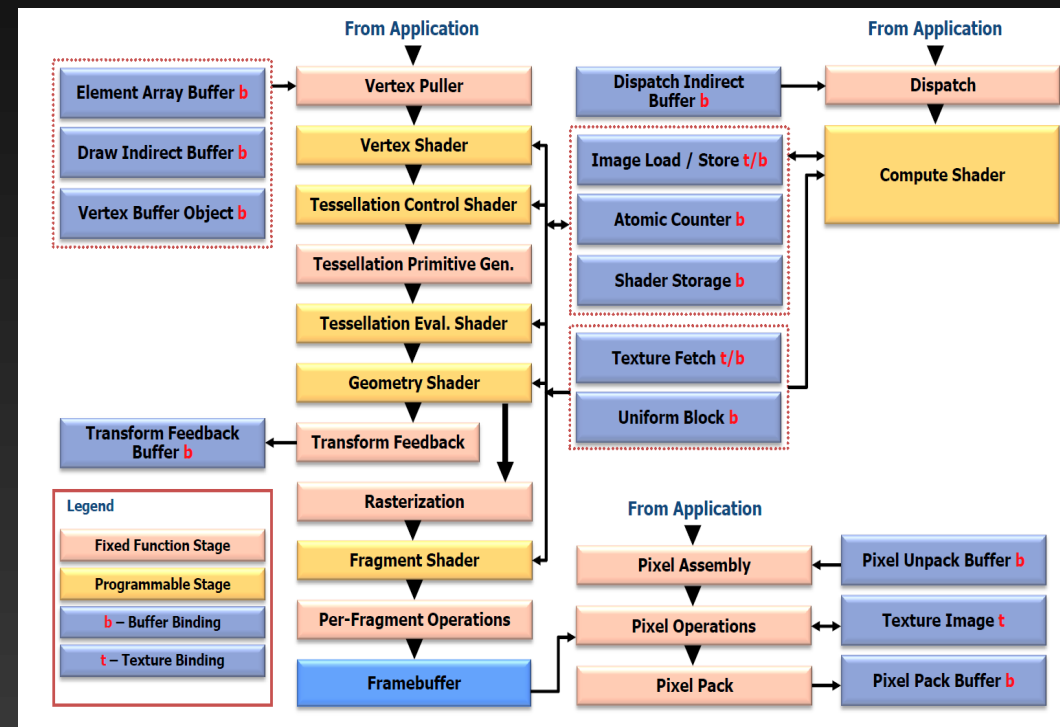


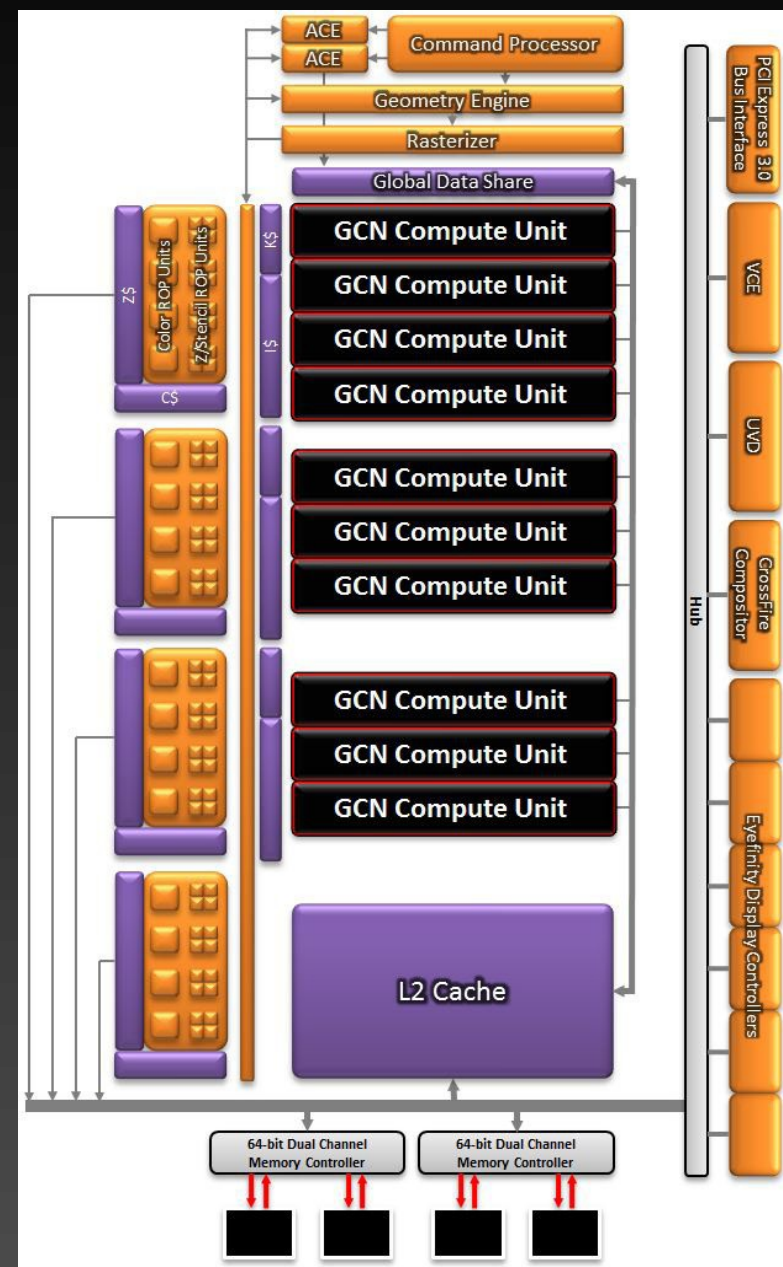
# Shaders

# What Is a Shader?

- Shader = small program, executed directly on GPU for each vertex (fragment)
- Shader defines function of the pipeline
- Usually cooperation of **at least** two programs
  - **vertex** shader and **fragment** shader



