Introduction to Graphics Programming and its Applications

繪圖程式設計與應用

Fragment Processing and the Framebuffer

Instructor: Hung-Kuo Chu

Department of Computer Science

National Tsing Hua University

CS4505

Codeblock Conventions

Yellow Codeblock => Application Program (CPU)

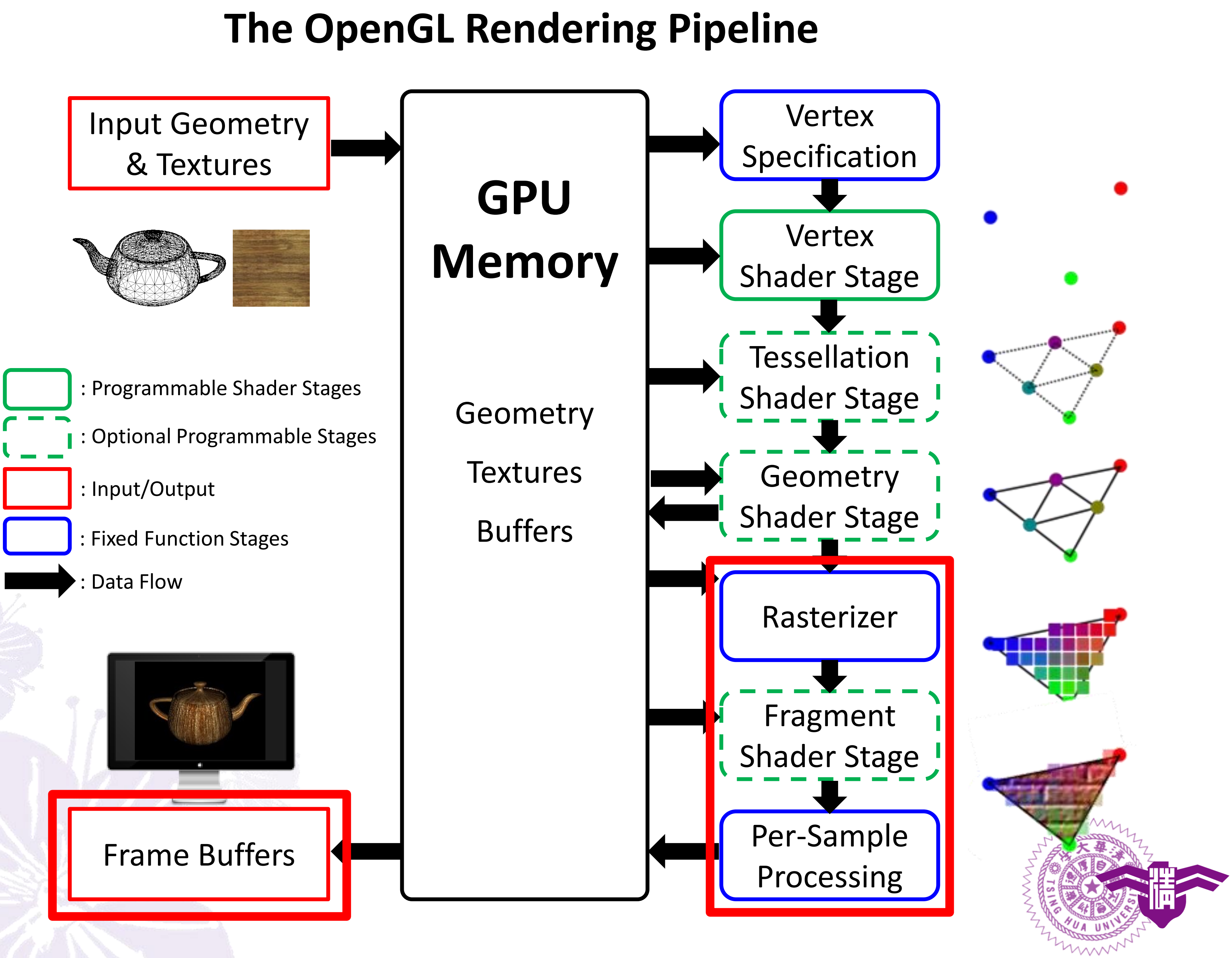
- Create OpenGL Context
- Create and Maintain OpenGL Objects
- Generate Works for the GPU to Consume

Blue Codeblock => Shader Program (GPU)

- Shader for a Certain Programmable Stage
- Process Geometry or Fragment in Parallel
- Starts with #version 410 core Declaration

What You'll Learn in This Lecture

- Operations performed before and after the fragment shader stage
- Details about the fragment shader stage, what its input/output is, and considerations when writing a fragment shader
- How to use framebuffer objects to create offscreen rendering setups

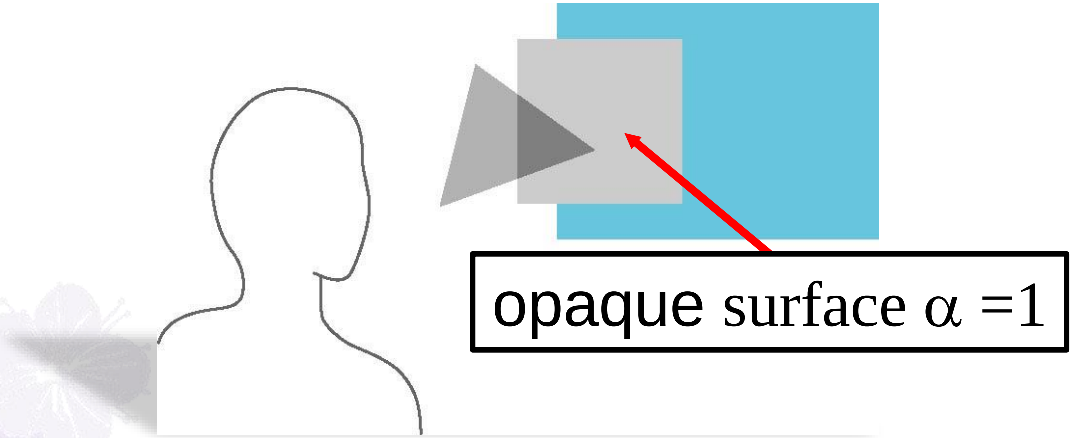


The OpenGL Rendering Pipeline
Input Geometry & Textures
GPU Memory
Geometry
Textures
Buffers
Vertex Specification
Vertex Shader Stage
Tessellation Shader Stage
Geometry Shader Stage
Rasterizer
Fragment Shader Stage
Per-Sample Processing
Frame Buffers
: Programmable Shader Stages
: Optional Programmable Stages
: Input/Output
: Fixed Function Stages
: Data Flow

Blending

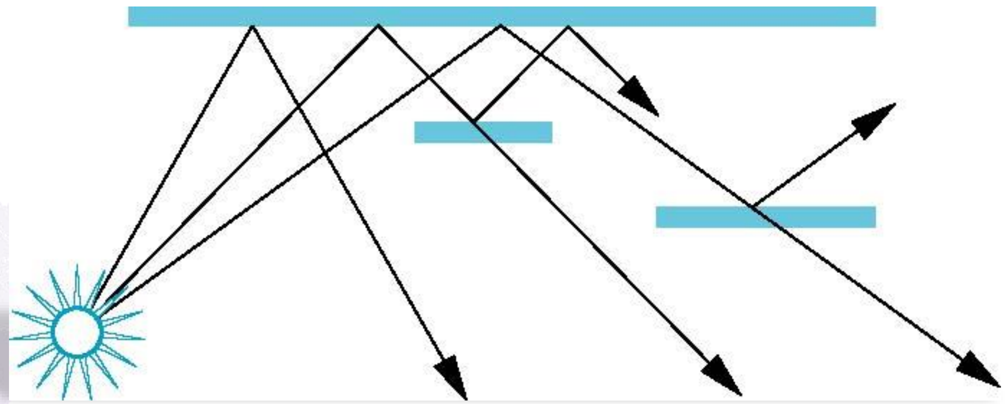
Opacity and Transparency

- Opaque surfaces permit no light to pass through
- Transparent surfaces permit all light to pass
- Translucent surfaces pass some light
  - Translucency = 1 – opacity (α)



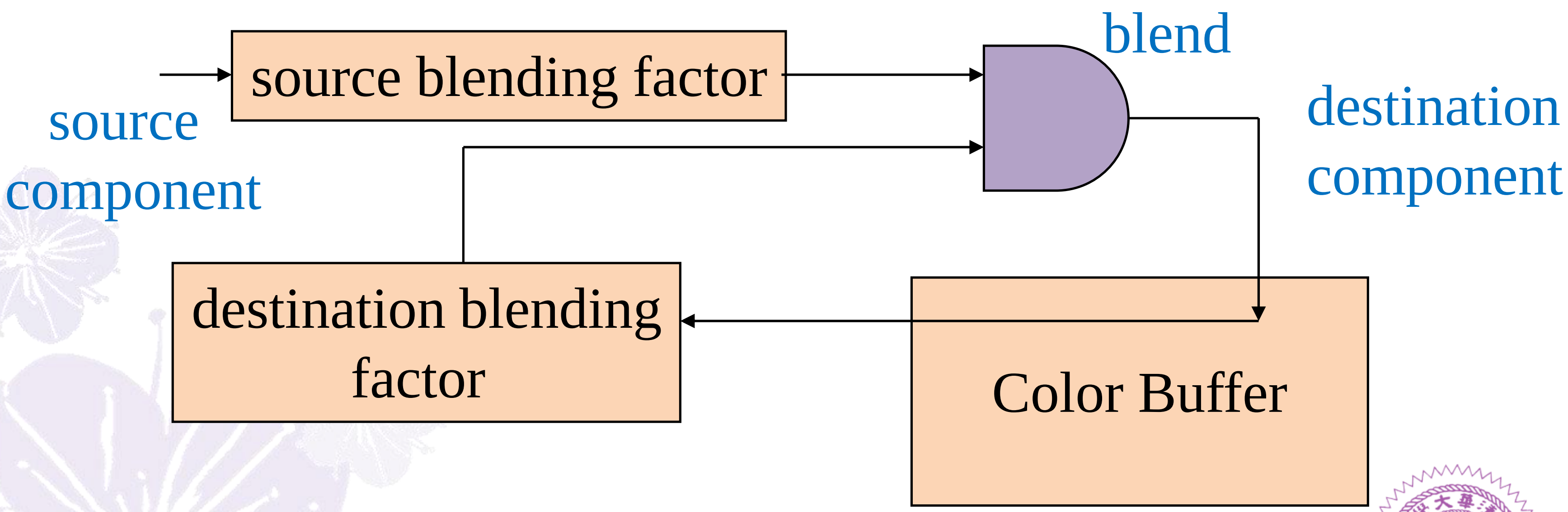
Physical Models

- Dealing with translucency in a physically correct manner is difficult due to
  - The complexity of the internal interactions of light and matter
  - Using a pipeline renderer



