OpenGL
IV -Advanced Shaders

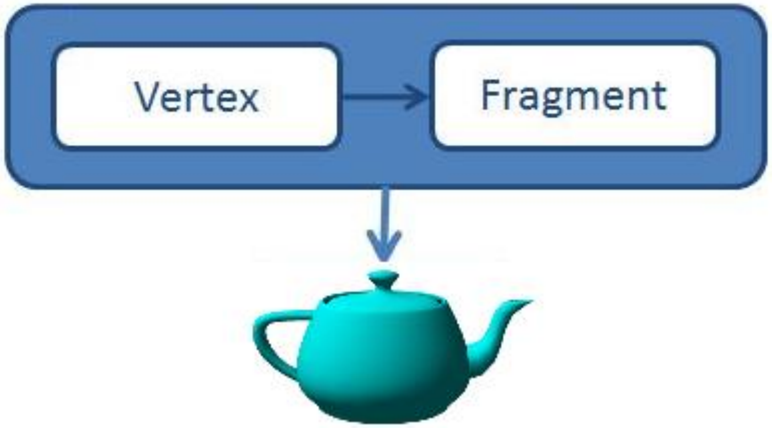
Stefan BORNHOFEN

Single pass rendering

By default, OpenGL renders to the screen, i.e. the default framebuffer that commonly contains a color and a depth buffer.

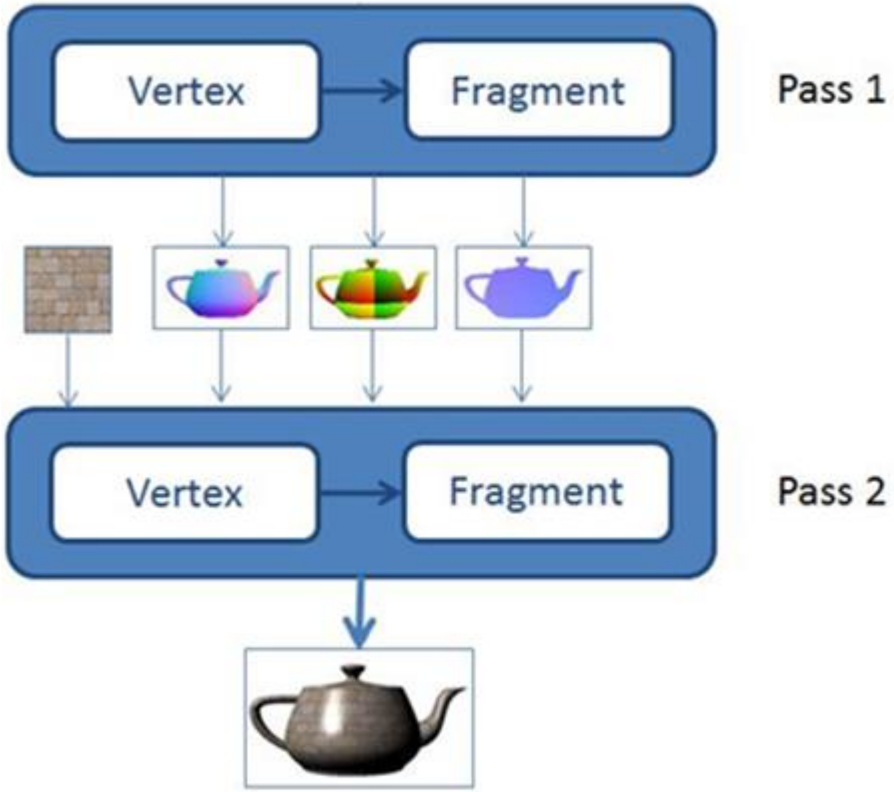
This is great when a pipeline consists of a single “pass” (= shader program).

The pipeline takes one triangle at a time, so only local information, and pre-computed maps, are available.