# Evolution of hardware



# Why shaders?

- Used mostly for lighting and shading
- **Lighting** – compute intensity and color in point of scene, as viewed by user
  - function of scene geometry (model, lights, camera and their positions) and material properties
  - per vertex/fragment = **vertex or fragment shader**
- **Shading** – process of interpolation of color and intensity between point, where exact lighting has been computed
  - mostly in real-time graphics (trade-off between quality and speed)
  - per fragment = **fragment shader**

# Vertex shading (fixed pipeline default)

- few vertices, lots of fragments – almost every triangle is larger than single fragment
  - compute light only for vertex → save computations
- compute lighting in vertex, interpolate for fragments
  - Gouraud – linear interpolation
  - flat – constant fill

# Fragment shading

- compute lighting in every fragment – per-pixel lighting
  - Phong shading
    - more precise, time consuming, better reflections
    - to reach the same quality with Gouraud it would be necessary to increase polygon count → tessellation, ...



# Literature

- OpenGL Programming Guide
- OpenGL Superbible
- Computer Graphics Programming in OpenGL with C++
- [http://registry.khronos.org/OpenGL/index\\_gl.php](http://registry.khronos.org/OpenGL/index_gl.php)
- [www.shadertoy.com](http://www.shadertoy.com)
  - online WebGL shader development

# Shader language

- 2 main possibilities
  - **GLSL** (OpenGL Shading Language)
    - part of OpenGL since version 1.5 (r. 2003)
    - part of OpenGL ES since version 1.0 (r. 2003)
  - HLSL (High Level Shading Language)
    - Microsoft Direct3D 8+ (r. 2000)
    - Nvidia + Microsoft
- RSL (Renderman Shading Language)
- Cg (C for graphics)
- assembler of target device
  - almost unusable and unused
- ...



GPU ShaderAnalyzer - DetailTessellation.hlsl - DX HLSL

File Edit Help

Source Code

Function VS\_NoTessellation

```
148 VS_OUTPUT_NO_TESSELLATION VS_NoTessellation( VS_INPUT i )
149 {
150     VS_OUTPUT_NO_TESSELLATION Out;
151
152     // Compute position in world space
153     float4 vPositionWS = mul( i.inPositionOS.xyz, g_mWorld
154
155     // Compute denormalized light vector in world space
156     float3 vLightWS = g_LightPosition.xyz - vPositionWS.xyz
157     // Need to invert Z for correct lighting
158     vLightWS.z = -vLightWS.z;
159
160     // Propagate texture coordinate through:
161     Out.texCoord = i.inTexCoord * g_fBaseTextureRepeat.x;
162
163     // Transform normal, tangent and binormal vectors from
164     float3 vNormalWS = mul( i.vInNormalOS, (float3x3) g
165     float3 vBinormalWS = mul( i.vInBinormalOS, (float3x3) g
166     float3 vTangentWS = mul( i.vInTangentOS, (float3x3) g
167
168     // Normalize tangent space vectors
169     vNormalWS = normalize( vNormalWS );
170     vBinormalWS = normalize( vBinormalWS );
171     vTangentWS = normalize( vTangentWS );
172
173     // Calculate tangent basis
174     float3x3 mWorldToTangent = float3x3( vTangentWS, vBinor
175
176     // Calculate tangent space light vector
177     float3 vLightTS = mul( mWorldToTangent, vLightWS.xyz );
178
179     #if PERPIXEL_DIFFUSE_LIGHTING==1
180     // Per-pixel lighting
181
```

