FRAMEBUFFERS

The combination of the color buffer, depth buffer, and stencil buffer is stored in GPU memory known as the **framebuffer**.
OpenGL gives us the flexibility to define our own framebuffers and, thus, define our own color (and optionally a depth and stencil) buffer.

The rendering options we've done so far were all done on top of the render buffers attached to the **default framebuffer**.
- The default framebuffer is created and configured when you create your window (GLFW does this for us). By creating our own framebuffer, we can get an additional target to render to.
- Rendering your scene to a different framebuffer allows us to to use that result create mirrors in a scene, add post-processing effects, etc.

Creating a Framebuffer
Generate and bind an OpenGL framebuffer object.
- [Generating and Binding a Framebuffer]()

NOTE: It is possible to bind a framebuffer to a read or write target specifically by binding to `GL_READ_FRAMEBUFFER` or `GL_DRAW_FRAMEBUFFER`, respectively.
- The framebuffer bound to GL_READ_FRAMEBUFFER is used for all read operations like glReadPixels.
- The framebuffer bound to GL_DRAW_FRAMEBUFFER is used as the destination for rendering, clearing, and other write operations.

We cannot use our framebuffer yet because it is not **complete**. To complete the framebuffer, the following requirements must be satisfied:
1. At least one buffer is attached.
2. There is at least one color attachment.
3. All atachments are complete as well (reserved memory).
4. Each buffer has the same number of samples (samples are discussed in a later chapter).

We should use `glCheckFramebufferStatus(GL_FRAMEBUFFER)` to check if the framebuffer is complete.
- `glCheckFramebufferStatus` checks the framebuffer for completeness and returns the completeness status of the framebuffer.
- [Specification for glCheckFramebufferStatus]()

Since our framebuffer is not the default framebuffer, the rendering commands will have no impact on the visual output of your window.
- For this reason, it is is called **off-screen rendering** when rendering to a different framebuffer.
- In order to make all rendering options have a visual impact again on the main window, you must re-bind the default framebuffer (0).

Check if the framebuffer is complete. Then, re-bind the default framebuffer.
- Don't forget to delete the framebuffer object you generated after you are done using it.
- [Checking Framebuffer Completeness]()

The first requirement for framebuffer completeness states that we need to attach one of more attachments to the framebuffer.
- An **attachment** is a memoy location that can act as a buffer for the framebuffer (think of it as an image).
- When creating an attachment, we have two options:
  - Textures
  - Renderbuffer objects

Texture Attachments
When attaching a texture to a framebuffer, all rendering commands will write to the texture as if it was a normal color/depth/stencil buffer.
- The advantage of using textures is that the render output is stored inside the texture image, which we can easily use in our shaders.

Creating a texture for a framebuffer is the same as creating a normal texture, except you don't (typically) specify wrapping or mipmaps, and you use the window size as the image size parameters.

You can attach a 2D texture to a framebuffer using
`glFramebufferTexture2D(GLenum target, GLenum attachment, GLenum textarget, GLuint texture, GLint level)`.
- **`target`**: The framebuffer type we're targetting (draw, read, or both)
- **`attachment`**: The type of attachment we're going to attach. Right now, we're attaching a color attachment.
- **`textarget`**: The type of the texture you want to attach.
- **`texture`**: The texture to attach.
- **`level`**: The mipmap level.

Create an empty texture object with linear filtering and attach it to the framebuffer.
- [Attaching a Texture]()

Next to the color attachment, we can also attach a depth texture (using `GL_DEPTH_ATTACHMENT`), which would cause the texture's format and internalformat types to become `GL_DEPTH_COMPONENT`, and a stencil texture (using `GL_STENCIL_ATTACHMENT`), which would cause the formats to become `GL_STENCIL_INDEX`.

It is also possible to attach both a depth buffer and a stencil buffer as a single texture.
- Each 32-bit value of the texture then contains 24 bits of depth information and 8 bits of stencil information.
- To attach a depth and stencil buffer as one texture, we use the `GL_DEPTH_STENCIL_ATTACHMENT` type and configure the texture's formats to contain combined depth and stencil values.
  - [Example of Attaching a Depth-Stencil Texture]()

Renderbuffer Object Attachments
Just like a texture image, a **renderbuffer object** is an actual buffer (e.g. an array of bytes, or integers, or pixels, etc.). However, a renderbuffer object cannot be directly read from.
- Not being able to be directly read from gives it the added advantage that OpenGL can do a few memory optimizations that can give it a performance edge over textures for off-screen rendering to a framebuffer.

