OpenGL & GLSL

From OpenGL 1.0 to 4.5 with examples

Karol Gasiński

Graphic Software Engineer

August, 22 2014

Graphics Software Development & Validation

Abstract

This presentation will guide you through real time computer graphics, based on examples from AAA game titles, and Next-Gen engines, that are currently in development. You will learn modern OpenGL and GLSL as well as their evolution and different variations based on API history.

Some content presented in this slides is a result of my activities performed outside Intel, out of working hours, and without use of any Intel assets. It is not influencing my work and cannot be treated as a concurrency to Intel business. All information showed here represents my personal believes and don't represent position of Intel Corporation.

This presentation was based on my earlier lectures given during different conferences and events. Therefore similar content can be found in web.

Agenda

- Introduction
  - Rendering Pipeline
  - Program
  - Program Pipeline
  - Shader Components
  - Preprocessor, Profiles
  - Data Types, Precision
  - Built-in’s
  - Naming Conventions
  - Attributes– VA’s, VBO’s
  - Uniforms – Arrays, UBO’s, SSBO’s
  - Layouts
  - Interpolation
  - Conclusion

Introduction

- Fixed-Function pipeline
  - First programmable extensions:
    - EXT_texture_env_combine
    - NV_register_combiners (GeForce 256)
    - NV_vertex_program (GeForce 3)
    - NV_texture_shader (GeForce 4)
    - NV_texture_shader3 (GeForce 4)
    - NV_vertex_program2 (GeForce FX)
    - NV_fragment_program (GeForce FX)
  - ARB Assembly:
    - ARB_vertex_program
    - ARB_fragment_program

Since OpenGL 1.2.1

Since OpenGL 1.3

Introduction

- ARB Assembly example:

Bonus Content

```
!!ARBvp1.0

PARAM  mvp[4] = { state.matrix.mvp };          # IN  - Macierz projekcji (rzutowania) i transformacji
ATTRIB vPos   = vertex.position;               # IN  - Wspolrzedne wierzcholka
OUTPUT vOut   = result.position;               # OUT - Wynikowe wspolrzedne

DP4 vOut.x, vPos, mvp[0];
DP4 vOut.y, vPos, mvp[1];
DP4 vOut.z, vPos, mvp[2];
DP4 vOut.w, vPos, mvp[3];

ATTRIB tPos = vertex.texcoord[0];              # IN  - Wspolrzedne textury
OUTPUT tOut = result.texcoord[0];              # OUT - Wspolrzedne textury (wynik)

MOV tOut, tPos;

PARAM  world[4] = { state.matrix.modelview }; # IN  - Macierz swiata    (transformacji)
PARAM  vLight   = program.env[0];              # IN  - Wektor padania promieni slonecznych i ich natezenie (juz po transorm)
OUTPUT oColor   = result.color.primary;        # OUT - Wynikowe natezenie swiatla
TEMP   angle, norm;

DP3 norm.x, vertex.normal, world[0];           # Transformacja wektoru normalnego.
DP3 norm.y, vertex.normal, world[1];           # (obliczenie jego pozycji koncowej)
DP3 norm.z, vertex.normal, world[2];
DP3 norm.w, norm, norm;                        # Znormalizowanie tak otrzymanego wektoru
RSQ norm.w, norm.w;                            # do zakresu [0-1].
MUL norm, norm, norm.w;
DP3 angle.w, vLight, norm;                     # Obliczenie naswietlenia wierzcholka (padanie promieni)
MUL oColor.xyzw, vLight.w, angle.w;            # Obliczenie naswietlenia wierzcholka (jasnosc promieni)

END
```

Introduction

- ARB Assembly example:

Bonus Content

ARB Vertex Program 1.0

```
!!ARBvp1.0

PARAM  mvp[4] = { state.matrix.mvp };
ATTRIB vPos   = vertex.position;
OUTPUT vOut   = result.position;

DP4 vOut.x, vPos, mvp[0];
DP4 vOut.y, vPos, mvp[1];
DP4 vOut.z, vPos, mvp[2];
DP4 vOut.w, vPos, mvp[3];

ATTRIB tPos = vertex.texcoord[0];
OUTPUT tOut = result.texcoord[0];

MOV tOut, tPos;

PARAM  world[4] = { state.matrix.modelview };
PARAM  vLight   = program.env[0];
OUTPUT oColor   = result.color.primary;
TEMP   angle, norm;

DP3 norm.x, vertex.normal, world[0];
DP3 norm.y, vertex.normal, world[1];
DP3 norm.z, vertex.normal, world[2];
DP3 norm.w, norm, norm;
RSQ norm.w, norm.w;
MUL norm, norm, norm.w;
DP3 angle.w, vLight, norm;
MUL oColor.xyzw, vLight.w, angle.w;
END
```

GLSL 4.50

```
#version 450

in vec3 position;
in vec3 normal;
in vec2 coords;

out vec3 Onormal;
out vec2 Ocoords;

uniform mat4 mvp;
uniform mat4 mv;

void main()
{
gl_Position = mul( mvp, vec4( position, 1.0 ) );
Onormal     = normalize( mul( mv, vec4( normal, 1.0 ) ).xyz );
Ocoords     = coords;
}
```

Introduction

- ARB Assembly example:

Bonus Content

ARB Fragment Program 1.0

```
!!ARBfp1.0

ATTRIB lInt = fragment.color.primary;

TEMP iTex0, iTex1, cNoc, cDzien, inv;

TEX iTex0, fragment.texcoord[0], texture[0], 2D;
TEX iTex1, fragment.texcoord[0], texture[1], 2D;

SUB inv, 1.0, lInt;                # inv    = (1-lINT)
MUL cNoc, inv, iTex1;              # cNoc   = (1-lINT)*NOC
MUL cDzien, lInt, iTex0;           # cDzien =    lINT *DZIEN
ADD result.color, cNoc, cDzien;    # color  = (1-lINT)*NOC + lINT *DZIEN

END
```

GLSL 4.50

```
#version 450

in vec3 normal;
in vec2 coords;

out vec4 Ocolor;

uniform vec4 L;
uniform sampler2D tex1;
uniform sampler2D tex2;

void main()
{
vec4 day   = texture(tex1, coords);
vec4 night = texture(tex2, coords);
vec3 N     = normalize(normal);
vec4 color = clamp( max( dot( L, N ), 0.0 ) * light.w, 0.0, 1.0 );
Ocolor = mix( night, day, color.w );
}
```

Bonus Question:
What is the difference in behavior of this two shader sets?

Introduction

- Fixed-Function, ARB Assembly result:

Bonus Content

It displays rotating earth with incendiary and fading lights of cities at the dawn and sunset over them.

Introduction

- Programmable pipeline

  Deprecated:
  - Extensions
  - ARB Assembly

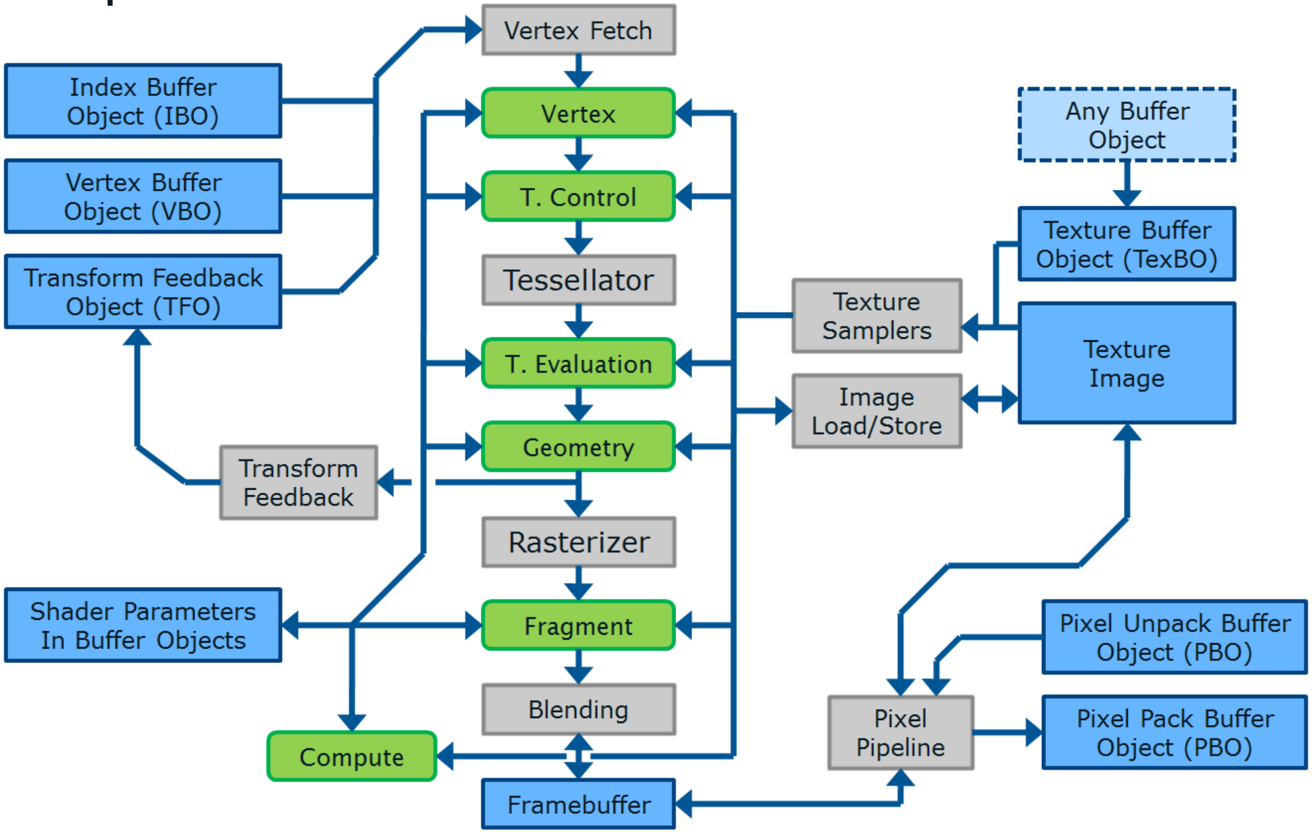
  Profiles:
  - Compatibility
  - Core
  - ES Compatibility

  More about features of each OpenGL version here.

OpenGL	GLSL	ESSL	ARB	Comp.	Core	ES
4.5	4.50	3.10	x	x	x	x
4.4	4.40	3.00	x	x	x	x
4.3	4.30	3.00	x	x	x	x
4.2	4.20	1.00	x	x	x	x
4.1	4.10	1.00	x	x	x	x
4.0	4.00		x	x	x	
3.3	3.30		x	x	x	
3.2	1.50		x	x	x	
3.1	1.40		x	x	~	
3.0	1.30		x	x		
2.1	1.20		x	x		
2.0	1.10		x	x		

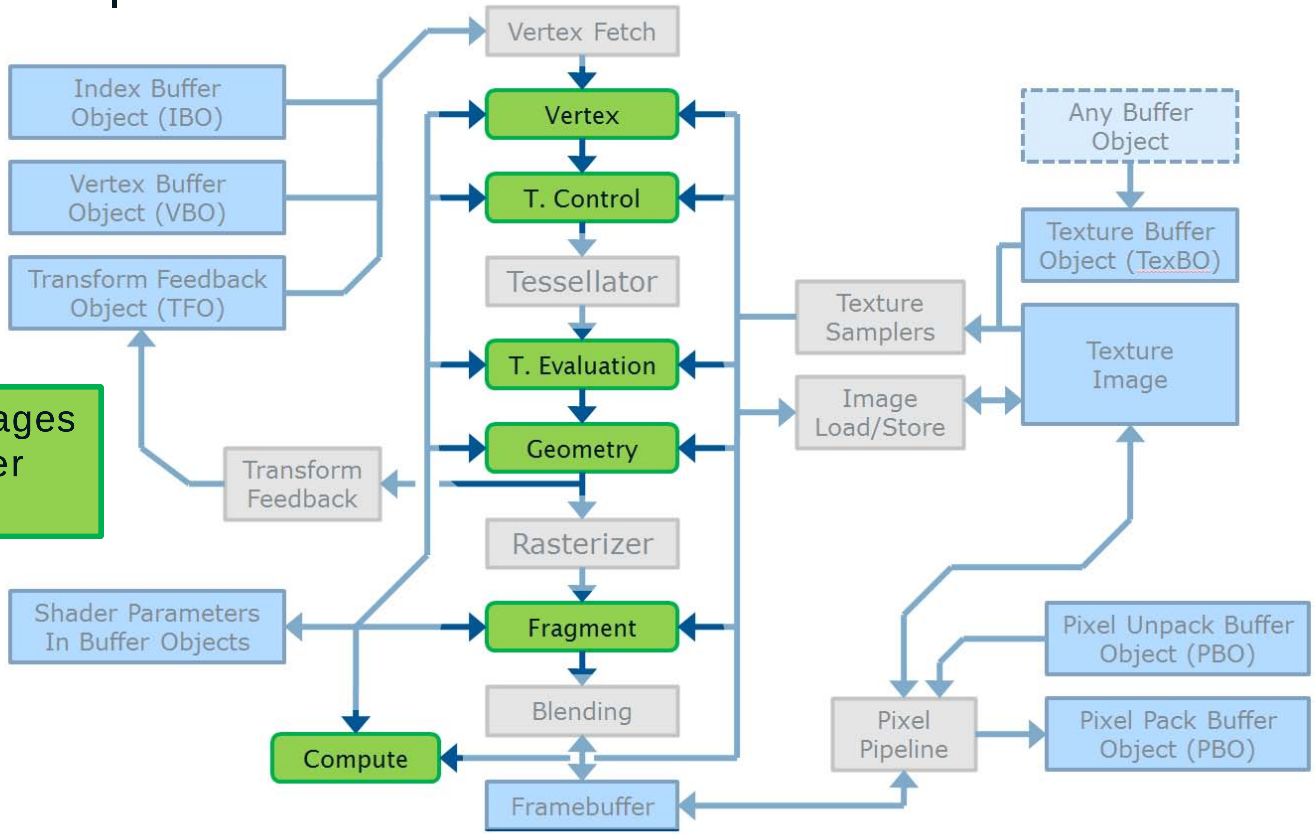
Rendering Pipeline

OpenGL 4.5 Pipeline:



Rendering Pipeline

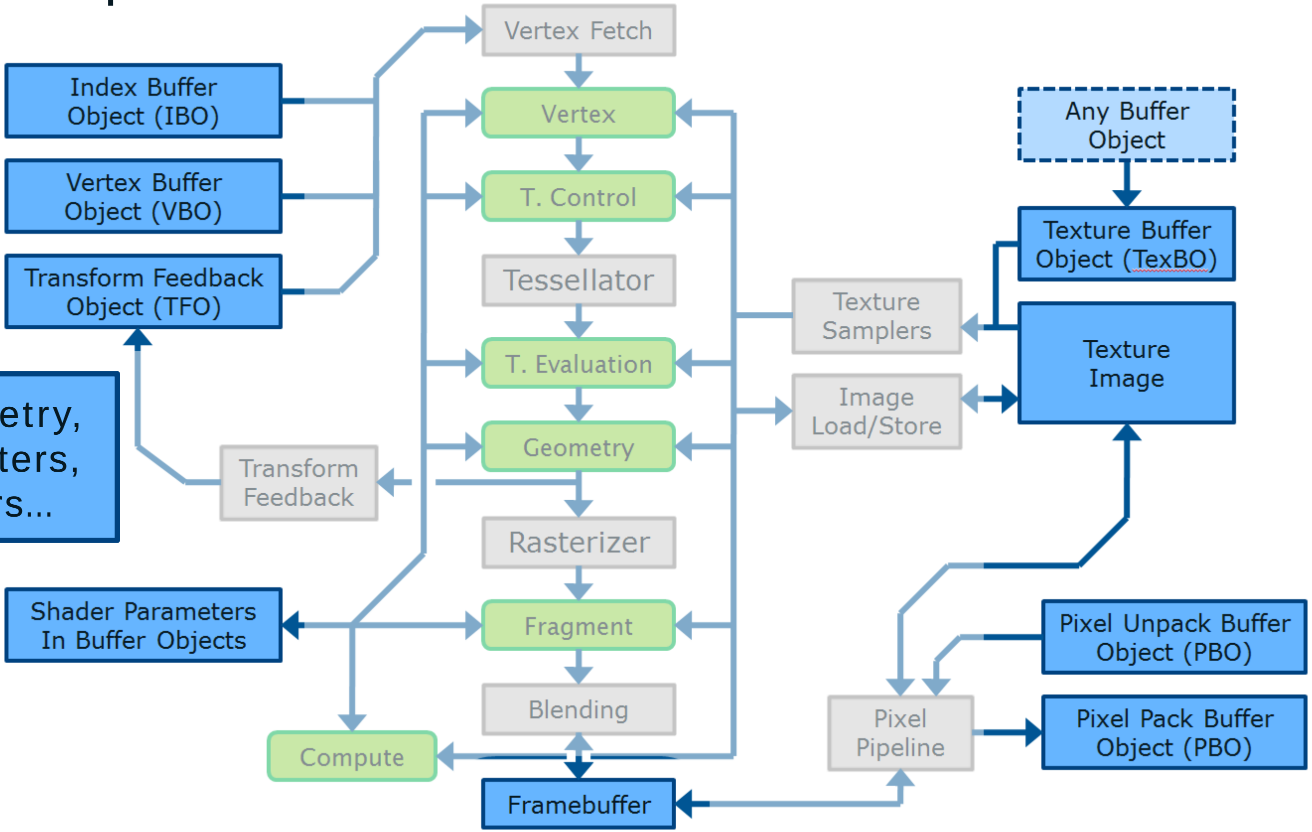
OpenGL 4.5 Pipeline:



Programmable Stages Executes Shader Programs

Rendering Pipeline

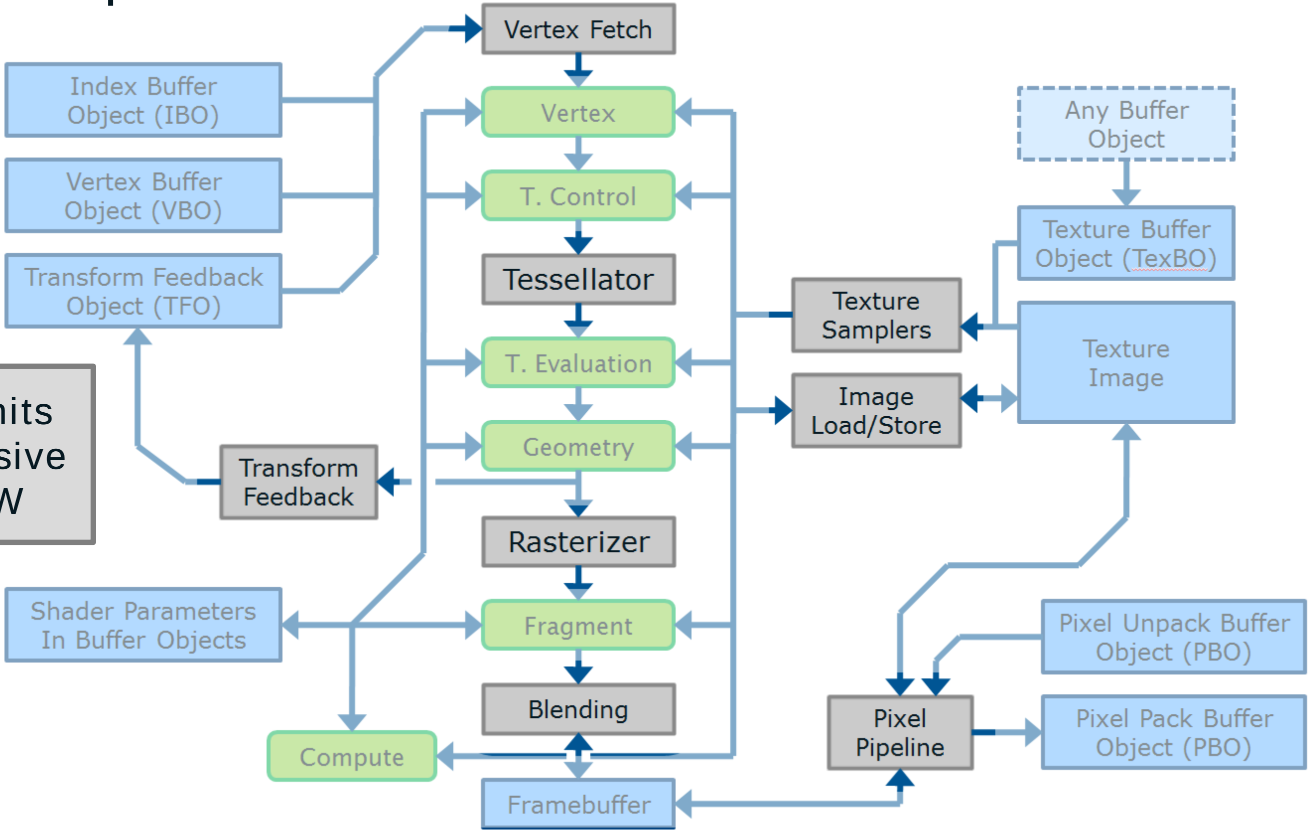
OpenGL 4.5 Pipeline:



Resources: Geometry, Textures, Parameters, Counters, Buffers…

Rendering Pipeline

OpenGL 4.5 Pipeline:



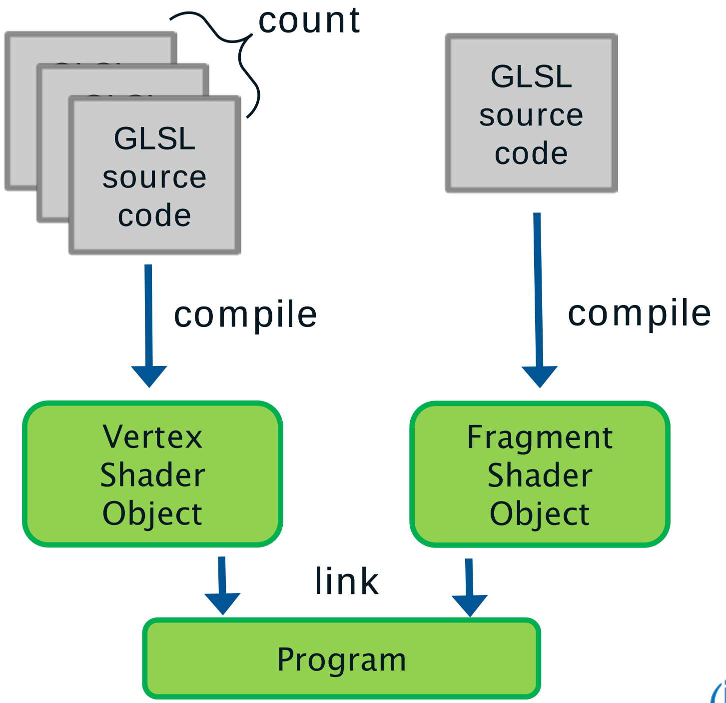
Fixed Function Units performing expensive operations in HW

Program

Since OpenGL 2.0

- *Program* – set of compiled shader binaries, ready for execution. Together they create logical pipeline.

```
char** sourceArray;

uint32 vertexId = glCreateShader( GL_VERTEX_SHADER );
glShaderSource( vertexId, count, sourceArray, NULL);
glCompileShader( vertexId );

// Link program
uint32 programId = glCreateProgram();
glAttachShader( programId, vertexId );
glAttachShader( programId, fragmentId );
glLinkProgram( programId );

// Cleanup intermediate resources
glDetachShader( programId, vertexId );
glDetachShader( programId, fragmentId );
glDeleteShader( vertexId );
glDeleteShader( fragmentId );
```



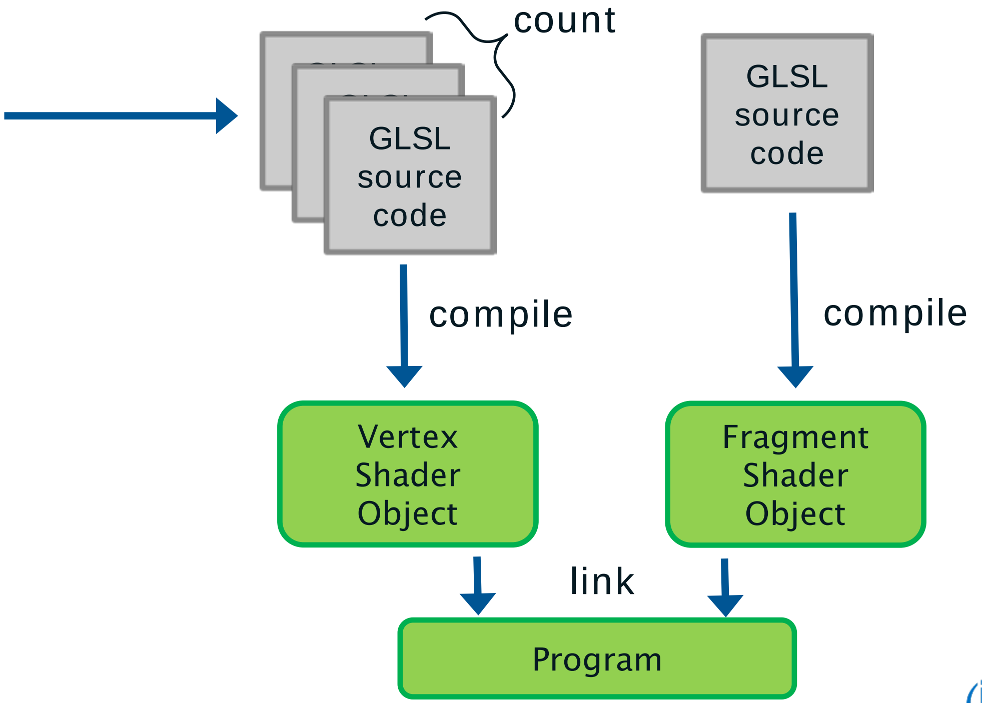
Program

- *Program* – set of compiled shader binaries, ready for execution. Together they create logical pipeline.

There can be multiple sources that will be combined together to create shader object.

But there can be only one main() function per shader object!

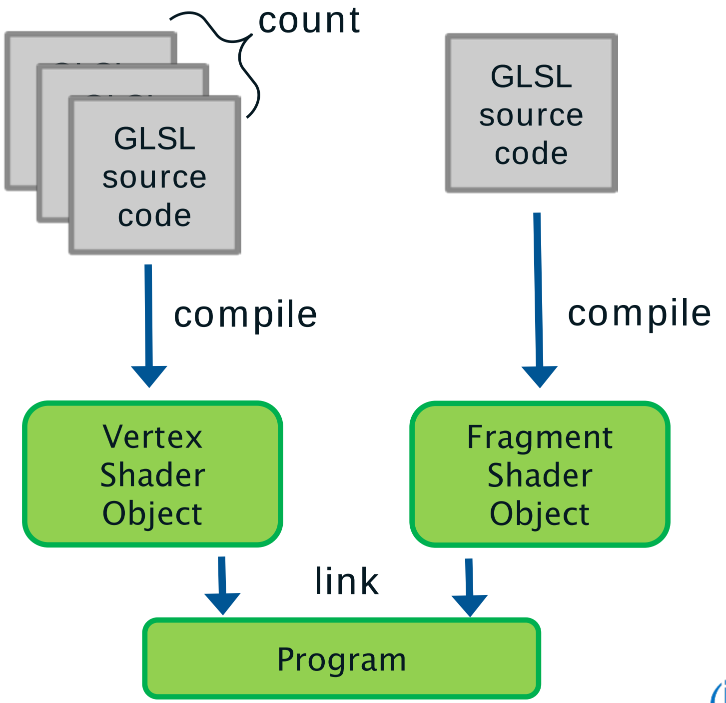
Shaders with different versions can be linked together (except of ES shaders)!



Program

- *Shader* compilation and *Program* linking process can be validated.

```
GLint ret, length;
GLsizei logged = 0;

// Validate shader compilation
glGetShaderiv(shaderId, GL_COMPILE_STATUS, &ret);
glGetShaderiv(shaderId, GL_INFO_LOG_LENGTH, &length);
if (ret == GL_FALSE) {
   char* log = new char[length];
   glGetShaderInfoLog(shaderId, length, &logged, log);
   cout << log; }

// Validate program linking
glGetProgramiv(programId, GL_LINK_STATUS, &ret);
glGetProgramiv(programId, GL_INFO_LOG_LENGTH, &length);
if (ret == GL_FALSE) {
   char* log = new char[length];
   glGetProgramInfoLog(programId, length, &logged, log);
   cout << log; }
```



Program Pipeline

Since OpenGL 4.1

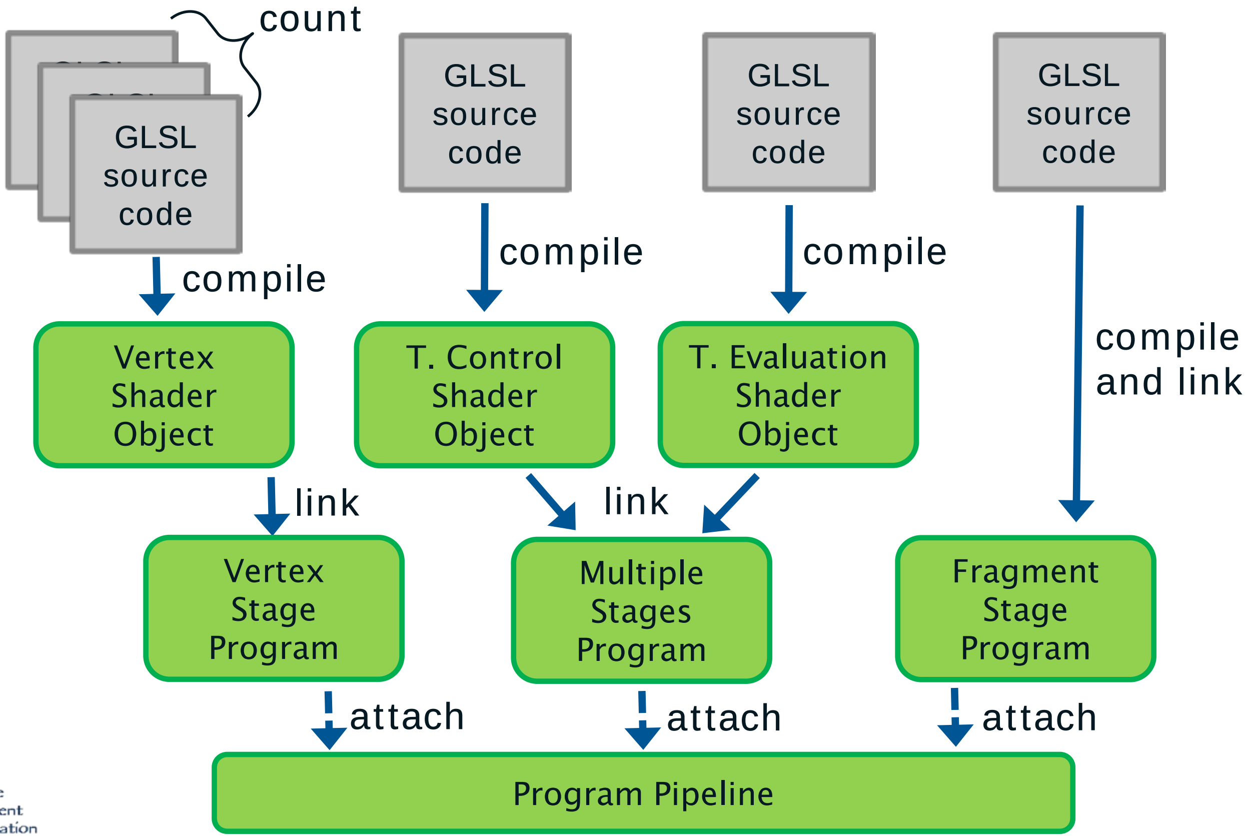
- *Pipeline* – allows dynamic creation of logic pipeline from different separable programs.

```
...
glProgramParameteri(multipleId, GL_PROGRAM_SEPARABLE, GL_TRUE);
glLinkProgram( multipleId );  // Link separable program containing multiple stages

// Compile and link separable program for one stage
char** sourceArray;
~~uint32 fragmentId = glCreateShader( GL_FRAGMENT_SHADER );~~
~~glShaderSource( fragmentId, count, sourceArray, NULL);~~
~~glCompileShader( fragmentId );~~
uint32 fragmentId = glCreateShaderProgramv(GL_FRAGMENT_SHADER, count, sourceArray);

// Attach separable programs to pipeline stages
uint32 pipelineId;
glGenProgramPipelines(1, &pipelineId);
glUseProgramStages(pipelineId, GL_VERTEX_SHADER_BIT |
                               GL_GEOMETRY_SHADER_BIT, multipleId);
glUseProgramStages(pipelineId, GL_FRAGMENT_SHADER_BIT, fragmentId);
```

ARB_separate_shader_objects

Program Pipeline

- Separable Programs linking:



Shader Components

- Basic components forming shader program:

Preprocessor macros

```
#version 110

attribute vec3 VertexPosition;
attribute vec3 VertexColor;
uniform   mat4 MVP;
varying   vec3 Color;

void main()
{
gl_Position = MVP * vec4( VertexPosition, 1.0 );
Color = VertexColor;
}
```

GLSL 1.10 Vertex Shader

Preprocessor macros – controls compilation process

Shader Components

- Basic components forming shader program:

Preprocessor macros

Input variables

```
#version 110

attribute vec3 VertexPosition;
attribute vec3 VertexColor;
uniform   mat4 MVP;
varying   vec3 Color;

void main()
{
gl_Position = MVP * vec4( VertexPosition, 1.0 );
Color = VertexColor;
}
```

GLSL 1.10 Vertex Shader

Input variables – provides per-vertex input data (only in geometry stages in different forms)

