Vertex & Fragment shaders

- The next big step in graphics hardware
- Adds programmability to the previously fixed rendering pipeline
- The OpenGL Shading Language (a.k.a. GLSL, glslang)
  - Vertex shaders: programs for per-vertex computations
  - Fragment shaders: program for per-fragment (pixel) computations
- Now standard part of OpenGL 2.0
  - OpenGL2.0 spec defines how to manage vertex and fragment shaders
  - OpenGL Shading Language spec defines the shading language itself

Vertex & Fragment shaders (cont.)

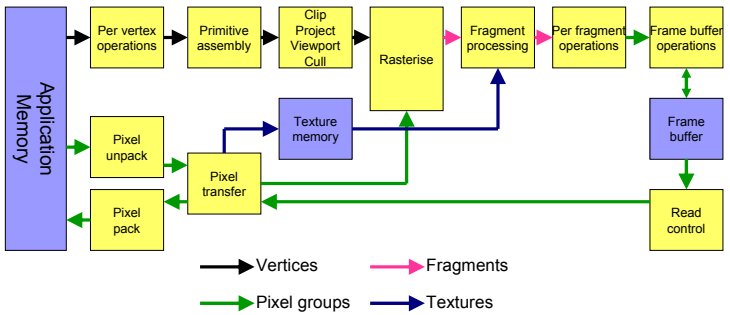
- Why?
  - Demand for more sophisticated effects
  - "If only I could add this to that on the card…"
- Why not before?
  - Computational power of GPUs is now at a level where they can execute arbitrary programs per vertex and pixel fast enough

Vertex & Fragment shaders (cont.)

- In the beginning…
  - 3D graphics drawn using the CPU only
  - Gobbles up all CPU power, leaving very little time to do anything else
  - CPUs not powerful enough
    - CPUs are good at everything, instead of excellent at one thing
  - Solution: use specialised hardware parallel to the CPU that can do 3D really well
    - Rendering pipeline is hardwired

Vertex & Fragment shaders (cont.)

Fixed OpenGL rendering pipeline (OpenGL1.5)

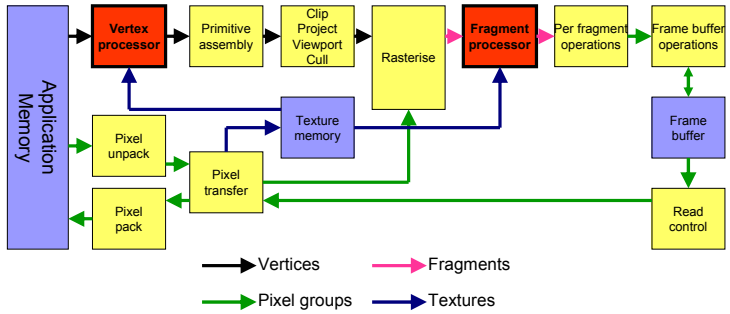


Vertex & Fragment shaders (cont.)

- Fixed pipeline geared towards textured polygons
  - Colours and normals given per-vertex
  - Lighting computed per-vertex, interpolated over polygon
- For speed, put as much on GPU as possible
  - But once there, how can we manipulate it?
    - Linear transformations through modelview matrix
    - But what about morphing or animation for example?
- What if we want to use a different lighting model?
- Various tricks to do effects like per-pixel lighting, bump mapping
  - Gfx card devs start adding extensions like NVidia register combiners
  - OpenGL starts adding extensions to fixed pipeline to make it more flexible (e.g. depth textures, texture combining)
- Keep adding functions to the fixed pipeline not a viable option
  - Takes more time to get a function in the standard than it take for people to come up with a new effect
  - Gfx card becoming more programmable to decrease hardware development time and reduce complexity
- Make the pipeline programmable

Vertex & Fragment shaders (cont.)

Programmable OpenGL rendering pipeline (OpenGL2.0)



Vertex & Fragment shaders (cont.)

