# Computer Graphics II – Blending

Kai Lawonn

# Introduction

- Blending in OpenGL is also commonly known as the technique to implement transparency within objects
- Transparency: objects not having a solid color, but a combination of colors from the object itself and any other object behind it with varying intensity
- 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
- Blending: blend several colors (of different objects) to a single color (Transparency allows to see through objects)

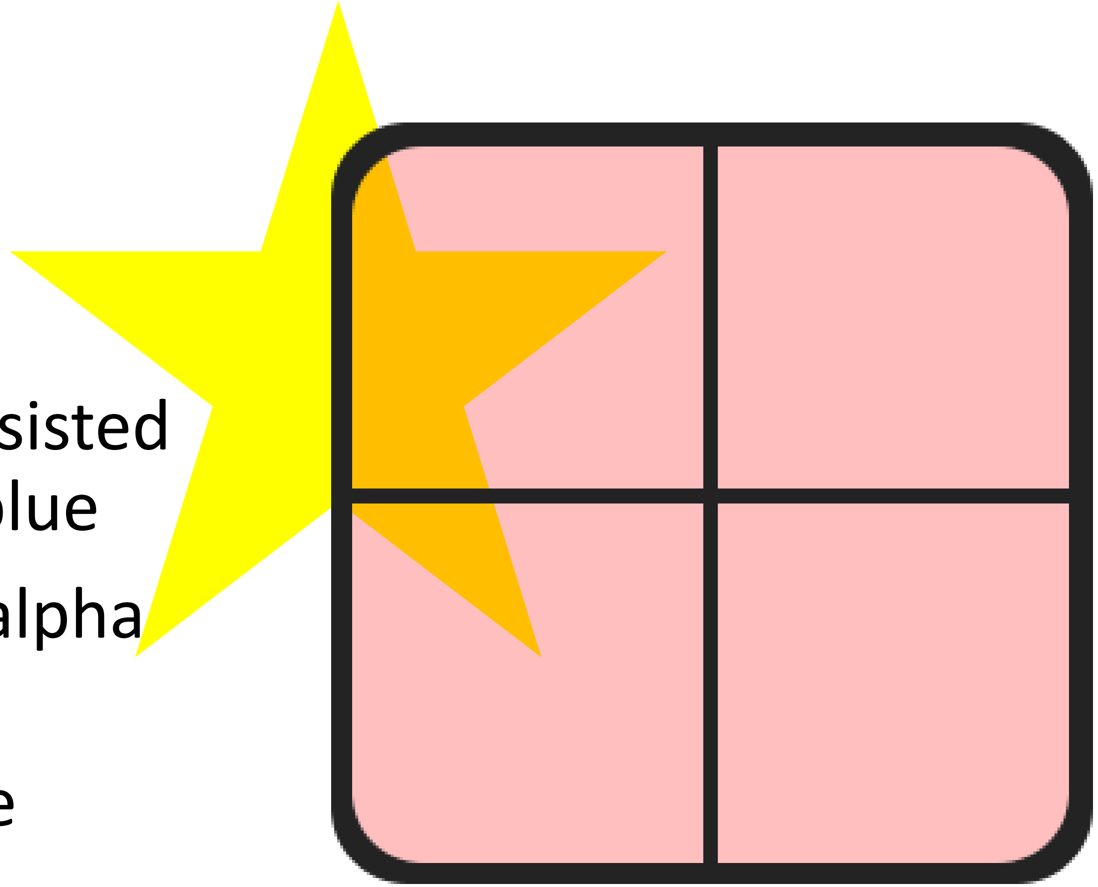
# Introduction

Full transparent window

Partially transparent window

- Transparent objects can be completely transparent (l.) or partially transparent (r.)
- Transparency of an object is defined by its color’s alpha value (4th component of a color vector)
- Kept the 4th component at a value of 1.0 giving the object 0.0 transparency, while an alpha value of 0.0 would result in the object having complete transparency
- An alpha value of 0.5 tells the object’s color consist of 50% of its own color and 50% of the colors behind the object

# Introduction



- The textures we have used so far all consisted of 3 color components: red, green and blue
- Some textures also have an embedded alpha channel
- This tells which parts of the texture have transparency and by how much
- For example, the following window texture has an alpha value of 0.25 at its glass part (it would normally be completely red, but since it has 75% transparency it largely shows the star in an orange color) and an alpha value of 0.0 at its corners

# Discard (again)

# Introduction

- Some images have full transparent parts, e.g., a grass texture
- Generally, paste a grass texture onto a 2D quad and place that quad into the scene
- However, grass is not exactly shaped like a 2D square so you only want to display some parts of the grass texture and ignore the others

# Introduction

- Example: it is either is full opaque (alpha = 1.0) or it is fully transparent (alpha = 0.0)
- You can see that wherever there is no grass

# Introduction

- Adding grass to the scene, we want to see the grass only → discard fragments showing the transparent parts of the texture

# Load Texture

- stb_image automatically loads an image's alpha channel if it's available
- Need to tell OpenGL that the texture uses an alpha channel:

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

# Shader

- Also make sure that you retrieve all 4 color components of the texture in the fragment shader, not just the RGB components:

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;

void main()
{
    // FragColor = vec4(vec3(texture(texture1, TexCoords)), 1.0);
    FragColor  = texture(texture1, TexCoords);
}
```

# Grass Leaves

- Add several of these leaves of grass throughout the basic scene (depth testing lecture)
- Create a small vector and add several glm::vec3 variables to represent the location of the grass leaves:

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

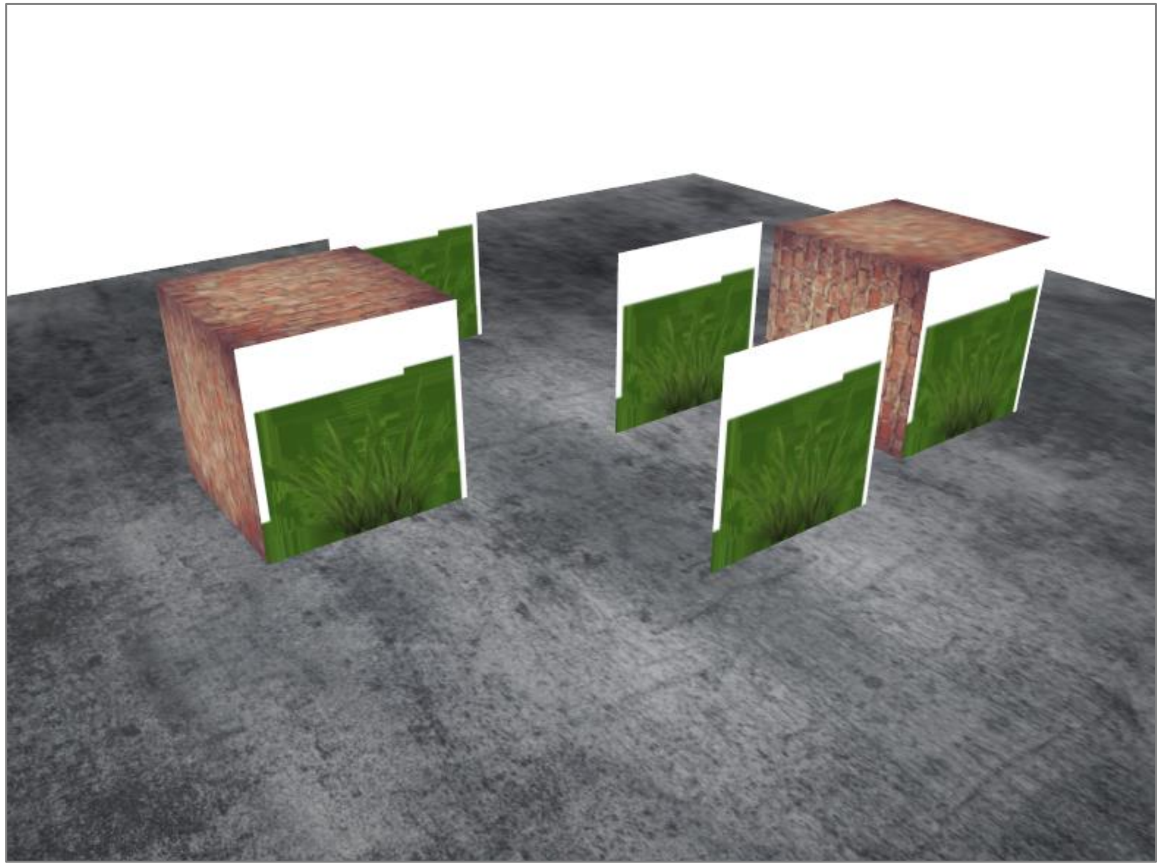
# Grass Leaves

- Each grass object is rendered as a single quad with the grass texture
- Not a perfect 3D representation of grass, but it’s efficient than actually loading complex models
- Trick: add several more rotated grass quads to get a better result
- Create another VAO, fill the VBO and add the grass leaves:

```
glBindVertexArray(transparentVAO);
glBindTexture(GL_TEXTURE_2D, transparentTexture);
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);
}
```

# F5…

- … we see the background



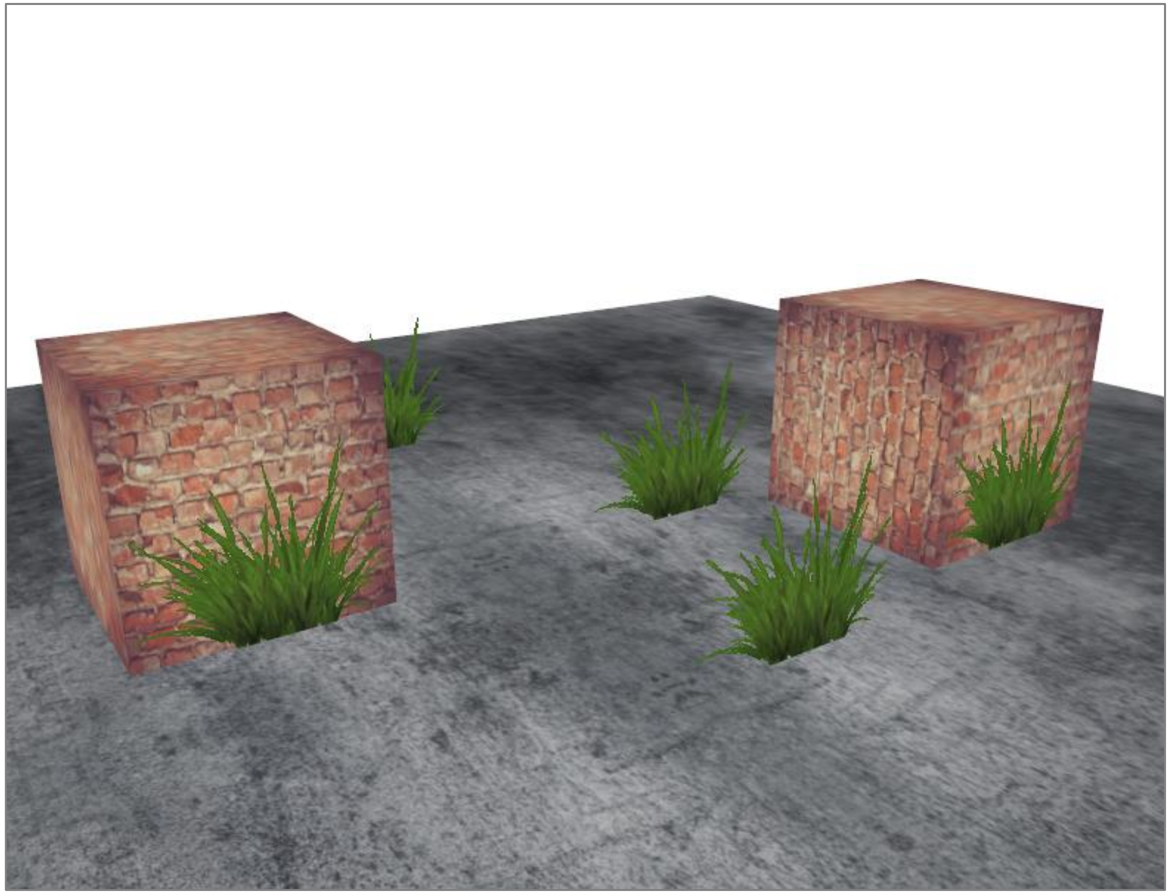
# Grass Leaves

- OpenGL by default does not know what to do with alpha values
- Check in the fragment shader the alpha value, if it is below a certain threshold, discard the fragment:

```
void main()
{
    vec4 texColor = texture(texture1, TexCoords);
    if(texColor.a < 0.1)
        discard;
    FragColor = texColor;
}
```

# F5…

- … looks good



# Texture

- OpenGL interpolates the border values of the texture with the next repeated value of the texture (wrapping parameters: GL_REPEAT)
- With transparent values, the top of the texture image gets its transparent value interpolated with the bottom border's solid color
- Result is a slightly semi-transparent colored border around the textured quad
- To prevent this, set the texture wrapping method to GL_CLAMP_TO_EDGE whenever you use alpha textures

# Rotation

- Change the coordinates of the quad:

```
 float transparentVertices[] = {
        -1.0f,  1.f,  0.0f,  0.0f,  0.0f,
        -1.0f, -1.f,  0.0f,  0.0f,  1.0f,
        1.0f, -1.f,  0.0f,  1.0f,  1.0f,

        -1.0f,  1.f,  0.0f,  0.0f,  0.0f,
        1.0f, -1.f,  0.0f,  1.0f,  1.0f,
        1.0f,  1.f,  0.0f,  1.0f,  0.0f
};
```

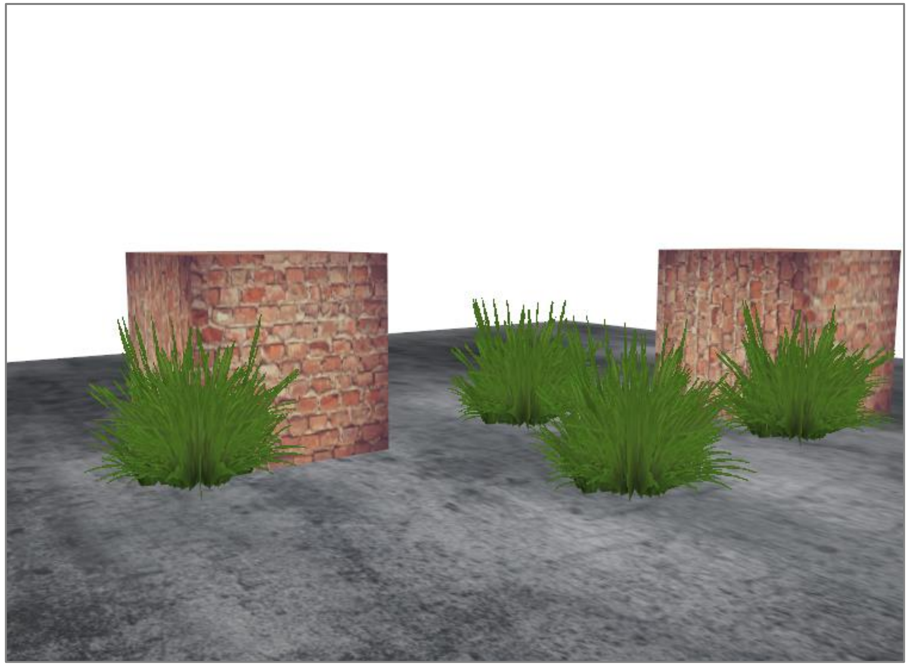
# Rotation

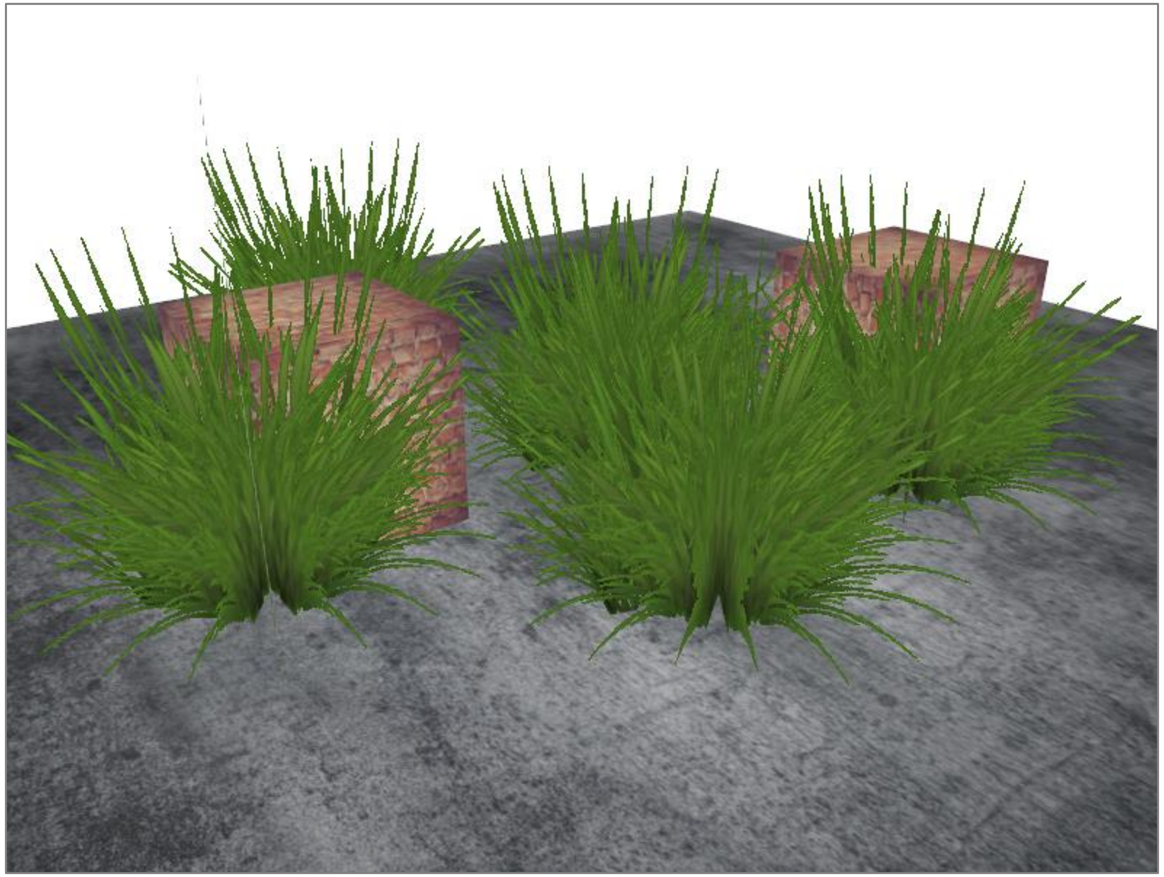
- Rotate the quad:

```
const int numQuads = 10;
const float pi_approx = 3.14159;
for (unsigned int i = 0; i < vegetation.size(); i++)
{
   for (unsigned int j = 0; j < numQuads; j++)
   {
        model = glm::mat4(1.0f);
        model = glm::translate(model, glm::vec3(0, 0.5, 0));
        model = glm::translate(model, vegetation[i]);
        model = glm::rotate(model, float(j) / numQuads * pi_approx,
                glm::vec3(0, 1, 0));
        shader.setMat4("model", model);
        glDrawArrays(GL_TRIANGLES, 0, 6);
    }
}
```

# F5…

- … and we get a better result





# Blending

# Introduction

- Discarding does not give us the possibility to render semi-transparent images
- To render images with different levels of transparency we have to enable blending:

