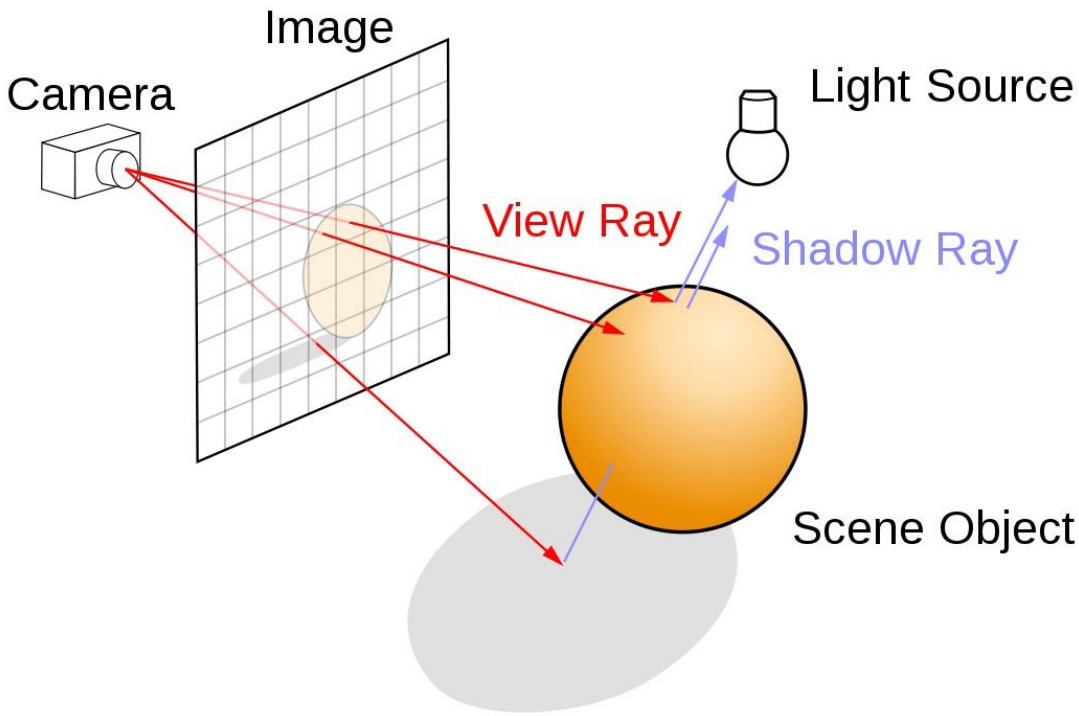
Advanced Pipelines and Introduction to GLSL

The Ray-tracing pipeline

The ray-tracing pipeline creates images from pixels on screen, and not driven by meshes triangles and their corresponding vertices.

For each fragment on screen, a ray is cast into the scene, and it is intersected with all the triangles of all the meshes in the 3D environment.

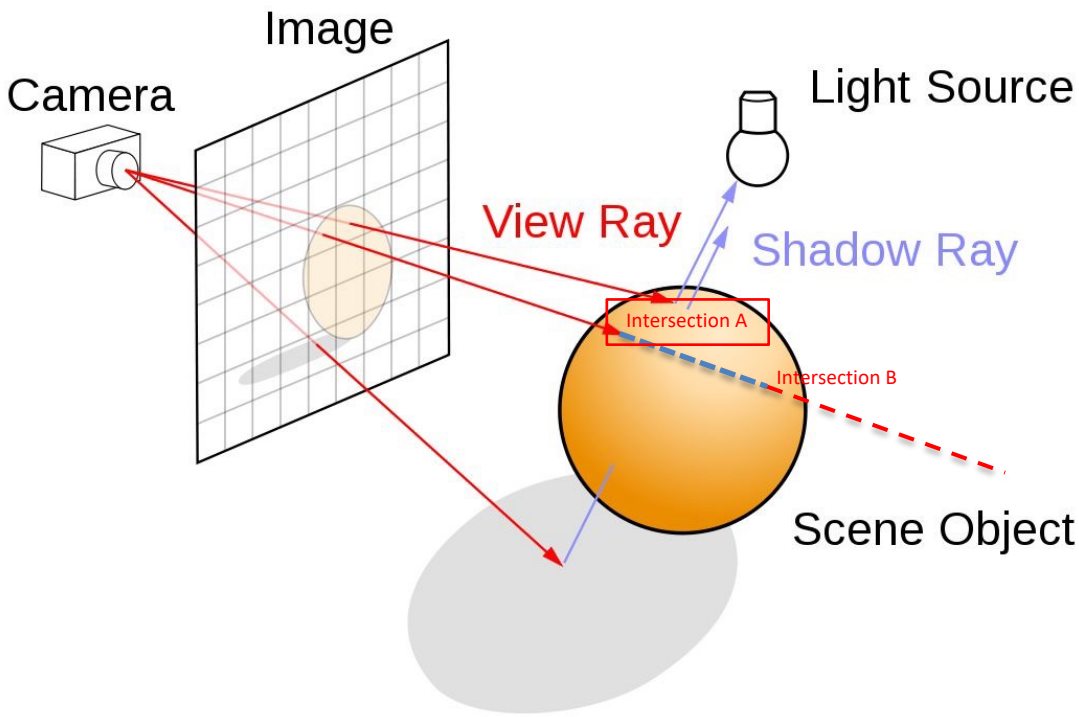


The Ray-tracing pipeline

Only the intersection closer to the viewer is considered.

In order to compute its color, extra rays can be traced to accurately reproduce reflections and transparencies.

We will return on the ray-tracing rendering technique in the following.

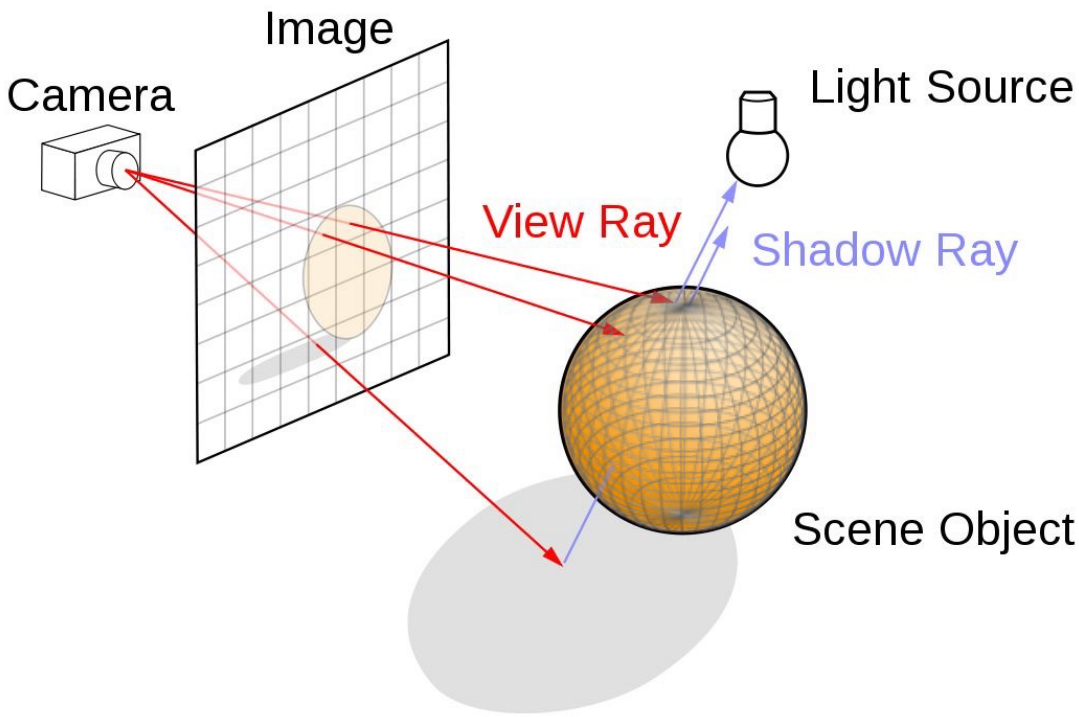


The Ray-tracing pipeline

Determining the intersection with all the triangles in the scene is not a simple task: without special care, it can have complexity $o(n^2)$ in the number of triangles.

In order to cope with this complexity, special acceleration structures must be provided by the user.

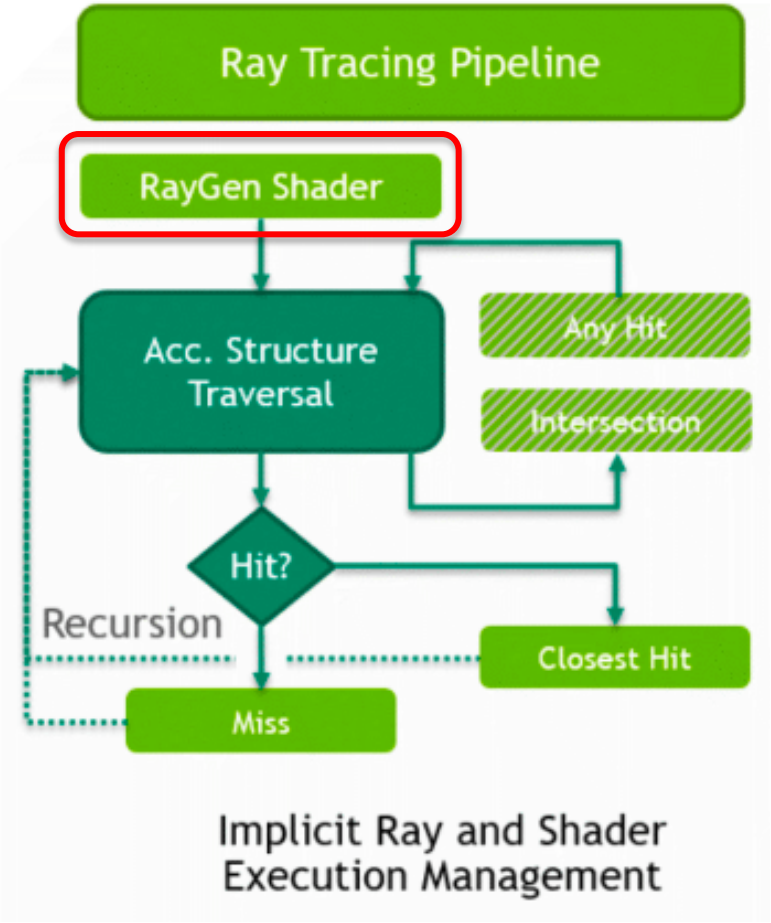
For time constraints we will not be able to present them in depth.



