UiO : University of Oslo

INF3320: Computer Graphics and Discrete Geometry

The programmable pipeline

Christopher Dyken, Martin Reimers and Johan Seland
Corrections and additions by André R. Brodtkorb

October 8, 2014

The programmable pipeline

Real Time Rendering:

- The Graphics Processing Unit (GPU) (Chapter 3)

The Red Book:

- The OpenGL Shading Language (Chapter 15)

Other sources:

- GLSL spec:
  `http://www.opengl.org/documentation/specs/`

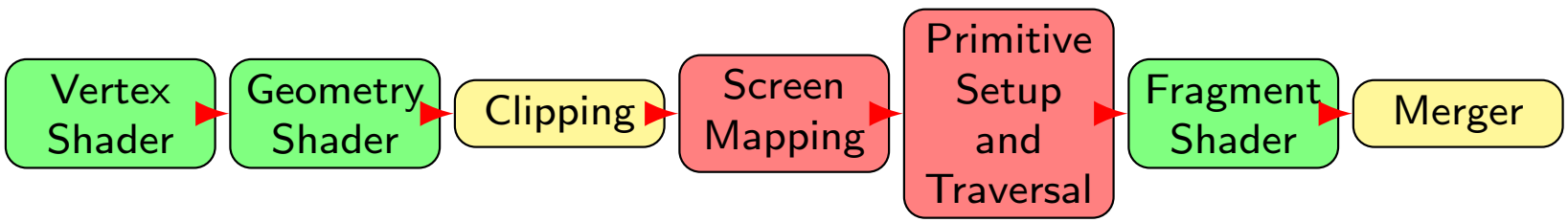
Pipeline operations

OpenGL 2.1 provides a *programmable* pipeline:

Program objects replace part of the pipeline ops with *shaders*, executed on the GPU:

- vertex shaders can replace vertex transform
- geometry shaders can replace some of primitive setup
- fragment shader can replace some of texturing and fragment operations

Green = programmable, yellow = configurable, red = fixed



Example shader setup

main.cpp

```
GLuint v,f,p;
void setShaders() {
  v = glCreateShader(GL_VERTEX_SHADER);
  f = glCreateShader(GL_FRAGMENT_SHADER);

  char *vs = textFileRead("vs-code.glsl");
  char *fs = textFileRead("fs-code.glsl");
  glShaderSource(v, 1, &vs,NULL);
  glShaderSource(f, 1, &fs,NULL);

  glCompileShader(v);
  glCompileShader(f);

  p = glCreateProgram();
  glAttachShader(p,v);
  glAttachShader(p,f);

  glLinkProgram(p);
  glUseProgram(p);
}
```

vs-code.glsl

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

fs-code.glsl

```
void main() {
  gl_FragColor = vec4(0.4,0.4,0.8,1.0);
}
```

Why a programmable pipeline?

The fixed function pipeline is, well, fixed - limited applications.
Programmability yields a whole new world of possibilities:

Examples

- Other lighting models
- Realistic materials (metals, stone, wood)
- Procedural textures
- Image Processing
- Non-photorealistic rendering
- General purpose programmming (GPGPU)
- High performance computing

The possibilities are endless!

GLSL - OpenGL Shading Language

Shading Languages

Can implement shaders using *shading languages*

- API for shader programming
- Shading languages started with Pixar Renderman, for making high quality movies etc.
- Today heavily used for real time rendering

Shading languages for consumer hardware:

- Nvidia Cg (OpenGL/DirectX)
- Microsoft HLSL (DirectX)
- OpenGL Shading Language - GLSL (OpenGL)

GLSL - The OpenGL shading language

- GLSL - API to OpenGL programmable pipeline
- Based on C/C++
- Shader is specified as source code
- Compile - Link - Run cycle
- Compiler is embedded in driver
- Hardware independent
- Communication with OpenGL through various mechanisms
- Version 1.3/1.4 with OpenGL 3.0/3.1
- Latest version: GLSL 4.5 (2014)
- Extension/deprecation mechanism

GLSL at a glance

- C/C++-like syntax

Examples

- looping `for`, `while`, `do-while`
- selection `if`, `if-else`
- basic datatypes `bool`, `int`, `float`, `struct`, arrays
- functions (call by value)
- no pointers
- no strings
- no type promotion `float f = 0.0;`

