Computer Graphics II – Framebuffer

Kai Lawonn

Introduction

- So far several types of screen buffers:
  - Color buffer (write color values)
  - Depth buffer (write depth information)
  - Stencil buffer (allows discarding fragments)
- The combination of these buffers is called a framebuffer
- OpenGL gives the flexibility to define own framebuffers (own color, depth and stencil buffer)

Introduction

- Rendering operations done on top of the render buffers attached to the default framebuffer
- Default framebuffer is automatically created and configured (GLFW does this for us)
- Creating own framebuffer → get an additional means to render to
- Framebuffer allows, e.g., to create mirrors, post-processing effects

Creating a Framebuffer

Introduction

- Like any other object, create a framebuffer object (FBO) with glGenFramebuffers:

```
unsigned int framebuffer;
glGenFramebuffers(1, &framebuffer);
```

- Usage functions similar to all the other object's:
  - 1. Create a FBO
  - 2. Bind it as the active framebuffer
  - 3. Do some operations and unbind the framebuffer

```
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```

Introduction

- By binding to the GL_FRAMEBUFFER target all the next read and write framebuffer operations will affect the currently bound framebuffer
- Possible to bind a framebuffer to a read or write target (binding GL_READ_FRAMEBUFFER, GL_DRAW_FRAMEBUFFER)
- GL_READ_FRAMEBUFFER read operations, e.g., glReadPixels
- GL_DRAW_FRAMEBUFFER write operations, e.g., rendering, clearing
- Mostly bind to both with GL_FRAMEBUFFER

Introduction

- Framebuffer is not complete
- For completion the following requirements have to be satisfied:
  - Have to attach at least one buffer (color, depth or stencil buffer)
  - At least one color attachment
  - All attachments should be complete as well (reserved memory)
  - Each buffer should have the same number of samples

Introduction

- Need to create some kind of attachment for the framebuffer
- After completing all requirements, check the status with glCheckFramebufferStatus(GL_FRAMEBUFFER):

```
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
       cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << endl;
```

Introduction

- All subsequent rendering operations will now render to the attachments of the currently bound framebuffer
- Our framebuffer is not the default framebuffer → rendering commands no impact on the visual output
- Therefore, it is called off-screen rendering (rendering to a different framebuffer)
- Ensure all rendering operations have a visual impact, make the default framebuffer active again by binding to 0:

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Introduction

- After all framebuffer operations, delete the framebuffer object:

```
glDeleteFramebuffers(1, &fbo);
```

- Before the completeness check is executed, need to attach one or more attachments to the framebuffer
- An attachment is a memory location that can act as a buffer for the framebuffer
- When creating an attachment we have two options to take: textures or renderbuffer objects

Texture Attachments

Introduction

- Attaching a texture to a framebuffer → rendering commands write to the texture (like a normal color/depth or stencil buffer)
- Advantage: rendering operations stored as a texture, that can used in the shaders
- Creating a texture for a framebuffer is roughly the same as a normal texture:

```
unsigned int textureColorbuffer;
glGenTextures(1, &textureColorbuffer);
glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB,
GL_UNSIGNED_BYTE, NULL);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Introduction

- Main differences: set the dimensions equal to the screen size (not required) and pass NULL as the texture's data parameter
- For this texture, allocating memory only (not actually filling it)
- Filling happens as soon as we render to the framebuffer
- Also note: we do not care about wrapping/mipmapping (won't needing those in most cases)

```
unsigned int textureColorbuffer;
glGenTextures(1, &textureColorbuffer);
glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB,
GL_UNSIGNED_BYTE, NULL);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Render the screen to a smaller/larger texture? → Call glViewport again (before rendering to your framebuffer) with the new dimensions, otherwise only a small part of the texture or screen would be drawn onto the texture

glFrameBufferTexture2D

- Finally, attach the texture to the framebuffer:

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
texture, 0);
```

- The glFrameBufferTexture2D has the following parameters:
  - Target: the targeting framebuffer type (draw, read or both).
  - Attachment: the type of attachment, here a color attachment (several attachments possible)
  - Textarget: the type of the texture to attach
  - Texture: the actual texture to attach
  - Level: the mipmap level (keep this at 0)

Attachments

- Aside from the color attachments: depth, stencil texture to FBO
- Attach a depth attachment: GL_DEPTH_ATTACHMENT
  - Note that the texture's format and internalformat type: GL_DEPTH_COMPONENT to reflect the depth buffer's storage format
- Attach a stencil buffer: GL_STENCIL_ATTACHMENT
  - Texture's formats as GL_STENCIL_INDEX

Attachments

- Possible to attach a depth and a stencil buffer as a single texture
- Each 32 bit value of the texture then consists for 24 bits of depth and 8 bits of stencil information
- Use GL_DEPTH_STENCIL_ATTACHMENT type and configure the texture's formats to contain combined depth and stencil values:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH24_STENCIL8, SCR_WIDTH, SCR_HEIGHT, 0,
GL_DEPTH_STENCIL, GL_UNSIGNED_INT_24_8, NULL);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
GL_TEXTURE_2D, texture, 0);
```

Renderbuffer Object Attachments

Renderbuffer Object

- Renderbuffer objects (RBOs) are a possible type of framebuffer attachments
- Like a texture image, an RBO is an actual buffer (an array of bytes, integers, pixels,...)
- RBO advantage that it stores its data in OpenGL's native rendering format → optimized for off-screen rendering to a framebuffer

Renderbuffer Object

- RBOs store render data directly into their buffer without texture-specific formats conversions → faster as a writeable storage medium
- However, RBOs are generally write-only, thus you cannot read from them (like with texture-access)
- Possible to read via glReadPixels from the currently bound framebuffer, but not directly from the attachment itself

Renderbuffer Object

- Because their data is already in its native format → quite fast when writing data or copying their data to other buffers
- Operations like switching buffers are thus quite fast when using RBOs (glfwSwapBuffers implemented with RBOs: write to a renderbuffer image, and swap to the other one at the end)
- Renderbuffer objects are perfect for these kind of operations

Renderbuffer Object

- Creating an RBO looks similar to the framebuffer's code:

```
unsigned int rbo;
glGenRenderbuffers(1, &rbo);
```

- Similarly bind the RBO so all subsequent renderbuffer operations affect the current rbo:

```
glBindRenderbuffer(GL_RENDERBUFFER, rbo);
```

Renderbuffer Object

- RBOs are generally write-only, often used as depth and stencil attachments (mostly do not need to read values from them, but care about depth and stencil testing)
- Need depth and stencil values for testing, but do not need to sample these values → RBO suits this perfectly
- Not sampling from these buffers → RBO is generally preferred since it’s more optimized

Renderbuffer Object

- Creating a depth and stencil renderbuffer object is done by calling the glRenderbufferStorage function:

