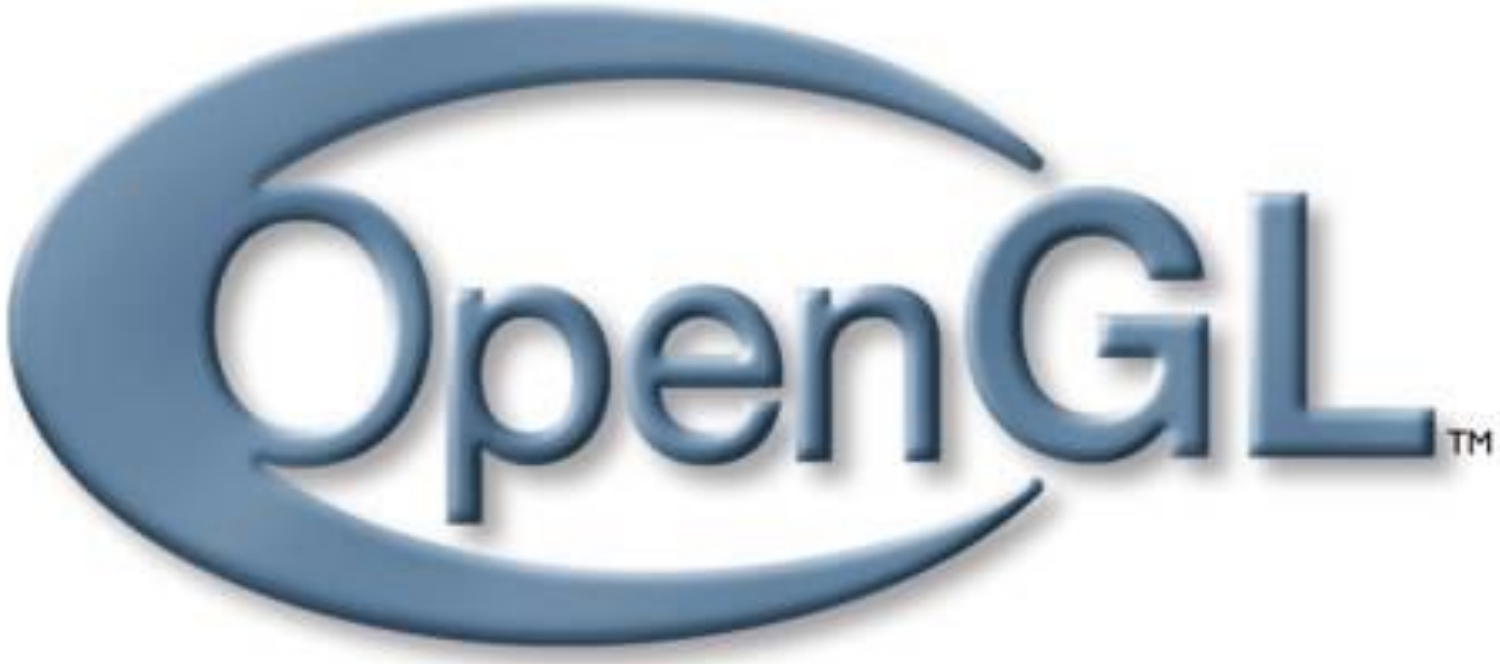


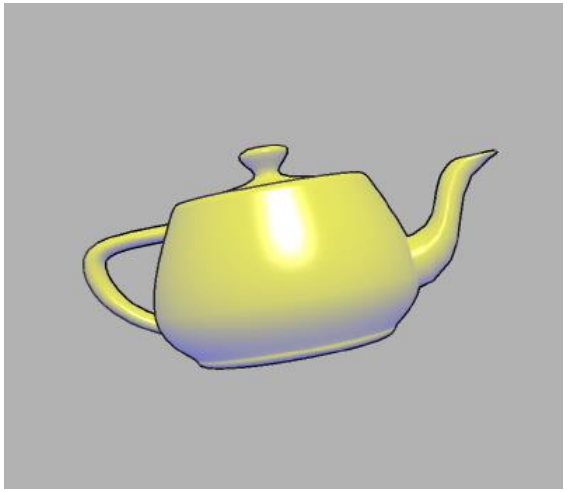
part 6

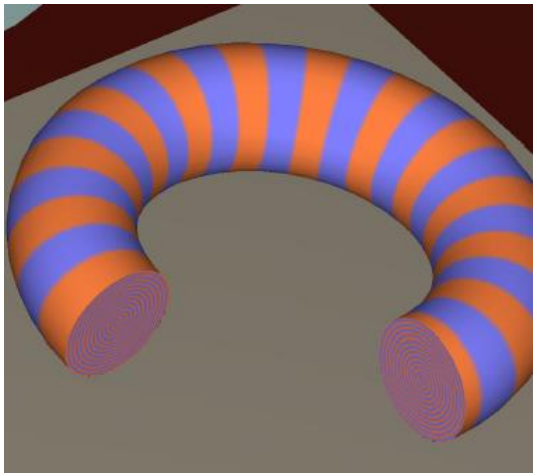
Martin Samuelčík
http://www.sccg.sk/~samuelcik
Room I4

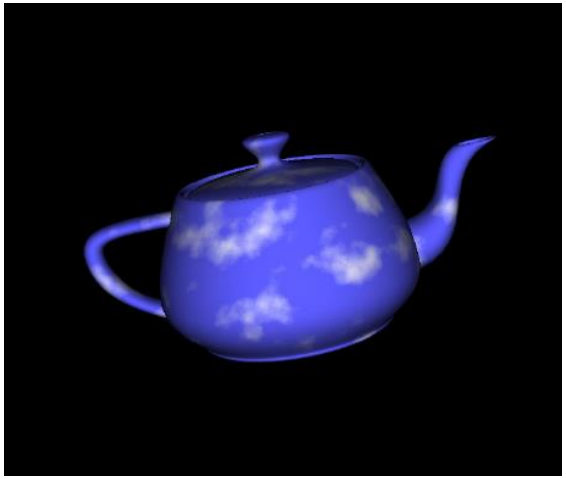
Shaders

- User defined programs that run on graphics hardware
- Replacing some fixed functionality with arbitrary code
- Replacing in several parts of pipeline – vertex, fragment, geometry, tessellation, compute shaders
- Adding great flexibility to rendering
- Bringing new effects

