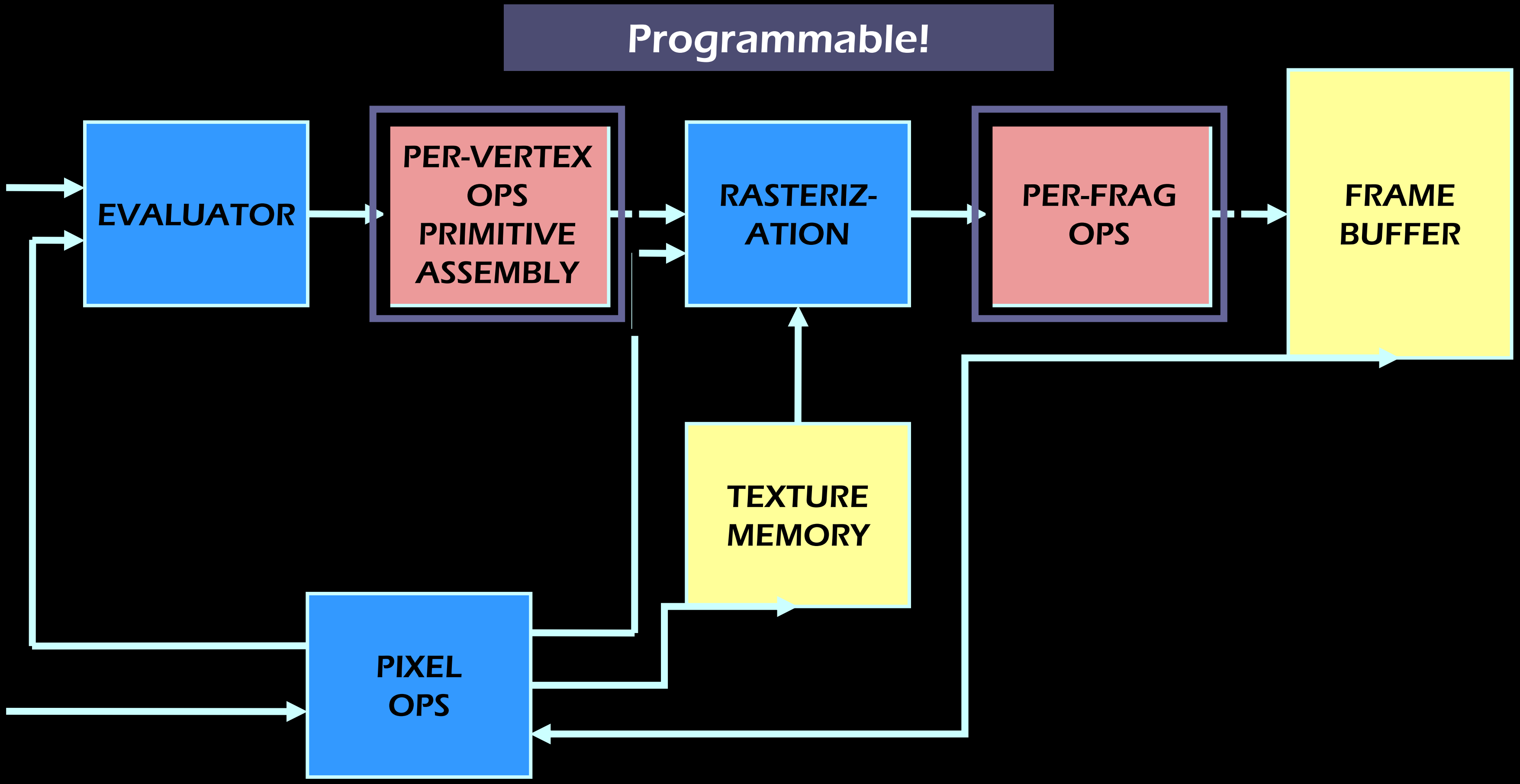
GPU Programming using OpenGL Shading Language

Jimmy Johansson
Norrköping Visualization and Interaction Studio
Linköping University

NVIS

Linköpings universitet

OpenGL Block Diagram



What is a shader?

- "*A shader in the field of computer graphics is a set of software instructions, which is used by the graphics resources primarily to perform rendering effects*". (Wikipedia)

- Vertex Shader
- (Geometry Shader)
- Fragment Shader

Shaders

- Replace parts of the fixed-functionality graphics pipeline

- Many application areas
  - Increasingly realistic materials – metal, stone, wood…
  - Per-pixel lighting
  - Natural phenomena - fire, smoke, clouds...
  - Non-photorealistic materials
  - Image processing
  - …

Vertex Shader (1/2)

- Operates on a single vertex
  - ”No notion” of line, triangle, polygon, etc.
- Replaces parts of the fixed function pipeline
  - Clipping, viewport culling remains
- Input is vertex data
  - Position
  - Color
  - Normals

Vertex Shader (2/2)

- Vertex position transformation using the modelview and projection matrices
- Normal transformations
- Texture coordinate generation and transformation
- Lighting per vertex (or computing values for lighting per pixel)
- Color computation

Fragment Shader (1/2)

- Operates on a single fragment
  - ”No notion” of neighboring fragments
- Replaces parts of the fixed function pipeline
  - Alpha blending, depth test, etc. Remains
- Parallel processing of fragments

Fragment Shader (2/2)

- Computing colors
- Texture application
- Fog computation
- Per-pixel lighting

(Geometry Shader)

- Executed after the vertex shader
- Can generate new primitives from existing primitives
- Input is the whole primitive (3 vertices for a triangle)
- Can then emit zero or more primitives