```
glRenderbufferStorage(GL_RENDERBUFFER,GL_DEPTH24_STENCIL8,SCR_WIDTH,SCR_HEIGHT);
```

- Creating a RBO is similar to texture objects, difference is that this object is specifically designed to be used as an image, instead of a general purpose data buffer like a texture
- Here GL_DEPTH24_STENCIL8 is used as the internal format, which holds both the depth and stencil buffer with 24 and 8 bits

Renderbuffer Object

- Now, attach the renderbuffer object:

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
GL_RENDERBUFFER, rbo);
```

- RBOs provide some optimizations in framebuffer projects, but it is important when to use RBOs and when to use textures
- General rule: never need to sample data from a specific buffer → use an RBO
- Sample data from a specific buffer like colors or depth values → use a texture attachment
- Performance-wise it doesn't have an enormous impact though

Rendering to a Texture

Rendering to a Texture

- Now it’s time to render the scene into a color texture attached to a framebuffer object
- Then draw this texture over a simple quad that spans the whole screen $\rightarrow$ visual output exactly the same as without a framebuffer
- Why is this useful, we will see ...

Rendering to a Texture

- Create an actual framebuffer object and bind it:

```
unsigned int framebuffer;
glGenFramebuffers(1, &framebuffer);
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```

Rendering to a Texture

- Next, create a texture image and attach it as a color attachment to the framebuffer
- Set the dimensions equal to the width and height of the window and keep its data uninitialized:

```
unsigned int textureColorbuffer;
glGenTextures(1, &textureColorbuffer);
glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB,
GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
textureColorbuffer, 0);
```

Rendering to a Texture

- Want to make sure OpenGL is able to do depth testing (and optionally stencil testing) → add a depth (and stencil) attachment to the framebuffer as well
- Only for sampling the color buffer and not the other buffers → create an RBO (remember it is a good choice…)

Rendering to a Texture

- Create an RBO with a depth and stencil attachment
- Set its internal format to GL_DEPTH24_STENCIL8:

```
unsigned int rbo;
glGenRenderbuffers(1, &rbo);
glBindRenderbuffer(GL_RENDERBUFFER, rbo);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, SCR_WIDTH,
SCR_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

- Once enough memory for the RBO is allocated: unbind it

Rendering to a Texture

- Create an RBO with a depth and stencil attachment
- Set its internal format to GL_DEPTH24_STENCIL8:

```
unsigned int rbo;
glGenRenderbuffers(1, &rbo);
glBindRenderbuffer(GL_RENDERBUFFER, rbo);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, SCR_WIDTH,
SCR_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

- Once enough memory for the RBO is allocated: unbind it

Rendering to a Texture

- Then, attach the RBO to the depth and stencil attachment:

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
GL_RENDERBUFFER, rbo);
```

- Also check if framebuffer is complete:

```
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
        cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << endl;
```

- Unbind the framebuffer to make sure to not rendering to the wrong framebuffer

Rendering to a Texture

- Altogether:

```
unsigned int framebuffer;
glGenFramebuffers(1, &framebuffer);
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);

unsigned int textureColorbuffer;
glGenTextures(1, &textureColorbuffer);
glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, textureColorbuffer, 0);

unsigned int rbo;
glGenRenderbuffers(1, &rbo);
glBindRenderbuffer(GL_RENDERBUFFER, rbo);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, SCR_WIDTH, SCR_HEIGHT);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rbo);
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
        cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << endl;
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Rendering to a Texture

- Now need to render to the framebuffer's buffers instead of the default framebuffers: bind to the framebuffer object
- Subsequent rendering commands will influence the currently bound framebuffer
- Depth and stencil operations will also read from the currently bound framebuffer's depth and stencil attachments (if available)
- If we omit a depth buffer, depth testing operations will no longer work (because there's not a depth buffer present in the currently bound framebuffer)

Rendering to a Texture

- Draw the scene to a single texture:
    1. Render the scene as usual with the new active framebuffer bound
    2. Bind to the default framebuffer
    3. Draw a quad that spans the entire screen with the new framebuffer's color buffer as its texture
- We draw the same scene as in the previous lecture

Quad's Shader

- To draw the quad create a new set of simple shaders
- We use vertex coordinates as normalized device coordinates (NDCs)

```
float quadVertices[] = {
        // positions   // texCoords
        -1.0f,  1.0f,  0.0f, 1.0f,
        -1.0f, -1.0f,  0.0f, 0.0f,
         1.0f, -1.0f,  1.0f, 0.0f,

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

Quad's Shader

- The vertex shader looks like this:

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec2 aTexCoords;

out vec2 TexCoords;

void main()
{
    TexCoords = aTexCoords;
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
}
```

Quad's Shader

- The fragment shader:

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D screenTexture;

void main()
{
    vec3 col = texture(screenTexture, TexCoords).rgb;
    FragColor = vec4(col, 1.0);
}
```

Render

- Render iteration of the framebuffer procedure:

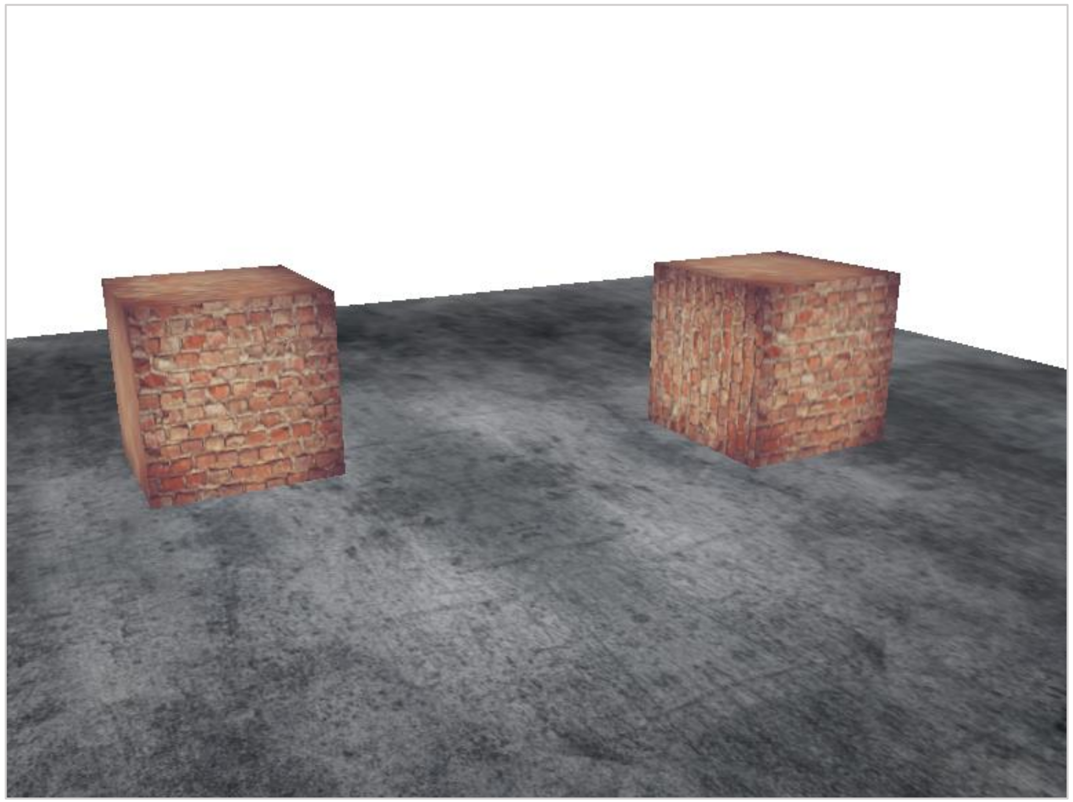
```
// first pass
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glClearColor(1.f, 1.f, 1.f, 1.f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);
DrawScene();
// second pass
glBindFramebuffer(GL_FRAMEBUFFER, 0); // back to default
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
screenShader.use();
glBindVertexArray(quadVAO);
glDisable(GL_DEPTH_TEST);
glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
glDrawArrays(GL_TRIANGLES, 0, 6);
```

