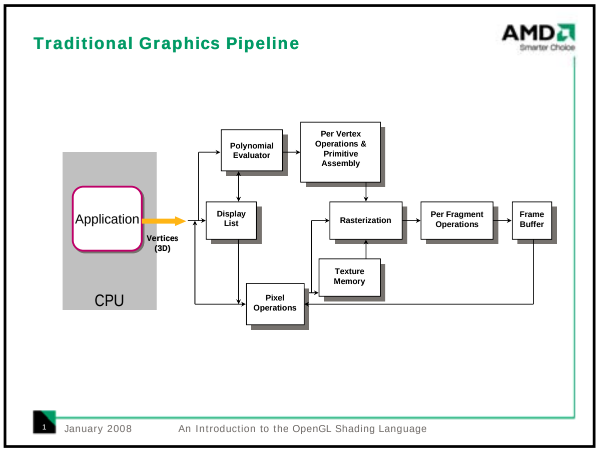
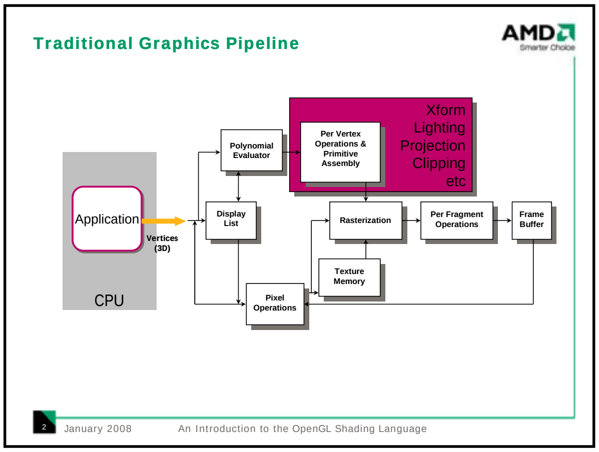


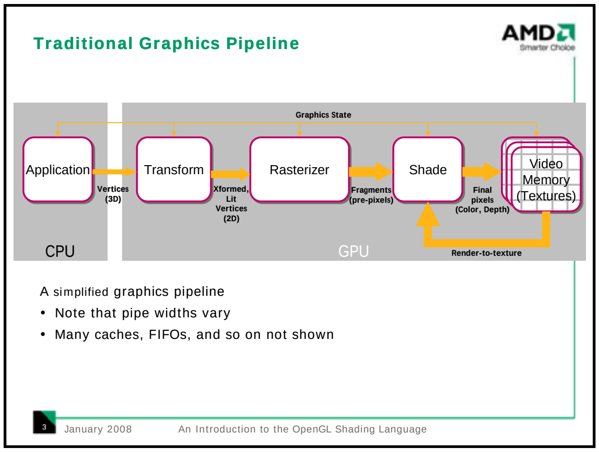
Traditional Graphics Pipeline

1 January 2008 An Introduction to the OpenGL Shading Language



Traditional Graphics Pipeline

2 January 2008 An Introduction to the OpenGL Shading Language

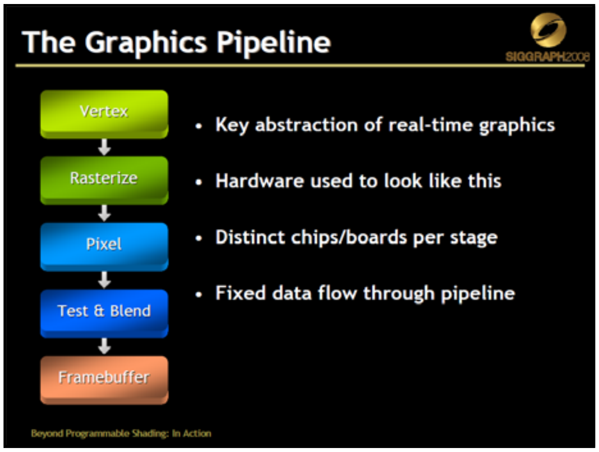


Traditional Graphics Pipeline

A simplified graphics pipeline

- Note that pipe widths vary
- Many caches, FIFOs, and so on not shown

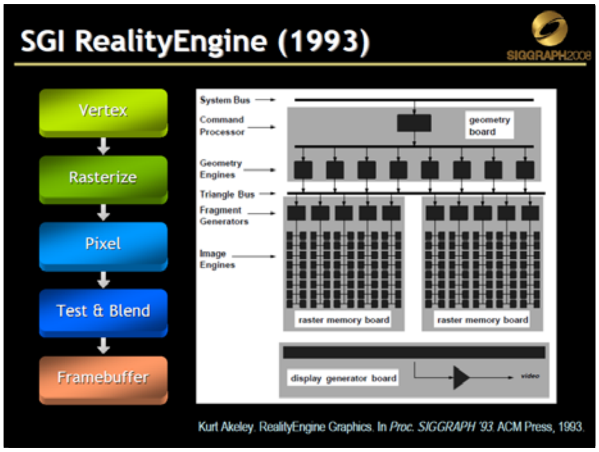
3 January 2008 An Introduction to the OpenGL Shading Language



The Graphics Pipeline

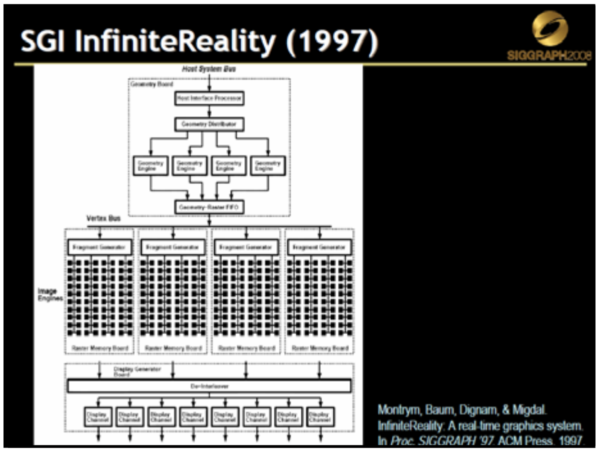
- Key abstraction of real-time graphics
- Hardware used to look like this
- Distinct chips/boards per stage
- Fixed data flow through pipeline

Beyond Programmable Shading: In Action



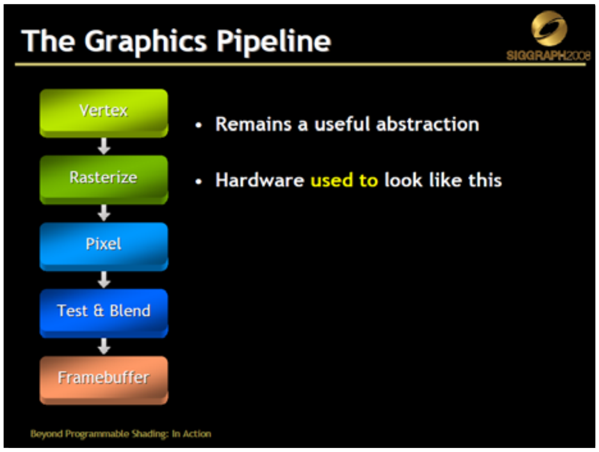
SGI RealityEngine (1993)

Kurt Akeley. RealityEngine Graphics. In *Proc. SIGGRAPH '93*. ACM Press, 1993.



SGI InfiniteReality (1997)

Montrym, Baum, Dignam, & Migdal. InfiniteReality: A real-time graphics system. In *Proc. SIGGRAPH '97*. ACM Press. 1997.



