Ambient, Emission and Vertex Attributes

Ambient lighting

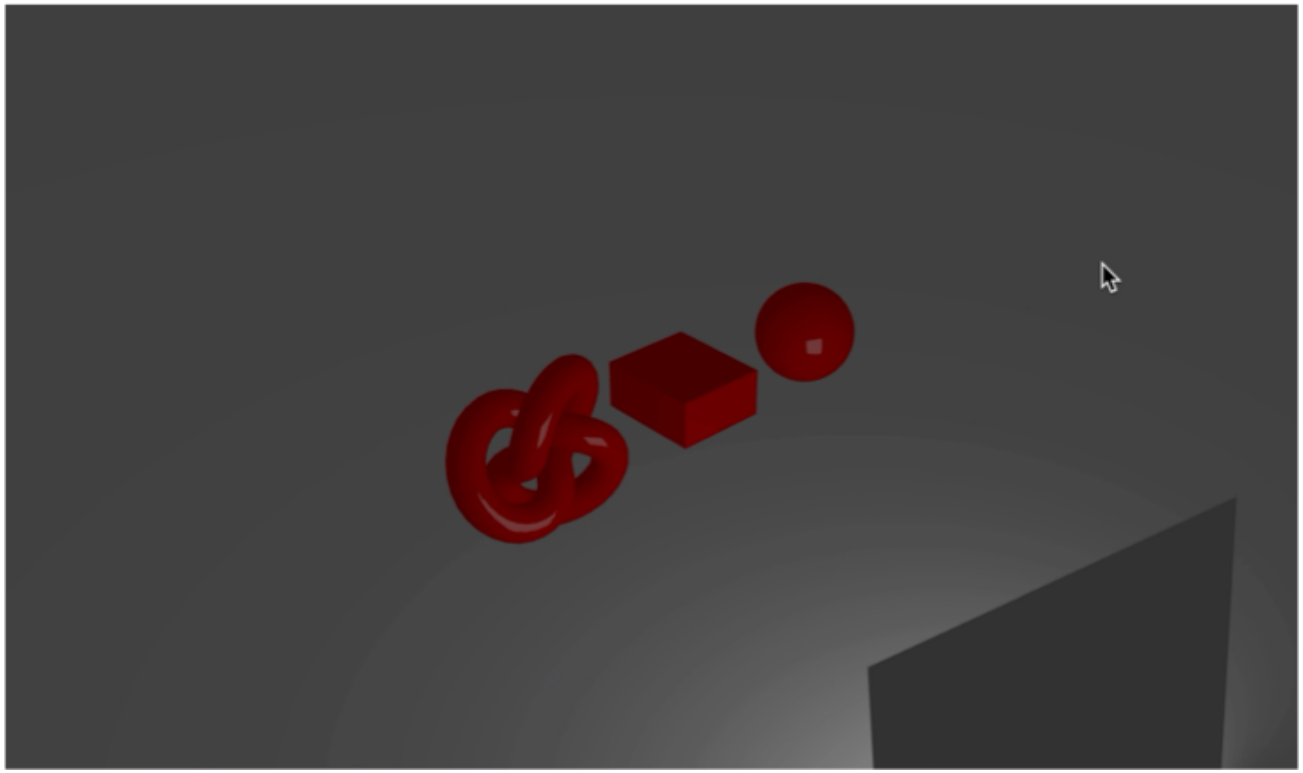
Considering only direct light sources (such as directional, point or spotlight) can produce very dark images.

Realistic rendering techniques, tries to consider also *indirect lighting*: illumination caused by lights that bounces from other objects.

Ambient lighting is the simplest approximation for indirect illumination.