The GLSL Model

- High-Level Shading Language
- Compiler and Linker part of the OpenGL driver
  - Programs are compiled and linked in run-time
- Alternatives
  - NVIDIA Cg (cross platform: OpenGL & DirectX)
  - Microsoft HLSL (only Windows and DirectX)

GLSL Language Introduction

- Very ”C” like – simple
  - No objects, but structures
- No pointers
- No gotos, no switches

Basic Types

- Scalars
  - float, int, bool, void
- Vectors (2, 3, 4 elements)
  - vec2, vec3, vec4, [b,i]vec{2,3,4}
- Matrices (2x2, 3x3, 4x4 elements)
  - mat2, mat3, mat4
- Samplers
  - sampler1D, sampler2D, sampler3D

Structures

- Combine basic types into structures

```
struct light {
   float    intensity;
   vec3     position;
};
```

Type conversions, type casting

- Very strict type checking

```
float  a = 0;      // Error: int to float
vec2   v = 0.0;    // Error: scalar to vec2
int    i = 0.0;    // Error: float to int
```

- Type conversion (à la C++)

```
float  a = float(0);
vec2   v = vec2(0.0);
int    i = int(0.0);
```

Vector types

- Element groups
  - {x,y,z,w} – Points or Normals
  - {r,g,b,a} – Colors
  - {s,t,p,q} – Texture coordinates
  - Groups cannot be mixed (vec3.xrs is illegal)

Matrices

- Column-major order

```
mat4 m;
m[1] = vec4(2.0);  // Second col all 2.0
m[0][0] = 1.0;     // Upper-left corner set to 1.0
m[2][3] = 3.0;     // Third col, Fourth element
```

- Type conversions

```
mat4 m = mat4(1.0);   // Creates an identity matrix
mat3 n = mat3(vec3(1.0), vec3(2.0), vec3(3.0));
```

Type Qualifiers

- const - compile time constant
- attribute - variables that may change per vertex,
  - Are passed from the OpenGL application to vertex shaders
  - Can only be used in vertex shaders
  - This is a read-only variable
- uniform - variables that may change per primitive
  - May not be set inside glBegin,/glEnd
  - Passed from the OpenGL application to the shaders
  - Can be used in both vertex and fragment shaders
- varying - used for interpolated data between a vertex shader and a fragment shader
  - Available for writing in the vertex shader
  - Read-only in a fragment shader

Built-in Functions 1

- Trigonometry Functions
  - sin, cos, tan, asin, acos, atan, radians, degrees
- Exponential Functions
  - pow, exp, log, exp2, log2, sqrt, inversesqrt
- Common Functions
  - abs, sign, floor, ceil, fract, mod, min, max, clamp, mix, step, smoothstep
- Geometric Functions
  - length, distance, dot, cross, normalize, ftransform, faceforward, reflect, refract

Built-in Functions 2

- Matrix Functions
  - matrixCompMult – component-wise matrix-matrix multiplication
- Vector Relational Functions
  - lessThan, lessThanEqual, greaterThan, greaterThanEqual, equal, notEqual, any, all, not
- Texture Lookup Functions
  - texture{1,2,3}D

Simple Shader Example

- Vertex program

```
void main(void) {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

- Fragment program