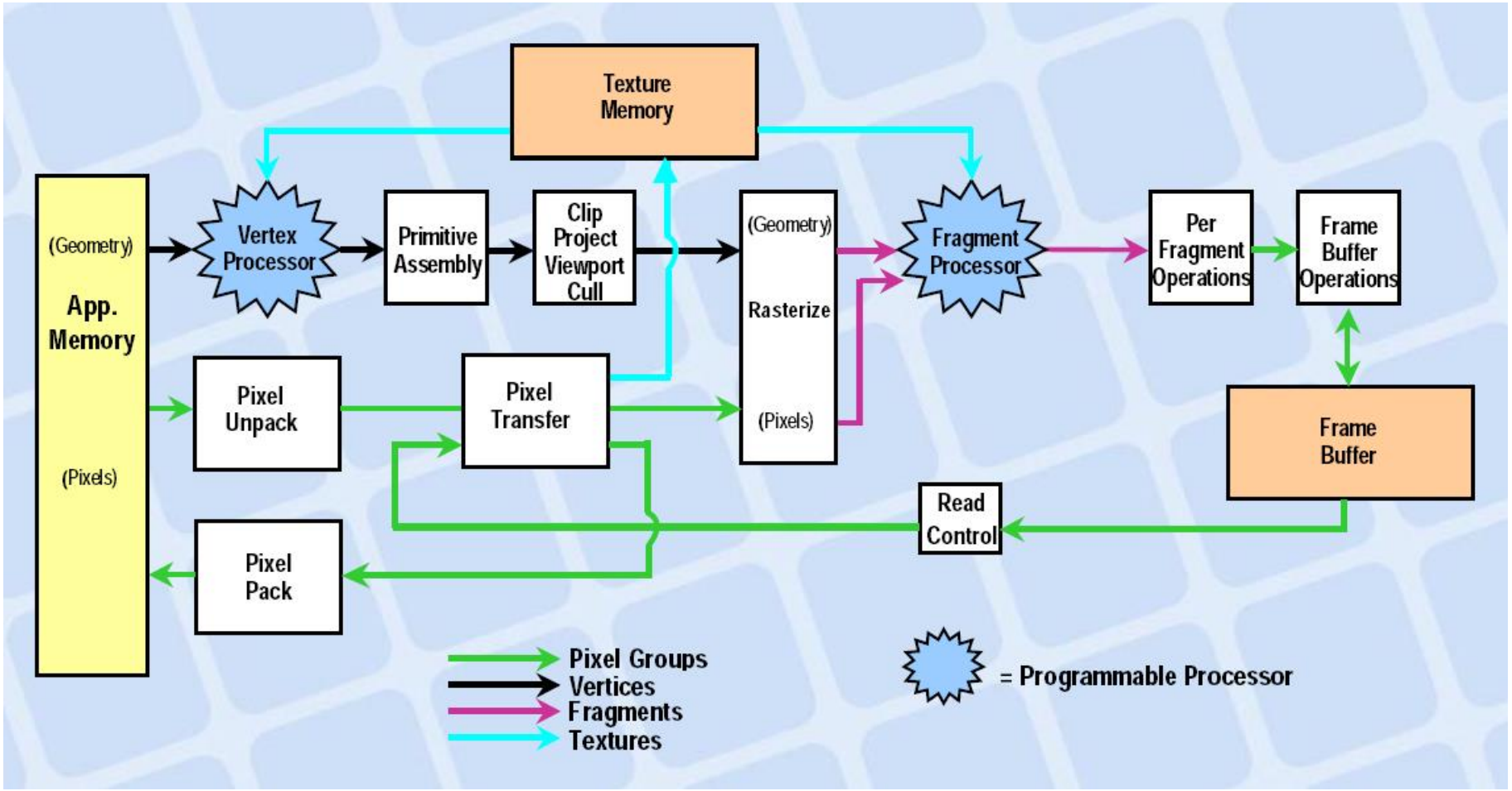
Shaders







Vertex & Fragment shaders

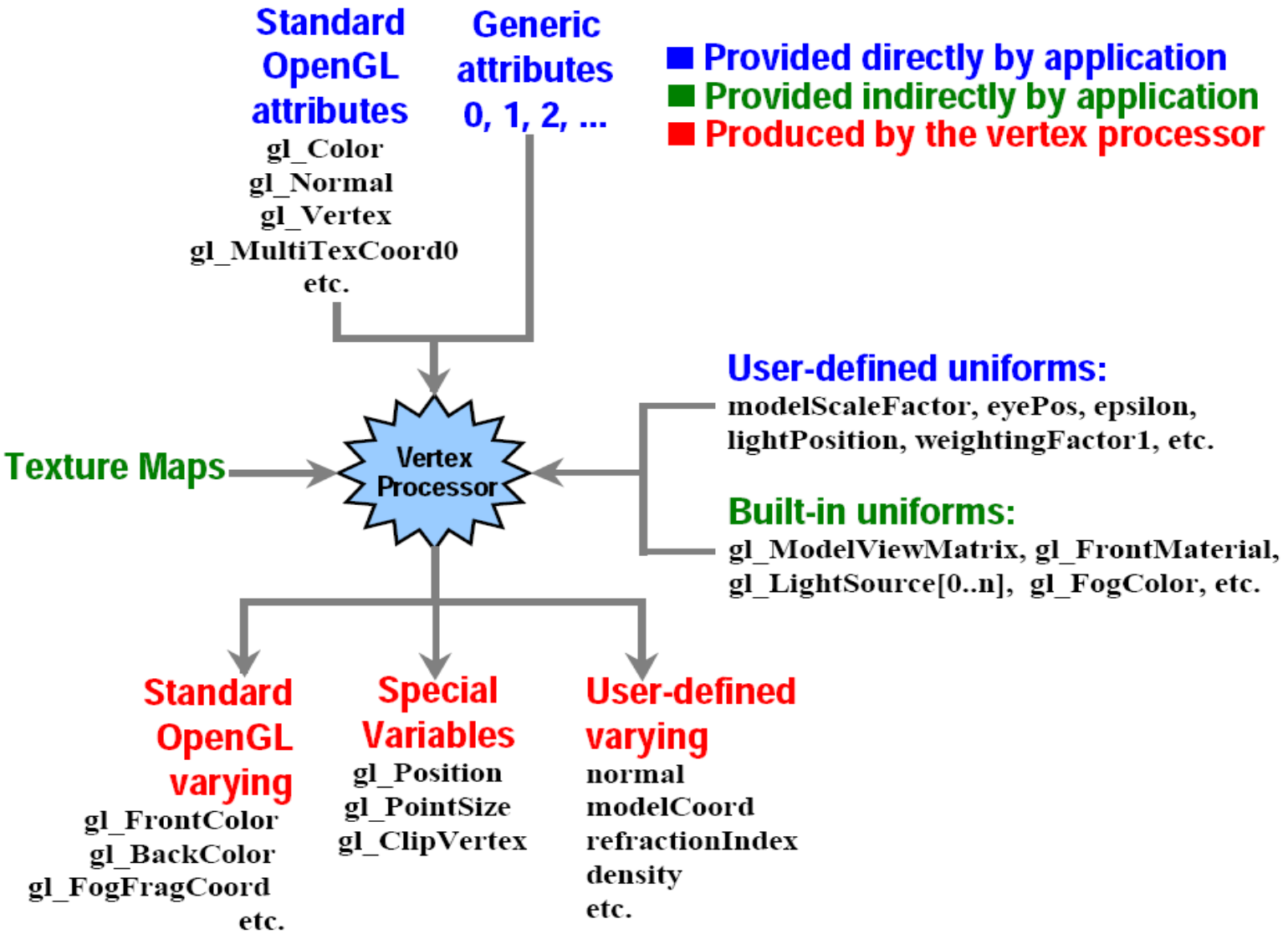


OpenGL®

Vertex processor

- Running for each vertex separately
- Changing attributes of each vertex
- Replaces: Vertex transformation, Normal transformation, normalization and rescaling, Lighting, Color material application, Clamping of colors, Texture coordinate generation, Texture coordinate transformation
- Does not replace:– Perspective divide and viewport mapping, Frustum and user clipping, Backface culling, Primitive assembly, Two-sided lighting selection, Polygon offset, Polygon mode