The Ray-tracing pipeline

The ray-tracing pipeline requires five shaders to compute and handle all the triangle-ray intersections required to render an image.

The *RayGen* shader is executed for each output fragment of the images, and it must determine the starting point and the direction of the corresponding ray in the scene.

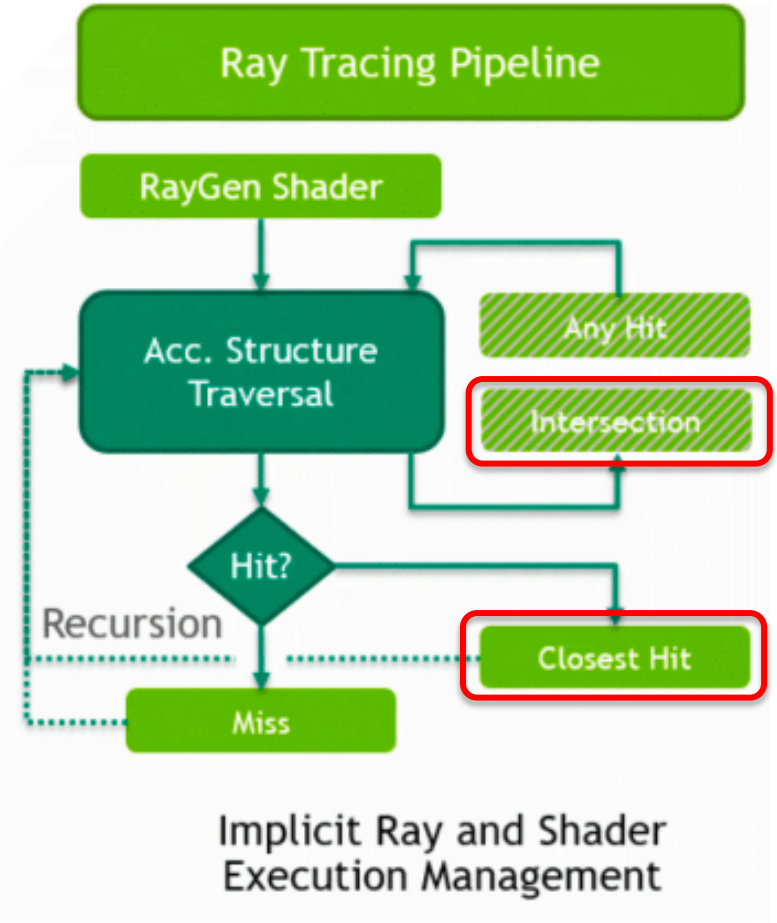


The Ray-tracing pipeline

The *Intersection* shader allows to compute custom ray triangle intersection procedure.

The *Closest Hit* shader is instead called on the point which is closer to the viewer.

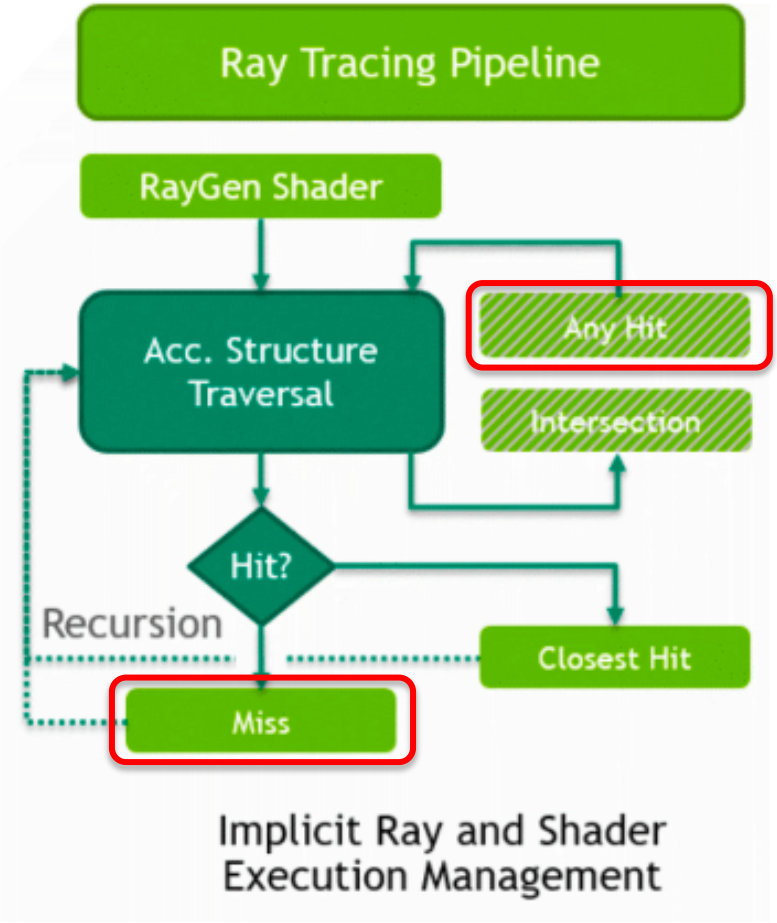
Its purpose is computing its color, and for doing this, it can recursively cast other rays.



The Ray-tracing pipeline

The *Any Hit* shader is used to filter out intersections that should not be considered, for example to handle partially transparent objects.

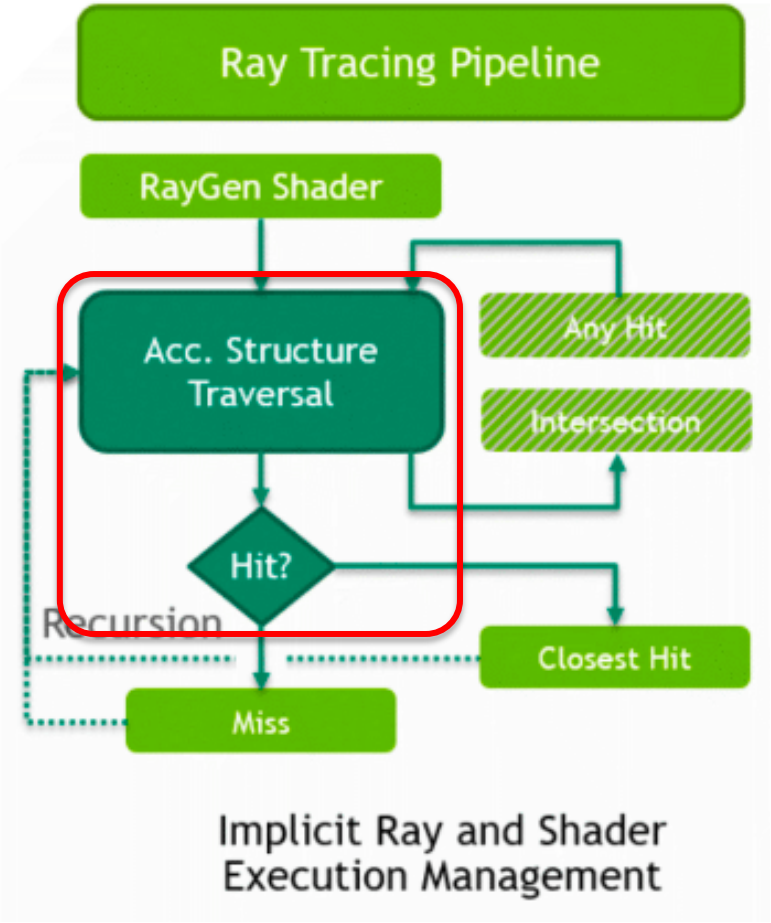
The *Miss* shader is instead called if the ray does not hit any object: generally it is used to draw some sort of background that should appear behind everything else.



The Ray-tracing pipeline

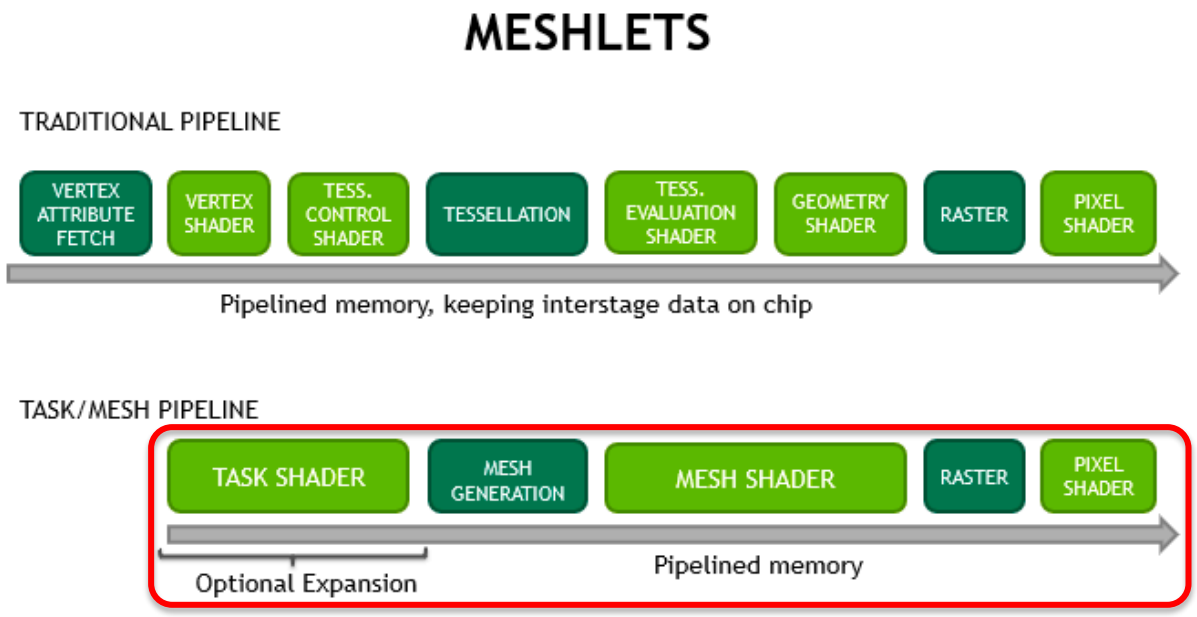
The fixed part of the pipeline controls the acceleration structure traversal, and the determination of the closest hit.