Compile

Source type HLSL

HLSL Compiler

Target vs\_5\_0

☐ Skip Validation

☒ No Geometry Shader ☐ Use DX9 Semantics

☐ Enable Strictness ☐ IEEE Strictness

Flow Control Not Set

Optimization Not Set

Pack Matrix Not Set

GLSL Compiler

Macro Definitions

Symbol	Value
PERPIXEL_DIFFUSE_LIGHTING	1
DEBUG_VIEW	0

Bool Constants

Object Code

Format Radeon HD 5870 (Cypress) Assembly

```
z: DOT4_e      ___, R1.z, KC1[12].z
w: DOT4_e      ___, PV0.w, KC1[12].w
2  x: DOT4_e      ___, R1.x, KC1[13].x
  y: DOT4_e      R2.y, R1.y, KC1[13].y
  z: DOT4_e      ___, R1.z, KC1[13].z
  w: DOT4_e      ___, T0.w, KC1[13].w
3  x: DOT4_e      ___, R1.x, KC1[14].x
  y: DOT4_e      ___, R1.y, KC1[14].y
  z: DOT4_e      R2.z, R1.z, KC1[14].z
  w: DOT4_e      ___, T0.w, KC1[14].w
4  x: DOT4_e      ___, R1.x, KC1[15].x
  y: DOT4_e      ___, R1.y, KC1[15].y
  z: DOT4_e      ___, R1.z, KC1[15].z
  w: DOT4_e      R2.w, T0.w, KC1[15].w
5  x: DOT4_e      T3.x, R5.x, KC1[0].x
  y: DOT4_e      ___, R5.y, KC1[0].y
  z: DOT4_e      ___, T0.x, 1.0f      VEC_120
  w: DOT4_e      ___, (0x80000000, -0.0f).x, 0.0f
  t: MULADD_e    T0.x, R4.y, KC1[0].y, T0.z      VEC_
6  x: DOT4_e      ___, R5.x, KC1[1].x
  y: DOT4_e      T0.y, R5.y, KC1[1].y
  z: DOT4_e      ___, R5.z, KC1[1].z
  w: DOT4_e      ___, (0x80000000, -0.0f).x, 0.0f
  t: MUL_e       T1.x, R4.z, KC1[1].z
7  x: DOT4_e      ___, R5.x, KC1[2].x
  y: DOT4_e      ___, R5.y, KC1[2].y
  z: DOT4_e      T0.z, R5.z, KC1[2].z
  w: DOT4_e      ___, (0x80000000, -0.0f).x, 0.0f
  t: MUL_e       T2.x, R4.z, KC1[2].z
8  x: MULADD_e    T0.x, R4.x, KC1[0].x, T0.x      VEC_
  y: MUL_e       ___, PV7.x, PV7.x
  t: MULADD_e    ___, R4.y, KC1[1].y, T1.x      VEC_
9  x: MULADD_e    ___, R4.y, KC1[2].y, T2.x
  y: MULADD_e    T1.y, R4.x, KC1[1].x, PS8
```

Compiler Statistics - Old

Name	GPR	Scratch Reg	Min	Max	Avg	Est Cycles(Bi)	ALU:TEX(Bi)	Est Cycles(Tri)	ALU:TEX(Tri)	Est Cycles(Aniso)	ALU:TEX(Aniso)	BottleNeck(Bi)	BottleNeck(Tri)	BottleNeck(Aniso)	Item/Clock(Bi)	Item/Clock(Tri)	Item/Clock(Aniso)
Radeon HD 4550	9	0	11.50	11.50	11.50	11.50	11.50	11.50	11.50	11.50	11.50	ALU Ops	ALU Ops	ALU Ops	0.70	0.70	0.70
Radeon HD 4670	9	0	3.00	3.00	3.00	3.00	2.88	3.00	2.88	3.00	2.88	Exports	Exports	Exports	2.67	2.67	2.67
Radeon HD 4770	9	0	3.00	3.00	3.00	3.00	2.88	3.00	2.88	3.00	2.88	Exports	Exports	Exports	5.33	5.33	5.33
Radeon HD 4870	9	0	3.00	3.00	3.00	3.00	2.20	3.00	2.20	3.00	2.20	Exports	Exports	Exports	5.33	5.33	5.33
Radeon HD 4890	9	0	3.00	3.00	3.00	3.00	2.20	3.00	2.20	3.00	2.20	Exports	Exports	Exports	5.33	5.33	5.33
Radeon HD 5450	6	0	11.50	11.50	11.50	11.50	11.50	11.50	11.50	11.50	11.50	ALU Ops	ALU Ops	ALU Ops	0.35	0.35	0.35
Radeon HD 5670	6	0	4.60	4.60	4.60	4.60	4.60	4.60	4.60	4.60	4.60	ALU Ops	ALU Ops	ALU Ops	1.74	1.74	1.74
Radeon HD 5770	6	0	3.00	3.00	3.00	3.00	2.30	3.00	2.30	3.00	2.30	Exports	Exports	Exports	5.33	5.33	5.33
Radeon HD 5870	6	0	3.00	3.00	3.00	3.00	2.30	3.00	2.30	3.00	2.30	Exports	Exports	Exports	10.67	10.67	10.67

D3D Assembly Statistics

GPU ShaderAnalyzer Screenshot

# GLSL – OpenGL Shading Language

- Similar to C, same syntax, commands ...

```
void main( void ) { ... }
```

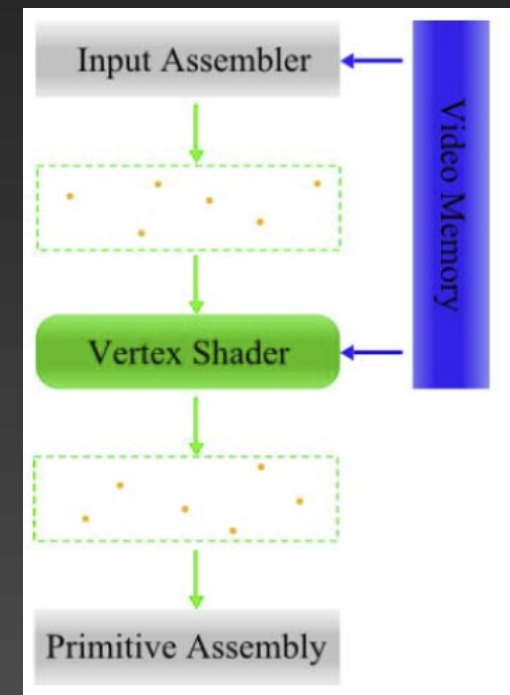
- ... but more restrictions
  - **parameters** passed **only by value**, function returns direct value, **non-existent pointers** (automatic memory management by GPU drivers)
  - **strong typing**, no automatic conversion (float x int etc.)
- Default internal variables for input and output
  - starting with gl\_
- Data types for matrices and vectors
  - and operations with them: item order, dot(), normalize(), length(), distance(), clamp(), sin(), cos(), pow(),...