Ambient lighting

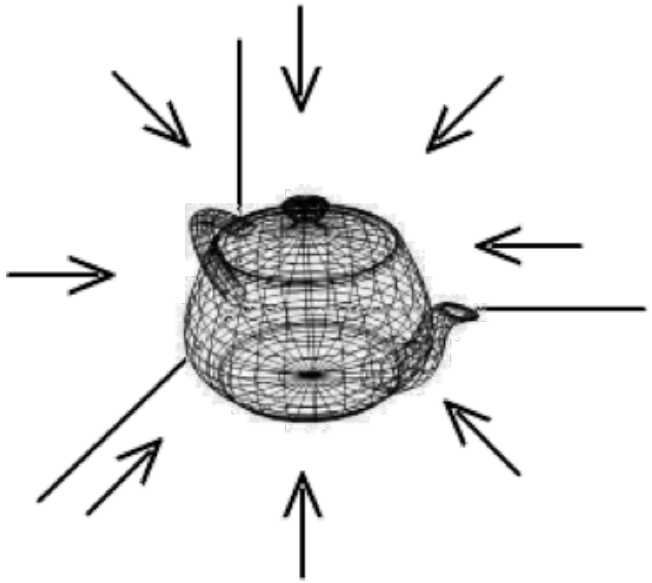
The *ambient light emission factor* (constant for the entire scene) accounts for the light reflected by all the objects in all the directions.



Ambient lighting

The ambient light emission is specified by a constant RGB color value $\boldsymbol{l}_A$.

$$\boldsymbol{l}_A = (l_{AR}, l_{AG}, l_{AB})$$



Ambient lighting

The BRDF of the object, is then extended by adding another component $f_A(x, \omega_r)$ that specifically considers ambient lighting.

Such component does not depend on the light direction.

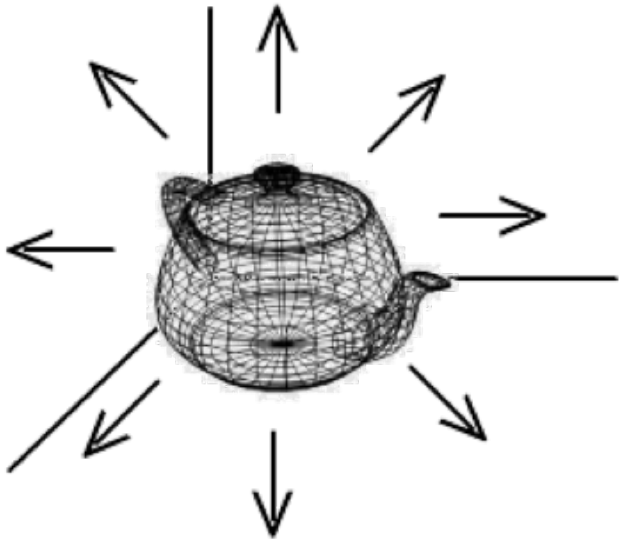
$$L(x, \omega_r) = \sum_l L\left(l, \overrightarrow{lx}\right) f_r\left(x, \overrightarrow{lx}, \omega_r\right) + \boldsymbol{l}_A f_A(x, \omega_r)$$

Ambient lighting

In most of the cases the BRDF for the ambient term $f_A(x, \omega_r)$ is a constant known as the *ambient light reflection color* m_A.

$$\boldsymbol{m}_A = (m_{AR}, m_{AG}, m_{AB})$$

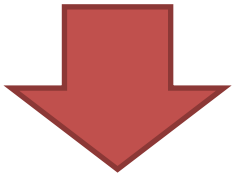
Generally, m_A corresponds to the main color of the object, but it can be tuned to obtain special lighting for particular objects.



Ambient lighting

In case of a single direct light, plus the ambient term (which is assumed to be constant), the rendering equations reduces to:

$$L(x,\omega_r) = \sum_l L\left(l,\overrightarrow{lx}\right) f_r\left(x,\overrightarrow{lx},\omega_r\right) + \boldsymbol{l}_A f_A(x,\omega_r)$$



$$L(x,\omega_r) = \boldsymbol{l} * f_r(x,\boldsymbol{d},\omega_r) + \boldsymbol{l}_A * \boldsymbol{m}_A$$

Hemispheric lighting

A slight extension of ambient lighting is the *hemispheric lighting*.

In this case, there are two ambient light colors (the “upper” or “sky” color, and the “lower” or “ground” color) and a direction vector.