Although being powerful, it is very resource demanding and only today's top GPUs can offer decent performance with ray tracing.



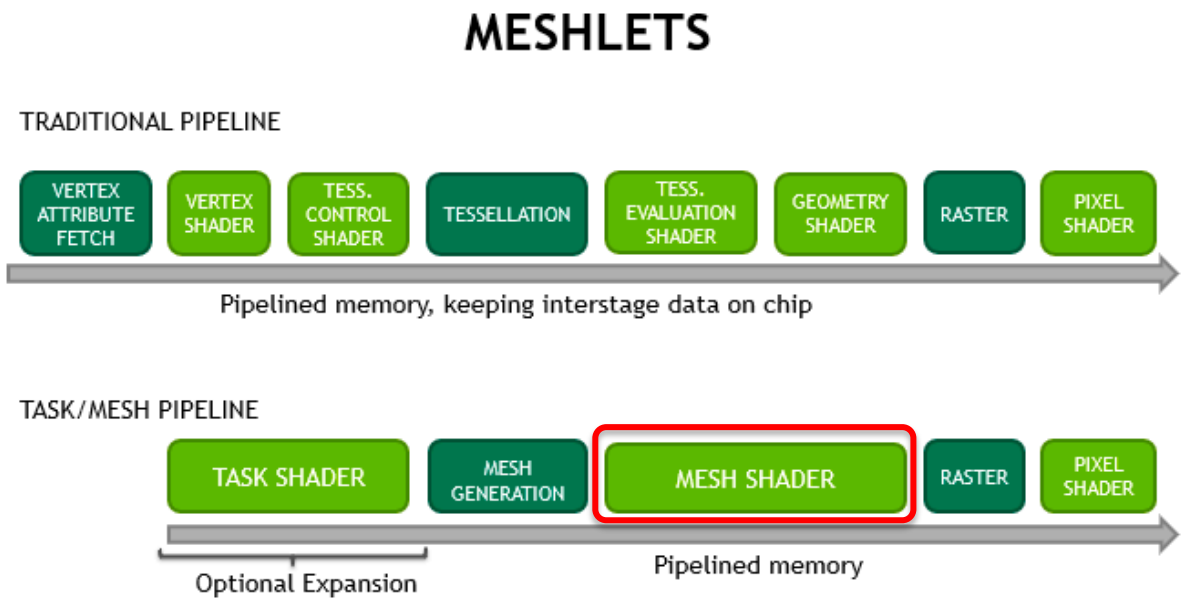
The Mesh Shader pipeline

The *Mesh Shader* pipeline is a generalization of the graphics pipeline where no initial fixed part of the pipeline is considered and mesh generation can be entirely handled by shaders.



The Mesh Shader pipeline

Mesh shaders computes indexed triangle lists, returnd as a set of vertices and groups of three indices for each triangle, to the rasterization part of the pipeline. Vertices are in normalized screen coordinates.

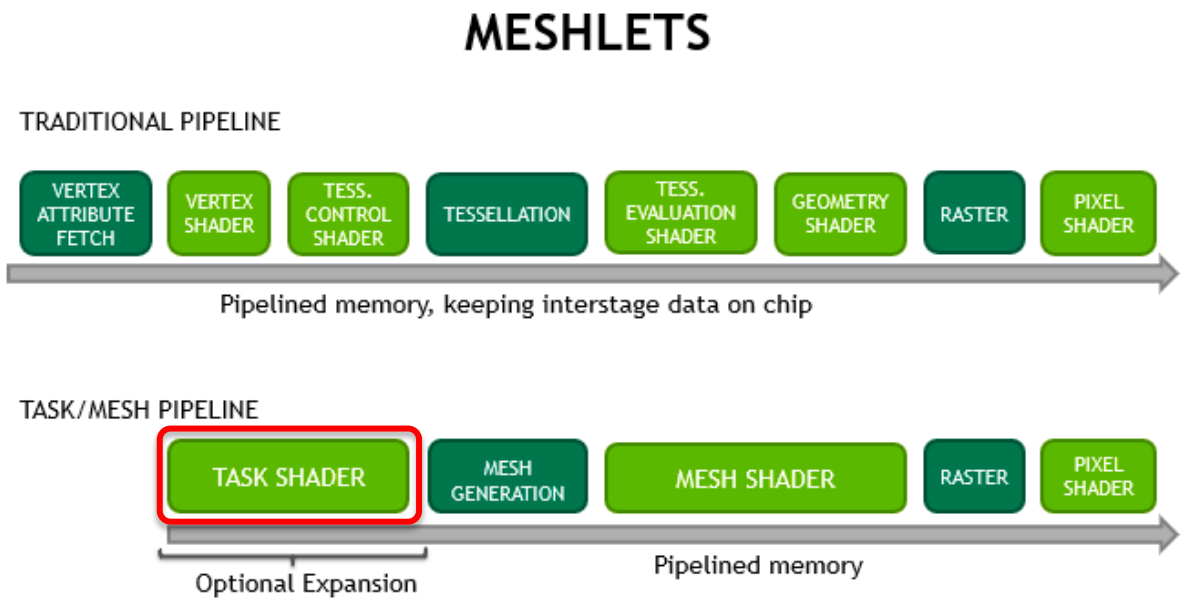


The Mesh Shader pipeline

The number of vertices and triangles that a Mesh shader can generate is limited. For this reason each object is divided in so called *Meshlets:* small patches of a mesh.

The Mesh Shader pipeline

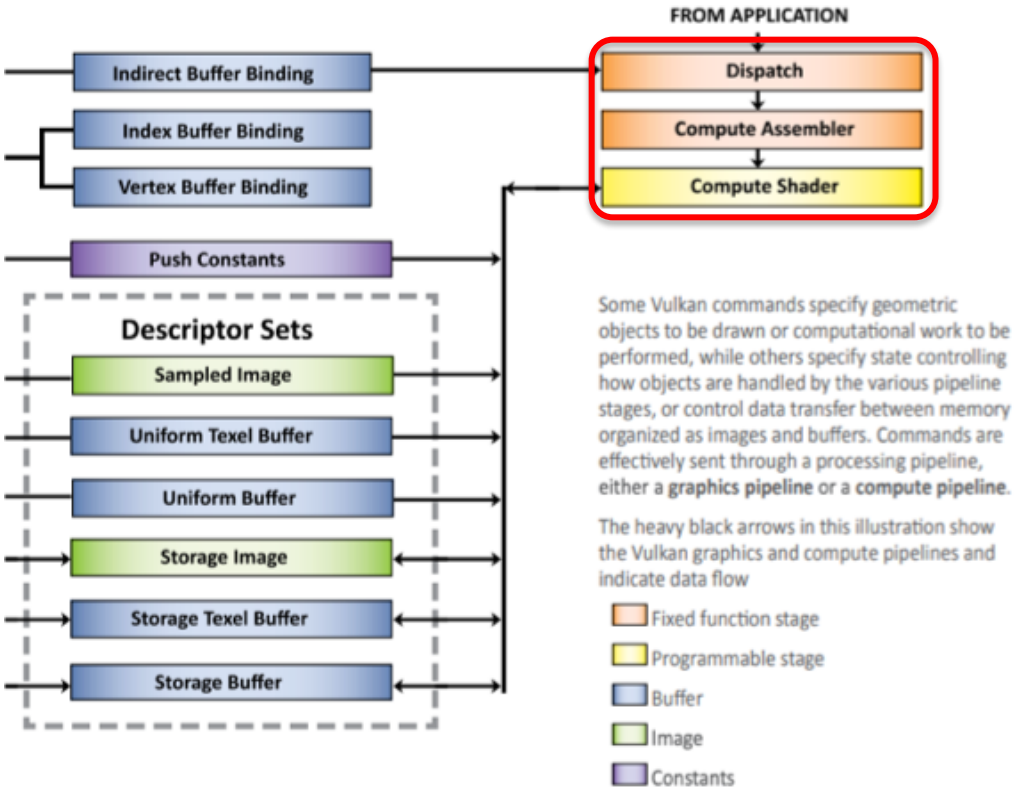
The purpose of the optional *Task* shader is to subdivide a larger mesh into smaller Meshlets, and control the corresponding *Mesh* shader for generating all the required patches.



The Compute pipeline

The *Compute* pipeline is not for rendering images, but for performing GPGPU computing.

In this case the application provides a single shader, the *Compute* Shader, that performs the desired computations.