Notes

- Each framebuffer has its own set of buffers, we want to clear each of those buffers with the appropriate bits set by calling glClear
- When drawing the quad, disable depth testing (depth testing is not important here)
- Enable depth testing again when we draw the normal scene though

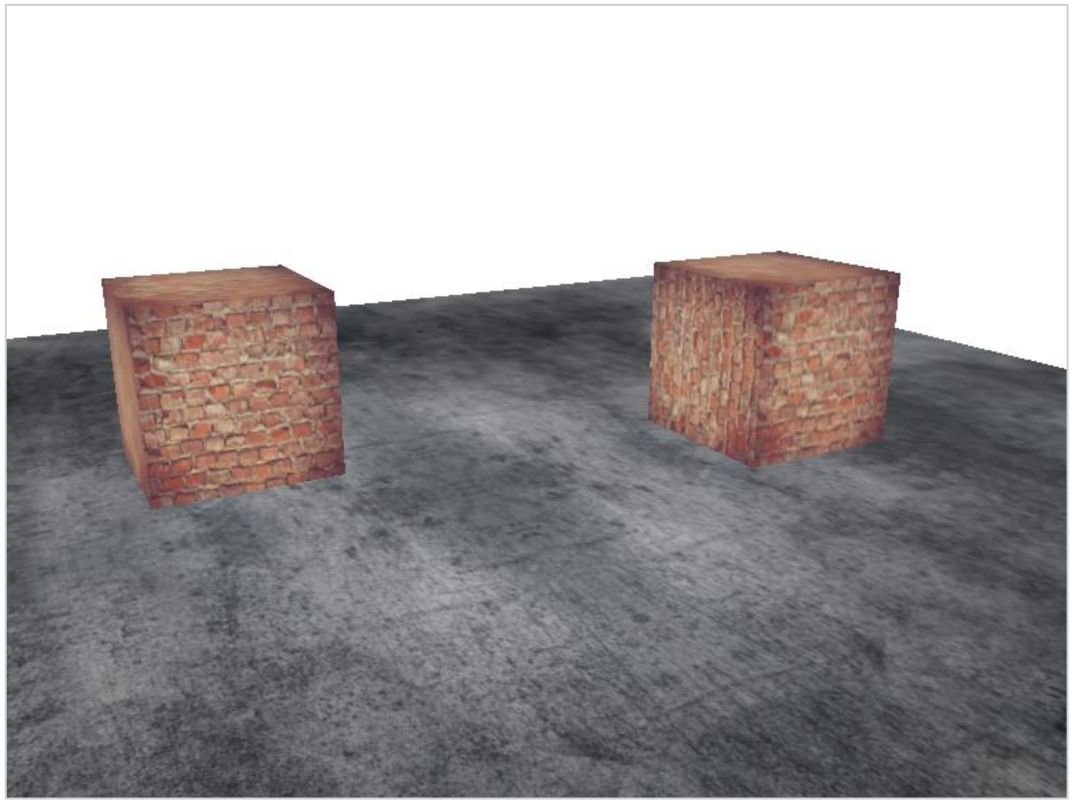
F5…

- … it works



F5…

- This time, we rendered the output on a simple quad
- So what was the use of this again?
- Now access freely of each of the pixels of the completely rendered scene → create some interesting effects in the fragment shader (post-processing effects)



Post–Processing

Post–Processing

- Now we can start to create some interesting effects by manipulating the texture data
- Now, we want to apply popular post-processing effects
- Let’s start with one of the simplest post-processing effects

