POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

DEIB

2024

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2024

Computer Graphics

- GLSL, Ray tracing and other Rendering techniques

Ray tracing

Ray tracing considers for each pixel also the light emitted by other objects in two specific directions: the *mirror reflection* and the *refraction* (for transparent objects).

$$L(x, \omega_r) = L_e(x, \omega_r) + \int L(y, \overrightarrow{yx}) f_r(x, \overrightarrow{yx}, \omega_r) G(x, y) V(x, y) dy$$

$$L(x, \omega_r) = L_e(x, \omega_r) + \sum_l L(l, \overrightarrow{lx}) f_{r'}(x, \overrightarrow{lx}, \omega_r) V(x, l) +$$

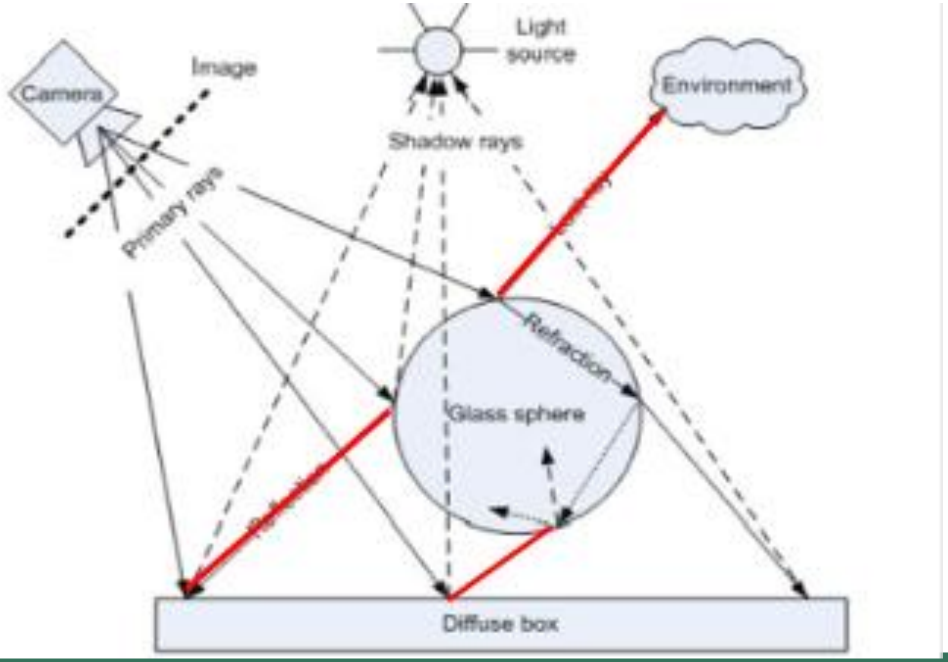
Reflection $\boxed{L(r, \overrightarrow{rx}) f_{r'}(x, \overrightarrow{rx}, \omega_r) V(x, r)} +$

Refraction $\boxed{L\left(t, \overrightarrow{tx'}\right) f_{t'}\left(x', \overrightarrow{tx'}, \omega_r\right) V(x, t)}$

Ray tracing

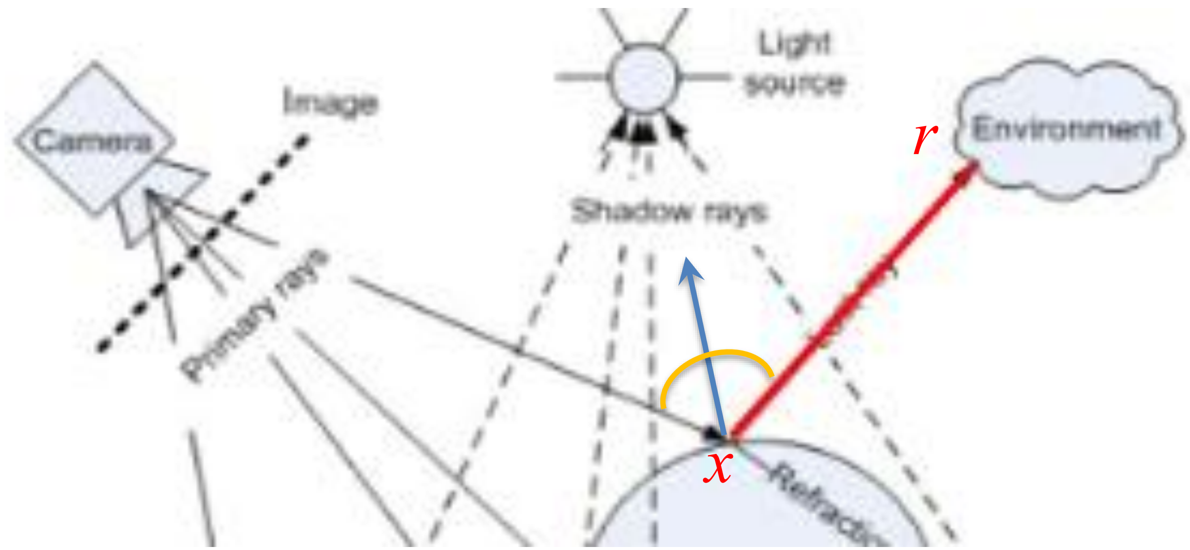
For reflection, this direction corresponds to the *mirror direction:* same angle, *but on the other side* with respect to the normal vector of the surface in the hit point.

This allows the reproduction of realistic perfect (mirror) reflections.



Ray tracing

In particular, for each point *x*, the algorithm looks for the points *r* on all the objects in the scene, along the mirror direction $\overrightarrow{rx}$, and selects the one closest to *x*.

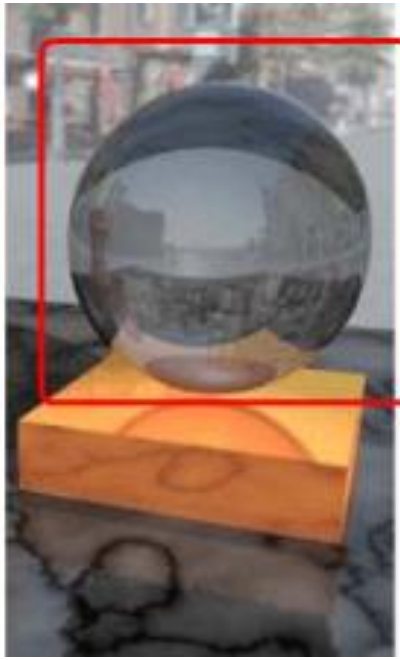


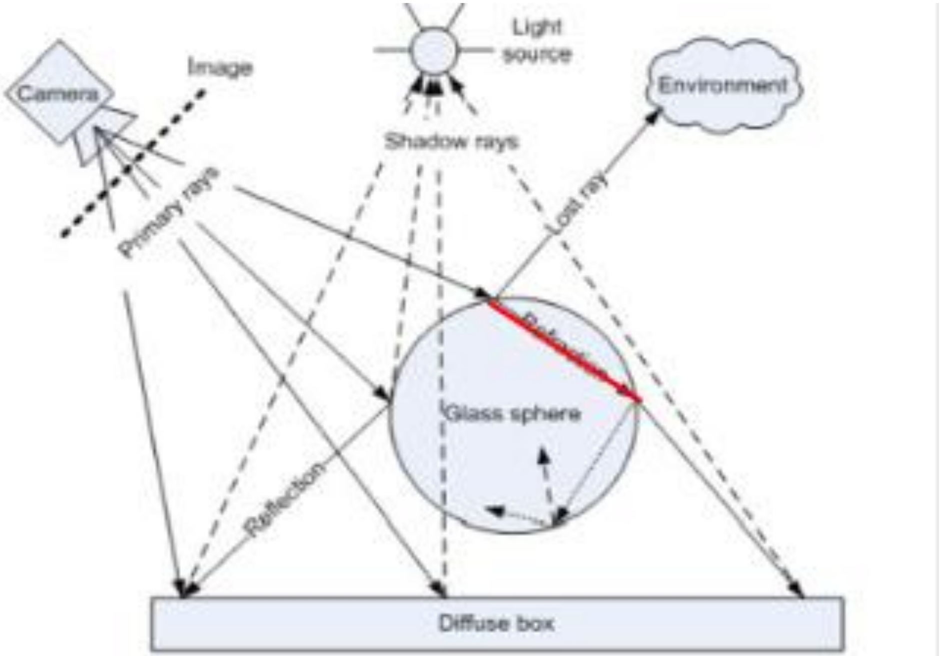
$$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L(l,\overrightarrow{lx}) f_{r'}(x,\overrightarrow{lx},\omega_r) V(x,l) +$$

Reflection $\boxed{L(r,\overrightarrow{rx}) f_{r'}(x,\overrightarrow{rx},\omega_r) V(x,r) +}$

$$L\left(t,\overrightarrow{tx'}\right) f_{t'}\left(x',\overrightarrow{tx'},\omega_r\right) V(x,t)$$

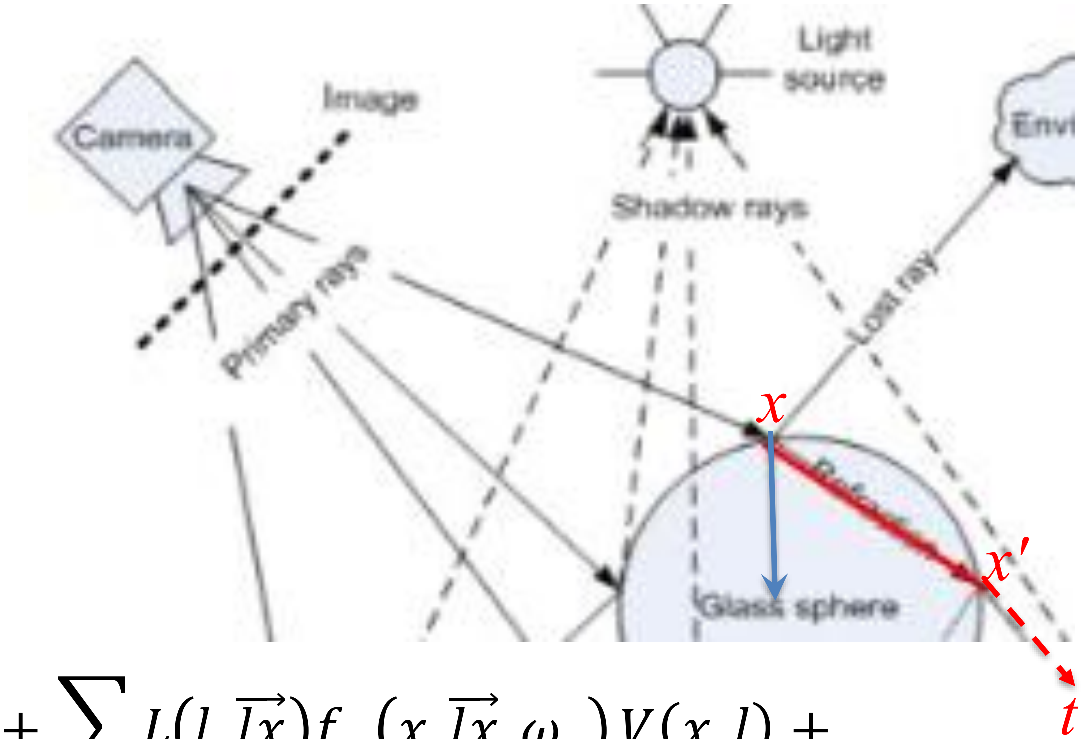
For refraction, the physical properties of the objects are emulated by considering the *index of refraction* for the material to determine the angle at which the refraction ray is cast.





Ray tracing

In this case, for each point x, the algorithm first searches for the point x' from which the ray will exit on the other side, by considering the different refraction indices of the solids separated by the surface; then it looks for the points t on all the objects along the direction $\overrightarrow{tx'}$.



