AMD
Smarter Choice

# An Introduction to the OpenGL Shading Language

Benj Lipchak
Rob Simpson
Bill Licea-Kane

## Outline

A little history

How the fixed function pipeline works

How it's replaced by GLSL

Structure & syntax nitty-gritty

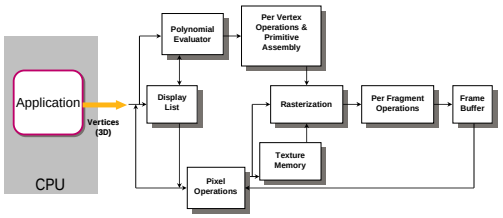
How to integrate GLSL into OpenGL apps

Some simple examples

Resources

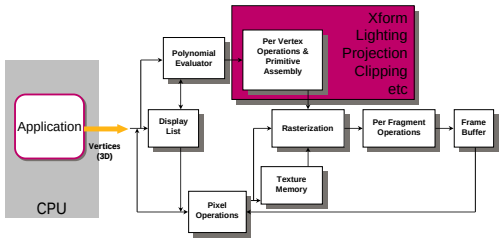


## Traditional Graphics Pipeline



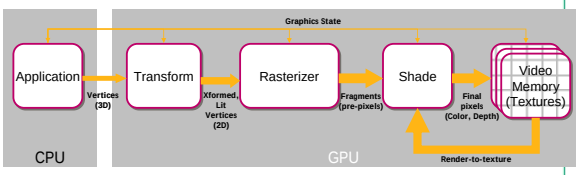



## Traditional Graphics Pipeline






## Traditional Graphics Pipeline



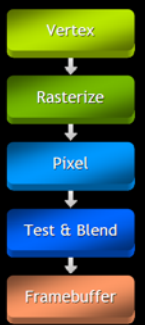

A simplified graphics pipeline

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



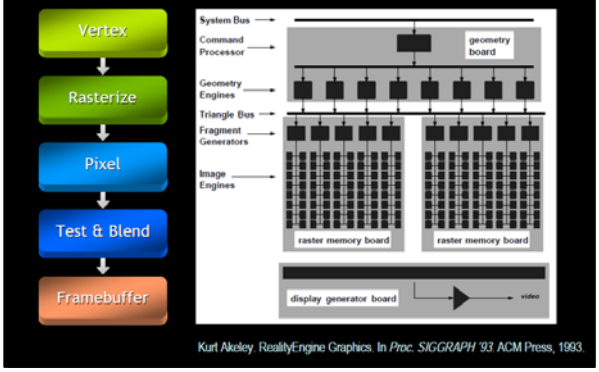
## The Graphics Pipeline

SIGGRAPH2008



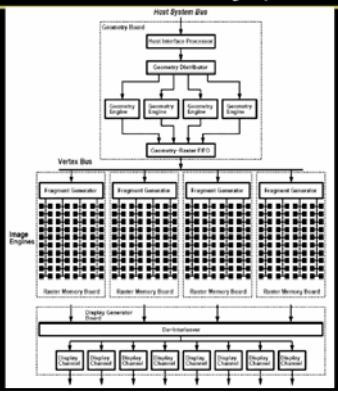

- Key abstraction of real-time graphics
- Hardware used to look like this
- Distinct chips/boards per stage
- Fixed data flow through pipeline

## SGI RealityEngine (1993)



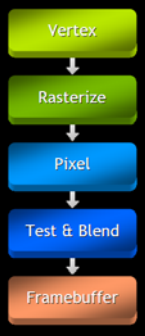

Kurt Akeley. RealityEngine Graphics. In *Proc. SIGGRAPH '93*. ACM Press, 1993.