Renderbuffer objects store all the render data directly into their buffer without any conversions to texture-specific formats, making them faster as a writeable storage medium.
- You can't read from a renderbuffer object directly, but you can use `glReadPixels` (very slow) to return a specified area of pixels from the currently bound framebuffer. This, however, does not return data directly from the attachment itself.

Because their data is in a native format, they are quite fast when writing data or copying data to other buffers.
- Operations like switching buffers are, therefore, quite fast when using renderbuffer objects.
- The `glfwSwapBuffers` function we've been using at the end of each frame may be implemented with renderbuffer objects (we simply write to a renderbuffer image and swap to the other one at the end). Renderbuffer objects are perfect for these kinds of operations.

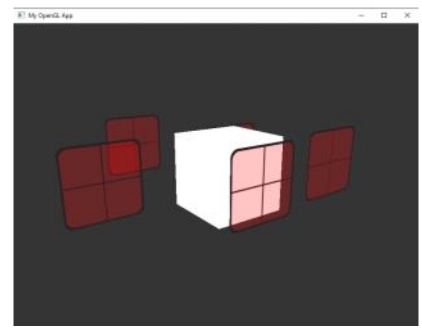
Since renderbuffer objects are write-only, they are often used as depth and stencil attachments, since, most of the time, we don't really need to read values from them, but we care about depth and stencil testing.
- We need the depth and stencil values for testing, but we don't need to sample these values. So, a renderbuffer object suits this perfectly.
- When we're not sampling from these buffers, a renderbuffer object is generally preferred.

Creating a depth and stencil renderbuffer object is done by calling the `glRenderbufferStorage` function. You can then attach the renderbuffer object using the `glFramebufferRenderbuffer` function.

Create a depth and stencil renderbuffer object and attach it to the framebuffer.
- Attaching a Renderbuffer Object

It is important to realize when to use renderbuffer objects and when to use textures.
- The general rule is that:
  - If you never need to sample data from a specific buffer, it is wise to use a renderbuffer object for that specific buffer.
  - However, if you need to sample data from a specific buffer (like color or depth values), you should use a texture attachment instead.

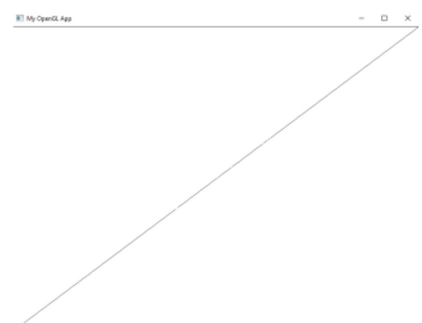
Rendering to a Texture



We're going to render the scene into a texture attached to a framebuffer object, and then draw this texture over a simple quad that spans the whole screen.

With a texture color buffer and a depth-stencil renderbuffer object attached, our framebuffer object is now complete.
- **NOTE:** The framebuffer object was complete as soon as we performed the texture color buffer attachment, but we wanted an attachment for handling depth and stencil testing, so we attached a depth-stencil renderbuffer object as well.
- Now, all we need to do to render to our framebuffer's buffers instead of the default framebuffer's is to bind the framebuffer object.
  - **NOTE:** If you were to omit a depth buffer attachment for your framebuffer, depth testing operations would no longer work.

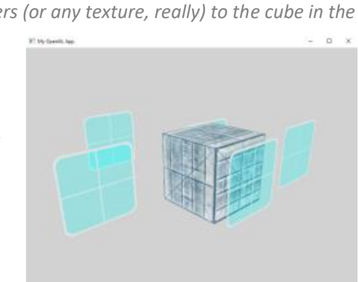
To draw the scene to a single texture:
1. Render the scene as usual with the new framebuffer bound as the active framebuffer.
2. Bind to the default framebuffer.
3. Draw a quad that spans the entire scene with the new framebuffer's color buffer as its texture.



Execute the steps above to draw the scene to a single texture that is drawn on a quad that covers to whole screen.
- You can test whether you did this successfully by changing the drawing mode to GL_LINE with `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`. If the whole screen is just two triangles, but looks normal when not drawn as a wireframe, then you've implemented this successfully.
- Rendering to a Texture

Post-Processing
Now that the entire scene is rendered to a single texture, we can create post-processing effects by manipulating the scene texture.
For the following sections, I recommend you apply the container texture from previous chapters (or any texture, really) to the cube in the scene so you can better see the effects of post-processing.

