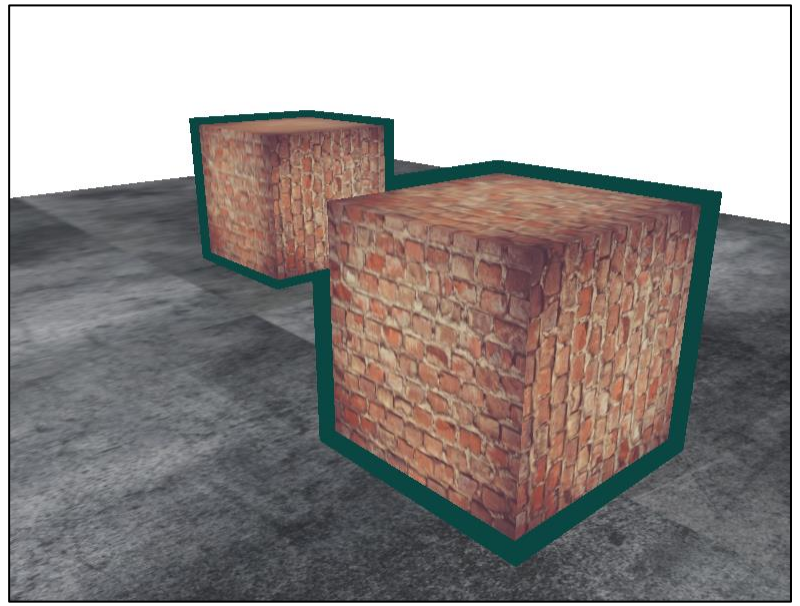
Object Outlining

- Object outlining routine:

```
glBindTexture(GL_TEXTURE_2D, cubeTexture);
model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(-1.0f, 0.0f, -1.0f));
model = glm::scale(model, glm::vec3(1.1, 1.1, 1.1));
shaderSingleColor.setMat4("model", model);
glDrawArrays(GL_TRIANGLES, 0, 36);
model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(2.0f, 0.0f, 0.0f));
model = glm::scale(model, glm::vec3(1.1, 1.1, 1.1));
shaderSingleColor.setMat4("model", model);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
glStencilMask(0xFF);
glStencilFunc(GL_ALWAYS, 0, 0xFF);
glEnable(GL_DEPTH_TEST);
```

F5…

… nice outlines!



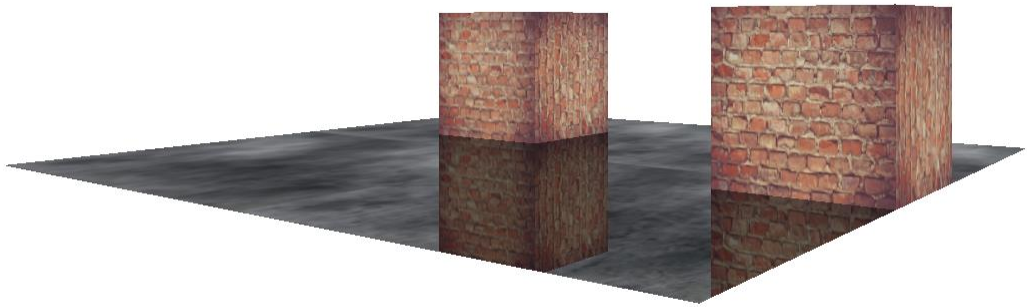
Borders overlap between both containers which is usually the desired effect that we want

If you want a complete border per object you'd have to clear the stencil buffer per object and get a little creative with the depth buffer

Reflection*

Reflection

- Now, we want to experiment with stencil buffers and create a very simple reflection:



Reflection

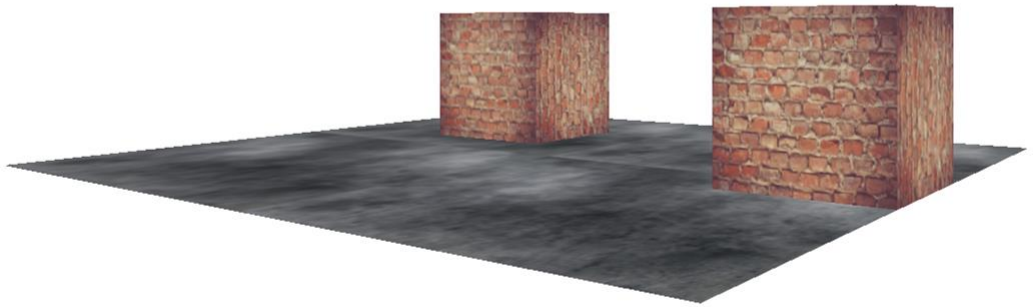
- First, we create two cubes:

```
glEnable(GL_DEPTH_TEST);
glm::mat4 model, view, projection = ...
shader.use();
glBindVertexArray(cubeVAO);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, cubeTexture);
model = …;
shader.setMat4(…);
glDrawArrays(GL_TRIANGLES, 0, 36);
model = …;
shader.setMat4(…);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Reflection

- Then, create the floor:

```
shader.use();
model = …;
shader.setMat4(…);

glBindVertexArray(planeVAO);
glBindTexture(GL_TEXTURE_2D, floorTexture);
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindVertexArray(0);
```



Reflection

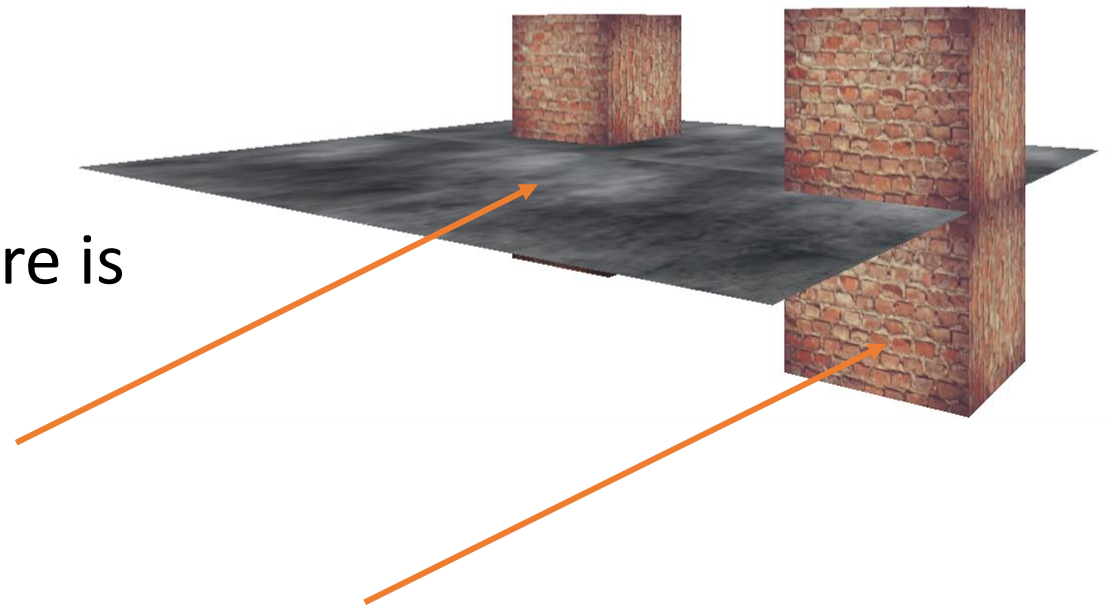
- Then, create translated boxes:

```
shader.use();
glBindVertexArray(cubeVAO);
glBindTexture(GL_TEXTURE_2D, cubeTexture);
model = …;
shader.setMat4(…);
glDrawArrays(GL_TRIANGLES, 0, 36);
model = …;
shader.setMat4(…);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
```

Reflection

- What is wrong?
- Missing reflection
- Visible box

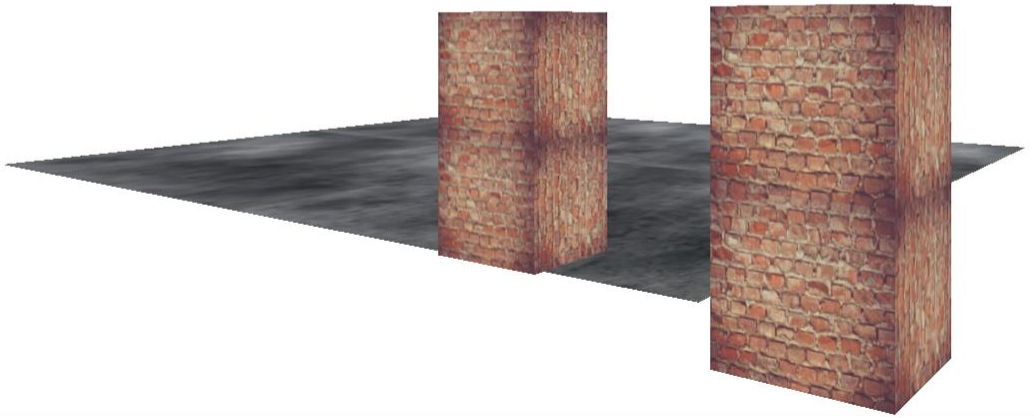
→ Container that is underneath the floor should only be visible where there is also a floor



Reflection

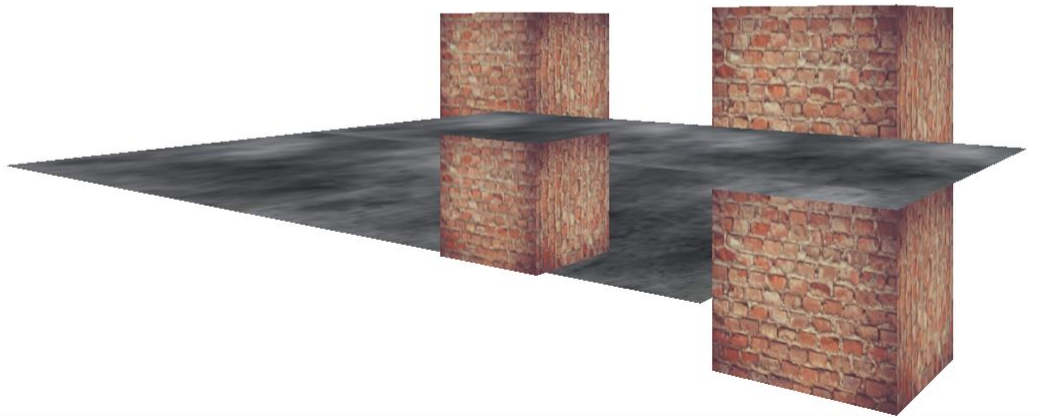
- Missing reflection is easy:

```
// Draw floor
glDepthMask(GL_FALSE);
shader.use();
…
glDrawArrays(GL_TRIANGLES, 0, 6);
glDepthMask(GL_TRUE);
```



Reflection

- Deactivating the depth test gives us unwanted results → stick with `glDepthMask(GL_FALSE);`



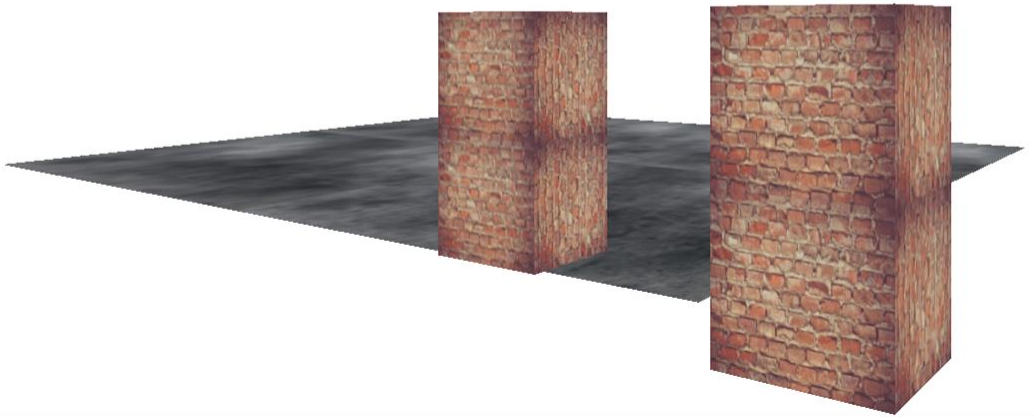
Procedure

- Floor:
  - Activate stencil test
  - Write 1s to all stencils of the floor
  - Draw floor
- Reflected boxes:
  - Stencil should only pass of the stencil values are 1 (then the floor was drawn)
  - Draw the reflected boxes only at the abovementioned fragments

Reflection

- For the floor:

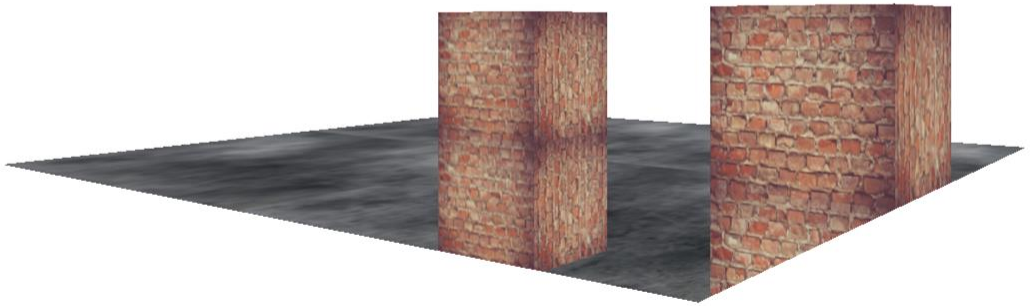
```
// Draw floor
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS, 1, 0xFF);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glStencilMask(0xFF);
shader.use();
…
glDrawArrays(GL_TRIANGLES, 0, 6);
```



Reflection

- Reflected boxes:

```
glStencilFunc(GL_EQUAL, 1, 0xFF);
glDepthMask(GL_TRUE);

shader.use();
glBindVertexArray(cubeVAO);
glBindTexture(GL_TEXTURE_2D, cubeTexture);
model = …;
shader.setMat4(…);
glDrawArrays(GL_TRIANGLES, 0, 36);
model = …;
shader.setMat4(…);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
```

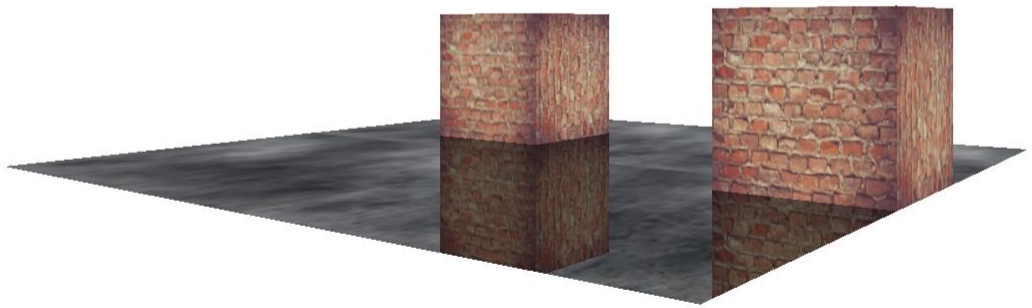


Reflection

- Add attenuation

```
…
shader.setFloat("attenuation", 1.);
…
shader.setFloat("attenuation", .5);
…
```

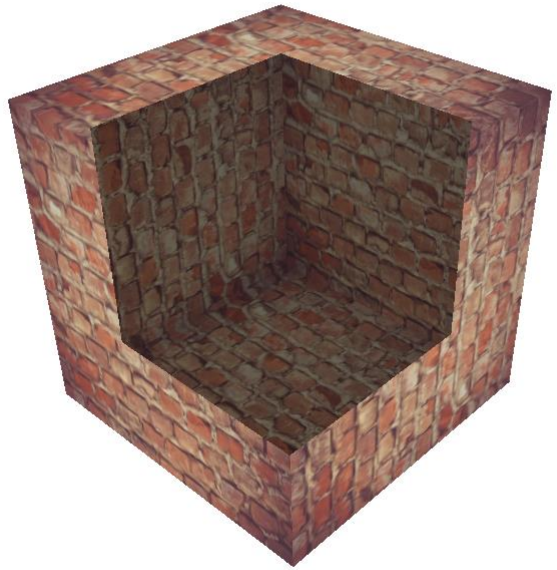
```
…
uniform float attenuation;