## SGI InfiniteReality (1997)




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

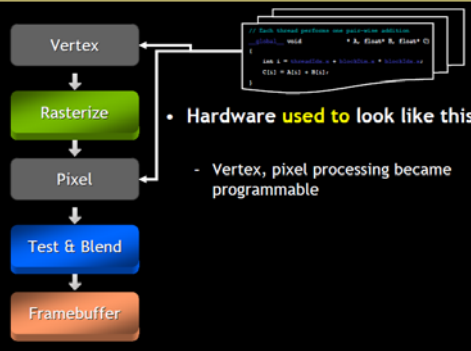
## The Graphics Pipeline




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

Beyond Programmable Shading: In Action

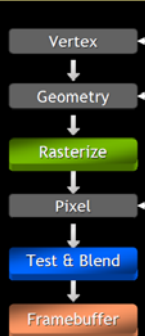
## The Graphics Pipeline




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

Beyond Programmable Shading: In Action

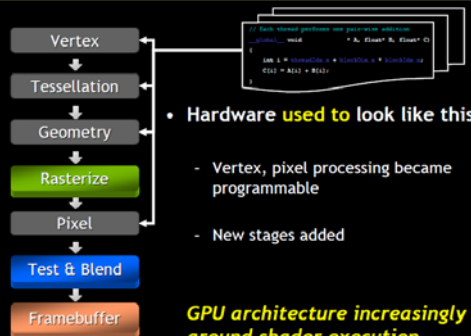
## The Graphics Pipeline




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

Beyond Programmable Shading: In Action

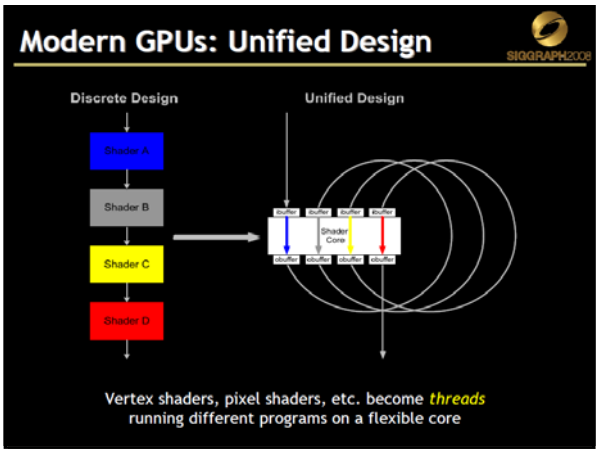
## The Graphics Pipeline

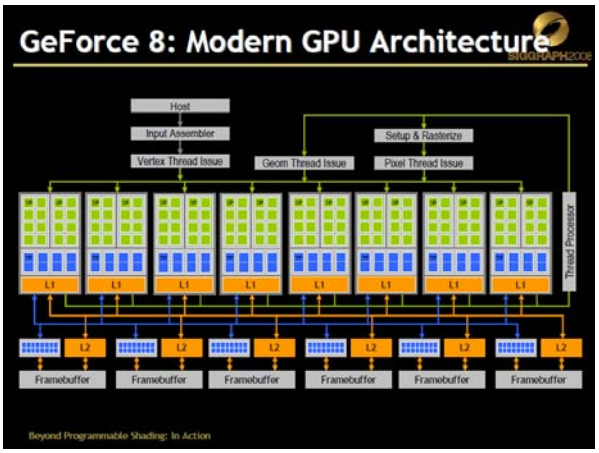


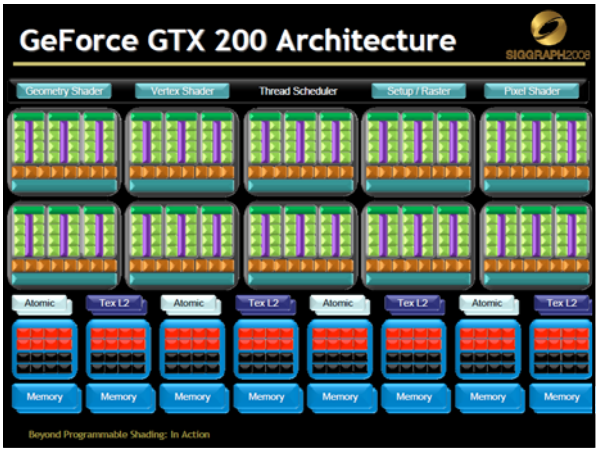

- Hardware used to look like this
  - Vertex, pixel processing became programmable
  - New stages added