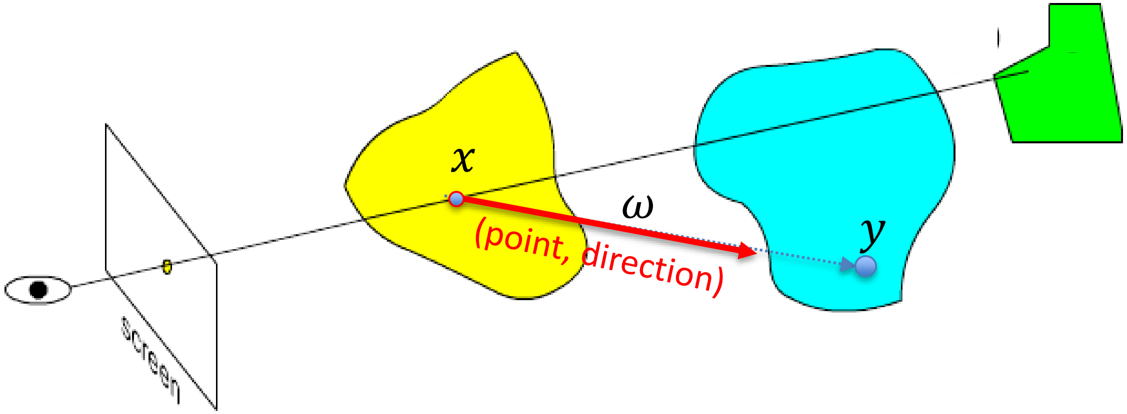
$$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L(l,\overrightarrow{lx}) f_{r'}(x,\overrightarrow{lx},\omega_r) V(x,l) +$$
$$L(r,\overrightarrow{rx}) f_{r'}(x,\overrightarrow{rx},\omega_r) V(x,r) +$$

Refraction $\boxed{L\left(t,\overrightarrow{tx'}\right) f_{t'}\left(x',\overrightarrow{tx'},\omega_r\right) V(x,t)}$

Ray tracing

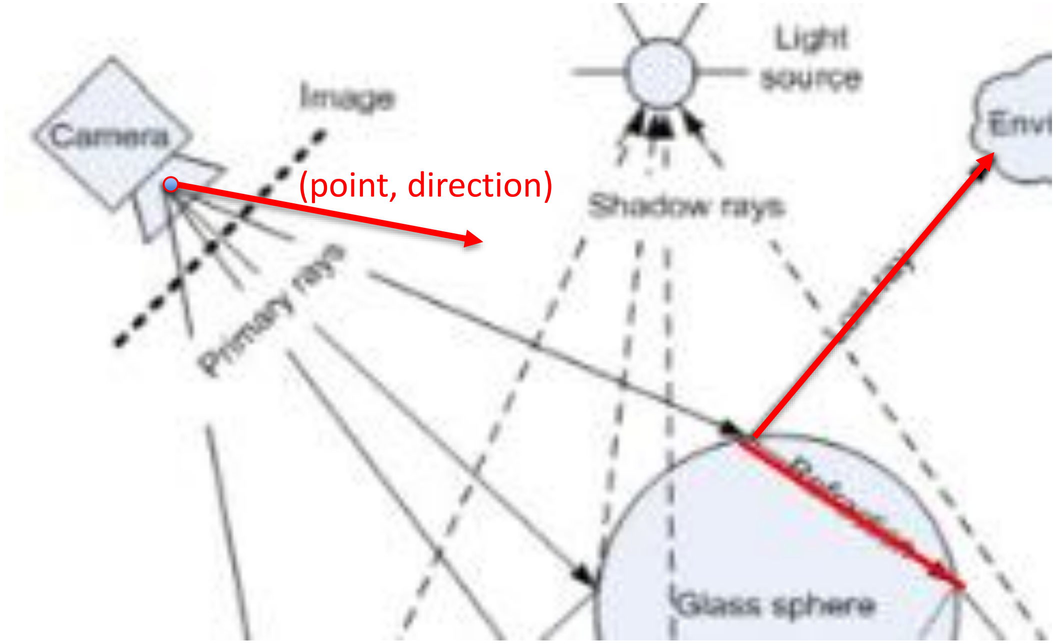
The algorithm relies on a ray-casting procedure that computes the colors seen from a given (point-in-space x, direction ω) couple.

The procedure searches the closest object to point y in the given direction ω, and applies the approximated rendering equation to compute $L(y, \omega)$.



Ray tracing

The algorithm starts considering each point on the projection plane (each pixel of the generated image), in the direction of the projection ray, and applies the ray-casting procedure to it.



Ray tracing

For considering the reflection and refraction part of each pixel, the procedure is called recursively with the computed points and directions.

$$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L\left(l,\overrightarrow{lx}\right) f_{r'}\left(x,\overrightarrow{lx},\omega_r\right) V(x,l) +$$

$$L(r,\overrightarrow{rx}) f_{r'}(x,\overrightarrow{rx},\omega_r) V(x,r) +$$

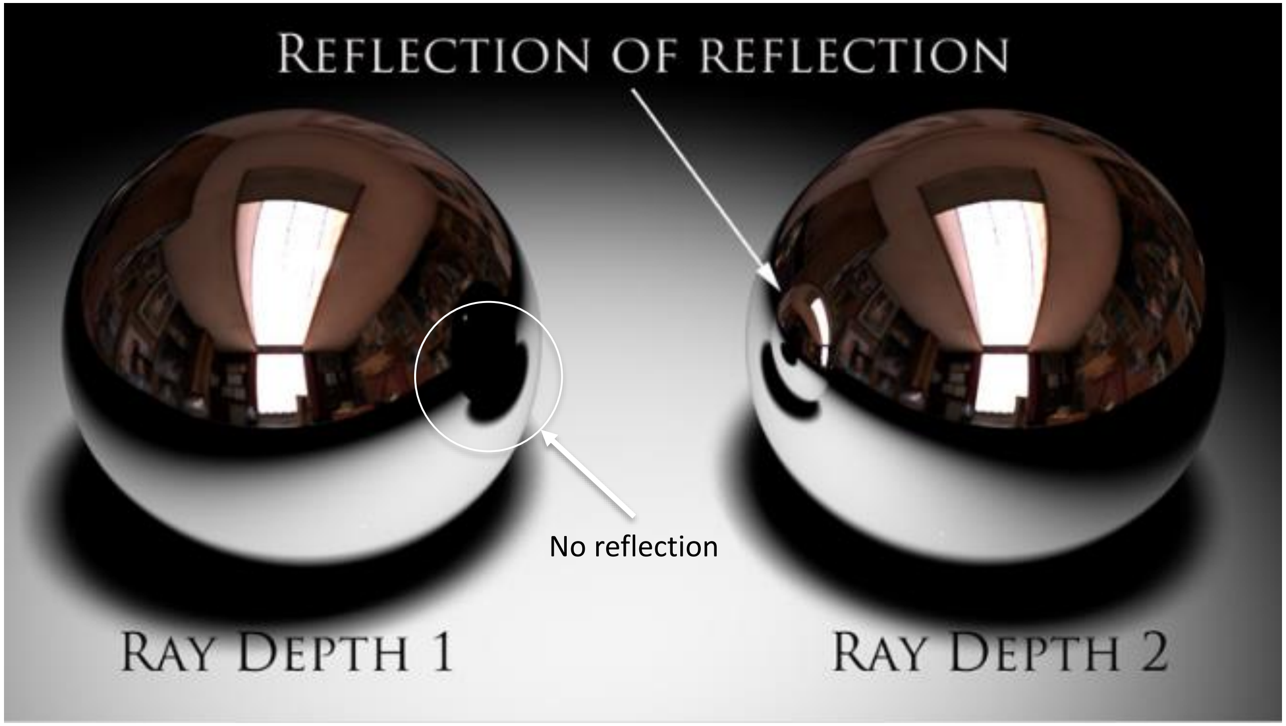
$$L\left(t,\overrightarrow{tx'}\right) f_{t'}\left(x',\overrightarrow{tx'},\omega_r\right) V(x,t)$$

Recursion to compute reflection:
r is the point seen in the reflection.

Recursion to compute refraction:
t is the point seen through the surface by the refraction.
x' is the point on the other side refracted from *x*

Ray tracing

The recursion is repeated up to a given number of bounces, called the *ray depth*.



Ray tracing

Light sources are still separated from the objects, and the visibility for light sources is also considered. This time however ray tracing is used to determine if a light is visible or not.

$$L(x, \omega_r) = L_e(x, \omega_r) + \int L(y, \overrightarrow{yx}) f_r(x, \overrightarrow{yx}, \omega_r) G(x, y) V(x, y) dy$$

$$L(x, \omega_r) = L_e(x, \omega_r) + \boxed{\sum_l L(l, \overrightarrow{lx}) f_{r'}(x, \overrightarrow{lx}, \omega_r) V(x, l)} +$$

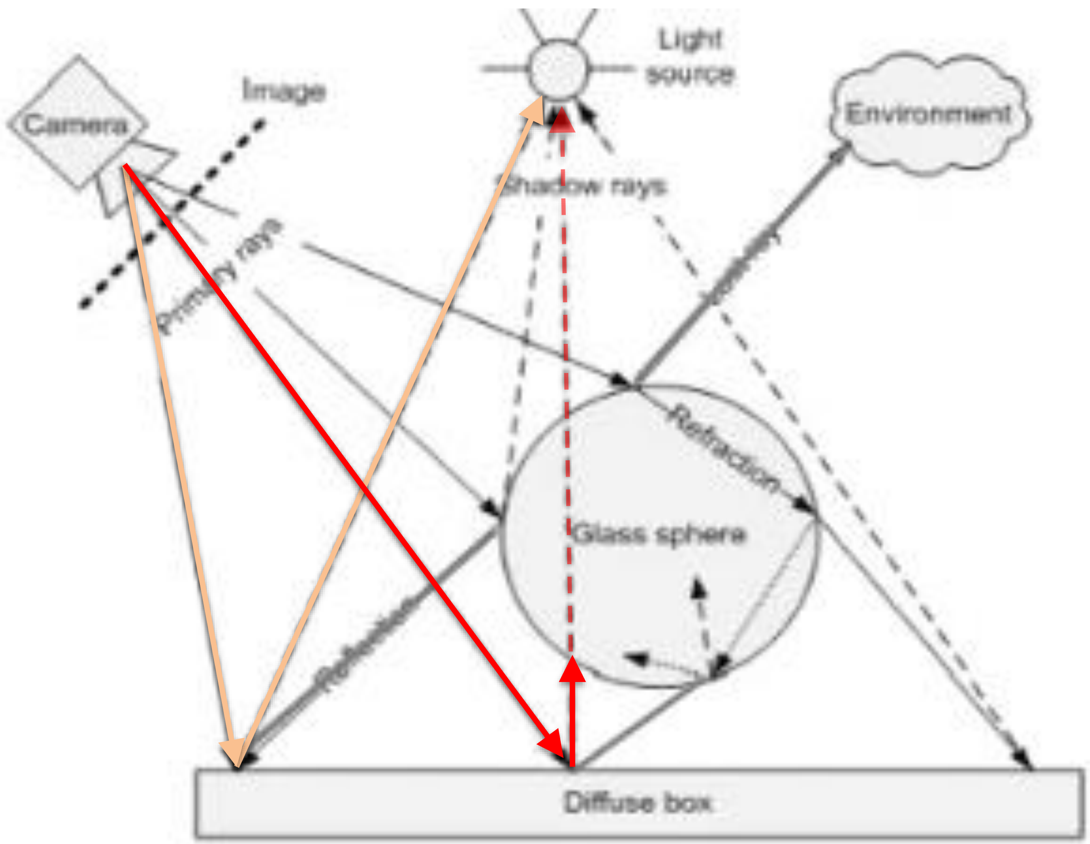
Reflection $L(r, \overrightarrow{rx}) f_{r'}(x, \overrightarrow{rx}, \omega_r) V(x, r) +$

Refraction $L\left(t, \overrightarrow{tx'}\right) f_{t'}\left(x', \overrightarrow{tx'}, \omega_r\right) V(x, t)$

Ray tracing

If the reflect ray does not encounter any other object in the direction of the light, then the source is considered.