Shader Components

- Basic components forming shader program:

Preprocessor macros

Input variables

Parameters

```
#version 110

attribute vec3 VertexPosition;
attribute vec3 VertexColor;
uniform   mat4 MVP;
varying   vec3 Color;

void main()
{
gl_Position = MVP * vec4( VertexPosition, 1.0 );
Color = VertexColor;
}
```

GLSL 1.10 Vertex Shader

Parameters – data source, constant during execution

Shader Components

- Basic components forming shader program:

Preprocessor macros

Input variables

Parameters

Temporary results

```
#version 110

attribute vec3 VertexPosition;
attribute vec3 VertexColor;
uniform   mat4 MVP;
varying   vec3 Color;

void main()
{
gl_Position = MVP * vec4( VertexPosition, 1.0 );
Color = VertexColor;
}
```

GLSL 1.10 Vertex Shader

Temporary – data passed between shader stages

Shader Components

- Basic components forming shader program:

Preprocessor macros

Input variables

Parameters

Temporary results

Functions

```
#version 110

attribute vec3 VertexPosition;
attribute vec3 VertexColor;
uniform mat4 MVP;
varying vec3 Color;

void main()
{
gl_Position = MVP * vec4( VertexPosition, 1.0 );
Color = VertexColor;
}
```

GLSL 1.10 Vertex Shader

Functions – main shader function and user defined

Shader Components

- Basic components forming shader program:

Preprocessor macros

Input variables

Parameters

Temporary results

Functions

Built-in variables and functions

```
#version 110

attribute vec3 VertexPosition;
attribute vec3 VertexColor;
uniform mat4 MVP;
varying vec3 Color;

void main()
{
gl_Position = MVP * vec4( VertexPosition, 1.0 );
Color = VertexColor;
}
```

GLSL 1.10 Vertex Shader

Built-in's – variables and functions provided by API

Shader Components

- Basic components forming shader program:

Preprocessor macros

Data types

Input variables

Parameters

Temporary results

Functions

Built-in variables and functions

```
#version 110

attribute vec3 VertexPosition;
attribute vec3 VertexColor;
uniform   mat4 MVP;
varying   vec3 Color;

void main()
{
gl_Position = MVP * vec4( VertexPosition, 1.0 );
Color = VertexColor;
}
```

GLSL 1.10 Vertex Shader

Data types – scalar and vector data types

Preprocessor

- First shader line always need to be #version directive that tells which GLSL/ESSL version was used as a base.

```
#version 120
```

- Later profiles are introduced, which extends declaration to:

```
#version number profile
```

Since GLSL 1.50

- Available profiles:
  - core, compatibility, es
- Version declared as „100” refers to ESSL 1.00.
- Version declared as „300” refers to ESSL 3.00.
- Version declared as „310” refers to ESSL 3.10.

Since OpenGL 4.1

Since OpenGL 4.3

Since OpenGL 4.5

- Profile declaration is optional, by default shaders are compiled in core mode.

To use ESSL shaders, OpenGL Rendering Context need to be created in ES Compatibility mode.

Preprocessor

- Profiles support can be checked using #ifdef :

```
#ifdef GL_core_profile . . .
#ifdef GL_compatibility_profile . . .
#ifdef GL_es_profile . . .
```

- GLSL shaders can also use functionalities introduced in extensions:

```
#extension extension_name : behavior
#extension all : behavior
```

- There are four possible behavior types:
  - require – compilation fails if extension is not supported
  - enable – warns if extension is not supported, error on „all"
  - warn – warns if any use of this extension is detected
  - disable – behaves like the extension is not supported
- By default it is assumed that all extensions are disabled.

Preprocessor

- Preprocessor directives:
  - #, #define, #undef
  - #if, #ifdef, #ifndef, #else, #elif, #endif
  - #error, #pragma
  - #extension, #version
  - #line, defined
  - ## Since GLSL 1.30
- Predefined macros:
  - __LINE__, __FILE__, __VERSION__
- Macros and directives can be used to throw errors if unsupported functionality is detected during compile time.

It is not reccomended to use preprocessor directives, for e.g. #define to define your own data types, as most of mobile vendors don't support them very well.

Data types

- GLSL shaders support scalar and vector types. Vector types has up to 4 components (*x,y,z,w* or *r,g,b,a* – depending on usage). Matrix types are column-major supersets of vectors.

- All data processed by GPU's, if not stated otherwise use the same precision. Floating point variables are represented as 32bit IEEE 754 type.

- GLSL allows precision qualifiers (*lowp*, *mediump*, *highp*) but they are only for ES compatibility and have no guaranteed meaning, example:

```
lowp float color;
out mediump vec2 P;
lowp ivec2 foo(lowp mat3);
highp mat4 m;
```

Data Types

float, vec2, vec3, vec4	– floating point IEEE 754	Since GLSL 1.10
int, ivec2, ivec3, ivec4	– 32bit integers	
bool	– booleans	
bvec2, bvec3, bvec4		
mat2, mat3, mat4	– matrices 2x2, 3x3, 4x4	
mat2x3, mat2x4, mat3x2 mat3x4, mat4x2, mat4x3	– extended matrix modes	Since GLSL 1.20
uint	– unsigned integers	Since GLSL 1.30
uvec2, uvec3, uvec4		
double	– double precision	Since GLSL 4.00
dvec2, dvec3, dvec4 dmat2, dmat3, dmat4 dmat2x3, dmat2x4, dmat3x2 dmat3x4, dmat4x2, dmat4x3		

More about GLSL data types here.

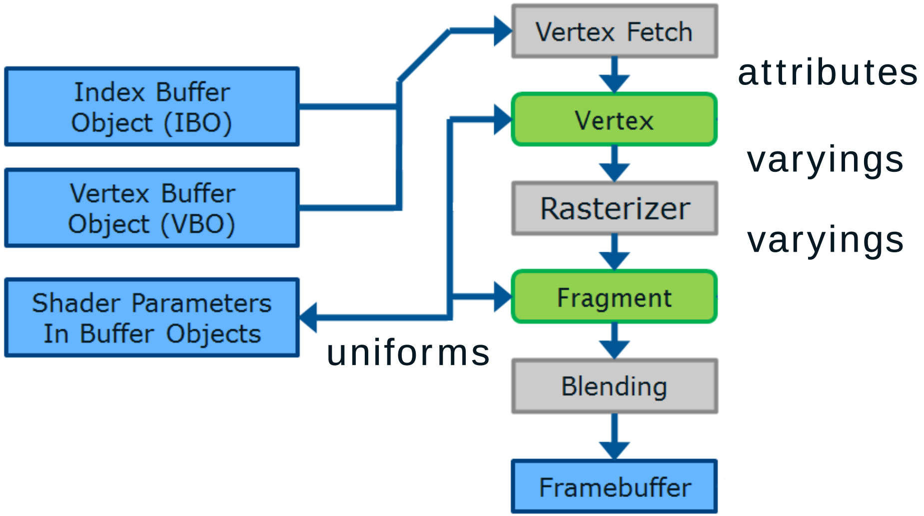
Built-in's

- With first versions of GLSL, OpenGL was still focused on Fixed-Function pipeline and shaders were treated as addition to it, instead of its replacement.
- This is why a lot of state information was available for shaders as built-in variables (mostly deprecated now).
- Most common ones:
  - gl_Vertex – input vertex position
  - gl_ModelViewProjectionMatrix – current MVP matrix
  - gl_Position – output position from Vertex Shader
  - gl_FragColor – output color from Fragment Shader
- There is much more built-in's for complete list check out latest GLSL spec. We will also mention several of them during this lecture.

Naming Conventions

GLSL 1.10 – 1.20

- Deprecated naming convention:
  - attribute – per-vertex value
  - varying – value passed between VS and FS that will be interpoated



GLSL 1.10 Vertex Shader

```
#version 110

attribute vec3 VertexPosition;
attribute vec3 VertexColor;
uniform   mat4 MVP;
varying   vec3 Color;

void main()
{
gl_Position = MVP * vec4( VertexPosition, 1.0 );
Color = VertexColor;
}
```

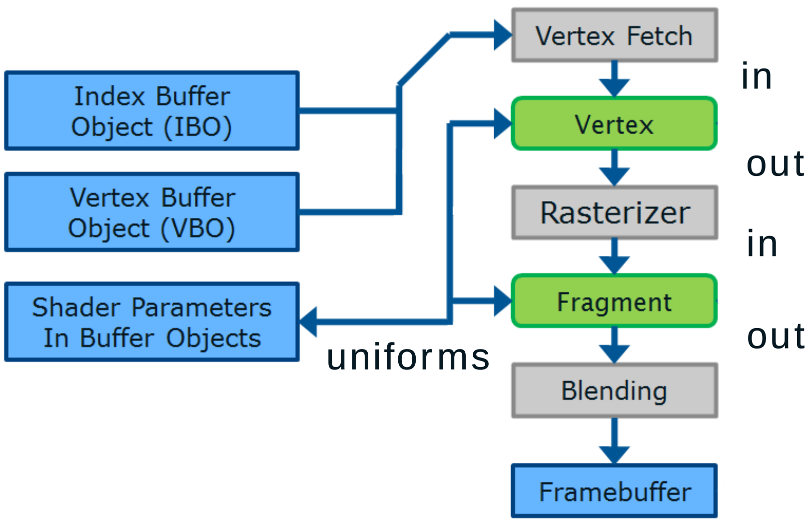
GLSL 1.10 Fragment Shader

```
#version 110

varying vec3 Color;

void main()
{
gl_FragColor = Color;
}
```

Naming Conventions

Since GLSL 1.30

- Current naming convention:
  - in – any input value (Read)
  - out – any output value (Write)



GLSL 4.50 Vertex Shader

```
#version 450

in      vec3 VertexPosition;
in      vec3 VertexColor;
uniform mat4 MVP;
out     vec3 Color;

void main()
{
gl_Position = MVP * vec4( VertexPosition, 1.0 );
Color = VertexColor;
}
```

GLSL 4.50 Fragment Shader

```
#version 450

in vec3 Color;
out vec4 color;

void main()
{
color = vec4( Color, 1.0 );
}
```

Attributes

- Each time Vertex Shader is going to be executed, it requires input data describing single vertex.
- These data can be stored in different locations, with different formats and access patterns.
- Fixed Function vertex assembly unit gathers, converts and unpacks all data used by shader based on information set by application.

GLSL 1.30 Vertex Shader

Attributes

```
#version 130

in vec3 VertexPosition;
in vec3 VertexColor;

. . .
```

Attributes

Since OpenGL 1.1

- *Vertex Array* – called VA is a memory buffer containing set of vertex data in application memory space (client space).

```
glBindBuffer( GL_ARRAY_BUFFER, vaId );

glEnableClientState( GL_VERTEX_ARRAY );
glVertexPointer(channels, type, stride, ptr );

glEnableClientState( GL_NORMAL_ARRAY );
glNormalPointer(channels, type, stride, ptr );

glClientActiveTexture(GL_TEXTURE0);
glEnableClientState(GL_TEXTURE_COORD_ARRAY );
glTexCoordPointer(channels, type, stride, ptr );
. . .
glClientActiveTexture(GL_TEXTURE7);
glEnableClientState(GL_TEXTURE_COORD_ARRAY );
glTexCoordPointer(channels, type, stride, ptr );
```

bind

GLSL 1.10 Vertex Shader

```
#version 110

...
gl_Vertex;         // On Nvidia location 0
gl_Normal;         // On Nvidia location 2
gl_Color;          // On Nvidia location 3
gl_SecondaryColor; // On Nvidia location 4
gl_FogCoord;       // On Nvidia location 5
gl_MultiTexCoord0; // On Nvidia location 8
...
gl_MultiTexCoord7; // On Nvidia location 15
```

Multitexturing is supported since OpenGL 1.2.1 as extension, in core is since OpenGL 1.3 and is deprecated since OpenGL 3.0.

Attributes

Since OpenGL 1.1

- *Vertex Array* – called VA is a memory buffer containing set of vertex data in application memory space (client space).

```
glBindBuffer( GL_ARRAY_BUFFER, vaId );

glEnableClientState( GL_VERTEX_ARRAY );
glVertexPointer(channels, type, stride, ptr );

glEnableClientState( GL_NORMAL_ARRAY );
glNormalPointer(channels, type, stride, ptr );

glClientActiveTexture(GL_TEXTURE0);
glEnableClientState(GL_TEXTURE_COORD_ARRAY );
glTexCoordPointer(channels, type, stride, ptr );
. . .
glClientActiveTexture(GL_TEXTURE7);
glEnableClientState(GL_TEXTURE_COORD_ARRAY );
glTexCoordPointer(channels, type, stride, ptr );
```

bind

GLSL 1.10 Vertex Shader

```
#version 110

...
gl_Vertex;         // On Nvidia location 0
gl_Normal;         // On Nvidia location 2
gl_Color;          // On Nvidia location 3
gl_SecondaryColor; // On Nvidia location 4
gl_FogCoord;       // On Nvidia location 5
gl_MultiTexCoord0; // On Nvidia location 8
...
gl_MultiTexCoord7; // On Nvidia location 15
```

Multitexturing is supported since OpenGL 1.2.1 as extension, in core is since OpenGL 1.3 and is deprecated since OpenGL 3.0.

Attributes

Since OpenGL 1.5

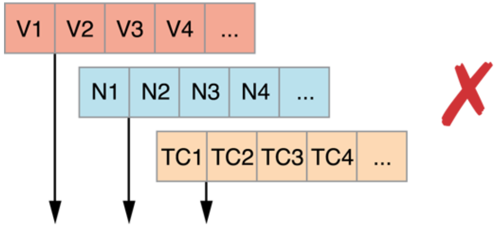
- *Vertex Buffer* – called VBO is a memory buffer similar to VA, but stored in driver memory space (server space).
- *Index Buffer* – called IBO is a buffer containing indexes of vertices in VBO from which primitives should be composed.

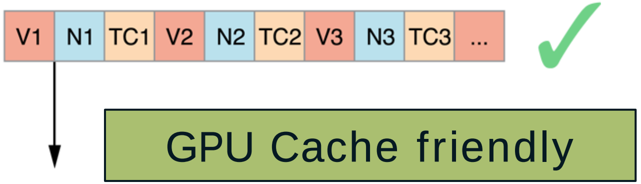
Replace all GL_ARRAY_BUFFER with GL_ELEMENT_ARRAY_BUFFER in example code to create IBO.

```
// Create VBO and send data first time
GLuint vboId;
glGenBuffers( 1, &vboId );
glBindBuffer( GL_ARRAY_BUFFER, vboId );
glBufferData( GL_ARRAY_BUFFER, size, ptr, GL_STATIC_DRAW );
glBindBuffer( GL_ARRAY_BUFFER, 0 );

// Later update VBO data in GPU
glBindBuffer( GL_ARRAY_BUFFER, vboId );
GLvoid* p = glMapBuffer( GL_ARRAY_BUFFER, GL_WRITE_ONLY );
memcpy(p, ptr, size);
glUnmapBuffer( GL_ARRAY_BUFFER );
```

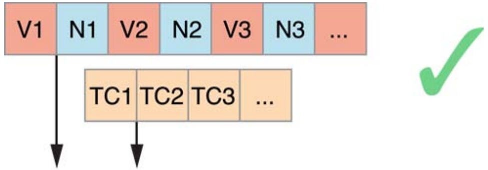
Attributes

- Data in VA and VBO can be placed in different ways, but it is adviced to always interleave attributes inside the buffer.





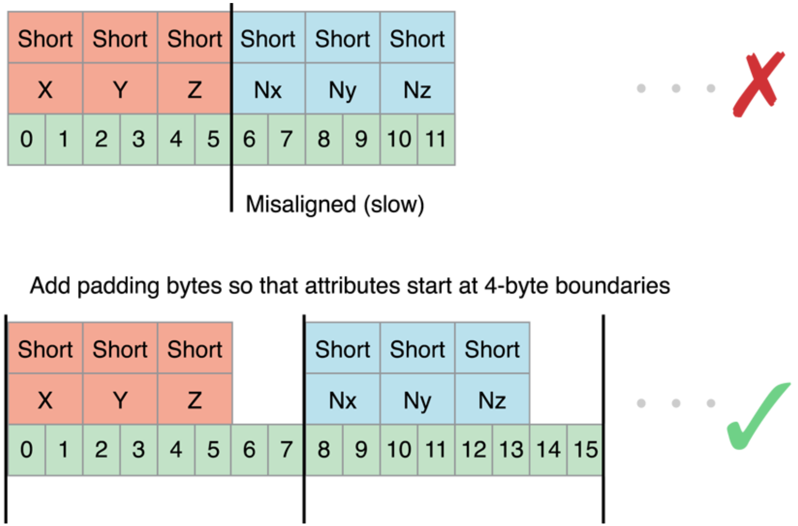
- If diferent attributes are used in different ways (or are optional) you can store them in separate VBO's. Game engines uses between 3 and 4 separate VBO's per mesh.



Attributes

- Example: Shadow casting objects need only position and UV data during shadow map generation. To save bandwith they are separated from normals and tangents used later.