Writing Model

- Use A component of RGBA color to store opacity
- During rendering we can expand our writing model to use RGBA values



Blending Equation

- We can define source and destination blending factors for each RGBA component

$\mathbf{s} = [s_r, s_g, s_b, s_\alpha]$

$\mathbf{d} = [d_r, d_g, d_b, d_\alpha]$

Suppose that the source and destination colors are

$\mathbf{b} = [b_r, b_g, b_b, b_\alpha]$

$\mathbf{c} = [c_r, c_g, c_b, c_\alpha]$

Blend as

$\mathbf{c}' = [b_r\, s_r + c_r\, d_r, b_g\, s_g + c_g\, d_g\, , b_b\, s_b + c_b\, d_b\, , b_\alpha\, s_\alpha + c_\alpha\, d_\alpha\,]$

Blending and Compositing in OpenGL

- Must enable blending and pick source and destination factors

```
glEnable(GL_BLEND);

void glBlendFunc(src_factor, dst_factor);
```

- Only certain factors supported
  - `GL_ZERO, GL_ONE`
  - `GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA`
  - `GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA`

glBlendFunc

void glBlendFunc (GLenum sfactor, GLenum dfactor);

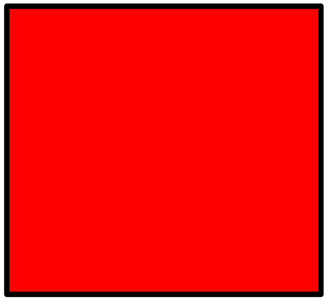
- **sfactor**: Specifies how the red, green, blue, and alpha source blending factors are computed.
- **dfactor**: Specifies how the red, green, blue, and alpha source blending factors are computed.
- **Description**: Specifies how the red, green, blue, and alpha destination blending factors are computed.

Blending Factor

Blending Factor Enum	Blending factor (R,G,B,A)
GL_ZERO	(0,0,0,0)
GL_ONE	(1,1,1,1)
GL_SRC_COLOR	(Rs, Gs, Bs, As)
GL_ONE_MINUS_SRC_COLOR	(1, 1, 1, 1) - (Rs, Gs, Bs, As)
GL_SRC_ALPHA	(As, As, As, As)
GL_ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) - (As, As, As, As)
GL_DST_COLOR	(Rd, Gd, Bd, Ad)
GL_ONE_MINUS_DST_COLOR	(1, 1, 1, 1) - (Rd, Gd, Bd, Ad)
GL_DST_ALPHA	(Ad, Ad, Ad, Ad)
GL_ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) - (Ad, Ad, Ad, Ad)

- Output = Src*sfactor + Destination*dfactor

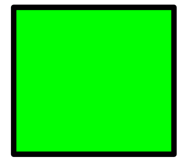
Example



Destination

(R,G,B,A)

(1.0,0.0,0.0,1.0)

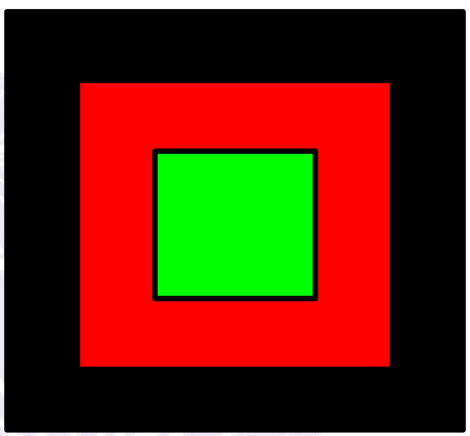


Source

(R,G,B,A)

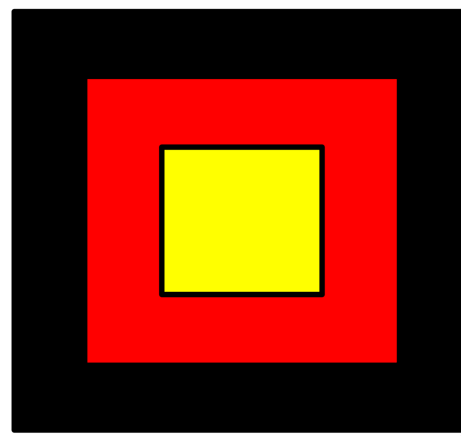
(0.0,1.0,0.0,1.0)

glDisable(GL_BLEND);



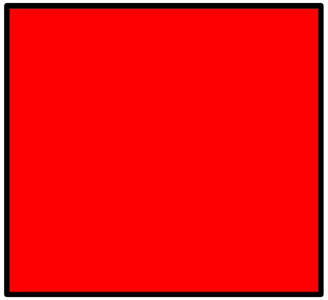
glEnable(GL_BLEND);

glBlendFunc(GL_ONE,GL_ONE);



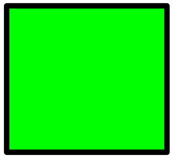
(R,G,B,A) (1.0,1.0,0.0,1.0)

Example



Destination

(R,G,B,A)
(1.0,0.0,0.0,1.0)

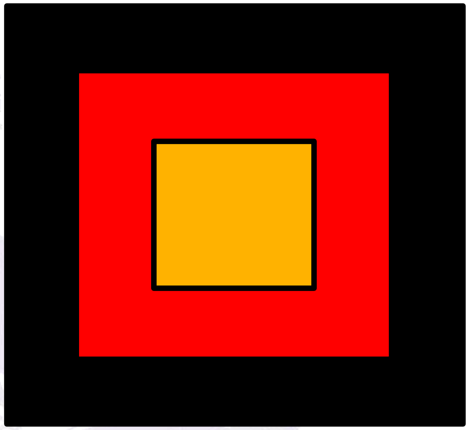


Source

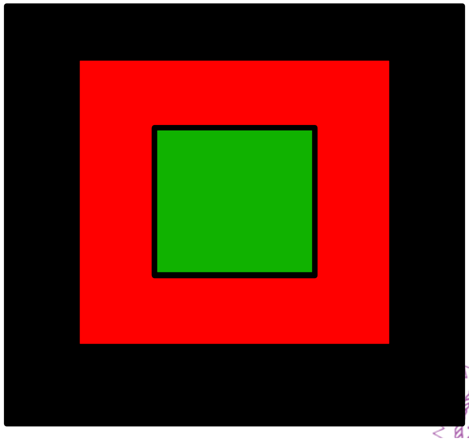
(R,G,B,A)
(0.0,1.0,0.0,0.7)

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

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

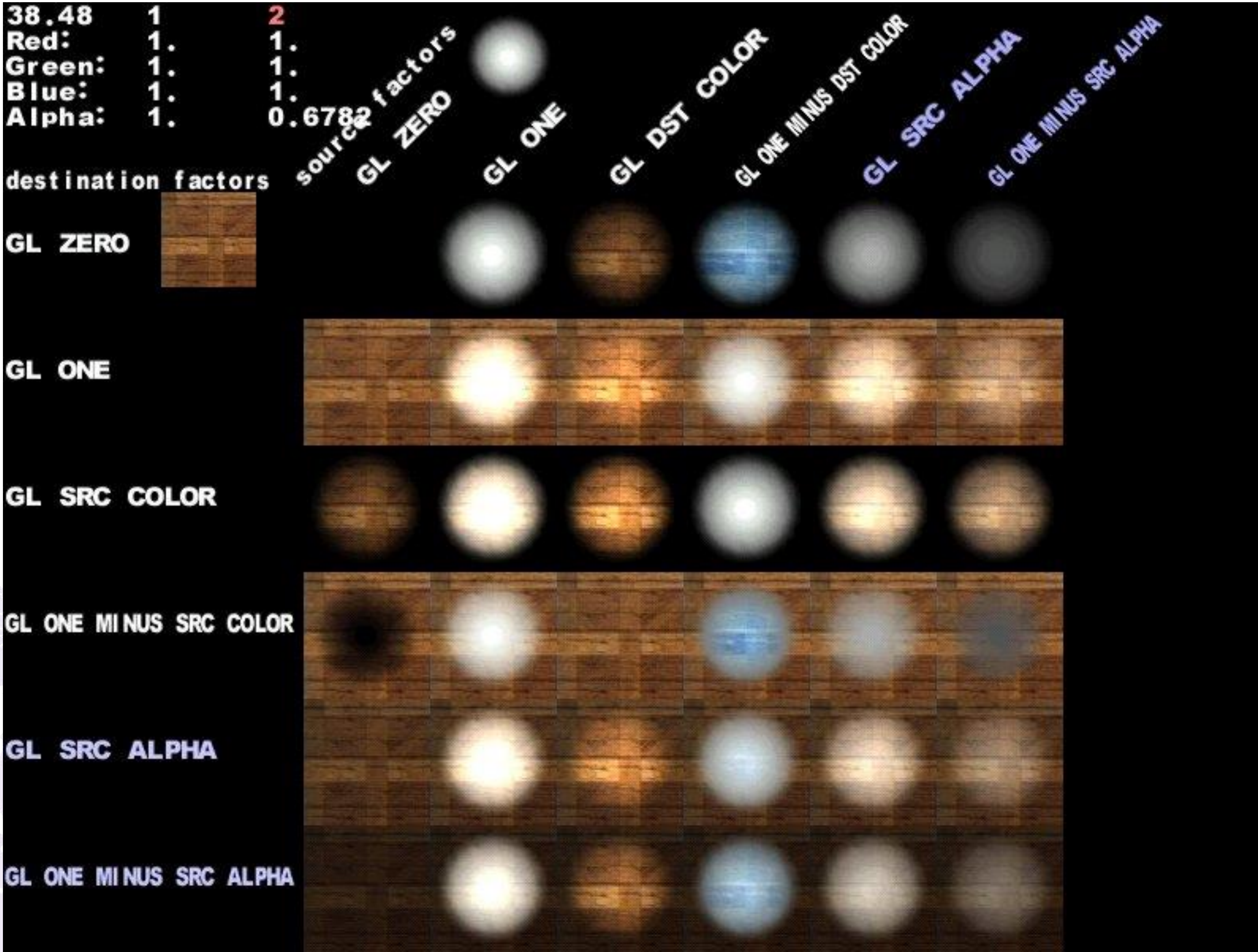


(R,G,B,A) (1.0,0.7,0.0,1.0)