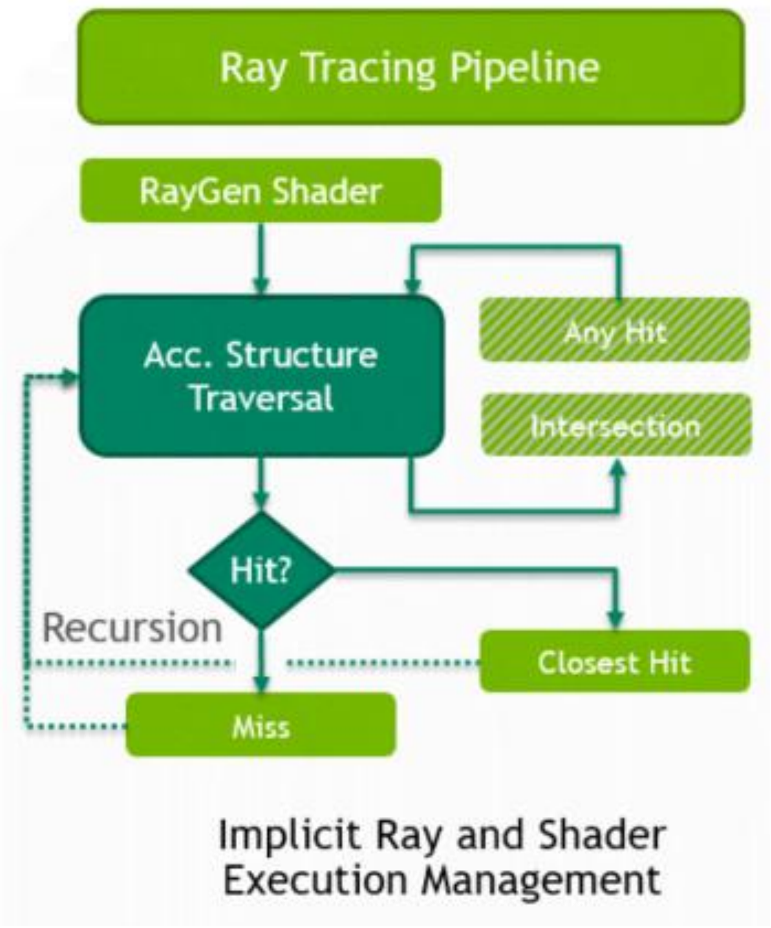
Otherwise, if it hits another object, the light is considered to be covered, and excluded from the rendering equation.



The Ray-tracing pipeline

Vulkan and DirectX, have both a specific pipeline to support ray tracing in real time (provided that suitable GPUs are available).

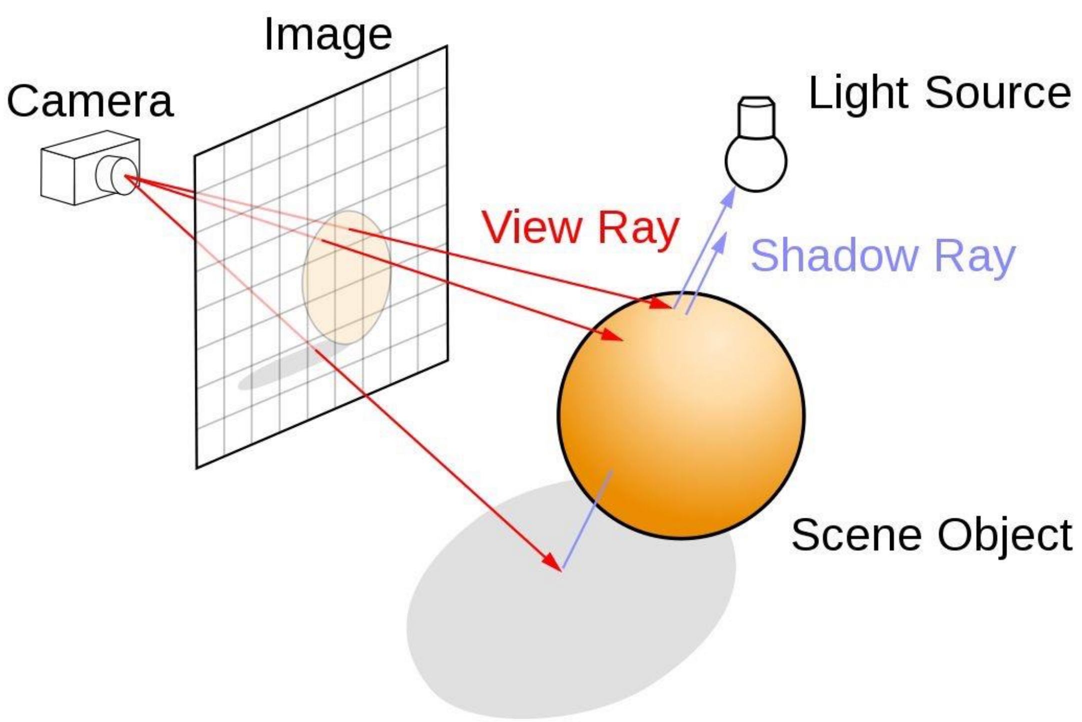
In Vulkan this pipeline is called *"the Ray Tracing pipeline"*.



The Ray-tracing pipeline

The *ray-tracing pipeline* creates images starting from the pixels on screen, and it is not driven by triangles and vertices as the graphic one.

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



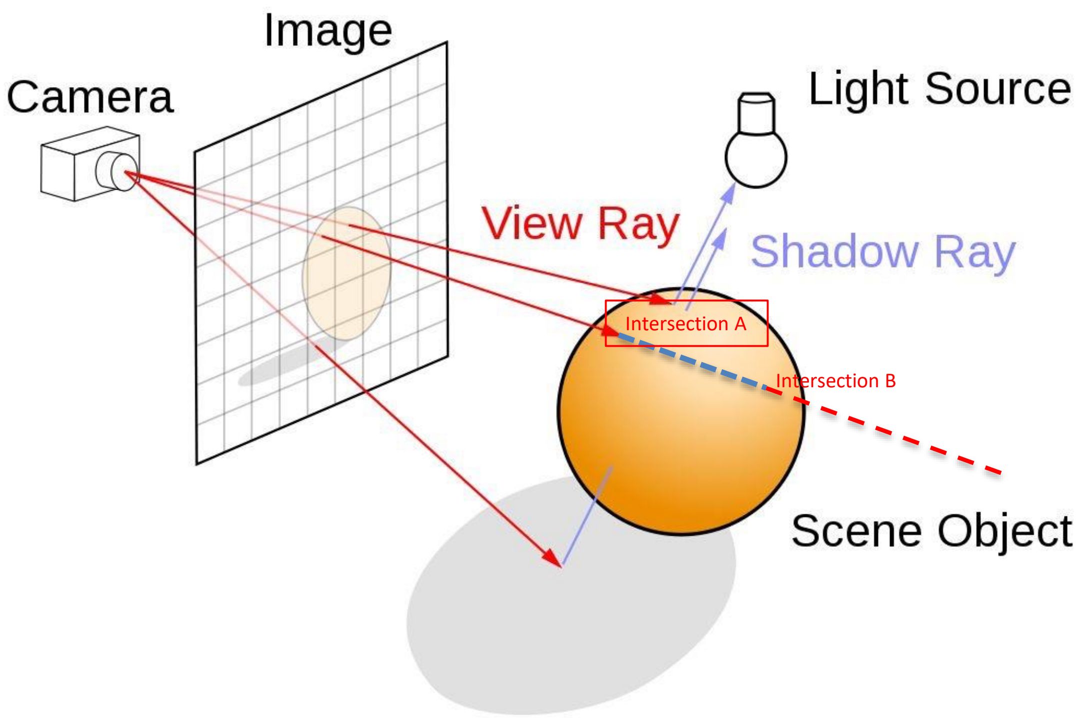
The Ray-tracing pipeline

Only the intersection closer to the viewer is considered.

In order to compute its color, extra rays are traced to accurately reproduce reflections and refraction (transparencies).

For the ray-tracing rendering technique just presented, these rays are generally:

- The perfect reflection ray
- The refraction ray
- The rays connecting the point to the light sources

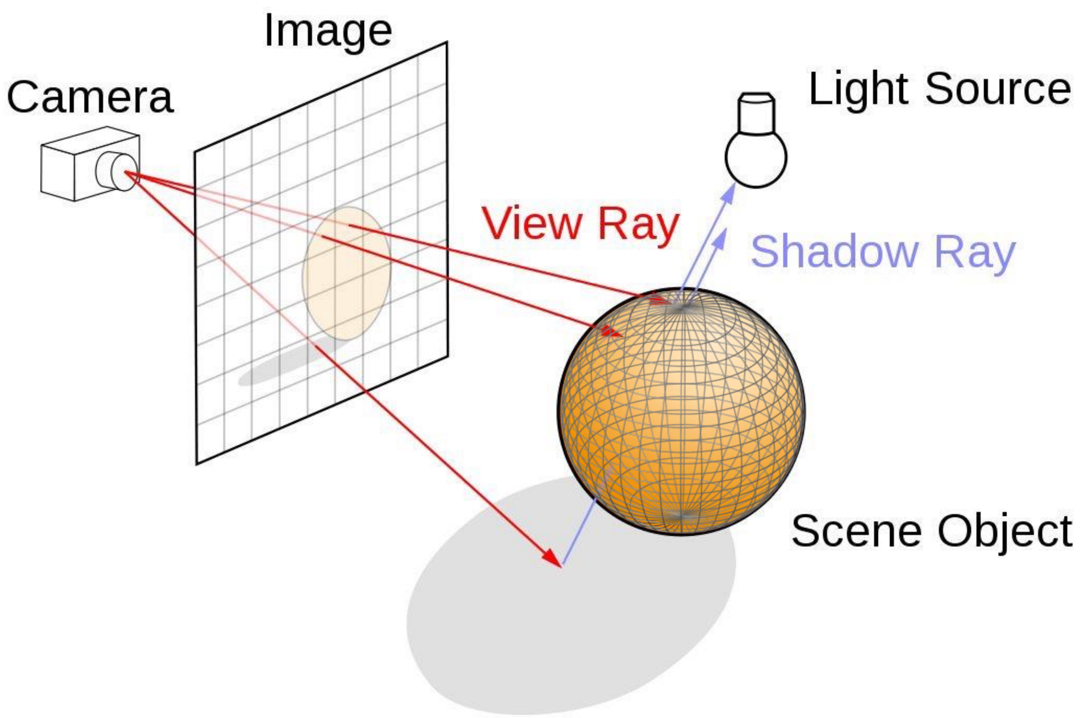


The Ray-tracing pipeline

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

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

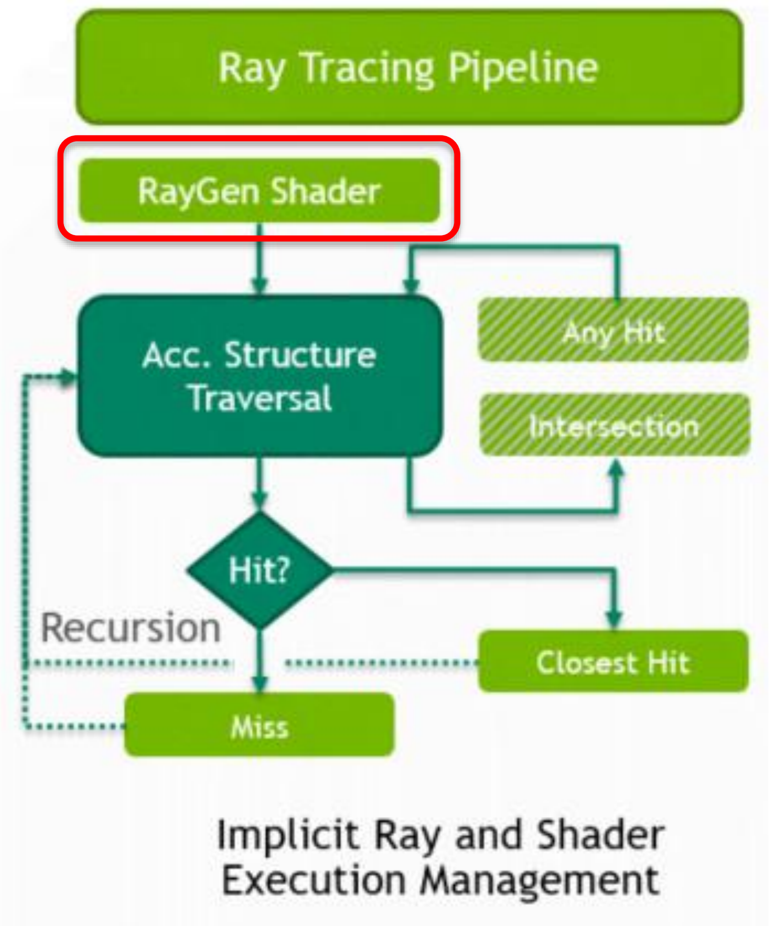
For time constraints we will not be able to present them in depth.