```
void main(void) {
gl_FragColor = Vec4(0.0,0.0,1.0,1.0);
}
```

Transformation Example

- RenderMonkey
  - A shader development environment (one of many) for programmers (and artists)
  - Allows rapid prototyping of shaders
  - Full support for DirectX and OpenGL

Texturing Examples

- In order to perform texturing operations in GLSL we need to have access to the texture coordinates per vertex
- GLSL provides attribute variables (one for each texture object)
  - attribute vec4 gl_MultiTexCoord0;
  - …
  - attribute vec4 gl_MultiTexCoord7;
- A Texture object stores texture coordinates and states of a texture
  - (more about this later)

Texturing Examples

- The vertex shader has access to gl_MultiTexCoord[i] to get the texture coordinates
  - specified in the OpenGL application.

- Set the vertex texture coordinates for texture object 0
  - copy the texture coordinates specified in the OpenGL application.
- gl_TexCoord[0] = gl_MultiTexCoord0;

- *gl_TexCoord[i] is a pre*defined varying variable

Texturing Examples

- gl_TexCoord is a varying variable, i.e. it will be used in the fragment shader to access the interpolated texture coordinates
- Access texture values in fragment shader
  - uniform sampler2D tex;
  - tex now contains the activated texture unit (0)
  - Get a texel using texture2D
  - vec4 texture2D(tex, gl_TexCoord[0].st);

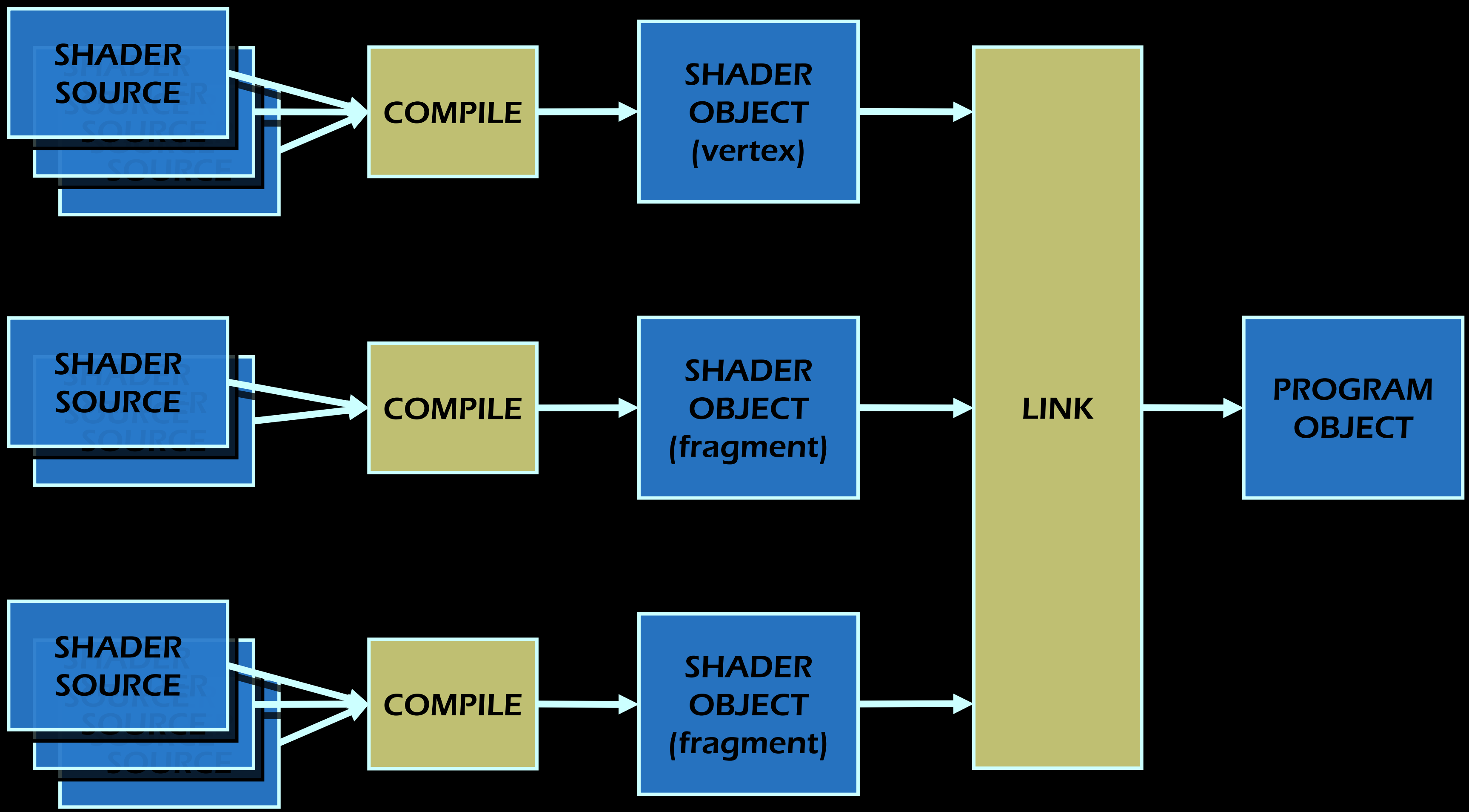
Objects

- Shader Object
  - An array of source strings to be compiled
  - Vertex and Fragment Shader Objects
- Program Object
  - Several shader objects are attached to a Program Object
  - Shader objects are linked together into a Program