```
glEnable(GL_BLEND);
```

# Introduction

- Need to tell OpenGL how it should actually blend
- Blending in OpenGL is done with the following equation:

$$\bar{C}_{result} = \bar{C}_{source} \cdot F_{source} + \bar{C}_{destination} \cdot F_{destination}$$

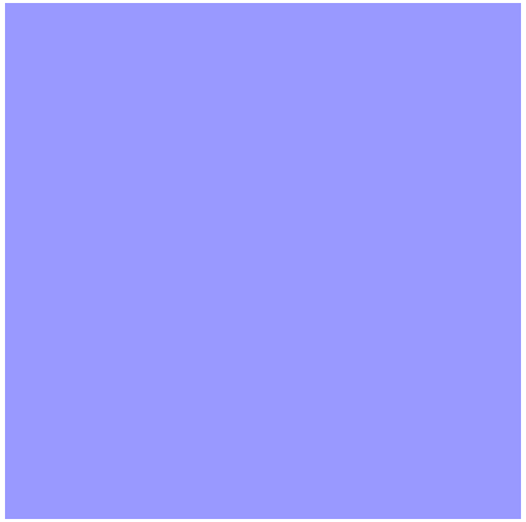
- $\bar{C}_{source}$: source color vector (originates from the texture)
- $\bar{C}_{destination}$: destination color vector (currently stored in the color buffer)
- $\bar{F}_{source}$: source factor value (impact of the alpha value on the source color)
- $\bar{F}_{destination}$: destination factor value (impact of the alpha value on the destination color)

# Introduction

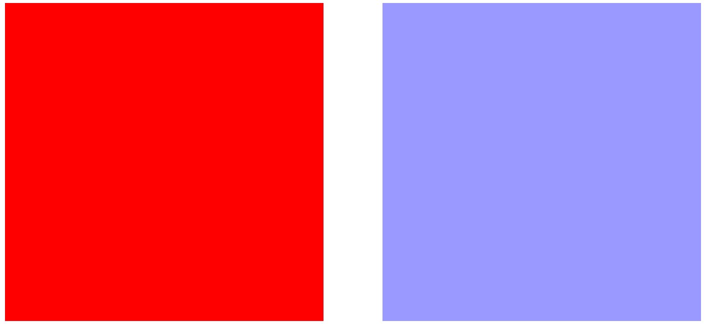
- After the fragment shader (and all tests have passed), this blend equation is applied on the fragment's color output with the currently stored color in the color buffer
- Source and destination colors will automatically be set by OpenGL, but the source and destination factor can be set to a value of our choosing

# Example


(1,0,0,1)


(0,0,1,0.6)

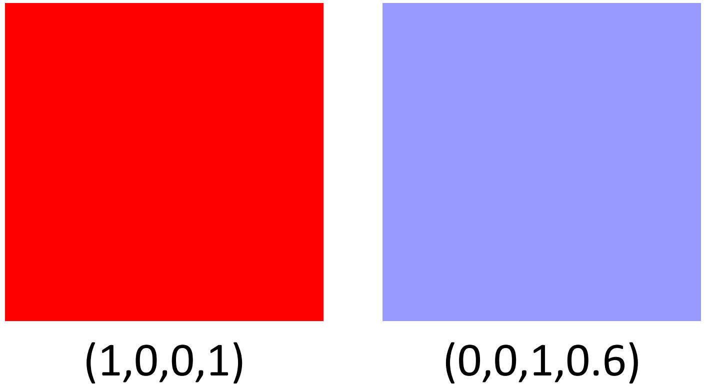
# Example



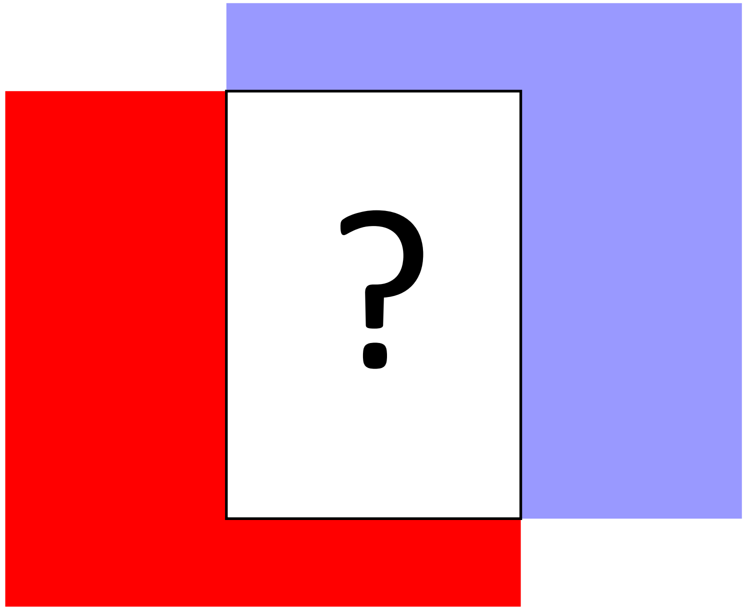
(1,0,0,1) (0,0,1,0.6)

- Want to draw the semi-transparent blue square on top of the red square
- Red square = destination color (→ should be first in the color buffer)
- Now draw the blue square over the red square

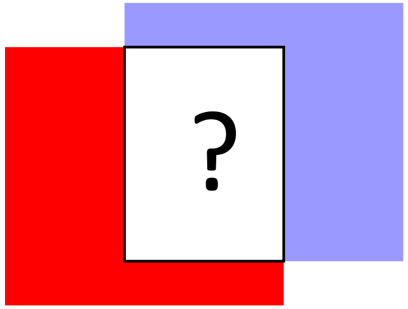
# Example




- The question then arises: what do we set the factor values to and what is the final color?



# Example




- Want to multiply the blue square with its alpha value:

$$\bar{F}_{source} = 0.6$$

- Destination square have a contribution equal to the remainder of the alpha value:

$$\bar{F}_{destination} = 1 - 0.6 = 0.4$$

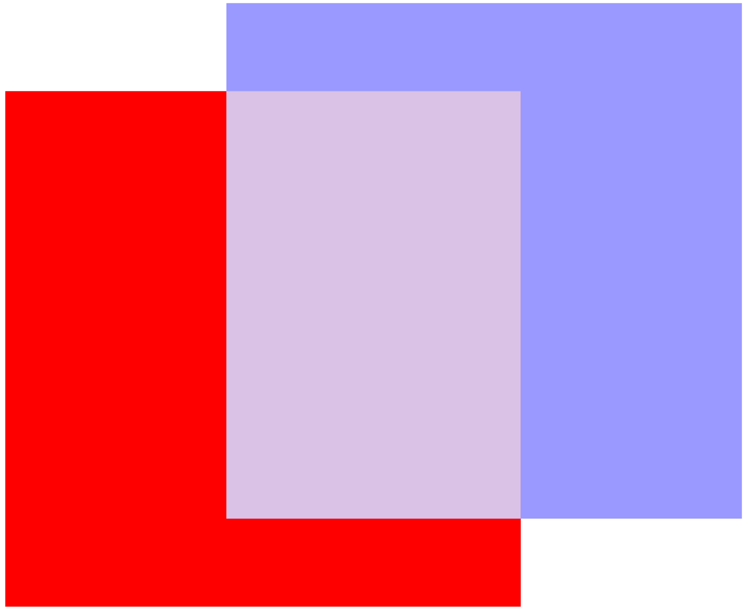
- The equation thus becomes:

$$\bar{C}_{result} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0.6 \end{pmatrix} \cdot 0.6 + \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \cdot (1 - 0.6) = \begin{pmatrix} 0.4 \\ 0 \\ 0.6 \\ 0.76 \end{pmatrix}$$

# Example

$$\bar{C}_{result} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0.6 \end{pmatrix} \cdot 0.6 + \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \cdot (1 - 0.6) = \begin{pmatrix} 0.4 \\ 0 \\ 0.6 \\ 0.76 \end{pmatrix}$$

- Final color:



# glBlendFunc

- How do we tell OpenGL to use factors like these?
- There is a function for this called: glBlendFunc
- glBlendFunc(GLenum sfactor, GLenum dfactor):
  - expects two parameters that set the option for the source and destination factor
- OpenGL defined quite a few options
- It is also possible to set a constant color $\bar{C}_{const}$ with

```
glBlendColor(GLfloat red,GLfloat green,GLfloat blue,GLfloat alpha);
```

# glBlendFunc

Previous:

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

| 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 $\bar{C}_{const}$. |
| GL_ONE_MINUS_CONSTANT_COLOR | Factor is equal to 1 - the constant color vector $\bar{C}_{const}$. |
| GL_CONSTANT_ALPHA | Factor is equal to the alpha component of the constant color vector $\bar{C}_{const}$. |
| GL_ONE_MINUS_CONSTANT_ALPHA | Factor is equal to 1 −alpha of the constant color vector $\bar{C}_{const}$. |

# glBlendFuncSeparate

- It is also possible to set different options for the RGB and alpha channel individually using glBlendFuncSeparate:

```
glBlendFuncSeparate(GLenum srcRGB, GLenum dstRGB, GLenum srcAlpha, GLenum dstAlpha);
```

- srcRGB: Specifies how the red, green, and blue blending factors are computed (initially GL_ONE)
- dstRGB: Specifies how the red, green, and blue destination blending factors are computed (initially GL_ZERO)
- srcAlpha: Specified how the alpha source blending factor is computed (initially GL_ONE)
- dstAlpha: Specified how the alpha destination blending factor is computed (initially GL_ZERO)



https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBlendFuncSeparate.xhtml

# glBlendFuncSeparate

- The calculations are:

```
glBlendFuncSeparate(GLenum srcRGB, GLenum dstRGB, GLenum srcAlpha, GLenum dstAlpha);
```

$$\bar{C}_{result} = \bar{C}_{source} \cdot srcRGB + \bar{C}_{destination} \cdot dstRGB$$
$$\bar{A}_{result} = \bar{A}_{source} \cdot srcAlpha + \bar{A}_{destination} \cdot dstAlpha$$

# glBlendFuncSeparate

| Parameter | RGB Factor | Alpha Factor |
|---|---|---|
| GL_ZERO | $(0,0,0)$ | $0$ |
| GL_ONE | $(1,1,1)$ | $1$ |
| GL_SRC_COLOR | $\left(\frac{R_{s0}}{k_R},\frac{G_{s0}}{k_G},\frac{B_{s0}}{k_B}\right)$ | $\frac{A_{s0}}{k_A}$ |
| GL_ONE_MINUS_SRC_COLOR | $(1,1,1)-\left(\frac{R_{s0}}{k_R},\frac{G_{s0}}{k_G},\frac{B_{s0}}{k_B}\right)$ | $1-\frac{A_{s0}}{k_A}$ |
| GL_DST_COLOR | $\left(\frac{R_d}{k_R},\frac{G_d}{k_G},\frac{B_d}{k_B}\right)$ | $\frac{A_d}{k_A}$ |
| GL_ONE_MINUS_DST_COLOR | $(1,1,1)-\left(\frac{R_d}{k_R},\frac{G_d}{k_G},\frac{B_d}{k_B}\right)$ | $1-\frac{A_d}{k_A}$ |
| GL_SRC_ALPHA | $\left(\frac{A_{s0}}{k_A},\frac{A_{s0}}{k_A},\frac{A_{s0}}{k_A}\right)$ | $\frac{A_{s0}}{k_A}$ |
| GL_ONE_MINUS_SRC_ALPHA | $(1,1,1)-\left(\frac{A_{s0}}{k_A},\frac{A_{s0}}{k_A},\frac{A_{s0}}{k_A}\right)$ | $1-\frac{A_{s0}}{k_A}$ |
| GL_DST_ALPHA | $\left(\frac{A_d}{k_A},\frac{A_d}{k_A},\frac{A_d}{k_A}\right)$ | $\frac{A_d}{k_A}$ |
| GL_ONE_MINUS_DST_ALPHA | $(1,1,1)-\left(\frac{A_d}{k_A},\frac{A_d}{k_A},\frac{A_d}{k_A}\right)$ | $1-\frac{A_d}{k_A}$ |
| GL_CONSTANT_COLOR | $(R_c,G_c,B_c)$ | $A_c$ |
| GL_ONE_MINUS_CONSTANT_COLOR | $(1,1,1)-(R_c,G_c,B_c)$ | $1-A_c$ |
| GL_CONSTANT_ALPHA | $(A_c,A_c,A_c)$ | $A_c$ |
| GL_ONE_MINUS_CONSTANT_ALPHA | $(1,1,1)-(A_c,A_c,A_c)$ | $1-A_c$ |
| GL_SRC_ALPHA_SATURATE | $(i,i,i)$ | $1$ |
| GL_SRC1_COLOR | $\left(\frac{R_{s1}}{k_R},\frac{G_{s1}}{k_G},\frac{B_{s1}}{k_B}\right)$ | $\frac{A_{s1}}{k_A}$ |
| GL_ONE_MINUS_SRC1_COLOR | $(1,1,1,1)-\left(\frac{R_{s1}}{k_R},\frac{G_{s1}}{k_G},\frac{B_{s1}}{k_B}\right)$ | $1-\frac{A_{s1}}{k_A}$ |
| GL_SRC1_ALPHA | $\left(\frac{A_{s1}}{k_A},\frac{A_{s1}}{k_A},\frac{A_{s1}}{k_A}\right)$ | $\frac{A_{s1}}{k_A}$ |
| GL_ONE_MINUS_SRC1_ALPHA | $(1,1,1)-\left(\frac{A_{s1}}{k_A},\frac{A_{s1}}{k_A},\frac{A_{s1}}{k_A}\right)$ | $1-\frac{A_{s1}}{k_A}$ |



https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBlendFuncSeparate.xhtml

# glBlendFuncSeparate

- Example: this sets the RGB components as previously, but only lets the resulting alpha component be influenced by the source's alpha value