VBO 0: Position, UV0
VBO 1: Normal, Tangent
VBO 2: Skin Weights, Indices
VBO 3: Decals UV1

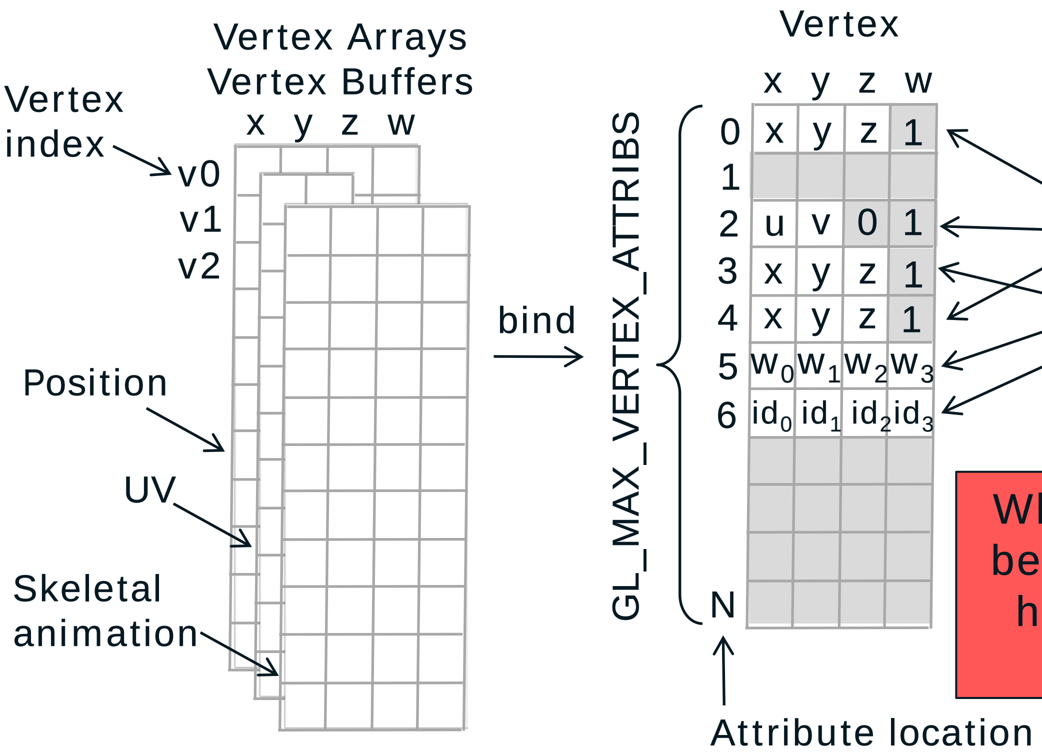


- It is also good practice to align data in buffers to 4bytes and whole vertice data to whole GPU cache line (32 bytes).

Attributes

Since OpenGL 2.0

- Vertex Shader can have up to GL_MAX_VERTEX_ATTRIBS attributes. Each attribute can have up to 4 components.



GLSL 4.50 Vertex Shader

```
#version 450

in   vec3 VertexPosition;
in   vec2 VertexUV0;
in   vec3 VertexNormal;
in   vec3 VertexTangent;
in   vec4 VertexSkinWeights;
in uvec4 VertexSkinBones;

...
```

When querying for attributes, they can be returned out of order. They can also have location index higher than their total count.

Attributes

- After program linking, attributes determined as active receive their locations. Only these attributes can be matched with Vertex Arrays and Vertex Buffer Objects.

```
GLint maxNameLength, attributesCount;

// Query list of active attributes - their names, locations, elements count and data types
glGetProgramiv( programId, GL_ACTIVE_ATTRIBUTE_MAX_LENGTH, &maxNameLength );
glGetProgramiv( programId, GL_ACTIVE_ATTRIBUTES, &attributesCount );
char* name = new char[maxNameLength];
for ( uint32 i=0; i<attributesCount; ++i )
    {
    GLenum type;    // Attribute type – vec2, mat4x3, uint. . .
    GLsizei written; // Length of current attribute name with null terminating sign
    GLint size;      // Elements count in array, otherwise 1
    glGetActiveAttrib( programId, i, maxNameLength, &written, &size, &type, name );
    GLint location = glGetAttribLocation( programId, name );

    // Add attribute parameter description to application structures . . .
    }
```

Attributes

```
for ( uint32 i=0; i<attributesCount; ++i )
    {
    // Find VBO that will be a source of data for this Attribute
    . . .
    // Bind VBO column to attribute
    glBindBuffer( GL_ARRAY_BUFFER, vboId );
    glEnableVertexAttribArray( attrib[i].location );
    glVertexAttribPointer( attrib[i].location,
                           attrib[i].channels,     // Channels count
                           attrib[i].type,         // OpenGL internal type
                           attrib[i].normalized, // Should input be normalized (to [0..1] or [-1..1] range) ?
                           rowSize,                 // Offset to next element, 0 means they are tightly packed
                           (const GLvoid*)offset ); // Pointer to VA in memory, or starting offset in VBO
    }
```

- To specify source of integer types use:

```
glVertexAttribIPointer( . . . );
```

Since OpenGL 3.0

- To specify source of double precision types use:

```
glVertexAttribLPointer( . . . );
```

Since OpenGL 4.1

Attributes

- Values in VBO's in most cases are compressed to save bandwith. There is plenty of different formats possible to use.

Since OpenGL 2.0

Channels	Type	Normalize	Int.	64bit	Format Description
1	GL_BYTE	GL_FALSE	-	-	float (8bit integer to float)
2	GL_BYTE	GL_FALSE	-	-	vec2 (8bit integer to float)
3	GL_BYTE	GL_FALSE	-	-	vec3 (8bit integer to float)
4	GL_BYTE	GL_FALSE	-	-	vec4 (8bit integer to float)
1	GL_SHORT	GL_FALSE	-	-	float (16bit integer to float)
2	GL_SHORT	GL_FALSE	-	-	vec2 (16bit integer to float)
3	GL_SHORT	GL_FALSE	-	-	vec3 (16bit integer to float)
4	GL_SHORT	GL_FALSE	-	-	vec4 (16bit integer to float)
1	GL_INT	GL_FALSE	-	-	float (32bit integer to float)
2	GL_INT	GL_FALSE	-	-	vec2 (32bit integer to float)
3	GL_INT	GL_FALSE	-	-	vec3 (32bit integer to float)

Attributes

Channels	Type	Normalize	Int.	64bit	Format Description
4	GL_INT	GL_FALSE	-	-	vec4 (32bit integer to float)
1	GL_UNSIGNED_BYTE	GL_FALSE	-	-	float (8bit unsigned integer to float)
2	GL_UNSIGNED_BYTE	GL_FALSE	-	-	vec2 (8bit unsigned integer to float)
3	GL_UNSIGNED_BYTE	GL_FALSE	-	-	vec3 (8bit unsigned integer to float)
4	GL_UNSIGNED_BYTE	GL_FALSE	-	-	vec4 (8bit unsigned integer to float)
1	GL_UNSIGNED_SHORT	GL_FALSE	-	-	float (16bit unsigned integer to float)
2	GL_UNSIGNED_SHORT	GL_FALSE	-	-	vec2 (16bit unsigned integer to float)
3	GL_UNSIGNED_SHORT	GL_FALSE	-	-	vec3 (16bit unsigned integer to float)
4	GL_UNSIGNED_SHORT	GL_FALSE	-	-	vec4 (16bit unsigned integer to float)
1	GL_UNSIGNED_INT	GL_FALSE	-	-	float (32bit unsigned integer to float)
2	GL_UNSIGNED_INT	GL_FALSE	-	-	vec2 (32bit unsigned integer to float)
3	GL_UNSIGNED_INT	GL_FALSE	-	-	vec3 (32bit unsigned integer to float)
4	GL_UNSIGNED_INT	GL_FALSE	-	-	vec4 (32bit unsigned integer to float)
1	GL_FLOAT	GL_FALSE	-	-	float (32bit float – no conversion)
2	GL_FLOAT	GL_FALSE	-	-	vec2 (32bit float – no conversion)

Attributes

Channels	Type	Normalize	Int.	64bit	Format Description
3	GL_FLOAT	GL_FALSE	-	-	vec3 (32bit float – no conversion)
4	GL_FLOAT	GL_FALSE	-	-	vec4 (32bit float – no conversion)
1	GL_BYTE	GL_TRUE	-	-	float (8bit int. normalized to [-1..1] float)
2	GL_BYTE	GL_TRUE	-	-	vec2 (8bit int. normalized to [-1..1] float)
3	GL_BYTE	GL_TRUE	-	-	vec3 (8bit int. normalized to [-1..1] float)
4	GL_BYTE	GL_TRUE	-	-	vec4 (8bit int. normalized to [-1..1] float)
1	GL_SHORT	GL_TRUE	-	-	float (16bit int. normalized to [-1..1] float)
2	GL_SHORT	GL_TRUE	-	-	vec2 (16bit int. normalized to [-1..1] float)
3	GL_SHORT	GL_TRUE	-	-	vec3 (16bit int. normalized to [-1..1] float)
4	GL_SHORT	GL_TRUE	-	-	vec4 (16bit int. normalized to [-1..1] float)
1	GL_INT	GL_TRUE	-	-	float (32bit int. normalized to [-1..1] float)
2	GL_INT	GL_TRUE	-	-	vec2 (32bit int. normalized to [-1..1] float)
3	GL_INT	GL_TRUE	-	-	vec3 (32bit int. normalized to [-1..1] float)
4	GL_INT	GL_TRUE	-	-	vec4 (32bit int. normalized to [-1..1] float)
1	GL_UNSIGNED_BYTE	GL_TRUE	-	-	float (8bit uint. normalized to [0..1] float)

Attributes

Channels	Type	Normalize	Int.	64bit	Format Description
2	GL_UNSIGNED_BYTE	GL_TRUE	-	-	vec2 (8bit uint. normalized to [0..1] float)
3	GL_UNSIGNED_BYTE	GL_TRUE	-	-	vec3 (8bit uint. normalized to [0..1] float)
4	GL_UNSIGNED_BYTE	GL_TRUE	-	-	vec4 (8bit uint. normalized to [0..1] float)
1	GL_UNSIGNED_SHORT	GL_TRUE	-	-	float (16bit uint. normalized to [0..1] float)
2	GL_UNSIGNED_SHORT	GL_TRUE	-	-	vec2 (16bit uint. normalized to [0..1] float)
3	GL_UNSIGNED_SHORT	GL_TRUE	-	-	vec3 (16bit uint. normalized to [0..1] float)
4	GL_UNSIGNED_SHORT	GL_TRUE	-	-	vec4 (16bit uint. normalized to [0..1] float)
1	GL_UNSIGNED_INT	GL_TRUE	-	-	float (32bit uint. normalized to [0..1] float)
2	GL_UNSIGNED_INT	GL_TRUE	-	-	vec2 (32bit uint. normalized to [0..1] float)
3	GL_UNSIGNED_INT	GL_TRUE	-	-	vec3 (32bit uint. normalized to [0..1] float)
4	GL_UNSIGNED_INT	GL_TRUE	-	-	vec4 (32bit uint. normalized to [0..1] float)

Attributes

Since OpenGL 3.0

Channels	Type	Normalize	Int.	64bit	Format Description
1	GL_HALF_FLOAT	GL_FALSE	-	-	float (16bit half-float to float)
2	GL_HALF_FLOAT	GL_FALSE	-	-	vec2 (16bit half-float to float)
3	GL_HALF_FLOAT	GL_FALSE	-	-	vec3 (16bit half-float to float)
4	GL_HALF_FLOAT	GL_FALSE	-	-	vec4 (16bit half-float to float)
1	GL_BYTE	GL_FALSE	Y	-	int (8bit to 32bit integer)
2	GL_BYTE	GL_FALSE	Y	-	ivec2 (8bit to 32bit integer)
3	GL_BYTE	GL_FALSE	Y	-	ivec3 (8bit to 32bit integer)
4	GL_BYTE	GL_FALSE	Y	-	ivec4 (8bit to 32bit integer)
1	GL_SHORT	GL_FALSE	Y	-	int (16bit to 32bit integer)
2	GL_SHORT	GL_FALSE	Y	-	ivec2 (16bit to 32bit integer)
3	GL_SHORT	GL_FALSE	Y	-	ivec3 (16bit to 32bit integer)
4	GL_SHORT	GL_FALSE	Y	-	ivec4 (16bit to 32bit integer)
1	GL_INT	GL_FALSE	Y	-	int (32bit integer – no conversion)
2	GL_INT	GL_FALSE	Y	-	ivec2 (32bit integer – no conversion)
3	GL_INT	GL_FALSE	Y	-	ivec3 (32bit integer – no conversion)

Attributes

Channels	Type	Normalize	Int.	64bit	Format Description
4	GL_INT	GL_FALSE	Y	-	ivec4 (32bit integer – no conversion)
1	GL_UNSIGNED_BYTE	GL_FALSE	Y	-	uint (8bit to 32bit unsigned integer)
2	GL_UNSIGNED_BYTE	GL_FALSE	Y	-	uvec2 (8bit to 32bit unsigned integer)
3	GL_UNSIGNED_BYTE	GL_FALSE	Y	-	uvec3 (8bit to 32bit unsigned integer)
4	GL_UNSIGNED_BYTE	GL_FALSE	Y	-	uvec4 (8bit to 32bit unsigned integer)
1	GL_UNSIGNED_SHORT	GL_FALSE	Y	-	uint (16bit to 32bit unsigned integer)
2	GL_UNSIGNED_SHORT	GL_FALSE	Y	-	uvec2 (16bit to 32bit unsigned integer)
3	GL_UNSIGNED_SHORT	GL_FALSE	Y	-	uvec3 (16bit to 32bit unsigned integer)
4	GL_UNSIGNED_SHORT	GL_FALSE	Y	-	uvec4 (16bit to 32bit unsigned integer)
1	GL_UNSIGNED_INT	GL_FALSE	Y	-	uint (32bit uns. integer – no conversion)
2	GL_UNSIGNED_INT	GL_FALSE	Y	-	uvec2 (32bit uns. integer – no conversion)
3	GL_UNSIGNED_INT	GL_FALSE	Y	-	uvec3 (32bit uns. integer – no conversion)
4	GL_UNSIGNED_INT	GL_FALSE	Y	-	uvec4 (32bit uns. integer – no conversion)

Attributes

Since OpenGL 3.3

Channels	Type	Normalize	Int.	64bit	Format Description
GL_BGRA	GL_INT_ 2_10_10_10_REV	GL_TRUE	-	-	vec4 (A2R10G10B10 [-1..1] normalized) D3D comaptiblity format ZYXW see ARB_vertex_array_bgra
GL_BGRA	GL_UNSIGNED_INT_ 2_10_10_10_REV	GL_TRUE	-	-	vec4 (A2R10G10B10 [0..1] normalized) D3D comaptiblity format ZYXW see ARB_vertex_array_bgra

Since OpenGL 4.1

Channels	Type	Normalize	Int.	64bit	Format Description
1	GL_DOUBLE	GL_FALSE	-	Y	double (double – no conversion)
2	GL_DOUBLE	GL_FALSE	-	Y	dvec2 (double – no conversion)
3	GL_DOUBLE	GL_FALSE	-	Y	dvec3 (double – no conversion)
4	GL_DOUBLE	GL_FALSE	-	Y	dvec4 (double – no conversion)

Uniforms

- *uniform* – standard shader program parameter provided by application. It is constant through execution time (in opposite to per-vertex values)
  - Uniform variables in program are accessed through their location, which first need to be queried (or set explicitly).

```
GLfloat mvp[16]; // Model-View-Projection matrix
GLint    mvpLocation;

// Get Uniform location in shader (once at init)
mvpLocation = glGetUniformLocation(programId, „MVP”);

// Update „MVP” value (each time it changes)
glUniformMatrix4fv(mvpLocation, count, GL_FALSE, mvp);

. . .
// Draw call
```

GLSL 1.10 Vertex Shader

```
#version 110

attribute vec3 VertexPosition;
attribute vec3 VertexColor;
uniform   mat4 MVP;
varying   vec3 Color;

void main()
{
gl_Position = MVP * vec4( VertexPosition, 1.0 );
Color = VertexColor;
}
```

Uniforms

- Variables can be organized into arrays. In such situation application can query location of given element in array.
- It is also possible to upload several elements at the same time.

```
GLfloat bones[32*16]; // Skeletal animation matrices
GLint    location, fifth;

location = glGetUniformLocation(programId, „bones”);
fifth      = glGetUniformLocation(programId, „bones[4]”);

// Update all bones at once
glUniformMatrix4fv(location, 32, GL_FALSE, bones);
// Upload fifth bone in array
glUniformMatrix4fv(fifth, 1, GL_FALSE, &(bones[4*16]) );
```

GLSL 1.10 Vertex Shader

```
#version 110

..
uniform   mat4 bones[32];
..

void main()
{
..
}
```

Default name can be returned as „bones[0]”

Since GLSL 4.30

- In latest GLSL versions it is possible to declare and query multidimensional arrays.