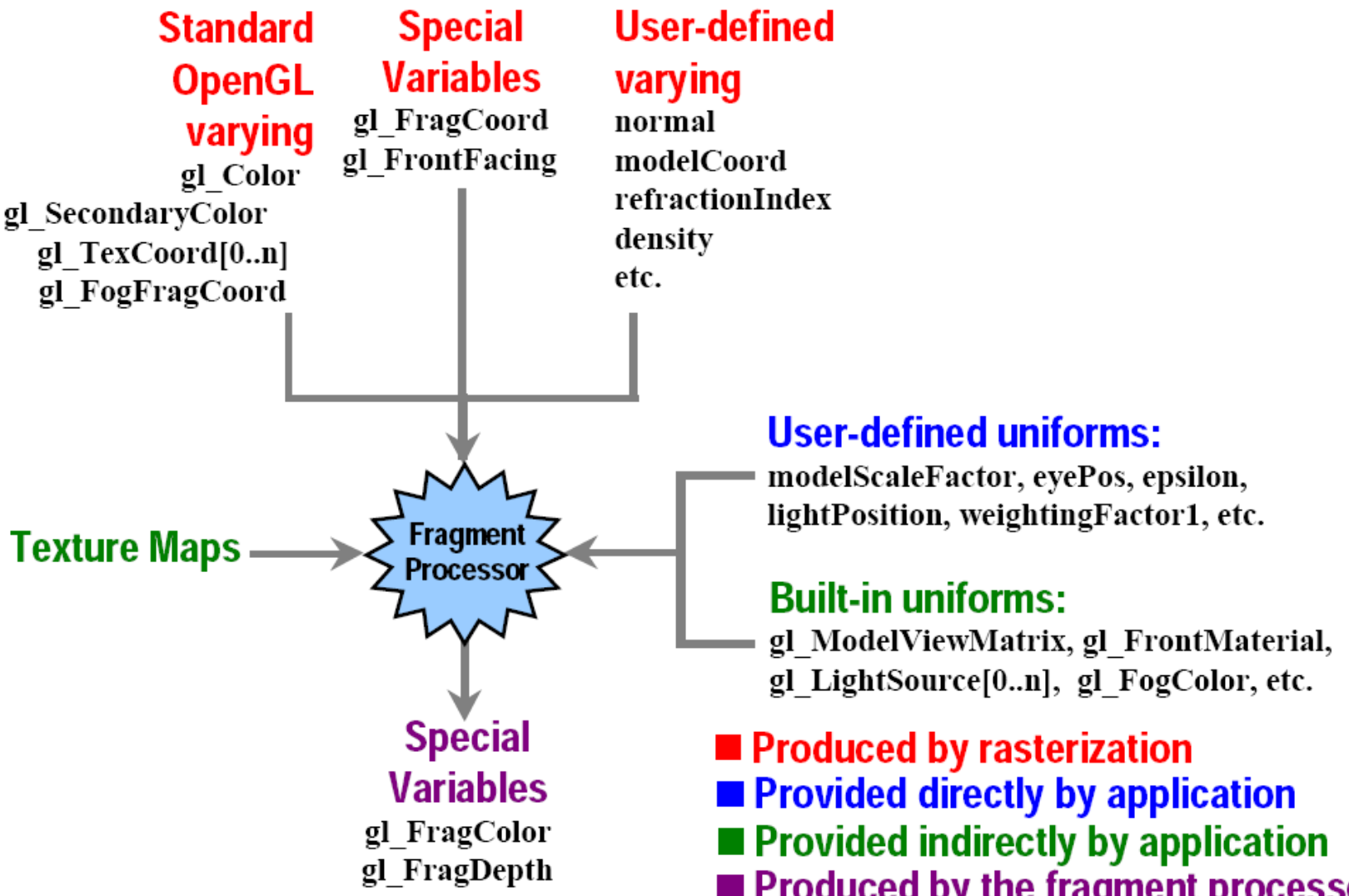
Vertex processor



Fragment processor

- Running for each fragment separately
- Computing final color and depth of every fragment from interpolated attributes
- Flexibility for texturing and per-pixel operations
- Replaces: Ops on interpolated values, Texture access, Texture application, Fog, Color sum

Fragment processor



Shaders programming

- Languages for programming shaders

OpenGL

OpenGL 2.0 shaders

- Unification of different approaches
- New language for writing shaders (GL_ARB_shading_language_100)
- New shader programs (GL_ARB_fragment_shader & GL_ARB_vertex_shader)
- Management using shader objects (GL_ARB_shader_objects)

Shader objects

- OpenGL managed data
- Consist of state and data
- Given handles, identifiers by OpenGL
- Handle is used by application to refer to the object
- Can be shared across contexts
- Applications can provide data for objects and modify their state

Shaders management

- **GLuint glCreateShader(GLenum shaderType)**
  - Creates shader object and returns its id
  - shaderType - GL_VERTEX_SHADER, GL_FRAGMENT_SHADER
- **void glShaderSource(GLuint shader, GLsizei count, const GLchar **string, const GLint *length)**
  - Shader source code is supplied to OpenGL
  - count – number of elements in string and length arrays
  - string – array of sources
  - length – array of sources lengths
- **void glCompileShader(GLuint shader)**
  - Compilation of shader object

Program objects

- Container for multiple shader objects
- Set of shaders that need to be linked together
- **GLuint glCreateProgram(void)**
  - Create empty program object, returns its id
- **void glAttachShader(GLuint program, GLuint shader)**