# Shader types

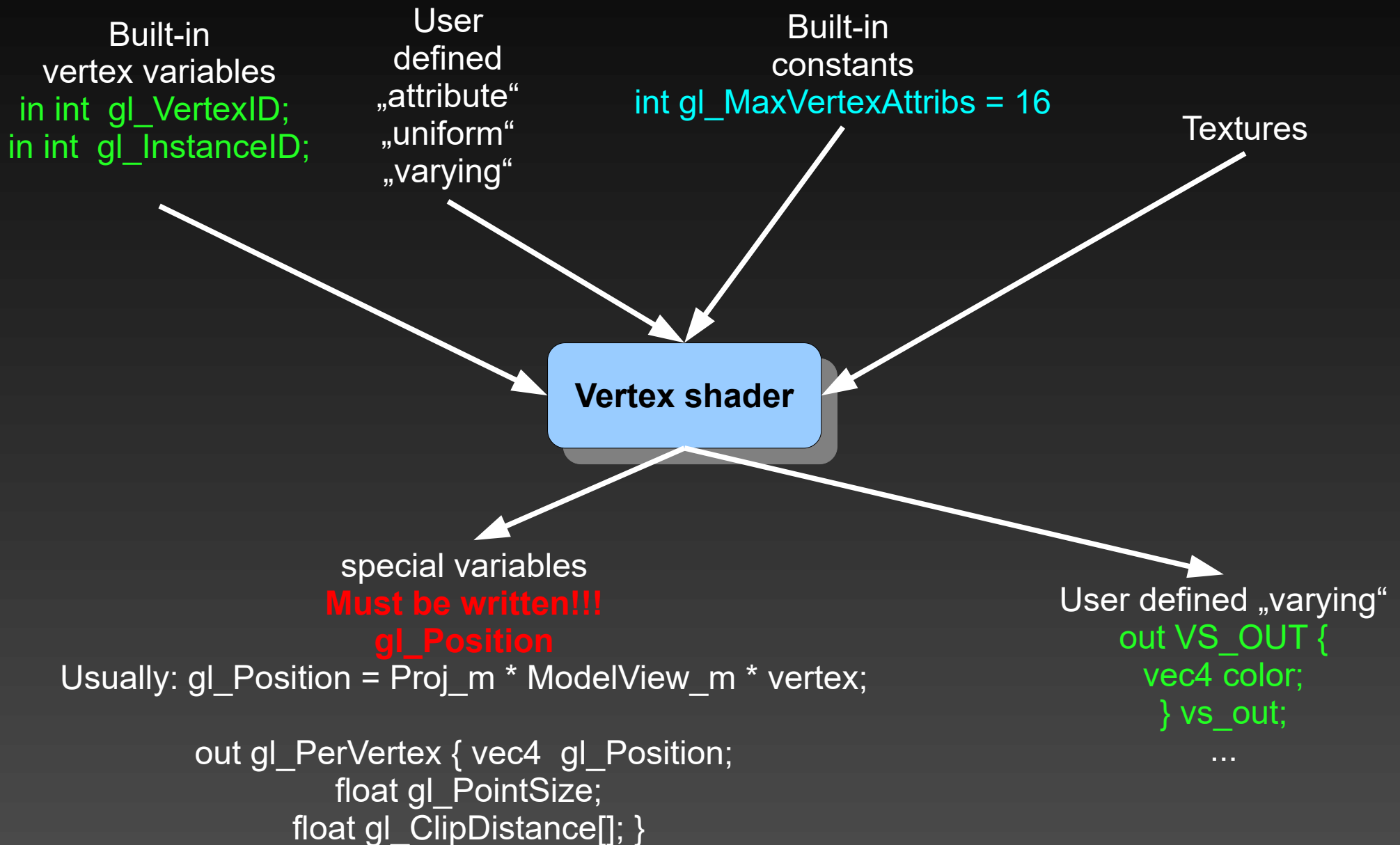
- Most common
  - **Vertex** shader
    - manipulation with single vertex
  - **Fragment** shader
    - fragment coloring
- Less common
  - Geometry shader
    - can create/drop vertices and primitives
- Least common
  - Tessellation control shader
  - Tessellation evaluation shader
  - Compute shader
  - Mesh shaders

# Vertex shader

- Usually does (can do)
  - transformations of vertex, normals, texture coordinates
  - create texture coordinates
  - compute lighting in vertex
  - set values for interpolation in next stage (fragment shader)
- Process **each vertex separately**  
→ Can **NOT** know anything about:
  - graphic primitives (!!!)
  - perspective, viewport
  - clipping planes