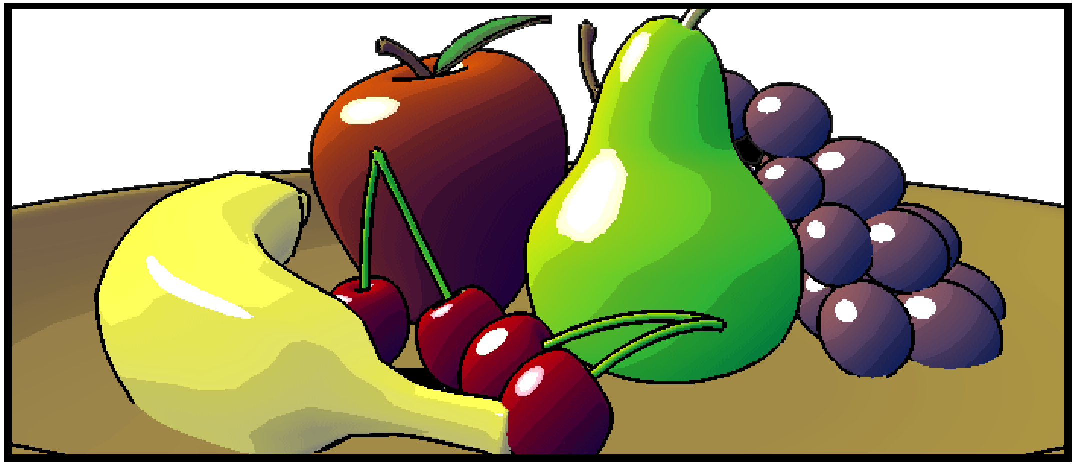
Multipass rendering

For more complex graphical effects, multiple passes are required, where the outputs of a pass are inputs of the following pass, typically in the form of textures.



Cel shading

- Cel shading or toon shading is a type of non-photorealistic rendering designed to mimic the style of a comic book or cartoon
- Cel shaders apply limited discrete shading values instead of a shade gradient, to create the characteristic flat look
- The black "ink" contour lines can be created with a variety of methods (we will use a simple one)
- Cel shading can be implemented by 2-pass rendering

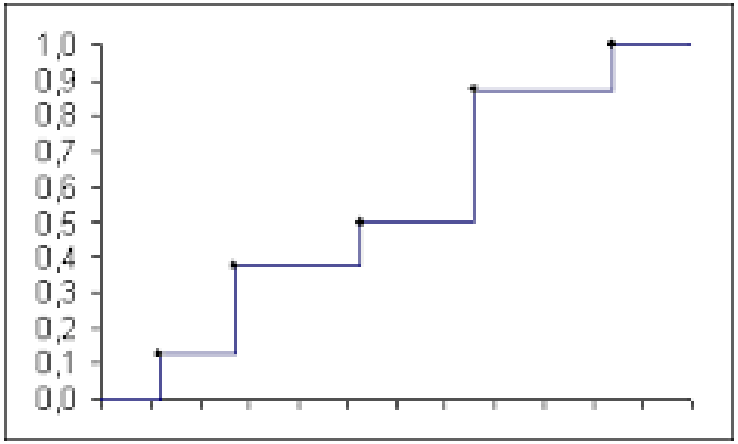


First pass: celOutline

- The vertex shader takes the vertices and moves them slightly outwards, i.e. in the direction of the normals.
- This way, the rendered model will be a little bit larger than the original.
- The fragment shader fills everything with black.

Second pass: celColor

- Take an illumination model
- Modify the fragment shader such that it breaks gradual light intensities into discrete values (apply a step function).