The Graphics Pipeline

SIGGRAPH2008

Vertex

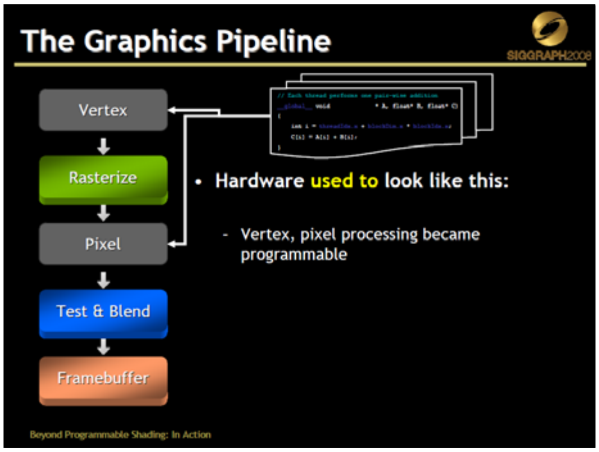
Rasterize

Pixel

Test & Blend

Framebuffer

- Remains a useful abstraction
- Hardware *used to* look like this



The Graphics Pipeline

SIGGRAPH2008

Vertex

Rasterize

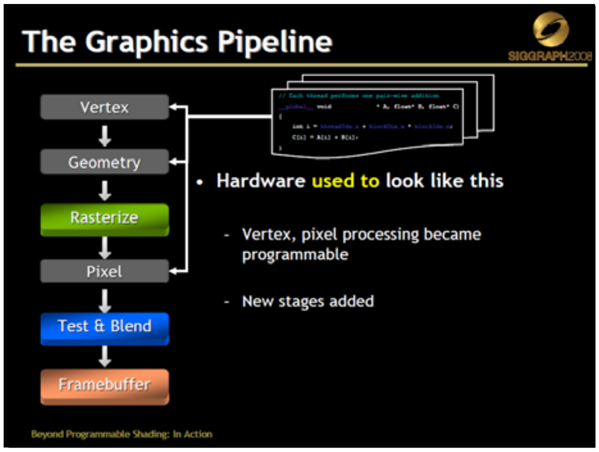
Pixel

Test & Blend

Framebuffer

```
// Each thread performs one pair-wise addition
__global__ void              * A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

- Hardware *used to* look like this:
  - Vertex, pixel processing became programmable



The Graphics Pipeline

SIGGRAPH2008

Vertex

Geometry

Rasterize

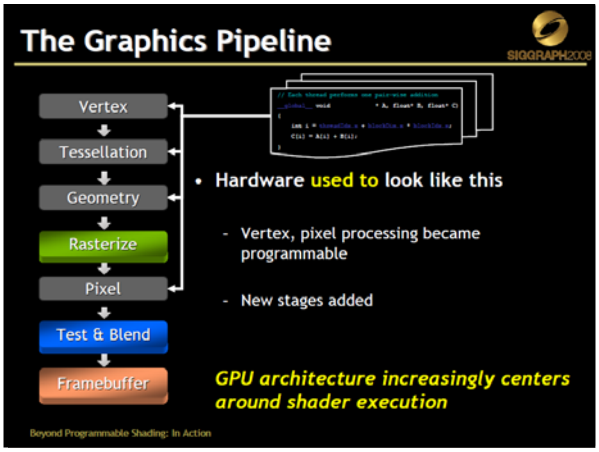
Pixel

Test & Blend

Framebuffer

```
// Each thread performs one pair-wise addition
__global__ void              * A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

- Hardware *used to* look like this
  - Vertex, pixel processing became programmable
  - New stages added