The Ray-tracing pipeline

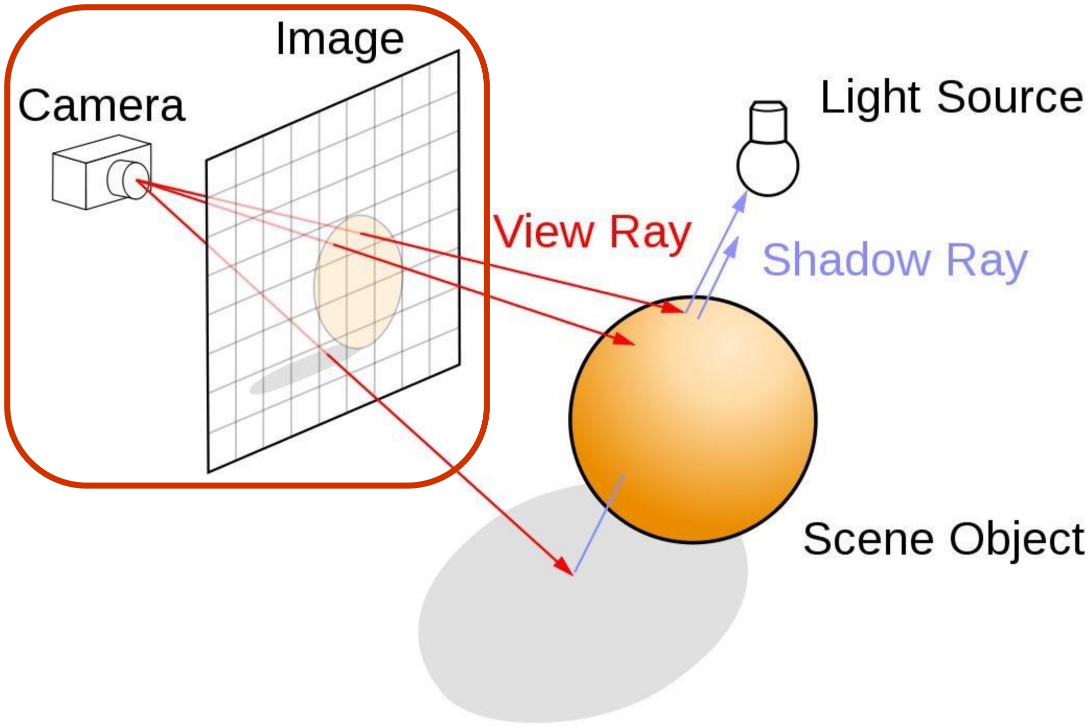
The ray-tracing pipeline requires five shaders to compute and handle all the triangle-ray intersections.

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.



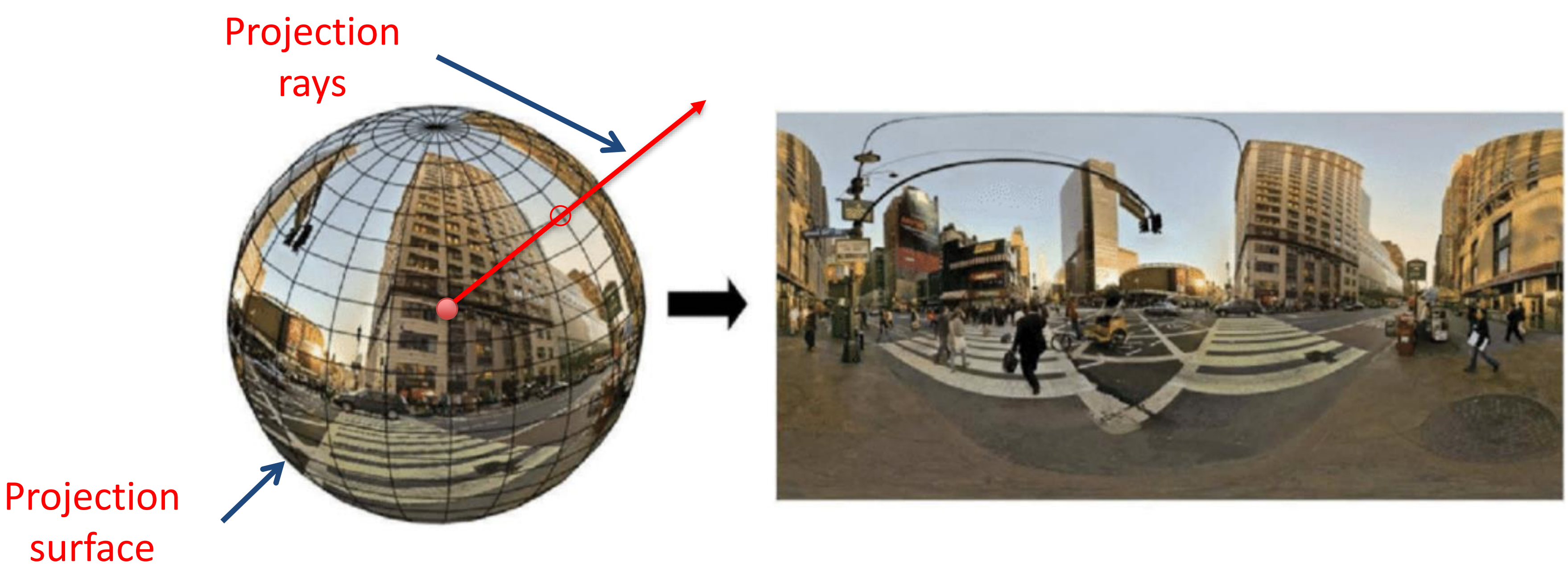
The Ray-tracing pipeline

It generally just considers each pixel on screen, and cast a ray according to the direction that starts from the camera.



The Ray-tracing pipeline

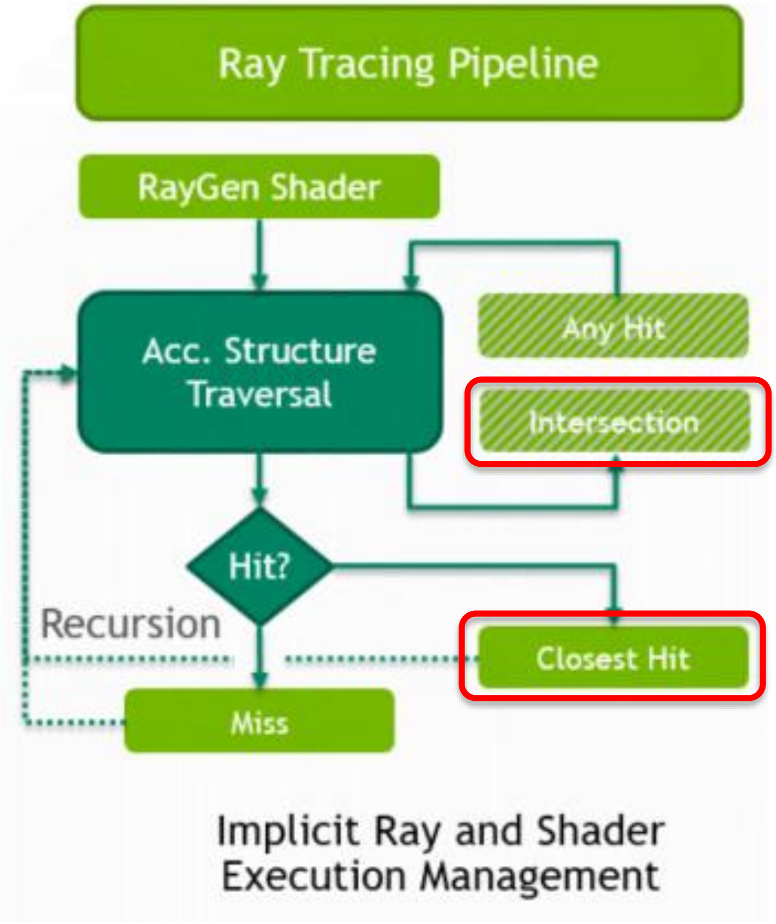
However, user is not constrained to consider a planar projection plane with equally-spaced pixels: cylindrical and spherical projections can be easily implemented by minor changes to the *RayGan* procedure.



The Ray-tracing pipeline

The *Intersection* shader allows to implement custom *ray-triangle intersection* procedures.

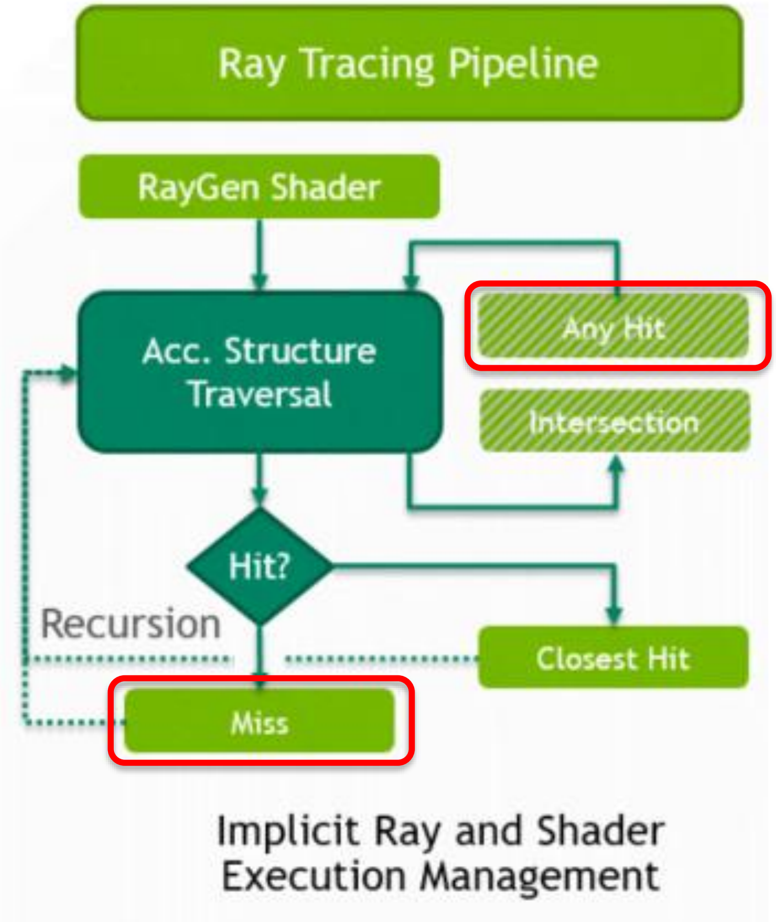
The *Closest Hit* shader is instead called on the point that is closer to the viewer. Its purpose is computing the color of the point, by approximating the rendering equation in a way similar to a fragment shader in the graphics pipeline, and for doing this it can recursively cast other rays.