```
glBlendFuncSeparate(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_ONE, GL_ZERO);
```

$$\bar{C}_{result} = \bar{C}_{source} \cdot \bar{A}_{source} + \bar{C}_{destination} \cdot (1 - \bar{A}_{source})$$
$$\bar{A}_{result} = \bar{A}_{source} \cdot 1 + \bar{A}_{destination} \cdot 0$$

# glBlendEquation

- More flexibility by changing the operator between the source and destination part of the equation
- Right now,  the source and destination components are added: more options

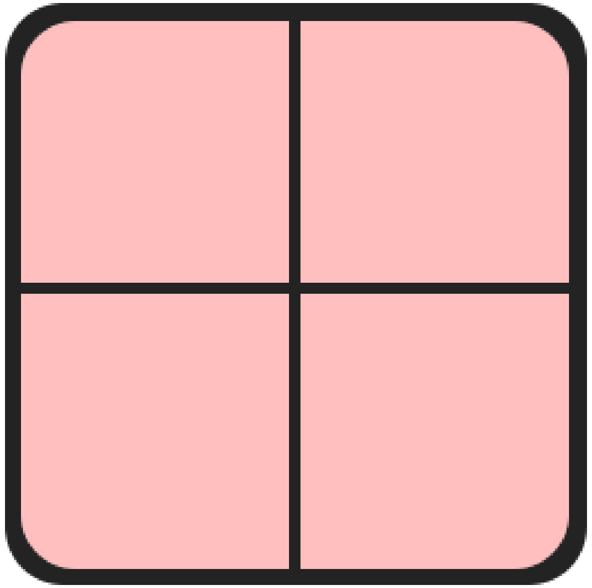
```
glBlendEquation(GLenum mode);
```

- GL_FUNC_ADD: the default: $\bar{C}_{result} = Src + Dst$
- GL_FUNC_SUBTRACT: $\bar{C}_{result} = Src - Dst$
- GL_FUNC_REVERSE_SUBTRACT: $\bar{C}_{result} = Dst - Src$
- GL_MIN: component-wise: $\bar{C}_{result} = \min(Src, Dst)$
- GL_MAX: $\bar{C}_{result} = \max(Src, Dst)$

# Semi-Transparent Textures

# Introduction

- Now that we know how OpenGL works, we are adding several semi-transparent windows
- Now, we are rendering the transparent window texture



# Introduction

- First, during initialization we enable blending and set the appropriate blending function:

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

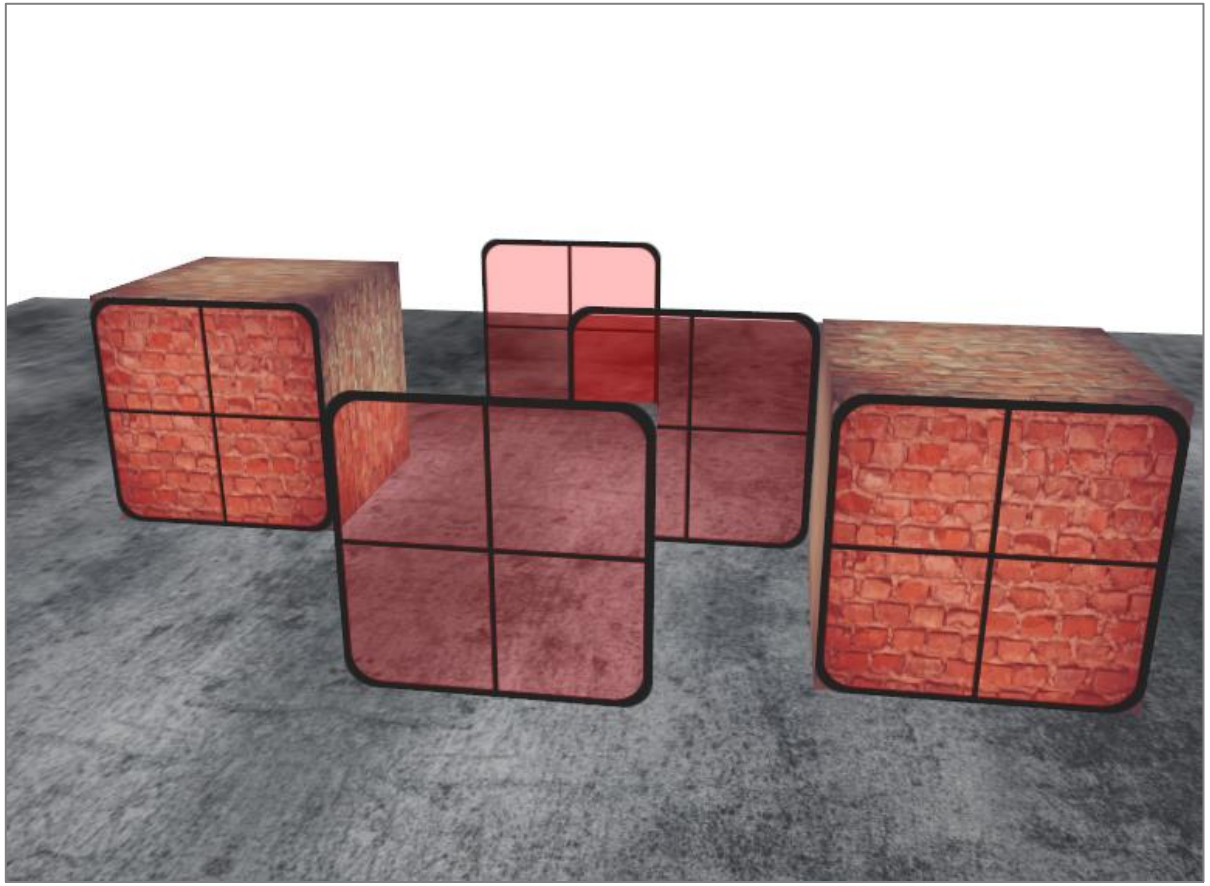
# Introduction

- Since we enabled blending there is no need to discard fragments so keep the original version:

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;

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

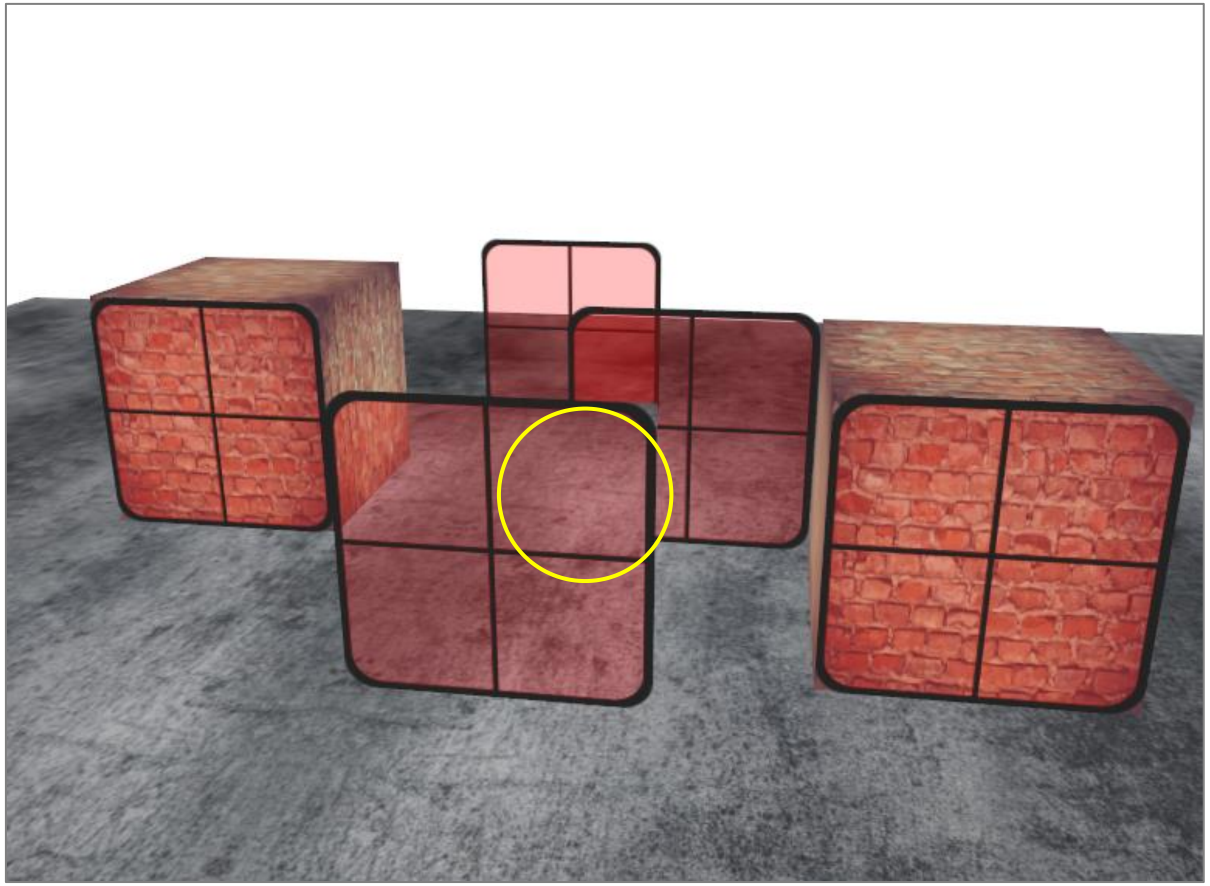
# F5…

- … the glass part of the window texture is semi-transparent we should be able to see the rest of the scene by looking through this window

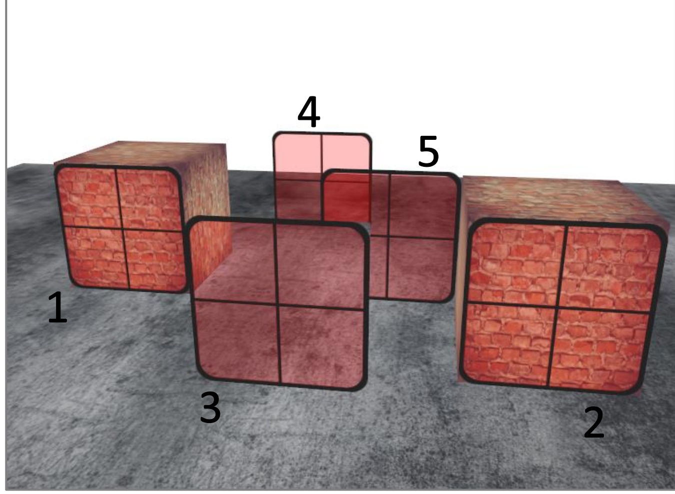


# F5…

- … transparent parts of the front window are occluding the windows in the background



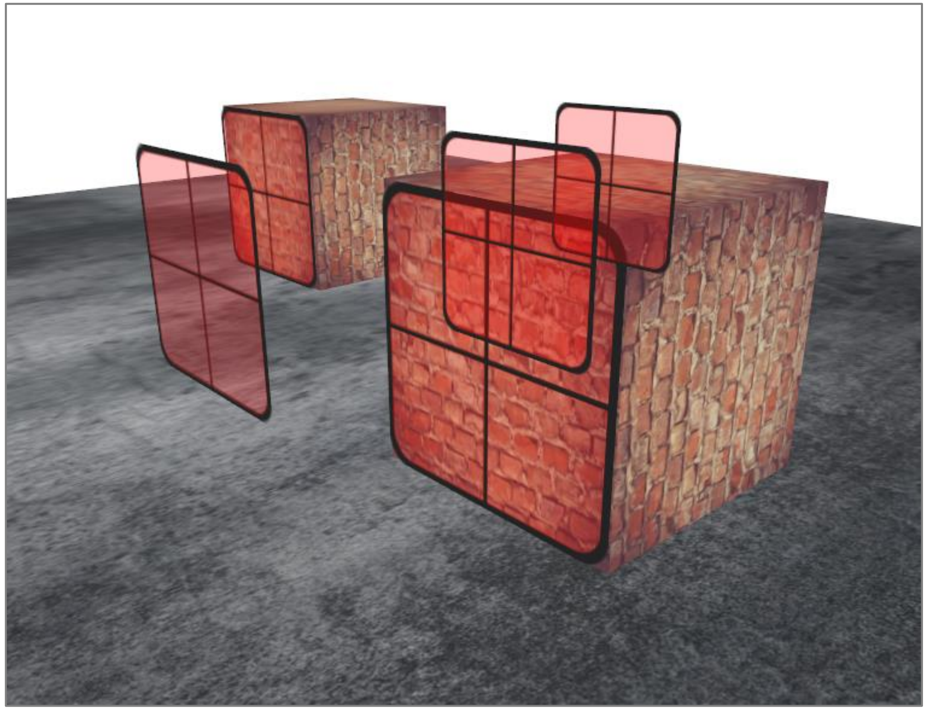
# Why?




- Depth testing tricky combined with blending
- When writing to the depth buffer, the depth test is independent of transparency
- Entire quad of the window is checked for depth testing regardless of transparency
- Even though the transparent part should show the windows behind it, the depth test discards them

# glDisable(GL_DEPTH_TEST)?

- This is also not a good idea…



# Why?

- Cannot render the windows however we want and expect the depth buffer to solve all our issues for us
- To make sure the windows show the windows behind them, we have to draw the windows in the background first
- This means we have to manually sort the windows from furthest to nearest and draw them accordingly ourselves

# Correct Order

- Have to draw the farthest object first and the closest object as last
- Non-blended objects can still be drawn as normal using the depth buffer (no need to sort), but need to be drawn first
- When drawing a scene with non-transparent and transparent objects the general outline 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.

# Sort

- Sorting: get distance of an object from the viewer’s perspective (distance between the camera’s position and the object’s position)
- Store the distance with the position vector in a map data structure (STL library)
- A map automatically sorts its values based on its keys