The Graphics Pipeline

SIGGRAPH2008

Vertex

Tessellation

Geometry

Rasterize

Pixel

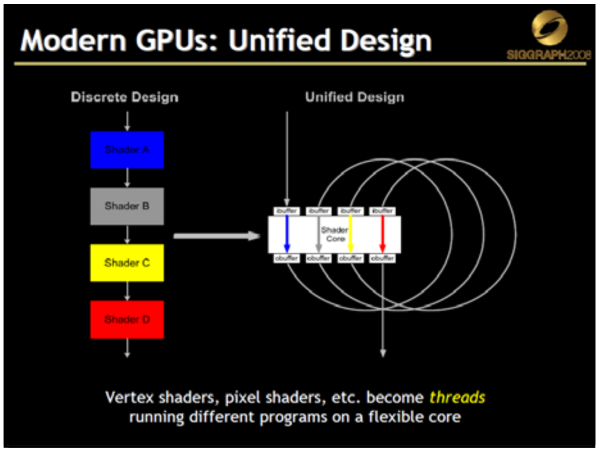
Test & Blend

Framebuffer

```
// Each thread performs one pair-wise addition
__global__ void              * A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

- Hardware *used to* look like this
  - Vertex, pixel processing became programmable
  - New stages added

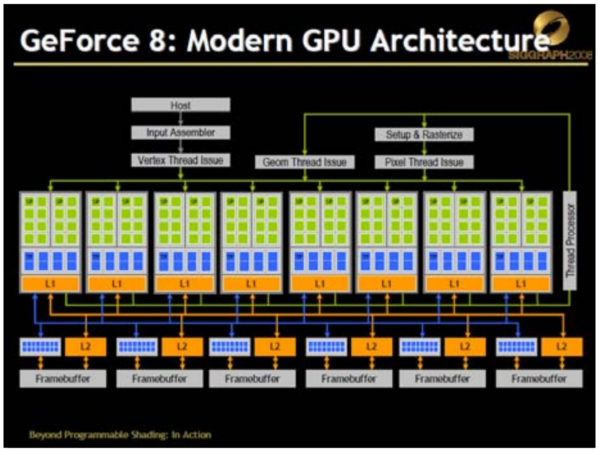
GPU architecture increasingly centers around shader execution



Modern GPUs: Unified Design

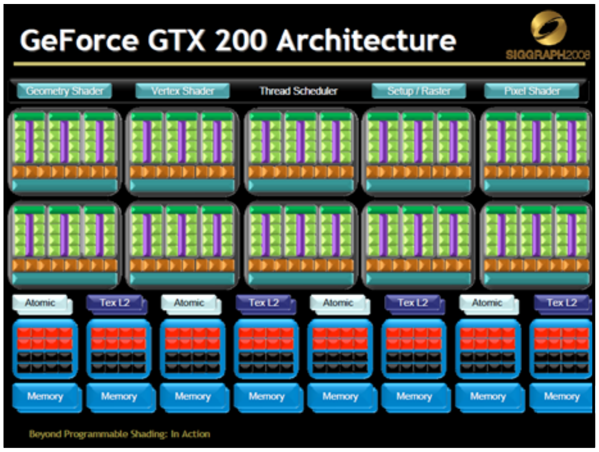
SIGGRAPH2008

Vertex shaders, pixel shaders, etc. become *threads* running different programs on a flexible core



GeForce 8: Modern GPU Architecture

SIGGRAPH2008



GeForce GTX 200 Architecture

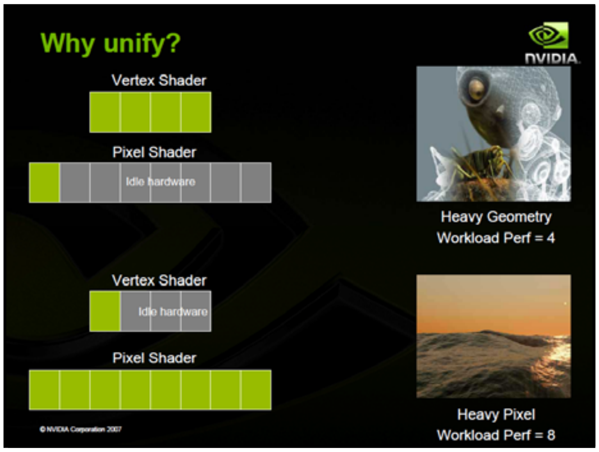
SIGGRAPH2008

Geometry Shader Vertex Shader Thread Scheduler Setup / Raster Pixel Shader

Atomic Tex L2 Atomic Tex L2 Atomic Tex L2 Atomic Tex L2

Memory Memory Memory Memory Memory Memory Memory Memory

Beyond Programmable Shading: In Action



Why unify?

Vertex Shader

Pixel Shader

Idle hardware

Heavy Geometry
Workload Perf = 4

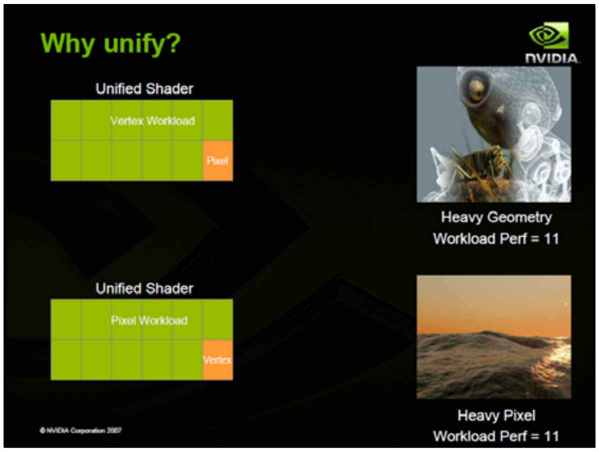
Vertex Shader

Idle hardware

Pixel Shader

Heavy Pixel
Workload Perf = 8

© NVIDIA Corporation 2007



Why unify?

Unified Shader

Vertex Workload

Pixel

Heavy Geometry
Workload Perf = 11

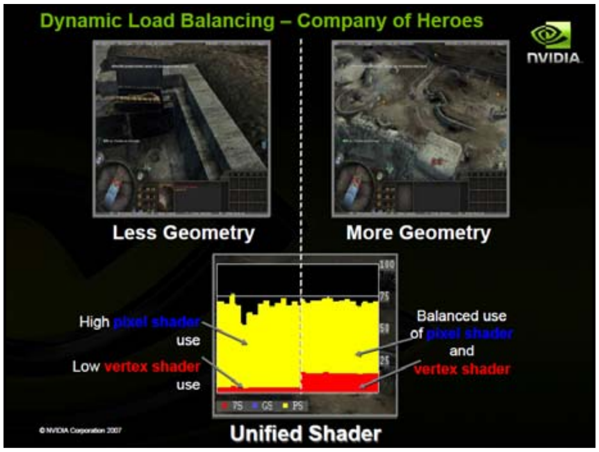
Unified Shader

Pixel Workload

Vertex

Heavy Pixel
Workload Perf = 11

© NVIDIA Corporation 2007



Dynamic Load Balancing – Company of Heroes

Less Geometry

More Geometry

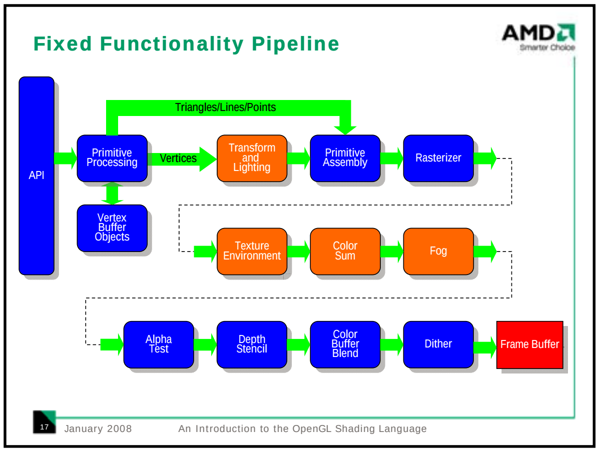
High pixel shader use

Low vertex shader use

Balanced use of pixel shader and vertex shader

Unified Shader

© NVIDIA Corporation 2007



Fixed Functionality Pipeline

API

Triangles/Lines/Points

Primitive Processing

Vertices

Transform and Lighting

Primitive Assembly

Rasterizer

Vertex Buffer Objects

Texture Environment

Color Sum

Fog

Alpha Test

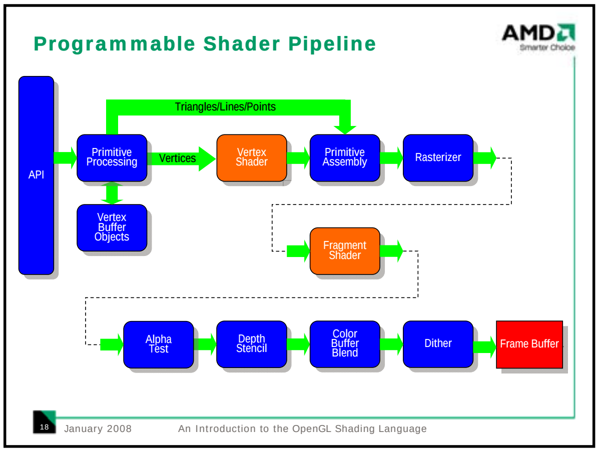
Depth Stencil

Color Buffer Blend

Dither

Frame Buffer

17 January 2008 An Introduction to the OpenGL Shading Language



Programmable Shader Pipeline

API

Triangles/Lines/Points

Primitive Processing

Vertices

Vertex Shader

Primitive Assembly

Rasterizer

Vertex Buffer Objects

Fragment Shader

Alpha Test

Depth Stencil

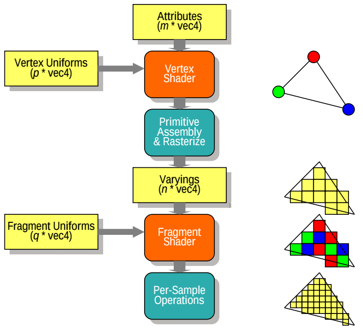
Color Buffer Blend

Dither

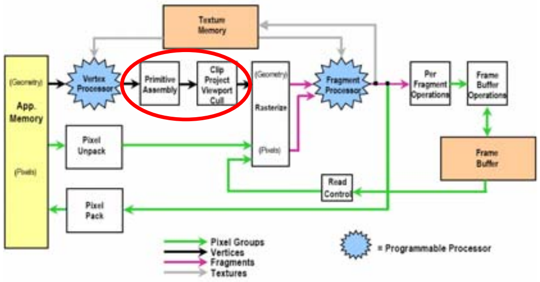
Frame Buffer