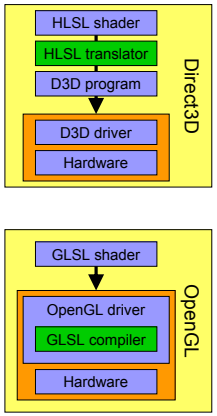
- What sort of language?
  - Low level (like assembler)
  - High level (like C/C++)
- Low level:
  - ✓ Easy to implement assembler
  - ✓ Closer to how the hardware works
    - Potentially faster hand-tuned code
    - What if tomorrow's hardware works differently?
  - × Low level optimisation only
  - × Virtually all OpenGL programming done in high level languages
- High level
  - × More difficult to implement compiler
  - ✓ Familiar to most people
  - ✓ High level optimisation
  - Further removed from hardware
    - × Relies on compiler to produce optimal hardware-specific code
    - ✓ More future-proof

Vertex & Fragment shaders (cont.)

- Efforts began around 2001
- Vertex and fragment *program* extensions
  - Low level
  - Based on assembler
- Vertex and fragment *shader* extensions
  - High level
  - Based on C/C++
- NVidia Cg language
  - High level
  - Based on C/C++
- Meanwhile Microsoft Direct3D gets HLSL high level shading language
- 2004: Vertex and fragment shader extensions promoted to core OpenGL
  - Called GLSL, or glslang

Vertex & Fragment shaders (cont.)

- GLSL is high level, and requires a compiler
- Where to put the compiler?
  1. Provide a stand-alone compiler outside of OpenGL driver which compiles to some standard assembler, object, or binary format (done in Direct3D)
     - Creates a market of shader compilers
     - Need to define yet another standard
     - Compiler would need to know how to best optimise for each gfx card
  2. Include compiler in driver which compiles to some undefined internal hardware-specific format
     - Can compile from high level directly to hardware
     - Compiler optimised for use with cards supported by the driver
     - Each gfx card manufacturer needs to make their own compiler
- Decided on 2nd method: compiler in the driver



Vertex & Fragment shaders (cont.)

- An example vertex shader

```
uniform vec3 LightPosition;
const float SpecularContribution = 0.3;
const float DiffuseContribution = 1.0 - SpecularContribution;
varying float LightIntensity;
varying vec2 MCposition;

void main(void)
{
  vec3 ecPosition = vec3 (gl_ModelViewMatrix * gl_Vertex);
  vec3 tnorm      = normalize(gl_NormalMatrix * gl_Normal);
  vec3 lightVec   = normalize(LightPosition - ecPosition);
  vec3 reflectVec = reflect(-lightVec, tnorm);
  vec3 viewVec    = normalize(-ecPosition);
  float diffuse   = max(dot(lightVec, tnorm), 0.0);
  float spec      = 0.0;
  if (diffuse > 0.0)
  {
    spec = max(dot(reflectVec, viewVec), 0.0);
    spec = pow(spec, 16.0);
  }
  LightIntensity  = DiffuseContribution * diffuse + SpecularContribution * spec;
  MCposition      = gl_Vertex.xy;
  gl_Position     = ftransform();
}
```


Vertex & Fragment shaders (cont.)

- An example fragment shader

```
uniform vec3 BrickColor, MortarColor;
uniform vec2 BrickSize;
uniform vec2 BrickPct;
varying vec2 MCposition;
varying float LightIntensity;

void main(void)
{
  vec3 color;
  vec2 position, useBrick;
  position = MCposition / BrickSize;
  if (fract(position.y * 0.5) > 0.5)
    position.x += 0.5;

  position = fract(position);
  useBrick = step(position, BrickPct);
  color    = mix(MortarColor, BrickColor, useBrick.x * useBrick.y);
  color   *= LightIntensity;
  gl_FragColor = vec4 (color, 1.0);
}
```


Vertex & Fragment shaders (cont.)

- Vertex shader
  - Executed for every vertex while active
    - Explicit `glVertex()` call
    - Implicit vertex calls, e.g. from vertex arrays
  - Must set the camera-relative position of each vertex
  - May set any per-vertex information
- Fragment shader
  - Executed for every fragment (pixel) being drawn while active
  - May set a colour and depth for fragment
  - May discard the fragment

Vertex & Fragment shaders (cont.)

- Vertex shader replaces fixed functionality such as:
  - Vertex transformation (modelview and projection)
  - Transformation and normalisation of normals
  - Texture coordinate generation
  - Texture coordinate transformation
  - Lighting
  - Color material application
- Vertex shaders limited to single vertex operations
- Vertex shaders do not replace:
  - Perspective divisions
  - Viewport mapping
  - Primitive assembly
  - Clipping
  - Backface culling
  - …and more things that depend on knowledge of more than one vertex at a time
- Vertex shaders can not access other vertices

Vertex & Fragment shaders (cont.)