ARB_arrays_of_arrays

Uniforms

- After program linking, uniforms determined as active receive their locations. Only these variables can be updated.

```
GLint maxNameLength, uniformsCount;

// Query list of active uniforms - their names, locations, elements count and data types
glGetProgramiv( programId, GL_ACTIVE_UNIFORM_MAX_LENGTH, &maxNameLength );
glGetProgramiv( programId, GL_ACTIVE_UNIFORMS, &uniformsCount );
char* name = new char[maxNameLength];
for ( uint32 i=0; i<uniformsCount; ++i )
    {
    GLenum type;    // Uniform type – vec2, mat4x3, sampler2D. . .
    GLsizei written; // Length of current uniform name with null terminating sign
    GLint size;       // Elements count in array, otherwise 1
    glGetActiveUniform( programId, i, maxNameLength, &written, &size, &type, name );
    GLint location = glGetUniformLocation( programId, name );

    // Add uniform parameter description to application structures . . .
    }
```

Uniform locations doesn't need to match their query index in glGetActiveUniform call.

Uniforms

- Uniform data types update functions:

Since GLSL 1.10

GLSL type	API enum	API Update function	Notes
float vec2 vec3 vec4	GL_FLOAT GL_FLOAT_VEC2 GL_FLOAT_VEC3 GL_FLOAT_VEC4	glUniform1fv glUniform2fv glUniform3fv glUniform4fv	
int ivec2 ivec3 ivec4	GL_INT GL_INT_VEC2 GL_INT_VEC3 GL_INT_VEC4	glUniform1iv glUniform2iv glUniform3iv glUniform4iv	
bool bvec2 bvec3 bvec4	GL_BOOL GL_BOOL_VEC2 GL_BOOL_VEC2 GL_BOOL_VEC2	glUniform1{f,i,ui}v glUniform2{f,i,ui}v glUniform3{f,i,ui}v glUniform4{f,i,ui}v	Any 0 or 0.0f will be converted to FALSE. All other values will be treated as TRUE.
mat2 mat3 mat4	GL_FLOAT_MAT2 GL_FLOAT_MAT3 GL_FLOAT_MAT4	glUniformMatrix2fv glUniformMatrix3fv glUniformMatrix4fv	

Uniforms

- Uniform data types update functions:

Since GLSL 1.20

GLSL type	API type	API Update function	Notes
mat2x3 mat2x4 mat3x2 mat3x4 mat4x2 mat4x3	GL_FLOAT_MAT2x3 GL_FLOAT_MAT2x4 GL_FLOAT_MAT3x2 GL_FLOAT_MAT3x4 GL_FLOAT_MAT4x2 GL_FLOAT_MAT4x3	glUniformMatrix2x3fv glUniformMatrix2x4fv glUniformMatrix3x2fv glUniformMatrix3x4fv glUniformMatrix4x2fv glUniformMatrix4x3fv	You can specify if matrix should be inverted to match column major order (useful for DirectX conversions).

Since GLSL 1.30

GLSL type	API type	API Update function	Notes
uint uvec2 uvec3 uvec4	GL_UNSIGNED_INT GL_UNSIGNED_INT_VEC2 GL_UNSIGNED_INT_VEC3 GL_UNSIGNED_INT_VEC4	glUniform1iv glUniform2iv glUniform3iv glUniform4iv	

Uniforms

- Uniform data types update functions:

Since GLSL 4.00

GLSL type	API type	API Upload function	Notes
double dvec2 dvec3 dvec4	GL_DOUBLE GL_DOUBLE_VEC2 GL_DOUBLE_VEC3 GL_DOUBLE_VEC4	glUniform1dv glUniform2dv glUniform3dv glUniform4dv	
dmat2 dmat3 dmat4	GL_DOUBLE_MAT2 GL_DOUBLE_MAT3 GL_DOUBLE_MAT4	glUniformMatrix2dv glUniformMatrix3dv glUniformMatrix4dv	
dmat2x3 dmat2x4 dmat3x2 dmat3x4 dmat4x2 dmat4x3	GL_DOUBLE_MAT2x3 GL_DOUBLE_MAT2x4 GL_DOUBLE_MAT3x2 GL_DOUBLE_MAT3x4 GL_DOUBLE_MAT4x2 GL_DOUBLE_MAT4x3	glUniformMatrix2x3dv glUniformMatrix2x4dv glUniformMatrix3x2dv glUniformMatrix3x4dv glUniformMatrix4x2dv glUniformMatrix4x3dv	You can specify if matrix should be inverted to match column major order (useful for DirectX conversions).

Uniform Blocks

Since GLSL 1.40

- Uniform blocks were introduced to minimize count of API function calls needed to update all active uniforms.
- Each block groups several uniforms together. Uniform Buffer Object can be matched with such block to store their values.
- Uniforms outside blocks lay in „default uniform block”.
- Uniform block can have optional instance name. Accessing such uniforms require instance name prefix.

```
#version 140

uniform BlobSettings
        {
        vec4  InnerColor;
        vec4  OuterColor;
        float InnerRadius;
        float OuterRadius;
        };

vec4 color = InnerColor;
```

```
#version 140

uniform BlobSettings
        {
        vec4  InnerColor;
        vec4  OuterColor;
        float InnerRadius;
        float OuterRadius;
        } Blob;

vec4 color = Blob.InnerColor;
```

More about interfacing blocks here.

Uniform Blocks

- For Uniform Buffer Objects and other buffers specified later, binding has been extended from one point to a table:
  - glBindBuffer(GL_UNIFORM_BUFFER, uboId);
  - glBindBufferBase(GL_UNIFORM_BUFFER, bindingId, uboId);
- This way, it is possible to use more than one UBO at the same time (it is also cleaner that using multiple VBO's).
- Maximum number of table enties can be queried using:
  - glGetIntegerv(GL_MAX_UNIFORM_BUFFER_BINDINGS, &max);
- Binding points are shared by all programs withing context.
- Each stage has it's own limit of uniform blocks. These blocks need to be bound to matching index to connect with UBO.
- It also allows binding UBO's and blocks for several different programs only once, which speeds up program switching.

More info here.

Uniform Blocks

```
GLint blockId = glGetUniformBlockIndex( programId, „BlobSettings” );

// Query uniforms offsets in buffer
GLchar* names[count] = { „InnerColor”, „OuterColor”, „InnerRadius”, OuterRadius” }
GLuint indices[count];
GLint offset[count];
glGetUniformIndices( programId, count, names, indices );
glGetActiveUniformsiv( programId, count, indices, GL_UNIFORM_OFFSET, offset );

// Save values to temporary buffer
GLint size;
glGetActiveUniformBlockiv( programId, blockId, GL_UNIFORM_BLOCK_DATA_SIZE, &size );
GLuint* ptr = new GLuint[ size ];
memcpy(ptr + offset[0], innerColor, 4 * sizeof( GLfloat ) );
. . .
// Sent data to UBO
GLint uboId;
glGenBuffers( 1, &uboId );
glBindBuffer( GL_UNIFORM_BUFFER, uboId );
glBufferData( GL_UNIFORM_BUFFER, size, ptr, GL_DYNAMIC_DRAW );

// Bind UBO and uniform block to common binding point from table
GLuint bindingId = [0 . . GL_MAX_UNIFORM_BUFFER_BINDINGS];
glBindBufferBase( GL_UNIFORM_BUFFER, bindingId, uboId );
glUniformBlockBinding(programId, blockId, bindingId);
```

UBO max sizes:
64KB AMD, Nvidia
16KB Intel

Shaders can only
Read UBO’s.

Shader Storage Blocks

Since GLSL 4.30

- Shader Storage Blocks in opposite to Uniform Blocks has no size limit (min. 16MB) and can be written to by shaders.
- Maximum number of SSBO table enties can be queried:
  - glGetIntegerv(GL_MAX_SHADER_STORAGE_BUFFER_BINDINGS, &max);
- Programmable stages has their own limits of Storage Blocks that can be declared and bound to the table:
  - GL_MAX_COMBINED_SHADER_STORAGE_BLOCKS
  - . . .
  - GL_MAX_GEOMETRY_SHADER_STORAGE_BLOCKS
  - GL_MAX_FRAGMENT_SHADER_STORAGE_BLOCKS
  - GL_MAX_COMPUTE_SHADER_STORAGE_BLOCKS

```
#version 430

buffer HugeBlock
       {
       mat4  mvp[];
       };
...
```

- Last variable in Storage Block can be unsized array. It allows shader execution with dynamic amount of instances data.

More info here.

Shader Storage Blocks

```
GLint blockId = glGetProgramResourceIndex( programId, GL_SHADER_STORAGE_BLOCK, „HugeBlock” );

// Sent data to SSBO first time
GLint size = 8192*16*4;
GLfloat ptr[8192*16]; // 8192 matrices in 512KB of local memory
GLuint ssboId;
glGenBuffers( 1, &ssboId );
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssboId );
glBufferData(GL_SHADER_STORAGE_BUFFER, size, ptr, GL_DYNAMIC_COPY);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);

// Bind SSBO and storage block to common binding point from table
GLuint bindingId = [0 . . GL_MAX_SHADER_STORAGE_BUFFER_BINDINGS];
glBindBufferBase( GL_SHADER_STORAGE_BUFFER, bindingId, ssboId );
glShaderStorageBlockBinding( programId, blockId, bindingId );

// Later update SSBO data in GPU
glBindBuffer( GL_SHADER_STORAGE_BUFFER, ssboId );
GLvoid* p = glMapBuffer( GL_SHADER_STORAGE_BUFFER, GL_WRITE_ONLY );
memcpy(p, ptr, size);
glUnmapBuffer( GL_SHADER_STORAGE_BUFFER );
```

ARB_shader_storage_buffer_object

Textures

- Textures are other type of resources accessible from GLSL. They are accessed through samplers which specify how raw data is processed before submiting it to shader.

Texture Type	GL enum	OpenGL support	OpenGL ES support
1D	GL_TEXTURE_1D	1.1	
1D Array	GL_TEXTURE_1D_ARRAY	3.0	
2D	GL_TEXTURE_2D	1.0	1.0
2D Array	GL_TEXTURE_2D_ARRAY	3.0	3.0
2D Rectangle	GL_TEXTURE_RECTANGLE	3.1	
2D Multisample	GL_TEXTURE_2D_MULTISAMPLE	3.2	3.1
2D Multisample Array	GL_TEXTURE_2D_MULTISAMPLE_ARRAY	3.2	
3D	GL_TEXTURE_3D	1.2	3.0
Buffer	GL_TEXTURE_BUFFER	3.1	
Cube Map	GL_TEXTURE_CUBE_MAP	1.3	2.0
Cube Map Array	GL_TEXTURE_CUBE_MAP_ARRAY	4.0	

Textures

- Textures are other type of resources accessible from GLSL. They are accessed through samplers which specify how raw data is processed before submiting it to shader.

Sorted by API version

Texture Type	GL enum	OpenGL support	OpenGL ES support
2D	GL_TEXTURE_2D	1.0	1.0
1D	GL_TEXTURE_1D	1.1	
3D	GL_TEXTURE_3D	1.2	3.0
Cube Map	GL_TEXTURE_CUBE_MAP	1.3	2.0
1D Array	GL_TEXTURE_1D_ARRAY	3.0	
2D Array	GL_TEXTURE_2D_ARRAY	3.0	3.0
2D Rectangle	GL_TEXTURE_RECTANGLE	3.1	
Buffer	GL_TEXTURE_BUFFER	3.1	
2D Multisample	GL_TEXTURE_2D_MULTISAMPLE	3.2	3.1
2D Multisample Array	GL_TEXTURE_2D_MULTISAMPLE_ARRAY	3.2	
Cube Map Array	GL_TEXTURE_CUBE_MAP_ARRAY	4.0	

Textures

- Simple texture creation:

```
// Create texture and reserve memory for it's mip-maps
Glint textureId;
glGenTextures(1, &textureId);
glBindTexture(GL_TEXTURE_2D, textureId);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT); // Not for Rectangle textures!
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT); // Not for Rectangle textures!
// glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_R, GL_REPEAT); <- For 3D textures only!
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
for (uint8 i=0; i<mipmapsCount; ++i)
    glTexImage2D(GL_TEXTURE_2D,
                 mipmap,     // Current Mip-Map
                 dstFormat,  // Sized Internal Format (see next slides)
                 width,      // Current Mip-Map width in texels
                 height,     // Current Mip-Map height in texels
                 0,          // Border, deprecated, should be 0
                 srcFormat,  // Source Format
                 srcType,    // Source Data
                 nullptr);   // Pointer to client memory, or NULL if just want to allocate space
```

Textures

- Magnification and Minification filtering modes:

Mag	Min	Enum	Type
x	x	GL_NEAREST	Nearest
-	x	GL_NEAREST_MIPMAP_NEAREST	Nearest MipMaped
-	x	GL_NEAREST_MIPMAP_LINEAR	Nearest MipMaped Smooth
x	x	GL_LINEAR	Linear
-	x	GL_LINEAR_MIPMAP_NEAREST	Bilinear
-	x	GL_LINEAR_MIPMAP_LINEAR	Trilinear

- Available wraping modes:
  - GL_CLAMP_TO_EDGE,
  - GL_REPEAT
  - GL_MIRRORED_REPEAT
  - GL_MIRROR_CLAMP_TO_EDGE

Anisotropic filtering modes are accessible through extension due to IP issue.

Since OpenGL 1.4 / OpenGL ES 2.0

Since OpenGL 4.4

Textures

- Below you will find most sane texture format configurations introduced in consecutive OpenGL versions.

- Legend:
  - (nothing) – default [0..1] normalized value
  - sn – signed normalized value to [-1..1]
  - hf, f – 16 or 32 bit floating poin, not normalized
  - u, s – unsigned or signed integer, no float conversion
  - [L] – marks lower bits side

- To understand more deeply tables on next slides please refer to this wiki page.

Textures

- Below you will find most sane texture format configurations introduced in consecutive OpenGL versions.

Since OpenGL 1.0

BPT	Sized Internal Format	Format	Type	Description
24	GL_RGB8	GL_RGB	GL_UNSIGNED_BYTE	R8 G8 B8 [L]
48	GL_RGB16	GL_RGB	GL_UNSIGNED_SHORT	R16 G16 B16 [L]
32	GL_RGBA8	GL_RGBA	GL_UNSIGNED_BYTE	R8 G8 B8 A8 [L]
64	GL_RGBA16	GL_RGBA	GL_UNSIGNED_SHORT	R16 G16 B16 A16 [L]

Since OpenGL 1.2

BPT	Sized Internal Format	Format	Type	Description
8	GL_R3_G3_B2	GL_RGB	GL_UNSIGNED_BYTE_3_3_2	R3 G3 B2 [L]
8	GL_R3_G3_B2	GL_RGB	GL_UNSIGNED_BYTE_2_3_3_REV	B2 G3 R3 [L]

Textures

BPT	Sized Internal Format	Format	Type	Description
16	GL_RGB565	GL_RGB	GL_UNSIGNED_SHORT _5_6_5	R5 G6 B5 [L]
16	GL_RGB565	GL_RGB	GL_UNSIGNED_SHORT _5_6_5_REV	B5 G6 R5 [L]
24	GL_RGB8	GL_BGR	GL_UNSIGNED_BYTE	B8 G8 R8 [L]
48	GL_RGB8	GL_BGR	GL_UNSIGNED_SHORT	B16 G16 R16 [L]
16	GL_RGBA4	GL_RGBA	GL_UNSIGNED_SHORT _4_4_4_4	R4 G4 B4 A4 [L]
16	GL_RGBA4	GL_RGBA	GL_UNSIGNED_SHORT _4_4_4_4_REV	A4 B4 G4 R4 [L]
16	GL_RGBA4	GL_BGRA	GL_UNSIGNED_SHORT _4_4_4_4	B4 G4 R4 A4 [L]
16	GL_RGBA4	GL_BGRA	GL_UNSIGNED_SHORT _4_4_4_4_REV	A4 R4 G4 B4 [L]
16	GL_RGB5_A1	GL_RGBA	GL_UNSIGNED_SHORT _5_5_5_1	R5 G5 B5 A1 [L]
16	GL_RGB5_A1	GL_RGBA	GL_UNSIGNED_SHORT _5_5_5_1_REV	A1 B5 G5 R5 [L]

Textures

BPT	Sized Internal Format	Format	Type	Description
16	GL_RGB5_A1	GL_BGRA	GL_UNSIGNED_SHORT_5_5_5_1	B5 G5 R5 A1 [L]
16	GL_RGB5_A1	GL_BGRA	GL_UNSIGNED_SHORT_5_5_5_1_REV	A1 R5 G5 B5 [L]
32	GL_RGBA8	GL_RGBA	GL_UNSIGNED_INT_8_8_8_8_REV	A8 B8 G8 R8 [L]
32	GL_RGBA8	GL_BGRA	GL_UNSIGNED_BYTE	B8 G8 R8 A8 [L]
32	GL_RGBA8	GL_BGRA	GL_UNSIGNED_INT_8_8_8_8_REV	A8 R8 G8 B8 [L]
32	GL_RGB10_A2	GL_RGBA	GL_UNSIGNED_SHORT_10_10_10_2	R10 G10 B10 A2 [L]
32	GL_RGB10_A2	GL_RGBA	GL_UNSIGNED_SHORT_10_10_10_2_REV	A2 B10 G10 R10 [L]
32	GL_RGB10_A2	GL_BGRA	GL_UNSIGNED_SHORT_10_10_10_2	B10 G10 R10 A2 [L]
32	GL_RGB10_A2	GL_BGRA	GL_UNSIGNED_SHORT_10_10_10_2_REV	A2 R10 G10 B10 [L]
64	GL_RGBA16	GL_BGRA	GL_UNSIGNED_SHORT	B16 G16 R16 A16 [L]