18 January 2008 An Introduction to the OpenGL Shading Language

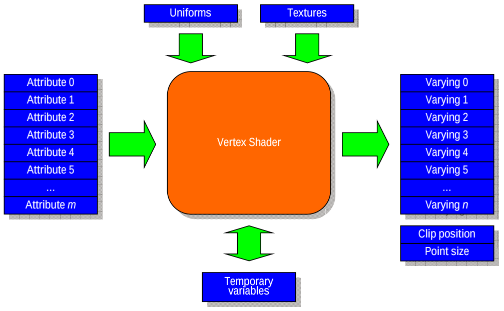
Programmer's Model



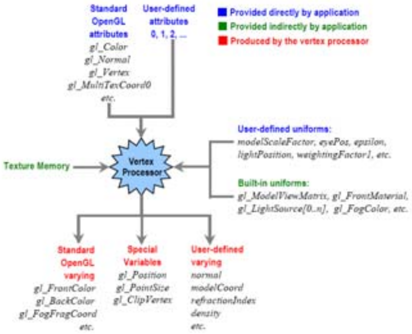
OpenGL 2.0 Logical Diagram



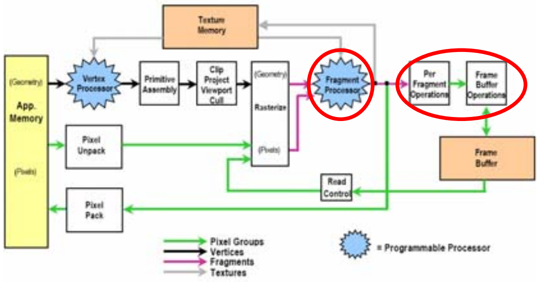
Vertex Shader Environment



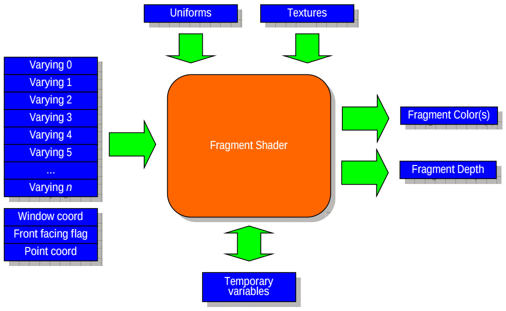
Vertex Processor Overview

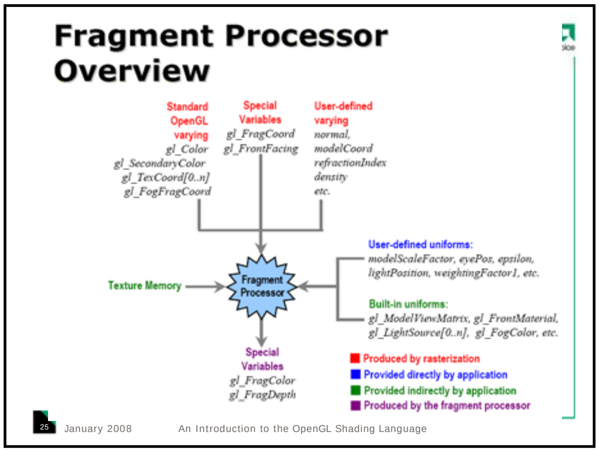


OpenGL 2.0 Logical Diagram



Fragment Shader Environment





Fragment Processor Overview

Standard OpenGL varying
gl_Color
gl_SecondaryColor
gl_TexCoord[0..n]
gl_FogFragCoord

Special Variables
gl_FragCoord
gl_FrontFacing

User-defined varying
normal,
modelCoord
refractionIndex
density
etc.

Texture Memory

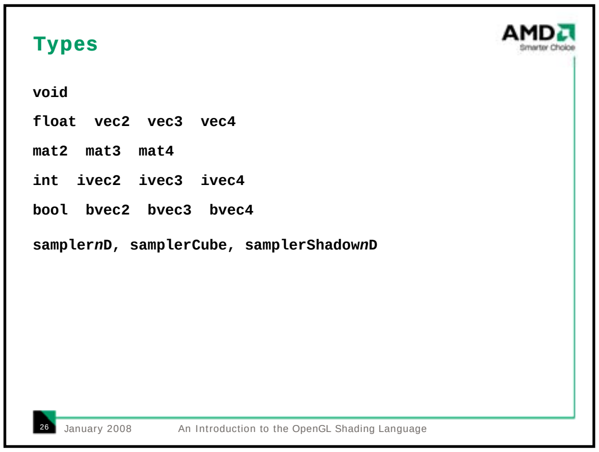
Fragment Processor

User-defined uniforms:
modelScaleFactor, eyePos, epsilon, lightPosition, weightingFactor1, etc.

Built-in uniforms:
gl_ModelViewMatrix, gl_FrontMaterial, gl_LightSource[0..n], gl_FogColor, etc.

Special Variables
gl_FragColor
gl_FragDepth

🟥 Produced by rasterization
🟦 Provided directly by application
🟩 Provided indirectly by application
🟪 Produced by the fragment processor



Types

`void`

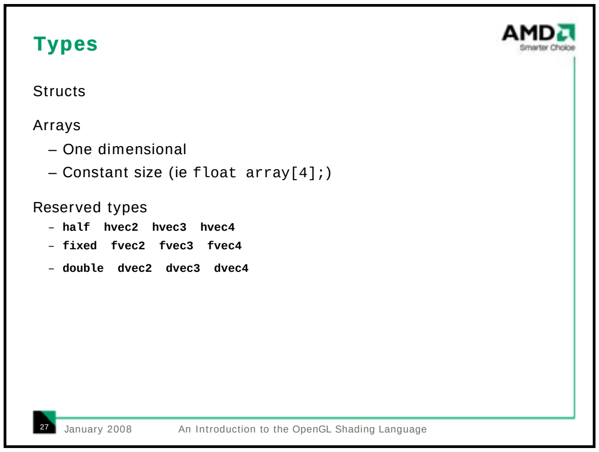
`float  vec2  vec3  vec4`

`mat2  mat3  mat4`

`int  ivec2  ivec3  ivec4`

`bool  bvec2  bvec3  bvec4`

sampler*n*D, samplerCube, samplerShadow*n*D

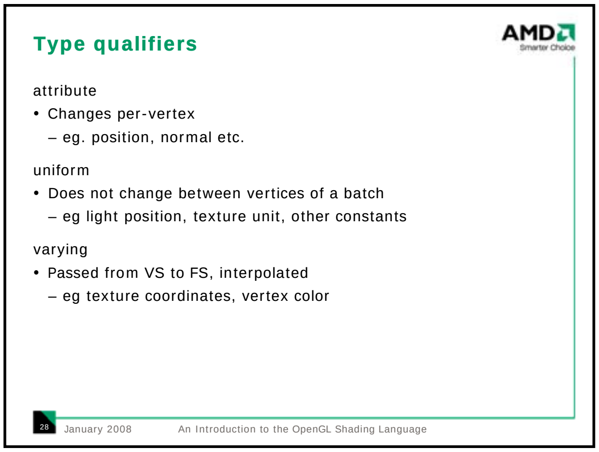


Types

Structs

Arrays
- One dimensional
- Constant size (ie `float array[4];`)

Reserved types
- `half  hvec2  hvec3  hvec4`
- `fixed  fvec2  fvec3  fvec4`
- `double  dvec2  dvec3  dvec4`

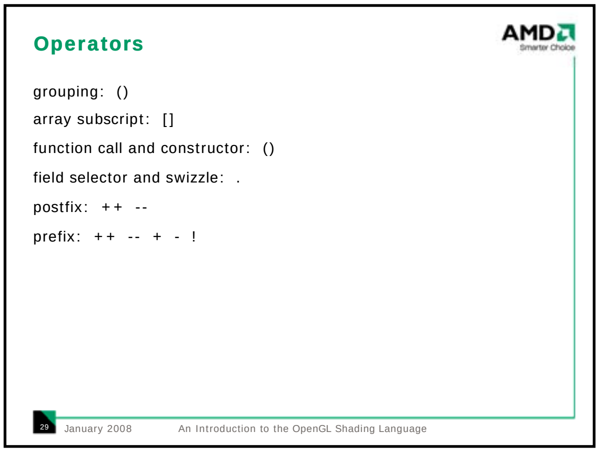


Type qualifiers

attribute
- Changes per-vertex
  - eg. position, normal etc.

uniform
- Does not change between vertices of a batch
  - eg light position, texture unit, other constants

varying
- Passed from VS to FS, interpolated
  - eg texture coordinates, vertex color



Operators

grouping: ()

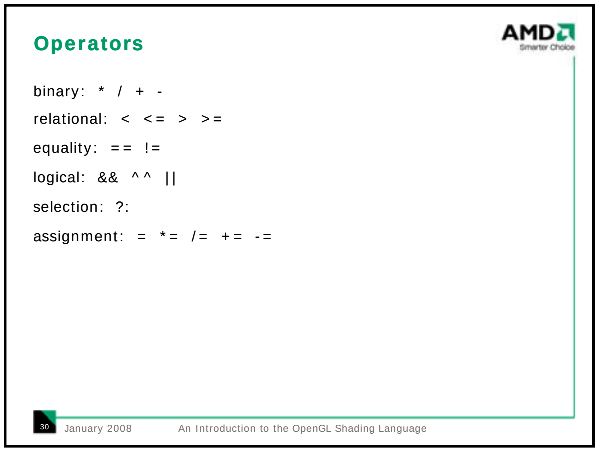
array subscript: []

function call and constructor: ()

field selector and swizzle: .

postfix: ++ --

prefix: ++ -- + - !



Operators

binary: * / + -

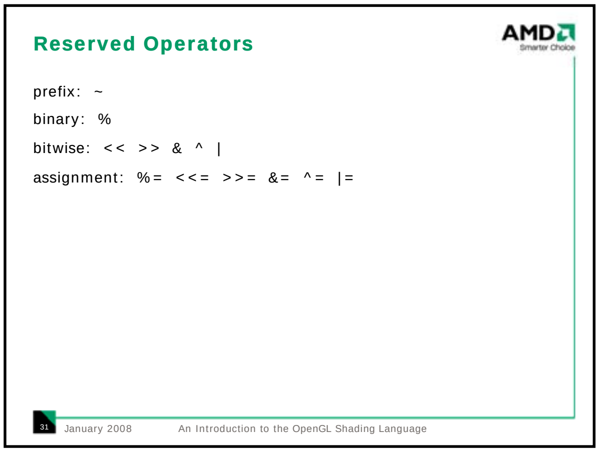
relational: < <= > >=

equality: == !=

logical: && ^^ ||

selection: ?:

assignment: = *= /= += -=



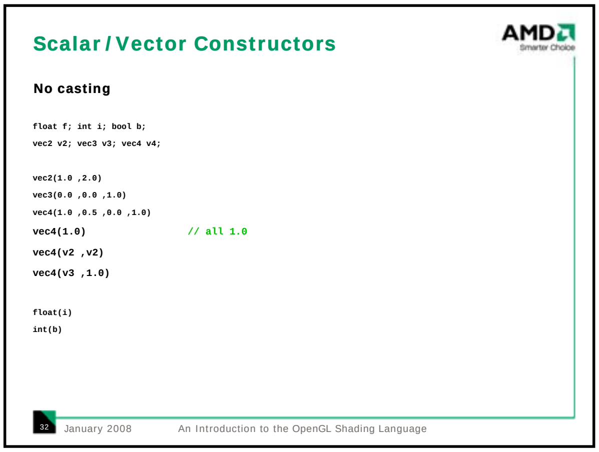
Reserved Operators

prefix: ~

binary: %

bitwise: << >> & ^ |

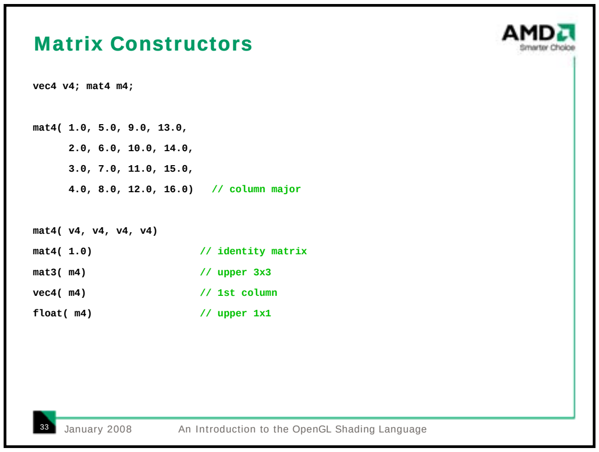
assignment: %= <<= >>= &= ^= |=



Scalar / Vector Constructors

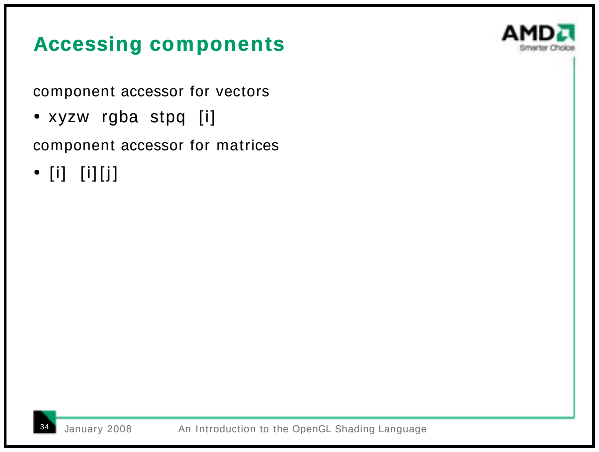
No casting