# Vertex shader



# Vertex Shader Example

```
#version 460 core
// shading language version specifier: must be the first

in vec4 position, color; // input variables

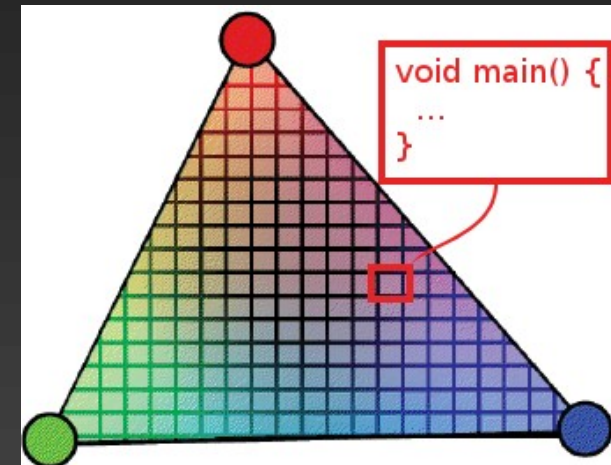
out VS_OUT                      // output variables, grouped to structure
{
    vec4 color;
} vs_out;

uniform mat4 mv_m, projection_m; // uniform variables

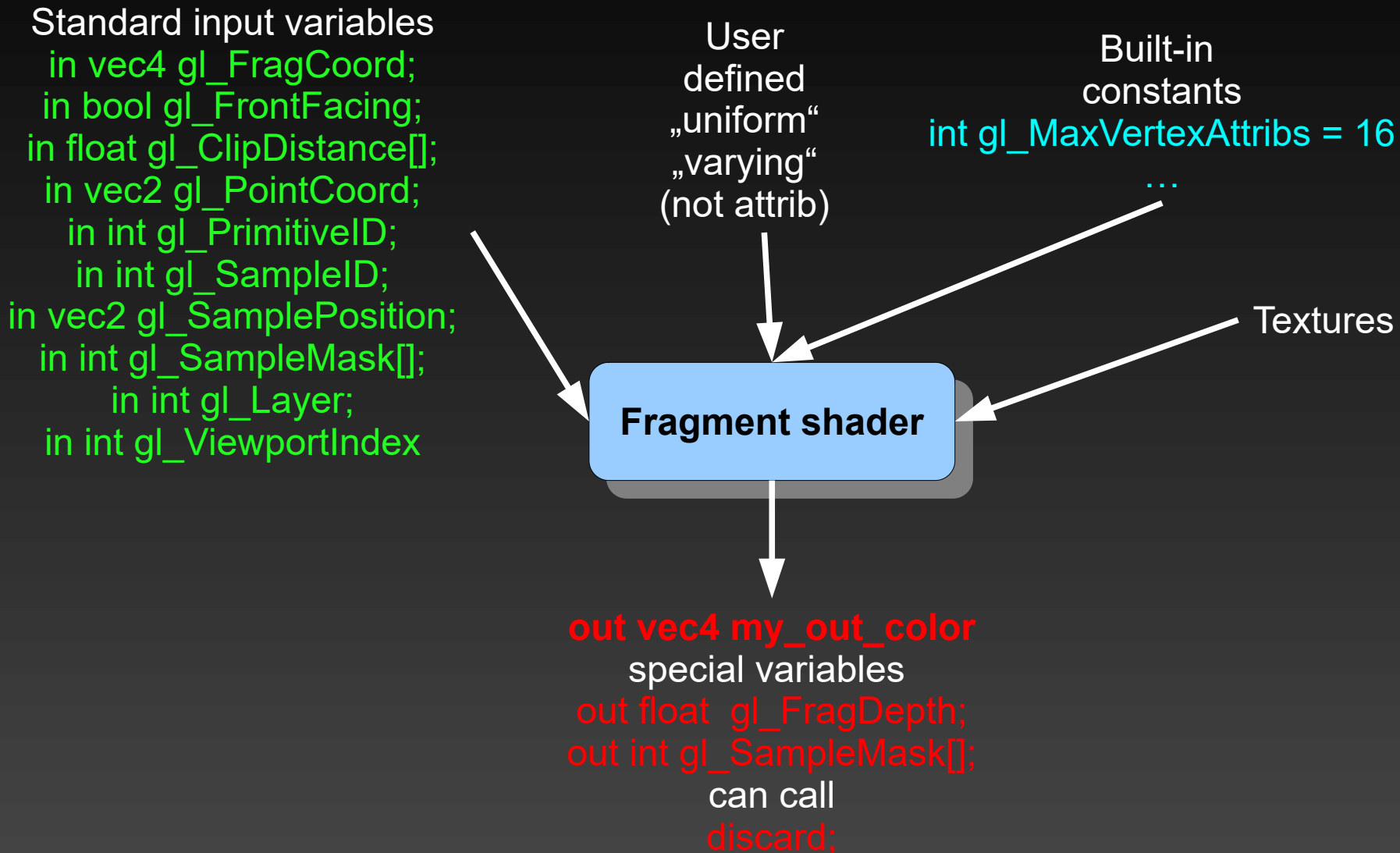
// VS main function
void main( void ) {
    // pass color
    vs_out.color = color;
    // transform and pass vertex position
    gl_Position = projection_m * mv_m * position;
}
```

# Fragment Shader (dx3d = pixel shader)

- Usually does
  - **set fragment color** – can use automatically interpolated data from vertex shader
  - fetch color from texture, multitexturing, bump maps, ...
  - compute fog and similar
  - procedural draw
- Can **NOT**
  - change fragment coordinates [x, y] (can change z)
  - write to textures
  - influence stencil, alpha, Z test, dithering, ...



# Fragment shader





# Fragment Shader Example

```
#version 460 core
// shading language version specifier: must be the first

out vec4 color; // mandatory output variable: fragment color

in VS_OUT {          // input variables grouped to structure, passed from vertex shader
    vec4 color;
} fs_in;

// FS main function
void main( void ) {
    // Pass color - color is „varying“ = „smooth in“,
    // i.e. it is already interpolated between values set in vertices

    color = fs_in.color;
}
```

# Shader Use HOWTO

## 1) Create shader

Allocate handle for each shader.

## 2) Specify shader

Pass shader source code as a string.