(R,G,B,A) (0.3,0.7,0.0,1.0)

glBlendFunc Example

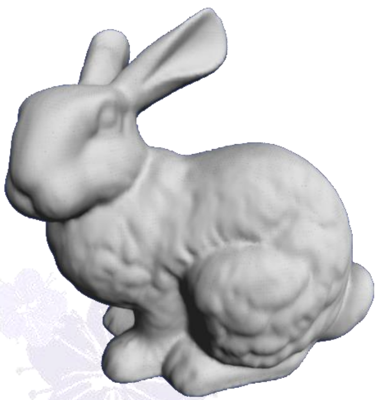


Off-Screen Rendering

Off-Screen Rendering

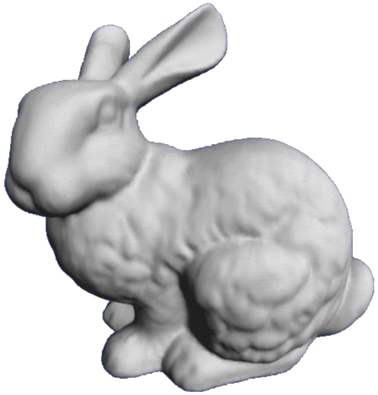
- OpenGL allows you to set up your own framebuffer and use it to draw directly into textures. You can then use these textures later for further rendering or processing