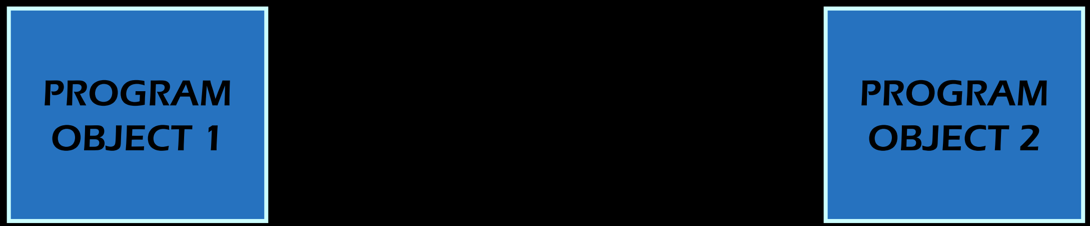
GLSL Program Objects

- One, and only one, active program object
  - Includes both vertex and fragment shaders
- One main() for Vertex and Fragment programs
  - One main() required in Vertex Program
  - One main() required in Fragment Program
- Multiple shader objects linked into one program
  - Convenient for re-use of common parts
  - Create ’libraries’ of useful functions

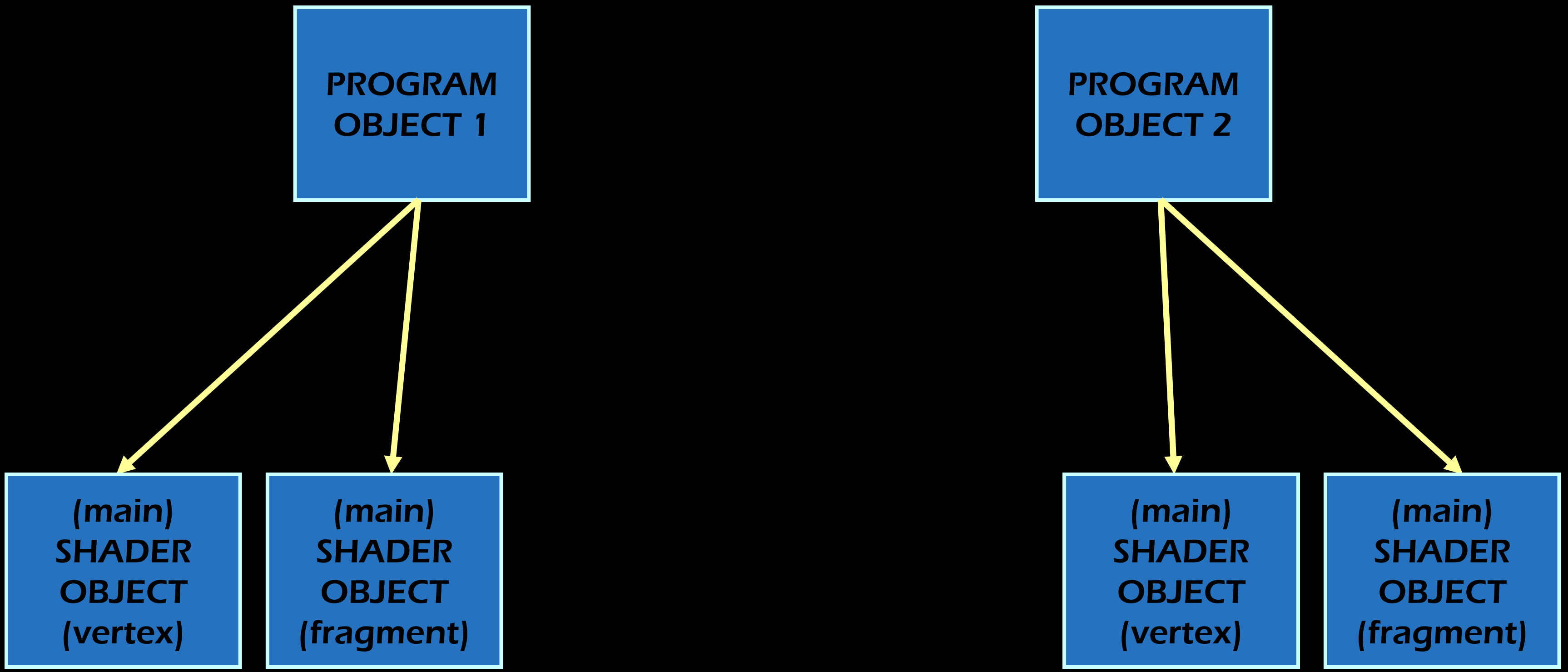
Creating a Shader Program Object



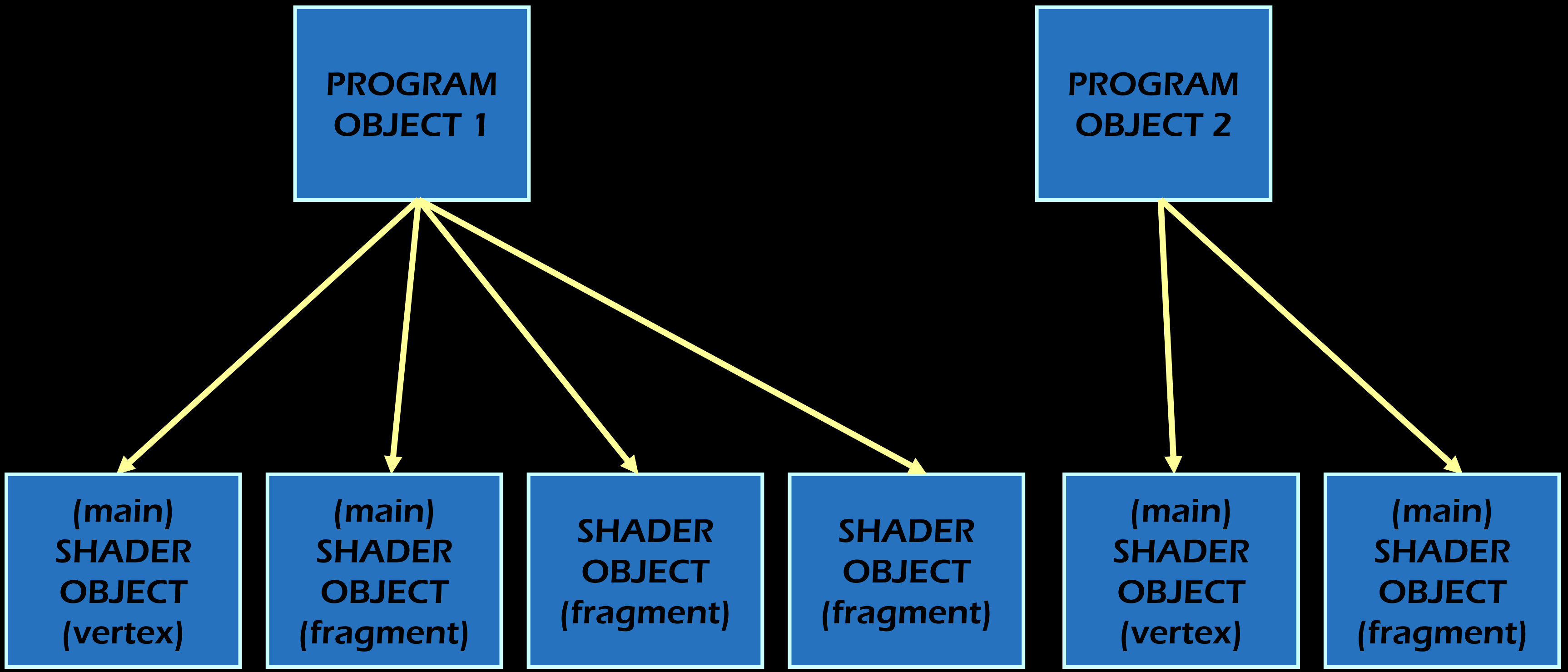
Reusing Shader Program Objects



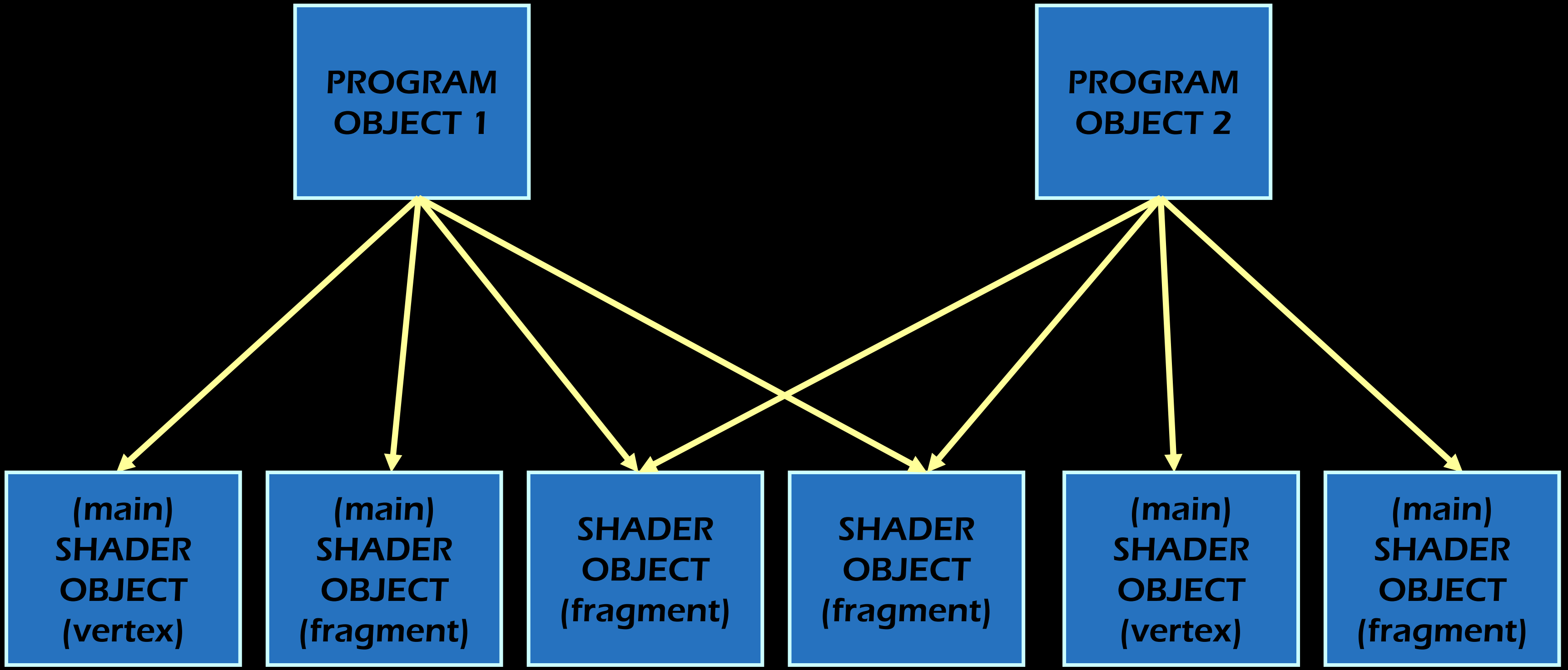
Reusing Shader Program Objects



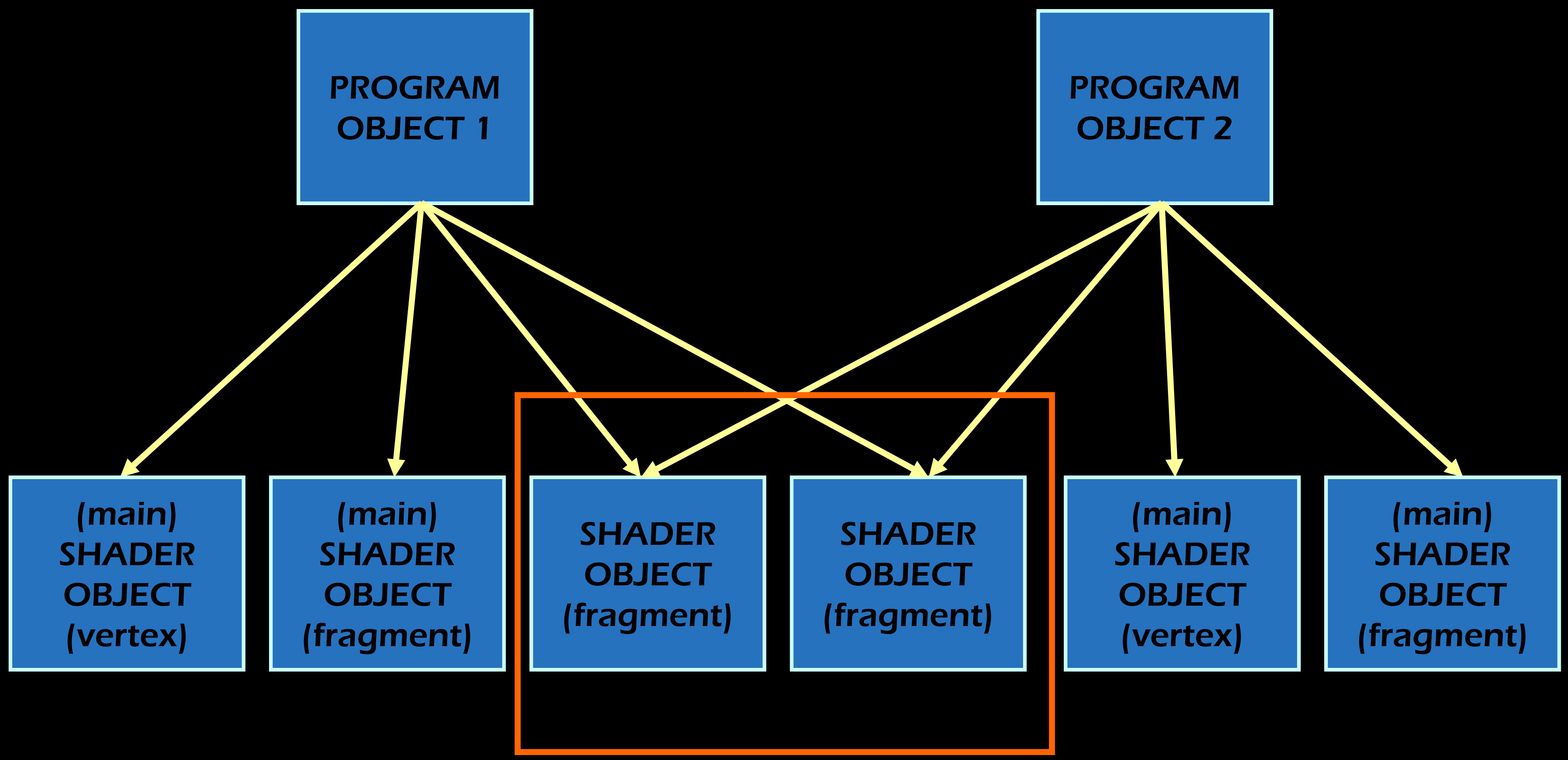
Reusing Shader Program Objects



Reusing Shader Program Objects



Reusing Shader Program Objects



OpenGL 2.1 C API (1/2)

- Creating a Shader Object