```
std::map<float, glm::vec3> sorted;
for (unsigned int i = 0; i < windows.size(); i++)
{
        float distance = glm::length(camera.Position - windows[i]);
        sorted[distance] = windows[i];
}
```

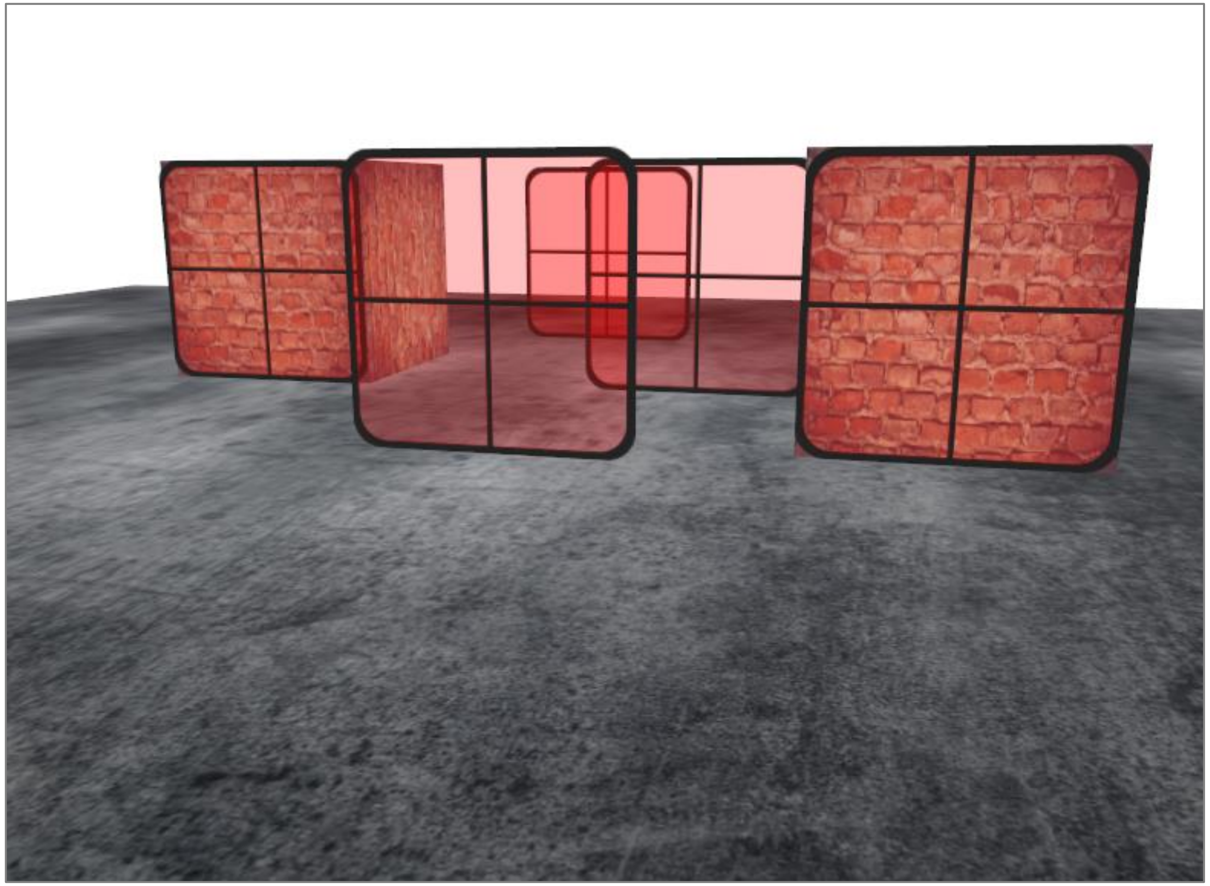
# Sort

- It results in a map that sorts each of the window positions based on their distance key value from lowest to highest distance
- For rendering, take the map’s values in reverse order (from farthest to nearest) and draw the corresponding windows in correct order:

```
for (std::map<float, glm::vec3>::reverse_iterator it = sorted.rbegin(); it !=
sorted.rend(); ++it)
{
        model = glm::mat4(1.0f);
        model = glm::translate(model, it->second);
        shader.setMat4("model", model);
        glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

# F5…

- … the glass part of the window texture is semi-transparent and correct now



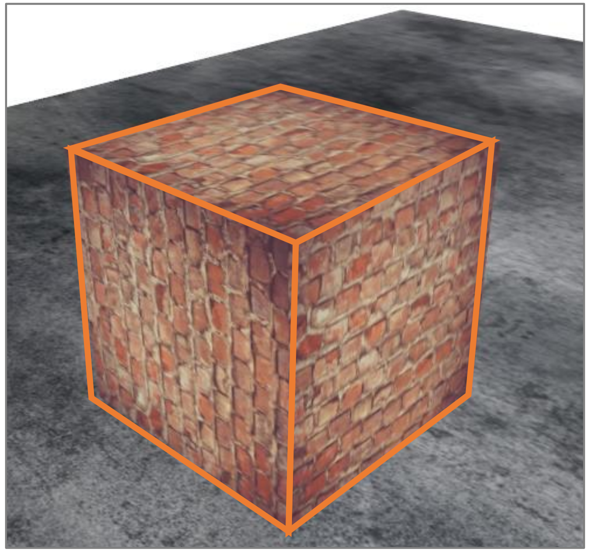
# Remarks

- This approach works well for this specific scenario, it doesn't take rotations, scaling or any other transformation into account and weirdly shaped objects need a different metric than simply a position vector
- Sorting objects in your scene is a difficult task
- More advanced techniques, e.g., order independent transparency
- For now it is ok, if we know the limitations, we can still get fairly decent blending implementations