Exercise

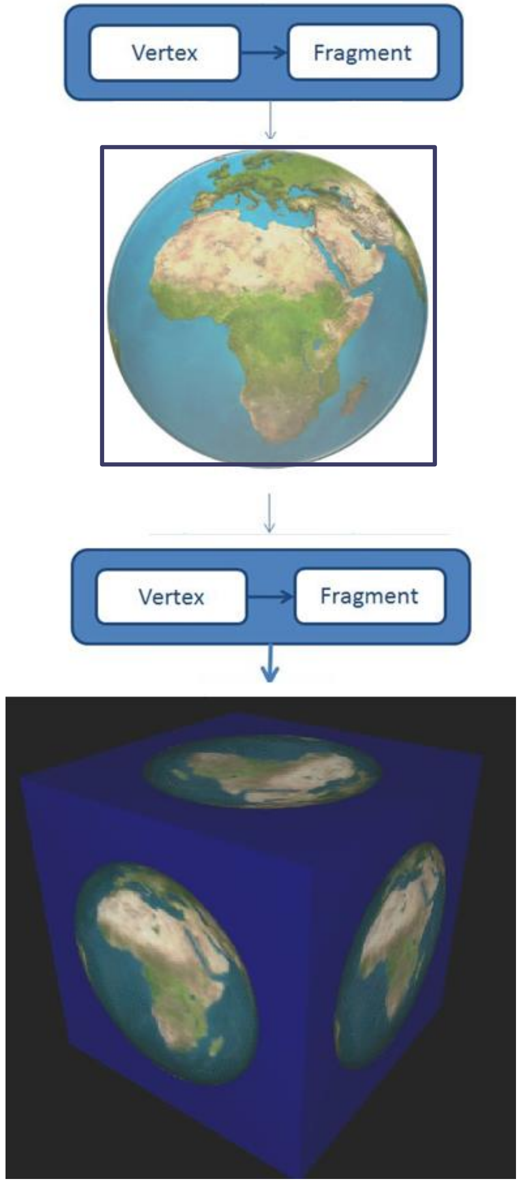
Write a cel shader by 2-pass rendering.

- Clear buffer (color + depth)
- Use program « celOutline »
- Render Suzanne
- Clear buffer (depth)
- Use program « celColor »
- Render Suzanne
- Swap buffer

Render-To-Texture

- Render-to-texture is a method to create a huge variety of effects.
- The idea is to render an image, but instead of writing to the screen buffer, we render to a texture. This texture can then be reused by another pass.

Let’s create a dynamic texture!



Define a Quad VAO

```
GLuint vaoquad, vboquad;

void initQuad()
{
  glGenVertexArrays(1, &vaoquad);
  glBindVertexArray(vaoquad);

  const GLfloat vertices[] =
      { -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, 1.0f, 1.0f, -1.0f };
  glGenBuffers(1, &vboquad);
  glBindBuffer(GL_ARRAY_BUFFER, vboquad);
  glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
  GL_STATIC_DRAW);

  glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, nullptr);
  glEnableVertexAttribArray(0);
}
```

Program « shaderTex »

Vertex shader

```
in vec2 in_position;
out vec2 UV;
void main() {
        // quad coordinates [-1,1] to tex coordinates [0,1]:
        UV = (1.f+in_position.xy)/2.f;
        // no MVP
        gl_Position = vec4(in_position.x,in_position.y,0.0f,1.0f);
}
```

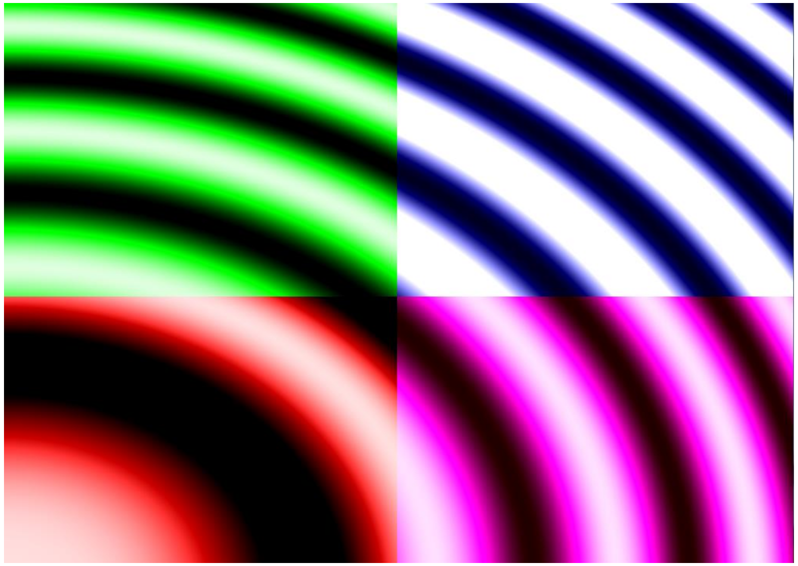
Fragment shader

```
in vec2 UV;
out vec4 fColor;
uniform float time; // current time in s or ms
void main() {
        // compute a nice color for this pixel
        // depending on UV.x, UV.y and time
        fColor = ...
}
```