Rendering



Screen FrameBuffer

Off-Screen Rendering

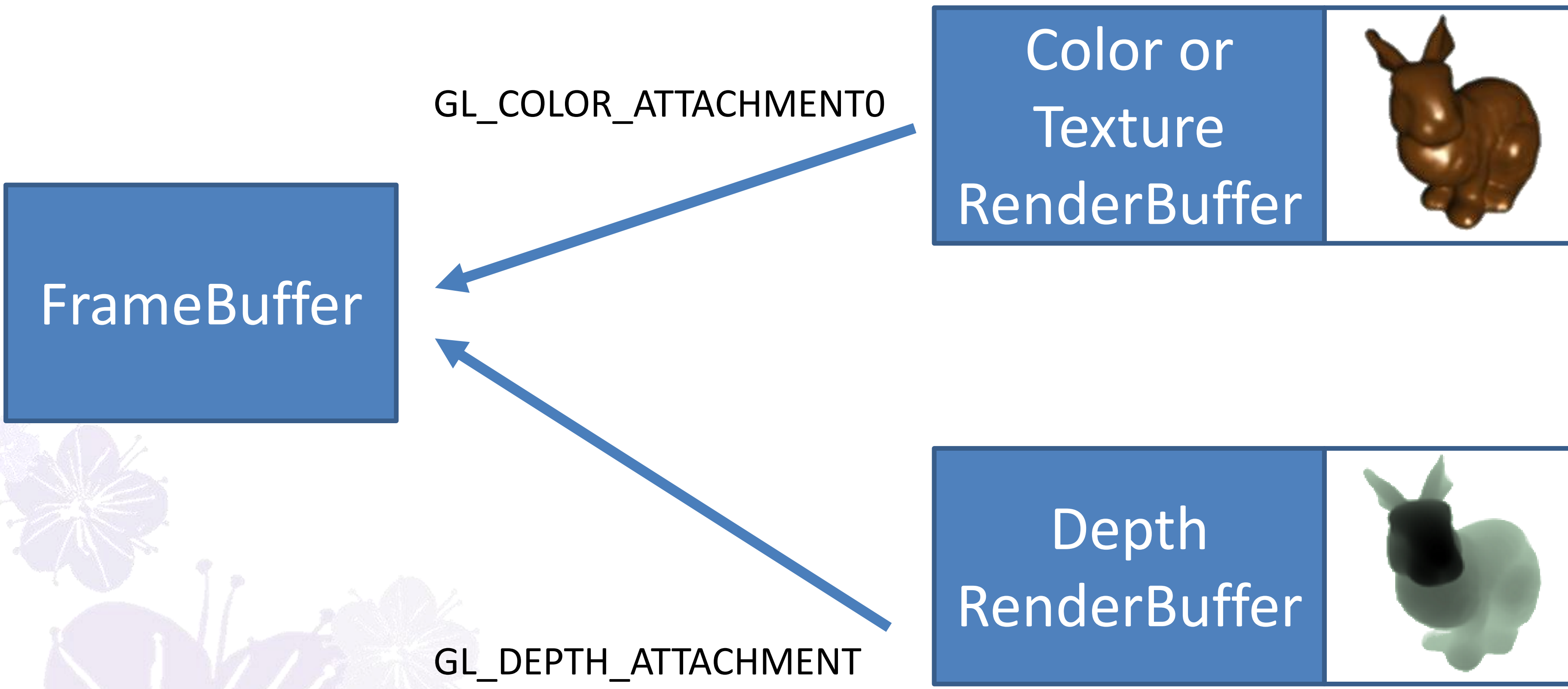


Custom FrameBuffer

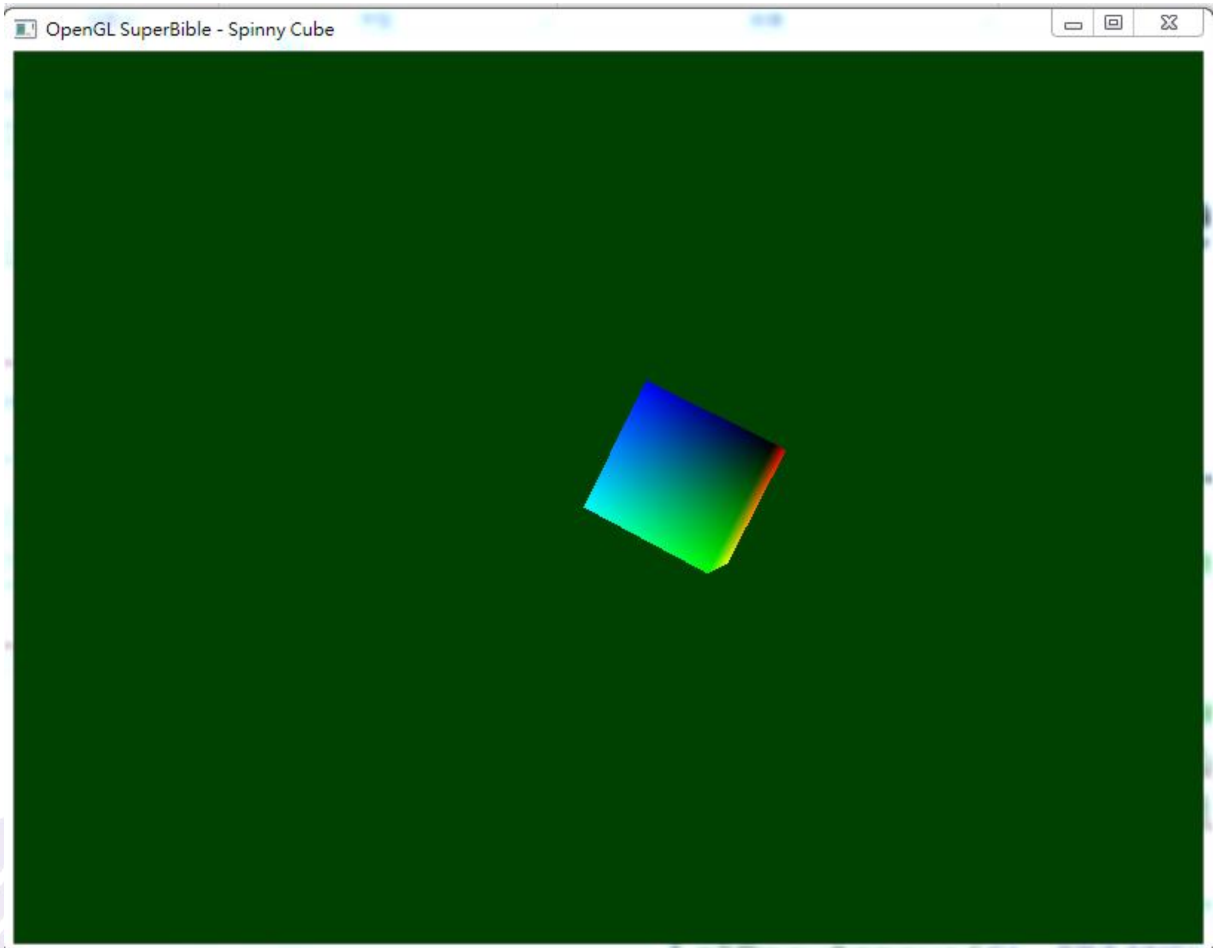
FrameBuffer Target Texture

Screen FrameBuffer

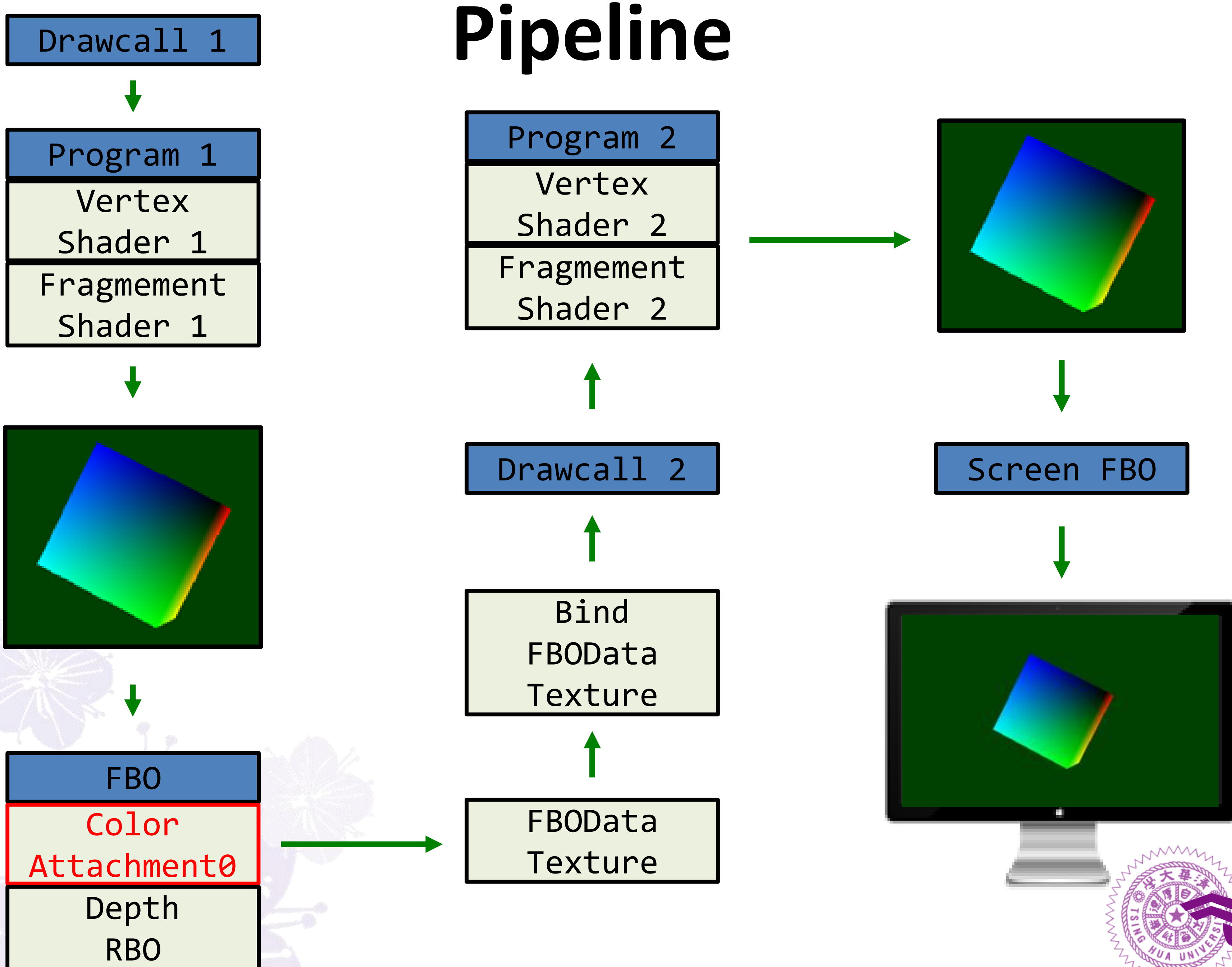
FrameBuffer Object



Spinning Cube ! – Off-Screen Rendering



Pipeline


Drawcall 1
Program 1
Vertex Shader 1
Fragment Shader 1
FBO
Color Attachment0
Depth RBO
FBOData Texture
Bind FBOData Texture
Drawcall 2
Program 2
Vertex Shader 2
Fragment Shader 2
Screen FBO

Framebuffer Object

- A framebuffer object (FBO) defines the **output imaging area of the pipeline**
- Consider a framebuffer object to be **a collection of images, called attachments**:
  - **Color Buffer Attachment**: the color (RGB) output
  - **Depth Buffer Attachment**: the depth output used to perform depth test
  - …
- A default framebuffer is created and bound when allocating the OpenGL window from the OS. Its ID will be #0, while custom FBOs start at ID#1.

Framebuffer Object

- Similar to the other OpenGL resource objects, to create and use a framebuffer object:
  1. Create FBOs with **glGenFramebuffers**
  2. Bind it with **glBindFramebuffer**
  3. Modify its **attachments** with **glFramebufferTexture2D**, etc…
  4. When ready to render, bind it to the pipeline output with **glBindFramebuffer** again
  5. Issue draw calls

Code: Vertex Shader (For Rendering Cube)

```
#version 410 core

in vec4 position;
out VS_OUT
{
   vec4 color;
} vs_out;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main(void)
{
    gl_Position = proj_matrix * mv_matrix * position;
    vs_out.color = position * 2.0 + vec4(0.5, 0.5, 0.5, 0.0);
}
```

Code: Fragment Shader (For Rendering Cube)

```
#version 410 core

in VS_OUT
{
    vec4 color;
} fs_in;

out vec4 color;

void main(void)
{
    color = fs_in.color;
}
```

Code: Vertex Shader2 (For Rendering FBO Texture)

```
#version 410 core

layout (location = 0) in vec2 position;
layout (location = 1) in vec2 texcoord;
out VS_OUT
{
    vec4 texcoord;
} vs_out;


void main(void)
{
    gl_Position = vec4(position, 1.0, 1.0);
    vs_out.texcoord = texcoord;
}
```

Code: Fragment Shader 2 (For Rendering FBO Texture)

```
#version 410 core

uniform sampler2D tex;
out vec4 color;

in VS_OUT
{
    vec2 texcoord;
} fs_in;

void main(void)
{
    color = texture(tex,fs_in.texcoord);
}
```

Code: Variables

```
// For Spinning Cube
GLuint          vao;
GLuint          program;
GLuint          buffer;
GLint           mv_location;
GLint           proj_location;
float           aspect;
glm::mat4       proj_matrix;

// For Frame Buffer Object(FBO)
GLuint          vao2;
GLuint          fbo;
GLuint          depthrbo;
GLuint          program2;
GLuint          window_vertex_buffer;
GLuint          fboDataTexture;
```

Code: Set Up Data

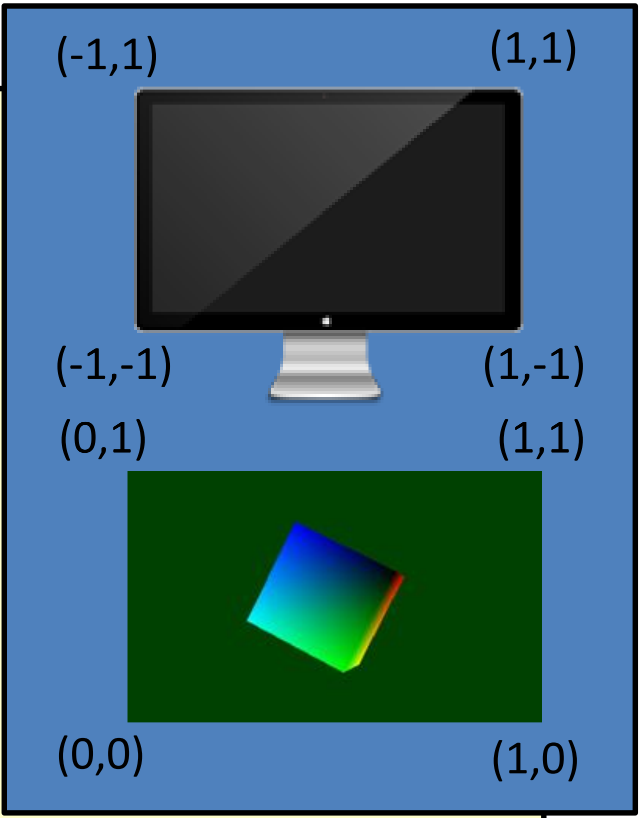
```
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

static const GLfloat vertex_positions[] =
{
  -0.25f, 0.25f, -0.25f, -0.25f, -0.25f, -0.25f,0.25f, -0.25f, -0.25f,
   0.25f, -0.25f, -0.25f,0.25f, 0.25f, -0.25f,-0.25f, 0.25f, -0.25f,
   /* MORE DATA HERE */
  -0.25f, 0.25f, -0.25f,0.25f, 0.25f, -0.25f,0.25f, 0.25f, 0.25f,
   0.25f, 0.25f, 0.25f,-0.25f, 0.25f, 0.25f,-0.25f, 0.25f, -0.25f
};
```

Code: Set Up Data

```
// Now generate some data and put it in a buffer object
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_positions),
             vertex_positions, GL_STATIC_DRAW);

// Set up our vertex attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(0);

mv_location = glGetUniformLocation(program, "mv_matrix");
proj_location = glGetUniformLocation(program, "proj_matrix");
```

Code: Set Up FBO Vertex Data

```
glGenVertexArrays(1, &vao2);
glBindVertexArray(vao2);

static const GLfloat window_vertex[] =
{
    //vec2 position vec2 texture_coord
    1.0f, -1.0f, 1.0f, 0.0f,
   -1.0f, -1.0f, 0.0f, 0.0f,
   -1.0f,  1.0f, 0.0f, 1.0f,
    1.0f,  1.0f, 1.0f, 1.0f
};

glGenBuffers(1, &window_vertex_buffer);
glBindBuffer(GL_ARRAY_BUFFER, window_vertex_buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(window_vertex),
             window_vertex, GL_STATIC_DRAW);
```



Code: Set Up FBO Vertex Data

```
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT)*4, 0);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(GL_FLOAT)*4,
                      (const GLvoid*)(sizeof(GL_FLOAT)*2));

glEnableVertexAttribArray( 0 );
glEnableVertexAttribArray( 1 );
```

Code: Set Up FBO

```
//Create FBO
glGenFramebuffers ( 1, &fbo );

//Create Depth RBO
glGenRenderbuffers( 1, &depthrbo );
glBindRenderbuffer( GL_RENDERBUFFER, depthrbo );
glRenderbufferStorage( GL_RENDERBUFFER,GL_DEPTH_COMPONENT32,
                        window_width, window_height);

//Create fboDataTexture
glGenTextures( 1, &fboDataTexture);
glBindTexture( GL_TEXTURE_2D, fboDataTexture);
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA,
        window_width, window_height, 0, GL_RGBA,GL_UNSIGNED_BYTE, NULL );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
```

Code: Set Up FBO (Cont'd)

```
glBindFramebuffer( GL_DRAW_FRAMEBUFFER, fbo);

//Set depthrbo to current fbo
glFramebufferRenderbuffer( GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                           GL_RENDERBUFFER, depthrbo );

//Set buffertexture to current fbo
glFramebufferTexture2D( GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                        GL_TEXTURE_2D, fboDataTexture, 0 );
```

Code: Reshape Function

```
glm::mat4 proj_matrix;

void OnReshape(int w, int h)
{
  aspect = (float)w / (float)f;

  //should use radius in glm!!
  proj_matrix = glm::perspective(deg2rad(50.0f),
                     aspect, 0.1f,1000.0f);
}
```

Code: Render Function

```
//Bind FBO
glBindFramebuffer( GL_DRAW_FRAMEBUFFER,fbo );

//which render buffer attachment is written
glDrawBuffer( GL_COLOR_ATTACHMENT0 );
glViewport( 0, 0, window_width, window_height);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Code: Render Function (Cont'd)

```
// Clear the framebuffer with dark green
static const GLfloat green[] = { 0.0f, 0.25f, 0.0f, 1.0f };
glClearBufferfv(GL_COLOR, 0, green);
// Render a spinning Cube
float f = (float)currentTime * 0.3f;
glm::mat4 Identy_Init(1.0);

glm::mat4 mv_matrix =
glm::translate(Identy_Init, glm:: vec3(0.0f, 0.0f, -4.0f));

mv_matrix = glm::translate(mv_matrix, glm::vec3(sinf(2.1f *f)*0.5f,
                                                 cosf(1.7f * f)*0.5f,
                                    sinf(1.3f * f)*cosf(1.5f*f)*2.0f));

mv_matrix = glm::rotate(mv_matrix, deg2rad(currentTime*45.0f),
                              glm::vec3(0.0f, 1.0f, 0.0f));

mv_matrix = glm::rotate(mv_matrix, deg2rad(currentTime*81.0f),
                              glm::vec3(1.0f, 0.0f, 0.0f));
```

Code: Render Function (Cont'd)

```
// Activate our program
glUseProgram(program);

// binding current vao
glBindVertexArray(vao);

// Set the model-view and projection matrices
glUniformMatrix4fv(mv_location, 1, GL_FALSE, mv_matrix);
glUniformMatrix4fv(proj_location, 1, GL_FALSE, proj_matrix);