- Fragment shader replaces fixed functionality such as:
  - Operations on interpolated values
  - Texture access
  - Texture application
  - Fog
- Fragment shaders do not replace such things as:
  - Shading model
  - Depth test
  - Alpha blending
  - Window clipping
- Note that many of the things not replaced by fragment shaders can be disabled
- Fragment shaders can not change the position of a fragment, or access other fragments

Vertex & Fragment shaders (cont.)

- Vertex and fragment shaders output values to the remaining fixed pipeline through special variables
- Vertex shader must set the position of the vertex in the `gl_Position` special variable
  - Vertex shader can read from `gl_Position` only after setting it
- Vertex shader may also set special variables:
  - `float gl_PointSize`: size of point primitives
  - `vec4 gl_ClipVertex`: position to use when determining clipping of vertex against user-defined clip planes
- Fragment shader writes to special variables:
  - `vec4 gl_FragColor`: colour of fragment
  - `float gl_FragDepth`: depthbuffer value of fragment
- Fragment shader has access to special read-only variables:
  - `vec4 gl_FragCoord`: window coordinates of fragment (including depth value as computed by fixed pipeline)
  - `bool gl_FrontFacing`: set to true if fragment belongs to front-facing primitive

Vertex & Fragment shaders (cont.)

- Types

void	for functions that do not return a value
bool	a conditional type, taking on values of true or false
int	a signed integer
float	a single floating-point scalar
vec2	a two component floating-point vector
vec3	a three component floating-point vector
vec4	a four component floating-point vector
bvec2	a two component Boolean vector
bvec3	a three component Boolean vector
bvec4	a four component Boolean vector
ivec2	a two component integer vector
ivec3	a three component integer vector
ivec4	a four component integer vector
mat2	a 2×2 floating-point matrix
mat3	a 3×3 floating-point matrix
mat4	a 4×4 floating-point matrix
sampler1D	a handle for accessing a 1D texture
sampler2D	a handle for accessing a 2D texture
sampler3D	a handle for accessing a 3D texture
samplerCube	a handle for accessing a cube mapped texture
sampler1DShadow	a handle for accessing a 1D depth texture with comparison
sampler2DShadow	a handle for accessing a 2D depth texture with comparison

Vertex & Fragment shaders (cont.)

- Booleans
  - true or false, used for conditionals
- Integers
  - Rarely used, as most computations are done using floats
  - Mostly meant for loop counts, array indexing
  - 16 bit signed integers
  - Result of overflow is undefined
- Floats
  - Single precision (32 bits) IEEE floating point values
- Vectors
  - A basic type, not an array or struct
- Matrices
  - A basic type, not an array or struct
  - Column major order
- Samplers
  - Black box handle for accessing textures

Vertex & Fragment shaders (cont.)

- Vector components
  - Components of a vector accessed like fields of a struct
    - vec3 v;
    - float xpos = v.x;
    - float zpos = v.z;
  - Vector used for position, colour, and texture coordinates
  - Can use different component names to suit usage:
    - vec3 col;
    - float red = col.r;
    - vec2 texcoord;
    - float s = texcoord.s;

$\{x, y, z, w\}$	useful when accessing vectors that represent points or normals
$\{r, g, b, a\}$	useful when accessing vectors that represent colors
$\{s, t, p, q\}$	useful when accessing vectors that represent texture coordinates

Vertex & Fragment shaders (cont.)

- Can create a new vector by using multiple components:
  - vec4 col;
  - vec4 othercol = col.rgba; // same as col
  - vec3 othercol = col.rgb;
  - vec2 othercol = col.xz;
- Components can be “swizzled”:
  - vec4 othercol = col.abgr;
  - vec2 othercol = col.rr;
- Can assign to selected components:
  - othercol.r = 1;
  - othercol.ar = vec2(0.7, 0.4);
  - otherpos.xy = pos.zz;
- Can use indexing to get a component:
  - col[2] // same as col.b

Vertex & Fragment shaders (cont.)

- Matrix
  - Matrix components accessed using array indices
    - mat4 m;
    - m[2][3] = 2.0; // 4th element of 3rd column
  - A column is a vector
    - vec4 v = m[1]; // sets v to 2nd column
- Structures
  - Like with C++
- Arrays
  - Only 1D arrays are supported
  - Array size has to be known at compile-time
    - Size specified by constants or expressions of constants
  - Array size can be set after declaration:

```
light lights[];
const int numLights = 2;
light lights[numLights];
```

  - Arrays indexed from 0
- Bits?
  - There is no concept of bits making up a basic type
  - No bit manipulation operations

Vertex & Fragment shaders (cont.)