Exercise

Texture to Screen

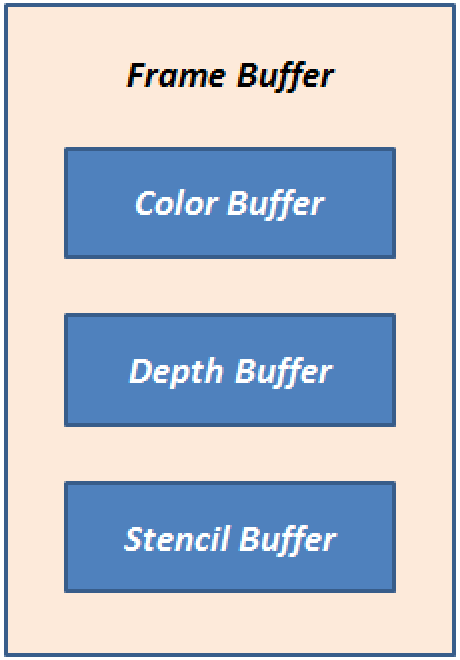
```
glViewport(0, 0, SCREEN_X, SCREEN_Y);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(shaderTex);
glUniform1f(glGetUniformLocation(shaderTex, "time"), whatTimeIsIt());
glBindVertexArray(vaoquad);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
glutSwapBuffers();
```



Now let’s apply this texture to Suzanne!