# Face Culling

# Introduction

- If you look at this box and count the maximum number of faces you ended up with a maximum number of 3
- So why would we waste the effort of actually drawing those other 3 faces
- If we could discard those in some way we would save 50% of fragment shader runs



# Introduction

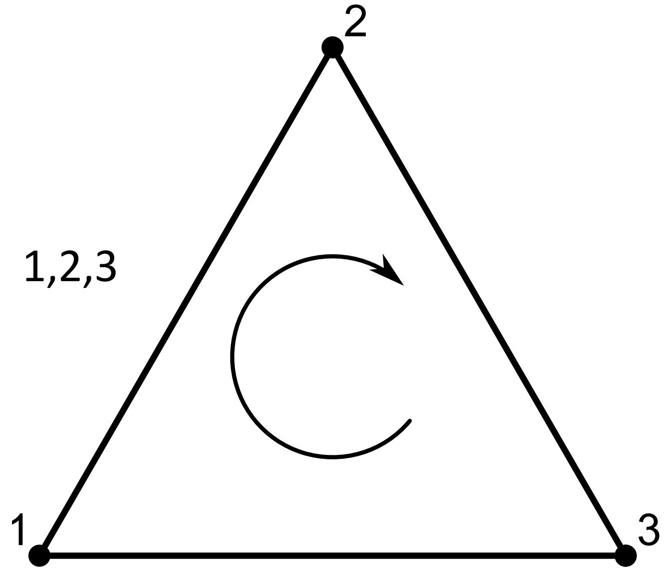
- Great idea, but how do we know if a face of an object is not visible from the viewer's point of view?
- If we imagine any closed convex shape, each of its faces has two sides
- Each side would either face the camera or show its back
- What if we could only render the faces that are facing the viewer?

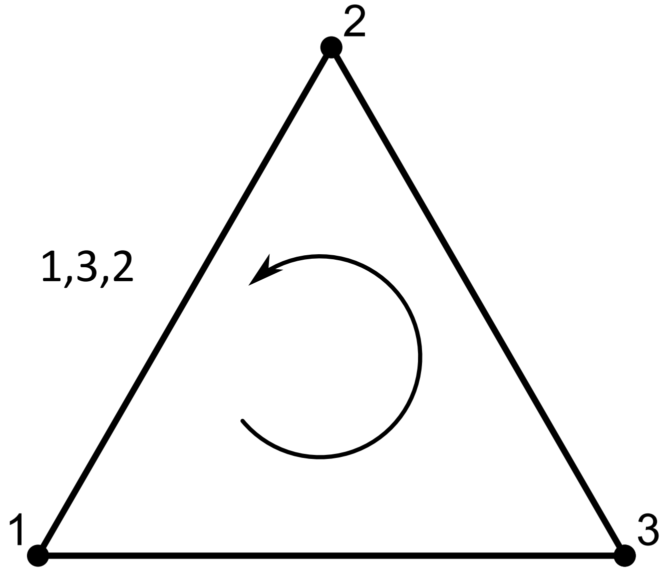
# Introduction

- This is exactly what face culling does
- OpenGL checks all the faces that are front facing towards the viewer and renders those while discarding all the faces that are back facing → saving us a lot of fragment shader calls (expensive!)
- We do need to tell OpenGL which of the faces we use are actually the front faces and which faces are the back faces
- OpenGL uses a clever trick for this by analyzing the winding order of the vertex data

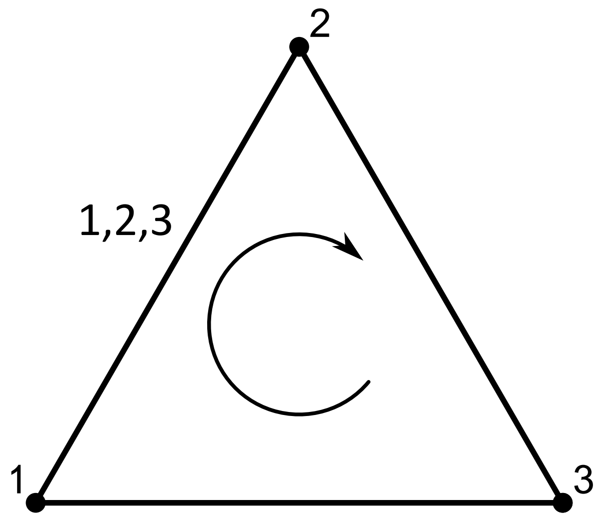
# Winding Order

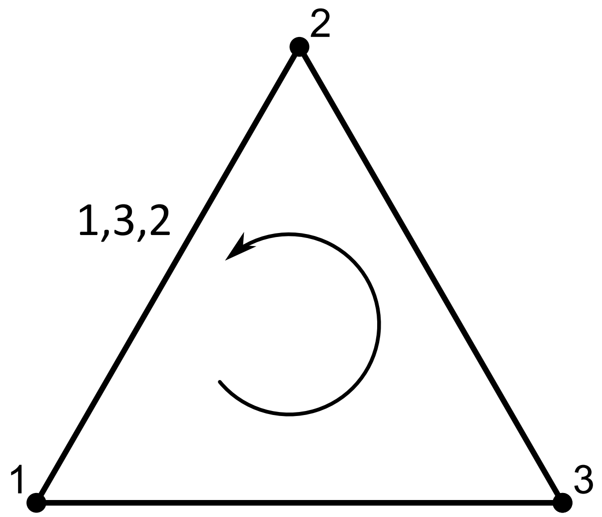
- When we define a set of triangle vertices we're defining them in a certain winding order that is either clockwise or counter-clockwise
- Each triangle consists of 3 vertices and we specify those 3 vertices in a winding order as seen from the center of the triangle
- Clockwise (left), counter-clockwise (right)



# Winding Order




- In the code:
- glDrawArrays:

```
float vertices[] = {
        V1, V2, V3, // clockwise (cw): 1,2,3
        V1, V3, V2 // counter-clockwise (ccw): 1,3,2
    };
```

- glDrawElements:

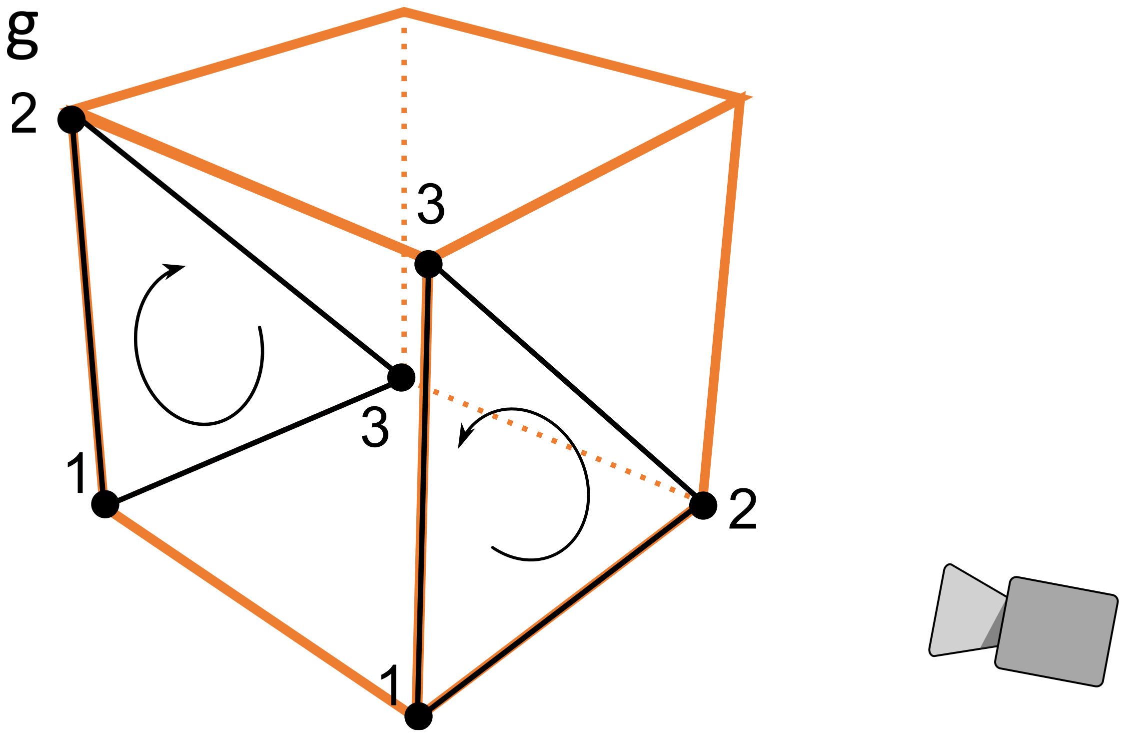
```
unsigned int indices[] = {  // note that we start from 0!
         0, 1, 2,  // clockwise (cw): 1,2,3
         0, 2, 1   // counter-clockwise (ccw): 1,3,2
    };
```

# Winding Order

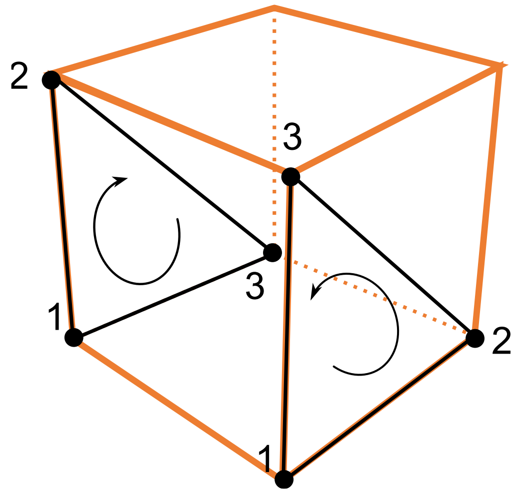
- Each set of 3 vertices forming a triangle primitive contains a winding order
- OpenGL uses this information to determine if a triangle is a front-facing or a back-facing triangle
- By default, triangles with counter-clockwise vertices are front-facing
- When defining the vertex order visualize the corresponding triangle as if it was facing you → each triangle should be counter-clockwise as if you're directly facing that triangle
- The actual winding order is calculated at the rasterization stage (after vertex shader) → vertices are then seen as from the viewer's point of view