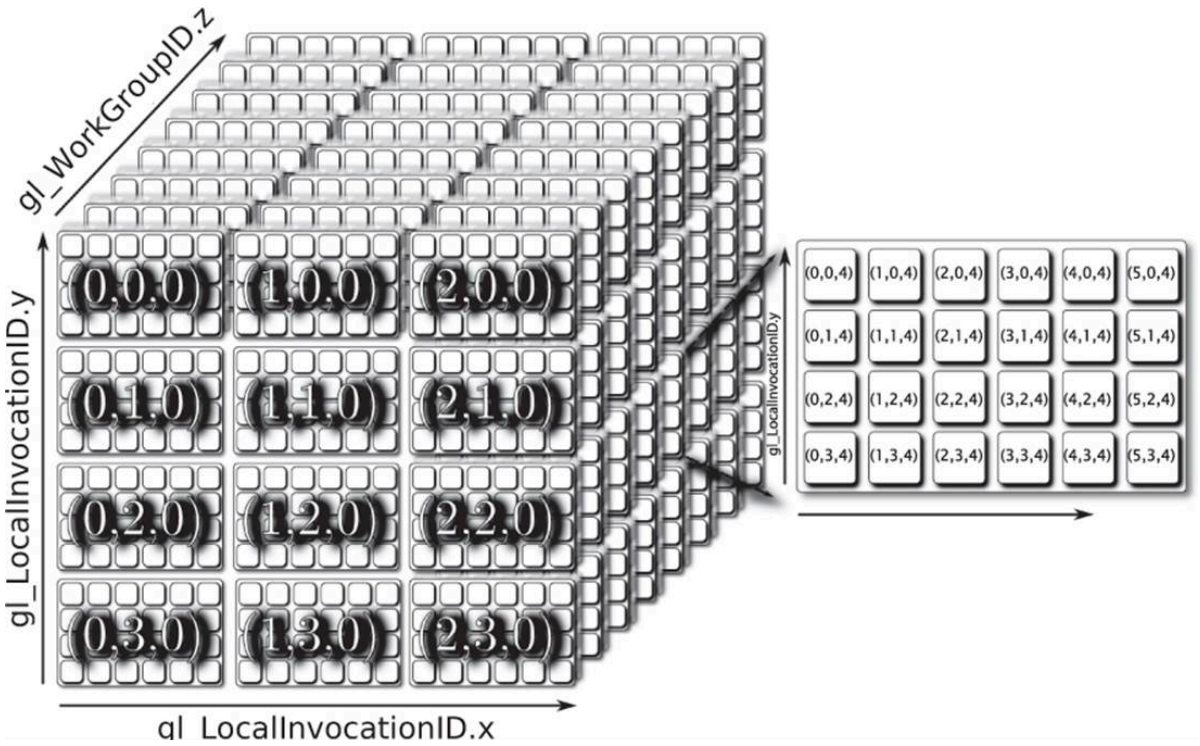


The Compute pipeline

Data is copied into buffers in the GPU memory (if available), and Compute shaders executions are identified by a (up to) tridimensional index.

Using this (vector) index, the shader can refer the data to find the partition on which it can work.

Follow the GPU course, if you are more interested!

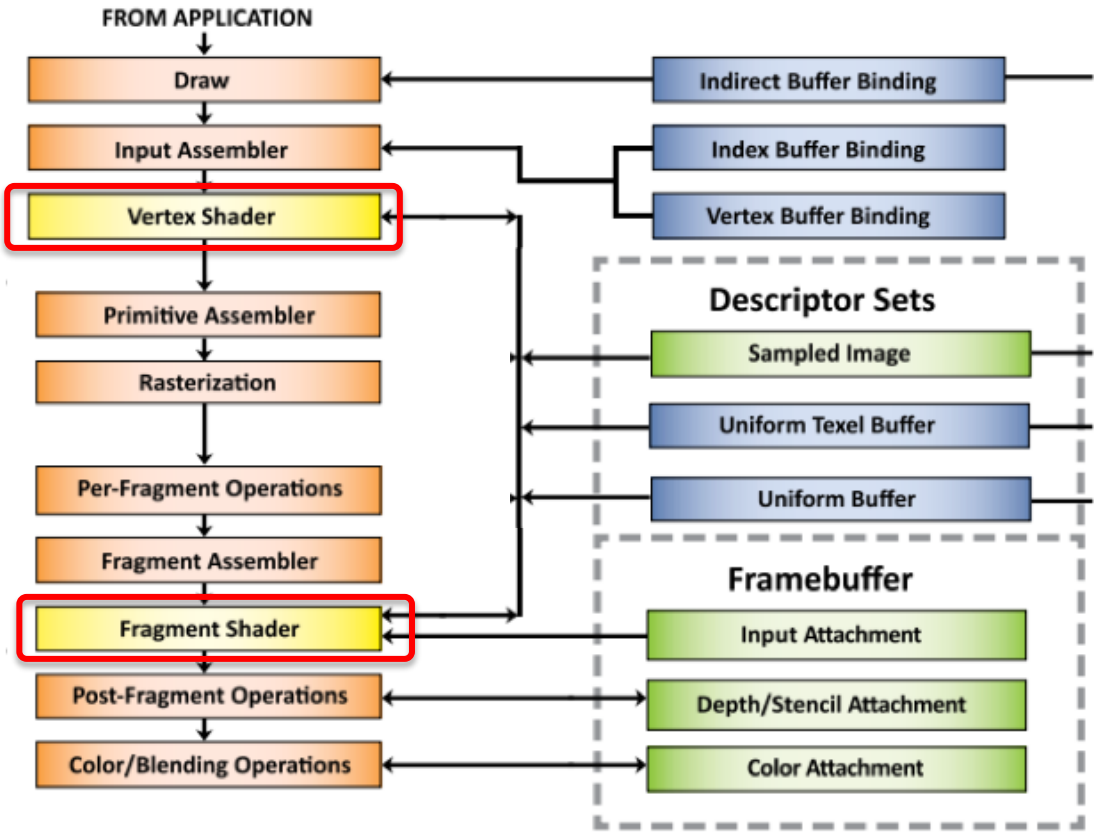


Introduction to GLSL

GLSL is the native shader language for OpenGL, and the official one for Vulkan.

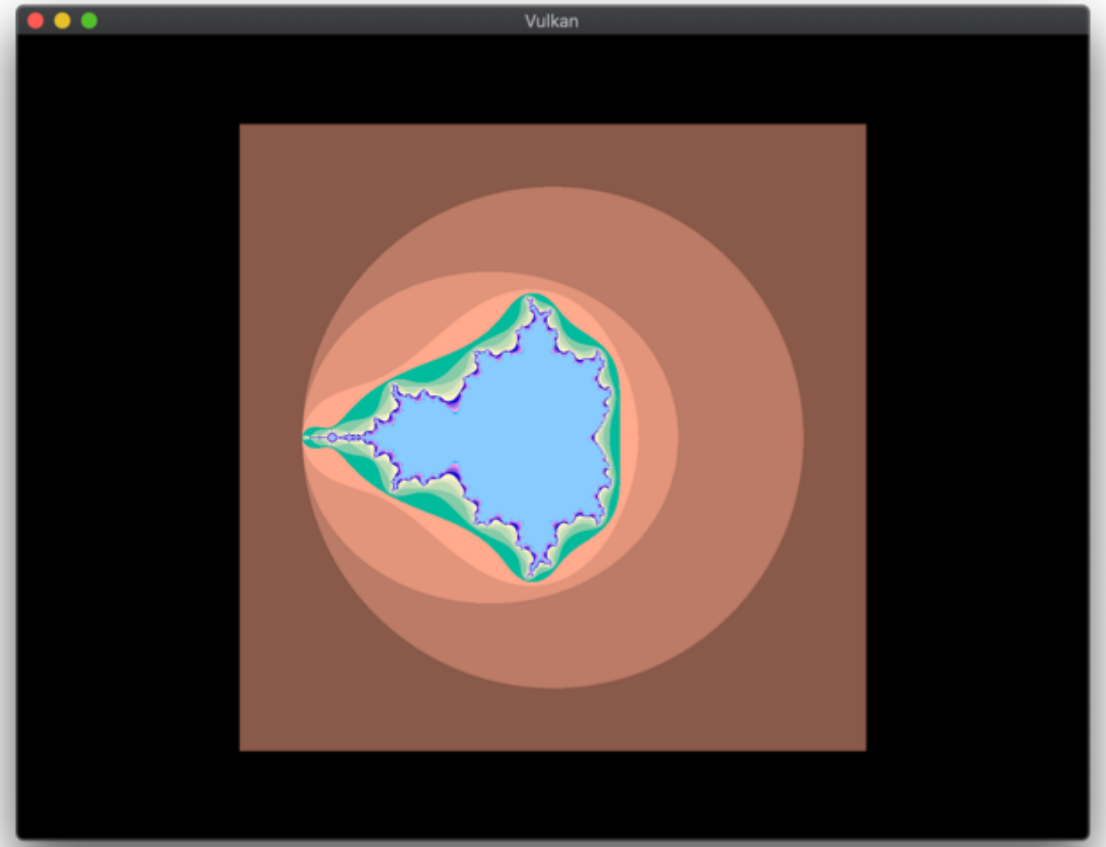
It is a C-like language, so it should be relatively easy to learn for people know C, C++, C#, Java or Javascript.

Although Vulkan supports a large number of different types of shaders, we will focus on a simplified pipeline using only *Vertex* and *Fragment* shaders.



Running example

To show the features, we will describe an example that computes the *Mandelbrot set* (the most famous fractal) using vertex and fragment shaders.



Program structure

Shaders follow the classical convention, having global variables and functions.

The entry point of the shader can be user defined in the code calling it: however it is generally called `main()`.

Vertex shader

Global definitions

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```

Program structure

Since shader compilers can compile different versions of GLSL shaders, for making them compatible with Vulkan, the must state the required language version, usually *4.5*.

Vertex shader

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```

Comments

Comments in code follows the classical *C* notations:

// for line comment

/* begins a comment block

*/ ends a comment block

Vertex shader

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```

Blocks

Blocks are also denoted using { }

Vertex shader

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```

Variables

Variables are typed and follows the same C naming convention (case sensitive, allowing letters, numbers and underscore, cannot start with number).

Variables are local to the block they are defined in.

Fragment shader

```
#version 450

layout(location = 0) in float real;
layout(location = 1) in float img;

layout(location = 0) out vec4 outColor;

layout(set = 0, binding = 1) uniform
GlobalUniformBufferObject {
      float time;
} gubo;

// The main procedure
void main() {
      float m_real = 0.0f, m_img = 0.0f, temp;
      int i;

      for(i = 0; i < 16; i++) {
            if(m_real * m_real + m_img * m_img > 4.0) {
                  break;
            }
            temp = m_real * m_real - m_img * m_img + real;
            m_img = 2.0 * m_real * m_img + img;
            m_real = temp;
      }
      outColor =
          vec4((float(i % 5) + sin(gubo.time*6.28)) / 5.0,
                float(i % 10) / 10.0, float(i) / 15.0, 1.0);
}
```

Variables

Beside type and name, variables can be preceded with a large number of different qualifiers.