// Draw 6 faces of 2 triangles of 3 vertices each = 36 vertices
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Code: Render Function (Cont'd)

```
// Now Return to the default framebuffer
glBindFramebuffer( GL_DRAW_FRAMEBUFFER, 0);

glViewport( 0, 0, window_width, window_height);

glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
glClearColor( 1.0f, 0.0f, 0.0f, 1.0f );

//Draw Final Rectangle Texture
glBindTexture( GL_TEXTURE_2D, fboDataTexture );

glBindVertexArray(vao2);
glUseProgram(program2);
glDrawArrays(GL_TRIANGLE_FAN,0,4 );
```

glGenFramebuffers

*void glGenFramebuffers (GLsizei n, GLuint *ids);*

- **n**: Specifies the number of framebuffer object names to generate.
- **ids:** Specifies an array in which the generated framebuffer object names are stored.
- **Description:** returns n framebuffer object names in ids.

glGenRenderbuffers

*void glGenRenderbuffers (GLsizei n, GLuint *renderbuffers);*

- **n**: Specifies the number of renderbuffer object names to generate.
- **renderbuffers:** Specifies an array in which the generated renderbuffer object names are stored.
- **Description:** returns n renderbuffer object names in renderbuffers.

glBindRenderbuffer

void glBindRenderbuffer (GLenum target, GLuint renderbuffer);

- **target:** Specifies the renderbuffer target of the binding operation. target must be GL_RENDERBUFFER.
- **renderbuffer:** Specifies the name of the renderbuffer object to bind.
- **Description:** glBindRenderbuffer binds the renderbuffer object with name renderbuffer to the renderbuffer target specified by target.

glRenderbufferStorage

void glRenderbufferStorage (GLenum target, GLenum internalformat, GLsizei width, GLsizei height);

- **target:** Specifies the renderbuffer target of the binding operation. target must be GL_RENDERBUFFER.
- **internalformat:** Specifies the internal format to use for the renderbuffer object's image.
- **Description:** glRenderbufferStorage establish the data storage, format of a renderbuffer object's image.

glBindFramebuffer

void glBindFramebuffer (GLenum target, GLuint framebuffer);

- **target:** Specifies the framebuffer target of the binding operation. target must be GL_DRAW_FRAMEBUFFER. or GL_READ_FRAMEBUFFER.
- **framebuffer:** Specifies the name of the framebuffer object to bind.
- **Description:** glBindRenderbuffer binds the framebuffer object with name framebuffer of both the read and draw framebuffer targets.

glFramebufferRenderbuffer

```
void glFramebufferRenderbuffer
        ( GLenum target, GLenum attachment,
        GLenum renderbuffertarget, GLuint renderbuffer);
```

- **target:** Specifies the target to which the framebuffer is bound for.
- **attachment:** Specifies the attachment point of the framebuffer.
- **renderbuffertarget:** Specifies the renderbuffer target. Must be GL_RENDERBUFFER.

glFramebufferRenderbuffer (Cont'd)

```
void glFramebufferRenderbuffer
        ( GLenum target, GLenum attachment,
        GLenum renderbuffertarget, GLuint renderbuffer);
```

- **renderbuffer:** Specifies the name of an existing renderbuffer object to attach.
- **Description:** Attaches a renderbuffer as one of the logical buffers of the specified framebuffer object. Renderbuffers **cannot be attached to the default draw and read framebuffer**, so they are not valid targets of these commands.

glFramebufferTexture2D

```
void glFramebufferTexture2D(GLenum target,
                            GLenum attachment, GLenum textarget,
                            GLuint texture, GLint level);
```

- **target:** Specifies the framebuffer target. The symbolic constant must be GL_FRAMEBUFFER.
- **attachment:** Specifies the attachment point to which an image from texture be attached.
- **textarget:** Specifies the texture target.

glFramebufferTexture2D (Cont'd)

```
void glFramebufferTexture2D(GLenum target,
                            GLenum attachment, GLenum textarget,
                            GLuint texture, GLint level);
```

- **texture:** Specifies the texture object whose image is to be attached.
- **level:** Specifies the mipmap level of the texture image to be attached, which must be 0.
- **Description:** glFramebufferTexture2D attaches the texture image specified by texture and level as one of the logical buffers of the currently bound framebuffer object.

Frame Buffer Attachment

Attachment	Target (Attach Point)
GL_COLOR_ATTACHMENTi	Color Image
GL_DEPTH_ATTACHMENT	Depth Buffer
GL_STENCIL_ATTACHMENT	Stencil Buffer
GL_DEPTH_STENCIL_ATTACHMENT	Depth and Stencil Buffer

Multiple Framebuffer Attachments

- Another extremely useful feature of user-defined framebuffers is that they support multiple attachments. That is, you can attach multiple textures to a single framebuffer and render into them simultaneously with a single fragment shader.
- We called **glFramebufferTexture()** and passed GL_COLOR_ATTACHMENT0 as the attachment parameter, but we mentioned that you can also pass GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMENT2, and so on.

Code: Set Up FBO

```
static const GLenum draw_buffers[] =
{
    GL_COLOR_ATTACHMENT0,
    GL_COLOR_ATTACHMENT1,
    GL_COLOR_ATTACHMENT2
};

// First, generate and bind our framebuffer object
glGenFramebuffers(1, &fbo);
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
// Generate three texture names
glGenTextures(3, &color_texture[0]);
```

Code: Set Up FBO

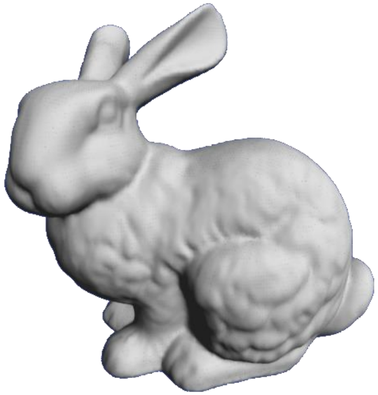
```
// For each one...
for (int i = 0; i < 3; i++)
{
    glBindTexture(GL_TEXTURE_2D, color_texture[i]);
    glTexStorage2D(GL_TEXTURE_2D, 9, GL_RGBA8, window_width, window_height);
    // Set its default filter parameters
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    // Attach it to our framebuffer object as color attachments
    glFramebufferTexture(GL_FRAMEBUFFER,draw_buffers[i], color_texture[i], 0);
}

// Set the draw buffers for the FBO to point to the color attachments
glDrawBuffers(3, draw_buffers);
```

Code: Fragment Shader 2 (For Rendering FBO Texture)

```
#version 410 core

uniform sampler2D tex;
layout (location = 0) out vec4 color_tex; // GL_COLOR_ATTACHMENT0
layout (location = 1) out vec4 color_normal; // GL_COLOR_ATTACHMENT1
layout (location = 2) out vec4 color_depth; // GL_COLOR_ATTACHMENT2

in VS_OUT
{
    vec2 texcoord;
    vec3 normal;
} fs_in;

void main(void)
{
    color_tex = texture(tex, fs_in.texcoord).rgba;
    color_normal = vec4(fs_in.normal, 1.0);
    color_depth = vec4(vec3(gl_FragCoord.z), 1.0);
}
```