  - Add shader to program object
- **void glDetachShader(GLuint program, GLuint shader)**
  - Remove shader from program object
- **void glLinkProgram(GLuint program)**
  - Link all shaders in program object together
- **void glUseProgram(GLuint program)**
  - Set program object as active one

Shaders control

- Getting result of shaders compilation and programs linkage + text description of errors

```
Glint status;
glGetShaderiv(shader, GL_COMPILE_STATUS,  &status);
GLchar log[MAX];
GLsizei length;
glGetShaderInfoLog(shader, MAX, &length, log);
GLint status;
glGetProgramiv(program, GL_LINK_STATUS, &status);
GLchar log[MAX];
GLsizei length;
glGetProgramInfoLog(program, MAX, &length, log);
```

Uniforms & attributes

- Uniforms – variables sent to shader from application, not changing often, for example transformations
- Attributes – variables sent to shader from application, these are generic vertex attributes that can change each vertex, for example texture coordinates
- Each uniform and attribute variable has identifier in program object - location

Uniforms

- **GLint glGetUniformLocation(GLint program, const GLchar* name)**
  - Returns identifier of uniform with given name in program object
- **void glUniform{1|2|3|4}{f|i}(GLint location, TYPE val)**
  - Set float or integer uniform variable with identifier location
- **void glUniform{1|2|3|4}{f|i}vARB(GLint location, GLuint count, const TYPE* vals)**
  - Set float or integer uniform vector variable with identifier location
- **void glUniformMatrix{2|3|4|}fv(GLint location, GLuint count, GLboolean transpose, const GLfloat* vals)**
  - Set float or integer matrix uniform variable with identifier location