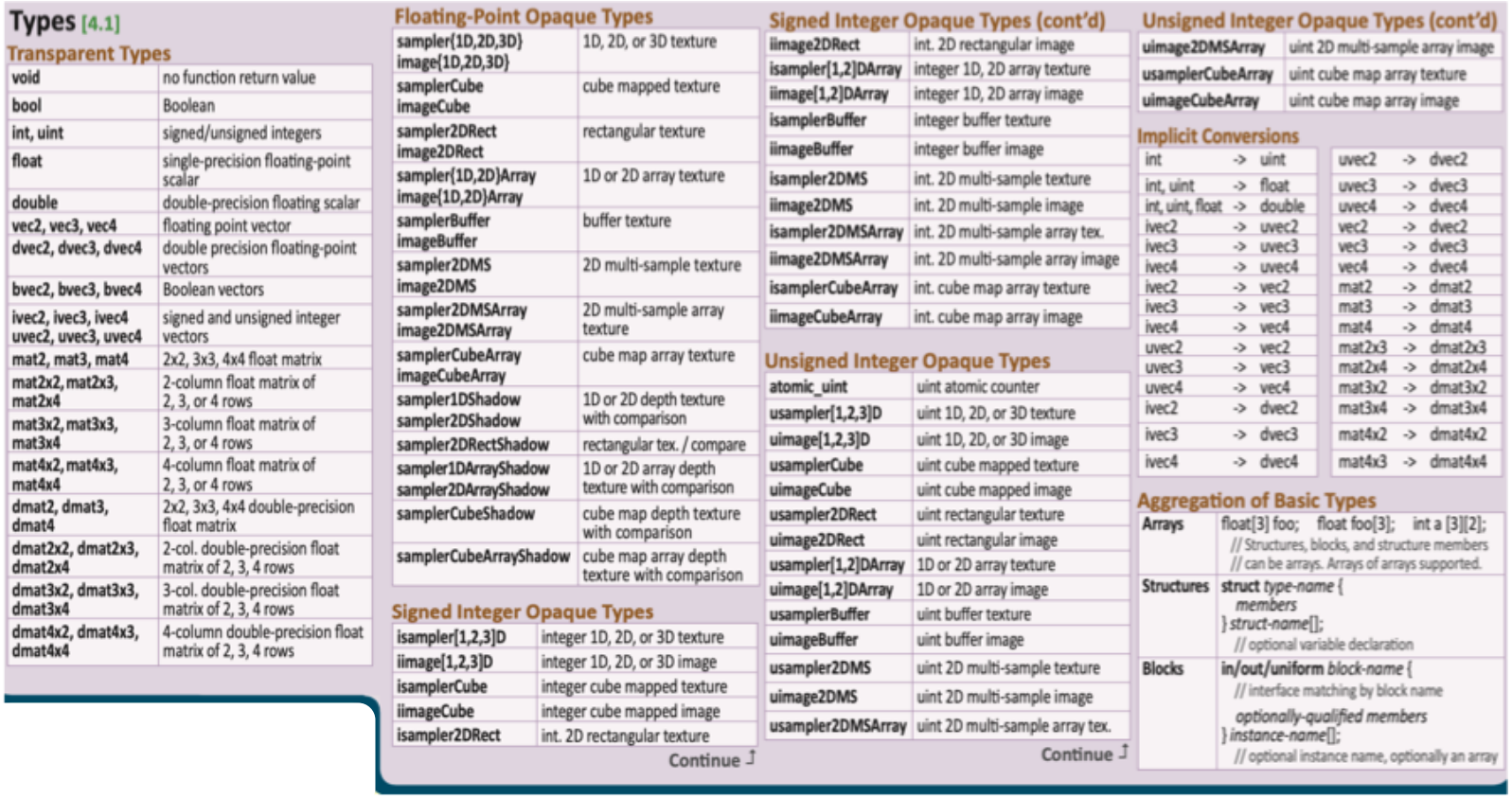
Most of them are required to interface them with the pipeline: we will return on that later.

Fragment shader

```
#version 450

layout(location = 0) in float real;
layout(location = 1) in float img;

layout(location = 0) out vec4 outColor;

layout(set = 0, binding = 1) uniform
GlobalUniformBufferObject {
      float time;
} gubo;

// The main procedure
void main() {
      float m_real = 0.0f, m_img = 0.0f, temp;
      int i;

      for(i = 0; i < 16; i++) {
            if(m_real * m_real + m_img * m_img > 4.0) {
                  break;
            }
            temp = m_real * m_real - m_img * m_img + real;
            m_img = 2.0 * m_real * m_img + img;
            m_real = temp;
      }
      outColor =
          vec4((float(i % 5) + sin(gubo.time*6.28)) / 5.0,
                float(i % 10) / 10.0, float(i) / 15.0, 1.0);
}
```

Data types

GLSL supports a large number of types:



Types [4.1]

Transparent Types

void	no function return value
bool	Boolean
int, uint	signed/unsigned integers
float	single-precision floating-point scalar
double	double-precision floating scalar
vec2, vec3, vec4	floating point vector
dvec2, dvec3, dvec4	double precision floating-point vectors
bvec2, bvec3, bvec4	Boolean vectors
ivec2, ivec3, ivec4 uvec2, uvec3, uvec4	signed and unsigned integer vectors
mat2, mat3, mat4	2x2, 3x3, 4x4 float matrix
mat2x2, mat2x3, mat2x4	2-column float matrix of 2, 3, or 4 rows
mat3x2, mat3x3, mat3x4	3-column float matrix of 2, 3, or 4 rows
mat4x2, mat4x3, mat4x4	4-column float matrix of 2, 3, or 4 rows
dmat2, dmat3, dmat4	2x2, 3x3, 4x4 double-precision float matrix
dmat2x2, dmat2x3, dmat2x4	2-col. double-precision float matrix of 2, 3, 4 rows
dmat3x2, dmat3x3, dmat3x4	3-col. double-precision float matrix of 2, 3, 4 rows
dmat4x2, dmat4x3, dmat4x4	4-column double-precision float matrix of 2, 3, 4 rows

Floating-Point Opaque Types

sampler{1D,2D,3D} image{1D,2D,3D}	1D, 2D, or 3D texture
samplerCube imageCube	cube mapped texture
sampler2DRect image2DRect	rectangular texture
sampler{1D,2D}Array image{1D,2D}Array	1D or 2D array texture
samplerBuffer imageBuffer	buffer texture
sampler2DMS image2DMS	2D multi-sample texture
sampler2DMSArray image2DMSArray	2D multi-sample array texture
samplerCubeArray imageCubeArray	cube map array texture
sampler1DShadow sampler2DShadow	1D or 2D depth texture with comparison
sampler2DRectShadow	rectangular tex. / compare
sampler1DArrayShadow sampler2DArrayShadow	1D or 2D array depth texture with comparison
samplerCubeShadow	cube map depth texture with comparison
samplerCubeArrayShadow	cube map array depth texture with comparison

Signed Integer Opaque Types

isampler[1,2,3]D	integer 1D, 2D, or 3D texture
iimage[1,2,3]D	integer 1D, 2D, or 3D image
isamplerCube	integer cube mapped texture
iimageCube	integer cube mapped image
isampler2DRect	int. 2D rectangular texture

Continue

Signed Integer Opaque Types (cont'd)

iimage2DRect	int. 2D rectangular image
isampler[1,2]DArray	integer 1D, 2D array texture
iimage[1,2]DArray	integer 1D, 2D array image
isamplerBuffer	integer buffer texture
iimageBuffer	integer buffer image
isampler2DMS	int. 2D multi-sample texture
iimage2DMS	int. 2D multi-sample image
isampler2DMSArray	int. 2D multi-sample array tex.
iimage2DMSArray	int. 2D multi-sample array image
isamplerCubeArray	int. cube map array texture
iimageCubeArray	int. cube map array image

Unsigned Integer Opaque Types

atomic_uint	uint atomic counter
usampler[1,2,3]D	uint 1D, 2D, or 3D texture
uimage[1,2,3]D	uint 1D, 2D, or 3D image
usamplerCube	uint cube mapped texture
uimageCube	uint cube mapped image
usampler2DRect	uint rectangular texture
uimage2DRect	uint rectangular image
usampler[1,2]DArray	1D or 2D array texture
uimage[1,2]DArray	1D or 2D array image
usamplerBuffer	uint buffer texture
uimageBuffer	uint buffer image
usampler2DMS	uint 2D multi-sample texture
uimage2DMS	uint 2D multi-sample image
usampler2DMSArray	uint 2D multi-sample array tex.

Continue

Unsigned Integer Opaque Types (cont'd)

uimage2DMSArray	uint 2D multi-sample array image
usamplerCubeArray	uint cube map array texture
uimageCubeArray	uint cube map array image

Implicit Conversions

int	->	uint	uvec2	->	dvec2
int, uint	->	float	uvec3	->	dvec3
int, uint, float	->	double	uvec4	->	dvec4
ivec2	->	uvec2	vec2	->	dvec2
ivec3	->	uvec3	vec3	->	dvec3
ivec4	->	uvec4	vec4	->	dvec4
ivec2	->	vec2	mat2	->	dmat2
ivec3	->	vec3	mat3	->	dmat3
ivec4	->	vec4	mat4	->	dmat4
uvec2	->	vec2	mat2x3	->	dmat2x3
uvec3	->	vec3	mat2x4	->	dmat2x4
uvec4	->	vec4	mat3x2	->	dmat3x2
ivec2	->	dvec2	mat3x4	->	dmat3x4
ivec3	->	dvec3	mat4x2	->	dmat4x2
ivec4	->	dvec4	mat4x3	->	dmat4x4

Aggregation of Basic Types

Arrays

```
float[3] foo;   float foo[3];   int a [3][2];
// Structures, blocks, and structure members
// can be arrays. Arrays of arrays supported.
```

Structures

```
struct type-name {
  members
} struct-name[];
  // optional variable declaration
```

Blocks

```
in/out/uniform block-name {
  // interface matching by block name
  optionally-qualified members
} instance-name[];
  // optional instance name, optionally an array
```

Data types

The most common scalar types are the following:

Types [4.1]

Transparent Types

void	no function return value
bool	Boolean
int, uint	signed/unsigned integers
float	single-precision floating-point scalar
double	double-precision floating scalar

Vectors

GLSL has also types for containing vectors of 2, 3 or 4 components.