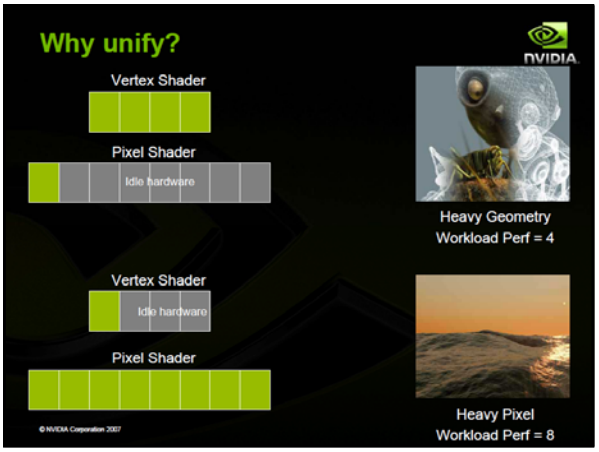
***GPU architecture increasingly centers around shader execution***

Beyond Programmable Shading: In Action

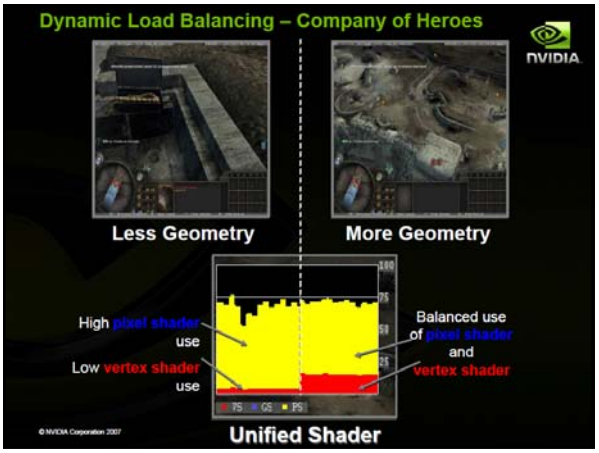

Modern GPUs: Unified Design
SIGGRAPH2008
Discrete Design
Unified Design
Shader A
Shader B
Shader C
Shader D
Shader Core
Vertex shaders, pixel shaders, etc. become threads
running different programs on a flexible core


GeForce 8: Modern GPU Architecture
SIGGRAPH2008
Host
Input Assembler
Vertex Thread Issue
Setup & Rasterize
Geom Thread Issue
Pixel Thread Issue
Thread Processor
L1
L2
Framebuffer
Beyond Programmable Shading: In Action

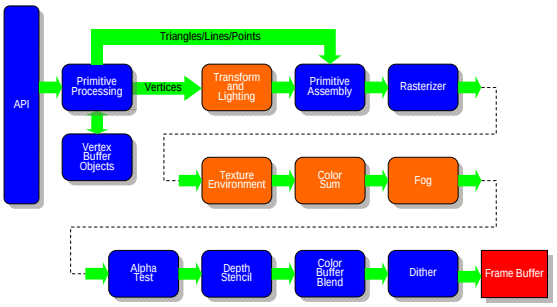

GeForce GTX 200 Architecture
SIGGRAPH2008
Geometry Shader
Vertex Shader
Thread Scheduler
Setup / Raster
Pixel Shader
Atomic
Tex L2
Memory
Beyond Programmable Shading: In Action


Why unify?
NVIDIA
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?
NVIDIA
Unified Shader
Vertex Workload
Pixel
Heavy Geometry
Workload Perf = 11
Unified Shader
Pixel Workload
Vertex
Heavy Pixel
Workload Perf = 11
© NVIDIA Corporation 2007

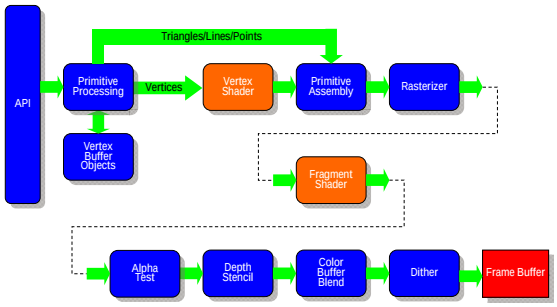

Dynamic Load Balancing – Company of Heroes
NVIDIA
Less Geometry
More Geometry
High pixel shader use
Low vertex shader use
Balanced use of pixel shader and vertex shader
VS
GS
PS
Unified Shader
© NVIDIA Corporation 2007