The Ray-tracing pipeline

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

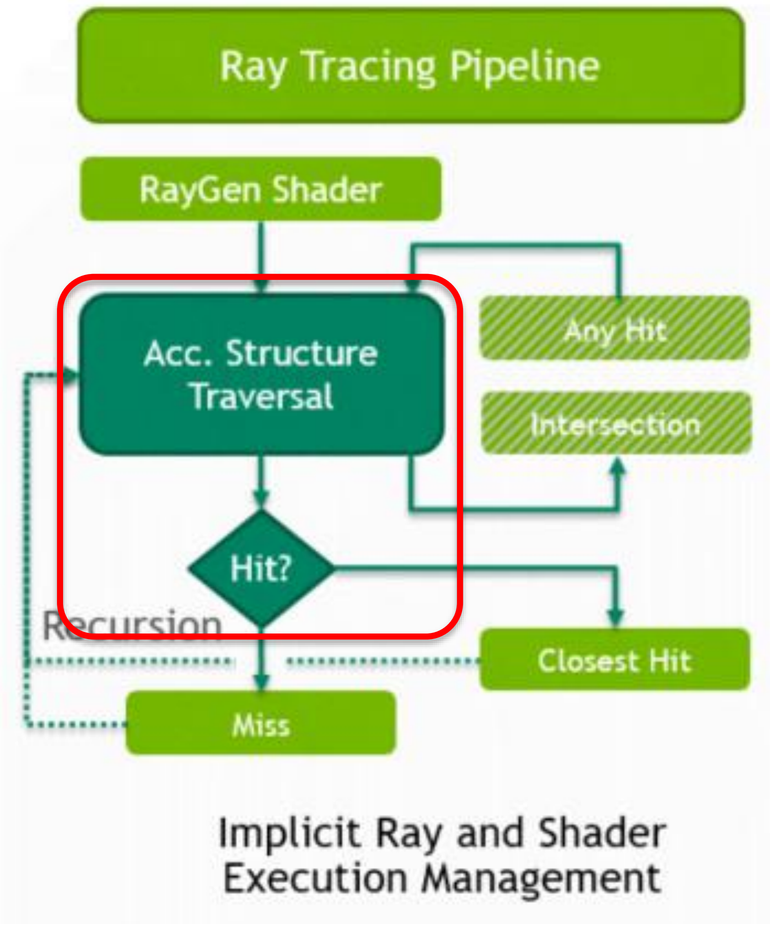
The *Miss* shader is instead called if the ray does not hit any object: generally it is used to draw some sort of background that should appear behind everything else.



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 real time ray tracing.



Ray tracing implementation

The pseudo-code of a ray tracing rendering algorithm is the following:

```
for each pixel x on screen do
  Compute color casting a ray from x according the projection
end
```

The ray-casting procedure, is the hearth of the technique:

```
Determine the point q of the closest object w.r.t. the ray
Set the pixel color C = 0
for each light l in the scene do
  if light l is not occluded (ray-casting) then
    Set C = C + contribution of light l to q
  end if
end
Set C = C + reflection contribution (recursion)
Set C = C + refraction contribution (recursion)
```

Ray tracing implementation

In a Ray Tracing pipeline, the *RayGen* Shader, and the *Closest Hit Shader* perform the tasks highlighted in the previous slide.

Ray Generation Shader

```
for each pixel x on screen do
   Compute color casting a ray from x according the projection
end
```

Intersection Shader → Hit? → N → *Miss Shader*

Hit? → Y → *Any hit Shader* → *Closest hit Shader*

```
Determine the point q of the closest object w.r.t. the ray
Set the pixel color C = 0
for each light l in the scene do
  if light l is not occluded (ray-casting) then
    Set C = C + contribution of light l to q
  end if
end
Set C = C + reflection contribution (recursion)
Set C = C + refraction contribution (recursion)
```

Ray tracing - discussion

The number of traced rays potentially doubles at every step: this can significantly increase the rendering times. Moreover, it requires computation of the closest intersection using the *acceleration structure* instead of the Z-buffer.

Ray tracing allows including mirror reflection and transparency with refraction: this can be used to realistically reproduce glass, fluid, shiny metals and many other objects.

However, ray tracing is not able to simulate indirect lighting or consider glossy reflections, limiting the level of achievable realism.

Radiosity

Radiosity proposes a different simplification of the rendering equation. In particular it considers only materials that have a constant BRDF.

$$f_r(x, \omega_i, \omega_r) = \rho_x$$

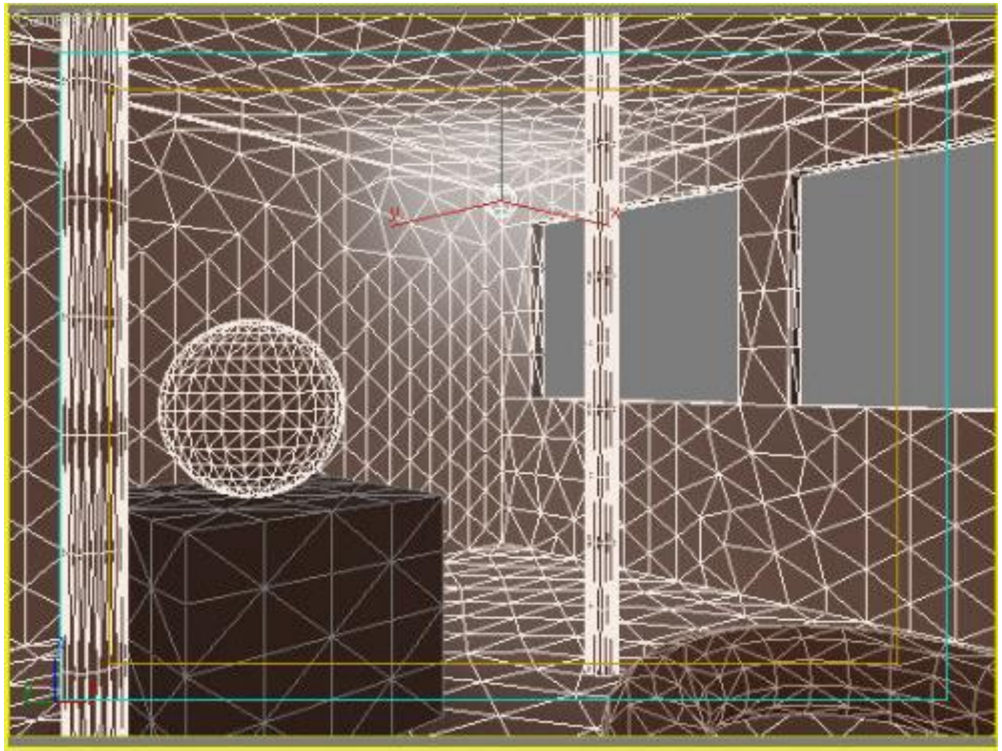
The unknowns of the rendering equations are thus reduced to one variable per point of the objects since in this way reflection does not depend on the direction from which it is seen. This unknown is called the *radiosity* of the object (that is the output counterparts of the irradiance).

$$L(x, \omega_r) = L_e(x, \omega_r) + \int L(y, \overrightarrow{yx}) f_r(x, \overrightarrow{yx}, \omega_r) G(x, y) V(x, y) dy$$

$$\Downarrow$$

$$L(x) = L_e(x) + \rho_x \int L(y) G(x, y) V(x, y) dy$$

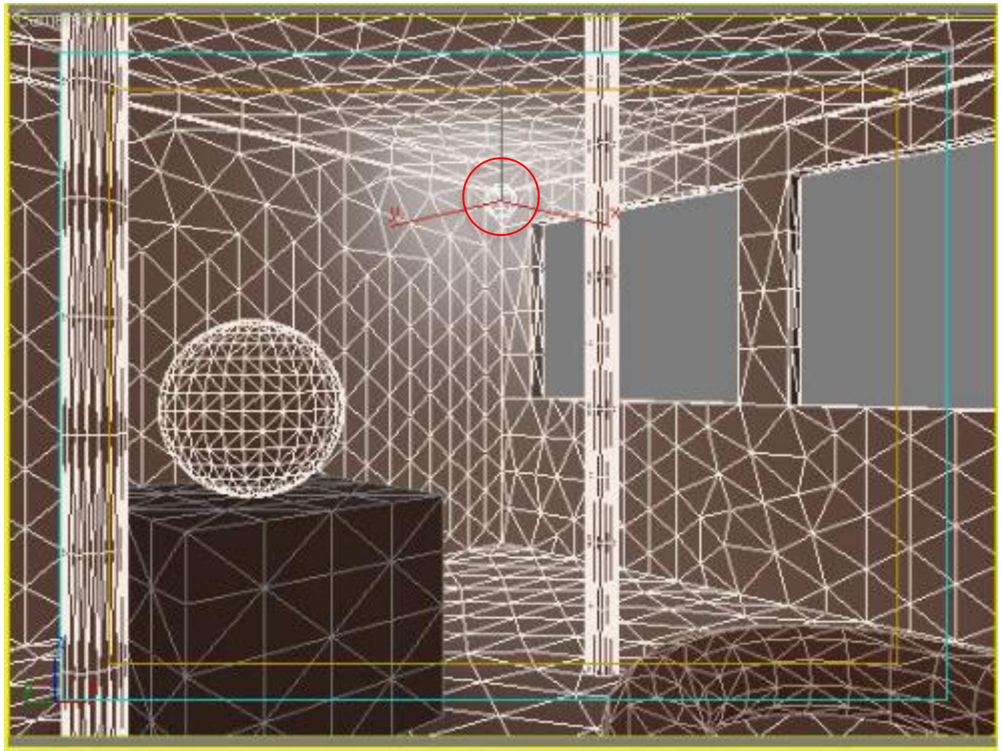
The surfaces of the objects are then split into patches, with one unknown per patch.



Radiosity

Light sources are implemented as patches that emits radiosity.

$$L(x) = L_e(x) + \rho_x \int L(y)G(x,y)V(x,y)dy$$



Radiosity

The rendering equation becomes a (large) system of linear equations that can be solved with an iterative technique.

$$L(x) = L_e(x) + \rho_x \int L(y) G(x,y) V(x,y) dy$$

$$\Downarrow$$

$$L(x_i) = L_e(x_i) + \rho_{x_i} \sum_{y_j} L(y_j) G(x_i, y_j) V(x_i, y_j)$$

Radiosity

In matrix notation, vector L has one element (per color frequency) per patch, and represents its *radiosity*.

Matrix *R* includes the visibility, the constant BRDF, and the geometric relations between any two patches of scene.