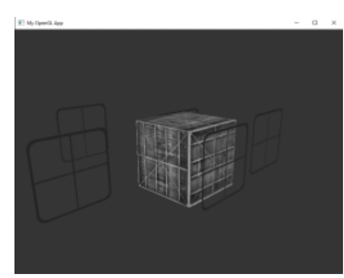
Inversion



We have access to each of the colors of the render output, so it's not difficult to return the inverse of these colors in the fragment shader. You can accomplish this by taking the color of the screen texture and inversing it by subtracting it from `1.0`.

Modify your screen shader so that the colors displayed are inverted.
- Inverted Colors

Grayscale



Another interesting effect is to remove all colors from the scene except white, gray, and black color, converting the displayed image to grayscale. This involves setting the RGB components of the fragment to the average of those components.
- **NOTE:** The human eye tends to be more sensitive to green colors, and less sensitive to blue. To get more realistic results, adjust the color values before calculating the average of the colors to reflect how the human eye receives color.

Modify your screen shader so that the colors displayed are in grayscale.
- Grayscale Colors

Kernel Effects
Another advantage of doing post-processing on a single texture image is that we can sample color values from other parts of the texture not specific to that fragment.
- Example: We could take a small area around the current texture coordinate and sample multiple texture values around the current texture value. We could then create interesting effects by combining them in different ways.

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

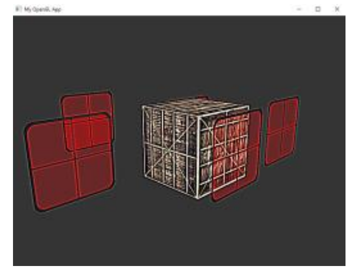
interesting effects by combining them in different ways.

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

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

The sample kernel in the image to the right takes 8 surrounding pixel values and multiplies them by 2 and the current pixel by -15 and sharpens the image.

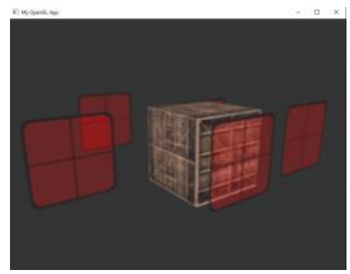
- This kernel multiplies the surrounding pixels by several weights determined in the kernel and balances the result by multiplying the current pixel by a large negative weight.
- **NOTE:** Most kernels you'll find over the internet all sum up to 1 if you add all the weights together. If they don't add up to 1, it means that the resulting texture color ends up brighter or darker than the original texture value.



Kernels are an extremely useful tool for post-processing since they're easy to experiment with and have a lot of examples available online.

Modify your screen shader to implement a sharpen kernel, which sharpens the image.

- $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ is an example of a sharpen kernel.
- [Sharpen Kernel]

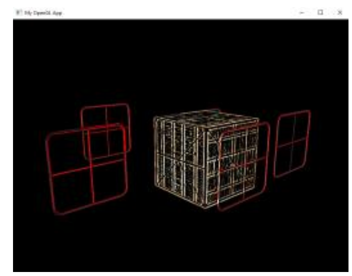


All you have to change is the kernel, and you can get very different post-processing effects. This makes using kernels very easy and powerful.

Blur

Modify your screen shader to implement a blur kernel, which blurs the image.

- $\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}/16$ is an example of a blur kernel.
- [Blur Kernel]



Edge Detection

Modify your screen shader to implement an edge-detection kernel, which highlights edges (color-wise) and darkens the rest.

- $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ is an example of an edge-detection kernel.
- [Edge-Detection Kernel]

EXERCISES

1. Can you use framebuffers to create a rear-view mirror? For this, you'll have to draw your scene twice: once with the camera rotated 180 degrees and once as normal. Try to create a small quad at the top of your screen to apply the mirror texture on.
2. Play around with the kernel values and create your own interesting post-processing effects. Try searching the internet as well for other interesting kernels.