void main()
{
    FragColor = texture(texture1,TexCoords);
    FragColor.rgb*=attenuation;
}
```



Cut Out*

Cut Out

- Now, we want to cut out a piece of the box to reveal the inside



Cut Out

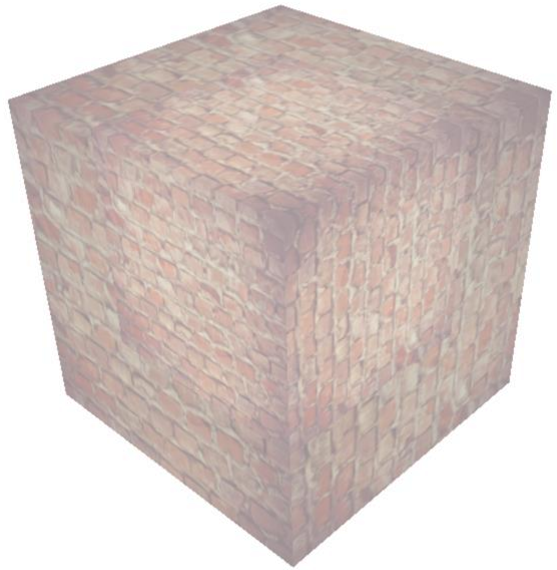
- First, we create a box:

```
glEnable(GL_DEPTH_TEST);
glEnable(GL_STENCIL_TEST);
glm::mat4 model, view, projection = ... shader.use();
glBindVertexArray(cubeVAO);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, cubeTexture);
model = …;
shader.setMat4(…);
shader.setFloat("attenuation", 1.0);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Cut Out

- Then, create the cut-out box:

```
// generate second smaller cut-out box
glStencilFunc(GL_ALWAYS, 1, 0xFF);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glStencilMask(0xFF);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE,
GL_FALSE);
glDepthMask(GL_FALSE);
model = glm::mat4(1.0f);
model = glm::scale(model, glm::vec3(0.5, 0.5, 0.5));
model = glm::translate(model, glm::vec3(1.0f, 1.0f, 1.0f));
shader.setMat4("model", model);
shader.setFloat("attenuation", 1.0);
glDrawArrays(GL_TRIANGLES, 0, 36);
```