```
float f; int i; bool b;
vec2 v2; vec3 v3; vec4 v4;

vec2(1.0 ,2.0)
vec3(0.0 ,0.0 ,1.0)
vec4(1.0 ,0.5 ,0.0 ,1.0)
vec4(1.0)                  // all 1.0
vec4(v2 ,v2)
vec4(v3 ,1.0)

float(i)
int(b)
```



Matrix Constructors

```
vec4 v4; mat4 m4;

mat4( 1.0, 5.0, 9.0, 13.0,
      2.0, 6.0, 10.0, 14.0,
      3.0, 7.0, 11.0, 15.0,
      4.0, 8.0, 12.0, 16.0)   // column major

mat4( v4, v4, v4, v4)
mat4( 1.0)                  // identity matrix
mat3( m4)                   // upper 3x3
vec4( m4)                   // 1st column
float( m4)                  // upper 1x1
```



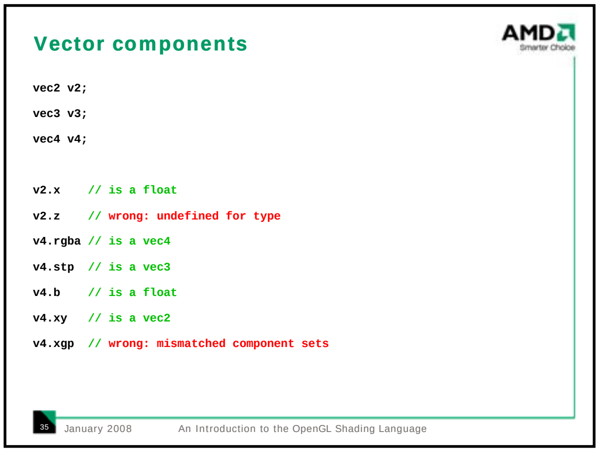
Accessing components

component accessor for vectors

- xyzw rgba stpq [i]

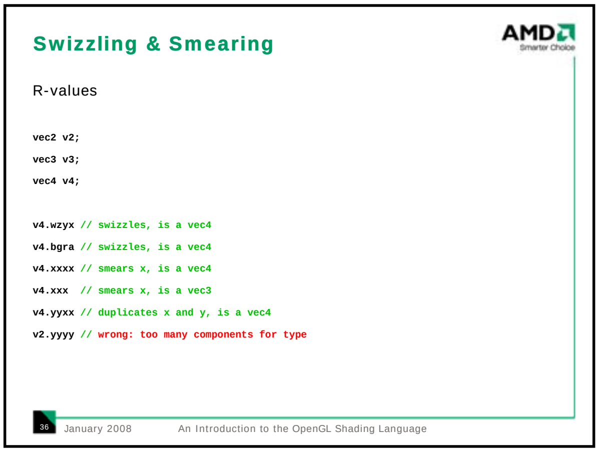
component accessor for matrices

- [i] [i][j]



Vector components

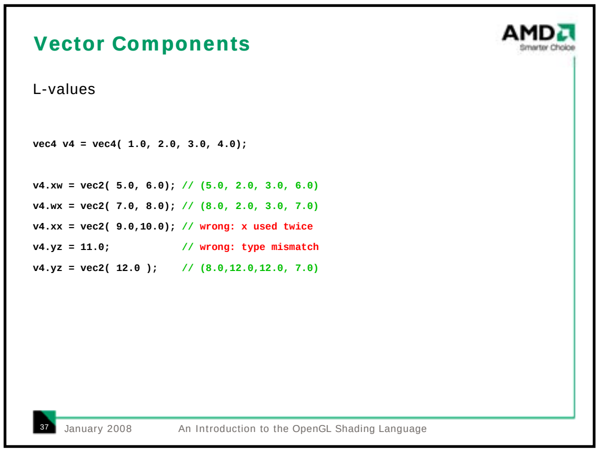
```
vec2 v2;
vec3 v3;
vec4 v4;

v2.x    // is a float
v2.z    // wrong: undefined for type
v4.rgba // is a vec4
v4.stp  // is a vec3
v4.b    // is a float
v4.xy   // is a vec2
v4.xgp  // wrong: mismatched component sets
```



Swizzling & Smearing

R-values

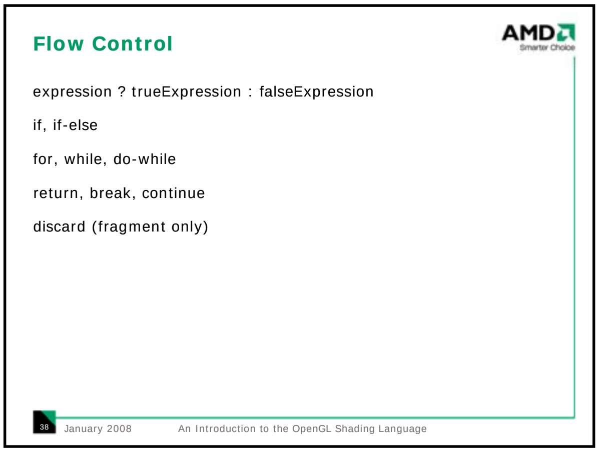
```
vec2 v2;
vec3 v3;
vec4 v4;

v4.wzyx // swizzles, is a vec4
v4.bgra // swizzles, is a vec4
v4.xxxx // smears x, is a vec4
v4.xxx  // smears x, is a vec3
v4.yyxx // duplicates x and y, is a vec4
v2.yyyy // wrong: too many components for type
```



Vector Components

L-values

```
vec4 v4 = vec4( 1.0, 2.0, 3.0, 4.0);

v4.xw = vec2( 5.0, 6.0); // (5.0, 2.0, 3.0, 6.0)
v4.wx = vec2( 7.0, 8.0); // (8.0, 2.0, 3.0, 7.0)
v4.xx = vec2( 9.0,10.0); // wrong: x used twice
v4.yz = 11.0;            // wrong: type mismatch
v4.yz = vec2( 12.0 );    // (8.0,12.0,12.0, 7.0)
```



Flow Control

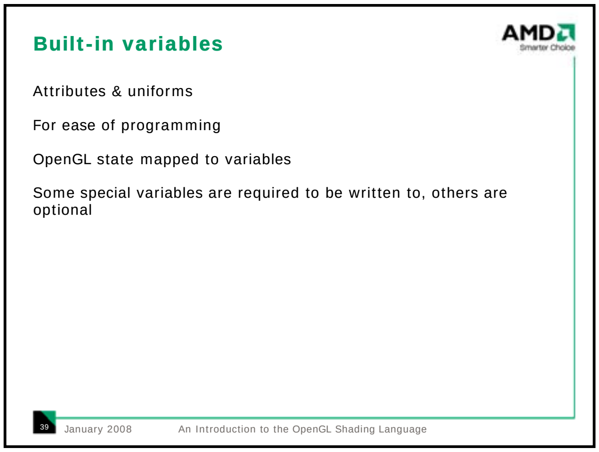
expression ? trueExpression : falseExpression

if, if-else

for, while, do-while

return, break, continue

discard (fragment only)



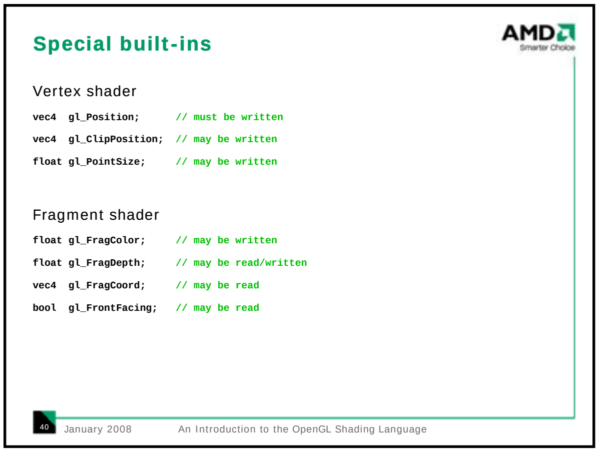
Built-in variables

Attributes & uniforms

For ease of programming

OpenGL state mapped to variables

Some special variables are required to be written to, others are optional



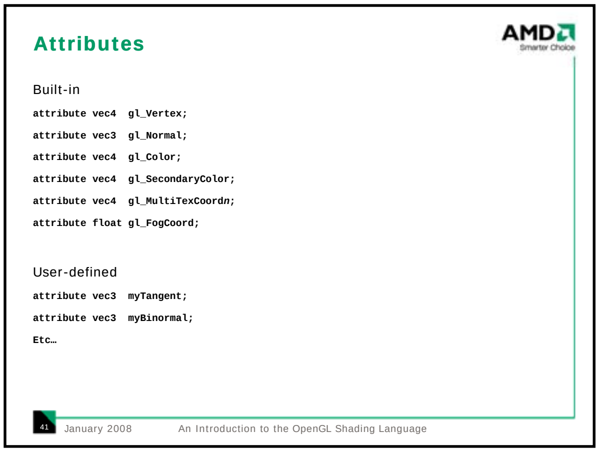
Special built-ins

Vertex shader

```
vec4  gl_Position;      // must be written
vec4  gl_ClipPosition;  // may be written
float gl_PointSize;     // may be written
```

Fragment shader

```
float gl_FragColor;     // may be written
float gl_FragDepth;     // may be read/written
vec4  gl_FragCoord;     // may be read
bool  gl_FrontFacing;   // may be read
```



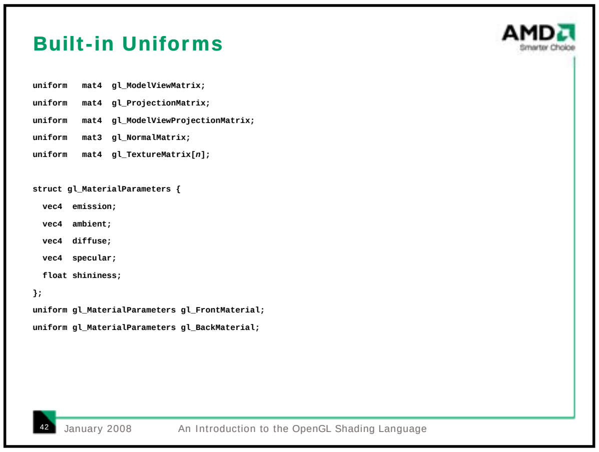
Attributes

Built-in