`vecn` floating points vectors are the most widely used to describe colors and coordinates.

vec2, vec3, vec4	floating point vector
dvec2, dvec3, dvec4	double precision floating-point vectors
bvec2, bvec3, bvec4	Boolean vectors
ivec2, ivec3, ivec4 **uvec2, uvec3, uvec4**	signed and unsigned integer vectors

Vector elements

Vector elements can be accessed individually using the “dot” syntax:

Vertex shader

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```

Vector elements

Since vectors can be used to store coordinates, colors or texture related feature, several aliases exist for each component:

```
light.x  =        light.r  =        light.s  =        1.0
light.y  =        light.g  =        light.t  =        0.9
light.z  =        light.b  =        light.p  =        0.5
light.w  =        light.a  =        light.q  =        1.0
```

More than one letter can be used to refer to more elements…

```
vec3 l1 = light.xyz;
vec2 l2 = light.rb;
```

Elements order can be mixed:

```
light.zxy = light.xyz;
```

Matrices

Matrix types are defined as well: again the most commonly used are the 2x2, 3x3 or 4x4 composed by single precision elements.

mat2, mat3, mat4	2x2, 3x3, 4x4 float matrix
mat2x2, mat2x3, mat2x4	2-column float matrix of 2, 3, or 4 rows
mat3x2, mat3x3, mat3x4	3-column float matrix of 2, 3, or 4 rows
mat4x2, mat4x3, mat4x4	4-column float matrix of 2, 3, or 4 rows
dmat2, dmat3, dmat4	2x2, 3x3, 4x4 double-precision float matrix
dmat2x2, dmat2x3, dmat2x4	2-col. double-precision float matrix of 2, 3, 4 rows
dmat3x2, dmat3x3, dmat3x4	3-col. double-precision float matrix of 2, 3, 4 rows
dmat4x2, dmat4x3, dmat4x4	4-column double-precision float matrix of 2, 3, 4 rows

Matrix elements

GLSL uses the column major encoding, and allows to access elements using inidices starting from zero in [ ][ ] brackets.

Matrix columns can also be accessed as vectors:

```
mat4 m;
m[1] = vec4(2.0); // sets the second column to all 2.0
m[0][0] = 1.0; // sets the upper left element to 1.0
m[2][3] = 2.0; // sets the 4th element of the third column to 2.0
```

GLM and GLSL types

The types used in GLM has been named to follow the equivalente GLSL conventions: this should simplify the interactions between shaders and the application code.

Algebraic operations

Algebraic operations and assignments, including the ones between vector and matrices, is done using the conventional operators.

Vertex shader

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```

GLSL operators

GLSL includes a large set of common C operators:

Operators and Expressions [5.1]

The following operators are numbered in order of precedence. Relational and equality operators evaluate to Boolean. Also See lessThan(), equal().

1.	( )	parenthetical grouping
2.	[ ] ( ) . ++ --	array subscript function call, constructor, structure field, selector, swizzle postfix increment and decrement
3.	++ -- + - ~ !	prefix increment and decrement unary
4.	* / %	multiplicative
5.	+ -	additive
6.	<< >>	bit-wise shift
7.	< > <= >=	relational
8.	== !=	equality
9.	&	bit-wise and
10.	^	bit-wise exclusive or
11.	|	bit-wise inclusive or
12.	&&	logical and
13.	^^	logical exclusive or
14.	| |	logical inclusive or
15.	? :	selects an entire operand
16.	= += -= *= /= %= <<= >>= &= ^= |=	assignment arithmetic assignments
17.	,	sequence

Vector and matrix literals

Matrix and vector element can be constructed using the *type name*, followed by the comma separated list of elements between ( )

Fragment shader

```
#version 450

layout(location = 0) in float real;
layout(location = 1) in float img;

layout(location = 0) out vec4 outColor;

layout(set = 0, binding = 1) uniform
GlobalUniformBufferObject {
      float time;
} gubo;

// The main procedure
void main() {
      float m_real = 0.0f, m_img = 0.0f, temp;
      int i;

      for(i = 0; i < 16; i++) {
            if(m_real * m_real + m_img * m_img > 4.0) {
                  break;
            }
            temp = m_real * m_real - m_img * m_img + real;
            m_img = 2.0 * m_real * m_img + img;
            m_real = temp;
      }
      outColor =
            vec4((float(i % 5) + sin(gubo.time*6.28)) / 5.0,
                  float(i % 10) / 10.0, float(i) / 15.0, 1.0);
}
```

Vector and matrix literals

Larger vectors can be composed adding elements to shorter ones.

Vertex shader

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```

Type Casting

The same syntax can also be used to perform explicit type casting, when available.

Fragment shader

```
#version 450

layout(location = 0) in float real;
layout(location = 1) in float img;

layout(location = 0) out vec4 outColor;

layout(set = 0, binding = 1) uniform
GlobalUniformBufferObject {
      float time;
} gubo;

// The main procedure
void main() {
      float m_real = 0.0f, m_img = 0.0f, temp;
      int i;

      for(i = 0; i < 16; i++) {
            if(m_real * m_real + m_img * m_img > 4.0) {
                  break;
            }
            temp = m_real * m_real - m_img * m_img + real;
            m_img = 2.0 * m_real * m_img + img;
            m_real = temp;
      }
      outColor =
          vec4((float(i % 5) + sin(gubo.time*6.28)) / 5.0,
                float(i % 10) / 10.0, float(i) / 15.0, 1.0);
}
```

Functions

Functions definition and call follows the classical C convention:

definition:

return_type name (args, …)

{…} // function body

call:

name(args, …)

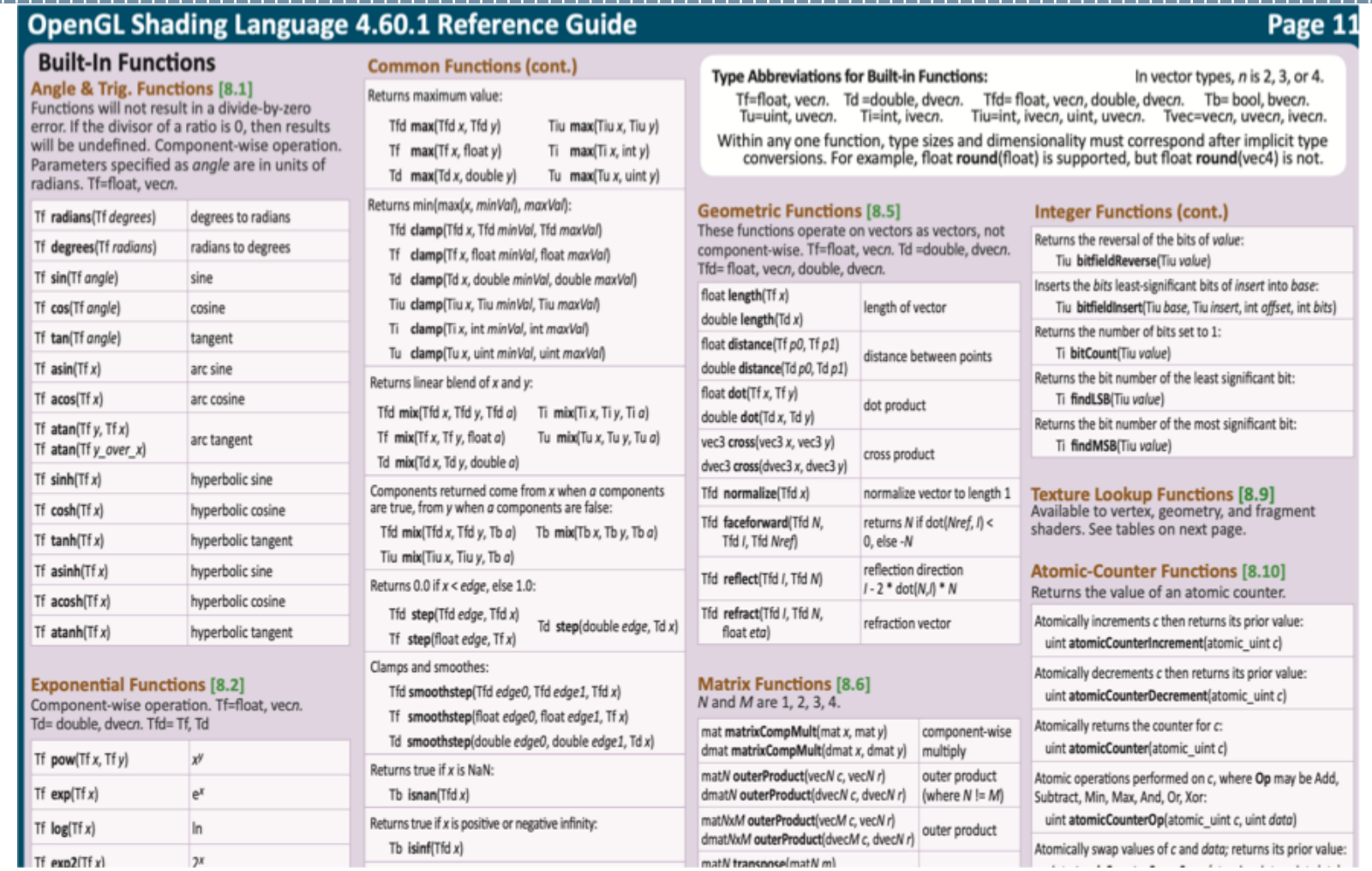
Fragment shader

```
#version 450

layout(location = 0) in float real;
layout(location = 1) in float img;

layout(location = 0) out vec4 outColor;

layout(set = 0, binding = 1) uniform
GlobalUniformBufferObject {
      float time;
} gubo;

// The main procedure
void main() {
      float m_real = 0.0f, m_img = 0.0f, temp;
      int i;

      for(i = 0; i < 16; i++) {
            if(m_real * m_real + m_img * m_img > 4.0) {
                  break;
            }
            temp = m_real * m_real - m_img * m_img + real;
            m_img = 2.0 * m_real * m_img + img;
            m_real = temp;
      }
      outColor =
          vec4((float(i % 5) + sin(gubo.time*6.28)) / 5.0,
                float(i % 10) / 10.0, float(i) / 15.0, 1.0);
}
```

Built in functions

And many more!