Post Image Processing using FBO

Off-Screen Rendering



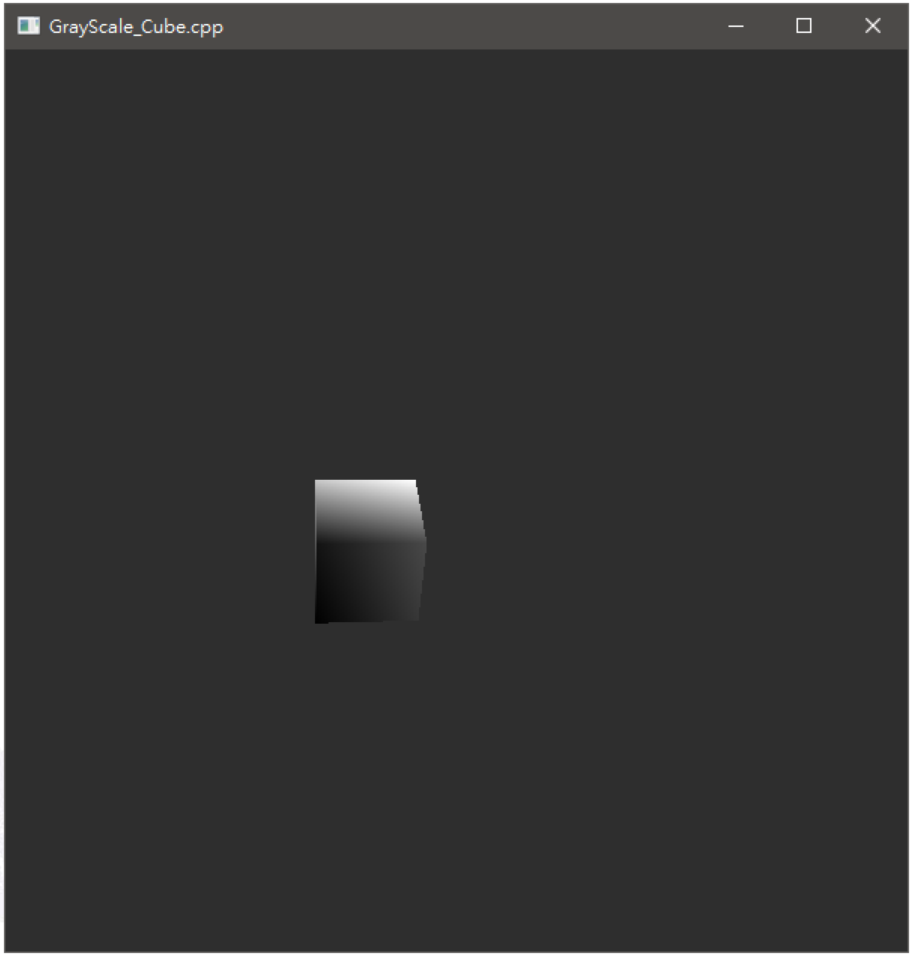
Own FrameBuffer

Image Processing

FrameBuffer Target Texture

OpenGL FrameBuffer

Grayscale



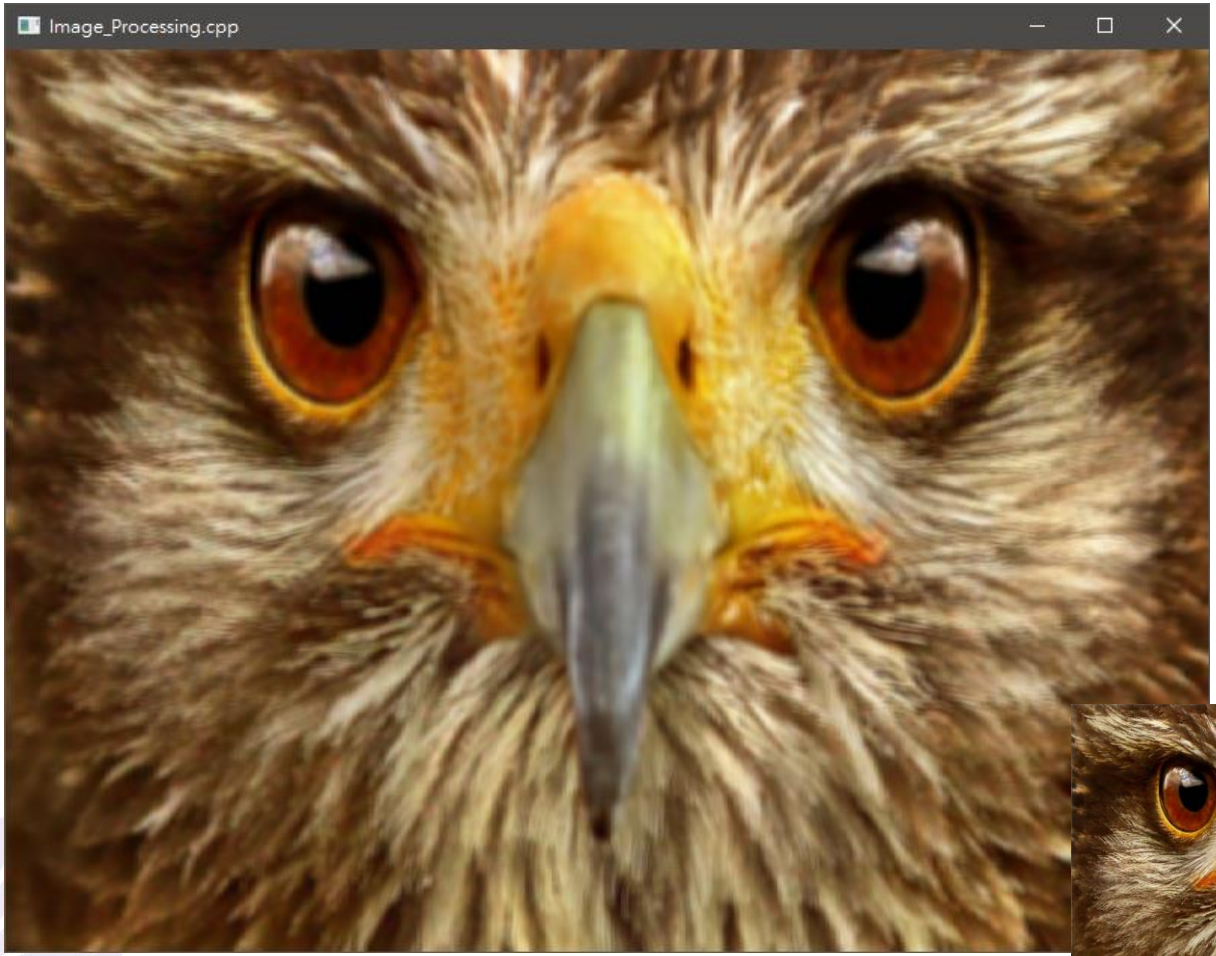
Code: Fragment Shader

```
#version 410 core

uniform sampler2D tex;
out vec4 color;

in VS_OUT
{
    vec2 texcoord;
} fs_in;

void main(void)
{
    vec3 tex_color = texture(tex, fs_in.texcoord).rgb;
    const vec3 grayscale = vec3(0.212, 0.715, 0.072);
    float Y = dot(tex_color, grayscale);
    color = vec4(vec3(Y), 1.0);
}
```

Code: Sequential (For Comparison)

```
for(int y = 0; y < img.height; y++)
{
    for(int x = 0; x < img.width; x++)
    {
        // process the pixel at (x, y)
        float R = img.at(x, y, 0);
        float G = img.at(x, y, 1);
        float B = img.at(x, y, 2);
        float Y = 0.2126 * R + 0.7152 * G + 0.722 * B;

        img.at(x, y, 0) = Y;
        img.at(x, y, 1) = Y;
        img.at(x, y, 2) = Y;
    }
}
```

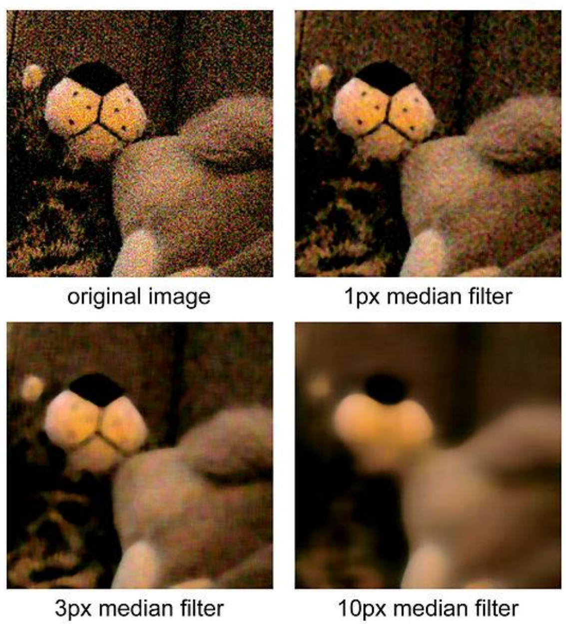
Median Blur



Original Image

Median Blur

- Also called “Box Filter”
- The larger the filter range, the blurrier the result will be



original image | 1px median filter | 3px median filter | 10px median filter

V_1	V_2	V_3
V_4	V_5	V_6
V_7	V_8	V_9

$$V_5 = \frac{1}{9} * \sum_{i=1}^{9} V_i$$

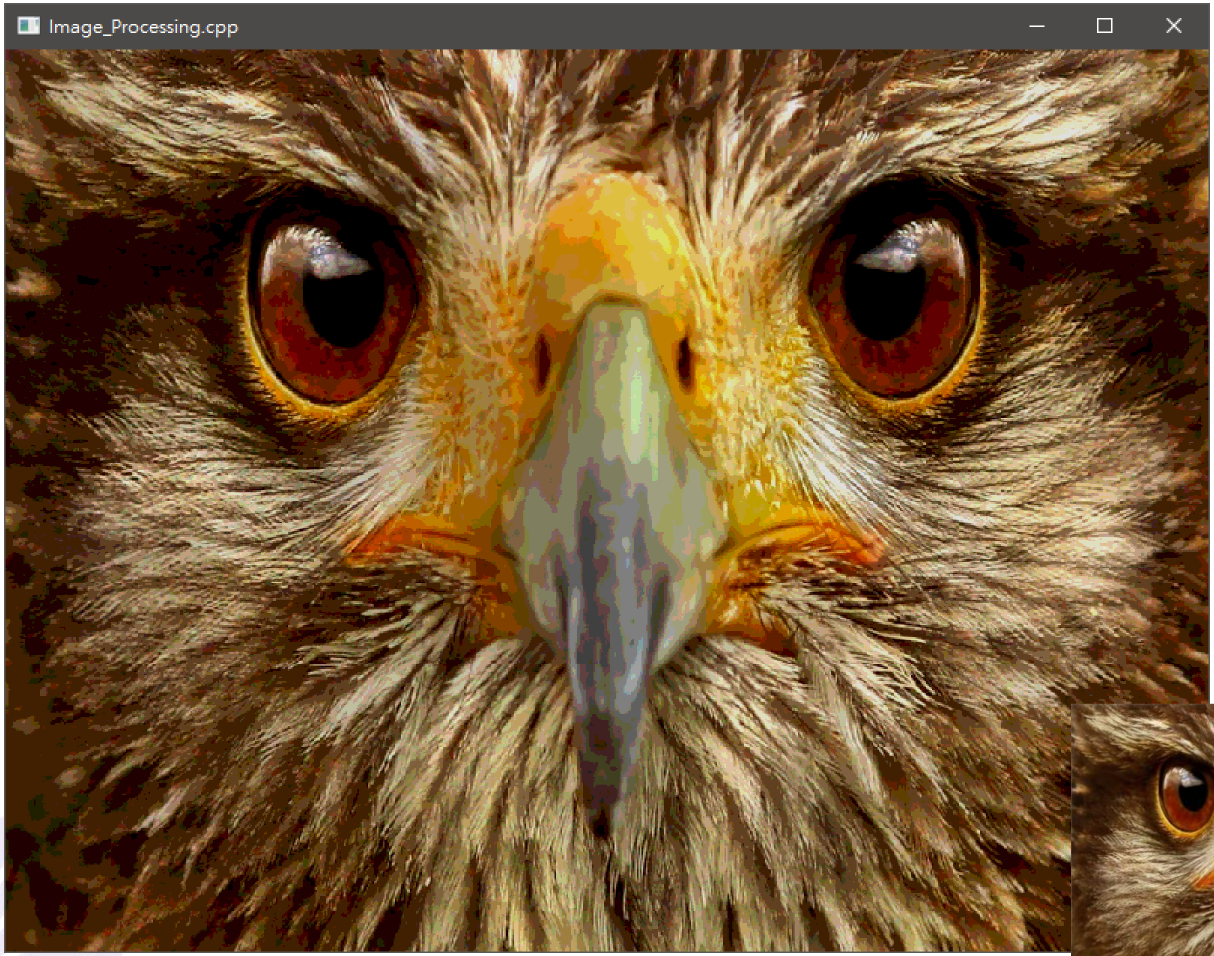
Code: Fragment Shader

```
#version 410 core

uniform sampler2D tex;
out vec4 color;

void main(void)
{
    int half_size = 2;
    vec4 color_sum = vec4(0);
    for (int i = -half_size; i <= half_size ; ++i)
    {
        for (int j = -half_size; j <= half_size ; ++j)
        {
            ivec2 coord = ivec2(gl_FragCoord.xy) + ivec2(i, j);
            color_sum += texelFetch(tex, coord, 0);
        }
    }
    int sample_count = (half_size * 2 + 1) * (half_size * 2 + 1);
    color = color_sum / sample_count;
}
```