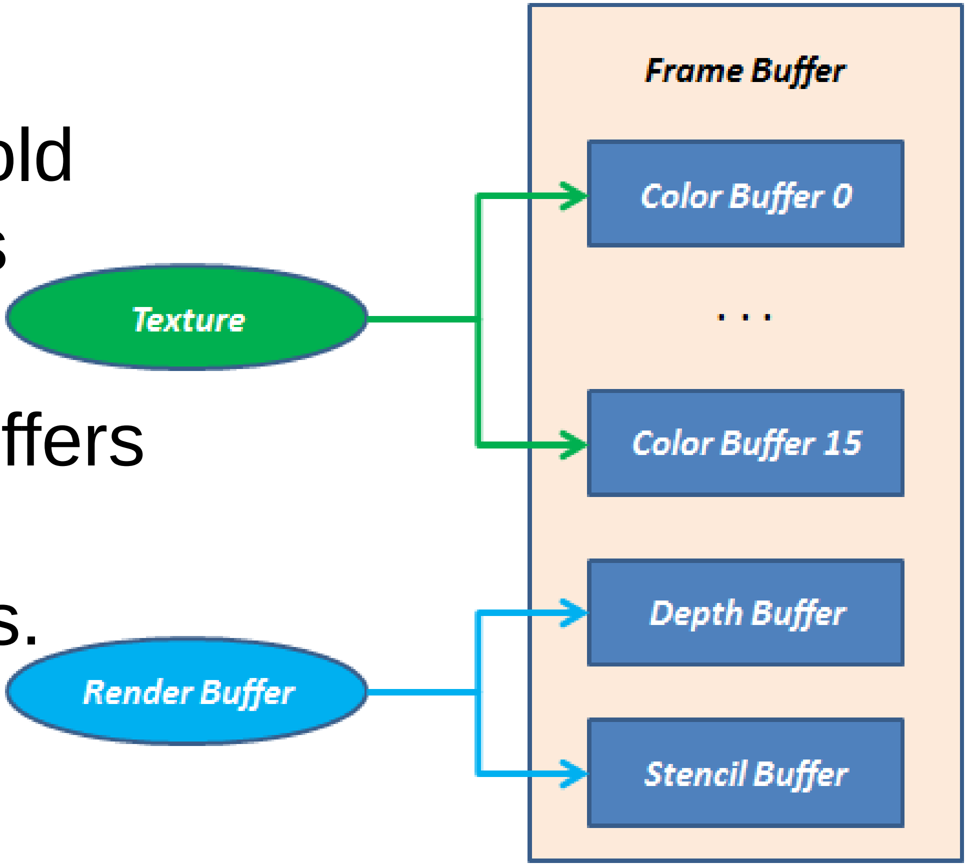
Frame Buffer

- Frame buffer objects (FBO) are the destination for rendering commands, like “hidden screens”
- You provide storage by attaching a set of ancillary buffers: Color - Depth - Stencil



Frame Buffer

- Color buffers are texture objects
- Frame buffers can hold up to 16 color buffers
- Depth and Stencil buffers are not textures but Render buffer objects.



Create a Framebuffer

```
GLuint framebuffer, renderedTexture, depthrenderbuffer;

void createFrameBuffer(int X, int Y) {
// creates a framebuffer of size (X,Y) with one color attachment (texture) and a depth buffer
        glGenFramebuffers(1, &framebuffer);
        glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
        glGenTextures(1, &renderedTexture);
        glBindTexture(GL_TEXTURE_2D, renderedTexture);
        // Give an empty image to OpenGL ( the last "0" )
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, X, Y, 0, GL_RGB, GL_UNSIGNED_BYTE, 0);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        // Set "renderedTexture" as our color attachment #0
        glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, renderedTexture, 0);
        GLenum DrawBuffers[1] = { GL_COLOR_ATTACHMENT0 };
        glDrawBuffers(1, DrawBuffers); // "1" is the size of DrawBuffers
        // The depth buffer
        glGenRenderbuffers(1, &depthrenderbuffer);
        glBindRenderbuffer(GL_RENDERBUFFER, depthrenderbuffer);
        glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, X, Y);
        glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthrenderbuffer);

        // check that framebuffer is ok
        if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
                fprintf(stderr, "Error creating Framebuffer\n"); exit(EXIT_FAILURE);
        }
}
```

Destroy a Framebuffer

Dont forget to clean up.

```
glDeleteTextures(1, &renderedTexture);
glDeleteRenderbuffers(1, &depthrenderbuffer);
glDeleteFramebuffers(1, &framebuffer);
```