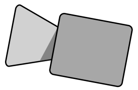
# Winding Order

- All the triangle vertices that the viewer is then facing are in the correct winding order (as we specified)
- Vertices of the triangles at the other side are now rendered in such a way that their winding order becomes reversed
- The result: facing triangles are seen as front-facing triangles and the triangles at the back are seen as back-facing triangles



# Winding Order



- In the vertex data we defined both triangles in ccw order (the front and back triangle as 1, 2, 3)
- From the viewer's direction the back triangle is rendered cw (1,2,3)
- Even if we specified the back triangle in ccw order, it is now rendered in a clockwise order
- This is exactly what we want to cull (discard) non-visible faces

# Face Culling

- Now that we know how to set the winding order of the vertices, we can start using OpenGL's face culling option (disabled by default)
- The cube vertex data was not defined with the ccw winding order (update→)

```
float cubeVertices[] = {
    -0.5f, -0.5f, -0.5f, 0.0f, 0.0f, // bottom-left
    0.5f, 0.5f, -0.5f, 1.0f, 1.0f, // top-right
    0.5f, -0.5f, -0.5f, 1.0f, 0.0f, // bottom-right
    0.5f, 0.5f, -0.5f, 1.0f, 1.0f, // top-right
    -0.5f, -0.5f, -0.5f, 0.0f, 0.0f, // bottom-left
    -0.5f, 0.5f, -0.5f, 0.0f, 1.0f, // top-left
    // front face
    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f, // bottom-left
    0.5f, -0.5f, 0.5f, 1.0f, 0.0f, // bottom-right
    0.5f, 0.5f, 0.5f, 1.0f, 1.0f, // top-right
    0.5f, 0.5f, 0.5f, 1.0f, 1.0f, // top-right
    -0.5f, 0.5f, 0.5f, 0.0f, 1.0f, // top-left
    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f, // bottom-left
    // left face
    -0.5f, 0.5f, 0.5f, 1.0f, 0.0f, // top-right
    -0.5f, 0.5f, -0.5f, 1.0f, 1.0f, // top-left
    -0.5f, -0.5f, -0.5f, 0.0f, 1.0f, // bottom-left
    -0.5f, -0.5f, -0.5f, 0.0f, 1.0f, // bottom-left
    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f, // bottom-right
    -0.5f, 0.5f, 0.5f, 1.0f, 0.0f, // top-right
    // right face
    0.5f, 0.5f, 0.5f, 1.0f, 0.0f, // top-left
    0.5f, -0.5f, -0.5f, 0.0f, 1.0f, // bottom-right
    0.5f, 0.5f, -0.5f, 1.0f, 1.0f, // top-right
    0.5f, -0.5f, -0.5f, 0.0f, 1.0f, // bottom-right
    0.5f, 0.5f, 0.5f, 1.0f, 0.0f, // top-left
    0.5f, -0.5f, 0.5f, 0.0f, 0.0f, // bottom-left
    // bottom face
    -0.5f, -0.5f, -0.5f, 0.0f, 1.0f, // top-right
    0.5f, -0.5f, -0.5f, 1.0f, 1.0f, // top-left
    0.5f, -0.5f, 0.5f, 1.0f, 0.0f, // bottom-left
    0.5f, -0.5f, 0.5f, 1.0f, 0.0f, // bottom-left
    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f, // bottom-right
    -0.5f, -0.5f, -0.5f, 0.0f, 1.0f, // top-right
    // top face
    -0.5f, 0.5f, -0.5f, 0.0f, 1.0f, // top-left
    0.5f, 0.5f, 0.5f, 1.0f, 0.0f, // bottom-right
    0.5f, 0.5f, -0.5f, 1.0f, 1.0f, // top-right
    0.5f, 0.5f, 0.5f, 1.0f, 0.0f, // bottom-right
    -0.5f, 0.5f, -0.5f, 0.0f, 1.0f, // top-left
    -0.5f, 0.5f, 0.5f, 0.0f, 0.0f // bottom-left
 };
```

# Face Culling

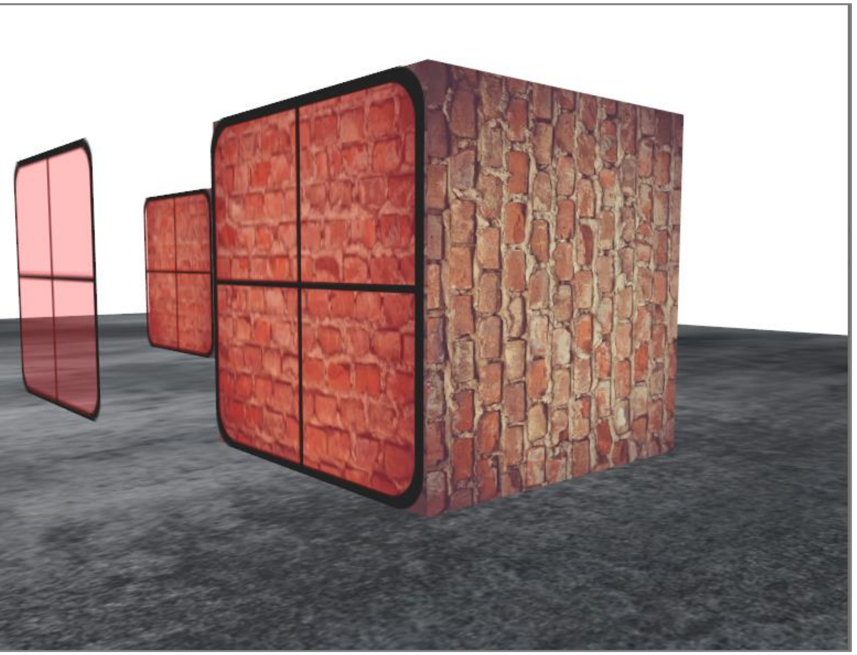
- To enable face culling we only have to enable OpenGL’s GL_CULL_FACE option:

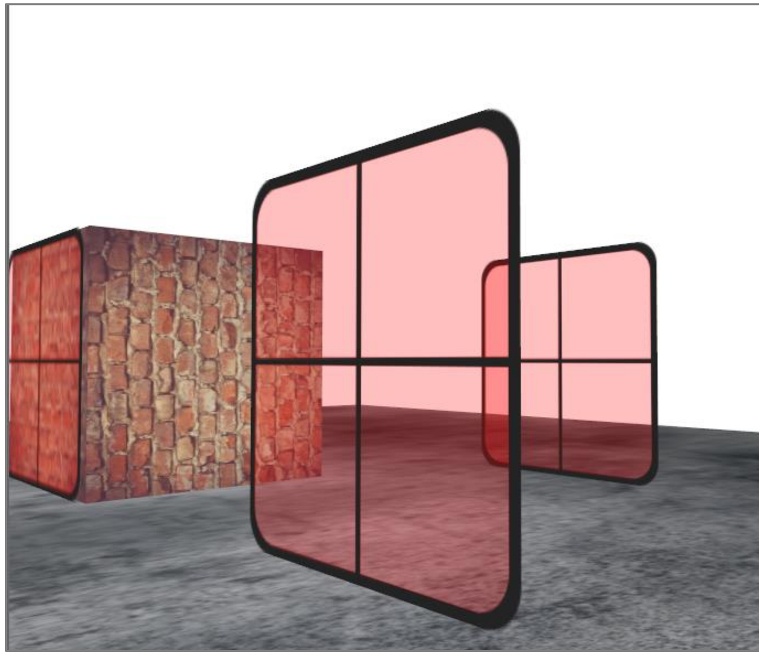
```
glEnable(GL_CULL_FACE);
```

- Now, all faces that are not front-faces are discarded → save 50% of performance on rendering fragments
- Only works with closed shapes like a cube, have to disable face culling again when we draw the grass leaves (front and back face)

```
...
glDisable(GL_CULL_FACE);
// render floor & windows
```

# Fly through the Box





# Face Culling

- Can change the type of face we want to cull:

```
glCullFace(GL_FRONT);
```

- The glCullFace function has three possible options:
  - GL_BACK: Culls only the back faces
  - GL_FRONT: Culls only the front faces
  - GL_FRONT_AND_BACK: Culls both the front and back faces

# Face Culling

- The initial value of glCullFace is GL_BACK
- We can also tell OpenGL to rather prefer clockwise faces as the front-faces instead of counter-clockwise faces via glFrontFace:

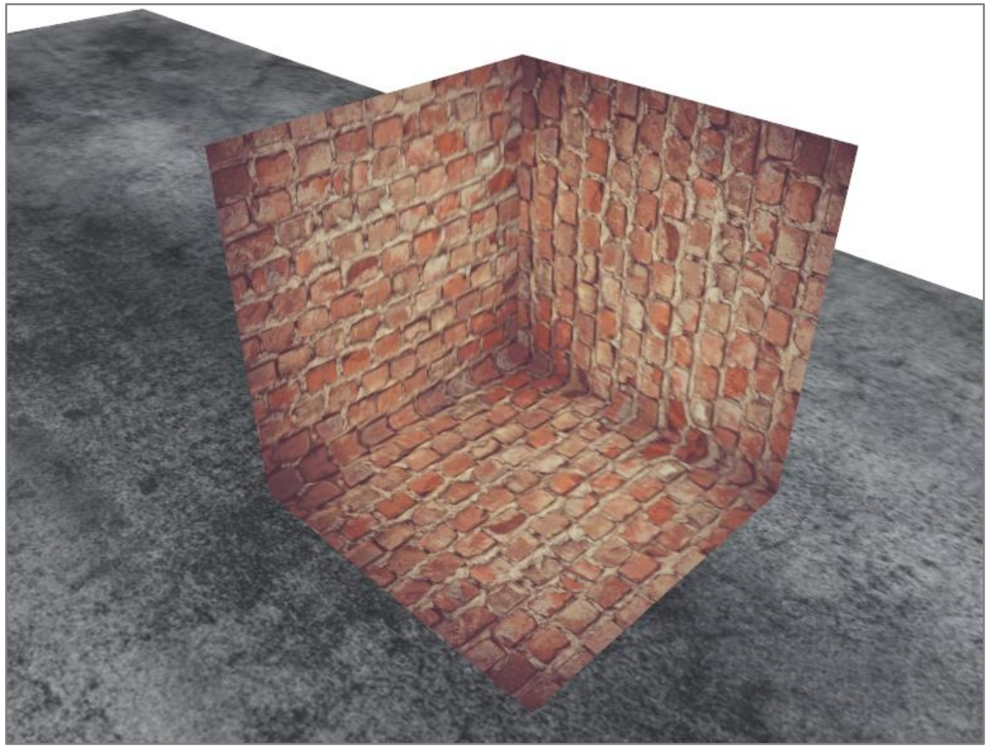
```
glFrontFace(GL_CCW);
```

- Default value is GL_CCW (counter-clockwise ordering)
- Other option: GL_CW (clockwise ordering)