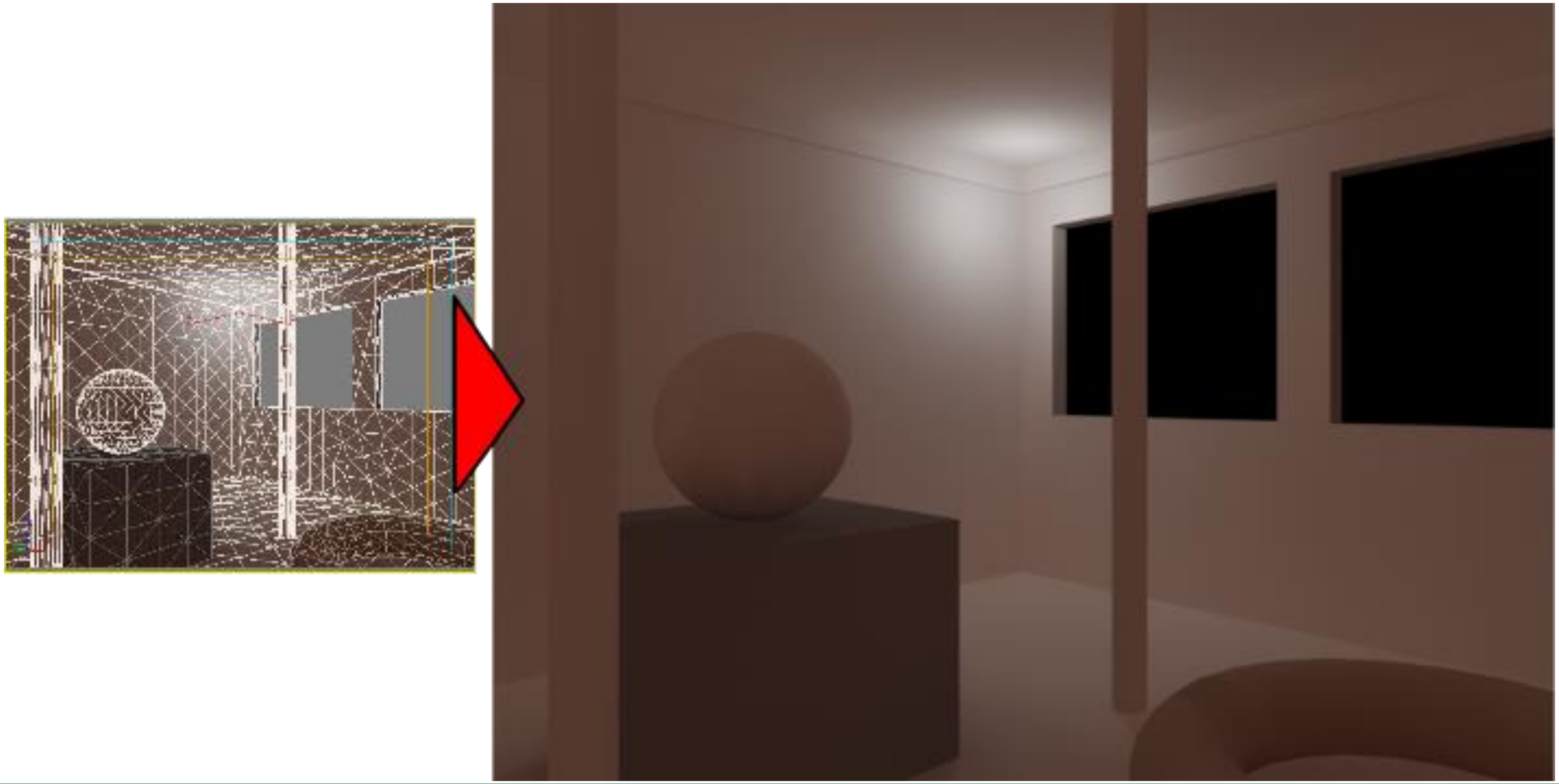
$$L(x_i) = L_e(x_i) + \rho_{x_i} \sum_{y_j} L(y_j) G(x_i, y_j) V(x_i, y_j)$$

$$L(x_i) = L_e(x_i) + \sum_{y_j} L(y_j) R(x_i, y_j) \qquad R(x_i, y_j) = \rho_{x_i} G(x_i, y_j) V(x_i, y_j)$$

$$L = L_e + R \cdot L$$

Radiosity

The solutions of the equations are then interpolated to produce a view of the scene.



Radiosity

The pseudo-code of a radiosity rendering algorithm is the following:

01 Discretize the scene, and compute matrix R
02 Compute the solution of equation $L = L_e + R \cdot L$
03 Render the scene using either scan-line or raytracing
04   Interpolate L to obtain the color for each pixel

Indeed the most time consuming steps are steps 01 and 02.

However, once they have been computed, the scene can very quickly be rendered from any point of view.

Radiosity

In most of the cases, the solution of the equation can be computed with a fixed-point iteration, starting from L = 0, and refining its value at every iteration.

```
Lold = Lnew = 0
Repeat
    Lold = Lnew
    Lnew = L_e + R · Lold
Until |Lnew - Lold| > threshold
```

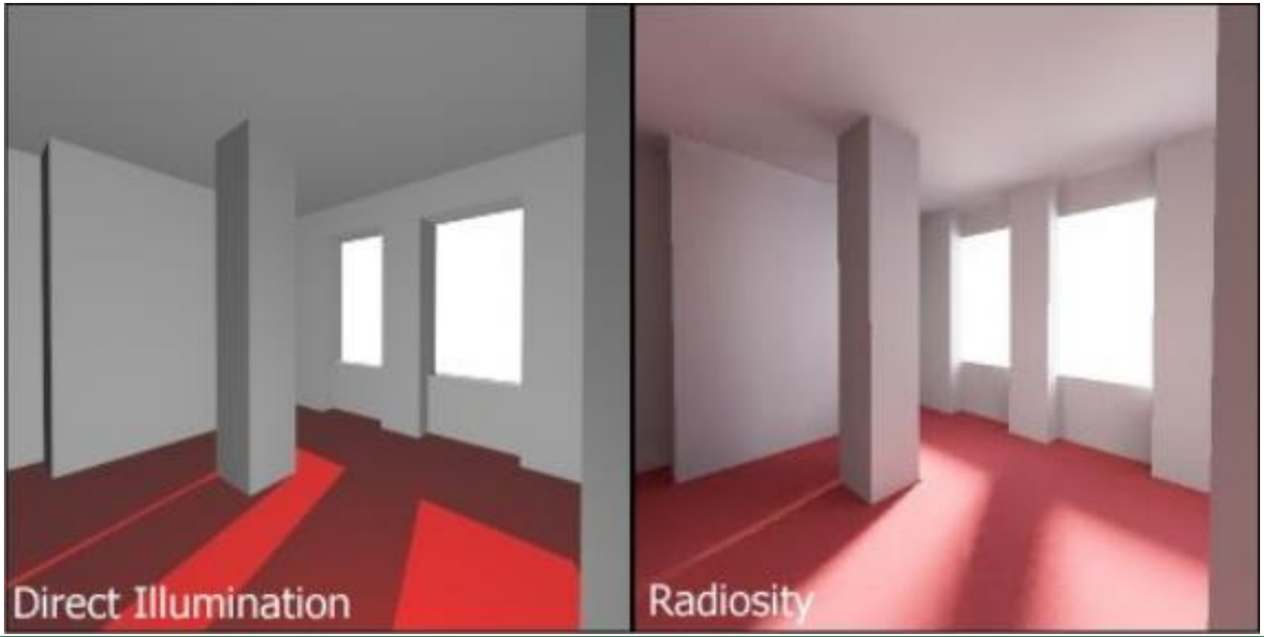
Even if the number of patches is big, the loop converges in a moderate number of iterations.

Radiosity

Radiosity is able to capture indirect illumination effects.

Shadows however are usually very poorly approximated due to the size of the patches.

Mirror reflections and refractions cannot be considered directly, since they greatly depend on the direction from which an object is seen.



Radiosity is usually an off-line technique, and almost never implemented in real time. For this reason Vulkan and other low-level graphics engine have no specific pipelines to support it.

Montecarlo techniques

Photorealistic results can only be achieved by approximating the solution of the complete rendering equation.

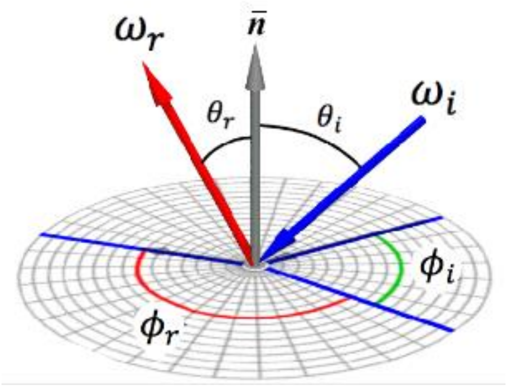
Due to its complexity, *Montecarlo* techniques are usually employed: the integral is computed by averaging several random points and directions chosen from the equations.

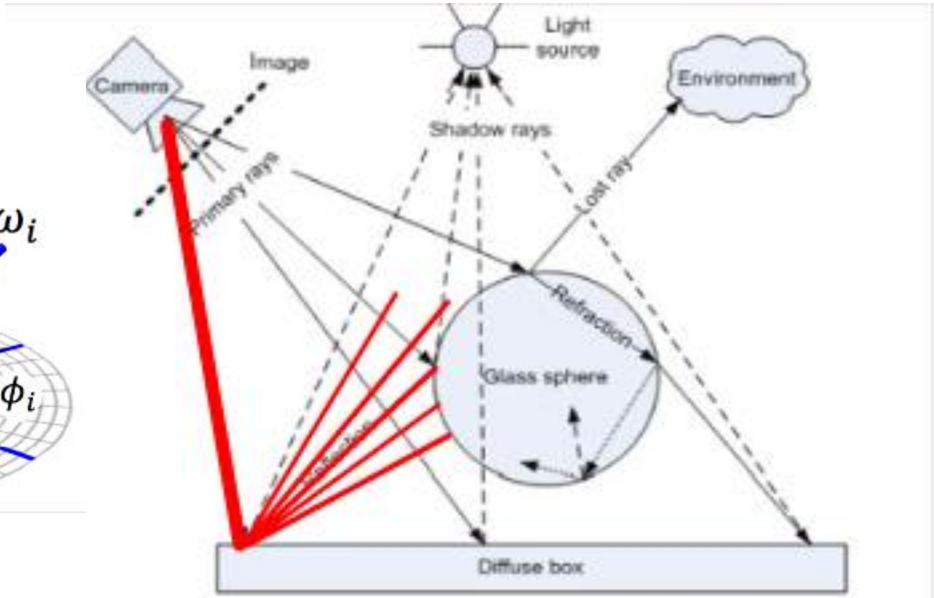
Many alternative approaches are possible: each advanced rendering engine exploits one of them.

Montecarlo techniques

Many techniques extend ray tracing: instead of sending a single ray in the mirror direction, a sampling of the *most probable* output directions is considered *(importance sampling)*.

A ray is thus traced for each selected direction, and the radiance is computed using the BRDF of the considered material.

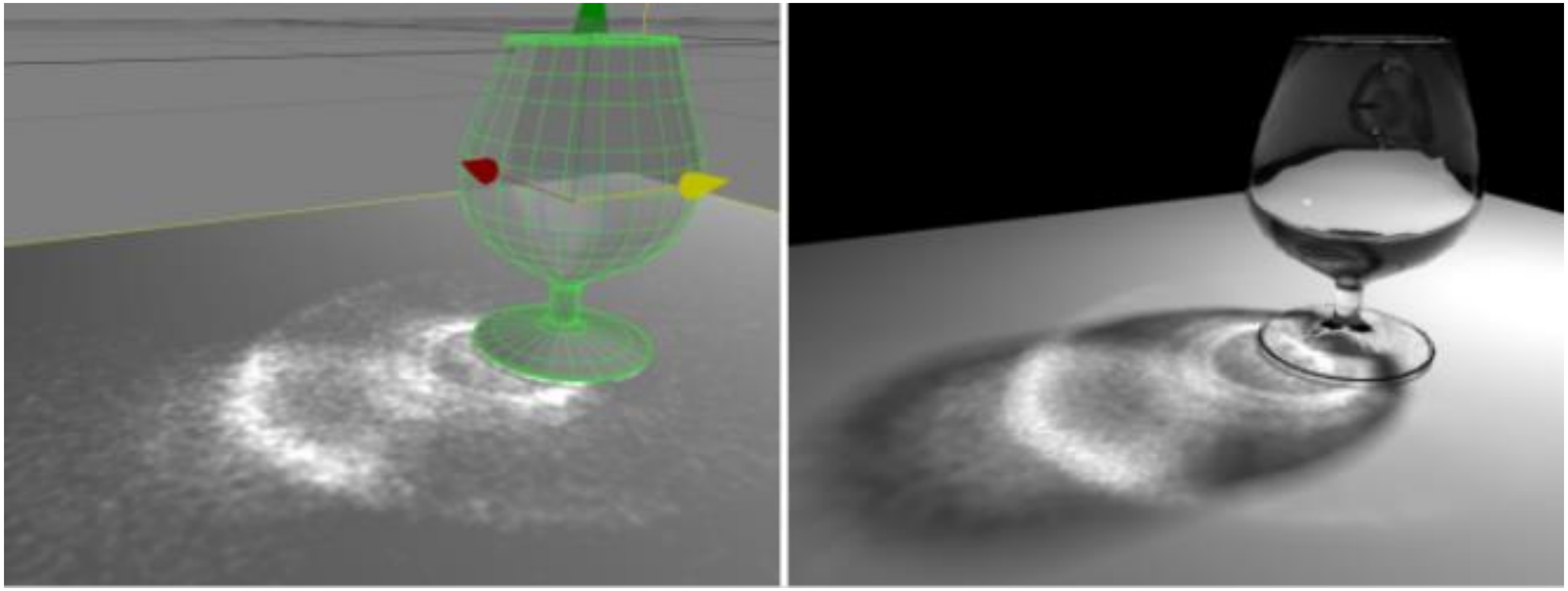




Montecarlo techniques (based on a limited number of rays) are sometimes implemented in real time, using the ray-tracing rendering pipeline.