```
int object = glCreateShader(GL_VERTEX_SHADER);
     or      glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(object, n, pstrings, plengths);
glCompileShader(object);
```

- Creating a Program Object

```
int program = glCreateProgram();
glAttachShader(program, object);
glLinkProgram(program);
```

- Activate and Deactivate Program

```
glUseProgram(program);
glUseProgram(0);
```

OpenGL 2.1 C API (2/2)

- Checking Compile Log

```
int val;
glGetShaderiv(object, GL_COMPILE_STATUS, &val);
glGetShaderiv(object, GL_INFO_LOG_LENGTH, &val);
glGetShaderInfoLog(object, len, &len, log);
```

- Checking Link Log

```
glGetProgramiv(program, GL_LINK_STATUS, &val);
glGetProgramiv(program, GL_INFO_LOG_LENGTH, &val);
glGetProgramInfoLog(program, len, &len, log);
```

- Cleaning up

```
glDeleteShader(object);
glDeleteProgram(program);
```

OpenGL 2.0 C API (Attributes)

- Used to specify per-vertex data
  - Pressure, temperature, etc.
- OpenGL provides a number of locations for passing in vertex attributes
  - Each location can store 4 floating point numbers (vec4)
- Specified attributes is part of the vertex data sent through the pipeline

OpenGL 2.0 C API (Attributes)

- Attribute Index Lookup

```
int idx;
idx = glGetAttribLocation(program, name);
```

- Specifying Attribute Data

```
glVertexAttrib3f(program, idx, x, y, z);
glVertexAttrib4sv(program, idx, svec);
```

OpenGL 2.0 C API (Uniforms)

- Used to provide a shader with arbitrary data
  - Typically used to supply state that stays constant over many primitives

- Uniform Location Lookup

```
int loc;
loc = glGetUniformLocation(program, name);
```

- Specifying Uniform Data

```
glUniform3f(loc, x, y, z);
glUniform4iv(loc, n, dataptr);
```

  - Only support for int and float

OpenGL 2.0 C API (Samplers)

- Same as Uniform Lookup

```
int loc;
loc = glGetUniformLocation(program, name);
```

- But only accepts scalar integer