GLSL at a glance

- Built-in Vector (2d, 3d, 4d)

Examples

- `vec4 v = vec4(1.0, 0.0, -3.0, 2.0 );`
- Swizzling

```
vec3 w = v.yzx;
vec3 c = v.gbr;
// (0.0, -3.0, 1.0)
```

- Indexing

```
float f = v[2]; // -3.0
```

- Component-wise operation

```
vec3 d = w + c;
w.x = v.x + f;
```

- Component-wise mathops

```
vec4 v = v+w;
```

GLSL at a glance

- Built-in Matrix
  $(2 \times 2, 3 \times 3, 4 \times 4)$

Examples

- `mat2 m = mat2(1.0, 2.0, 3.0, 4.0 ); //` $\begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{bmatrix}$
- `mat4 n = mat4(1.0); // n = diag(1.0)`
- `vec2 v = m[0]; //` $v = (1.0, 2.0)^T$
- `float f = m[1][0]; //` $f = 3$

GLSL at a glance

- Built-in standard library of functions

Examples

- Trigonometric: `sin`, `tan`, `atan`...
- Common: `abs`, `sign`, `min`, `max`...
- Geometric: `length`, `dot`, `cross`...
- Vector relational
- Interpolation:
  - mix(x,y,a) = (1.0-a)*x + a*y
  - step(e,x): x <= e ? 0.0 : 1.0
  - smoothstep(e0,e1,x)

GLSL at a glance

- Variable qualifiers

Examples

- `const`
- `uniform` (over primitive)
- `attribute` (GLSL<1.3)
- `varying` (GLSL<1.3)
- `in` (input variable)
- `out` (output variable)

GLSL at a glance

- Built in variables

Examples

- Uniforms: gl_ModelViewMatrix
- Attributes: gl_color, gl_vertex, gl_normal
- Varyings: gl_FrontColor, gl_BackColor
- Special: gl_Position, gl_FragDepth, gl_FragColor

More GLSL

- Preprocessor, similar to the C/C++ preprocessor macros, conditionals, pragma
- OpenGL extension handling, e.g.
  `#extension GL_EXT_geometry_shader4 : enable`
- user defined functions
  - mostly like C-functions
  - call by value (no references)
  - in/out/inout qualifiers, e.g
    `void getSomeValue(out int retval)`
    Don't confuse this with shader input and outputs!

GLSL - OpenGL Communication

The host (CPU) program pass data to shaders (GPU) through

1. State variables (accessible in shaders)
2. Used defined variables (uniform, attribute, varying)
3. Textures

The GPU passes data back to the CPU through a framebuffer

Passing variables to shaders

Three types of variables

1. Uniform variables
   - "Global" variables for shader program (common uniform namespace for all shaders)
   - Can be updated once pr. primitive
2. Vertex attribute variables
   - Allows for passing additional vertex data
   - Can be updated for every vertex
3. Interpolated variables (varyings)
   - For communication from vertex to fragment shader
   - Value is set per vertex and interpolated per fragment
     - smooth - perspective correct interpolation
     - flat - constant over primitve (flat shading)
     - noperspective - linear interpolation in screen coords

Setting a uniform

1. Activate the program
2. Get the location of the uniform
3. Pass value of uniform

Passing uniforms

```
glUseProgram( prog );
int loc = glGetUniformLocation( prog, ``beta'');
glUniform1f( loc, 0.25 );
```

- Can pass `glUniformMatrix` and other variants
- Cannot change between begin/end

Passing vertex attributes

1. Activate the program
2. Get the location of attributes
3. For every vertex, pass vertex data

Example

```
glUseProgram( prog );
int loc = glGetAttribLocation( prog, ``tangent'' );
gl_Begin( GL_TRIANGLES );
glNormal3fv( normals[0] );
glColor3fv( colors[0] );
glVertexAttrib3fv( loc, tangets[0] );
glVertex3fv( vertices[0] );
.
gl_End();
```

- Many variants of `glVertexAttrib*f`
- Can also use attribute arrays - like vertex arrays: `glEnableVertexAttribArray(GLint loc)` and `glVertexAttribPointer`

Varying (interpolated) Variables

1. Specify with same name/type in vertex and fragment shaders
2. Set value in vertex shader (attribute/out)
3. Read in fragment shader (varying/in)