Cut Out

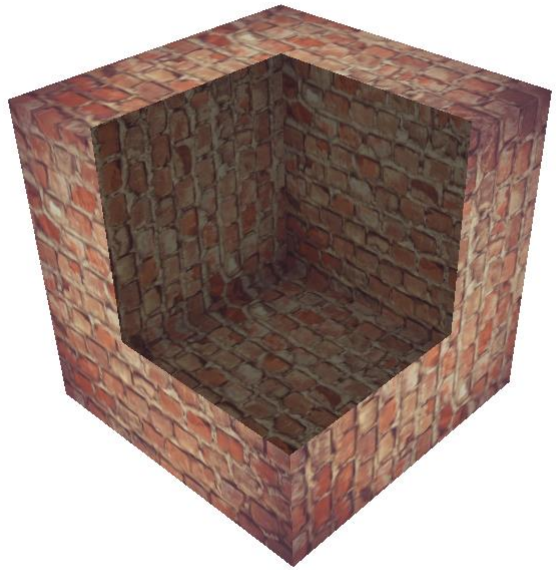
- Draw the inside of the box:

```
// draw the inside of the box
glStencilFunc(GL_EQUAL, 1, 0xFF);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glStencilMask(0xFF);
glClearDepth(0);
glClear(GL_DEPTH_BUFFER_BIT);
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE,
GL_TRUE);
glDepthMask(GL_TRUE);
glDepthFunc(GL_GREATER);
model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f));
shader.setMat4("model", model);
shader.setFloat("attenuation", 0.5);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Cut Out

- Draw the outside of the box:

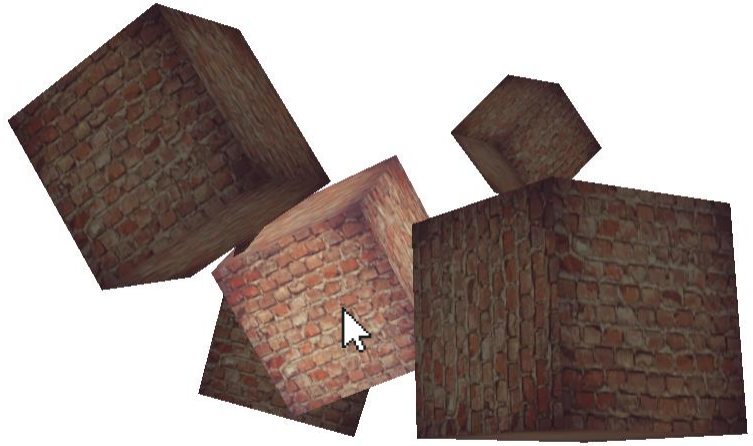
```
// draw the outside of the box
glStencilFunc(GL_EQUAL, 0, 0xFF);
glStencilMask(0xFF);
glClearDepth(1);
glClear(GL_DEPTH_BUFFER_BIT);
glDepthFunc(GL_LESS);
model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f));
shader.setMat4("model", model);
shader.setFloat("attenuation", 1.0);
glDrawArrays(GL_TRIANGLES, 0, 36);
```



Picking*

Picking

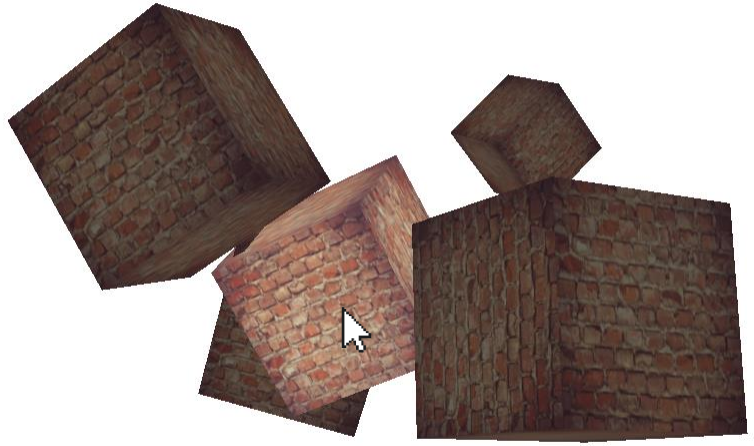
- Last but not least, we want pick the boxes (again)



Picking

- This is straightforward:

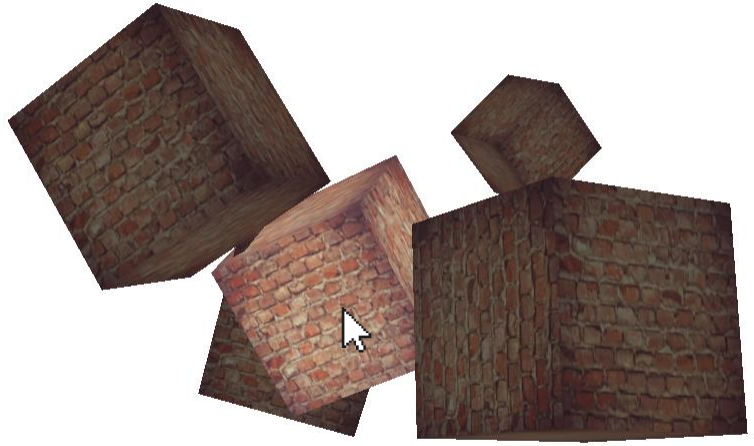
```
glfwSetInputMode(window, GLFW_CURSOR,
GLFW_CURSOR_NORMAL);
glEnable(GL_DEPTH_TEST);
glEnable(GL_STENCIL_TEST);
int StencilVal = -1;
while (!glfwWindowShouldClose(window)){
…
// set uniforms
glm::mat4 model, view, projection ...
shader.use();
shader.setMat4("view", view);
shader.setMat4("projection", projection);
glBindVertexArray(cubeVAO);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, cubeTexture);
```



Picking

- This is straightforward:

```
glReadPixels(lastX, SCR_HEIGHT-lastY,1,1,
GL_STENCIL_INDEX, GL_UNSIGNED_INT,
&StencilVal);

for (unsigned int i = 0; i < 10; i++){
glStencilFunc(GL_ALWAYS, i+1, 0xFF);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glStencilMask(0xFF);
model = glm::mat4(1.0f);
model = glm::translate(model, cubePositions[i]);
float angle = 20.0f * i;
model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f,
0.3f, 0.5f));
shader.setMat4("model", model);
shader.setFloat("attenuation",
float(i == StencilVal-1)/2.+0.5);
glDrawArrays(GL_TRIANGLES, 0, 36);}
```



Questions???