Filter Range

half_size = 2	half_size = 5	half_size = 8
		

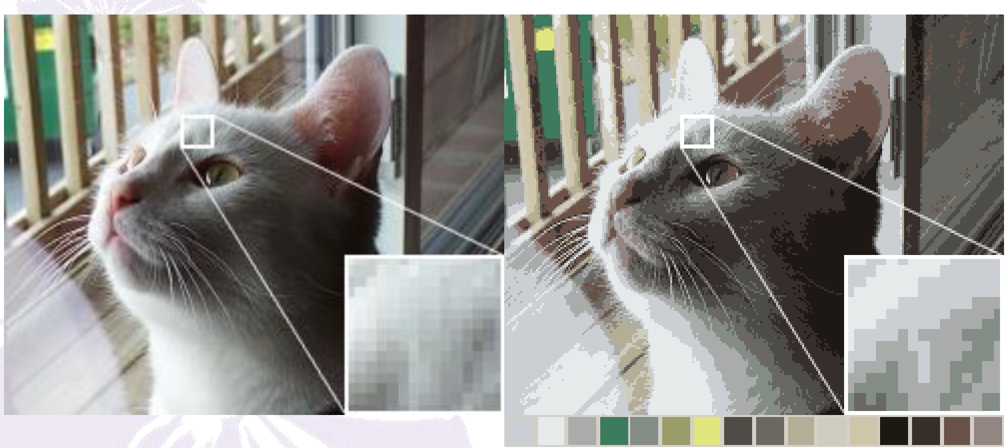
Quantization

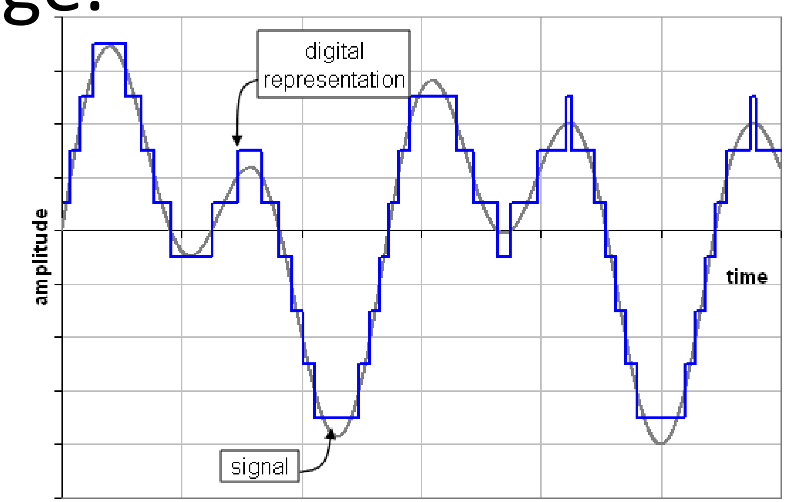


Original Image

Quantization

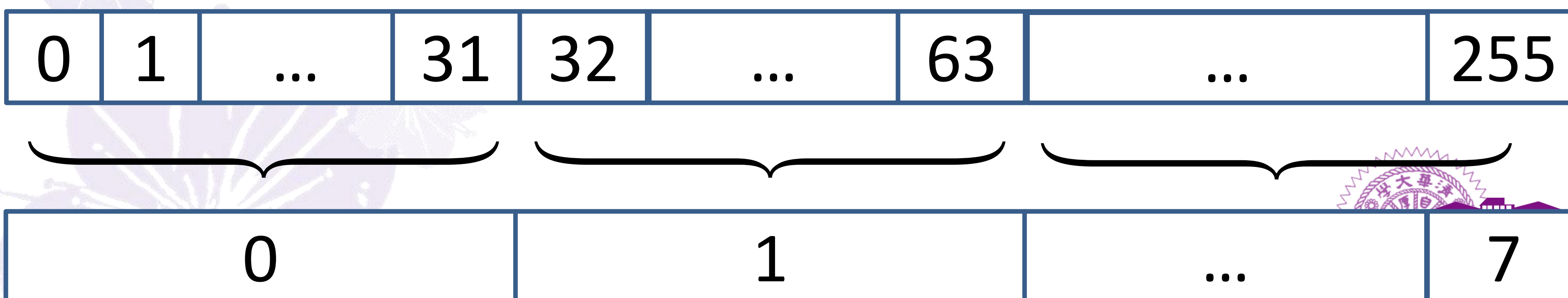
- In computer graphics, color quantization or color image quantization is a process that reduces the number of distinct colors used in an image, usually with the intention that the new image should be as visually similar as possible to the original image.





Code: Fragment Shader

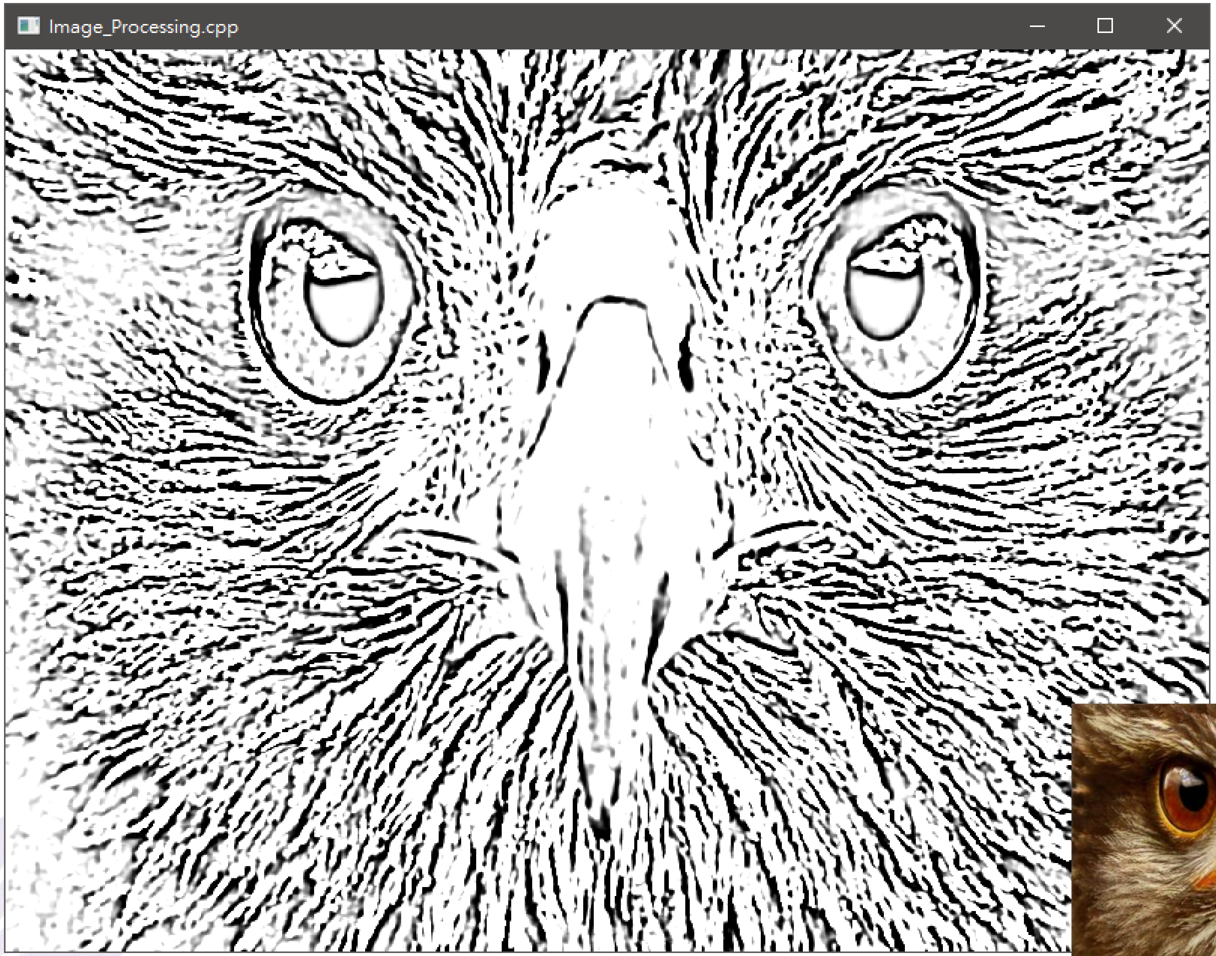
```
#version 410 core

uniform sampler2D tex;
out vec4 color;

void main(void)
{
    float nbins = 8.0;
    vec3 tex_color = texelFetch(tex, ivec2(gl_FragCoord.xy), 0).rgb;
    tex_color = floor(tex_color * nbins) / nbins;
    color = vec4(tex_color, 1.0);
}
```



Quantization Group

Group = 8	Group = 4	Group = 2
		

Difference of Gaussian (DoG)

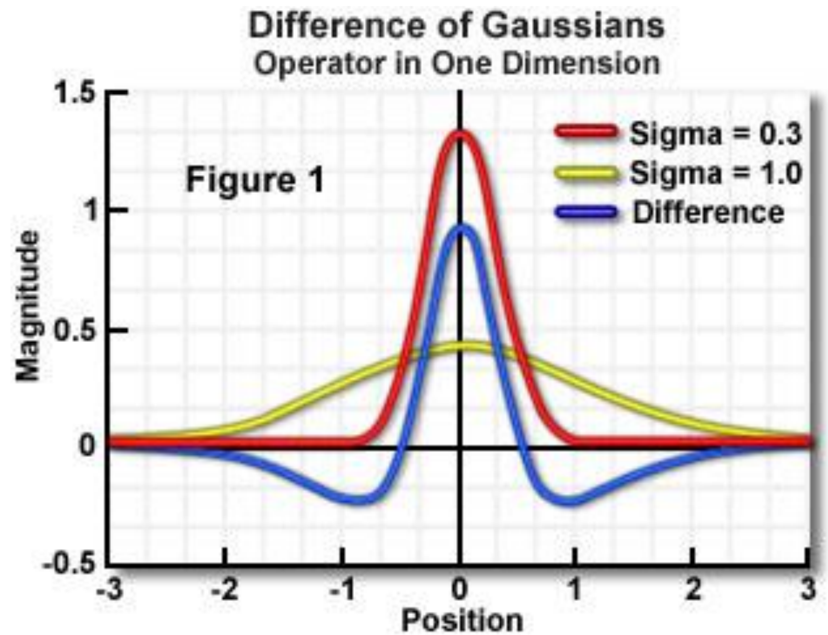


Original Image

Difference of Gaussian (DoG)

- Difference of Gaussians is a feature enhancement algorithm that involves the subtraction of one blurred version of an original image from another, less blurred version of the original

$$\Gamma_{\sigma_1,\sigma_2} = I * \frac{e^{-\frac{x^2}{2\sigma_1{}^2}}}{\sigma_1\sqrt{2\pi}} - I * \frac{e^{-\frac{x^2}{2\sigma_2{}^2}}}{\sigma_2\sqrt{2\pi}}$$