See the reference card for a complete list.



OpenGL Shading Language 4.60.1 Reference Guide

Built-In Functions

Angle & Trig. Functions [8.1]

Functions will not result in a divide-by-zero error. If the divisor of a ratio is 0, then results will be undefined. Component-wise operation. Parameters specified as *angle* are in units of radians. Tf=float, vec*n*.

Tf **radians**(Tf *degrees*)	degrees to radians
Tf **degrees**(Tf *radians*)	radians to degrees
Tf **sin**(Tf *angle*)	sine
Tf **cos**(Tf *angle*)	cosine
Tf **tan**(Tf *angle*)	tangent
Tf **asin**(Tf *x*)	arc sine
Tf **acos**(Tf *x*)	arc cosine
Tf **atan**(Tf *y*, Tf *x*) Tf **atan**(Tf *y_over_x*)	arc tangent
Tf **sinh**(Tf *x*)	hyperbolic sine
Tf **cosh**(Tf *x*)	hyperbolic cosine
Tf **tanh**(Tf *x*)	hyperbolic tangent
Tf **asinh**(Tf *x*)	hyperbolic sine
Tf **acosh**(Tf *x*)	hyperbolic cosine
Tf **atanh**(Tf *x*)	hyperbolic tangent

Exponential Functions [8.2]

Component-wise operation. Tf=float, vec*n*. Td= double, dvec*n*. Tfd= Tf, Td

Tf **pow**(Tf *x*, Tf *y*)	x^y
Tf **exp**(Tf *x*)	e^x
Tf **log**(Tf *x*)	ln
Tf **exp2**(Tf *x*)	2^x

Common Functions (cont.)

Returns maximum value:

Tfd **max**(Tfd *x*, Tfd *y*) — Tiu **max**(Tiu *x*, Tiu *y*)
Tf **max**(Tf *x*, float *y*) — Ti **max**(Ti *x*, int *y*)
Td **max**(Td *x*, double *y*) — Tu **max**(Tu *x*, uint *y*)

Returns min(max(*x*, *minVal*), *maxVal*):

Tfd **clamp**(Tfd *x*, Tfd *minVal*, Tfd *maxVal*)
Tf **clamp**(Tf *x*, float *minVal*, float *maxVal*)
Td **clamp**(Td *x*, double *minVal*, double *maxVal*)
Tiu **clamp**(Tiu *x*, Tiu *minVal*, Tiu *maxVal*)
Ti **clamp**(Ti *x*, int *minVal*, int *maxVal*)
Tu **clamp**(Tu *x*, uint *minVal*, uint *maxVal*)

Returns linear blend of *x* and *y*:

Tfd **mix**(Tfd *x*, Tfd *y*, Tfd *a*) — Ti **mix**(Ti *x*, Ti *y*, Ti *a*)
Tf **mix**(Tf *x*, Tf *y*, float *a*) — Tu **mix**(Tu *x*, Tu *y*, Tu *a*)
Td **mix**(Td *x*, Td *y*, double *a*)

Components returned come from *x* when *a* components are true, from *y* when *a* components are false:

Tfd **mix**(Tfd *x*, Tfd *y*, Tb *a*) — Tb **mix**(Tb *x*, Tb *y*, Tb *a*)
Tiu **mix**(Tiu *x*, Tiu *y*, Tb *a*)

Returns 0.0 if *x* < *edge*, else 1.0:

Tfd **step**(Tfd *edge*, Tfd *x*)
Tf **step**(float *edge*, Tf *x*) — Td **step**(double *edge*, Td *x*)

Clamps and smoothes:

Tfd **smoothstep**(Tfd *edge0*, Tfd *edge1*, Tfd *x*)
Tf **smoothstep**(float *edge0*, float *edge1*, Tf *x*)
Td **smoothstep**(double *edge0*, double *edge1*, Td *x*)

Returns true if *x* is NaN:

Tb **isnan**(Tfd *x*)

Returns true if *x* is positive or negative infinity:

Tb **isinf**(Tfd *x*)

Type Abbreviations for Built-in Functions: In vector types, *n* is 2, 3, or 4.

Tf=float, vec*n*. Td =double, dvec*n*. Tfd= float, vec*n*, double, dvec*n*. Tb= bool, bvec*n*. Tu=uint, uvec*n*. Ti=int, ivec*n*. Tiu=int, ivec*n*, uint, uvec*n*. Tvec=vec*n*, uvec*n*, ivec*n*.

Within any one function, type sizes and dimensionality must correspond after implicit type conversions. For example, float **round**(float) is supported, but float **round**(vec4) is not.

Geometric Functions [8.5]

These functions operate on vectors as vectors, not component-wise. Tf=float, vec*n*. Td =double, dvec*n*. Tfd= float, vec*n*, double, dvec*n*.

float **length**(Tf *x*) double **length**(Td *x*)	length of vector
float **distance**(Tf *p0*, Tf *p1*) double **distance**(Td *p0*, Td *p1*)	distance between points
float **dot**(Tf *x*, Tf *y*) double **dot**(Td *x*, Td *y*)	dot product
vec3 **cross**(vec3 *x*, vec3 *y*) dvec3 **cross**(dvec3 *x*, dvec3 *y*)	cross product
Tfd **normalize**(Tfd *x*)	normalize vector to length 1
Tfd **faceforward**(Tfd *N*, Tfd *I*, Tfd *Nref*)	returns *N* if dot(*Nref*, *I*) < 0, else -*N*
Tfd **reflect**(Tfd *I*, Tfd *N*)	reflection direction *I* - 2 * dot(*N*,*I*) * *N*
Tfd **refract**(Tfd *I*, Tfd *N*, float *eta*)	refraction vector

Matrix Functions [8.6]

N and *M* are 1, 2, 3, 4.

mat **matrixCompMult**(mat *x*, mat *y*) dmat **matrixCompMult**(dmat *x*, dmat *y*)	component-wise multiply
mat*N* **outerProduct**(vec*N* *c*, vec*N* *r*) dmat*N* **outerProduct**(dvec*N* *c*, dvec*N* *r*)	outer product (where *N* != *M*)
mat*NxM* **outerProduct**(vec*M* *c*, vec*N* *r*) dmat*NxM* **outerProduct**(dvec*M* *c*, dvec*N* *r*)	outer product
mat*N* **transpose**(mat*N* *m*)	

Integer Functions (cont.)

Returns the reversal of the bits of *value*:
Tiu **bitfieldReverse**(Tiu *value*)

Inserts the *bits* least-significant bits of *insert* into *base*:
Tiu **bitfieldInsert**(Tiu *base*, Tiu *insert*, int *offset*, int *bits*)

Returns the number of bits set to 1:
Ti **bitCount**(Tiu *value*)

Returns the bit number of the least significant bit:
Ti **findLSB**(Tiu *value*)

Returns the bit number of the most significant bit:
Ti **findMSB**(Tiu *value*)

Texture Lookup Functions [8.9]

Available to vertex, geometry, and fragment shaders. See tables on next page.

Atomic-Counter Functions [8.10]

Returns the value of an atomic counter.

Atomically increments *c* then returns its prior value:
uint **atomicCounterIncrement**(atomic_uint *c*)

Atomically decrements *c* then returns its prior value:
uint **atomicCounterDecrement**(atomic_uint *c*)

Atomically returns the counter for *c*:
uint **atomicCounter**(atomic_uint *c*)

Atomic operations performed on *c*, where **Op** may be Add, Subtract, Min, Max, And, Or, Xor:
uint **atomicCounterOp**(atomic_uint *c*, uint *data*)

Atomically swap values of *c* and *data*; returns its prior value:

Conditional statements

The classical **`if else`** statement, and the ***`? :`*** in-line syntax can be used to control the execution of a procedure.

Fragment shader

```
#version 450

layout(location = 0) in float real;
layout(location = 1) in float img;

layout(location = 0) out vec4 outColor;

layout(set = 0, binding = 1) uniform
GlobalUniformBufferObject {
      float time;
} gubo;

// The main procedure
void main() {
      float m_real = 0.0f, m_img = 0.0f, temp;
      int i;

      for(i = 0; i < 16; i++) {
            if(m_real * m_real + m_img * m_img > 4.0) {
                  break;
            }
            temp = m_real * m_real - m_img * m_img + real;
            m_img = 2.0 * m_real * m_img + img;
            m_real = temp;
      }
      outColor =
          vec4((float(i % 5) + sin(gubo.time*6.28)) / 5.0,
                float(i % 10) / 10.0, float(i) / 15.0, 1.0);
}
```

Loops

`for()` and **`while()`** loops are also available:

Iteration and Jumps [6.3-4]

Function	call by value-return

Iteration	for (;;) { break, continue }
	while ( ) { break, continue }
	do { break, continue } while ( );

Selection	if ( ) { }
	if ( ) { } else { }
	switch ( ) { case integer: ... break; ...
	default: ... }

Entry	void main()
Jump	break, continue, return (There is no 'goto')
Exit	return in main() discard // Fragment shader only

Fragment shader

```
#version 450

layout(location = 0) in float real;
layout(location = 1) in float img;

layout(location = 0) out vec4 outColor;

layout(set = 0, binding = 1) uniform
GlobalUniformBufferObject {
      float time;
} gubo;

// The main procedure
void main() {
      float m_real = 0.0f, m_img = 0.0f, temp;
      int i;

      for(i = 0; i < 16; i++) {
            if(m_real * m_real + m_img * m_img > 4.0) {
                  break;
            }
            temp = m_real * m_real - m_img * m_img + real;
            m_img = 2.0 * m_real * m_img + img;
            m_real = temp;
      }
      outColor =
          vec4((float(i % 5) + sin(gubo.time*6.28)) / 5.0,
                float(i % 10) / 10.0, float(i) / 15.0, 1.0);
}
```

Loop exit statements

Statements for exiting loops are also available.