Exercise

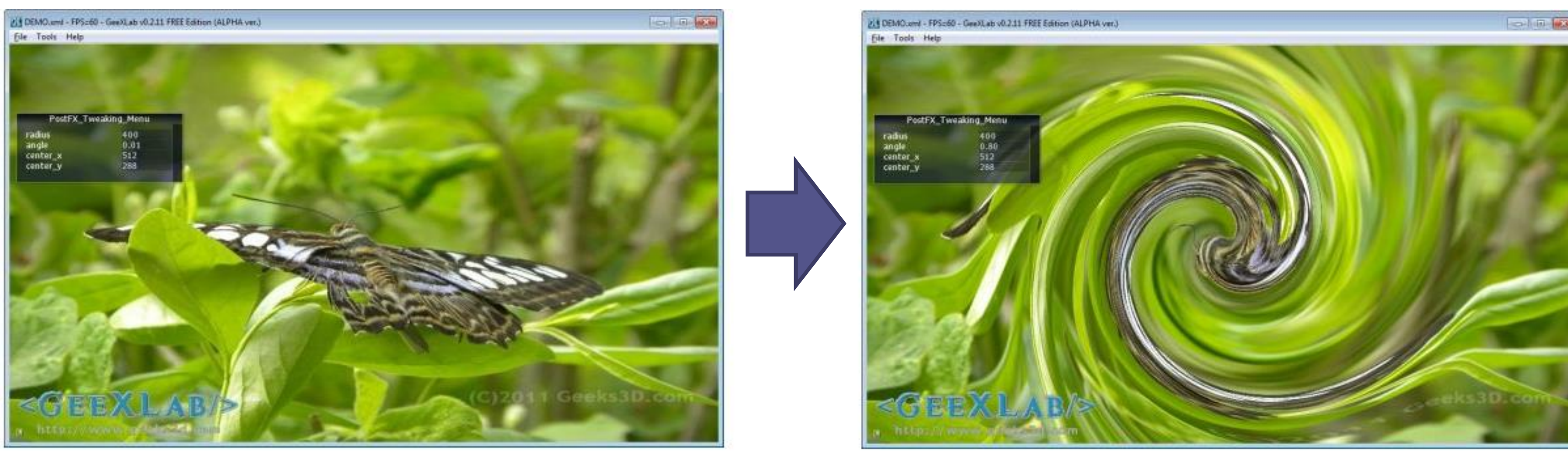
Texture to Suzanne

```
// render to texture
glViewport(0, 0, TEX_X, TEX_Y);
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(shaderTex);
glUniform1f(glGetUniformLocation(shaderTex, "time"), whatTimeIsIt());
glBindVertexArray(vaoquad);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

// use texture, render to screen
glViewport(0, 0, SCREEN_X, SCREEN_Y);
glBindFramebuffer(GL_FRAMEBUFFER, 0); // 0 = screen
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glBindTexture(GL_TEXTURE_2D, renderedTexture);
glBindVertexArray(vaoSuzanne);

Render textured Suzanne

glutSwapBuffers();
```

Post Processing

Post processing is a special case of render-to-texture.

- Create a framebuffer with a texture and a depth buffer
- Render your original image to the texture
- Write a post processing shader that takes the texture and renders the final image on a quad



```
// bind framebuffer to render the original image
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

render Suzanne

// the image is now in the frame buffer and
// can be used by another shader for post processing

// bind screen buffer
glBindFramebuffer(GL_FRAMEBUFFER, 0); // screen
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glBindTexture(GL_TEXTURE_2D, renderedTexture);
glUseProgram(postprocess);
glBindVertexArray(vaoquad);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

glutSwapBuffers();
```

Program « passthrough »

The passthrough post process does nothing but copy the texture to the screen.

Vertex shader

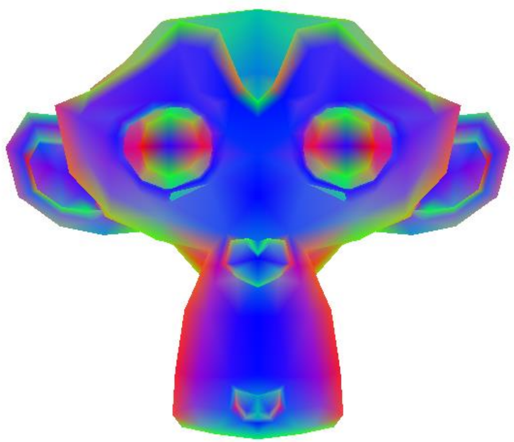
```
in vec4 in_position;
out vec2 UV;
void main() {
	// quad coordinates [-1,1] to tex coordinates [0,1]
	UV = (1.f+in_position.xy)/2.f;
	gl_Position = in_position;
}
```

Fragment shader

```
in vec2 UV;
uniform sampler2D renderedTexture;
out vec4 fColor;
void main() {
	fColor = vec4(texture(renderedTexture, UV).xyz, 1.0);
}
```

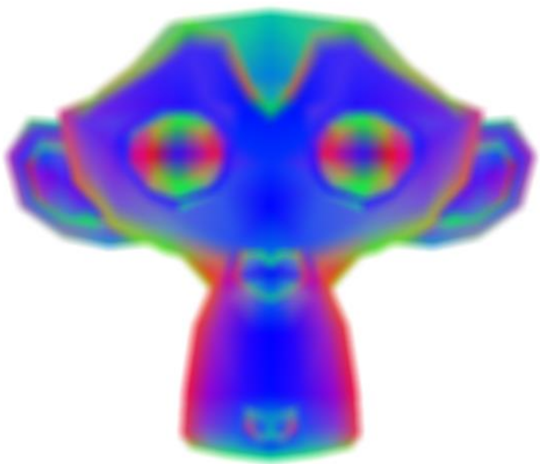
Exercise

Implement the following effects through post processing.