Code: Fragment Shader Parameters

```
#version 410 core

uniform sampler2D tex;
uniform vec2 img_size;
out vec4 color;

in VS_OUT
{
   vec2 texcoord;
} fs_in;

float sigma_e = 2.0f;
float sigma_r = 2.8f;
float phi = 3.4f;
float tau = 0.99f;

float twoSigmaESquared = 2.0 * sigma_e * sigma_e;
float twoSigmaRSquared = 2.0 * sigma_r * sigma_r;
int halfWidth = int(ceil( 2.0 * sigma_r ));
```

$$I * \frac{e^{-\frac{x^2}{\boxed{2{\sigma_1}^2}}}}{\sigma_1\sqrt{2\pi}}$$

Code: Fragment Shader

$$I * \frac{e^{-\frac{x^2}{2\sigma_1{}^2}}}{\sigma_1\sqrt{2\pi}}$$

```
void main(void)
{
        vec2 sum = vec2(0.0);
        vec2 norm = vec2(0.0);

        for ( int i = -halfWidth; i <= halfWidth; ++i ) {
                for ( int j = -halfWidth; j <= halfWidth; ++j ) {
                float d = length(vec2(i,j));
                vec2 kernel= vec2( exp( -d * d / twoSigmaESquared ),
                                   exp( -d * d / twoSigmaRSquared ));

                vec4 c= texture(tex,fs_in.texcoord+vec2(i,j)/img_size);
                vec2 L= vec2(0.299 * c.r + 0.587 * c.g + 0.114 * c.b);

                norm += 2.0 * kernel;
                sum += kernel * L;
               }
        }

        sum /= norm;
        float H = 100.0 * (sum.x - tau * sum.y);
        float edge =( H > 0.0 )?1.0:2.0 *smoothstep(-2.0, 2.0, phi * H );
        color = vec4(edge,edge,edge,1.0 );
}
```

smoothstep

smoothstep(genType edge0, genType edge1, genType x);

- **edge0:** Specifies the value of the lower edge of the Hermite function.
- **edge1:** Specifies the value of the upper edge of the Hermite function.
- **x:** Specifies the source value for interpolation.

smoothstep (Cont'd)

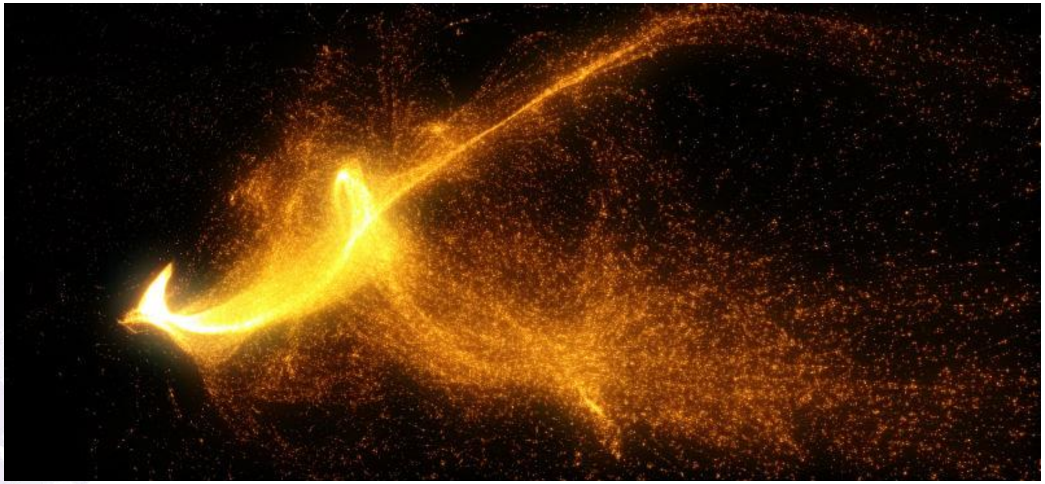
```
smoothstep(genType edge0, genType edge1, genType x);
```

- **Description:** smoothstep performs smooth Hermite interpolation between 0 and 1 when edge0 < x < edge1. smoothstep is equivalent to:

```
genType t;
t = clamp((x - edge0) / (edge1 - edge0), 0.0, 1.0);
return t * t * (3.0 - 2.0 * t);
```

Point Sprites

- In many cases, we would like to render textured points to create particle-like effects
- OpenGL provides a special texture coordinate generation mechanism for point render mode, which is called "Point Sprites"

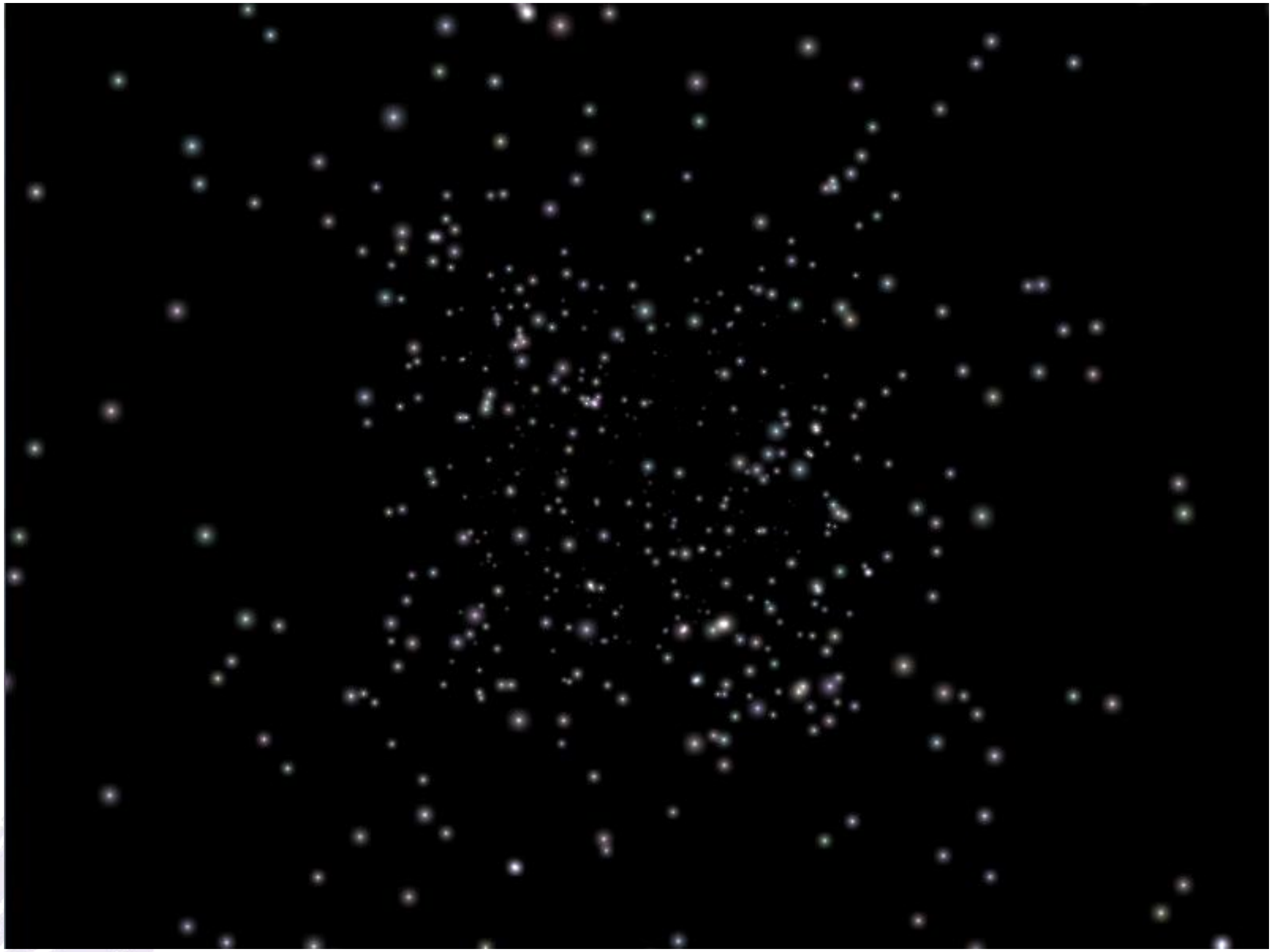


Particle Effect

Point Sprites

- Without point sprites, achieving this type of effect would be a matter of drawing a large number of textured quads
- To use point sprites:
  1. Enable GL_POINT_SPRITES state
  2. Issue draw calls with GL_POINTS primitive mode
  3. In fragment shader, read the generated texture coordinate from gl_FragCoord system variable

Flying Star Field



Code: Vertex Shader

```
#version 410 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;

uniform float time;
uniform mat4 proj_matrix;
flat out vec4 starColor;

void main(void)
{
    vec4 starPosition = vec4(position.xy, position.z + time, 1.0);
    float starSize = 20.0 * starPosition.z * starPosition.z;
    starColor = vec4(color * smoothstep(1.0, 7.0, starSize), 1.0);
    starPosition.z = (999.9 * starPosition.z) - 1000.0;
    gl_Position = proj_matrix * starPosition;
    gl_PointSize = starSize;
}
```

Code: Fragment Shader

```
#version 410 core

layout (location = 0) out vec4 color;

uniform sampler2D tex_star;
flat in vec4 starColor;

void main(void)
{
   color = starColor * texture(tex_star, gl_PointCoord);
}
```

Code: Set Up Data

```
// declare type to represent a star
struct star_t  {
    glm::vec3 position;
    glm::vec3 color;
};

// create and map a vertex buffer object
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, NUM_STARS * sizeof(star_t),
             NULL, GL_STATIC_DRAW);
star_t *star = (star_t*)glMapBufferRange(
       GL_ARRAY_BUFFER, 0,
       NUM_STARS * sizeof(star_t),
       GL_MAP_WRITE_BIT | GL_MAP_INVALIDATE_BUFFER_BIT);
```

Code: Set Up Data (Cont'd)

```
// fill in star data with random values
for(int i = 0; i < NUM_STARS; i++)
{
    star[i].position[0] = (random_float() * 2.0f - 1.0f) * 100.0f;
    star[i].position[1] = (random_float() * 2.0f - 1.0f) * 100.0f;
    star[i].position[2] = random_float();
    star[i].color[0] = 0.8f + random_float() * 0.2f;
    star[i].color[1] = 0.8f + random_float() * 0.2f;
    star[i].color[2] = 0.8f + random_float() * 0.2f;
}
glUnmapBuffer(GL_ARRAY_BUFFER);

// setup vertex array states
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(star_t), NULL);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(star_t),
                      (void *)sizeof(glm::vec3));
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

Code: Render Function

```
glUniform1f(time, currentTime);
glUniformMatrix4fv(proj_location, 1, GL_FALSE, &proj_matrix[0][0]);
glBindTexture(GL_TEXTURE_2D, m_texture);

// enable point sprite
glEnable(GL_POINT_SPRITE);
glEnable(GL_PROGRAM_POINT_SIZE);
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);

glDrawArrays(GL_POINTS, 0, NUM_STARS);
```