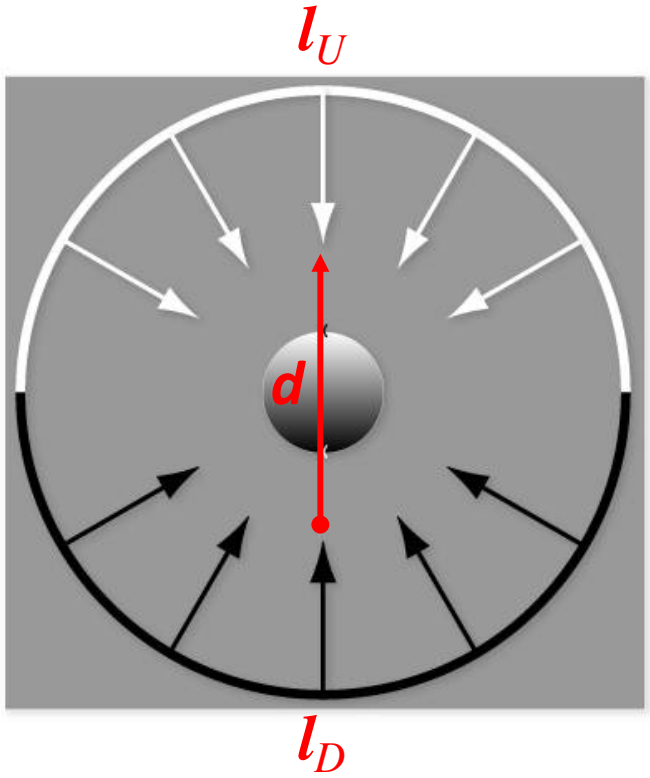
This model simulates the fact that both the color of the sky and the one of the ground have a big impact on the indirect light component for the considered object.

Hemispheric lighting

The technique creates an ambient light color factor by “blending” the two colors, with respect to the orientation of the object.

Hemispheric lighting

The two colors, $\boldsymbol{l}_U$ and $\boldsymbol{l}_D$, represent the values of the ambient light at the two extremes, and the direction vector ***d*** orients the blending of the two colors.



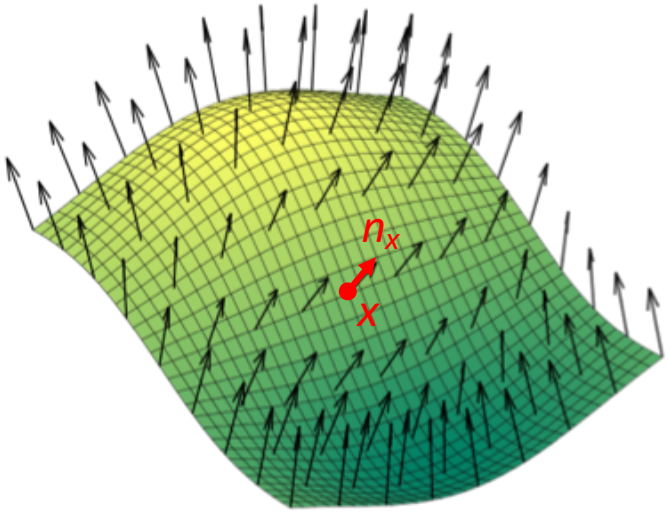
Hemispheric lighting

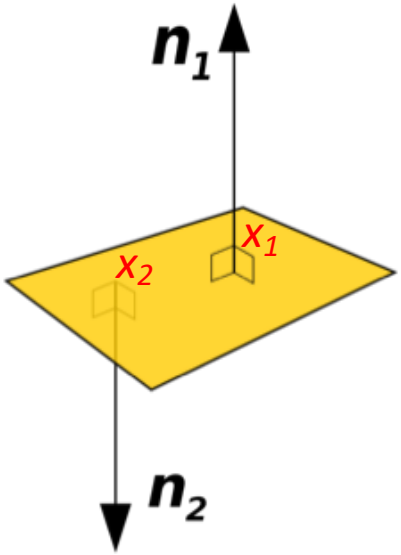
The rendering equation is then modified to have the ambient light term $l_A(x)$ that depends on the orientation of the surface of the object in the considered point *x*.