## 3) Compile shader

Driver really does compilation, checking syntax etc. Result is a binary object, that must be linked into a program. DO check compilation return code and compiler log!

## 4) Create program object

Compiled shaders will be linked into that.

## 5) Attach compiled shaders to the program object

Attach already compiled binary shader objects using handles.

## 6) Link all attached shaders to the final program

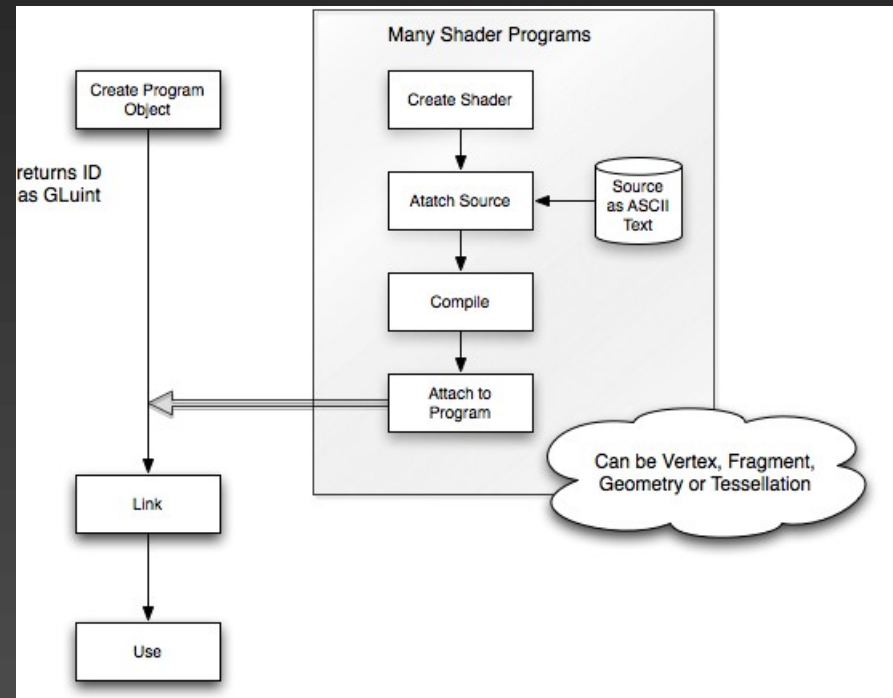
That means, that you can use single compiled shader in many different programs = shared libraries. DO check linker return code and linker log!!

## 7) Enable linked program

Since now the program will be used.

# Shader HOWTO

- 0) GLuint VS\_h, FS\_h, prog\_h;
- 1) VS\_h = glCreateShader(GL\_VERTEX\_SHADER);  
FS\_h = glCreateShader(GL\_FRAGMENT\_SHADER);
- 2) glShaderSource(VS\_h, 1, &VS\_string, NULL);  
glShaderSource(FS\_h, 1, &FS\_string, NULL);
- 3) glCompileShader(VS\_h);  
glCompileShader(FS\_h);
- 4) prog\_h = glCreateProgram();
- 5) glAttachShader(prog\_h, VS\_h);  
glAttachShader(prog\_h, FS\_h);
- 6) glLinkProgram(prog\_h);
- 7) glUseProgram(prog\_h);
- 8) ( glDeleteShader(), glDeleteProgram())



# GLSL Details

# Data types

- Simple
  - void, float, double int, uint, bool
    - limits and bit precision **NOT** specified!
- Compound
  - fp: vec2, vec3, vec4, mat2, mat3, mat4, dvec{2..4}, dmat{2..4}, mat2x3, mat3x2, ...
  - int: ivec2, ivec3, ivec4, uvec{2..4}
  - bool: bvec2, bvec3, bvec4
  - arrays (one-dimensional), structures
- Samplers
  - for accessing textures
  - sampler{1D..3D}, image{1D..3D}, sampler2Drect, samplerCube, samplerBuffer, ...
- Variable declaration as in C++
  - not only at the beginning of code block

# Variable initialization

```
float f = 10;
```

- **error** – STRONG TYPING (or warning in later vers.)

```
int i = 10;
```

```
float f = float(i);
```

- similar for bool(a), int(a), float(a)

```
vec3 accel = vec3(0.0, -9.81, 0.0)
```

```
vec3 white = vec3(1.0)
```

- **shortening, enlarging**

```
vec4 color_rgba;
```

```
vec3 color_rgb = vec3(color_rgba);
```

```
vec4 my_rgba = vec4( vec3(r,g,b), alpha);
```

# Matrix initialization

- Item enumeration

- 4, 9 or 16 items

- Diagonal matrix

mat3 diamat = mat3(1.0)

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$$

- **Columns** enumeration

vec3 col1 = vec3(1.0, 0.0, 0.0)

vec3 col2 = vec3(0.0, 1.0, 0.0)

vec3 col3 = vec3(0.0, 0.0, 1.0)

- mat3 diamat = mat3(col1,col2,col3)

# Vector items **swizzle**

- vector is like structure – items are accessible using „.“, usable to change items order
- three possibilities: .xyzw, .rgba, .stpq

```
vec4 v4;
```

```
v4.rgba //same as v4
```

```
v4.rgb //result is vec3
```

```
v4.b //result is scalar float
```

```
v4.xy //result is vec2
```

```
v4.xgba //error, items not from same set
```

```
v4.rrrr //result is vec4, items can repeat or change order
```



# GLSL Type Modifiers I/II

[ [smooth/flat] in/out/uniform/const] [float/vec3/mat4/...]