Vertex attributes

- **GLint glGetAttribLocation(GLint program, const GLchar* name)**
  - Returns identifier of attribute with given name in program object
- **void glVertexAttrib{1|2|3|4}{s|f|d} (GLuint index, TYPE val)**
- **void glVertexAttrib{1|2|3|4}{s|f|d}v (GLuint index, const TYPE * vals)**
  - Setting current value of vertex attribute, index is location of attribute
  - Usually used in immediate mode
- **void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid* pointer)**
  - Specifying data of generic vertex attribute with location index
  - size – number of components, must be 1, 2, 3, 4
  - type – type of each component
  - normalized – specifies whether values should be normalized or not
  - stride - byte offset between consecutive generic vertex attributes
  - pointer – pointer to data
- **void glEnableVertexAttribArray(GLuint index)**
- **void glDisableVertexAttribArray(GLuint index)**
  - Enable or disable a generic vertex attribute array with given locataion

Example

```
GLint g_vertexShader, g_fragmentShader;

g_vertexShader = glCreateShader( GL_VERTEX_SHADER );
unsigned char *vertexShaderAssembly = readShaderFile( "vertex_shader.vert" );
glShaderSource( g_vertexShader, 1, &vertexShaderAssembly, NULL );
glCompileShader( g_vertexShader);
delete vertexShaderAssembly;
Glint status;
glGetShaderiv(g_vertexShader, GL_COMPILE_STATUS,  &status);
if (status == GL_FALSE)  {
   GLchar log[1024];
   GLsizei length;
   glGetShaderInfoLog(g_vertexShader, 1024, &length, log);
   printf(log);
}

g_fragmentShader = glCreateShader( GL_FRAGMENT_SHADER );
unsigned char *fragmentShaderAssembly = readShaderFile( "fragment_shader.frag" );
glShaderSource( g_fragmentShader, 1, &fragmentShaderAssembly, NULL );
glCompileShader( g_fragmentShader );
delete fragmentShaderAssembly;
Glint status;
glGetShaderiv(g_fragmentShader, GL_COMPILE_STATUS,  &status);
if (status == GL_FALSE)  {
   GLchar log[1024];
   GLsizei length;
   glGetShaderInfoLog(g_fragmentShader, 1024, &length, log);
   printf(log);
}
```

Example

```
Glint g_programObj;

g_programObj = glCreateProgram();
glAttachShader( g_programObj, g_vertexShader );
glAttachShader( g_programObj, g_fragmentShader );

glLinkProgram( g_programObj );
GLint status;
glGetProgramiv(g_programObj, GL_LINK_STATUS, &status);
If (status == GL_FALSE)  {
   GLchar log[1024];
   GLsizei length;
   glGetProgramInfoLog(g_programObj, 1024, &length, log);
   printf(log);
}

// use prepared set of shaders
glUseProgram(g_programObj);

// use fixed function pipeline, without shaders
glUseProgram(NULL);
```

GLSL

- GL shading language
- Language for writing shaders
- Part of the core OpenGL specification
- Based on ANSI C
- Extended with mechanisms from C++ and vector and matrix types
- Used also in OpenGL ES and WebGL
- www.opengl.org/documentation/glsl/

Basic types

- void
- float vec2 vec3 vec4
- mat2 mat3 mat4
- int ivec2 ivec3 ivec4
- bool bool bvec2 bvec3 bvec4
- sampler1D, sampler2D, samplerCube, samplerShadow2D
  - Textures, cube textures, shadow textures

Type qualifiers

- Const – compile-time constant
- Attribute – for passing data for vertex
- Uniform – global variable, read-only
- Varying – data going from vertex to fragment shader, interpolated by rasterizer
- In – parameters passed into shader
- Out – passed out of shader
- Inout – in & out of function or shader