## Fixed Functionality Pipeline



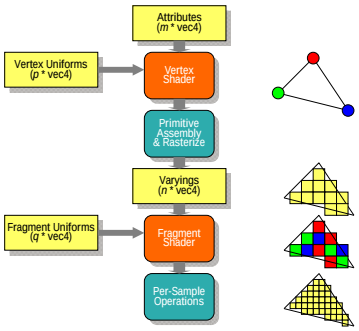



## Programmable Shader Pipeline






## Programmer's Model






## Previous programmability

Texture Shaders

Register Combiners

Assembly programs
- ARB_vertex_program
- ARB_fragment_program
- Messy!

Needed general, readable & maintainable language



## Using Programmability

~2000-2002: ASM

```
!!VP1.0
#
# c[0-3]  = modelview projection (comp
# c[4-7]  = modelview inverse transpos
# c[32]   = eye-space light direction
# c[33]   = eye-space half-angle vecto
# c[35].x = diffuse light * mat.
# c[35].y = ambient light * mat.
# c[36]   = specular color
# c[38].x = specular power
# outputs homogenous position and colo
#
DP4   o[HPOS].x, c[0], v[OPOS];      #
DP4   o[HPOS].y, c[1], v[OPOS];
DP4   o[HPOS].z, c[2], v[OPOS];
DP4   o[HPOS].w, c[3], v[OPOS];
DP3   R0.x, c[4], v[NRML];           #
DP3   R0.y, c[5], v[NRML];
DP3   R0.z, c[6], v[NRML];           #
DP3   R1.x, c[32], R0;               #
DP3   R1.y, c[33], R0;               #
MOV   R1.w, c[38].x;                 #
LIT   R2, R1;                        #
MAD   R3, c[35].x, R2.y, c[35].y;    #
MAD   o[COL0].xyz, c[36], R2.z, R3;  #
END
```

Now: C-like

```
vertout main(appin IN,

    uniform float4x4 ModelViewProj,
    uniform float4x4 ModelViewIT,
    uniform float3 lightVec,
    uniform float3 halfVec,
    uniform float3 diffuseMaterial,
    uniform float3 ambientCol,
    uniform float3 specularMaterial,
    uniform float specexp){
vertout OUT; //struct w/ HPosition, Color

OUT.HPosition = mul(ModelViewProj,
    IN.Position);


float3 normalVec = normalize(
            mul(ModelViewIT,IN.Normal).xyz);

float diffuse = dot(normalVec, lightVec);
float spec = dot(normalVec, halfVec);

float4 lighting = lit(diffuse,spec,specexp);
OUT.Color.rgb = lighting.y * diffuseMaterial
+ambientCol + lighting.z * specularMaterial;
OUT.Color.a = 1.0;
return OUT;
}
```



## Example Apps.

Custom transform, lighting, and skinning

## Example Apps.

Custom cartoon-style lighting



## Example Apps.

- Per-vertex set up for per-pixel bump mapping



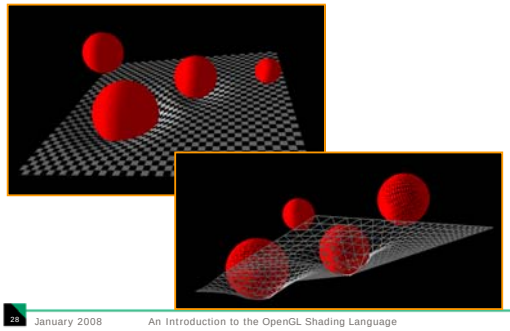
## Example Apps.

- Character morphing & shadow volume projection



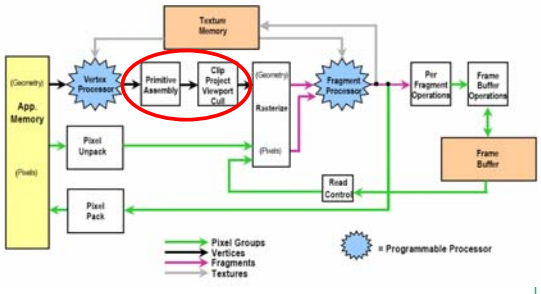
## Example Apps.