Textures

Since OpenGL 1.3

BPT	Sized Internal Format	Format	Type	Description
12	GL_RGB4	GL_RGB	GL_UNSIGNED_BYTE	R8G8B8 -> R4 G4 B4 [L]
15	GL_RGB5	GL_RGB	GL_UNSIGNED_BYTE	R8G8B8 -> R5 G5 B5 [L]
30	GL_RGB10	GL_RGB	GL_UNSIGNED_SHORT	RGB16->R10 G10 B10[L]
36	GL_RGB12	GL_RGB	GL_UNSIGNED_SHORT	RGB16->R12 G12 B12[L]
8	GL_RGBA2	GL_RGBA	GL_UNSIGNED_BYTE	RGBA8->R2 G2 B2 A2[L]
48	GL_RGBA12	GL_RGBA	GL_UNSIGNED_SHORT	RGBA16->RGBA12[L]

Since OpenGL 1.4

BPT	Sized Internal Format	Format	Type	Description
16	GL_DEPTH_COMPONENT16	GL_DEPTH_COMPONENT	GL_UNSIGNED_SHORT	D16 [L]
24	GL_DEPTH_COMPONENT24	GL_DEPTH_COMPONENT	GL_UNSIGNED_INT	UI32 low 24bit ->D24 [L]
32	GL_DEPTH_COMPONENT32	GL_DEPTH_COMPONENT	GL_UNSIGNED_INT	D32 [L]

Textures

Since OpenGL 3.0

BPT	Sized Internal Format	Format	Type	Description
8	GL_R8	GL_RED	GL_UNSIGNED_BYTE	R8 [L]
8	GL_R8UI	GL_RED_INTEGER	GL_UNSIGNED_BYTE	R8u [L]
8	GL_R8I	GL_RED_INTEGER	GL_BYTE	R8s [L]
16	GL_R16	GL_RED	GL_UNSIGNED_SHORT	R16 [L]
16	GL_R16UI	GL_RED_INTEGER	GL_UNSIGNED_SHORT	R16u [L]
16	GL_R16I	GL_RED_INTEGER	GL_SHORT	R16s [L]
16	GL_R16F	GL_RED	GL_HALF_FLOAT	R16hf [L]
16	GL_R16F	GL_RED	GL_FLOAT	R32f -> R16hf [L]
32	GL_R32UI	GL_RED_INTEGER	GL_UNSIGNED_INT	R32u [L]
32	GL_R32I	GL_RED_INTEGER	GL_INT	R32s [L]
32	GL_R32F	GL_RED	GL_FLOAT	R32f [L]
16	GL_RG8	GL_RG	GL_UNSIGNED_BYTE	R8 G8 [L]
16	GL_RG8UI	GL_RG_INTEGER	GL_UNSIGNED_BYTE	R8u G8u [L]
16	GL_RG8I	GL_RG_INTEGER	GL_BYTE	R8s G8s [L]

Textures

BPT	Sized Internal Format	Format	Type	Description
32	GL_RG16	GL_RG	GL_UNSIGNED_SHORT	R16 G16 [L]
32	GL_RG16UI	GL_RG_INTEGER	GL_UNSIGNED_SHORT	R16u G16u [L]
32	GL_RG16I	GL_RG_INTEGER	GL_SHORT	R16s G16s [L]
32	GL_RG16F	GL_RG	GL_HALF_FLOAT	R16hf G16hf [L]
32	GL_RG16F	GL_RG	GL_FLOAT	RG32f -> RG16hf [L]
64	GL_RG32UI	GL_RG_INTEGER	GL_UNSIGNED_INT	R32u G32u [L]
64	GL_RG32I	GL_RG_INTEGER	GL_INT	R32s G32s [L]
64	GL_RG32F	GL_RG	GL_FLOAT	R32f G32f [L]
24	GL_RGB8UI	GL_RGB_INTEGER	GL_UNSIGNED_BYTE	R8u G8u B8u [L]
24	GL_RGB8I	GL_RGB_INTEGER	GL_BYTE	R8s G8s B8s [L]
24	GL_SRGB8	GL_RGB	GL_UNSIGNED_BYTE	(sRGB) R8 G8 B8 [L]
32	GL_R11F_G11F_B10F	GL_RGB	GL_UNSIGNED_INT_ 10F_11F_11F_REV	B10 G11 R11 [L]
32	GL_RGB9_E5	GL_RGB	GL_UNSIGNED_INT_ 5_9_9_9_REV	E5 B9 G9 R9 [L] (shared exponent)

Textures

BPT	Sized Internal Format	Format	Type	Description
48	GL_RGB16UI	GL_RGB_INTEGER	GL_UNSIGNED_SHORT	R16u G16u B16u [L]
48	GL_RGB16I	GL_RGB_INTEGER	GL_SHORT	R16s G16s B16s [L]
48	GL_RGB16F	GL_RGB	GL_HALF_FLOAT	R16hf G16hf B16hf [L]
48	GL_RGB16F	GL_RGB	GL_FLOAT	RGB32f -> RGB16hf [L]
96	GL_RGB32UI	GL_RGB_INTEGER	GL_UNSIGNED_INT	R32u G32u B32u [L]
96	GL_RGB32I	GL_RGB_INTEGER	GL_INT	R32s G32s B32s [L]
96	GL_RGB32F	GL_RGB	GL_FLOAT	R32f G32f B32f [L]
32	GL_SRGB8_ALPHA8	GL_RGBA	GL_UNSIGNED_BYTE	(sRGB) R8 G8 B8 A8 [L]
32	GL_RGBA8UI	GL_RGBA_INTEGER	GL_UNSIGNED_BYTE	R8u G8u B8u A8u [L]
32	GL_RGBA8I	GL_RGBA_INTEGER	GL_BYTE	R8s G8s B8s A8s [L]
64	GL_RGBA16UI	GL_RGBA_INTEGER	GL_UNSIGNED_SHORT	RGBA16u [L]
64	GL_RGBA16I	GL_RGBA_INTEGER	GL_SHORT	RGBA16s [L]
64	GL_RGBA16F	GL_RGBA	GL_HALF_FLOAT	RGBA16hf [L]
64	GL_RGBA16F	GL_RGBA	GL_FLOAT	RGBA32f->RGBA16hf [L]

Textures

BPT	Sized Internal Format	Format	Type	Description
128	GL_RGBA32UI	GL_RGBA_INTEGER	GL_UNSIGNED_INT	RGBA32u [L]
128	GL_RGBA32I	GL_RGBA_INTEGER	GL_INT	RGBA32s [L]
128	GL_RGBA32F	GL_RGBA	GL_FLOAT	RGBA32f [L]
32	GL_DEPTH_COMPONENT32F	GL_DEPTH_COMPONENT	GL_FLOAT	D32f [L]
32	GL_DEPTH24_STENCIL8	GL_DEPTH_STENCIL	GL_UNSIGNED_INT_24_8	D24 S8 [L]
64	GL_DEPTH32F_STENCIL8	GL_DEPTH_STENCIL	GL_FLOAT_32_UNSIGNED_INT_24_8_REV	(unused24) S8 D32f [L]

Textures

Since OpenGL 3.1

BPT	Sized Internal Format	Format	Type	Description
8	GL_R8_SNORM	GL_RED	GL_BYTE	R8sn [L]
16	GL_R16_SNORM	GL_RED	GL_SHORT	R16sn [L]
16	GL_RG8_SNORM	GL_RG	GL_BYTE	R8sn G8sn [L]
32	GL_RG16_SNORM	GL_RG	GL_SHORT	R16sn G16sn [L]
24	GL_RGB8_SNORM	GL_RGB	GL_BYTE	R8sn G8sn B8sn [L]
48	GL_RGB16_SNORM	GL_RGB	GL_SHORT	R16sn G16sn B16sn [L]
32	GL_RGBA8_SNORM	GL_RGBA	GL_BYTE	RGBA8sn[L]
64	GL_RGBA16_SNORM	GL_RGBA	GL_SHORT	RGBA16sn[L]

Textures

Since OpenGL 3.3

BPT	Sized Internal Format	Format	Type	Description
32	GL_RGB10_A2UI	GL_RGBA_INTEGER	GL_UNSIGNED_INT_10_10_10_2	R10u G10u B10u A2u [L]
32	GL_RGB10_A2UI	GL_RGBA_INTEGER	GL_UNSIGNED_INT_10_10_10_2_REV	A2u B10u G10u R10u [L]
32	GL_RGB10_A2UI	GL_BGRA_INTEGER	GL_UNSIGNED_INT_10_10_10_2	B10u G10u R10u A2u [L]
32	GL_RGB10_A2UI	GL_BGRA_INTEGER	GL_UNSIGNED_INT_10_10_10_2_REV	A2u R10u G10u B10u [L]

Since OpenGL 4.3

- It is possible to specify if we want to read Depth or Stencil value of DepthStencil texture. When sampling stencil value **usampler2D** shouble be used (more info here).

Textures

- Attaching texture and uniform sampler to texture unit:

```
glUseProgram(programId);

for (uint8 i=0; i<samplersToMatch; ++i)
    {
    glActiveTexture(GL_TEXTURE0 + i);    // Activating texture unit from range:
                                         // [0..GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS] (min. 80)
    glBindTexture(type, texture[i].id);  // Bind texture of given TYPE to this texture unit
    glUniform1i(sampler[i].location, i); // Bind sampler from LOCATION to I'TH texture unit
    }
```

- Uniform Samplers are treated the same way as regular Uniform Parameters. This means they are listed together as active uniforms, and need to be distinguished by type.
- glUniform1i and glUniform1v are used to bind samplers.

Uniform Samplers

- Samplers and UV coordinates are used by built-in texture lookup functions that return acquired color or other data stored in textures.

GLSL 1.10 Fragment Shader

```
#version 110

uniform sampler2D colorSampler;
...
vec2 uv = vec2(0.5, 0.5);
vec4 color = texture2D(colorSampler, uv);
```

GLSL 1.10 – 1.20

GLSL 1.30 Fragment Shader

```
#version 130

uniform sampler2D colorSampler;
...
vec2 uv = vec2(0.5, 0.5);
vec4 color = texture(colorSampler, uv);
```

Since GLSL 1.30

Uniform Samplers

- Types of samplers:

Since OpenGL 2.0 / GLSL 1.10

Enum	GLSL variable
GL_SAMPLER_1D	sampler1D
GL_SAMPLER_2D	sampler2D
GL_SAMPLER_3D	sampler3D
GL_SAMPLER_CUBE	samplerCube
GL_SAMPLER_1D_SHADOW	sampler1DShadow
GL_SAMPLER_2D_SHADOW	sampler2DShadow

Uniform Samplers

- Types of samplers:

Since OpenGL 3.0 / GLSL 1.30

Enum	GLSL variable
GL_SAMPLER_1D_ARRAY	sampler1DArray
GL_SAMPLER_2D_ARRAY	sampler2DArray
GL_SAMPLER_CUBE_SHADOW	samplerCubeShadow
GL_SAMPLER_1D_ARRAY_SHADOW	sampler1DArrayShadow
GL_SAMPLER_2D_ARRAY_SHADOW	sampler2DArrayShadow
GL_INT_SAMPLER_1D	isampler1D
GL_INT_SAMPLER_2D	isampler2D
GL_INT_SAMPLER_3D	isampler3D
GL_INT_SAMPLER_CUBE	isamplerCube
GL_INT_SAMPLER_1D_ARRAY	isampler1DArray
GL_INT_SAMPLER_2D_ARRAY	isampler2DArray
GL_UNSIGNED_INT_SAMPLER_1D	usampler1D
GL_UNSIGNED_INT_SAMPLER_2D	usampler2D
GL_UNSIGNED_INT_SAMPLER_3D	usampler3D
GL_UNSIGNED_INT_SAMPLER_CUBE	usamplerCube
GL_UNSIGNED_INT_SAMPLER_1D_ARRAY	usampler1DArray
GL_UNSIGNED_INT_SAMPLER_2D_ARRAY	usampler2DArray

Uniform Samplers

- Types of samplers:

Since OpenGL 3.1 / GLSL 1.40

Enum	GLSL variable
GL_SAMPLER_BUFFER	samplerBuffer
GL_SAMPLER_2D_RECT	sampler2DRect
GL_SAMPLER_2D_RECT_SHADOW	sampler2DRectShadow
GL_INT_SAMPLER_BUFFER	isamplerBuffer
GL_INT_SAMPLER_2D_RECT	isampler2DRect
GL_UNSIGNED_INT_SAMPLER_BUFFER	usamplerBuffer
GL_UNSIGNED_INT_SAMPLER_2D_RECT	usampler2DRect

Since OpenGL 3.2 / GLSL 1.50

Enum	GLSL variable
GL_SAMPLER_2D_MULTISAMPLE	sampler2DMS
GL_SAMPLER_2D_MULTISAMPLE_ARRAY	sampler2DMSArray
GL_INT_SAMPLER_2D_MULTISAMPLE	isampler2DMS
GL_INT_SAMPLER_2D_MULTISAMPLE_ARRAY	isampler2DMSArray
GL_UNSIGNED_INT_SAMPLER_2D_MULTISAMPLE	usampler2DMS
GL_UNSIGNED_INT_SAMPLER_2D_MULTISAMPLE_ARRAY	usampler2DMSArray

Uniform Samplers

- Types of samplers:

Since OpenGL 4.0 / GLSL 4.00

Enum	GLSL variable
GL_SAMPLER_CUBE_MAP_ARRAY GL_SAMPLER_CUBE_MAP_ARRAY_SHADOW GL_INT_SAMPLER_CUBE_MAP_ARRAY GL_UNSIGNED_INT_SAMPLER_CUBE_MAP_ARRAY	samplerCubeArray samplerCubeArrayShadow isamplerCubeArray usamplerCubeArray

Sampler Objects

Since OpenGL 3.3

- Texture state, and Sampler state were separated to allow different access types to the same texture at the same time.

```
// Create sampler and specify sampling parameters
Glint samplerId;
glGenSampler(1, &samplerId);
glSamplerParameteri(samplerId, GL_TEXTURE_WRAP_S, GL_REPEAT);
glSamplerParameteri(samplerId, GL_TEXTURE_WRAP_T, GL_REPEAT);
glSamplerParameteri(samplerId, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glSamplerParameteri(samplerId, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

glUseProgram(programId);
for (uint8 i=0; i<samplersToMatch; ++i)
    {
    glActiveTexture(GL_TEXTURE0 + i);   // Activating texture unit from range:
                                        // [0..GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS] (min. 80)
    glBindTexture(type, texture[i].id);     // Bind texture of given TYPE to this texture unit
    glUniform1i(sampler[i].location, i);    // Bind sampler from LOCATION to I'TH texture unit
    glBindSampler(i, samplerId);            // Bind Sampler Object to I'TH texture unit
    }
```

If sampler object is bound together with texture object to thesame texture unit, sampler state superseeds texture state.

Images

Since OpenGL 4.2 / GLSL 4.20

- Images are very similar to textures but they allow to:
  - They allow Read/Write access instead of Textures Read-Only mode.
  - They allow atomic operations on their content.
  - They are bound to „Image Units” instead od „Texture Units”.
- Images are treated by compiler in the same way as uniform parameters and uniform samplers. Therefore they also need to be distinguished by returned type.
- Given texture can be bind to one of „Image Units” to be used as a Image by the shader:

```
// Create Image by binding Texture to Image Unit
glBindImageTexture(imageUnit, textureId, mipmap, shouldImageBeLayered, textureLayer,
                   accessType, formatWhenWriting);
```

- More can be found here.