Example

```
/* VERTEX SHADER */
#version 130
smooth out vec3 Normal;
void main() {
  Normal = gl_NormalMatrix*gl_Normal;
  ...
}

// Varying interpolated over each primitive
// during rasterization

/* FRAGMENT SHADER */
#version 130
smooth in vec3 Normal;
void main() {
  vec3 diffuse = dot( Normal, lightVec );
  ...
}
```

Specifying textures

1. Activate texture as regular OpenGL
2. Specify texture as uniform (glGetLocation etc.)
   `glUniform1i( loc, i )` for `GL_TEXTUREi`
3. Read and use in either vertex or fragment shader

Shader texture example

```
uniform sampler2D texture;
void main() {
  vec4 tex = texture2D(texture, gl_TexCoord[0].st);
  .
}
```

- Other sampler types, 1-3D, Cube, Shadow1D/2D

Dependent texture reads

```
uniform sampler1D coords;
uniform sampler3D volume;
void main() {
  vec3 texCoords = texture1D(coords, gl_TexCoord[0].s);
  vec3 volumeColor = texture3D(volume, texCoords);
  .
}
```

Shaders

Vertex Shader

- Fully programmable stage, replace fixed stage
- Execute same shader for all vertices
- Has *no* knowledge of neighboring vertices
- Has *no* knowledge of primitive
- Can alter the (x,y,z,w) coordinate of a vertex
- 1 vertex in, 1 vertex out
- Read only attributes/out, uniforms
- Read texture (not required in spec)
- Write to varyings/out

Variables in Vertex shader

Inputs (Read Only)

```
attribute vec4 gl_Vertex;
attribute vec3 gl_Normal;
attribute vec4 gl_Color;
attribute vec4 gl_SecondaryColor;
attribute vec4 gl_MultiTexCoord0;
...
attribute vec4 gl_MultiTexCoord7;
attribute float gl_FogCoord;
```

Vertex shaders *must* output `gl_Position`

Varying Outputs (Read/Write)

```
varying vec4 gl_FrontColor;
//enable GL_VERTEX_PROGRAM_TWO_SIDE for back color
varying vec4 gl_BackColor;
varying vec4 gl_FrontSecondaryColor;
varying vec4 gl_BackSecondaryColor;
varying vec4 gl_TexCoord[]; MAX=gl_MaxTextureCoords
varying float gl_FogFragCoord;
```

Vertex shaders

Pass-through vertex shader

```
#version 130
void main() {
  // the following  lines provide the same result
  // gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix
  //                                   * gl_Vertex;
  // gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  gl_Position = ftransform();
}
```

Fragment (Pixel) Shader

- Fully programmable stage, replace fixed stage
- Execute same shader for all fragments
- Has *no* knowledge of neighboring fragments
- 1 fragment in, 1 or 0 fragments out (discard)
- Can not change the (x, y) coordinate of a fragment
- Read only varyings/in, uniforms, attribute, texture
- Can change a fragments RGBA value and depth
- Fragment can output to multiple buffers, using `drawBuffers` and `gl_FragData[]`

Variables in Fragment Shader

Varying Inputs (Read only)

```
in   vec4 gl_Color;
in   vec4 gl_SecondaryColor;
in   vec4 gl_TexCoord[ ]; // MAX=gl_MaxTextureCoords
in   float gl_FogFragCoord;
```

Special inputs (Read only)

```
vec4 gl_FragCoord; // pixel coordinates
bool gl_FrontFacing;
```