- Dynamic displacements of surfaces by objects



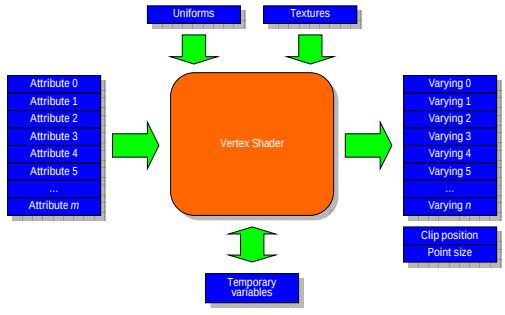


## OpenGL 2.0 Logical Diagram

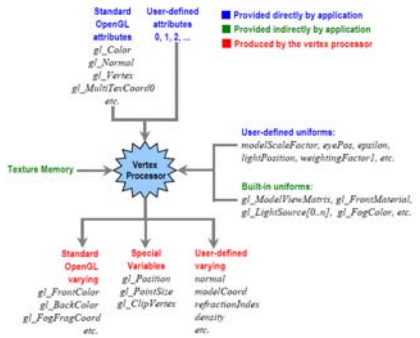





## Vertex Shader Environment



## Vertex Processor Overview






## Vertices: What You Don't Get

Connectivity (neighbor face, edge, vtx)

Can't Create/Destroy Vertices (coming soon)

Large Writable Memory



## Vertices: Expensive (Slow!) Ops

Branches (if, for, while)

Large R/O Memory (textures)



## Vertices: Workarounds

Connectivity (neighbor face, edge, vtx)
- *Encode neighbor info as attributes*

Can't Create/Destroy Vertices
- *Create: start w/ more than you need & specialize*
- *Destroy: move outside clip volume*

Large Writable Memory
- *But fragments do (frame buffer)*



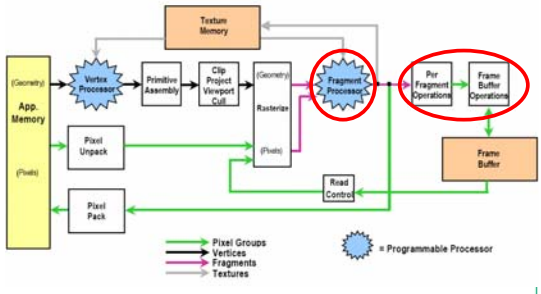
## Vertices: Efficiency

Branches (if, for, while)
- *(a<1)?b:c; unroll loops*

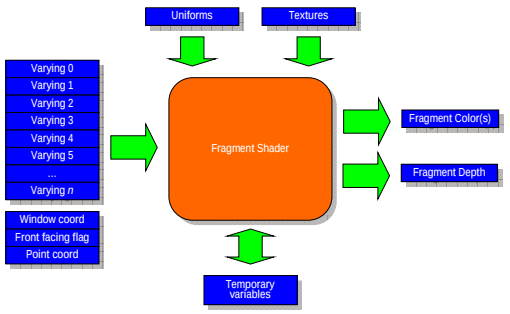
Large R/O Memory (textures)
- *Can put small tables in uniform arrays*



## OpenGL 2.0 Logical Diagram

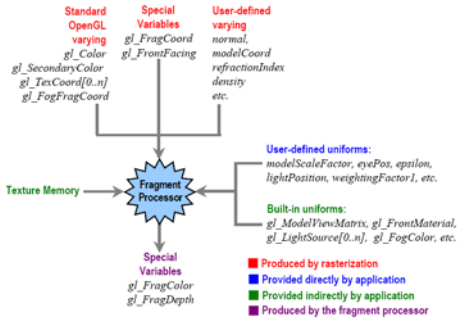


## Fragment Shader Environment






## Fragment Processor Overview






## Fragment Processor

Flexibility for texturing and per-pixel pixel operations

Fragment processing replaces the following:

- Ops on interpolated values
- Texture access
- Texture application
- Fog
- Color sum
- Pixel Zoom
- Scale and bias
- Color table lookup
- Convolution
- Color matrix



## FP does NOT replace

- Shading model
- Coverage
- Pixel ownership test
- Scissor
- Stipple
- Alpha test
- Depth test
- Stencil test
- Alpha blending
- Logical ops
- Dithering
- Plane masking
- Histogram
- Minmax
- Pixel packing and unpacking



## 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);
}
```



## In General…

Vertex processes bypassed

- Vertex Transformation
- Normal Transformation, Normalization
- Lighting
- Texture Coordinate Generation and Transformation

Fragment processes bypassed

- Texture accesses & application
- Fog

## 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, 2.0, 3.0, 4.0,
      5.0, 6.0, 7.0, 8.0,
      9.0, 10., 11., 12.,
      13., 14., 15., 16.)   // row 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

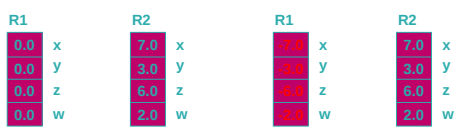
```
vec2 v2;
vec3 v3;
vec4 v4;

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
```



## Assembly Language

**Source registers can be negated:**

```
MOV    R1, -R2;
```

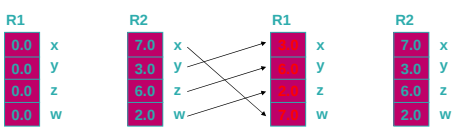





## Assembly Language

**Source registers can be "swizzled":**

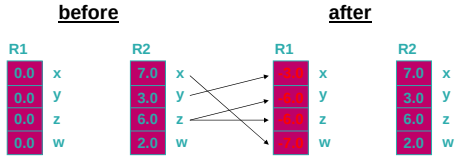
```
MOV    R1, R2.yzwx;
```



## Assembly Language

Source registers can be negated and “swizzled":

```
MOV    R1, -R2.yzzx;
```

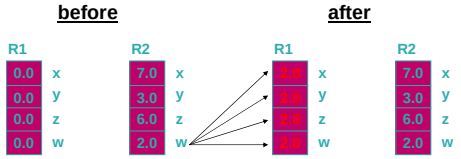





## Assembly Language

Source registers can be swizzled by “smearing":

```
MOV    R1, R2.w;     # alternative to
                     # using R2.wwww
```






## 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**



## Example: Vertex Shader

```
varying vec4 diffuseColor;
varying vec3 fragNormal;
varying vec3 lightVector;

uniform vec3 eyeSpaceLightVector;

void main(){

        vec3 eyeSpaceVertex= vec3(gl_ModelViewMatrix * gl_Vertex);
        lightVector= vec3(normalize(eyeSpaceLightVector - eyeSpaceVertex));
        fragNormal = normalize(gl_NormalMatrix * gl_Normal);

        diffuseColor = gl_Color;
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

}
```



## Example: Fragment Shader

```
varying vec4 diffuseColor;
varying vec3 lightVector;
varying vec3 fragNormal;

void main(){

        float perFragmentLighting=max(dot(lightVector,fragNormal),0.0);

        gl_FragColor = diffuseColor * lightingFactor;

}
```



## Basic method

2 basic object types

- Shader object
- Program object

Create Vertex & Fragment Shader Objects

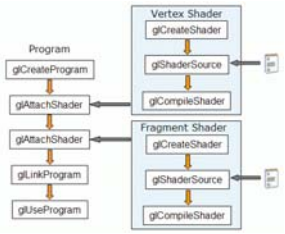
Compile both

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}{f|i} (GLint location,…);
```

Location obtained with

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

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



## Loading Attributes

```
void glVertexAttrib{1234}{sfd} (GLint   index,…);
```

Index obtained with

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

Alternate method

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

- Program must be linked **after** binding attrib locations



## 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

```
glUniform1i (glGetUniformLocation ( programObject,"myFirstSampler"),0);
glUniform1i (glGetUniformLocation ( programObject,"mySecondSampler"),1);
```