Images

- Types of images:

Since OpenGL 4.2 / GLSL 4.20

Enum	GLSL variable
GL_IMAGE_1D	image1D
GL_IMAGE_2D	image2D
GL_IMAGE_3D	image3D
GL_IMAGE_2D_RECT	image2DRect
GL_IMAGE_CUBE	imageCube
GL_IMAGE_BUFFER	imageBuffer
GL_IMAGE_1D_ARRAY	image1DArray
GL_IMAGE_2D_ARRAY	image2DArray
GL_IMAGE_2D_MULTISAMPLE	image2DMS
GL_IMAGE_2D_MULTISAMPLE_ARRAY	image2DMSArray
GL_INT_IMAGE_1D	iimage1D
GL_INT_IMAGE_2D	iimage2D
GL_INT_IMAGE_3D	iimage3D
GL_INT_IMAGE_2D_RECT	iimage2DRect
GL_INT_IMAGE_CUBE	iimageCube

Images

- Types of images:

Since OpenGL 4.2 / GLSL 4.20

Enum	GLSL variable
GL_INT_IMAGE_BUFFER	iimageBuffer
GL_INT_IMAGE_1D_ARRAY	iimage1DArray
GL_INT_IMAGE_2D_ARRAY	iimage2DArray
GL_INT_IMAGE_2D_MULTISAMPLE	iimage2DMS
GL_INT_IMAGE_2D_MULTISAMPLE_ARRAY	iimage2DMSArray
GL_UNSIGNED_INT_IMAGE_1D	uimage1D
GL_UNSIGNED_INT_IMAGE_2D	uimage2D
GL_UNSIGNED_INT_IMAGE_3D	uimage3D
GL_UNSIGNED_INT_IMAGE_2D_RECT	uimage2DRect
GL_UNSIGNED_INT_IMAGE_CUBE	uimageCube
GL_UNSIGNED_INT_IMAGE_BUFFER	uimageBuffer
GL_UNSIGNED_INT_IMAGE_1D_ARRAY	uimage1DArray
GL_UNSIGNED_INT_IMAGE_2D_ARRAY	uimage2DArray
GL_UNSIGNED_INT_IMAGE_2D_MULTISAMPLE	uimage2DMS
GL_UNSIGNED_INT_IMAGE_2D_MULTISAMPLE_ARRAY	uimage2DMSArray

Layouts

- Up until this moment, we've always queried locations of attributes, uniforms, buffers and other data containers from linked program.
- Binding attribs location from application (before linking):

  glBindAttribLocation(programId, location, „name");

- It is also possible to directly specify locations from within the shader:

  GLSL 4.50 Vertex Shader

```
#version 450

layout(location = 0) in vec3 position;

Layout(location = 0) uniform vec3 color;
layout(location = 1) uniform vec2 uv0;
Layout(location = 2) uniform vec3 mvp;
...
```

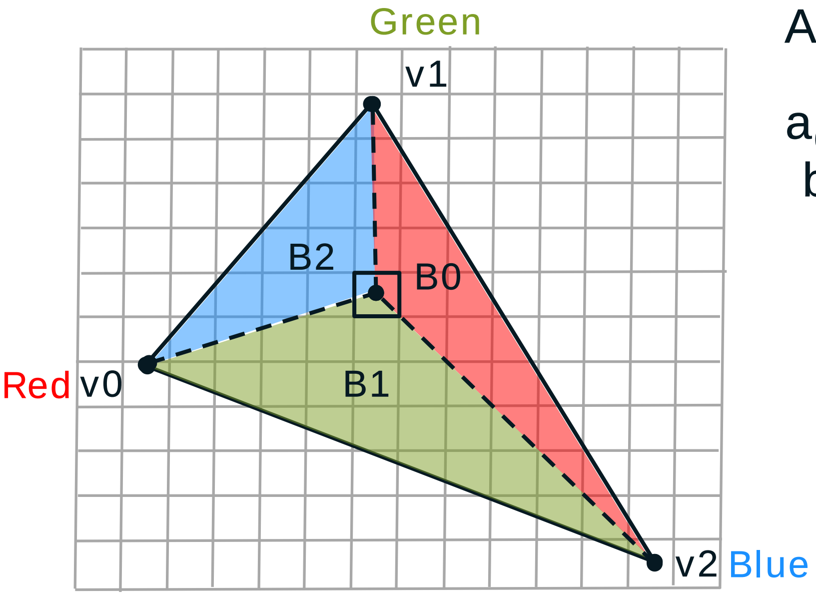
Since GLSL 4.30

More about layout qualifiers here.

Interpolation

- Interpolation qualifiers:
  - *smooth* – it is a default interpolation mode. Specifies that final value for given fragment will be perspective-correct and computed using *barycentric interpolation*.

Explicit since GLSL 1.30.4



$$A = (a_0 \cdot b_0) + (a_1 \cdot b_1) + (a_2 \cdot b_2)$$

a_0, a_1, a_2 – vertex values per-triangle

b_1, b_2 – barycentric parameters per-pixel

$$a_0 = \frac{v_0.xyz}{v_0.w} \qquad b_0 = 1 - b_1 - b_2$$

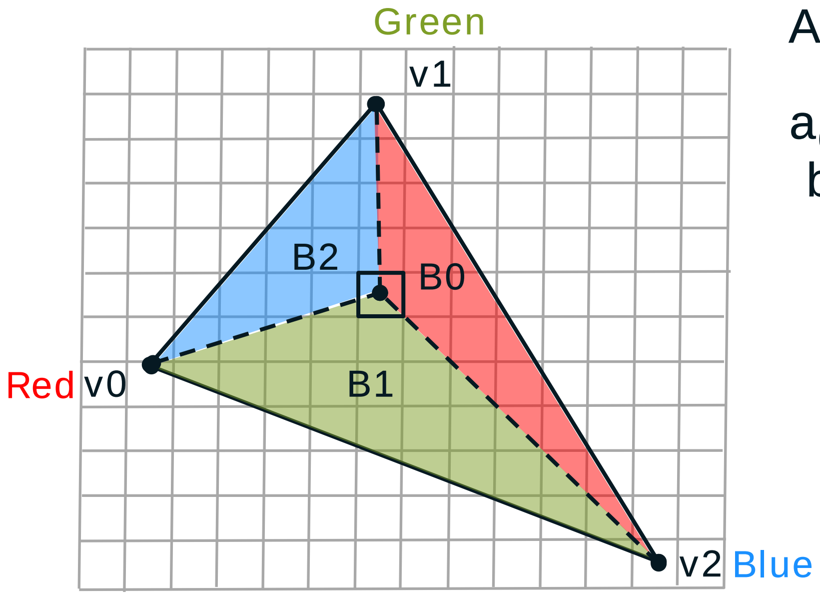
$$a_1 = \frac{v_1.xyz}{v_1.w} \qquad b_1 = \frac{B1}{(B0+B1+B2)}$$

$$a_2 = \frac{v_2.xyz}{v_2.w} \qquad b_2 = \frac{B2}{(B0+B1+B2)}$$

Interpolation

Since GLSL 1.30.4

- Interpolation qualifiers:
  - *nonperspective* – forces value to be interpolated linearly without perspective correction (no division by w component).



$$A = (a_0 \cdot b_0) + (a_1 \cdot b_1) + (a_2 \cdot b_2)$$

a_0, a_1, a_2 – vertex values per-triangle

b_1, b_2 – barycentric parameters per-pixel

$$a_0 = v_0 \qquad b_0 = 1 - b_1 - b_2$$

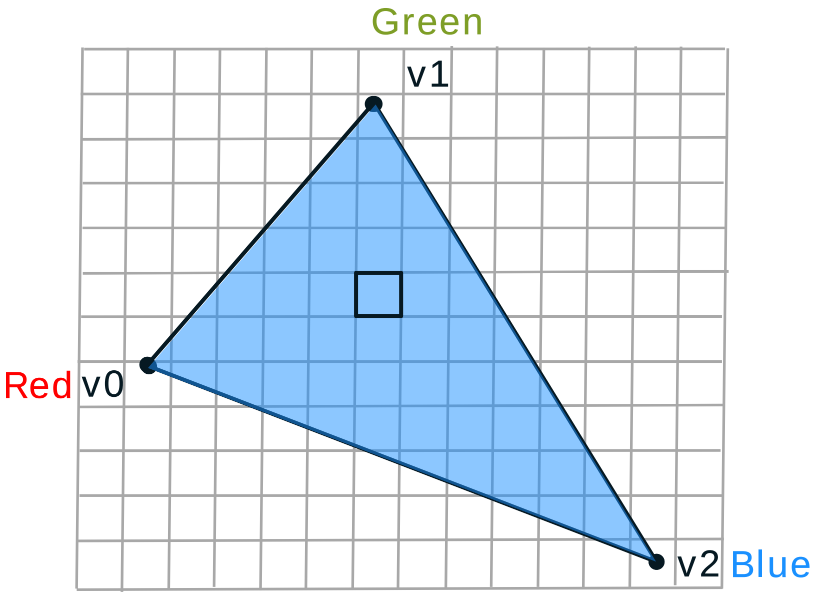
$$a_1 = v_1 \qquad b_1 = \frac{B1}{(B0+B1+B2)}$$

$$a_2 = v_2 \qquad b_2 = \frac{B2}{(B0+B1+B2)}$$

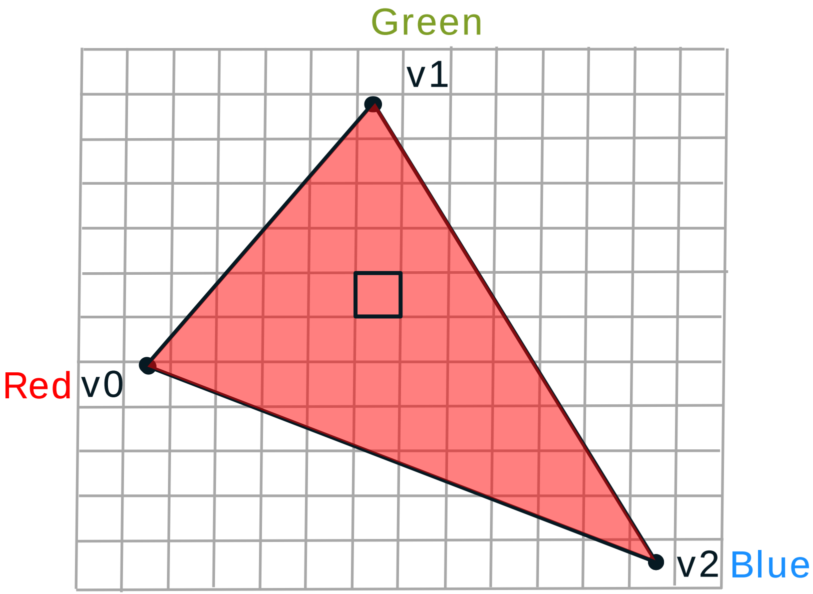
Interpolation

Since GLSL 1.30.4

- Interpolation qualifiers:
  - *flat* – means no interpolation. All fragments will use the same value from vertex specified as *provoking vertex.* By default last vertex in primitive is set as *provoking vertex*.



glProvokingVertex(GL_LAST_VERTEX_CONVENTION);



glProvokingVertex(GL_FIRST_VERTEX_CONVENTION);

Interpolation

Qualifier (optional)	GLSL	Description
flat	1.30.4+	No interpolation, constant value
smooth	1.30.4+	Barycentric interpolation (default)
nonperspective	1.30.4+	Linear interpolation (no division by w)

- When last geometry stage output qualifiers don’t match Fragment Shader input qualifiers, second one’s are used.

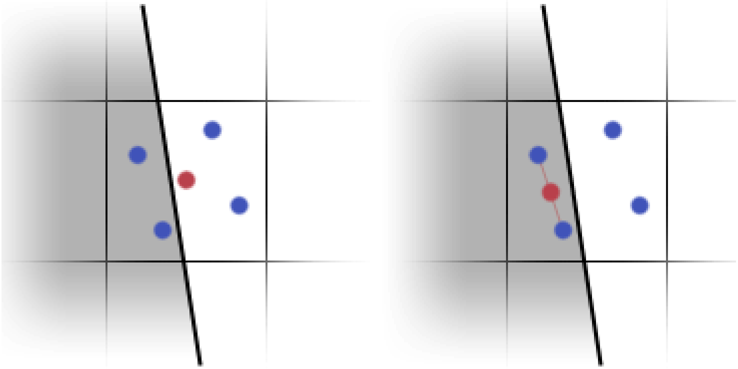
Normal vectors can change length during interpolation. Normalize them again.

Integer types and double values are not interpolated! Should always be specified as flat.

Interpolation

Since GLSL 1.10

- Additional multisample qualifiers:
  - *centroid* – instead of interpolating for pixel center, forces HW to interpolate for location within pixel area covered by primitive (in theory centroid of covered samples)



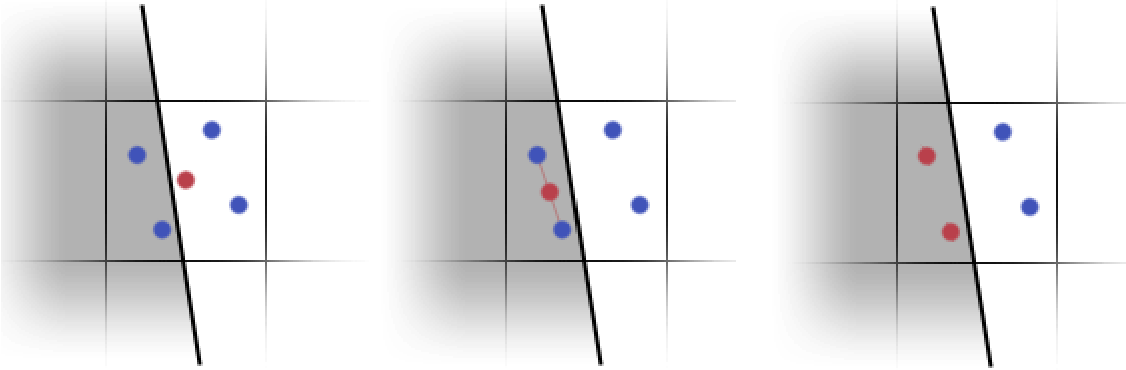
no-centroid centroid

Location of centroid sampling is not strictly specified. In most IHV's it is location of one of the covered samples.

Interpolation

Since GLSL 4.00

- Additional multisample qualifiers:
  - *sample* – interpolation of input variables from vertices is performed per-sample instead of per-pixel. It can be combined with *flat*, *smooth* and *nonperspective* qualifiers.



no-centroid centroid sample

Interpolation

Multisampling Qualifiers (optional)	GLSL	Description
Centroid – samples center interpolation, instead of pixel center interpolation		
centroid varying	1.10 - 1.20	Equivalent of „centroid smooth”
centroid smooth	1.30.4+	Multisampling correct „smooth”
centroid nonperspective	1.30.4+	Multisampling correct „nonperspective”
Sample – per-sample interpolation, instead of per-pixel interpolation		
sample flat	4.00+	Per-sample constant value
sample smooth	4.00+	Per-sample barycentric interpolation
sample nonperspective	4.00+	Per-sample linear interpolation

Interpolation

Since GLSL 4.00

- Explicit interpolation:
  - It is possible to explicitly interpolate given inputs using built-in functions:

```
<t> interpolateAtCentroid(<t> interpolant) ;
<t> interpolateAtSample(<t> interpolant, int sample);
<t> interpolateAtOffset(<t> interpolant, vec2 offset);
```

  - There are also new built-in variables:

```
in   int   gl_SampleID;          - id of currently processed sample
in   vec2  gl_SamplePosition;    - in-pixel position of current sample
in   int   gl_SampleMaskIn[];    - bitmask of samples covered by primitive
out  int   gl_SampleMask[];      - bitmask of samples that should be used
                                   to resolve multisampling in rendertarget
```

Using *gl_SampleID* in the Fragment Shader changes it to „Sample Shader” and means that whole code will be executed per-sample!