```
glUniform1i(loc, texunit);
```

- Texture Unit Assignment

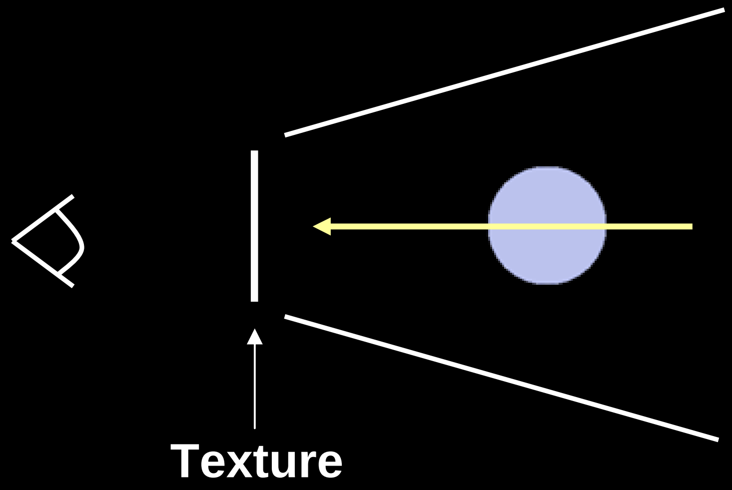
```
glActiveTexture(GL_TEXTUREN);  // N = texunit
glBindTexture(GL_TEXTURE_2D, tex);
```

Multipass Rendering

- Rendering objects (or an entire scene) multiple times
  - each time with different OpenGL settings
- Can achieve effects that are not normally possible in just a single rendering of a scene
  - Reflections
  - Refractions
  - Multipass texturing
  - Blur, Glow, Sharpen, etc.

Multipass Rendering Example - Blur

- First Pass
  - Render the scene to a texture



Multipass Rendering Example - Blur

- Second Pass
  - Render to back buffer
  - Use a fragment shader to compute blur effect using a kernel (mean, Gaussian)

Texture

1/273

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

Multipass Rendering Example - Mirror

- A car with rear-view mirrors
  - how do we draw the right thing in them?
- Render the world twice
  - First pass: draw the world facing forward
  - restrict drawing area to the rear-view mirror
  - set up camera to look back through mirror
  - perform a second drawing pass
- Gives a correct reflected image of the world behind the car

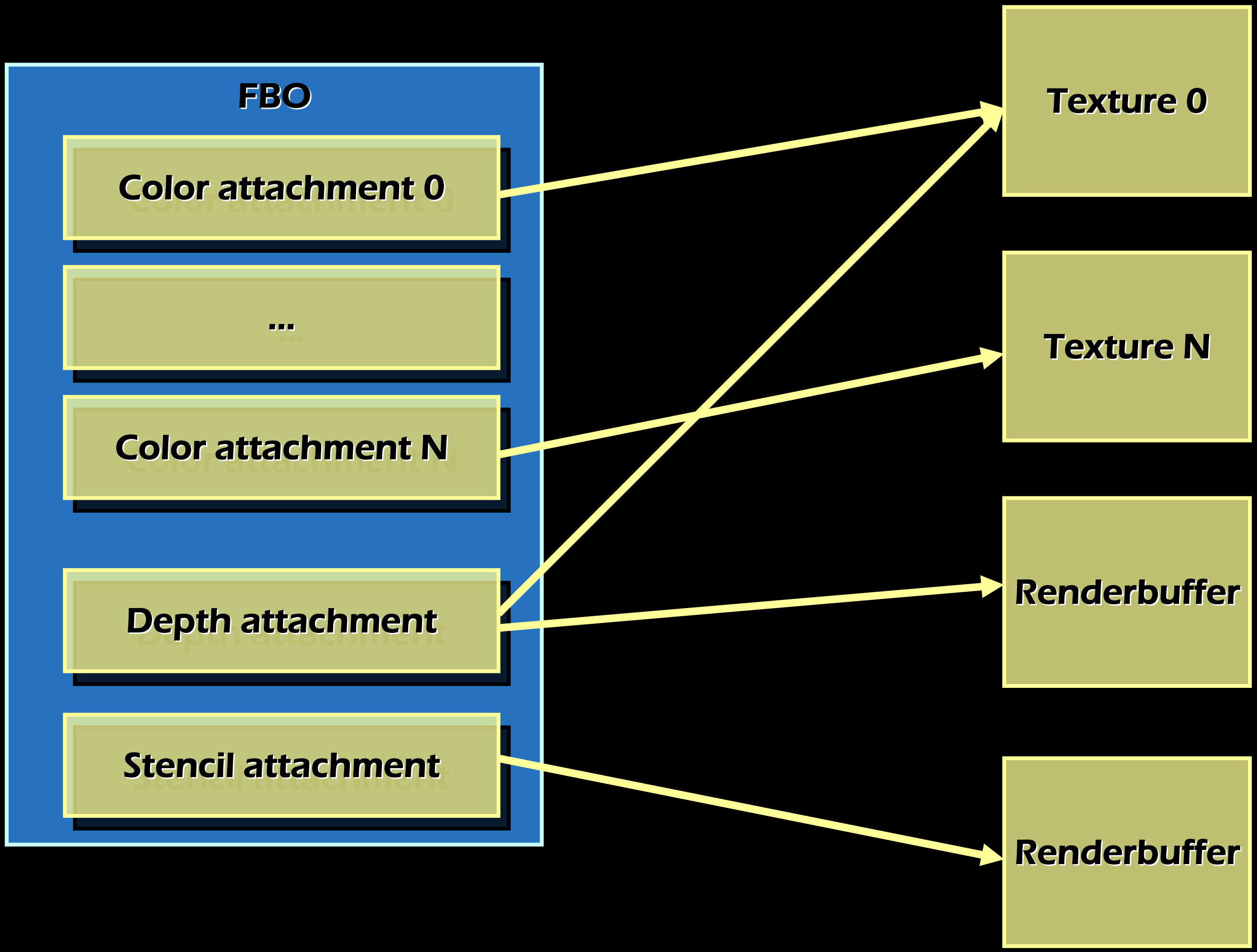
Framebuffer Objects

- An OpenGL framebuffer is a collection of buffers
  - color, depth, stencil, accumulation
- The framebuffer object extension provides a new mechanism for rendering to other destinations
- Destinations known as “framebuffer- attachable images”
  - Renderbuffers
  - Textures

Framebuffer Objects

- Direct rendering to off-screen buffers
- Old style:
  - Draw to framebuffer – copy to texture (slow!)
- Supports floating point format
  - HW blending, precision
- Multiple render targets

Framebuffer Objects



Renderbuffers

- Contains a simple 2D image
- Stores pixel data resulting from rendering
- Cannot be used as textures
- High precision depth buffer

Texture Objects

- Stores texture data
- May control many textures
- Possible to go back to textures previously loaded into texture memory
- The following 3 steps are required
  - Generate texture names
  - Bind (create) texture objects
  - Bind and rebind texture objects (making them available for rendering textured objects)

Texture Objects