- Constructors
  - Used to convert between types, create a value for a larger type from smaller types, or reduce a larger type to a smaller type
  - Convert between scalar types:
    - bool b = true;
    - int a = int(b); // int(true) == 1
    - a = int(2.3);
  - Convert to a scalar type (picks first component)
    - vec3 v;
    - float f = float(v); // assigns v.x to f
    - int a = int(v);
  - Vector constructors
    - vec2 v = vec3(1.2); // v == (1.2, 1.2)
    - v = vec2(1.2, 3.4);
    - vec3 w = vec3(v, 5.6); // w = (1.2, 3.4, 5.6)
    - vec2 q = vec2(w); // drops last components
  - Matrix constructors
    - mat3 m = mat3(2.3); // Sets diagonal components to 2.3, rest to 0
    - vec3 v,w,q;
    - m = mat3(v, w, q);

Vertex & Fragment shaders (cont.)

- Structure constructors

```
struct light {
    float intensity;
    vec3 position;
};
light lightVar = light(3.0, vec3(1.0, 2.0, 3.0));
```


Vertex & Fragment shaders (cont.)

- Type qualifiers

<none>	local read/write memory, or an input parameter to a function
const	a compile-time constant, or a function parameter that is read-only
attribute	linkage between a vertex shader and OpenGL for per-vertex data
uniform	value does not change across the primitive being processed, uniforms form the linkage between a shader, OpenGL, and the application
varying	linkage between a vertex shader and a fragment shader for interpolated data
in	for function parameters passed into a function
out	for function parameters passed back out of a function, but not initialized for use when passed in
inout	for function parameters passed both into and out of a function

Vertex & Fragment shaders (cont.)

- Attribute
  - Value that is associated with a vertex
  - Set per-vertex by an OpenGL program
  - Vertex shader only
  - Read-only in a shader
  - Attributes are global in scope in a shader
  - Standard pre-defined vertex attributes
    - `attribute vec4 gl_Color`
    - `attribute vec3 gl_Normal`
    - `attribute vec4 gl_Vertex`
    - `attrribute vec4 gl_MultiTexCoord0, glMultiTexCoord1, …`
    - etc.
  - Attributes are defined in a vertex shader, but given a value by an OpenGL program

Vertex & Fragment shaders (cont.)

- Uniform
  - Per-primitive constant value
  - Give the OpenGL state at the time the primitive is being processed
  - Read-only, global
  - Available in vertex and fragment shaders
  - Defined in a shader, but given a value by an OpenGL program
  - Some pre-defined uniforms:
    - `uniform mat4 gl_ModelViewMatrix`
    - `uniform mat4 gl_ProjectionMatrixInverse`
    - `uniform mat4 glTextureMatrixTranspose[gl_MaxTextureCoords]`
    - `uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights]`

Vertex & Fragment shaders (cont.)

- Varying
  - Values that are interpolated over the primitive
  - Vertex shader computes the values per-vertex
  - OpenGL interpolates the values over the primitive
    - Interpolation is perspective-corrected
  - Fragment shader gets interpolated value for fragment
  - Global, read/write in vertex shader, read-only in fragment shader
  - If a varying variable is used by a fragment shader (or the fixed pipeline if no fragment shader) then the vertex shader must set a value
  - Some pre-defined varying variables that can be written to in a vertex shader:
    - `varying vec4 gl_FrontColor`
    - `varying vec4 gl_TexCoord[]`
  - Some pre-define varying variables that can be read from in a fragment shader:
    - `varying vec4 gl_Color`
    - `varying vec4 gl_TexCoord[]`

Vertex & Fragment shaders (cont.)

- In, out, inout
  - Defines which parameters of a function are input parameters, output parameters, or both
  - Value of out parameter is undefined until assigned a value by the function
  - Function parameters passed by value
    - All computations in a function are done on copies, not affecting the original variables
  - Value of out and inout parameters copied back out to original variables when function returns

Vertex & Fragment shaders (cont.)

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

Vertex & Fragment shaders (cont.)