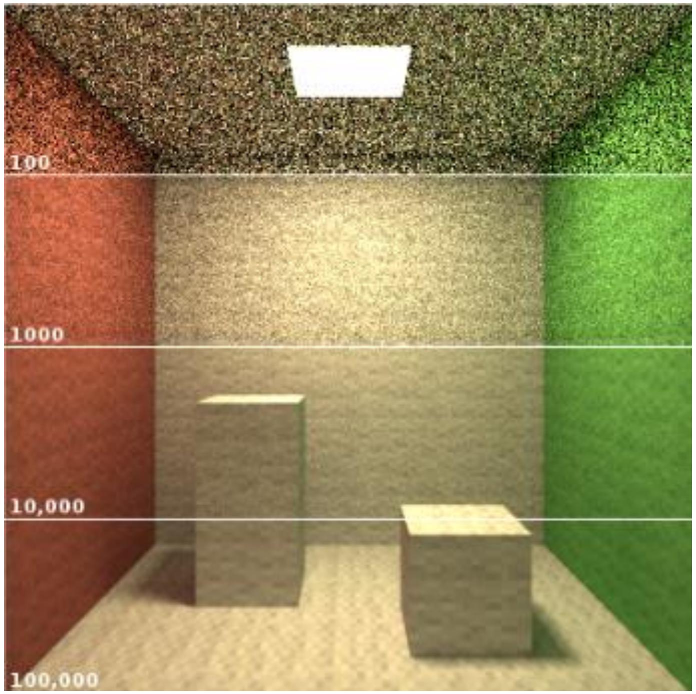
Montecarlo techniques

Photon mapping instead emulates the movements of photons in the scene, considering bounces, focalizations and other advanced phenomenon such as caustics.



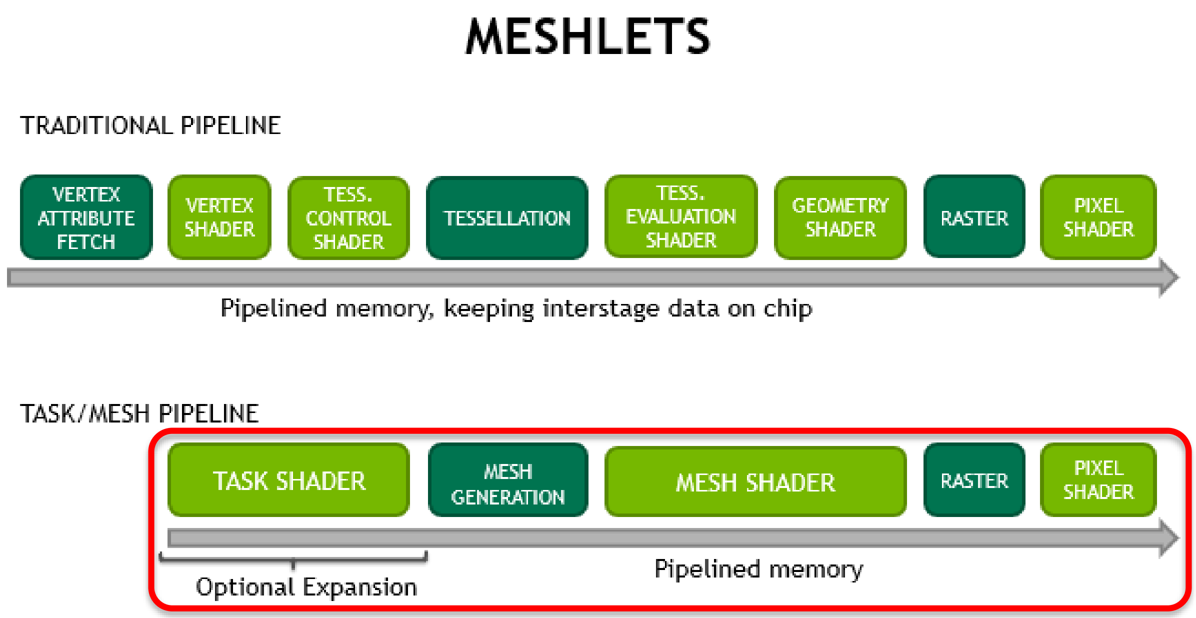
Montecarlo techniques

Due to the randomness that drives the techniques, *Montecarlo* based rendering algorithms tend to produce noisy images, whose effect can only be reduced by increasing the number of rays (and consequently, the rendering time).



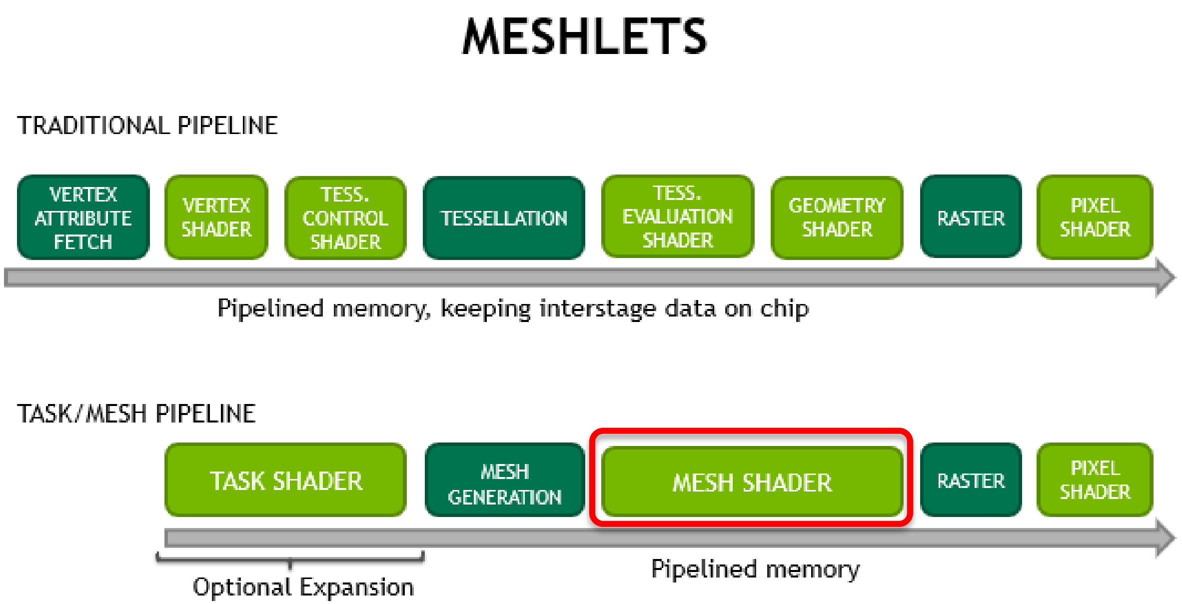
The Mesh Shader pipeline

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 the shaders.



The Mesh Shader pipeline

Mesh shaders computes indexed triangle lists, returned as a set of vertices and groups of three indices for each triangle. Vertices are computed in normalized screen coordinates to simplify rasterization.

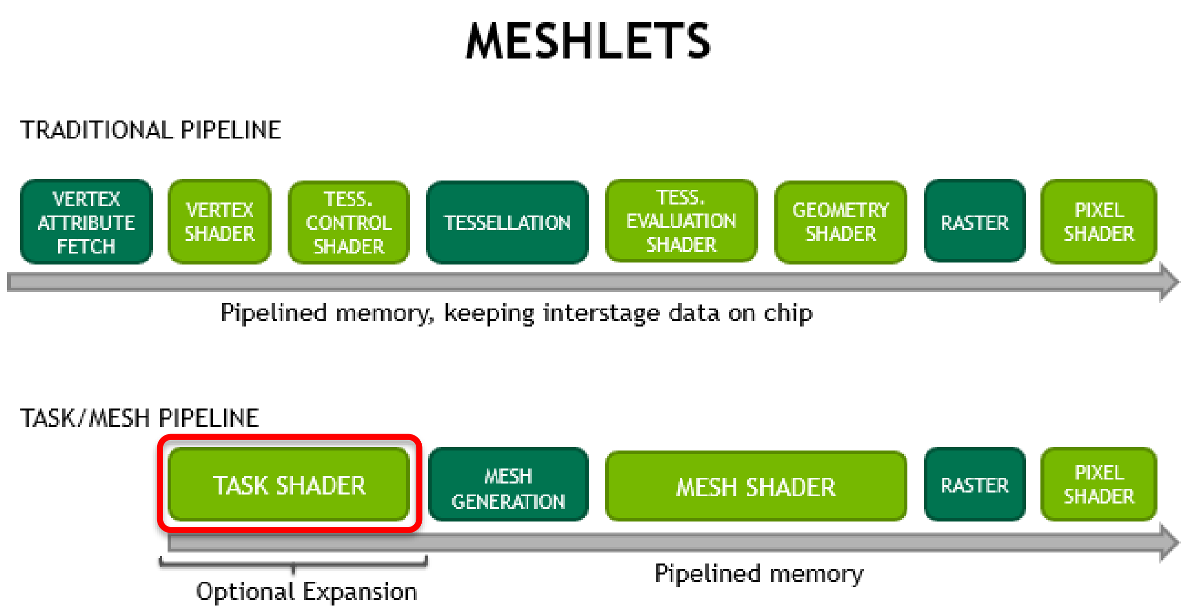


The Mesh Shader pipeline

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

The Mesh Shader pipeline

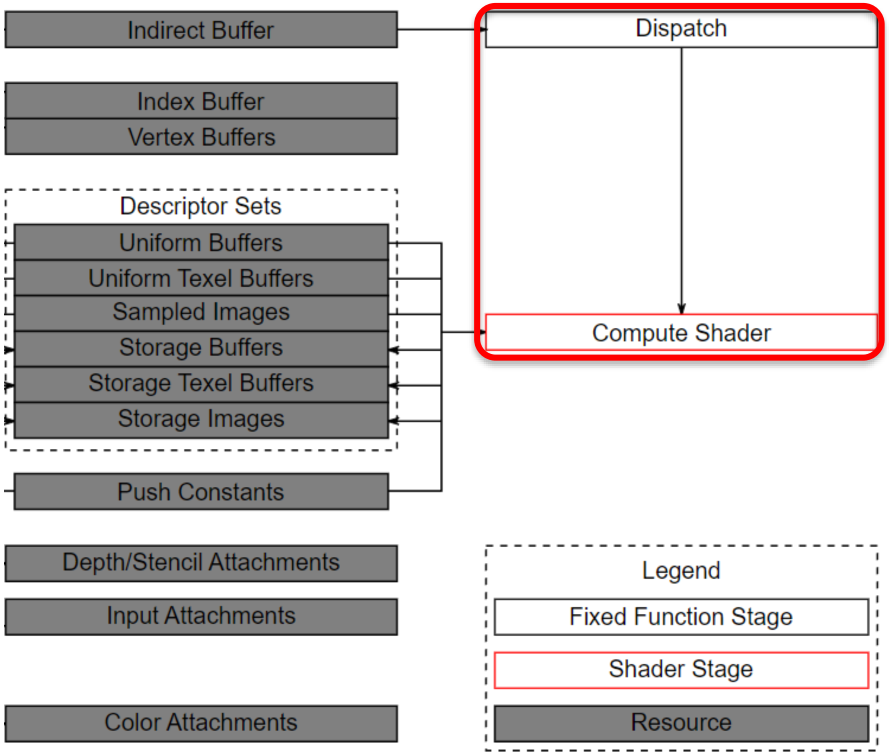
The purpose of the optional *Task* shader is to subdivide a larger mesh into smaller Meshlets, and control the corresponding *Mesh* shader for generating all the required patches.