## Starter Shaders: color manipulation

```
// simple.fs
//
// copy primary color

void main(void)
{
    // Copy the primary color
    gl_FragColor = gl_Color;
}


// colorinvert.fs
//
// invert like a color negative

void main(void)
{
    // invert color components
    gl_FragColor.rgb = 1.0 - gl_Color.rgb;
    gl_FragColor.a = 1.0;
}
```



## Starter Shaders: color manipulation

```
// grayscale.fs
//
// convert RGB to grayscale

void main(void)
{
    // Convert to grayscale using NTSC conversion weights
    float gray = dot(gl_Color.rgb, vec3(0.299, 0.587, 0.114));

    // replicate grayscale to RGB components
    gl_FragColor = vec4(gray, gray, gray, 1.0);
}

// sepia.fs
//
// convert RGB to sepia tone

void main(void)
{
    // Convert to grayscale using NTSC conversion weights
    float gray = dot(gl_Color.rgb, vec3(0.299, 0.587, 0.114));

    // convert grayscale to sepia
    gl_FragColor = vec4(gray * vec3(1.2, 1.0, 0.8), 1.0);
}
```



## Starter Shaders: color manipulation

```
// heatsig.fs
//
// map grayscale to heat signature

uniform sampler1D sampler0;

void main(void)
{
    // Convert to grayscale using NTSC conversion weights
    float gray = dot(gl_Color.rgb, vec3(0.299, 0.587, 0.114));

    // look up heatsig value
    gl_FragColor = texture1D(sampler0, gray);
}
```

## Starter Shaders: color manipulation

```
// fog.fs
//
// per-pixel fog

uniform float density;

void main(void)
{
    const vec4 fogColor = vec4(0.5, 0.8, 0.5, 1.0);

    // calculate 2nd order exponential fog factor
    // based on fragment's Z distance
    const float e = 2.71828;
    float fogFactor = (density * gl_FragCoord.z);
    fogFactor *= fogFactor;
    fogFactor = clamp(pow(e, -fogFactor), 0.0, 1.0);

    // Blend fog color with incoming color
    gl_FragColor = mix(fogColor, gl_Color, fogFactor);
}
```



## Starter Shaders: convolution

```
// passthrough.fs
//
// pass through a single texel value

uniform sampler2D sampler0;

void main(void)
{
    gl_FragColor = texture2D(sampler0, gl_TexCoord[0].st);
}
```



## Starter Shaders: convolution

```
// blur.fs
//
// blur (low-pass) 3x3 kernel

uniform sampler2D sampler0;
uniform vec2 tc_offset[9];

void main(void)
{
    vec4 sample[9];

    for (int i = 0; i < 9; i++)
    {
        sample[i] = texture2D(sampler0,
                              gl_TexCoord[0].st + tc_offset[i]);
    }

//   1 2 1
//   2 1 2   / 13
//   1 2 1

    gl_FragColor = (sample[0] + (2.0*sample[1]) + sample[2] +
                    (2.0*sample[3]) + sample[4] + (2.0*sample[5]) +
                    sample[6] + (2.0*sample[7]) + sample[8]) / 13.0;
}
```



## Starter Shaders: convolution

| | |
|---|---|
| Blur | 1 2 1<br>2 1 2 / 13<br>1 2 1 |
| Sharpen | -1 -1 -1<br>-1 9 -1<br>-1 -1 -1 |
| LaPlacian | -1 -1 -1<br>-1 8 -1<br>-1 -1 -1 |
| Dilation | max(kernel) |
| Erosion | min(kernel) |



## Starter Shaders: vertex shaders

```
// simple.vs
//
// Generic vertex transformation,
// copy primary color

void main(void)
{
    // normal MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // Copy the primary color
    gl_FrontColor = gl_Color;
}
```



## Starter Shaders: vertex shaders

```
// diffuse.vs
//
// Generic vertex transformation,
// diffuse lighting based on one
// white light

uniform vec3 lightPos[1];

void main(void)
{
    // normal MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    vec3 N = normalize(gl_NormalMatrix * gl_Normal);
    vec4 V = gl_ModelViewMatrix * gl_Vertex;
    vec3 L = normalize(lightPos[0] - V.xyz);

    // output the diffuse color
    float NdotL = dot(N, L);
    gl_FrontColor = gl_Color * vec4(max(0.0, NdotL));
}
```