$$L(x, \omega_r) = \sum_l L\left(l, \overrightarrow{lx}\right) f_r\left(x, \overrightarrow{lx}, \omega_r\right) + \boldsymbol{l}_A(x) f_A(x, \omega_r)$$

Hemispheric lighting

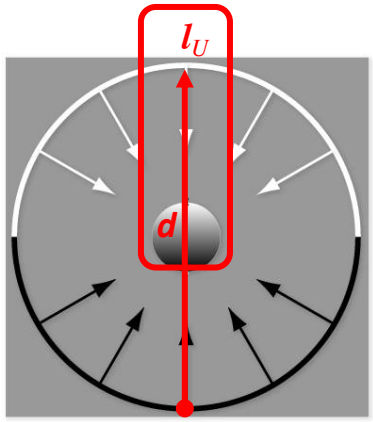
The orientation of the object is characterized by n_x the normal vector to the surface in point x.

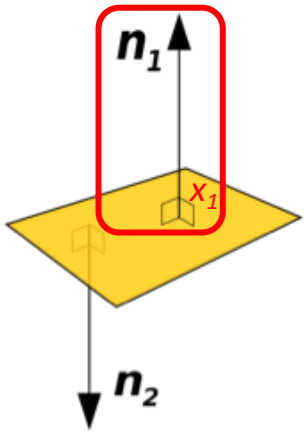




Hemispheric lighting

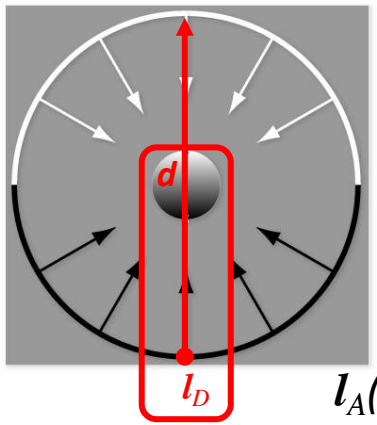
If the normal vector is aligned and in the same direction as ***d***, ambient color l_U is used.

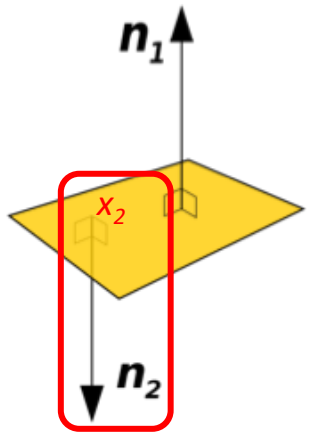




$l_A(x_1) = l_U$

If the normal vector is instead aligned, but in the opposite direction of ***d***, ambient color l_D is used.