The Compute pipeline

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

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

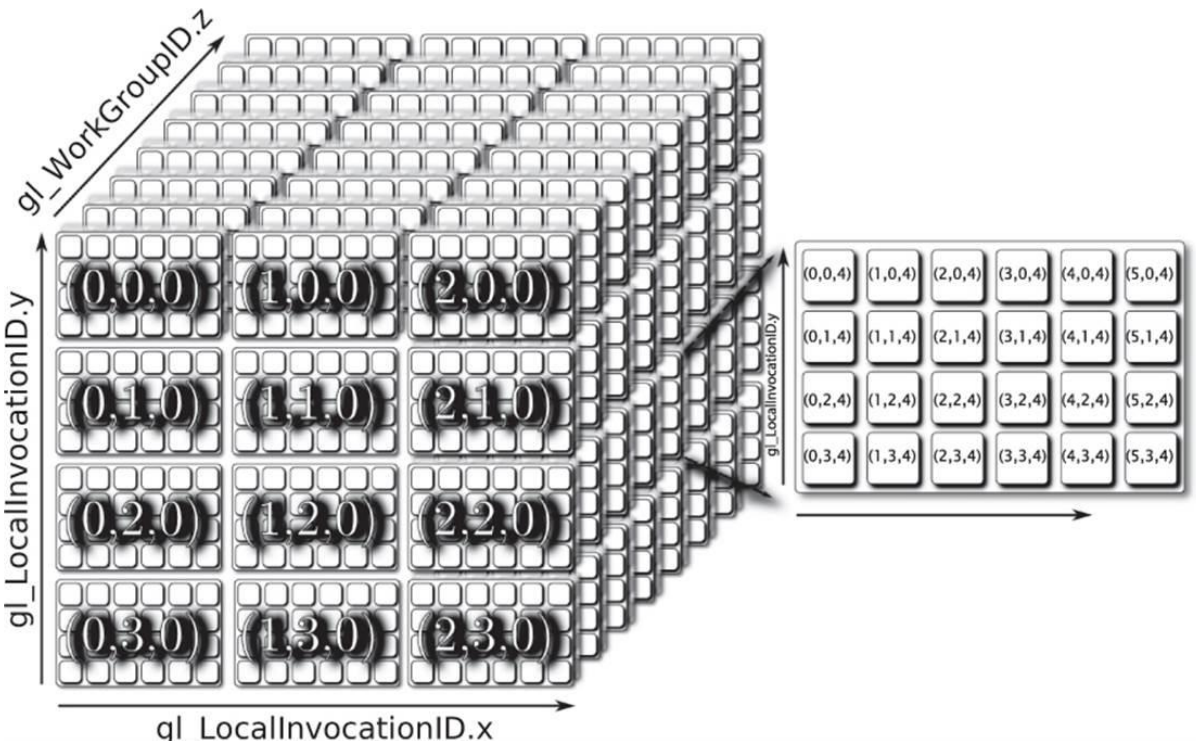


The Compute pipeline

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.

Please follow the GPU course, if you are more interested in the topic!



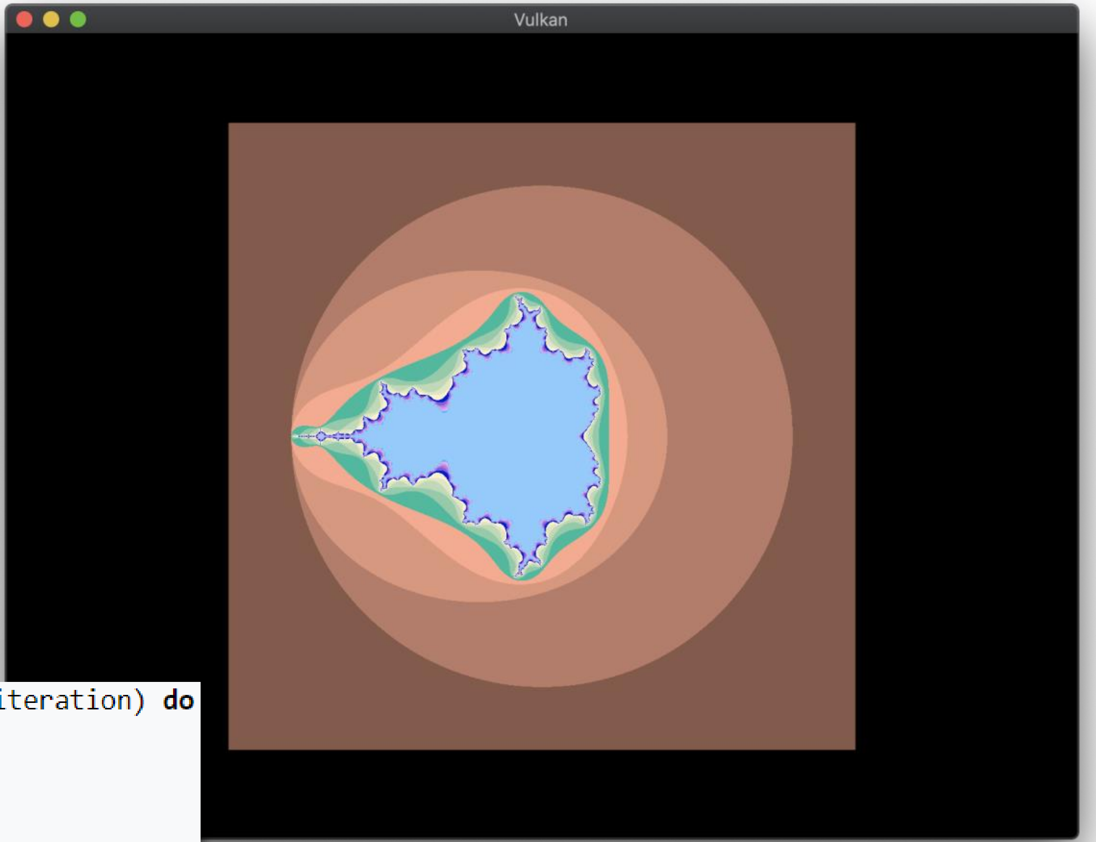
Most of the BRDF functions, indirect light approximation and light emission models will be implemented using Shaders written with the GLSL language.

We have already seen how we can compile a shader from its GLSL source code to its SPIR-V intermediate representation required by Vulkan.

GLSL is a C-like language, and it is very similar to C, C++, C#, JAVA, JavaScript, PHP and many others.

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.




```
while (x*x + y*y ≤ 2*2 AND iteration < max_iteration) do
    xtemp := x*x - y*y + x0
    y := 2*x*y + y0
    x := xtemp
    iteration := iteration + 1

color := palette[iteration]
```

Program structure

Shaders follow the classical convention, having global variables and functions in the main scope of the file.

The entry point of the shader can be user defined in the code that calls it: however it is generally the function `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

Vertex shader

Vulkan require that at least GLSL version is supported.

For this reason, a shader source code should start with the `#version` directive, asking at least for version 4.5

```
#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

Vertex shader

Comments in the code follows the classical *C* notations:

// for comments to the end of the line

/* begins a comment block

*/ ends a comment block

```
#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

Vertex shader

Blocks are denoted using { }

```
#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 their name follows the same C convention (case sensitive, allowing letters, numbers and underscore, but cannot start with a number).

As in C, variables are local to the block they are defined into.

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	->	dmat4x3

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

Vertex shader

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

```
#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 world coordinates, colors or texture coordinates, 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 to shuffle a vector:

```
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 indices starting from zero in [ ][ ] brackets.

Matrix columns can also be accessed as vectors:

```
vec4 v;
mat4 m;
m[1] = v;          // sets the second column v
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 defined in the *GLM* library we used in our C++ source code, have been named to follow the equivalent GLSL conventions: this simplifies the interactions between the shaders and the application code.

GLSL

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,
mat2x2, mat2x3, mat2x4	2-column 2, 3, or 4
mat3x2, mat3x3, mat3x4	3-column 2, 3, or 4
mat4x2, mat4x3, mat4x4	4-column 2, 3, or 4
dmat2, dmat3, dmat4	2x2, 3x3, float matr
dmat2x2, dmat2x3, dmat2x4	2-col. dou matrix of
dmat3x2, dmat3x3, dmat3x4	3-col. dou matrix of
dmat4x2, dmat4x3, dmat4x4	4-column matrix of

GLM (in C++)

```
glm::mat4 M1, M2, M;
M1 = translate(glm::mat4(1), glm::vec3(1, 2, 3));
M2 = glm::mat4(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16);
M = M1 * M2;
for(int i = 0; i < 4; i++) {
  for(int j = 0; j < 4; j++) {
    std::cout << M1[j][i] << "\t";
  }
  std::cout << "\t";
  for(int j = 0; j < 4; j++) {
    std::cout << M2[j][i] << "\t";
  }
  std::cout << "\n";
}
std::cout << "\n";
for(int i = 0; i < 4; i++) {
  for(int j = 0; j < 4; j++) {
    std::cout << M[j][i] << "\t";
  }
  std::cout << "\n";
}
std::cout << "\n";
```

```
gribaudo@Marcos-MBP other st
1       0       0       1
0       1       0       2
0       0       1       3
0       0       0       1

5       13      21      29
10      22      34      46
15      31      47      63
4       8       12      16

gribaudo@Marcos-MBP other st
```

Vector and matrix literals

Matrix and vector elements 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

Vertex shader

Larger vectors can be composed adding elements to shorter ones.

```
#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);
}
```

Algebraic operations

Vertex shader

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

```
#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

Functions

Fragment shader

Functions definitions and function calls follow the classical C convention:

definition:

return_type name (args, …)

{…} // function body

call:

name(args, …)

```
#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.

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

Fragment shader

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

```
#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

Common C statements for exiting loops can also be used.

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 the GPU behaves differently than on the CPU.

The SIMD architecture of GPU, which always processes many elements at the same time, have some nasty implications:

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

This is why it is always a good idea trying to avoid loops (with variable number of iterations) and conditional statements as much as possible.

Shader-pipeline communication

Vertex shader

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

```
#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*

Vertex shader

`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.

```
#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

Vertex shader

Communication with the fixed part of the pipeline also occurs through some 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.

```
#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 one, which is always required in any applications using the graphic pipeline, a lot more global variables exit, mainly to interface with other shaders types.

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

Please also note that some variables have changed name in Vulkan, even if a complete list of the changes is not easy to find.

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

Vertex shader

Communication between the Shaders and the application occurs using special types of external variables, usually called *Uniforms*.

The most common on are the *Uniform Variables Blocks.*

```
#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

Vertex shader

Each block is similar to a C `typedef struct` element: 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.

```
#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

Vertex shader

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

```
#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;
}
```

Marco Gribaudo

Associate Professor

CONTACTS

Tel. +39 02 2399 3568

marco.gribaudo@polimi.it

https://www.deib.polimi.it/eng/home-page

(Remember to use the phone, since mails might require a lot of time to be answered. Microsoft Teams messages might also be faster than regular mails)