Special Output Variables

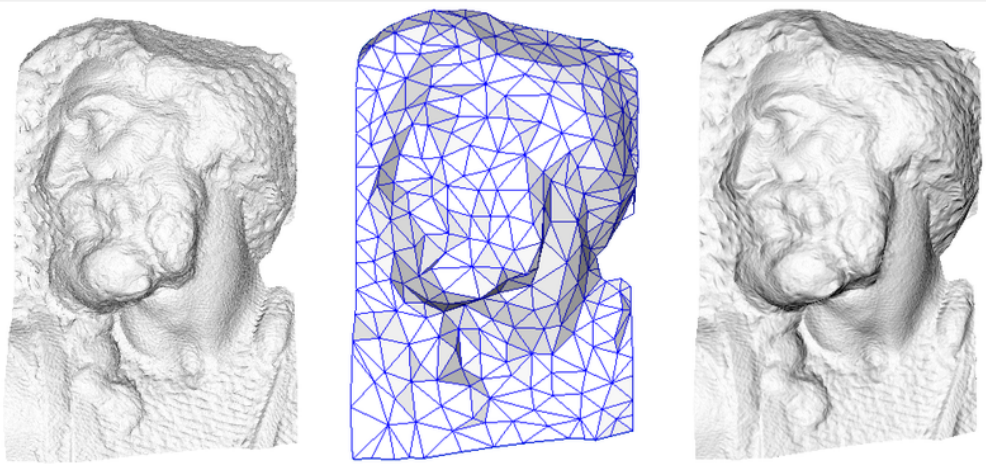
```
vec4 gl_FragColor;
vec4 gl_FragData[ ]; // MAX= gl_MaxDrawBuffers
float gl_FragDepth; // DEFAULT=glFragCoord.z
```

Fragment shader example: per pixel shading

```
smooth in vec3 Normal;
void main () {
  vec3 difMat  = gl_FrontMaterial.diffuse.xyz;
  vec3 specMat = gl_FrontMaterial.specular.xyz;

  vec3 l = gl_LightSource[0].position.xyz - gl_FragCoord.xyz
  vec3 lightVec  = normalize(l);
  vec3 normalVec = normalize(Normal);
  vec3 eyeVec = vec3(0.0, 0.0, 1.0);
  vec3 halfVec = normalize(lightVec + eyeVec);

  // calculate diffuse component
  vec3 diffuse = max(dot(normalVec, lightVec), 0.0) * difMat;

  // calculate specular component
  vec3 s = max(dot(normalVec, halfVec), 0.0);
  vec3 specular = pow(s, 32.0) * specMat;

  // combine diffuse and specular contributions
  gl_FragColor.rgb = diffuse + specular;
}
```

Object space normal mapping

```
uniform sampler2D normal_tex;
void main() {
  // decompress normal (map comps from [0,1] to [-1,1])
  vec3 n = 2*texture2D( normal_tex, gl_TexCoord[0].xy )-1;
  n = gl_NormalMatrix*n; // transform normal
  vec3 lightVec = gl_LightSource[0].position;
  vec3 eyeVec   = vec3(0.0, 0.0, 1.0);
  vec3 halfvec  = normalize(lightVec + eyeVec);
  gl_FragColor = max(dot(lightVec,n),0.0)*gl_Color
               + pow(max(dot(halfVec,n),0.0),40)*vec4(1.0);
}
```



original mesh
4M triangles

simplified mesh
500 triangles

simplified mesh
and normal mapping
500 triangles

Excellent results, but difficult to animate (must update texture).

Geometry Shader

Only on DX10 HW (SM 4) - *not mentioned in Red Book*

- Extension: `GL_ARB_geometry_shader4`
- Programmable additional stage, after vertex transform
- Per primitive processing, i.e. point, line, triangle
- 1 primitive in, n primitives out (limited amount, fixed type)
- Read uniform, vertex attributes, texture
- Write varyings, vertex attributes
- Stream out: can write generated data to a buffer
- Pipeline may terminate here

Usage:

- Adjacency computations
- Kill primitive, refine primitives
- Simple tessellation
- Particle effects, fur, hair, ...
- ...

Geometry Shader in GLSL

- About the same output types as VS
- Inputs: arrays of length `gl_VerticesIn`
- EmitVertex() adds vertex `gl_Position` to current primitive
- EndPrimitive() ends the current primitive
- Set input type with `glProgramParameteriEXT(prog, GL_GEOMETRY_INPUT_TYPE_EXT, type)`
- Set output type with `glProgramParameteriEXT(prog, GL_GEOMETRY_OUTPUT_TYPE_EXT, type)`

Simple Geometry Shader

A pass-through geometry shader (GLSL 1.30)

```
#version 130
#extension GL_ARB_geometry_shader4 : enable
void main() {
  for(int i = 0; i < gl_VerticesIn; ++i) {
    gl_FrontColor = gl_FrontColorIn[i]; // copy color
    gl_Position = gl_PositionIn[i]; // copy position
    EmitVertex();
  }
  EndPrimitive();
}
```