```
attribute vec4  gl_Vertex;
attribute vec3  gl_Normal;
attribute vec4  gl_Color;
attribute vec4  gl_SecondaryColor;
attribute vec4  gl_MultiTexCoordn;
attribute float gl_FogCoord;
```

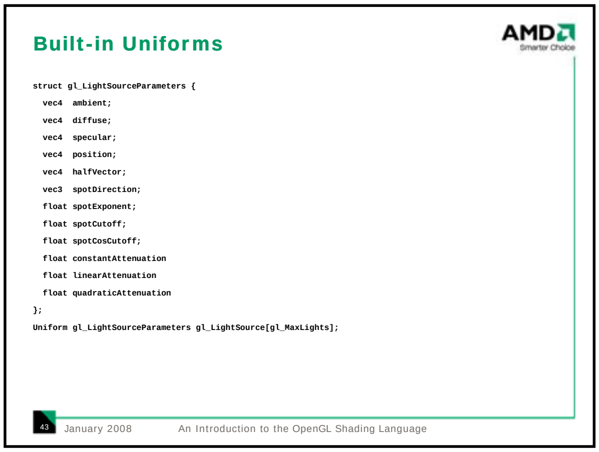
User-defined

```
attribute vec3  myTangent;
attribute vec3  myBinormal;
Etc…
```



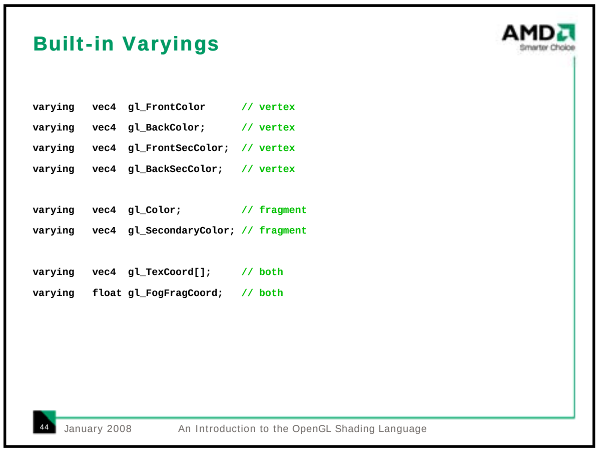
Built-in Uniforms

```
uniform   mat4  gl_ModelViewMatrix;
uniform   mat4  gl_ProjectionMatrix;
uniform   mat4  gl_ModelViewProjectionMatrix;
uniform   mat3  gl_NormalMatrix;
uniform   mat4  gl_TextureMatrix[n];

struct gl_MaterialParameters {
  vec4  emission;
  vec4  ambient;
  vec4  diffuse;
  vec4  specular;
  float shininess;
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```



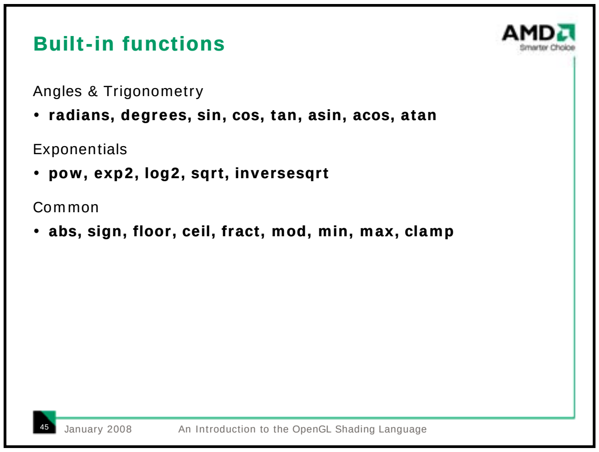
Built-in Uniforms

```
struct gl_LightSourceParameters {
  vec4  ambient;
  vec4  diffuse;
  vec4  specular;
  vec4  position;
  vec4  halfVector;
  vec3  spotDirection;
  float spotExponent;
  float spotCutoff;
  float spotCosCutoff;
  float constantAttenuation
  float linearAttenuation
  float quadraticAttenuation
};
Uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
```



Built-in Varyings

```
varying   vec4  gl_FrontColor      // vertex
varying   vec4  gl_BackColor;      // vertex
varying   vec4  gl_FrontSecColor;  // vertex
varying   vec4  gl_BackSecColor;   // vertex

varying   vec4  gl_Color;          // fragment
varying   vec4  gl_SecondaryColor; // fragment

varying   vec4  gl_TexCoord[];     // both
varying   float gl_FogFragCoord;   // both
```



Built-in functions

Angles & Trigonometry

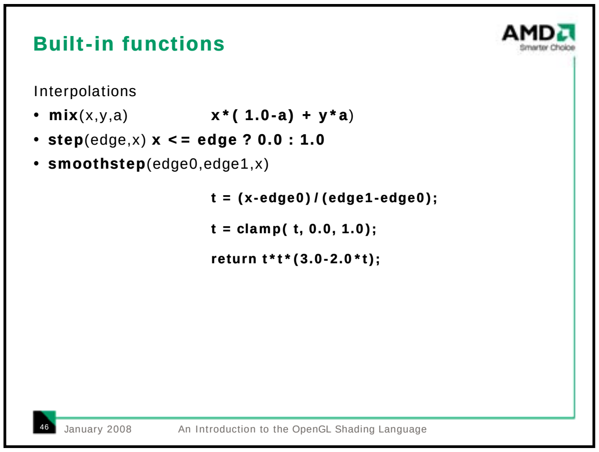
- **radians, degrees, sin, cos, tan, asin, acos, atan**

Exponentials

- **pow, exp2, log2, sqrt, inversesqrt**

Common

- **abs, sign, floor, ceil, fract, mod, min, max, clamp**

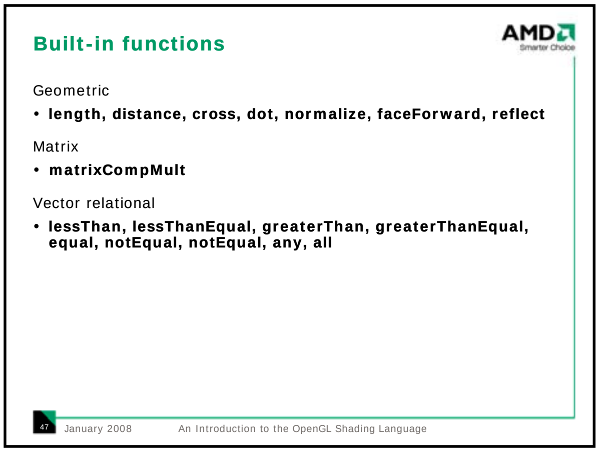


Built-in functions

Interpolations

- **mix**(x,y,a) **x*(1.0-a) + y*a**)
- **step**(edge,x) **x <= edge ? 0.0 : 1.0**
- **smoothstep**(edge0,edge1,x)

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



Built-in functions

Geometric

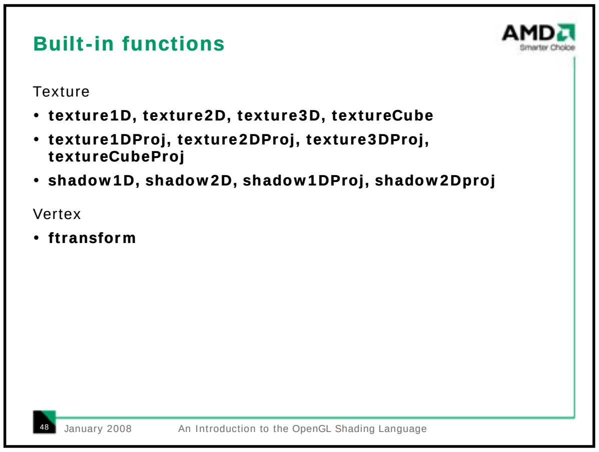
- **length, distance, cross, dot, normalize, faceForward, reflect**

Matrix

- **matrixCompMult**

Vector relational

- **lessThan, lessThanEqual, greaterThan, greaterThanEqual, equal, notEqual, notEqual, any, all**



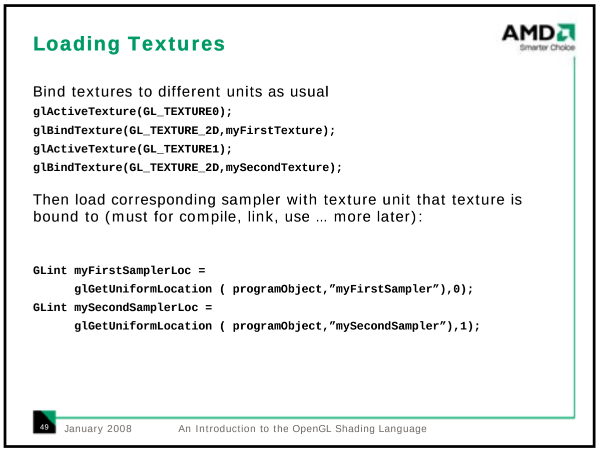
Built-in functions

Texture

- **texture1D, texture2D, texture3D, textureCube**
- **texture1DProj, texture2DProj, texture3DProj, textureCubeProj**
- **shadow1D, shadow2D, shadow1DProj, shadow2Dproj**