Generating and Binding a Framebuffer

Friday, April 29, 2022 2:49 PM

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

Checking Framebuffer Completeness

Friday, April 29, 2022 4:18 PM

```
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) == GL_FRAMEBUFFER_COMPLETE) {
   // framebuffer is complete
}

glBindFramebuffer(GL_FRAMEBUFFER, 0);

...

glDeleteFramebuffers(1, &fbo);
```

Attaching a Texture

Friday, April 29, 2022 5:26 PM

```
// Generates and binds a texture object
unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);

// Image size set to size of screen. Data set to null since we're just allocating memory, not filling it.
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, WIDTH, 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);

// We're done configuring this texture object, so we can unbind it.
glBindTexture(GL_TEXTURE_2D, 0);

// Attaches the texture to the framebuffer as a color attachment
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texture, 0);
```

NOTE: The 0 at the end of `GL_COLOR_ATTACHMENT0` suggests we can attach more than 1 color attachment.

Example of Attaching a Depth-Stencil Texture

Friday, April 29, 2022 5:47 PM

```
// Specifies a texture composed of 24 bits of depth info and 8 bits of stencil info.
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH24_STENCIL8, 800, 600, 0, GL_DEPTH_STENCIL, GL_UNSIGNED_INT_24_8, NULL);

// Attaches the depth-stencil texture to the framebuffer
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_TEXTURE_2D, texture, 0);
```

Attaching a Renderbuffer Object

Friday, April 29, 2022 6:32 PM

```
// Generates and binds a renderbuffer object
unsigned int rbo;
glGenRenderbuffers(1, &rbo);
glBindRenderbuffer(GL_RENDERBUFFER, rbo);

// Makes the RBO a depth and stencil RBO, with 24 bits allocated for depth info, and 8 bits for stencil info
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, WIDTH, HEIGHT);

// We're done configuring this renderbuffer object, so we can unbind it.
glBindRenderbuffer(GL_RENDERBUFFER, 0);

// Attaches the RBO to the framebuffer as a depth and stencil attachment
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rbo);
```

Rendering to a Texture

Friday, April 29, 2022 8:28 PM

screen.vert

```
#version 330 core

// aPos is a vec2 because the vertex position will be in normalized device coordinates (screen space)
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec2 aTexCoords;

out vec2 texCoords;

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

screen.frag

```
#version 330 core

in vec2 texCoords;

out vec4 FragColor;

uniform sampler2D screenTexture;

void main() {
   FragColor = texture(screenTexture, texCoords);
}
```

main.cpp

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

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

main()

```
Shader screenShader{ "Shaders/screen.vert", "Shaders/screen.frag" };
...
// Screen quad VAO setup
unsigned int screenQuadVAO, screenQuadVBO;
glGenVertexArrays(1, &screenQuadVAO);
glGenBuffers(1, &screenQuadVBO);
glBindVertexArray(screenQuadVAO);
glBindBuffer(GL_ARRAY_BUFFER, screenQuadVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(screenQuadVertices), screenQuadVertices, GL_STATIC_DRAW);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)0);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)(sizeof(float) * 2));
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glBindVertexArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);

// FBO setup (in previous pages)
[...]
```

Render Loop

```
// RENDERING
// Bind our framebuffer object to be drawn to, with color and depth testing
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
glClearColor(0.2f, 0.2f, 0.2f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Enables depth testing, since it is disabled later in a previous loop
glEnable(GL_DEPTH_TEST);
// Draw Scene
[...]

// Bind default framebuffer with white background color, with just color
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

```
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

// Draws the color buffer texture to the screen, in front of everything else
screenShader.use();
glBindVertexArray(screenQuadVAO);
glDisable(GL_DEPTH_TEST);
glBindTexture(GL_TEXTURE_2D, textureColorBuffer);
glDrawArrays(GL_TRIANGLES, 0, 6);

glfwSwapBuffers(window);
glfwPollEvents();
```

NOTE: Since each framebuffer we're using has its own set of buffers, we want to clear each of those buffers with the appropriate bits set by calling glClear. Also, when drawing the quad, we're disabling depth testing since we want to make sure the quad always renders in front of everything else (we'll have to enable depth testing again when we draw the normal scene though).

Inverted Colors

Monday, May 2, 2022 1:26 PM

screen.frag

```
void main() {
   FragColor = vec4(vec3(1.0 - texture(screenTexture, texCoords)), 1.0);
}
```

We don't want to modify the `w` component when implementing this effect, so we cast `1.0 - texture(screenTexture, texCoords)` to a vec3, then cast it back to a vec4 with `1.0` as the `w` component.