Operators

Precedence	Operator class	Operators	Associativity
1 (highest)	parenthetical grouping	( )	NA
2	array subscript function call and constructor structure field selector, swizzler post fix increment and decrement	[ ] ( ) . ++ --	Left to Right
3	prefix increment and decrement unary (tilde is reserved)	++ -- + - ~ !	Right to Left
4	multiplicative (modulus reserved)	* / %	Left to Right
5	additive	+ -	Left to Right
6	bit-wise shift (reserved)	<< >>	Left to Right
7	relational	< > <= >=	Left to Right
8	equality	== !=	Left to Right
9	bit-wise and (reserved)	&	Left to Right
10	bit-wise exclusive or (reserved)	^	Left to Right
11	bit-wise inclusive or (reserved)	|	Left to Right
12	logical and	&&	Left to Right
13	logical exclusive or	^^	Left to Right
14	logical inclusive or	||	Left to Right
15	selection	? :	Right to Left
16	assignment arithmetic assignments (modulus, shift, and bit-wise are reserved)	= += -= *= /= %= <<= >>= &= ^= |=	Right to Left
17 (lowest)	sequence	,	Left to Right

Constructors & Conversions

```
vec3(float)    // initializes each component of a vec3 with the float
vec4(ivec4)    // makes a vec4 from an ivec4, with component-wise conversion

vec2(float, float)              // initializes a vec2 with 2 floats
ivec3(int, int, int)            // initializes an ivec3 with 3 ints
bvec4(int, int, float, float)   // initializes with 4 Boolean conversions

vec2(vec3) // drops the third component of a vec3
vec3(vec4) // drops the fourth component of a vec4

vec3(vec2, float) // vec3.x = vec2.x, vec3.y = vec2.y, vec3.z = float
vec3(float, vec2) // vec3.x = float, vec3.y = vec2.x, vec3.z = vec2.y
vec4(vec3, float)
vec4(float, vec3)
vec4(vec2, vec2)
```

```
mat2(float)
mat3(float)
mat4(float)
```

```
mat2(vec2, vec2);
mat3(vec3, vec3, vec3);
mat4(vec4, vec4, vec4, vec4);

mat2(float, float,
     float, float);

mat3(float, float, float,
     float, float, float,
     float, float, float);

mat4(float, float, float, float,
     float, float, float, float,
     float, float, float, float,
     float, float, float, float);
```

int(bool) // converts a Boolean value to an int
int(float) // converts a float value to an int
float(bool) // converts a Boolean value to a float
float(int) // converts an integer value to a float
bool(float) // converts a float value to a Boolean
bool(int) // converts an integer value to a Boolean

Vector & Matrix components

- Supported component names:
  - {x,y,z,w} ; {r,g,b,a} ; {s,t,p,q}
- Allows access to multiple components
- Allows access using indexing []

```
vec4 pos  = vec4(1.0, 2.0, 3.0, 4.0);
vec4 swiz = pos.wzyx; // swiz = (4.0, 3.0, 2.0, 1.0)
vec4 dup  = pos.xxyy; // dup = (1.0, 1.0, 2.0, 2.0)
```