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

Vertex & Fragment shaders (cont.)

- Add, subtract, multiply, divide:
  - If one operand is a scalar, other a vector or matrix: scalar applied to each component
  - With two vectors: component-wise operations
  - Multiply of two matrices: matrix multiplication
  - Matrix with vector of same size: matrix-vector multiplication
- Equality: vector, matrices, structures are equal if their components/fields are the same
- Comparisons other than equality: only defined for scalars

Vertex & Fragment shaders (cont.)

- Have the usual statements and expressions
  - Selection

```
if (bool-expression)
    true-statement

if (bool-expression)
    true-statement
else
    false-statement
```

  - Iteration

```
for (init-expression; condition-expression; loop-expression)
    sub-statement

while (condition-expression)
    sub-statement

do
    statement
while (condition-expression)
```


Vertex & Fragment shaders (cont.)

- Jumps
  - `continue;`
  - `break;`
  - `return;`
  - `return expression;`
  - `discard;` `// fragment shaders only`
- Discard only used in fragment shaders to discard the current fragment
  - Processing stops on discarded fragments
  - Discarded fragments are never rendered

Vertex & Fragment shaders (cont.)

- Functions
  - Can define functions
    - Parameters qualified with `in, out, or inout`
      - If unqualified, default is `in`
    - Parameters can be qualified as being const
    - Parameter may be array or structure
    - If no parameters, parameter list either empty or void
    - Functions can return a value of any type except array
    - If no return value, return type is void
    - Function names can be overloaded, as long as the parameter types are different
  - Each shader must have a function called main with no parameters and no return:
    - `void main()`
    - `void main(void)`
  - When calling a function, values passed to out or inout parameters must be assignable ("lvalues")
  - Recursion is not allowed, and results in undefined behaviour

Vertex & Fragment shaders (cont.)

- Built-in functions
  - Lots, check the GLSL spec
  - Trig: `radians, degrees, sin, cos, tan, asin, acos, atan`
  - Exponents: `pow, exp, log, exp2, log2, sqrt, inversesqrt`
  - Math: `abs, sign, floor, ceil, fract, mod, min, max, clamp, mix, step, smoothstep`
  - Geometric: `length, distance, dot, cross, normalize, ftransform, faceforward, reflect, refract`
  - Matrix: `matrixCompMult`
  - Vector relational: `lessThan, lessThanEqual, greaterThan, greaterThanEqual, equal, notEqual, any, all, not`
  - Texture lookup: `texture1D, texture1DProj, texture1DLod, texture1DProjLod, texture2D, …, textureCube, textureCubeLod, shadow1D, …`
  - Fragment processing: `dFdx, dFdy, fwidth`
  - Random noise: `noise1, noise2, noise3, noise4`

Vertex & Fragment shaders (cont.)

- Some of note:
  - `vec4 ftransform()`
    - Vertex shader only
    - Transforms vertex exactly like the fixed functionality
  - `mat matrixCompMult(mat x, mat y)`
    - Component-wise multiplication of matrices
  - `bvec lessThan(vec x, vec y), greaterThanEqual()`, etc.
    - Component-wise comparisons of vectors
  - `vec4 texture2D(sampler2D sampler, vec2 coord)`, etc.
    - Sample a texture
    - Some differences between usage in vertex and fragment shaders
  - Noise functions
    - Fragment shader only
    - Return a random noise value in the range [-1,1]
    - Same input gives same random number
    - No specific implementation defined, but is likely to be something like Perlin noise
  - Derivatives
    - Fragment shader only
    - Computes numerical derivatives, possibly approximately only
    - No specific implementation defined, only some properties
    - genType dFdx(genType p), dFdy()
      - Computes derivative in x or y respectively for input
      - Rough idea: values of p for neighbouring fragments are used to compute a slope
      - Higher order derivatives like dFdx(dFdx(p)) or dFdx(dFdy(p)) undefined

Vertex & Fragment shaders (cont.)

- Preprocessor
  - Very similar to C preprocessor
    - `#define`, `#if`, etc.
    - No file-based directives like `#include`
    - `#version` *`number`*
      - Tell compiler what language version shader is written in
      - Current shader language version is 110
    - `#extension` *`extension_name`* `:` *`behaviour`*
      - Tells the compiler how to behave when encountering a language extension (enable, warn, disable, etc.)
    - `#pragma optimize(on)` `#pragma optimize(off)`
      - Turn on and off optimisation
    - `//` and `/* … */` are comments

Vertex & Fragment shaders (cont.)