Geometry Shader example

Geometry shader for drawing Bezier curve (GLSL 1.20)

```
#version 120
#extension GL_EXT_geometry_shader4 : enable
void main(void){
    for(int i=0; i<64; i++) {
      float t = i / (64.0-1.0);
      float m = 1.0-t;
      vec4 b = vec4(m*m*m, m*m*t, m*t*t, t*t*t);
      vec4 p = gl_PositionIn[0]*b.x
                    + gl_PositionIn[1]*b.y
                    + gl_PositionIn[2]*b.z
                    + gl_PositionIn[3]*b.w;
      gl_Position = p;
      EmitVertex();
     }
    EndPrimitive();
}
```

Some newer features of GLSL

GLSL 3.0 extensions

- Instanced drawing (replication)
- Transform feedback (record transformed data)
- 32 bit depth buffer support

GLSL $>$ 3.0 extensions

- Deprecation mechanism: some functionality/variables no longer available in shaders, but can use GL_ARB_compatibility_extension
- Additional shader steps (tessellation control and evaluation)
- Atomic counters (Opengl 4.2)
- Bindless textures (Opengl 4.4)
- Direct state access (Opengl 4.5)
- ...

Interfacing OpenGL

- GLSL is only half the story
- OpenGL has been extended with several API calls to allow for compiling, installing and interfacing shaders
- Unfortunately, using a shader involves many steps
- There are libraries to ease the process

Handling extensions with Glew

OpenGL Versions

- When you create an OpenGL context using GLUT, you get version 1.1
- Ask for higher versions, and you get these if supported.
- But no extensions are available.

Extensions-mechanism to add functionality

- One vendor: GL_*APPLE*_client_storage
- More vendors: GL_*EXT*_geometry_shader4
- Blessed by Arch. Review Board: GL_*ARB*_framebuffer_object

GL Extension Wrangler (GLEW): Crossplatform C/C++ library for handling extensions

Simple Glew example

```
#include <GL/glew.h>
void main(int argc, char **argv) {
  glutInit(&argc, argv);
  ...
  glewInit();
  if (GL_ARB_geometry_shader4)
    printf("Ready for geometry shading!\n");
  else {
    printf("NOT ready for geometry shading\n");
    exit(1);
  }
}
```

Create Shader Objects

1. Create OpenGL managed data structure for each shader
2. Provide strings of GLSL that provides the shaders source code

Example

```
GLhandle vs, fs;
vs = glCreateShader( GL_VERTEX_SHADER );
fs = glCreateShader( GL_FRAGMENT_SHADER );

vssrc = readFile(''vertex-shader.glsl'');
fssrc = readFile(''fragment-shader.glsl'');

glShaderSource( vs, 10, vssrc, NULL );
glShaderSource( fs, 20, fssrc, NULL );
```

Compile shaders and create program object

1. Compile each shader separately
2. Vertex and Fragment Shaders combined in a Program Object

Example

```
glCompileShader(vs);
glCompileShader(fs);

uint prog = glCreateProgram();
glAttachShader( prog, vs );
glAttachShader( prog, fs );
```

Link, install and use program

1. Our compiled program must be linked
2. then install the program as part of the current GL-state
3. We issue geometry the normal way, our shader is executed instead of the fixed function pipeline

Example

```
glLinkProgram(prog):
glUseProgram(prog);
renderSomething();
```

Creating a shader

1. `glCreateShader` creates a shader of vertex, fragment, or geometry type.
2. `glShaderSource` specifies the source code of a shader
3. `glCompileShader` compiles the source code of a shader to object code
4. `glGetShader` and `glGetShaderInfoLog` reports errors.