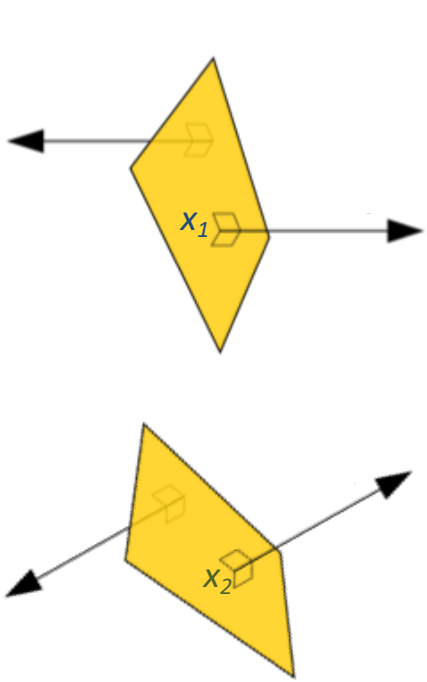


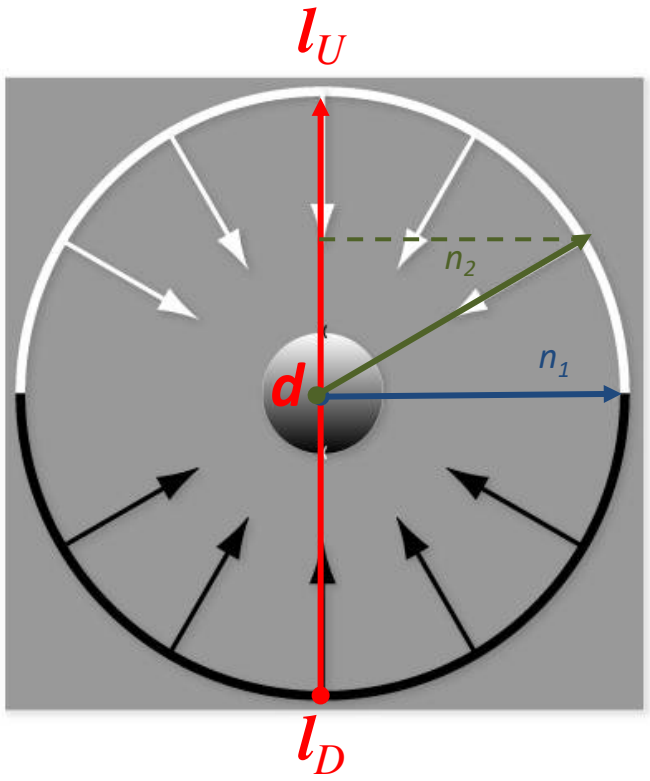


$l_A(x_2) = l_D$

Hemispheric lighting

For normal vectors oriented in other directions, the two colors are blended proportionally to the cosine of their angle with vector ***d***.





$l_A(x_2) = \frac{3}{4}\, l_U + \frac{1}{4}\, l_D$

$l_A(x_1) = \frac{1}{2}\, l_U + \frac{1}{2}\, l_D$

Hemispheric lighting

In particular, $\boldsymbol{l}_A(x)$ is defined as follows:

$$\boldsymbol{l}_A(x) = \frac{n_x \cdot d + 1}{2}\boldsymbol{l}_U + \frac{1 - n_x \cdot d}{2}\boldsymbol{l}_D$$

In case of a single direct light plus the hemispheric ambient term, the rendering equations reduces to:

$$L(x, \omega_r) = \boldsymbol{l} * f_r(x, \boldsymbol{d}, \omega_r) + \left(\frac{\boldsymbol{n}_x \cdot \boldsymbol{d} + 1}{2}\boldsymbol{l}_U + \frac{1 - \boldsymbol{n}_x \cdot \boldsymbol{d}}{2}\boldsymbol{l}_D\right) * \boldsymbol{m}_A$$

The road to Image Based Lighting

More accurate reproduction of light sources and more advanced approximations to the rendering equations are the key to the photo-realistic effects we are now used to see in high-end 3D applications.

Although a complete description of such techniques is outside the scope of this course, we can briefly highlight some ideas that drives some of them.

Image based lighting

The hemispheric lighting we have just introduced, computes the light received by one object as function of the direction in which the points on its surface are oriented, as specified by the direction of the corresponding normal vector.

In this case, however, the function is very simple: it just interpolates two colors according to relative orientation with respect to a given direction.