# Face Culling

- Simple test: reverse the winding order by telling OpenGL that the front-faces are now determined by a clockwise ordering instead of a counter-clockwise ordering:

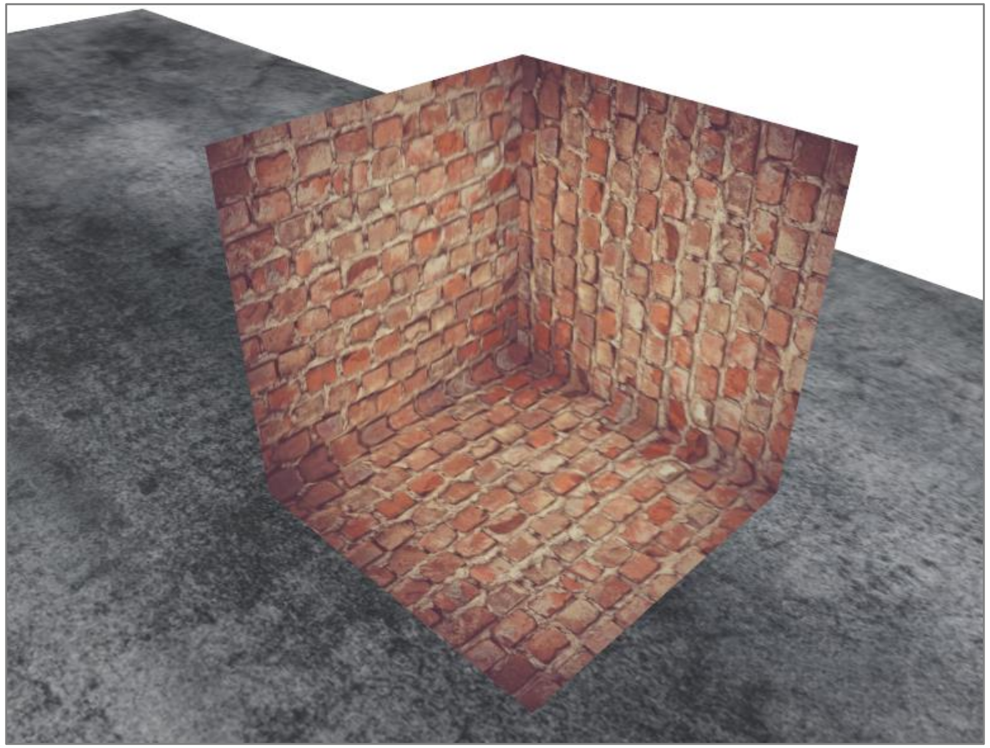
```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
glFrontFace(GL_CW);
```



# Face Culling

- Note that you can create the same effect by culling front faces with the default counter-clockwise winding order:

```
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
```



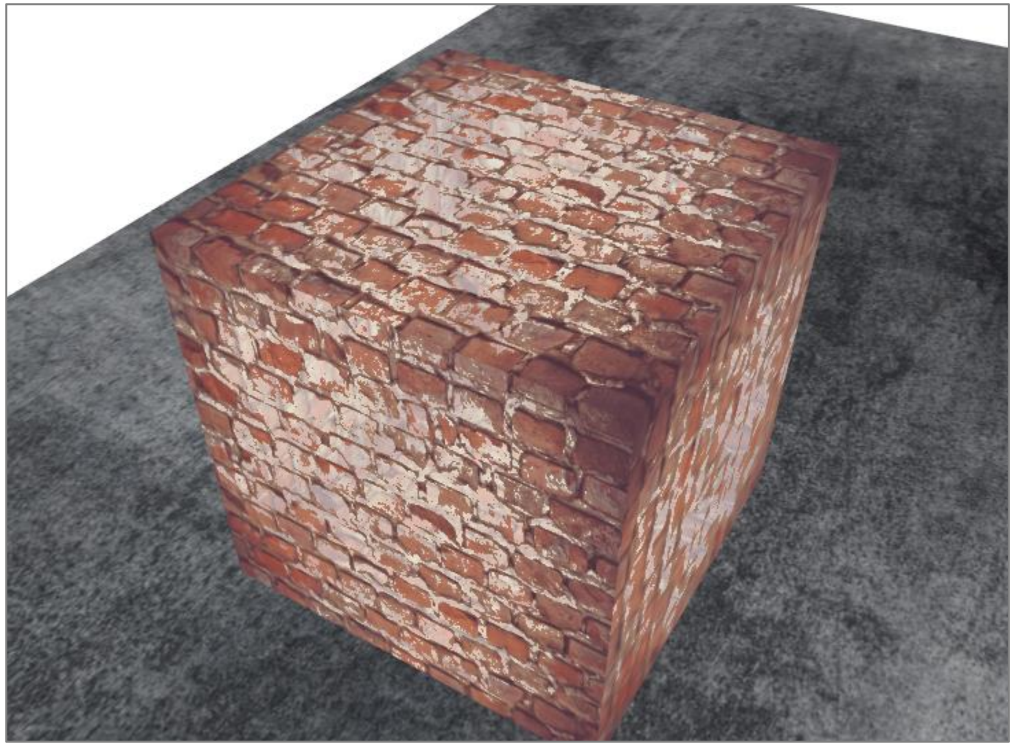
# Conclusion

- Face culling is a great for increasing performance minimal effort
- You do have to keep track of which objects will actually benefit from face culling and which objects shouldn’t be culled

# Example*

# Introduction

- Let us watch inside the box
- We will draw the back faces first
- Then, the front faces with transparency at certain regions



# Back Faces

- First, we draw the back faces

```
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
glBindVertexArray(cubeVAO);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, cubeTexture);
model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(-3.0f, 0.0001f, -1.0f));
shader.setMat4("model", model);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

# Front Faces

- Afterwards, we draw the front faces

```
glCullFace(GL_BACK);
model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(-3.0f, 0.0001f, -1.0f));
shader.setMat4("model", model);
glDrawArrays(GL_TRIANGLES, 0, 36);
glDisable(GL_CULL_FACE);
```

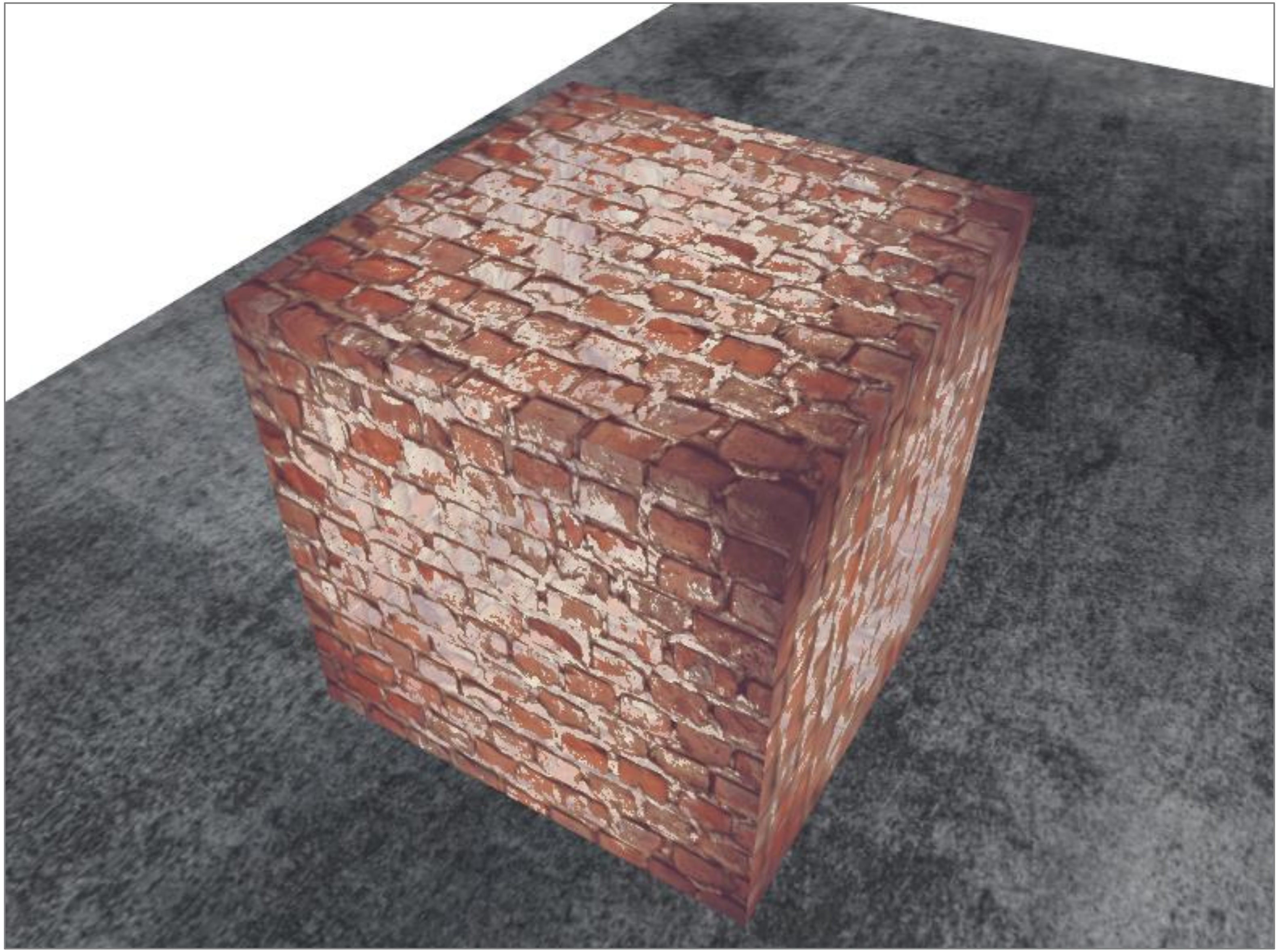
# Fragment Shader

- Fragment Shader → distinguish between front and back faces (gl_FrontFacing):

```
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;
uniform sampler2D texture1;
void main()
{
    FragColor = texture(texture1, TexCoords);
    if(gl_FrontFacing && length(FragColor.rgb)>0.95)
       FragColor=vec4(1,1,1,0.5);
}
```

# F5…

- … some parts are transparent



# Questions???