Vertex

- **ftransform**



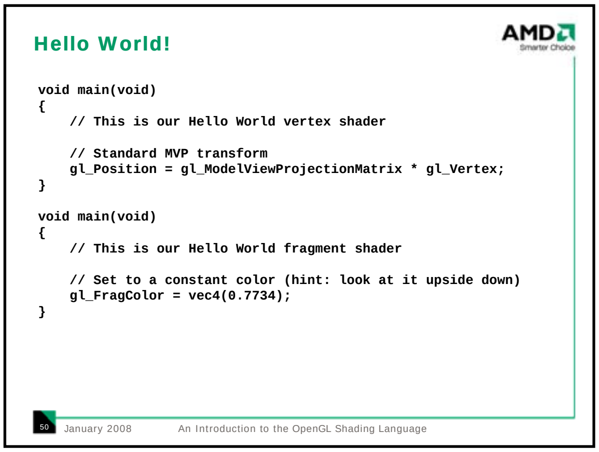
Loading Textures

Bind textures to different units as usual

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D,myFirstTexture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D,mySecondTexture);
```

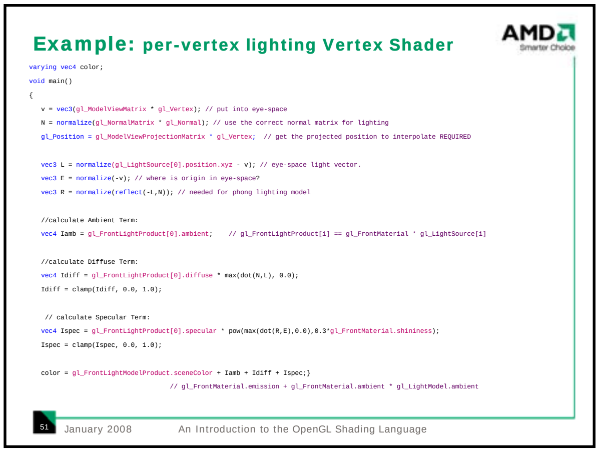
Then load corresponding sampler with texture unit that texture is bound to (must for compile, link, use … more later):

```
GLint myFirstSamplerLoc =
      glGetUniformLocation ( programObject,”myFirstSampler”),0);
GLint mySecondSamplerLoc =
      glGetUniformLocation ( programObject,”mySecondSampler”),1);
```



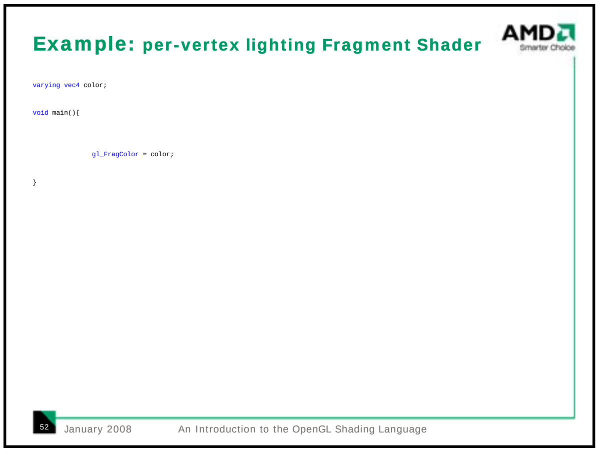
Hello World!

```
void main(void)
{
    // This is our Hello World vertex shader

    // Standard MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

void main(void)
{
    // This is our Hello World fragment shader

    // Set to a constant color (hint: look at it upside down)
    gl_FragColor = vec4(0.7734);
}
```



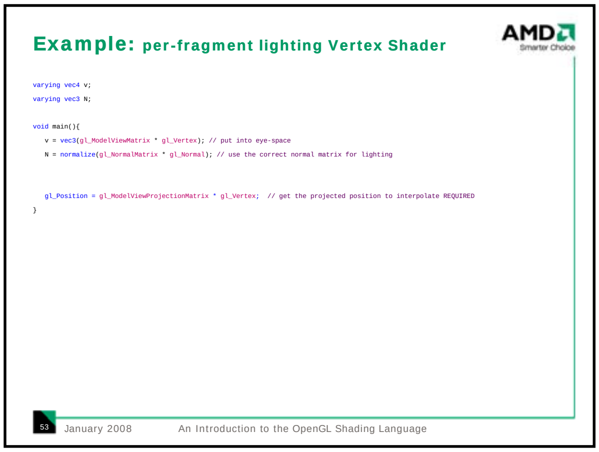
Example: per-vertex lighting Vertex Shader

```
varying vec4 color;
void main()
{
   v = vec3(gl_ModelViewMatrix * gl_Vertex); // put into eye-space
   N = normalize(gl_NormalMatrix * gl_Normal); // use the correct normal matrix for lighting
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  // get the projected position to interpolate REQUIRED

   vec3 L = normalize(gl_LightSource[0].position.xyz - v); // eye-space light vector.
   vec3 E = normalize(-v); // where is origin in eye-space?
   vec3 R = normalize(reflect(-L,N)); // needed for phong lighting model

   //calculate Ambient Term:
   vec4 Iamb = gl_FrontLightProduct[0].ambient;    // gl_FrontLightProduct[i] == gl_FrontMaterial * gl_LightSource[i]

   //calculate Diffuse Term:
   vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);
   Idiff = clamp(Idiff, 0.0, 1.0);

    // calculate Specular Term:
   vec4 Ispec = gl_FrontLightProduct[0].specular * pow(max(dot(R,E),0.0),0.3*gl_FrontMaterial.shininess);
   Ispec = clamp(Ispec, 0.0, 1.0);

   color = gl_FrontLightModelProduct.sceneColor + Iamb + Idiff + Ispec;}
                                 // gl_FrontMaterial.emission + gl_FrontMaterial.ambient * gl_LightModel.ambient
```



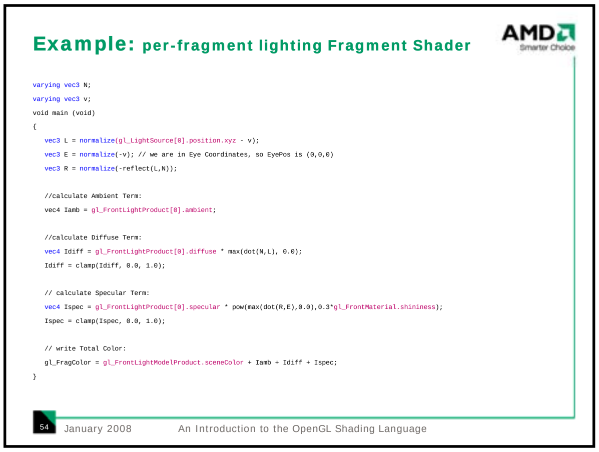
Example: per-vertex lighting Fragment Shader

```
varying vec4 color;

void main(){

               gl_FragColor = color;

}
```



Example: per-fragment lighting Vertex Shader

```
varying vec4 v;
varying vec3 N;

void main(){
   v = vec3(gl_ModelViewMatrix * gl_Vertex); // put into eye-space
   N = normalize(gl_NormalMatrix * gl_Normal); // use the correct normal matrix for lighting


   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  // get the projected position to interpolate REQUIRED
}
```



Example: per-fragment lighting Fragment Shader

```
varying vec3 N;
varying vec3 v;
void main (void)
{
   vec3 L = normalize(gl_LightSource[0].position.xyz - v);
   vec3 E = normalize(-v); // we are in Eye Coordinates, so EyePos is (0,0,0)
   vec3 R = normalize(-reflect(L,N));

   //calculate Ambient Term:
   vec4 Iamb = gl_FrontLightProduct[0].ambient;

   //calculate Diffuse Term:
   vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);
   Idiff = clamp(Idiff, 0.0, 1.0);

   // calculate Specular Term:
   vec4 Ispec = gl_FrontLightProduct[0].specular * pow(max(dot(R,E),0.0),0.3*gl_FrontMaterial.shininess);
   Ispec = clamp(Ispec, 0.0, 1.0);

   // write Total Color:
   gl_FragColor = gl_FrontLightModelProduct.sceneColor + Iamb + Idiff + Ispec;
}
```

Basic method

2 basic object types

- Shader object
- Program object

Create Vertex & Fragment Shader Objects

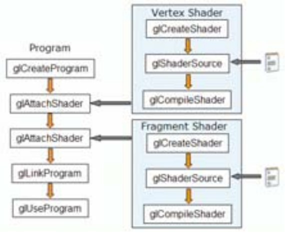
Compile both

Create program object & attach shaders

Link program

Use program

Creating Shaders



Compiling

```
void glShaderSource(GLuint shader, GLsizei nstrings, const GLchar **strings,
   const GLint *lengths)
    //if lengths==NULL, assumed to be null-terminated


void glCompileShader (GLuint shader);
```

Attaching & Linking

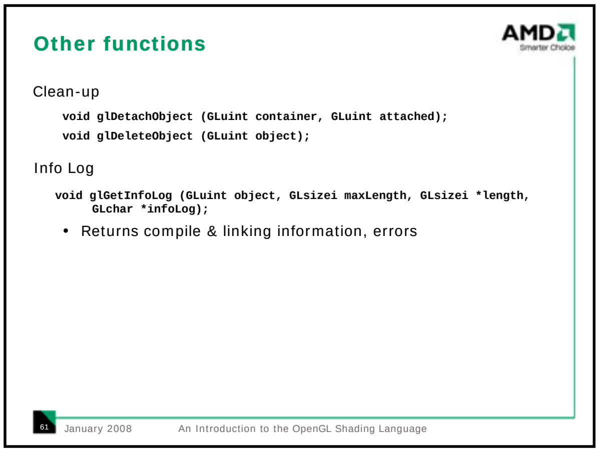
```
void glAttachShader(GLuint program, GLuint shader);
    //twice, once for vertex shader & once for fragment shader

void glLinkProgram(GLuint program);
   //program now ready to use

void glUseProgram(GLuint program);
   //switches on shader, bypasses FFP
   //if program==0, shaders turned off, returns to FFP
```

In short…