- „in“
  - Input to a shader stage
  - Usually internally as vec4 → group scalars together
  - „smooth in“ = „in“ = varying, **default**
    - variable for data transfer into fragment shader, only float (+vector, matrix)
    - in FS **automatically interpolated** in polygon including perspective  
eg.: in vec3 lightVec;
  - „flat in“
    - Non-interpolated input into FS
- „out“
  - Output from a shader to next pipeline stage

# GLSL Type Modifiers II/II

- „uniform“
  - set as a parameter by CPU application
  - constant in whole primitive and all shaders  
eg.: `uniform bool lightsOn;`
- „const“
  - constant value
- „buffer“
  - Data accessible by shader and CPU
- „shared“
  - Compute shaders, data shared in workgroup

# Accessing Uniforms - Statefull

- CPU can not write to GPU directly

- get **location** of the variable in the compiled source

`GLint location = glGetUniformLocation(GLuint prog, char * varName)`

- activate shader **program**

`glUseProgram(GLuint prog)`

- set **value** to **location**

`void glUniform{1|2|3|4}{f|i|ui}(GLint location, TYPE value1, TYPE value2, ... )`

`void glUniform{1|2|3|4}{f|i|ui}v(GLint location, GLsizei cnt, TYPE * values)`

`void glUniformMatrix{2|3|4|2x3|...}fv(GLint location, GLsizei cnt,  
GLboolean transpose, const float * values)`

```
// C++ program
int main()
{
    // ...
    glm::vec4 rgba = ...;
    // ...
    while (!glfwWindowShouldClose(window))
    {
        // ...
        glUseProgram(prog_ID);
        GLint h = glGetUniformLocation(prog_ID, "myrgba");
        glUniform4fv(h, 1, glm::value_ptr(rgba));
        // ...
    }
    // ...
}
```

```
#version 460 core
out vec4 color;

uniform vec4 myrgba;

void main( void ) {
    color = myrgba;
}
```

# Accessing Uniforms - Stateless

- CPU can not write to GPU directly
  - get **location** of the variable in the compiled source  
`GLint location = glGetUniformLocation( GLuint prog, char * varName)`
  - set **value** to **location** in **program**  
`void glProgramUniform{1|2|3|4}{f|i|ui}(GLuint program, GLint location, TYPE value1, TYPE value2, ... )`
  - `void glProgramUniform{1|2|3|4}{f|i|ui}v(GLuint program, GLint location, GLsizei cnt, TYPE * values)`
  - `void glProgramUniformMatrix{2|3|4|2x3|...}fv(GLuint program, GLint location, GLsizei cnt, GLboolean transpose, const float * values)`

```
// C++ program
int main()
{
    // ...
    glm::vec4 rgba = ...;
    // ...
    while (!glfwWindowShouldClose(window))
    {
        // ...
        GLint h = glGetUniformLocation(prog_ID, "myrgba");
        glProgramUniform4fv(prog_ID, h, 1, glm::value_ptr(rgba));
        // ...
    }
    // ...
}
```

```
#version 460 core
out vec4 color;

uniform vec4 myrgba;

void main( void ) {
    color = myrgba;
}
```

# Vector and matrix operations

- multiplication is overloaded, dimensions must match
- matrix multiply is not comutative!
  - `mat = mat*mat`
  - `vec = mat*vec`
- Examples
  - `vec = vec * vec // component-wise`
  - `vec = dot(vec, vec) // scalar product`
  - `vec = cross(vec, vec) // vector product`
  - `mat = matrixCompMult(mat,mat) // component-wise`

# Flow control

- Like standard C++
  - if – else
  - for, while, do – while
  - break, continue, functions + return
- In fragment shader
  - discard – discard fragment output and may (or may NOT) end shader execution

# Functions

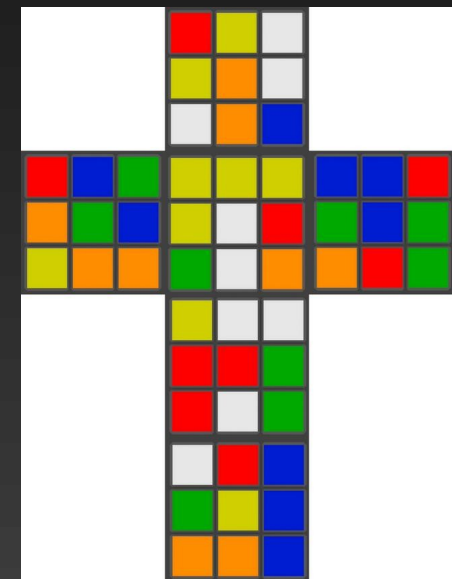
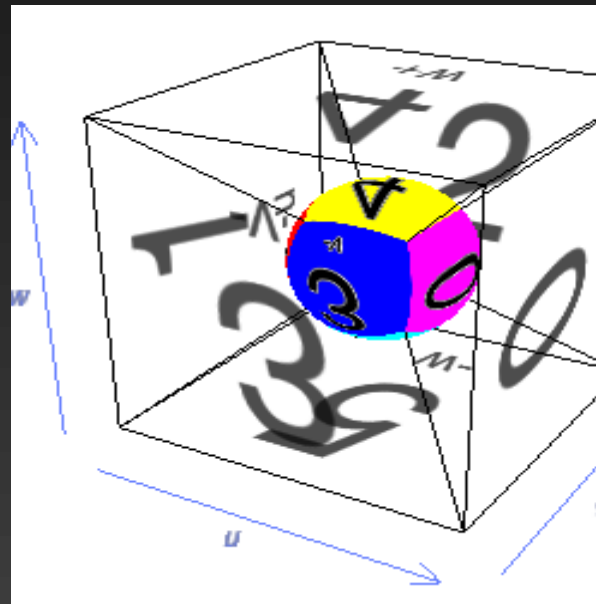
`return_type func( [in/out/inout] type param1, ... )`