Creating a program

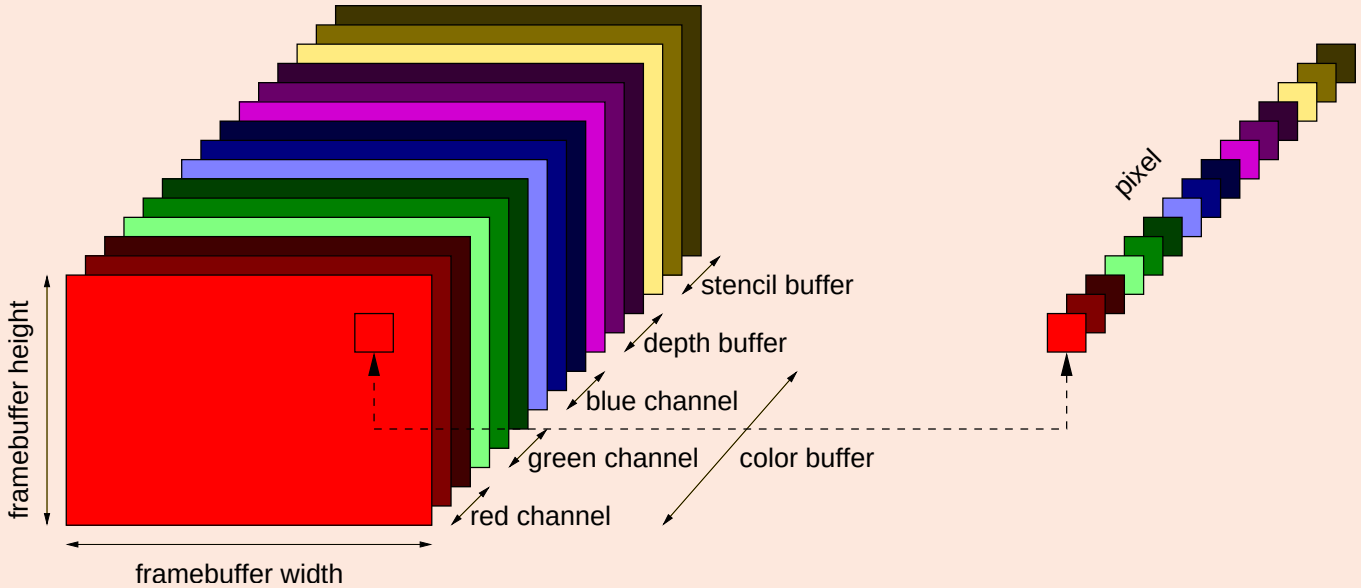
1. `glCreateProgram` creates a program object.
2. `glAttachShader` attaches a shader to a program object.
   Several shaders can be attached, but only one of each type.
3. if a geometry shader is attached, use `glProgramParameteriEXT` to specify input and output primitive types and max generated vertices.
4. `glLinkProgram` links attached shaders to form a program.
5. `glGetProgramiv` and `glGetProgramInfoLog` reports errors.

Using a program

1. `glUseProgram( program )` says that we want to use that program.
2. `glUseProgram( 0 )` says that we want to use the fixed-function pipeline.

Framebuffer Objects

A framebuffer is a set of logical buffers



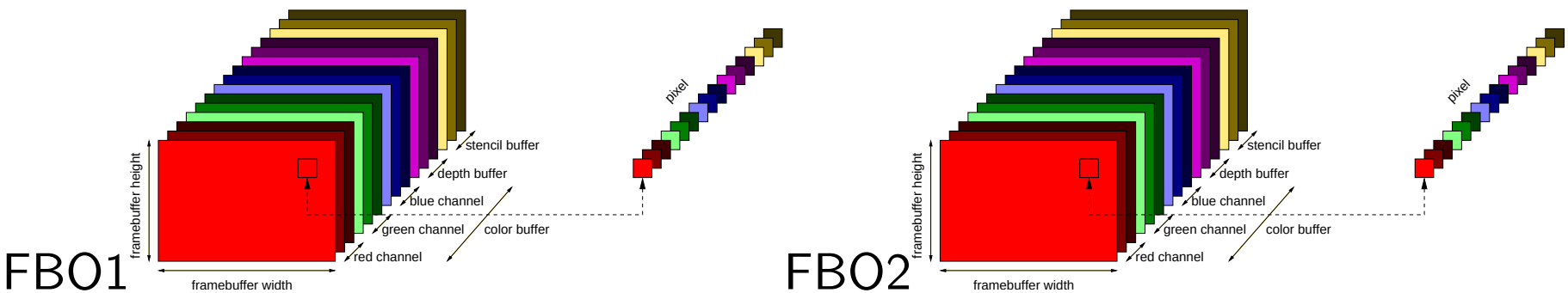
- color buffers (RGBA, front/back, left/right, typically 16–32 bits)
- depth buffer (typically 24–32 bits)
- stencil buffer (typically 0–8 bits)
- accumulation buffer (precision like color buffer or higher), ...

A pixel is the contents of all logical buffers for a location.

GLUT/QT/GTK creates the *onscreen framebuffer*.