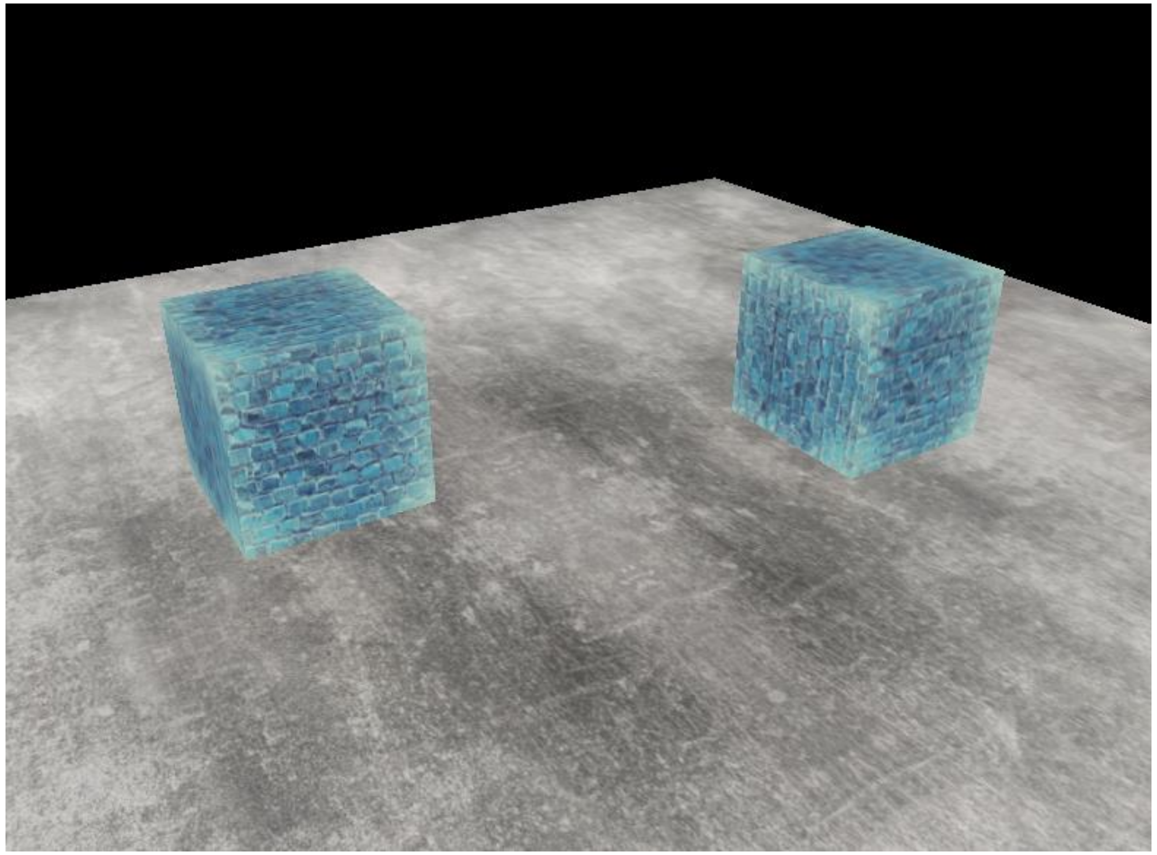
Inversion

- With access to each of the colors of the render output so it’s not so hard to return the inverse of these colors in the fragment shader
- We’re taking the color of the screen texture and inverse it by subtracting it from 1.0:

```
void main()
{
    vec3 col = texture(screenTexture, TexCoords).rgb;
    FragColor = vec4(1.0 - col, 1.0);
}
```

F5…

- … inversion is a simple post-processing effect
- Scene has all its colors inversed with a single line of code in the fragment shader



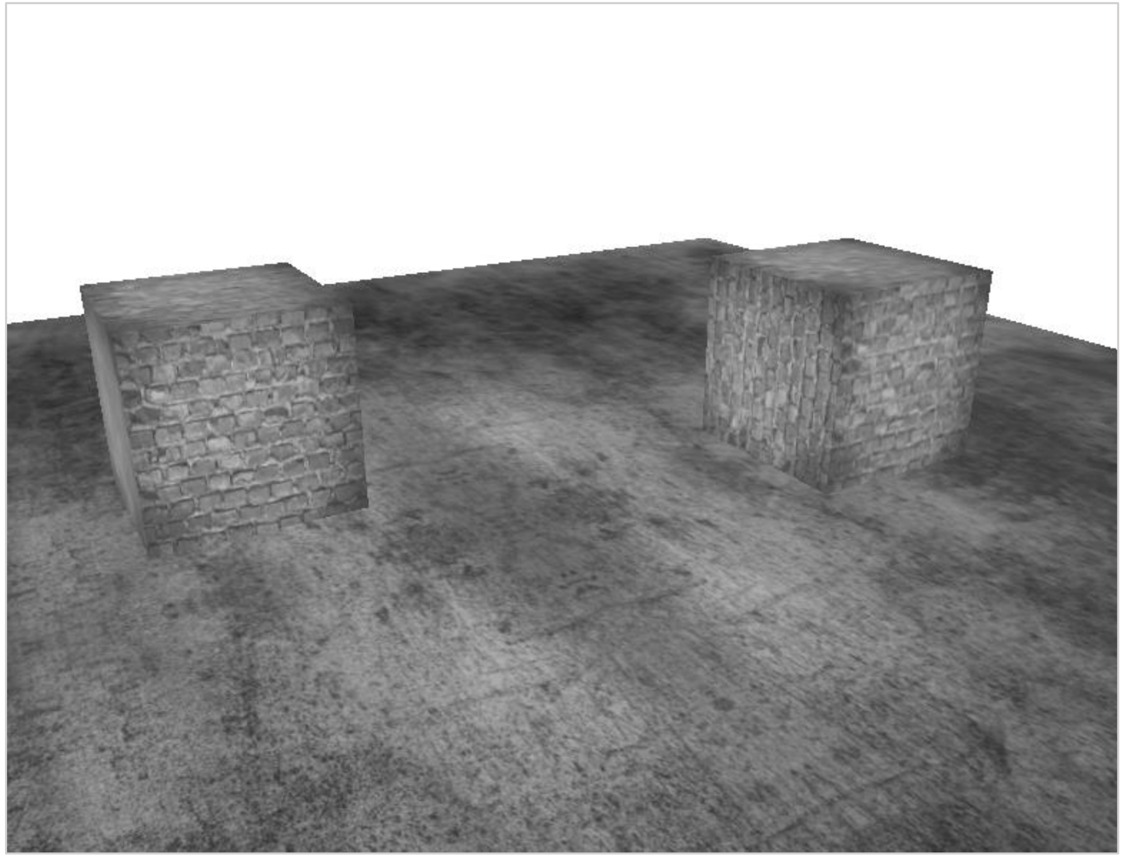
Grayscale

- Now, remove all colors from → let us grayscaling the entire image
- An easy way: average the color components:

```
void main()
{
    vec3 col = texture(screenTexture, TexCoords).rgb;
    float average = (col.r + col.g + col.b) / 3.0;
    FragColor = vec4(average, average, average, 1.0);
}
```

F5…

- Creates good results, but the human eye tends to be more sensitive to green colors and the least to blue → for physically accurate results use weighted channels