Passthrough
(no effect)

Grayscale



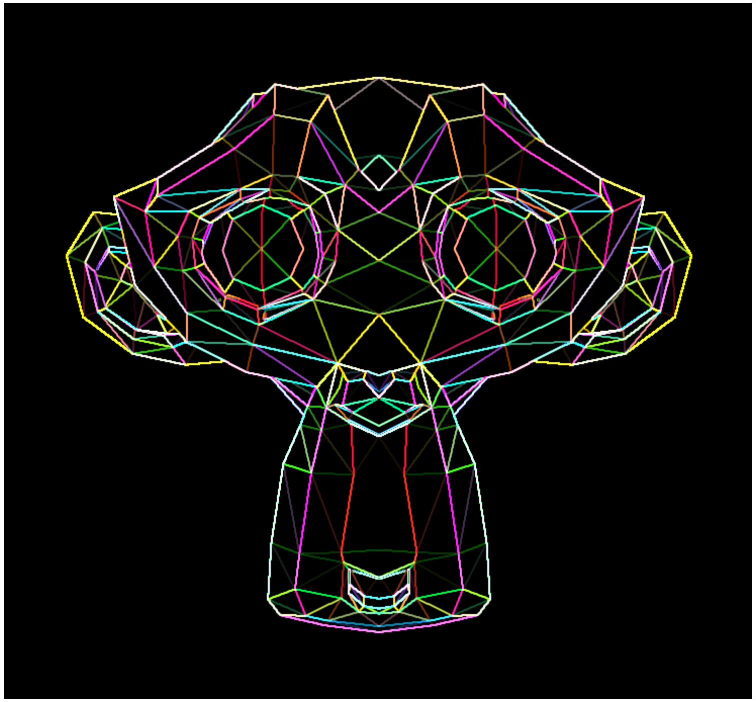
Blur

Pixelize

Exercise

Sobel: a really cool shader!

The Sobel operator is widely used in image processing and computer vision, particularly for edge detection.



<u>Fragment shader</u>

```
in vec2 UV;
out vec4 fColor; // final fragment color

uniform sampler2D renderedTexture;
uniform float SIZEX;
uniform float SIZEY;

void main() {
            vec4 top         = texture(renderedTexture, vec2(UV.x, UV.y + 1.0 / SIZEY));
            vec4 bottom      = texture(renderedTexture, vec2(UV.x, UV.y - 1.0 / SIZEY));
            vec4 left        = texture(renderedTexture, vec2(UV.x - 1.0 / SIZEX, UV.y));
            vec4 right       = texture(renderedTexture, vec2(UV.x + 1.0 / SIZEX, UV.y));
            vec4 topLeft     = texture(renderedTexture, vec2(UV.x - 1.0 / SIZEX, UV.y + 1.0 / SIZEY));
            vec4 topRight    = texture(renderedTexture, vec2(UV.x + 1.0 / SIZEX, UV.y + 1.0 / SIZEY));
            vec4 bottomLeft  = texture(renderedTexture, vec2(UV.x - 1.0 / SIZEX, UV.y - 1.0 / SIZEY));
            vec4 bottomRight = texture(renderedTexture, vec2(UV.x + 1.0 / SIZEX, UV.y - 1.0 / SIZEY));
            vec4 sx = -topLeft - 2 * left - bottomLeft + topRight   + 2 * right  + bottomRight;
            vec4 sy = -topLeft - 2 * top  - topRight   + bottomLeft + 2 * bottom + bottomRight;
            vec4 sobel = sqrt(sx * sx + sy * sy);
            fColor = sobel;
}
```