- one value returned via `return_type`
- no pointers – **how to return more values?**
  - define & return structure
  - or with type modifier
    - **in** = input variable (default)
    - **const in** = constant input (read-only)
    - **out** = value copied out after execution finished (write-only)
    - **inout** = copied in and out (read-write)

# Accessing textures

- Mostly in FS (rarely in VS or others)
- `sampler{1D..3D}`

- `samplerCube`
  - cube maps



- `sampler1DShadow`, `sampler2DShadow`
  - shadow textures



# Accessing textures

- Bind texture unit and sampler

```
// C++ program
int main()
{
    // ...
    while (!glfwWindowShouldClose(window))
    {
        // ...

        GLint texSampler_h; //handle to „tex“ variable
        texSampler_h = glGetUniformLocation(FS_h, "tex");
        glUniform1i(texSampler_h, 2); //use GL_TEXTURE2

        // or glProgramUniform1i(...)

        // ...
    }
    // ...
}
```

```
#version 460 core

// FS

in vec2 texcoord;
out vec4 out_color;
uniform sampler2D tex;

void main(void) {
    out_color = texture(tex, in_texcoord);
}
```

# Summary

- allow full pipeline control
  - user specified vertex, tessellation, geometry & fragment program allow non-standard effects  
→ higher quality
  - all math & control must be programmed manually  
→ more complicated
  - unnecessary steps can be fully eliminated  
→ can be faster

# Create Shaders: example, no error checking

```
// create and use shaders
GLuint VS_h, FS_h, prog_h;

VS_h = glCreateShader(GL_VERTEX_SHADER);
FS_h = glCreateShader(GL_FRAGMENT_SHADER);

glShaderSource(VS_h, 1, &VS_string, NULL);
glShaderSource(FS_h, 1, &FS_string, NULL);

glCompileShader(VS_h);
glCompileShader(FS_h);

prog_h = glCreateProgram();

glAttachShader(prog_h, VS_h);
glAttachShader(prog_h, FS_h);

glLinkProgram(prog_h);

glUseProgram(prog_h);
```

# Shaders: with error checking

```
// load text file
#include <iostream>
#include <fstream>
#include <sstream>
#include <filesystem>

std::string textFileRead(const std::filesystem::path& fn) {
    std::ifstream file(fn);
    if (!file.is_open())
        throw std::runtime_error("Error opening file.\n");

    std::stringstream ss;
    ss << file.rdbuf();
    return ss.str();
}

// get shader compilation errors
std::string getShaderInfoLog(const GLuint obj) {
    int infologLength = 0;
    std::string s;
    glGetShaderiv(obj, GL_INFO_LOG_LENGTH, &infologLength);
    if (infologLength > 0) {
        std::vector<char> v(infologLength);
        glGetShaderInfoLog(obj, infologLength, NULL, v.data());
        s.assign(begin(v), end(v));
    }
    return s;
}

// get shader linker errors
std::string getProgramInfoLog(const GLuint obj) {
    int infologLength = 0;
    std::string s;
    glGetProgramiv(obj, GL_INFO_LOG_LENGTH, &infologLength);
    if (infologLength > 0) {
        std::vector<char> v(infologLength);
        glGetProgramInfoLog(obj, infologLength, NULL, v.data());
        s.assign(begin(v), end(v));
    }
    return s;
}
```

```
// create and use shaders
GLuint VS_h, FS_h, prog_h;

VS_h = glCreateShader(GL_VERTEX_SHADER);
FS_h = glCreateShader(GL_FRAGMENT_SHADER);

std::string VSrc = textFileRead(source_file_VS);
const char* VS_string = VSrc.c_str();
std::string FSrc = textFileRead(source_file_FS);
const char* FS_string = FSrc.c_str();

glShaderSource(VS_h, 1, &VS_string, NULL);
glShaderSource(FS_h, 1, &FS_string, NULL);

glCompileShader(VS_h);
{ // check compile result, display error (if any)
    GLint cpl_status;
    glGetShaderiv(VS_h, GL_COMPILE_STATUS, &cpl_status);
    if (cpl_status == GL_FALSE) {
        std::cerr << getShaderInfoLog(VS_h);
        throw std::runtime_error("Shader compile err.\n");
    }
}

// similar for FRAGMENT shader
glCompileShader(FS_h);
{ ... glGetShaderiv(FS_h, GL_COMPILE_STATUS, &cpl_status);
  ... }

prog_h = glCreateProgram();
glAttachShader(prog_h, VS_h);
glAttachShader(prog_h, FS_h);
glLinkProgram(prog_h);
{ // check link result, display error (if any)
    GLint status;
    glGetProgramiv(prog_h, GL_LINK_STATUS, &status);
    if (isLinked == GL_FALSE) {
        std::cerr << getProgramInfoLog(prog_h);
        throw std::runtime_error("Link err.\n");
    }
}

glUseProgram(prog_h);
```