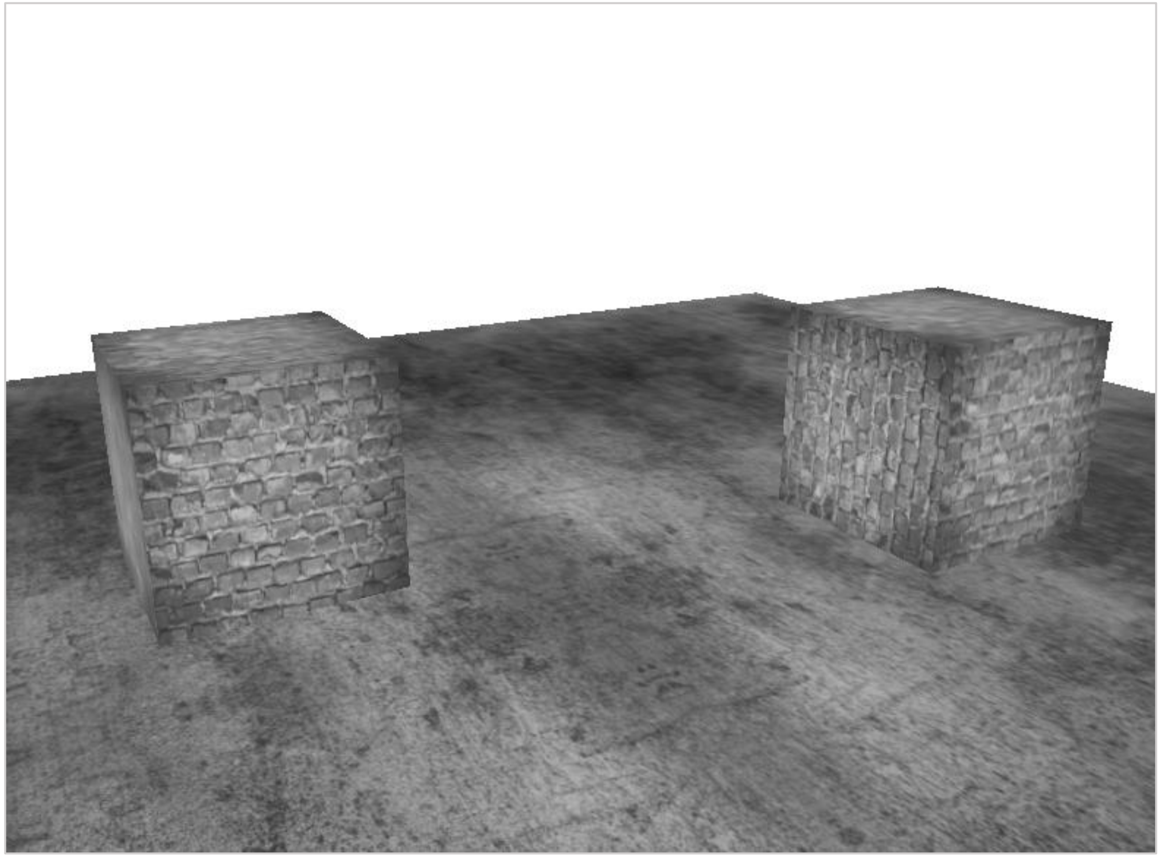
Grayscale

- We use weighted channels for grayscaling:

```
void main()
{
    vec3 col = texture(screenTexture, TexCoords).rgb;
    float average = 0.2126 * col.r + 0.7152 * col.g + 0.0722 * col.b;
    FragColor = vec4(average, average, average, 1.0);
}
```

F5…

- … difference is hard to notice, but with more complicated scenes, a weighted grayscaling effect tends to be more realistic



Kernel Effects

- Another advantage is that we can sample color values from other parts of the texture
- For example: take a small area around the current texture coordinate and sample multiple texture values around the current texture value
- We can then create interesting effects by combining them in creative ways

Kernel Effects

- A kernel (or convolution matrix) is a small matrix-like array of values centered on the current pixel that multiplies surrounding pixel values by its kernel values and adds them all together to form a single value
- Basically add a small offset to the texture coordinates in surrounding directions of the current pixel and combine the results based on the kernel
- Example of a kernel:

$$\begin{pmatrix} 2 & 2 & 2 \\ 2 & -15 & 2 \\ 2 & 2 & 2 \end{pmatrix}$$

Kernel Effects

- This kernel takes 8 surrounding pixel values and multiplies them by 2 and the current pixel by -15
- Basically it multiplies the surrounding pixels by a weight determined in the kernel and balances the result by multiplying the current pixel by a large negative weight

$$\begin{pmatrix} 2 & 2 & 2 \\ 2 & -15 & 2 \\ 2 & 2 & 2 \end{pmatrix}$$

Sum up to 1?

Most kernels sum up to 1 if you add all the weights together.

If they don’t add up to 1 it means that the resulting texture color ends brighter or darker than the original texture value.

Kernel Effects

- Kernels useful for post-processing
- We have to adapt the fragment shader to actually support kernels
- We make the assumption that each kernel is a 3x3 kernel (which most kernels are)

Kernel Effects

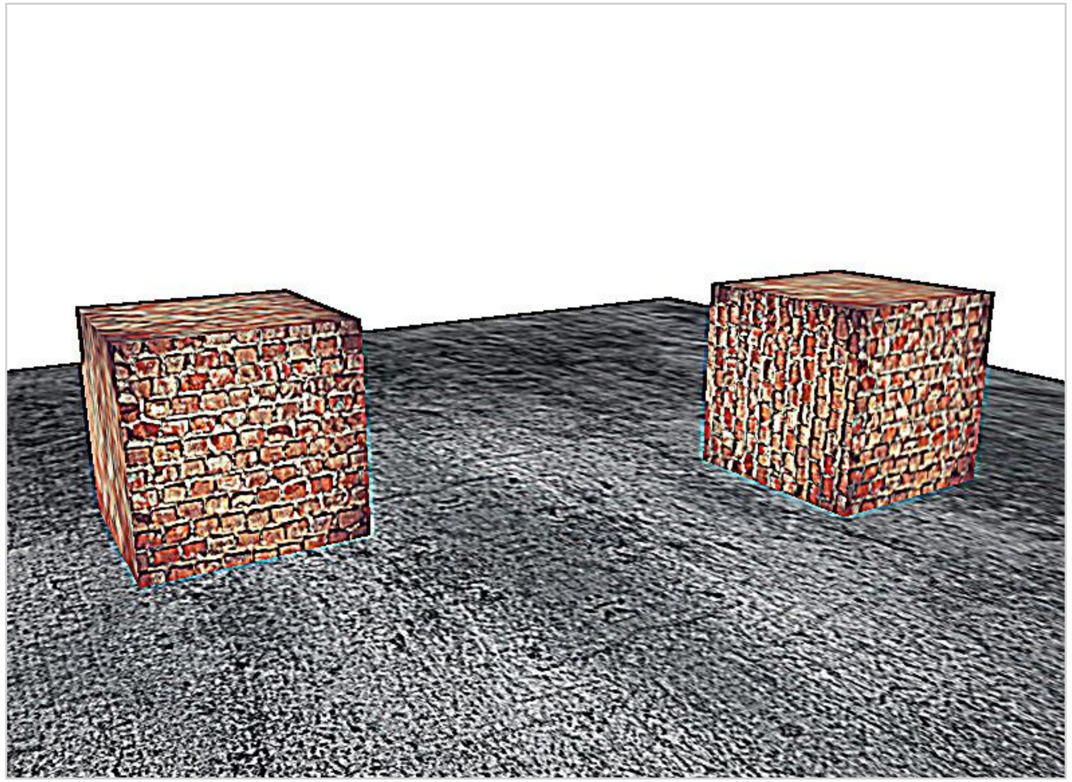
```
const float offset = 1.0 / 300.0;
void main()
{
vec2 offsets[9] = vec2[](vec2(-offset, offset), // top-left
                            vec2( 0.0f,offset), // top-center
                            vec2( offset, offset), // top-right
                            vec2(-offset, 0.0f), // center-left
                            vec2( 0.0f,0.0f), // center-center
                            vec2( offset, 0.0f), // center-right
                            vec2(-offset, -offset), // bottom-left
                            vec2( 0.0f, -offset), // bottom-center
                            vec2( offset, -offset) // bottom-right
);

float kernel[9] = float[](
-1, -1, -1,
-1, 9, -1,
-1, -1, -1
);
```

