



COMP220: Graphics & Simulation
8: Post-Processing



Worksheet Schedule

| Worksheet | Start | Formative deadline |
|--------------------|--------------|-----------------------------------|
| 1: Framework | Week 2 | Mon 15th Feb 4pm (Week 4) |
| 2: Basic scene | Week 4 | Mon 1st Mar 4pm (Week 6) |
| 3: Plan/prototype | Week 6 | Mon 15th Mar 4pm (Week 8) |
| 4: Final iteration | Week 8 | Mon 12th Apr 4pm (Week 10) |

Learning outcomes

By the end of this week, you should be able to:

- ▶ **Explain** what the framebuffer is and how it can be used to generate 2D effects.
- ▶ **Implement** post-processing effects in your application.

Agenda

Agenda

- ▶ Lecture (async):
 - ▶ **Introduce** the framebuffer and its uses.
 - ▶ **Describe** a variety of 2D post-processing effects.

Agenda

- ▶ Lecture (async):
 - ▶ **Introduce** the framebuffer and its uses.
 - ▶ **Describe** a variety of 2D post-processing effects.
- ▶ Workshop (sync):
 - ▶ **Extend** the use of OpenGL textures to store rendered results.
 - ▶ **Apply** a selection of post-processing techniques to the rendered texture in a shader.

Schedule

| | |
|-------------|------------------------------------------|
| 16:00-16:10 | Arrival, sign-in & overview |
| 16:10-17:00 | Demo & Exercise: Rendering to Texture |
| 17:00-18:00 | Demo & Exercise: Post-processing effects |

Rendering To Texture



Brief Overview

Brief Overview

1. Create a texture of the required dimensions

Brief Overview

1. Create a texture of the required dimensions
2. Create Depth Buffer Object

Brief Overview

1. Create a texture of the required dimensions
2. Create Depth Buffer Object
3. Create a Framebuffer Object (FBO)

Brief Overview

1. Create a texture of the required dimensions
2. Create Depth Buffer Object
3. Create a Framebuffer Object (FBO)
4. Bind the texture and the Depth Buffer Object into the FBO

Brief Overview

1. Create a texture of the required dimensions
2. Create Depth Buffer Object
3. Create a Framebuffer Object (FBO)
4. Bind the texture and the Depth Buffer Object into the FBO
5. Bind the FBO to the pipeline

Brief Overview

1. Create a texture of the required dimensions
2. Create Depth Buffer Object
3. Create a Framebuffer Object (FBO)
4. Bind the texture and the Depth Buffer Object into the FBO
5. Bind the FBO to the pipeline
6. Render the scene to the new framebuffer

Creating a Texture

```
//The texture we are going to render to
GLuint renderTextureID;
 glGenTextures(1, &renderTextureID);

//Bind Texture
 glBindTexture(GL_TEXTURE_2D, renderTextureID);

//fill with empty data
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB,
              840, 680, 0, GL_RGB, GL_UNSIGNED_BYTE, 0);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                 GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                 GL_LINEAR);
```

Creating Depth Buffer Object

```
//The depth buffer
GLuint depthBufferID;
 glGenRenderbuffers(1,&depthBufferID);

//Bind the depth buffer
 glBindRenderbuffer(GL_RENDERBUFFER, depthBufferID);
 //Set the format of the depth buffer
 glBindRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
 840, 680);
```

Creating a Frame Buffer

```
//The framebuffer
GLuint frameBufferID;
 glGenFramebuffers(1, &frameBufferID);
```



Bind Texture and Depth Buffer

```
//Bind the framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, frameBufferID);

//Bind the texture as a colour attachment 0 to the
//active framebuffer
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                     renderTextureID, 0);

//Bind the depth buffer as a depth attachment
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
                          GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthBufferID);

if (glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
    GL_FRAMEBUFFER_COMPLETE)
{
    //error message!
}
```

Render to Framebuffer

```
//Bind the framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, frameBufferID);

//Drawn everything as normal!
```

Using Our Texture



Brief Overview

Brief Overview

- ▶ Now we have our scene stored on a texture

Brief Overview

- ▶ Now we have our scene stored on a texture
- ▶ We need to map this texture onto a surface

Brief Overview

- ▶ Now we have our scene stored on a texture
- ▶ We need to map this texture onto a surface
- ▶ This is usually a screen-aligned quad, but it can be any 3D object!

Brief Overview

- ▶ Now we have our scene stored on a texture
- ▶ We need to map this texture onto a surface
- ▶ This is usually a screen-aligned quad, but it can be any 3D object!
- ▶ In the fragment shader, we can do some processing...

Steps

Steps

1. Create a Vertex Buffer Object (VBO) for our quad

Steps

1. Create a Vertex Buffer Object (VBO) for our quad
2. Create a Vertex Array Object (VAO)

Steps

1. Create a Vertex Buffer Object (VBO) for our quad
2. Create a Vertex Array Object (VAO)
3. Load in a 'pass through' vertex shader and a fragment shader which takes in a texture

Steps

1. Create a Vertex Buffer Object (VBO) for our quad
2. Create a Vertex Array Object (VAO)
3. Load in a 'pass through' vertex shader and a fragment shader which takes in a texture
4. Render the quad and send across the texture that was bound to the framebuffer

Creating our Vertex Buffer Object

```
float quadVertices[] =
{
    -1, -1,
    1, -1,
    -1, 1,
    1, 1,
};

GLuint screenQuadVBOID;
glGenBuffers(1, &screenQuadVBOID);
 glBindBuffer(GL_ARRAY_BUFFER, screenQuadVBOID);
 glBufferData(GL_ARRAY_BUFFER, 8 * sizeof(float), quadVertices,
              GL_STATIC_DRAW);
```

Creating our Vertex Array

```
GLuint screenVAOID;  
glGenVertexArrays(1, &screenVAOID);  
 glBindVertexArray(screenVAOID);  
 glBindBuffer(GL_ARRAY_BUFFER, screenQuadVBOID);  
  
 glEnableVertexAttribArray(0);  
 glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, NULL);
```

Pass Through Vertex Shader

```
#version 330 core

layout(location=0) in vec2 vertexPosition;

out vec2 textureCoords;

void main()
{
    //Calculate Texture Coordinates for the Vertex
    textureCoords = (vertexPosition + 1.0) / 2.0;
    gl_Position = vec4(vertexPosition, 0.0, 1.0);
}
```

Example Fragment Shader

```
#version 330 core

out vec4 color;
in vec2 textureCoords;

uniform sampler2D texture0;

void main()
{
    //Read the texture and do some processing!
    color = texture(texture0, textureCoords);
}
```

Rendering

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glBindVertexArray(screenVAOID);
glBindTexture(GL_TEXTURE_2D, renderTextureID);
// etc.

//Bind our Postprocessing Program

//Send across any values to the shader

//Draw the quad!
```