- Using shaders in an OpenGL program
  - Shaders and their state are encapsulated in an OpenGL object (similar to a texture object)
  - Shaders are defined as an array of strings
  - Basic steps for using a shader:
    - Send shader source to OpenGL
    - Compile the shader
    - Create an "executable" by linking compiled shaders
    - Install the executable as part of the current OpenGL state
  - Shaders can be compiled at any time, executables created at any time

Vertex & Fragment shaders (cont.)

- Create a shader object:
  - `GLuint shader = glCreateShader(GLenum type);`
    - type is `GL_VERTEX_SHADER` or `GL_FRAGMENT_SHADER`
- Set source code for shader:
  - `glShaderSource(GLuint shader, GLsizei count, const char **strings, const int *lengths);`
    - Adds an array of strings
    - lengths is array containing the length of each string; may be NULL to indicate that all strings are null-terminated
- Compile shader:
  - `glCompileShader(GLuint shader);`
- Delete a shader:
  - `glDeleteShader(GLuint shader);`
    - Deletes shader as soon as it is no longer needed

Vertex & Fragment shaders (cont.)

- Putting shaders together in an executable program
  - Create a program object:
    - `GLuint program = glCreateProgram();`
  - Attach shaders to the program:
    - `glAttachShader(GLuint program, GLuint shader);`
  - Link the attached shaders together:
    - `glLinkProgram(GLuint program);`
  - Start using a linked program:
    - `glUseProgram(GLuint program);`
- A program that is in use can be modified
  - Modifications will only come into effect when program is relinked
- To remove program and return to fixed pipeline:
    - `glUseProgram(0)`
- Shader may be put in multiple programs
- Detaching a shader from a program:
    - `glDetachShader(GLuint program, GLuint shader);`

Vertex & Fragment shaders (cont.)

- Program may contain only a vertex shader, fragment shader, or both
- Program may contain multiple vertex and fragment shaders [which will be used???]
- When linking a vertex shader and fragment shader together, they must match
  - Any varying variables used by the fragment shader must be provided by the vertex shader
- Various functions to get info about a program, like:
  - `glGetProgramiv(GLuint program, GLenum pname, T params);`

Vertex & Fragment shaders (cont.)

- Setting attributes
  - Attributes are defined per-vertex
    - `glVertexAttrib*(GLuint index, T values)`
    - `glVertexAttribPointer(GLuint index, int size, GLenum type, GLboolean` normalised, GLsizei stride, const void *pointer);
  - Attributes are referenced by an index
  - Before use, must have the index of an attribute
    - Associate an index with an attribute explicitly before linking:
      - `glBindAttribLocation(GLuint program, GLuint index, const char *name);`
    - Or if not bound before linking, an index will be assigned by OpenGL; get index using:
      - `glGetAttribLocation(GLuint program, const char *name);`
  - To get info about an attribute in use by a program:
    - `glGetActiveAttrib(GLuint program, GLuint index, GLsizei bufSize, GLsizei *length, int *size, GLenum *type, char *name);`
  - Attribute index 0 is always gl_Vertex

Vertex & Fragment shaders (cont.)

- Setting uniforms
  - Similar to attributes
  - Not set per-vertex, but per-primitive (outside `glBegin()...glEnd()`)
  - To set a uniform value:
    - `glUniform*(int location, T value);`
  - Get the location of a uniform:
    - `glGetUniformLocation(GLuint program, const char *name);`
  - Get info about a uniform that is in use by a program:
    - `glGetActiveUniform();`
- Setting texture samplers
  - Texture samplers are uniforms
  - Bind a texture and make it active as usual (`glBindTexture()`, `glActiveTexture()`)
  - Get the location of the sampler with `glGetUniformLocation()`
  - Set the sampler to the texture unit with `glUniform1i()`

Vertex & Fragment shaders (cont.)

- Getting info from the compiler and linker
  - After compiling:
    - `glGetShaderInfoLog(GLuint shader, GLsizei bufSize, GLsizei *length, char *infoLog);`
    - Returns errors, warnings, etc. from the last compile
    - Get needed buffer size from `glGetShaderiv();`
  - After linking:
    - `glGetProgramInfoLog()`
    - Use `glGetProgramiv()` to get needed buffer size