Kernel Effects

```
vec3 sampleTex[9];
for(int i = 0; i < 9; i++)
    sampleTex[i] = vec3(texture(screenTexture, TexCoords.st + offsets[i]));

vec3 col = vec3(0.0);
for(int i = 0; i < 9; i++)
    col += sampleTex[i] * kernel[i];

FragColor = vec4(col, 1.0);
}
```

F5…

- … this sharpen kernel looks like this:



Blur

- A kernel that creates a blur effect is defined as follows

$$\frac{1}{16} \cdot \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

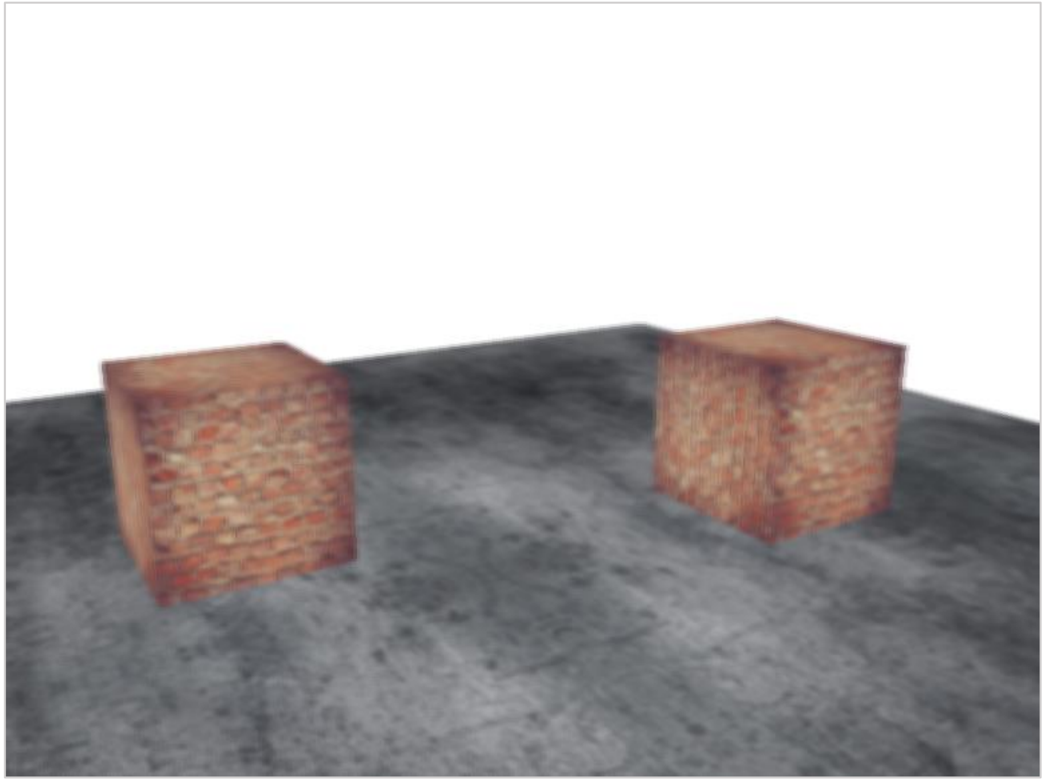
Blur

- All values add up to 16, so divide each value by 16 (otherwise result in an extremely bright color)
- The resulting kernel array would then become:

```
float kernel[9] = float[](
1.0 / 16, 2.0 / 16, 1.0 / 16,
2.0 / 16, 4.0 / 16, 2.0 / 16,
1.0 / 16, 2.0 / 16, 1.0 / 16
);
```

F5…

- … such a blur effect creates interesting possibilities (drunk effects, not wearing glasses, …)



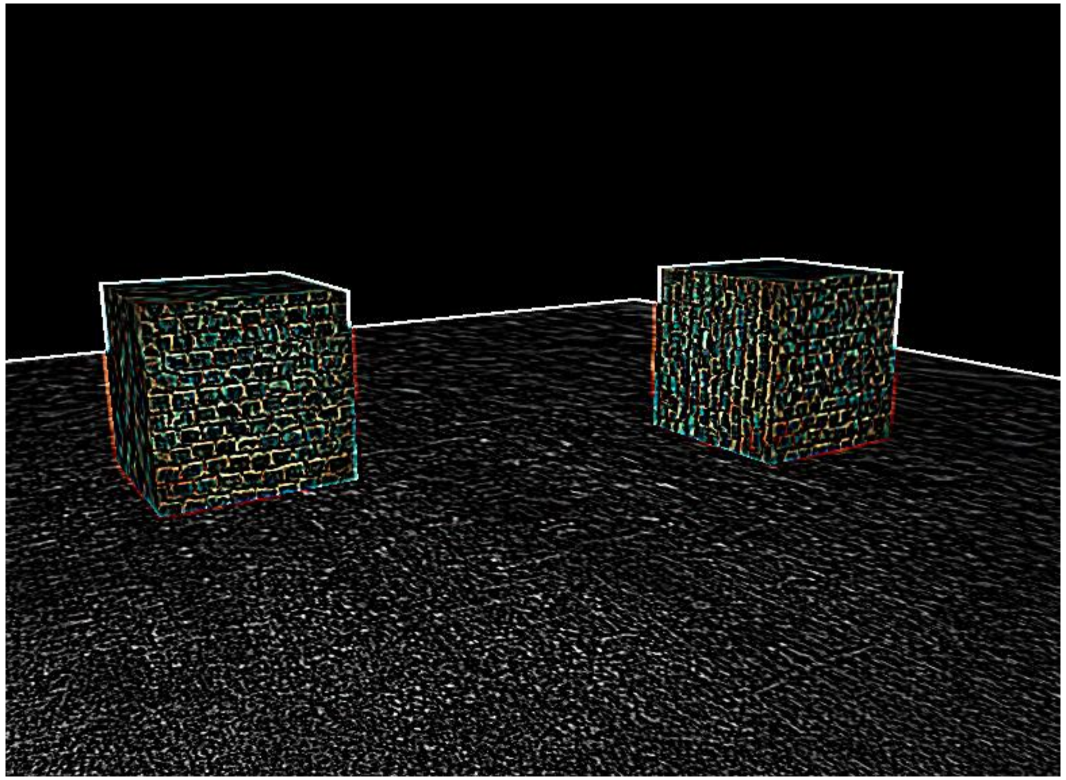
Edge Detection

- An edge-detection kernel that is similar to the sharpen kernel:

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

F5…

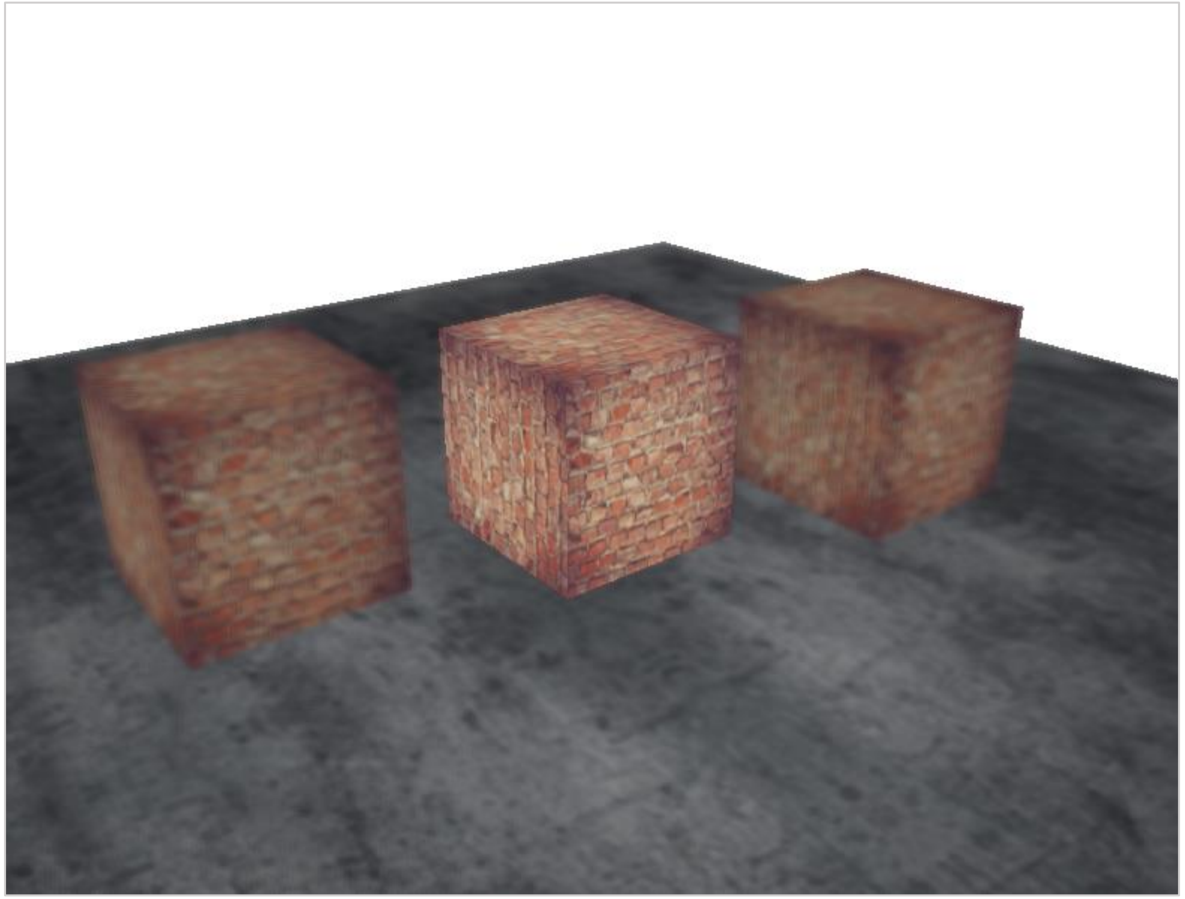
- … highlights all edges and darkens the rest
- Used in tools like Photoshop (graphic card’s ability to process fragments in parallel, allows to manipulate images on a per-pixel basis in real-time with relative ease)
- Image-editing tools therefore tend to use graphics cards more often for image-processing



Multiple Color-Attachments*

Multiple Color-Attachments

- Goal:
- Apply blur effect to objects
- Keep one box as the focus object



Multiple Color-Attachments

- First, we need to add another texture color buffer:

```
unsigned int framebuffer;
glGenFramebuffers(1, &framebuffer);
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
unsigned int textureColorbuffer;
glGenTextures(1, &textureColorbuffer);
glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, textureColorbuffer, 0);

unsigned int textureColorbuffer2;
glGenTextures(1, &textureColorbuffer2);
glBindTexture(GL_TEXTURE_2D, textureColorbuffer2);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB,
GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D,
textureColorbuffer2, 0);
```

Multiple Color-Attachments

- Define outputs into which the fragment shader will be written

```
GLenum color_attachments[] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 };
glDrawBuffers(2, color_attachments);
```

Multiple Color-Attachments

- Define focus object with uniform int ‚focus‘

```
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glEnable(GL_DEPTH_TEST);
glClearColor(1.f, 1.f, 1.f, 1.f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
shader.use();
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);
shader.setMat4("view", view);
shader.setMat4("projection", projection);
shader.setInt("focus", 0);
// 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);
// floor
glBindVertexArray(planeVAO);
glBindTexture(GL_TEXTURE_2D, floorTexture);
shader.setMat4("model", glm::mat4(1.0f));
glDrawArrays(GL_TRIANGLES, 0, 6);
model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(0.5f, 0.0f, -0.5f));
shader.setMat4("model", model);
shader.setInt("focus", 1);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

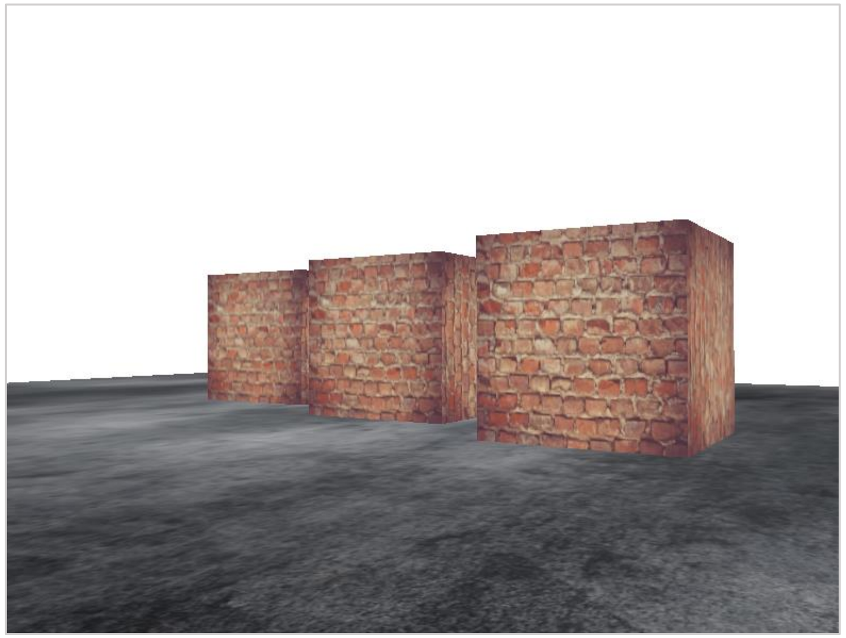
Multiple Color-Attachments

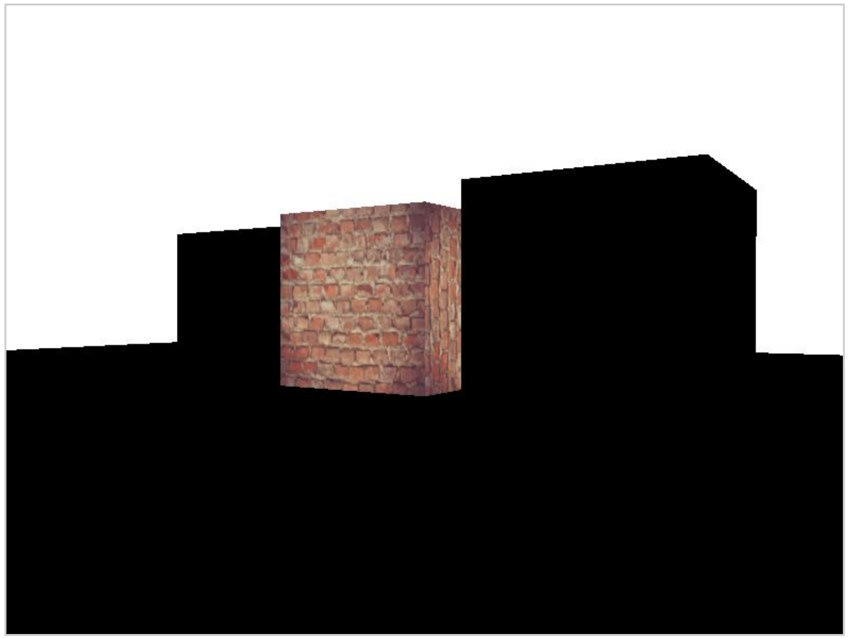
- In the fragment shader output of two colors (two attachments)