Iteration and Jumps [6.3-4]

Function	call by value-return

Iteration	for (;;) { break, continue }
	while ( ) { break, continue }
	do { break, continue } while ( );

Selection	if ( ) { }
	if ( ) { } else { }
	switch ( ) { case integer: ... break; ...
	default: ... }

Entry	void main()
Jump	break, continue, return (There is no 'goto')
Exit	return in main() discard // Fragment shader only

Fragment shader

```
#version 450

layout(location = 0) in float real;
layout(location = 1) in float img;

layout(location = 0) out vec4 outColor;

layout(set = 0, binding = 1) uniform
GlobalUniformBufferObject {
      float time;
} gubo;

// The main procedure
void main() {
      float m_real = 0.0f, m_img = 0.0f, temp;
      int i;

      for(i = 0; i < 16; i++) {
            if(m_real * m_real + m_img * m_img > 4.0) {
                  break;
            }
            temp = m_real * m_real - m_img * m_img + real;
            m_img = 2.0 * m_real * m_img + img;
            m_real = temp;
      }
      outColor =
          vec4((float(i % 5) + sin(gubo.time*6.28)) / 5.0,
                float(i % 10) / 10.0, float(i) / 15.0, 1.0);
}
```

Note on flow control

Please note that flow control statements on GPU behaves differently than on CPU, due to its SIMD architecture: many elements are always processed at the same time!

- Both the *if* and *else* branches are always executed.
- In variable length loops, all executions are conditioned by the longest one.

This is why it is always a good idea trying to avoid conditional statements as much as possible.

Shader-pipeline communication

Communication between the Shaders and the Pipeline occurs through global variables.

Vertex shader

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```

Shader-pipeline communication: *in* and *out*

in and out variables are used to interface with the programmable or configurable part of the pipeline.

We will consider the mechanism in detail in the following lessons.

Vertex shader

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```

Shader-pipeline communication: built-in variables

Communication with the fixed part of the pipeline occurs using predefined global variables.

For example, in a Vertex shader, *gl_Position* is a *vec4* variable that must be filled with the clipping coordinates of the corresponding vertex.

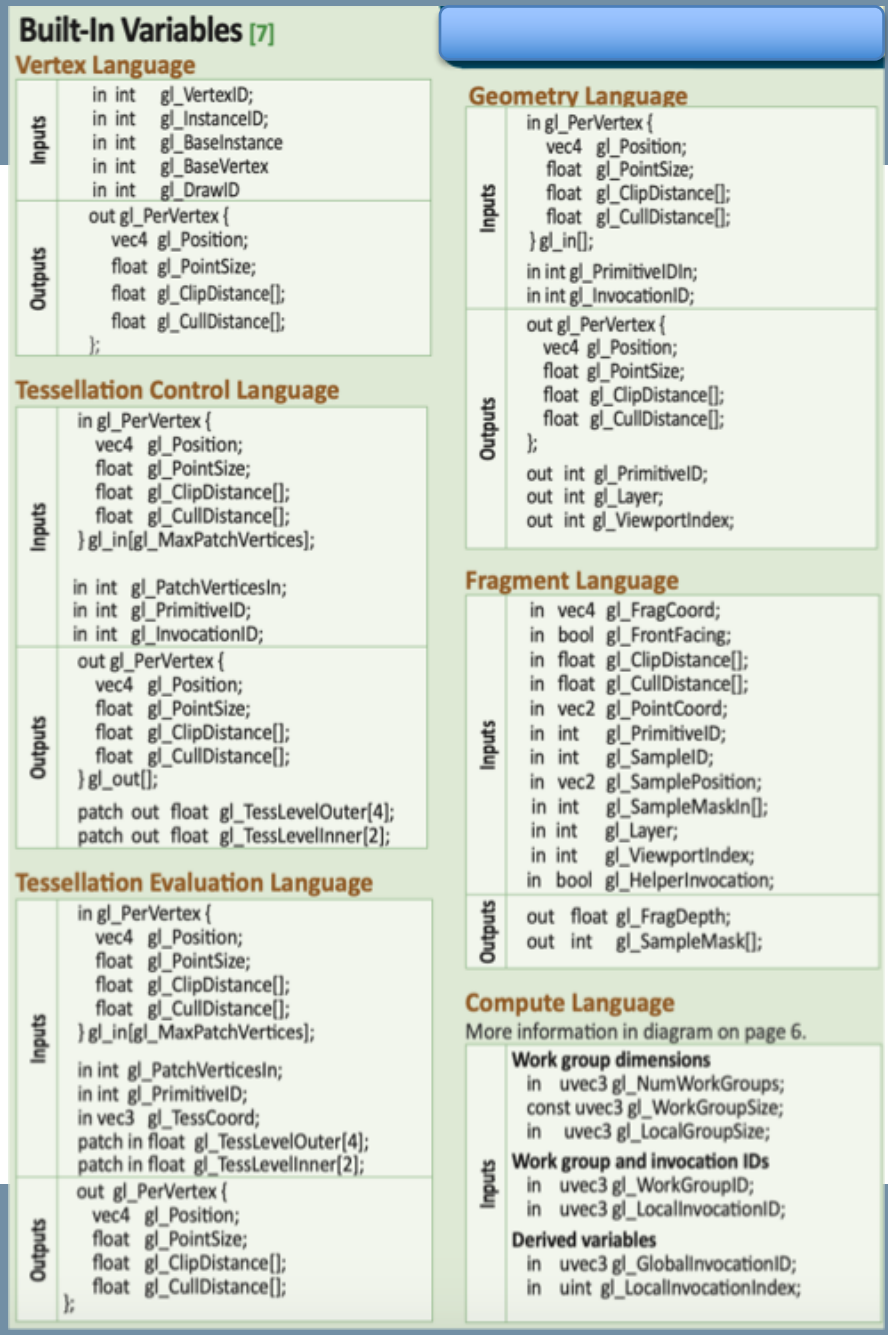
Vertex shader

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```

Shader-pipeline communication: built-in variables

Although *gl_Position* is the most important, which is always required in any applications using the graphic pipeline, other variable exits to interface other shaders types.

For time constraints, we will not cover them in detail.



Built-In Variables [7]

Vertex Language

Inputs:
```
in int   gl_VertexID;
in int   gl_InstanceID;
in int   gl_BaseInstance
in int   gl_BaseVertex
in int   gl_DrawID
```

Outputs:
```
out gl_PerVertex {
   vec4  gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
   float gl_CullDistance[];
};
```

Tessellation Control Language

Inputs:
```
in gl_PerVertex {
   vec4  gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
   float gl_CullDistance[];
} gl_in[gl_MaxPatchVertices];

in int  gl_PatchVerticesIn;
in int  gl_PrimitiveID;
in int  gl_InvocationID;
```

Outputs:
```
out gl_PerVertex {
   vec4  gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
   float gl_CullDistance[];
} gl_out[];

patch out float gl_TessLevelOuter[4];
patch out float gl_TessLevelInner[2];
```

Tessellation Evaluation Language

Inputs:
```
in gl_PerVertex {
   vec4  gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
   float gl_CullDistance[];
} gl_in[gl_MaxPatchVertices];

in int  gl_PatchVerticesIn;
in int  gl_PrimitiveID;
in vec3 gl_TessCoord;
patch in float gl_TessLevelOuter[4];
patch in float gl_TessLevelInner[2];
```

Outputs:
```
out gl_PerVertex {
   vec4  gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
   float gl_CullDistance[];
};
```

Geometry Language

Inputs:
```
in gl_PerVertex {
   vec4  gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
   float gl_CullDistance[];
} gl_in[];
in int gl_PrimitiveIDIn;
in int gl_InvocationID;
```

Outputs:
```
out gl_PerVertex {
   vec4  gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
   float gl_CullDistance[];
};
out int gl_PrimitiveID;
out int gl_Layer;
out int gl_ViewportIndex;
```

Fragment Language

Inputs:
```
in  vec4  gl_FragCoord;
in  bool  gl_FrontFacing;
in  float gl_ClipDistance[];
in  float gl_CullDistance[];
in  vec2  gl_PointCoord;
in  int   gl_PrimitiveID;
in  int   gl_SampleID;
in  vec2  gl_SamplePosition;
in  int   gl_SampleMaskIn[];
in  int   gl_Layer;
in  int   gl_ViewportIndex;
in  bool  gl_HelperInvocation;
```

Outputs:
```
out  float gl_FragDepth;
out  int   gl_SampleMask[];
```

Compute Language

More information in diagram on page 6.

Inputs:
```
Work group dimensions
   in   uvec3 gl_NumWorkGroups;
   const uvec3 gl_WorkGroupSize;
   in   uvec3 gl_LocalGroupSize;
Work group and invocation IDs
   in   uvec3 gl_WorkGroupID;
   in   uvec3 gl_LocalInvocationID;
Derived variables
   in   uvec3 gl_GlobalInvocationID;
   in   uint  gl_LocalInvocationIndex;
```

Shader-application communication

Communication between the Shaders and the application occurs using *Uniform Variables Blocks.*

Vertex shader

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```

Shader-application communication

Each block is similar to a C structure: it has assigned a tag and a name, and it contains a set of components.

In the following lessons, we will focus on the way in which blocks are connected between the shaders and the application.

Vertex shader

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```

Shader-application communication

Elements of the blocks are accessed in a Shader program exactly as structure fields in C code.

Vertex shader

```
#version 450

layout(set = 0, binding = 0) uniform
UniformBufferObject {
    mat4 worldMat;
    mat4 vpMat;
} ubo;

layout(location = 0) in vec3 inPosition;

layout(location = 0) out float real;
layout(location = 1) out float img;

// The main procedure
void main() {
    gl_Position = ubo.vpMat * ubo.worldMat *
                  vec4(inPosition, 1.0);
    real = inPosition.x * 2.5;
    img  = inPosition.y * 2.5;
}
```