## Example: Fragment Shader

```
varying vec4 diffuseColor;
varying vec3 lightVector;
varying vec3 fragNormal;

void main(){

    float perFragmentLighting=max(dot(lightVector,fragNormal),0.0);

    gl_FragColor = diffuseColor * lightingFactor;

}
```



## Starter Shaders: vertex shaders

```
// ptsize.vs
//
// Generic vertex transformation,
// attenuated point size

void main(void)
{
    // normal MVP transform
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    vec4 V = gl_ModelViewMatrix * gl_Vertex;

    gl_FrontColor = gl_Color;

    // calculate point size based on distance from eye
    float ptSize = length(V);
    ptSize = ptSize * ptSize * ptSize;
    gl_PointSize = 20000000.0 / ptSize;
}
```



## Starter Shaders: vertex shaders

```
// stretch.vs
//
// Generic vertex transformation,
// followed by squash/stretch

uniform vec3 lightPos[1];
uniform vec3 squashStretch;

void main(void)
{
    // normal MVP transform, followed by squash/stretch
    vec4 stretchedCoord = gl_Vertex;
    stretchedCoord.xyz *= squashStretch;
    gl_Position = gl_ModelViewProjectionMatrix * stretchedCoord;

    ...
}
```



## Ivory – vertex shader

```
uniform vec4 lightPos;

varying vec3 normal;
varying vec3 lightVec;
varying vec3 viewVec;

void main(){
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vec4 vert = gl_ModelViewMatrix * gl_Vertex;

    normal   = gl_NormalMatrix * gl_Normal;
    lightVec = vec3(lightPos - vert);
    viewVec  = -vec3(vert);
}
```



## Ivory – fragment shader

```
varying vec3 normal;
varying vec3 lightVec;
varying vec3 viewVec;

void main(){
      vec3 norm = normalize(normal);

      vec3 L = normalize(lightVec);
      vec3 V = normalize(viewVec);
      vec3 halfAngle = normalize(L + V);

      float NdotL = dot(L, norm);
    float NdotH = clamp(dot(halfAngle, norm), 0.0, 1.0);

    // "Half-Lambert" technique for more pleasing diffuse term
      float diffuse  = 0.5 * NdotL + 0.5;
      float specular = pow(NdotH, 64.0);

      float result = diffuse + specular;

      gl_FragColor = vec4(result);
}
```



## Gooch – vertex shader

```
uniform vec4 lightPos;

varying vec3 normal;
varying vec3 lightVec;
varying vec3 viewVec;

void main(){
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vec4 vert = gl_ModelViewMatrix * gl_Vertex;

    normal   = gl_NormalMatrix * gl_Normal;
    lightVec = vec3(lightPos - vert);
    viewVec  = -vec3(vert);
}
```

## Gooch – fragment shader

```
uniform vec3 ambient;

varying vec3 normal;
varying vec3 lightVec;
varying vec3 viewVec;

void main(){
      const float b = 0.55;
      const float y = 0.3;
      const float Ka = 1.0;
      const float Kd = 0.8;
      const float Ks = 0.9;

      vec3 specularcolor = vec3(1.0, 1.0, 1.0);

      vec3 norm = normalize(normal);
      vec3 L = normalize (lightVec);
      vec3 V = normalize (viewVec);
      vec3 halfAngle = normalize (L + V);
```



## Gooch – fragment shader (2)

```
      vec3 orange = vec3(.88,.81,.49);
      vec3 purple = vec3(.58,.10,.76);

      vec3 kCool = purple;
      vec3 kWarm = orange;

      float NdotL = dot(L, norm);
      float NdotH = clamp(dot(halfAngle, norm), 0.0, 1.0);
      float specular = pow(NdotH, 64.0);

      float blendval  = 0.5 * NdotL + 0.5;
      vec3 Cgooch = mix(kWarm, kCool, blendval);

      vec3 result = Ka * ambient + Kd * Cgooch + specularcolor * Ks * specular;

      gl_FragColor = vec4(result, 1.0);
}
```



## 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