Grayscale Colors

Monday, May 2, 2022 1:35 PM

<u>screen.frag</u>

```
void main() {
   FragColor = texture(screenTexture, texCoords);
   float average = (FragColor.r + FragColor.g + FragColor.b) / 3.0; // True grayscale
   FragColor = vec4(average, average, average, 1.0);
}
```

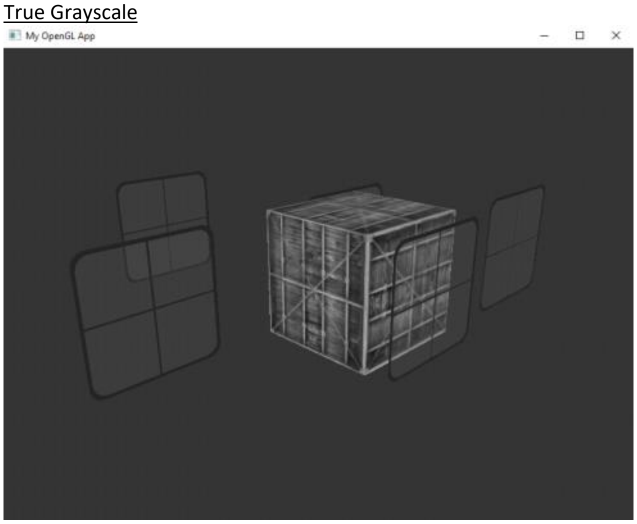
To get a more realistic-looking grayscale, use the color calculations below.

- **NOTE:** The difference between the two might not be immediately visible, but more complex scene will look more realistic with the grayscale calculation used below as opposed to true grayscale.

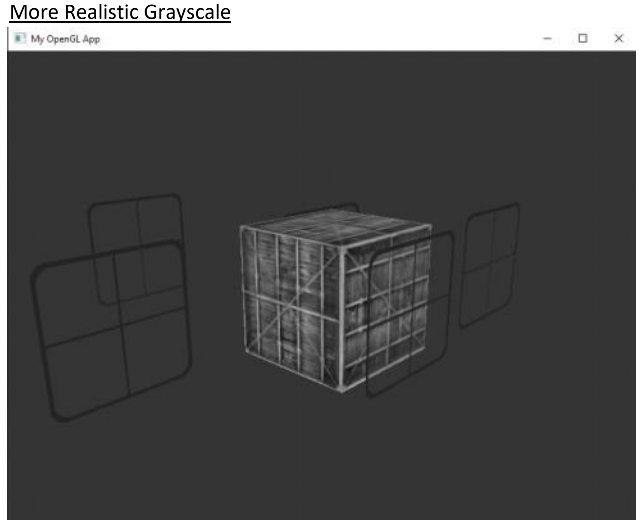
<u>screen.frag</u>

```
void main() {
   FragColor = texture(screenTexture, texCoords);
   float average = FragColor.r * 0.2126 + FragColor.g * 0.7152 + FragColor.b * 0.0722; // More realistic grayscale
   FragColor = vec4(average, average, average, 1.0);
}
```

<u>True Grayscale</u>



<u>More Realistic Grayscale</u>



Sharpen Kernel

Monday, May 2, 2022 3:53 PM

```
...

const float offset = 1.0 / 300.0;

void main() {
    // Offset values that are used in determining what adjacent fragments to sample
    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
    );

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

    // Stores the adjacent fragment colors for kernelling
    vec3 sampleTex[9];
    for (int i = 0; i < 9; ++i) {
        sampleTex[i] = vec3(texture(screenTexture, texCoords.st + offsets[i]));
    }

    // Applies the kernel to the adjacent fragments' colors to determine this fragment's color
    vec3 color = vec3(0.0);
    for (int i = 0; i < 9; ++i) {
        color += sampleTex[i] * kernel[i];
    }

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

You can modify the `offset` value to influence the accuracy of the sharpen.

Blur Kernel

Monday, May 2, 2022 4:11 PM

```
// Blur kernel
float kernel[9] = float[](
    1.0 / 16.0,  2.0 / 16.0, 1.0 / 16.0,
    2.0 / 16.0,  4.0 / 16.0, 2.0 / 16.0,
    1.0 / 16.0,  2.0 / 16.0, 1.0 / 16.0
);
```

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Because all values add up to 16, directly returning the combined sampled colors would result in an extremely bright color, so we have to divide each value of the kernel by 16.

Edge-Detection Kernel

Monday, May 2, 2022 4:25 PM

```
// Edge-detection kernel
float kernel[9] = float[](
    1.0,  1.0, 1.0,
    1.0, -8.0, 1.0,
    1.0,  1.0, 1.0
);
```