```
#version 330 core

layout (location = 0) out vec4 FragColor;
layout (location = 1) out vec4 FragColor2;

in vec2 TexCoords;
uniform int focus;
uniform sampler2D texture1;

void main()
{
    FragColor = texture(texture1, TexCoords);
    if(focus==1)
        FragColor2 = texture(texture1, TexCoords);
}
```

Multiple Color-Attachments

- Left FragColor, Right FragColor2
- Black regions, because we still have the same depth test, it passes and replace the color in FragColor2





Multiple Color-Attachments

- Now it is time to process both textures on the quad
- Add the second texture:

```
screenShader.use();
screenShader.setInt("screenTexture", 0);
screenShader.setInt("screenTexture2", 1);
```

Multiple Color-Attachments

- In the render loop bind both textures:

```
screenShader.use();
glBindVertexArray(quadVAO);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, textureColorbuffer2);
glDrawArrays(GL_TRIANGLES, 0, 6);
```

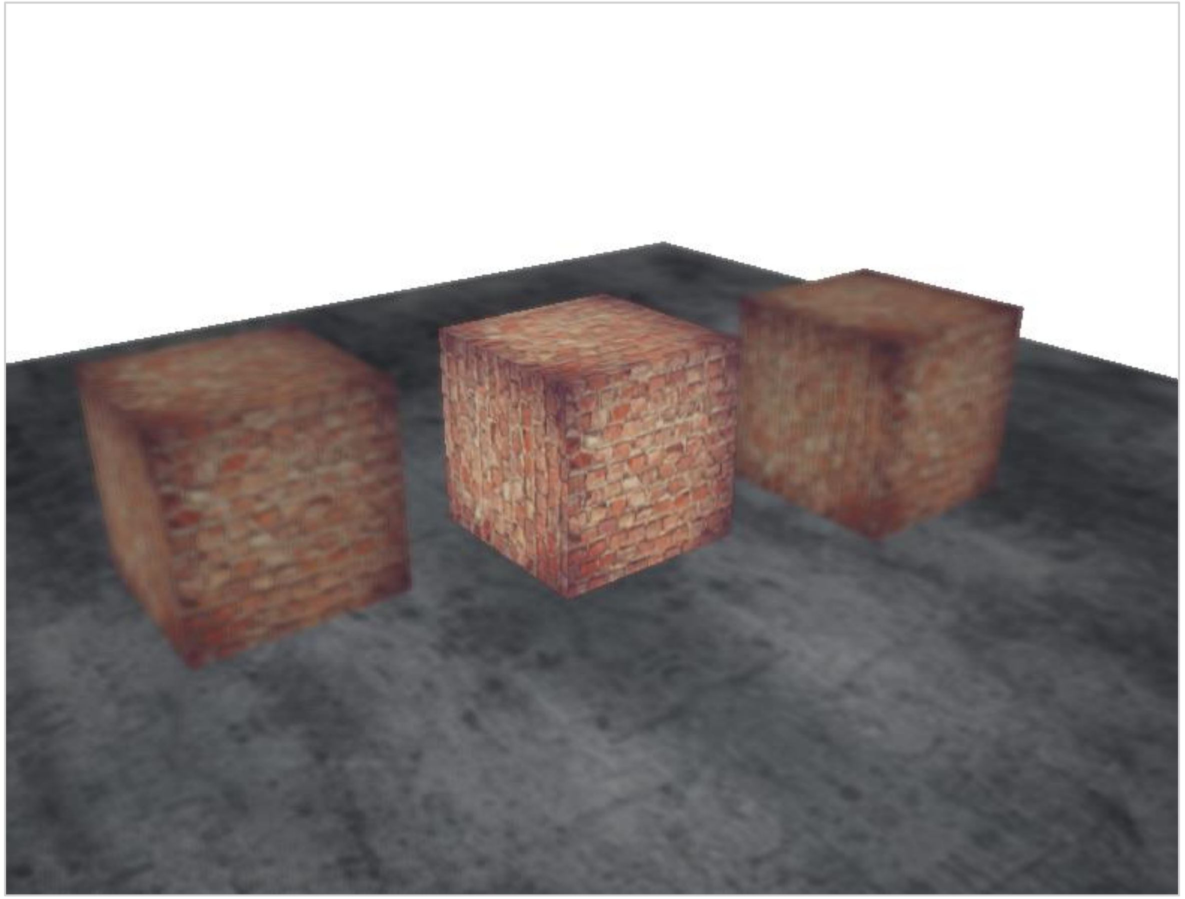
Multiple Color-Attachments

- Process them in the fragment shader

```
uniform sampler2D screenTexture;
uniform sampler2D screenTexture2;
…
void main(){
…
vec3 sampleTex[9];
for(int i = 0; i < 9; i++)
  sampleTex[i] = vec3(texture(screenTexture, TexCoords.st + offsets[i]));
vec3 col = vec3(0.0);
for(int i = 0; i < 9; i++)
     col += sampleTex[i] * kernel[i] * 0.75;

     vec3 col2 = vec3(texture(screenTexture2, TexCoords.st));
     if(all(col2 == vec3(0)))
          FragColor = vec4(col, 1.0);
     else
          FragColor = vec4(col2, 1.0);
}
```

F5…

- … nice



Questions???