```
glGenTextures(1, &tex);
// Generates texture name

glBindTexture(GL_TEXTURE_2D, tex);
// When used for the first time, a texture object is
  created
// If previously created, make it active
```

FBO Example*

```
GLuintfb, depth_rb, tex;

// setup objects
glGenFramebuffersEXT(1, &fb); // frame buffer
glGenRenderbuffersEXT(1, &depth_rb); // render buffer
glGenTextures(1, &tex); // texture
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

// initialize texture
glBindTexture(GL_TEXTURE_2D, tex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
GL_RGBA, GL_UNSIGNED_BYTE, NULL);
// (set texture parameters here)
// ...
```

*The OpenGL Framebuffer Object Extension, Simon Green, NVIDIA Corporation

FBO Example

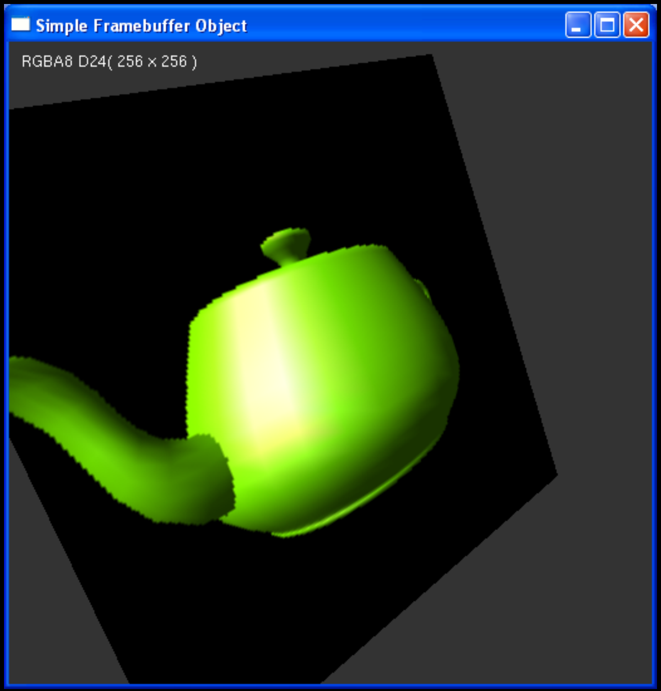
```
// attach texture to framebuffer color buffer
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
  GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, tex, 0);

// render to the FBO
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
// (now rendering to texture)

// render to the window (using the texture)
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
glBindTexture(GL_TEXTURE_2D, tex);
```

Switching between rendering destinations

- Multiple FBOs
  - A separate FBO for each texture
  - Switch using **`BindFramebuffer()`**
- Single FBO
  - Attach textures to different color attachments
  - Switch using **`glDrawBuffer()`** or **`glDrawBuffers()`**

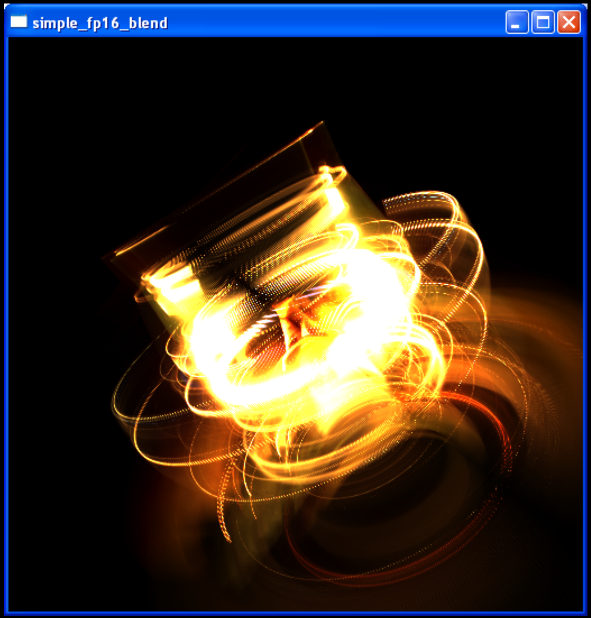
Simple Framebuffer Object

1. Render teapot to FBO
2. Apply to rotating polygon



Additive Floating Point Blending

1. Render scene to FBO using additive blending
2. Display FBO
3. Change scene (for example, rotate object)
4. Go to step 1



GPGPU Programming

- General-purpose computation on GPUs
- Move computations (not graphics computations) from the CPU to the GPU
- Exploit the GPU's extremely parallel architecture

GPGPU Programming

- In general – only a speed-up of 2 – 3 times
  - Specific cases: 10 - 20 times
- Very specific environment (compare multicore CPUs)
- GPUs are designed for and driven by video games
  - Programming model is unusual & tied to computer graphics
  - Programming environment is tightly constrained
- Underlying architectures are:
  - Inherently parallel
  - Largely secret
- Cannot directly “port” code written for the CPU

GPGPU Examples

- Signal processing
- Physics simulations
- Speech/image recognition
- Image segmentation and processing
- …

Shader Debugging

- Actually quite hard
  - Traditional stepping not always applicable
  - Application specific problems
- Create shader incrementally
  - Encode values into RGBA of output (graphical printfs)
  - Make modules, separate functions/files
- Debugging tools
  - Render Monkey, etc.