Graphic Primitives

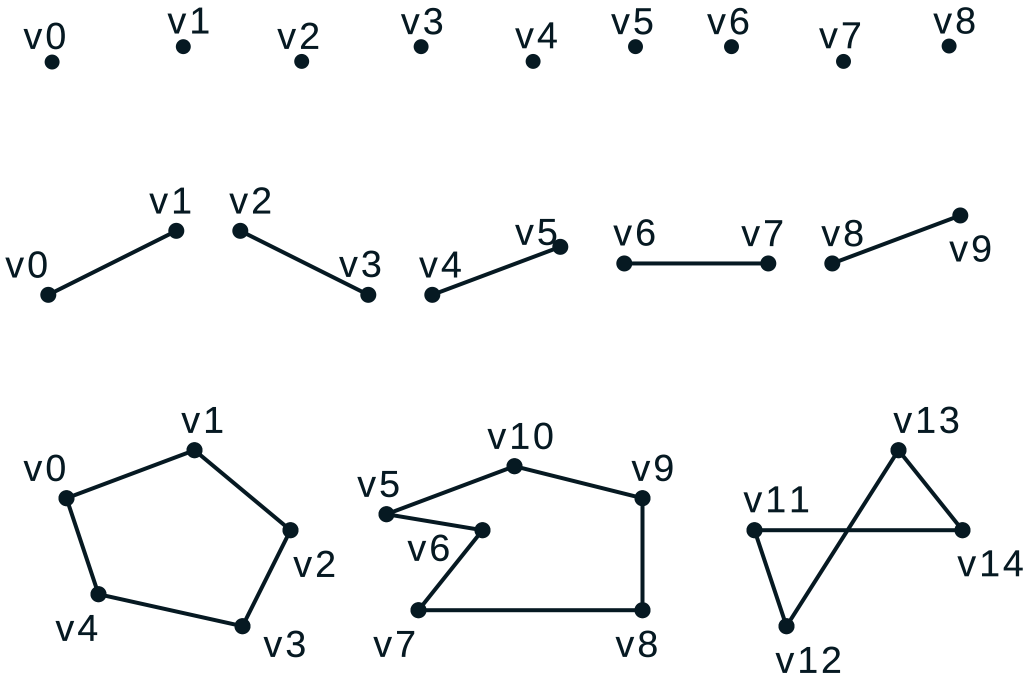
Bonus Content

Vertex and Line types accepted by OpenGL:

Points
(GL_POINTS)

Separate Lines
(GL_LINES)

Line Loops
(GL_LINE_LOOP)



Graphic Primitives

Bonus Content

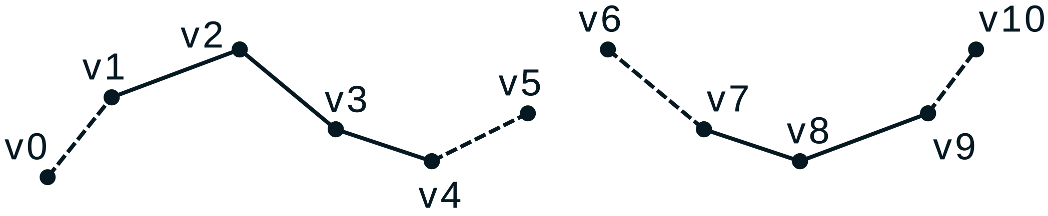
Vertex and Line types accepted by OpenGL:

Line Stripes
(GL_LINE_STRIP)

Special Types*:

Separate Lines
with Adjacency
(GL_LINES_ADJACENCY)

Line Stripes
with Adjacency
(GL_LINE_STRIP_ADJACENCY)



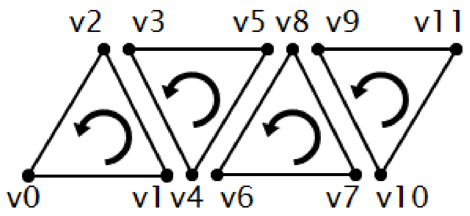
*Will be explained with Geometry Shaders.

Graphic Primitives

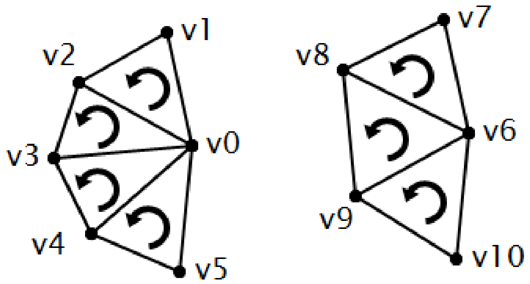
Bonus Content

Triangle types accepted by OpenGL:

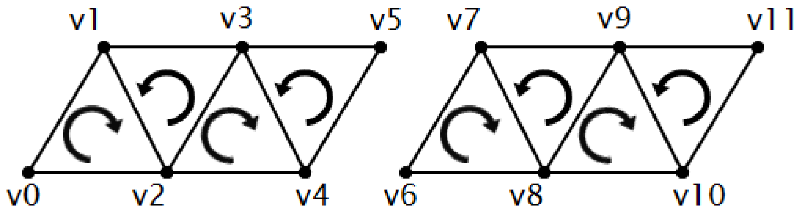
Separate Triangles
(GL_TRIANGLES)



Triangle Fans
(GL_TRIANGLE_FAN)



Triangle Stripes
(GL_TRIANGLE_STRIP)

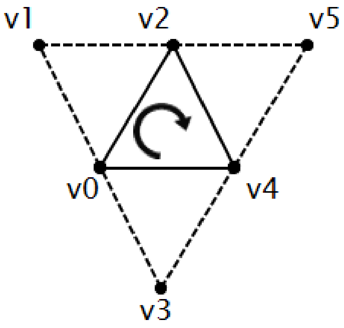


Graphic Primitives

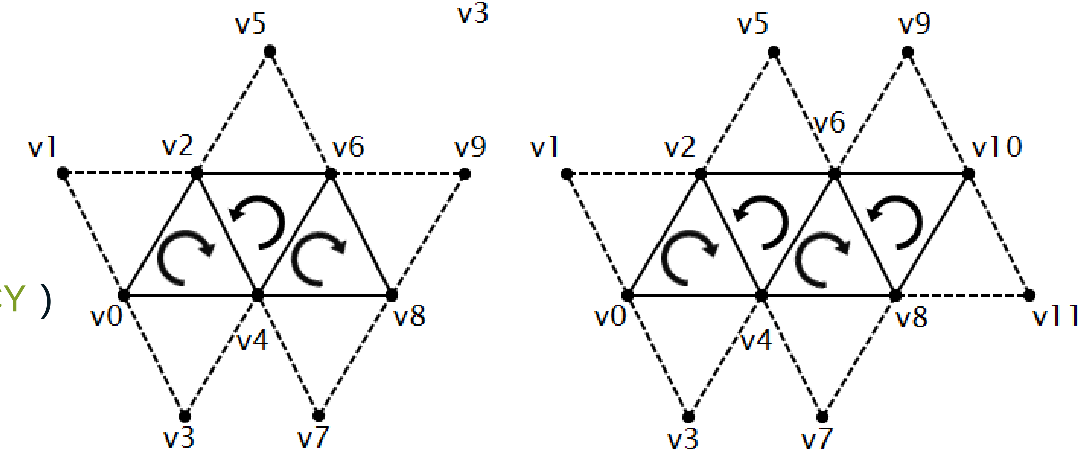
Bonus Content

*Special Triangle types accepted by OpenGL:

Separate Triangles
with Adjacency
(GL_TRIANGLES_ADJACENCY)



Triangle Stripes
with Adjacency
(GL_TRIANGLE_STRIP_ADJACENCY)



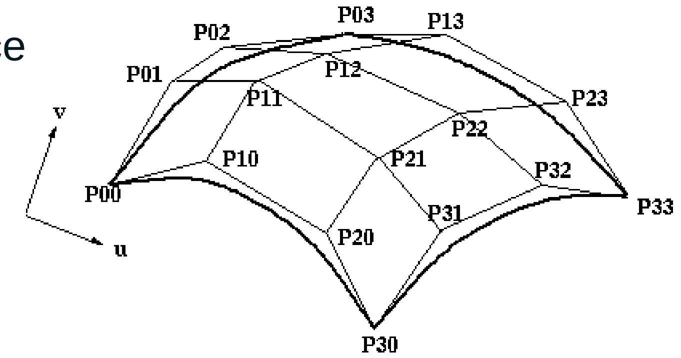
*Will be explained with Geometry Shaders.

Graphic Primitives

Bonus Content

Patch

- PATCH is a generalized abstract description of surface or line
- PATCH allows to describe any simple primitive like point, line, triangle or quad
- PATCH can also describe complex surfaces like Bezier bi-cubic surface
- PATCH is constructed from Control Points
- Control Points helps describe shape of PATCH surface



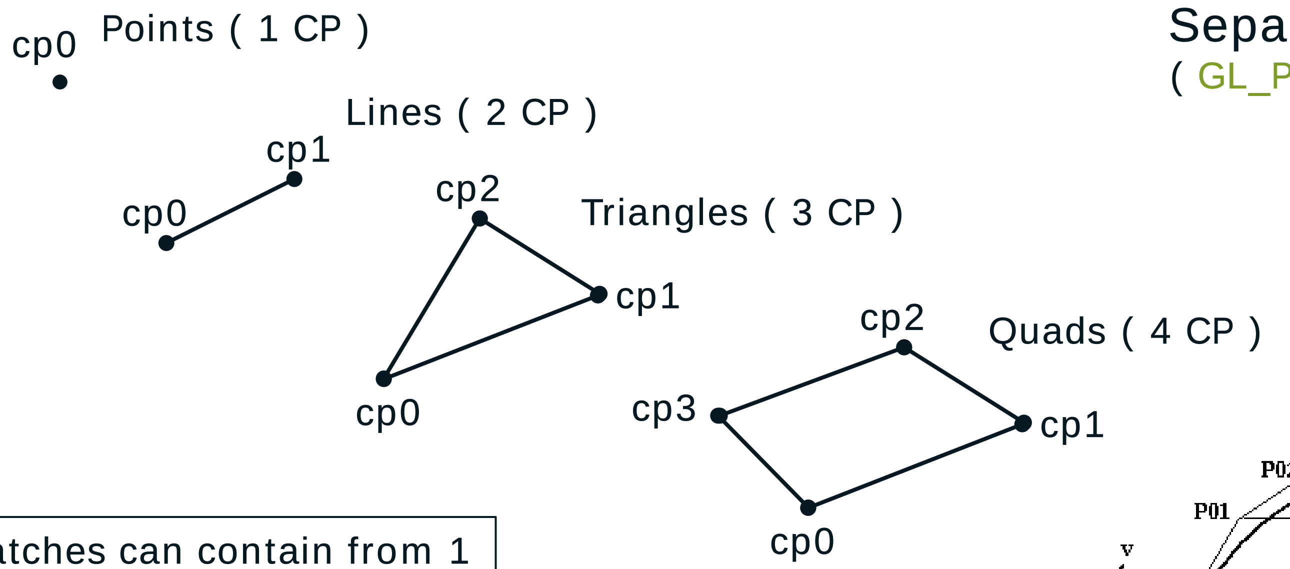
- PATCHes are transparent and backward compatible
- We can treat basic input primitive as PATCH
- Then it's Vertices are treated like Control Points
- This allows to use tessellation without changes in mesh description
- In fact there is no difference between them, important is how you use them!

Graphic Primitives

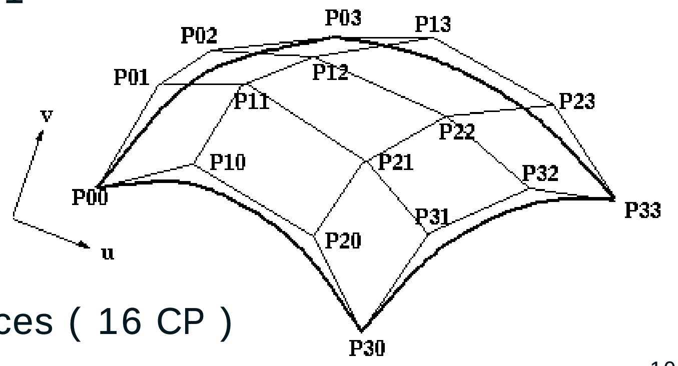
Bonus Content

Patch types accepted by OpenGL:

Separate Patches
(GL_PATCHES)

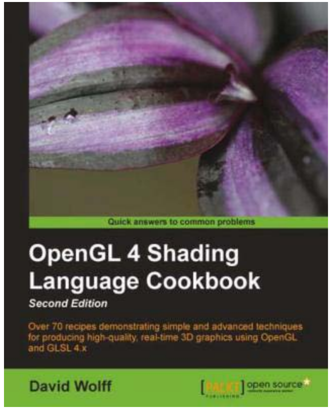


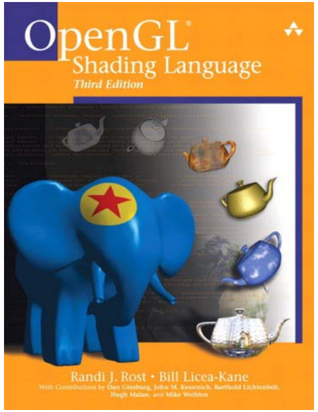
Patches can contain from 1 to up to 32 Control Points.

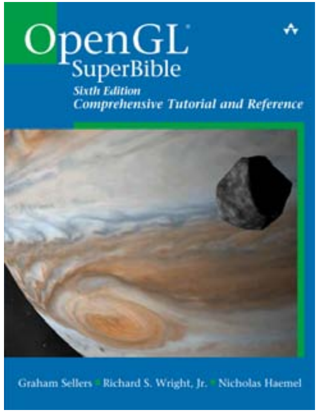


Bezier bi-cubic surfaces (16 CP)

Where to start ?







- OpenGL Shading Language CookBook

  (Source code examples can be found here.)
- http://open.gl/ , http://www.g-truc.net/
- GLEW, GLUT will make your work trivial, simple app can be done in 1h!

Conclusion

Vertex Shader

Explosions, gas, smoke

- Animation and movement of big amount of simple particles with given lifecycle and emitters

Foliage and plants

- Animation of grass, leaves and small plants based on force of wind and it's direction
- Can also take into notice for eg. Surrounding explosions

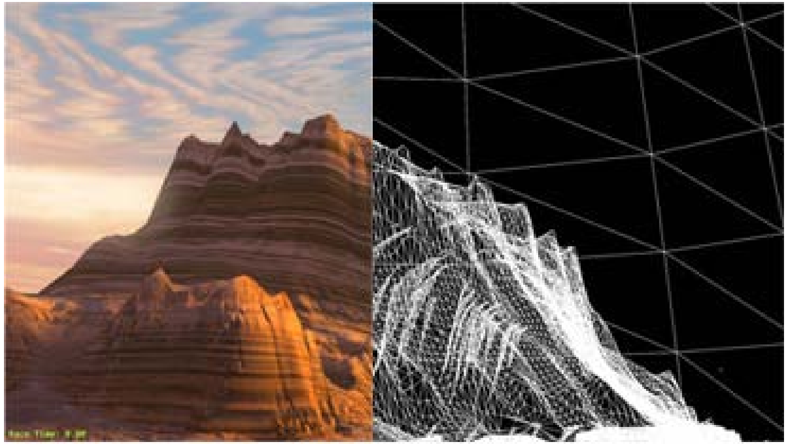
Tessellation Shaders

Deformation, transformation

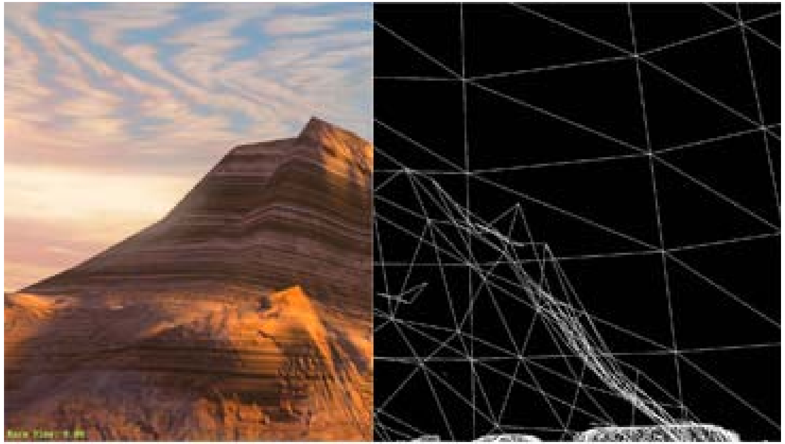
- Skin disease deformation
- Terrain modification and transformation
- Vehicles more realistic destruction

Adaptive geometry

- Amount of geometry details dynamically matched to their visibility, distance from observer
- No need for few mesh versions



Tesselation On



Tesselation Off

Geometry Shader

Particle systems:

- Hair rendering
- Water, fire, smoke, dust, foam, clouds, explosion debris etc.

Geometry output:

- Allows procedural geometry by it's gradual update in time
- General purpose algorithms on Hardware without Compute Shaders support

Fragment Shader

Colors Calculation

- Shading surfaces of geometry visible on screen, to enhance visual quality of rendered image.

- Simple texture mapping
- Increasing surface details
- Illumination models
- Procedural algorithms

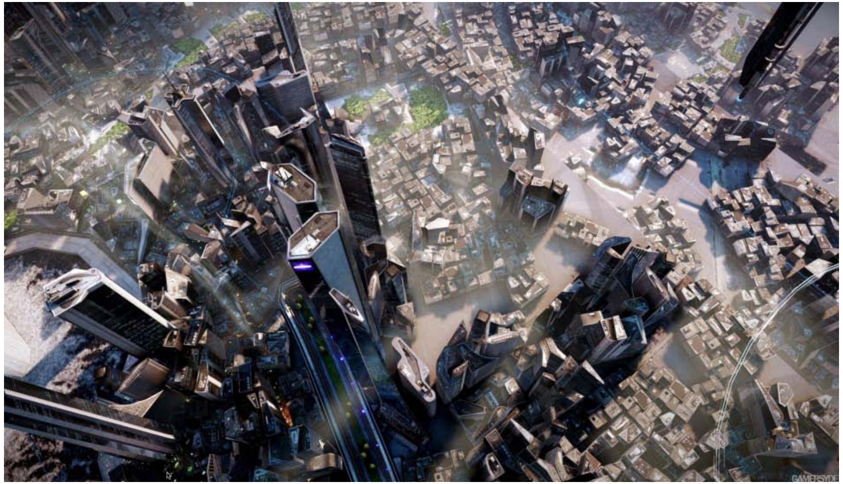
Compute Shader

Post-processing effects

- Apply lighting techniques to enhance the mood in a scene. Secondary light bounces and Global Illumination.

Depth of field and defocus

- More realistic transitions of focal points – imagine looking through a gun sight or a camera lens. Bokeh DOF algorithms.



Thank You