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
pos.xw = vec2(5.0, 6.0);      // pos = (5.0, 2.0, 3.0, 6.0)
pos.wx = vec2(7.0, 8.0);      // pos = (8.0, 2.0, 3.0, 7.0)
pos.xx = vec2(3.0, 4.0);      // illegal - 'x' used twice
pos.xy = vec3(1.0, 2.0, 3.0);// illegal - mismatch between vec2 and vec3
```

Extended operations

- Vector & Matrix operations, for easy coding
- Many component-wise operations
- *vec3 u, v, w*
  - *w = u + v* – component-wise addition
  - *w = u * v* – component-wise multiplication
- *mat3 m, n, o*
  - *o = m + o* – matrices addition
  - *o = m * o* – matrices multiplication
  - *v = m * u* – multiplication of matrix and vector

Statements

- Function definitions possible
- *void main()* must be present – starting point of each shader
- Selection: if, else
- Iteration: for, while, do
  - Arbitrary loops not recommended
- Jumps: continue, break, return, discard
  - discard discards fragment in fragment shader

Built-in variables

- For access of data from OpenGL state
- For access of user defined data
- For access of variables for input and output from shaders
- Based on fixed functionality pipeline
- Not recommended in recent versions

Vertex shader variables

- Built-in attributes
  - attribute vec4 gl_Color;
  - attribute vec4 gl_SecondaryColor;
  - attribute vec3 gl_Normal;
  - attribute vec4 gl_Vertex;
  - attribute vec4 gl_MultiTexCoord0;
  - attribute vec4 gl_MultiTexCoord1;
  - attribute vec4 gl_MultiTexCoord2;
  - attribute vec4 gl_MultiTexCoord3;
  - attribute vec4 gl_MultiTexCoord4;
  - attribute vec4 gl_MultiTexCoord5;
  - attribute vec4 gl_MultiTexCoord6;
  - attribute vec4 gl_MultiTexCoord7;
  - attribute float gl_FogCoord;
- Built-in uniforms
- Built-in varying outputs
  - varying vec4 gl_FrontColor
  - varying vec4 gl_BackColor
  - varying vec4 gl_FrontSecondaryColor
  - varying vec4 gl_BackSecondaryColor
  - varying vec4 gl_TexCoord[]
  - varying float gl_FogFragCoord
  - varying vec4 gl_Position – homogenous vertex position, must be written to, will be used by next operations on vertex
  - varying float gl_PointSize – size of point to be rasterized
  - varying vec4 gl_ClipVertex – user coordinate to be used by user clipping planes

Fragment shader variables

- Built-in varying inputs
  - varying vec4 gl_Color
  - varying vec4 gl_SecondaryColor
  - varying vec4 gl_TexCoord[]
  - varying float gl_FogFragCoord
- Built-in uniforms
- Built-in specials
  - vec4 gl_FragCoord – fragment coordinates
  - bool gl_FrontFacing – is front facing?
- Built-in outputs
  - vec4 gl_FragColor - color of fragment
  - vec4 gl_FragData[gl_MaxDrawBuffers] – colors written to multiple buffers
  - float gl_FragDepth – user defined depth of fragment

Built-in uniforms

- Matrices
  - gl_ModelViewMatrix
  - gl_ModelViewProjectionMatrix
  - gl_NormalMatrix
- Lights
  - gl_LightSource
- Materials
  - gl_FrontMaterial
  - gl_BackMaterial

Functions

- Angle and Trigonometry
  - radians, degrees, sin, cos, tan, asin, acos, …
- Exponential
  - pow, exp, log, sqrt, …
- Common
  - abs, floor, mod, min, max, clamp, mix, step, …
- Geometric
  - length, distance, dot, cross, normalize, ftransform, reflect, …
- Matrix
  - matrixCompMult
- Vector Relational
  - LessThan, graterThan, equal, any, all, …
- Texture Lookup
  - texturenD, texturenDLod, texturenDProj, textureCube, shadownD, …

Basic shaders

Vertex Shader:

```
attribute vec2 tex_coords;
varying vec2 texture_coordinate;
void main()
{
    gl_FrontColor = gl_Color;
    // passing generic attribute as generic varying variable
    texture_coordinate = tex_coords;
    // Transforming The Vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fragment Shader:

```
varying vec2 texture_coordinate;
uniform sampler2D my_color_texture;
void main()
{
    // sampling texture
    gl_FragColor = gl_Color * texture2D(my_color_texture, texture_coordinate);
}
```

Data to shaders example

```
// setting current texture in texture unit 0 and sending unit number to shader
int texture_location = glGetUniformLocation(g_programObj, "my_color_texture");
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, my_texture_object);
glUniform1i(texture_location, 0);

// use shader to render quad
// send texture coordinates to shaders as generic attributes
glUseProgram(g_programObj);
int texture_coord_location = glGetAttribLocation(g_programObj, "tex_coords");
glBegin(GL_QUADS);
   glColor3f(1.0f, 0.0f, 0.0f);
   glVertexAttrib2f(texture_ coord_ location, 0.0f, 0.0f);
   glVertex3f(-1.0f, -1.0f, 0.0f);

   glColor3f(0.0f, 1.0f, 0.0f);
   glVertexAttrib2f(texture_ coord_ location, 1.0f, 0.0f);
   glVertex3f(1.0f, -1.0f, 0.0f);

   glColor3f(0.0f, 0.0f, 1.0f);
   glVertexAttrib2f(texture_ coord_ location, 1.0f, 1.0f);
   glVertex3f(1.0f, 1.0f, 0.0f);

   glColor3f(1.0f, 1.0f, 1.0f);
   glVertexAttrib2f(texture_ coord_ location, 0.0f, 1.0f);
   glVertex3f(-1.0f, 1.0f, 0.0f);
glEnd();
```

The End!

Questions?