```
GLuint programObject;
GLuint vertexShaderObject;
GLuint fragmentShaderObject;

unsigned char *vertexShaderSource = readShaderFile(vertexShaderFilename);
unsigned char *fragmentShaderSource = readShaderFile(fragmentShaderFilename);

programObject=glCreateProgram ();
vertexShaderObject=glCreateShader (GL_VERTEX_SHADER);
fragmentShaderObject=glCreateShader (GL_FRAGMENT_SHADER);

glShaderSource (vertexShaderObject,1,(const char**)&vertexShaderSource,NULL);
glShaderSource (fragmentShaderObject,1,(const char**)&fragmentShaderSource,NULL);

glCompileShader (vertexShaderObject);
glCompileShader (fragmentShaderObject);

glAttachObject (programObject, vertexShaderObject);
glAttachObject (programObject, fragmentShaderObject);

glLinkProgram (programObject);

glUseProgram (programObject);
```

Example

```
void setShaders() {
        char *vs,*fs;

        v = glCreateShader(GL_VERTEX_SHADER);
        f = glCreateShader(GL_FRAGMENT_SHADER);

        vs = textFileRead("toon.vert");
        fs = textFileRead("toon.frag");

        const char * vv = vs;
        const char * ff = fs;

        glShaderSource(v, 1, &vv,NULL);
        glShaderSource(f, 1, &ff,NULL);

        free(vs);free(fs);

        glCompileShader(v);
        glCompileShader(f);

        p = glCreateProgram();

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

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



Other functions

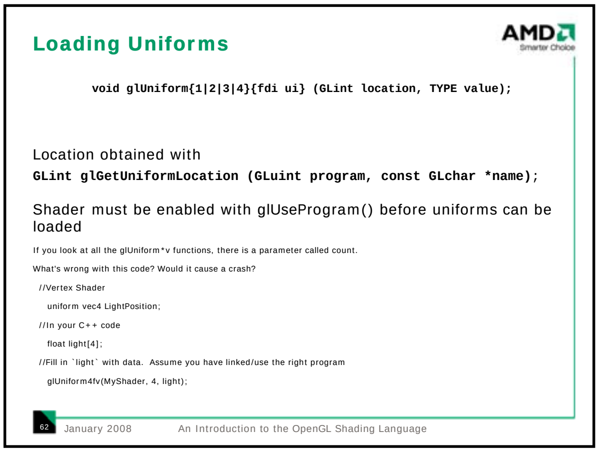
Clean-up

```
void glDetachObject (GLuint container, GLuint attached);
void glDeleteObject (GLuint object);
```

Info Log

```
void glGetInfoLog (GLuint object, GLsizei maxLength, GLsizei *length,
     GLchar *infoLog);
```

- Returns compile & linking information, errors



Loading Uniforms

```
void glUniform{1|2|3|4}{fdi ui} (GLint location, TYPE value);
```

Location obtained with

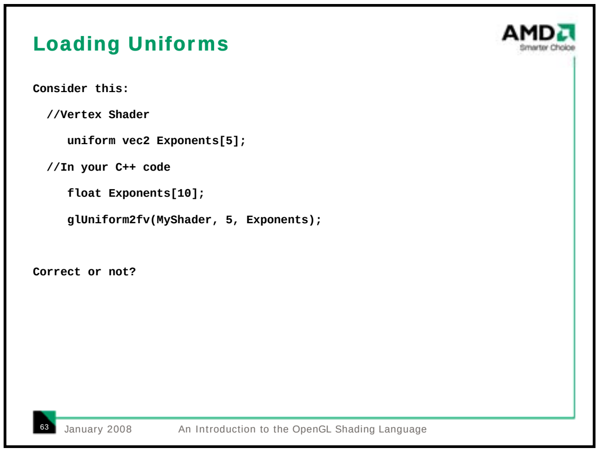
```
GLint glGetUniformLocation (GLuint program, const GLchar *name);
```

Shader must be enabled with glUseProgram() before uniforms can be loaded

If you look at all the glUniform*v functions, there is a parameter called count.

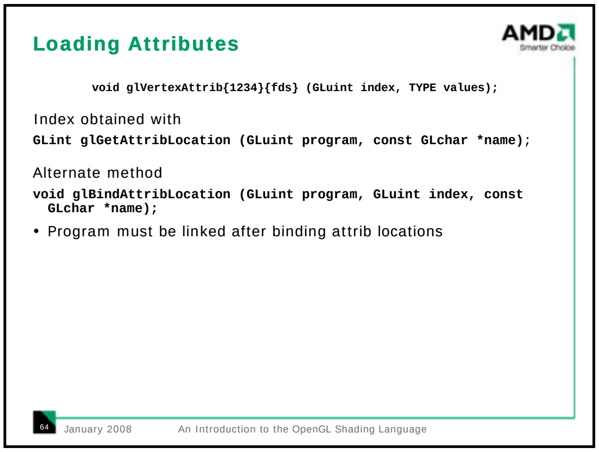
What's wrong with this code? Would it cause a crash?

```
 //Vertex Shader
   uniform vec4 LightPosition;
 //In your C++ code
   float light[4];
 //Fill in `light` with data.  Assume you have linked/use the right program
   glUniform4fv(MyShader, 4, light);
```



Loading Uniforms

```
Consider this:
  //Vertex Shader
     uniform vec2 Exponents[5];
  //In your C++ code
     float Exponents[10];
     glUniform2fv(MyShader, 5, Exponents);


Correct or not?
```



Loading Attributes

```
void glVertexAttrib{1234}{fds} (GLuint index, TYPE values);
```

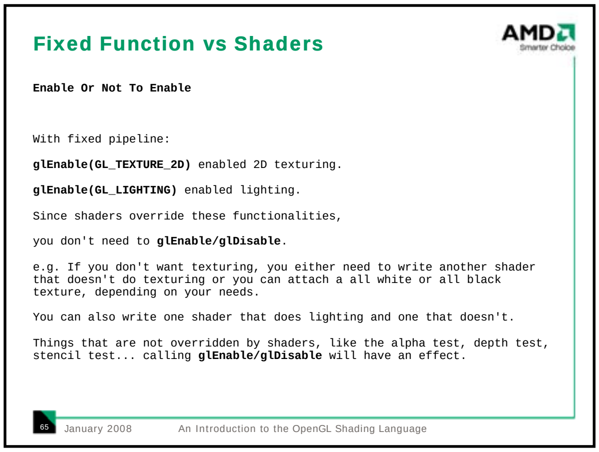
Index obtained with

```
GLint glGetAttribLocation (GLuint program, const GLchar *name);
```

Alternate method

```
void glBindAttribLocation (GLuint program, GLuint index, const
  GLchar *name);
```

- Program must be linked after binding attrib locations



Fixed Function vs Shaders

Enable Or Not To Enable

With fixed pipeline:

glEnable(GL_TEXTURE_2D) enabled 2D texturing.

glEnable(GL_LIGHTING) enabled lighting.

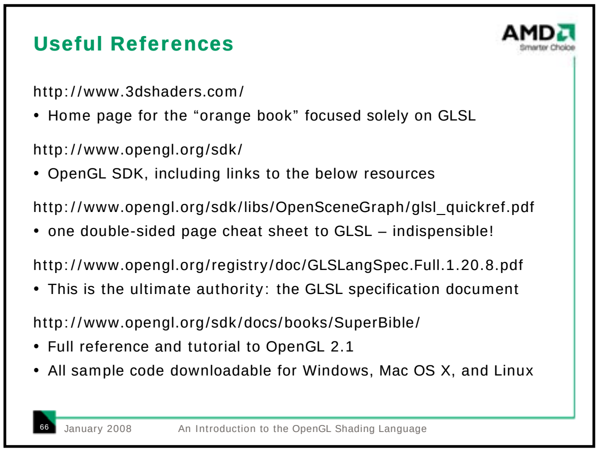
Since shaders override these functionalities,

you don't need to **glEnable/glDisable**.

e.g. If you don't want texturing, you either need to write another shader that doesn't do texturing or you can attach a all white or all black texture, depending on your needs.

You can also write one shader that does lighting and one that doesn't.

Things that are not overridden by shaders, like the alpha test, depth test, stencil test... calling **glEnable/glDisable** will have an effect.



Useful References

http://www.3dshaders.com/

- Home page for the “orange book” focused solely on GLSL

http://www.opengl.org/sdk/

- OpenGL SDK, including links to the below resources

http://www.opengl.org/sdk/libs/OpenSceneGraph/glsl_quickref.pdf

- one double-sided page cheat sheet to GLSL – indispensible!

http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf

- This is the ultimate authority: the GLSL specification document

http://www.opengl.org/sdk/docs/books/SuperBible/

- Full reference and tutorial to OpenGL 2.1
- All sample code downloadable for Windows, Mac OS X, and Linux