FrameBuffer Objects - FBOs - are *offscreen framebuffers*:

- use rendering to define textures
- ping-pong rendering
- multi-pass rendering with textures as in-between storage
- full-screen effects like toning, motion blur, HDR...
- GPGPU, etc...
- FBO = Render targets in DirectX



An extension (`GL_EXT_framebuffer_object`) in OpenGL 3.0

Create and render to different framebuffers using the same context.

A FBO is a set of *logical buffer attachment points*

Can attach *render buffers* and *textures* to the attachment points

- create using `glGenFramebuffers`
- bind to using `glBindFramebuffer`
- attach render buffer using `glFramebufferRenderbuffer`
- attach texture using `glFramebufferTexture2D`
- check if OK using `glCheckFramebufferStatus`

Create and use a framebuffer object

```
GLuint fbo;
glGenFramebuffers(1, &fbo);
glBindFramebuffer(GL_FRAMEBUFFER, fbo);

// Set up a render-buffer (depth)
GLuint depthbuffer;
glGenRenderbuffers(1, &depthbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, depthbuffer);
glRenderbufferStorage(GL_RENDERBUFFER,GL_DEPTH_COMPONENT,
                      width,height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
                          GL_DEPTH_ATTACHMENT,
                          GL_RENDERBUFFER,depthbuffer);

// Render
glPushAttrib(GL_VIEWPORT_BIT);
glViewport(0,0,width, height);
renderSomething(); // Output goes to the FBO

// Restore and unbind
glPopAttrib();
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Render buffers are off-screen render targets

- create using `glGenRenderbuffers`
- bind to using `glBindRenderbuffer`
- specify size and storage using `glRenderbufferStorage`
- attach to an FBO using `glFramebufferRenderbuffer`

Creating and attaching a render buffer

create a 256 × 256 depth buffer

```
glGenRenderbuffers( 1, &depth );
glBindRenderbuffer( GL_RENDERBUFFER, depth );
glRenderbufferStorage( GL_RENDERBUFFER,
                       GL_DEPTH_COMPONENT24,
                       256, 256 );
```

attach the render buffer to a FBO

```
glBindFramebuffer( GL_FRAMEBUFFER, fbo ):
glFramebufferRenderbuffer( GL_FRAMEBUFFER,
                           GL_DEPTH_ATTACHMENT,
                           GL_RENDERBUFFER, depth);
```

Render to textures

- render directly to an existing texture
- attach to an FBO using `glFramebufferTexture2D`

Render to a texture (color buffer)

Initialize texture target

```
glGenTextures(1, &img);
glBindTexture(GL_TEXTURE_2D, img);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
```

Attach texture

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, img, 0);
```

Render texture

```
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
glPushAttrib(GL_VIEWPORT_BIT);
glViewport(0,0,width, height);
renderSomething(); // output to texture render-buffer img
glPopAttrib();
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Use texture in next rendering step

```
glGenTextures(1, &img);
...
```

Can also generate mipmaps, e.g. with `glGenerateMipmapEXT`

Multiple render targets in a FBO

- Each texture attached with `glFramebufferTexture2D(GL_FRAMEBUFFER,GL_COLOR_ATTACHMENTi, GL_TEXTURE_2D,img,0);`
- Check how many with `glGetIntergeri(GL_MAX_COLOR_ATTACHMENTS, &maxbuffers);`
- Use `glDrawBuffers` to tell which to render to
- each target rendered differently with shader program

Render to multiple buffers

Initialize render buffers

```
GLenum buffers[] = {GL_COLOR_ATTACHMENT0,GL_COLOR_ATTACHMENT1;}
glDrawBuffers(2, buffers);
```

Shader code

```
#version 130
void main() {
    gl_FragData[0] = vec4(0.0, 1.0, 0.0);
    gl_FragData[1] = vec4(0.0, 0.0, 1.0);
}
```

The last slide

Some litterature

- "The Orange Book", OpenGL Shading Language, 3rd edition
- GLSL info: `http://www.opengl.org/documentation/glsl/`
- GLSL quick-ref:
  `http://www.opengl.org/sdk/libs/OpenSceneGraph/glsl_quickref.pdf`
- GLSL tutorial:
  `http://www.lighthouse3d.com/opengl/glsl/`
- FBO spec.:
  `http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt`

Next time: Polygonal meshes