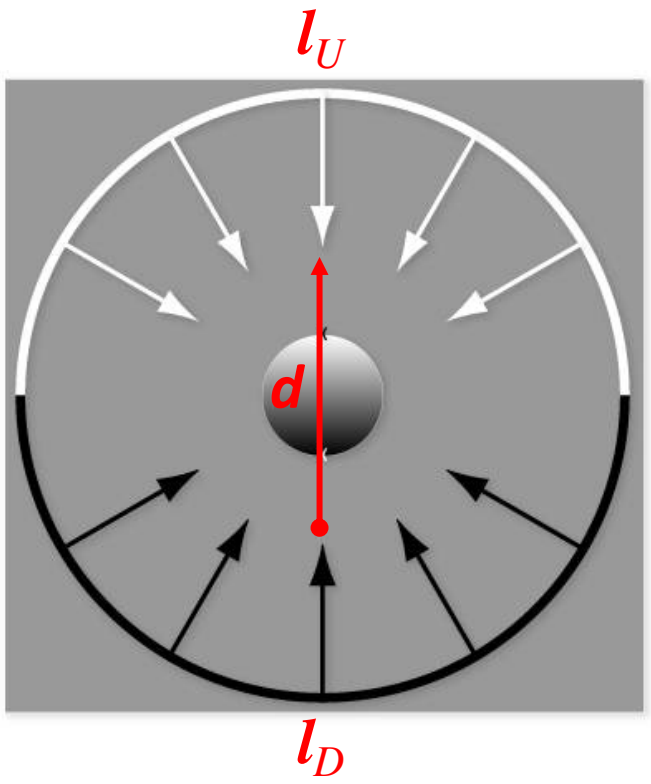


Image based lighting

The next idea is then to have generic functions $l_A(x_i)$ that return the color received from the outside environment by a point x_i on a surface oriented in the direction described by its normal vector n_i.

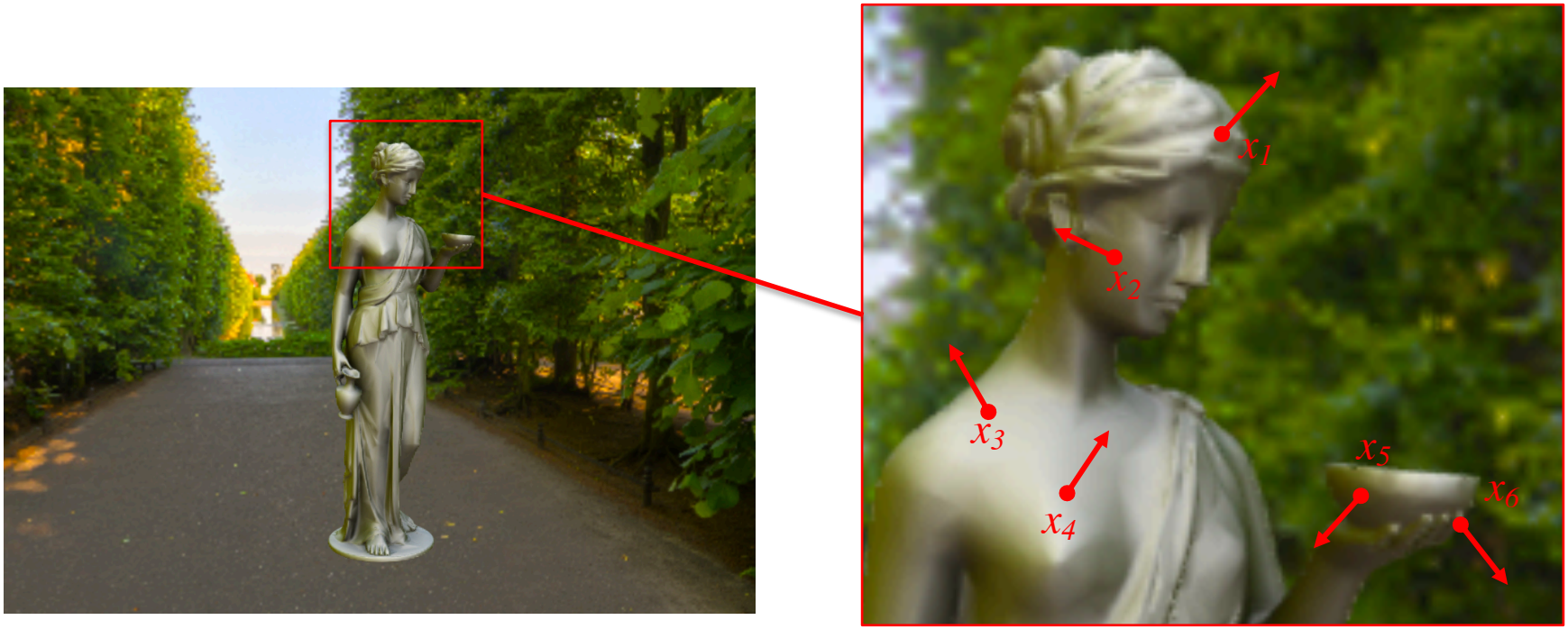


Image based lighting

Each point is thus illuminated according to $l_A(x_i)$.

Image based lighting

These functions $l_A(x_i)$ can be computed either from specially taken pictures or from high quality off-line rendering of the environment, and encoded in a way that can be used in real time.

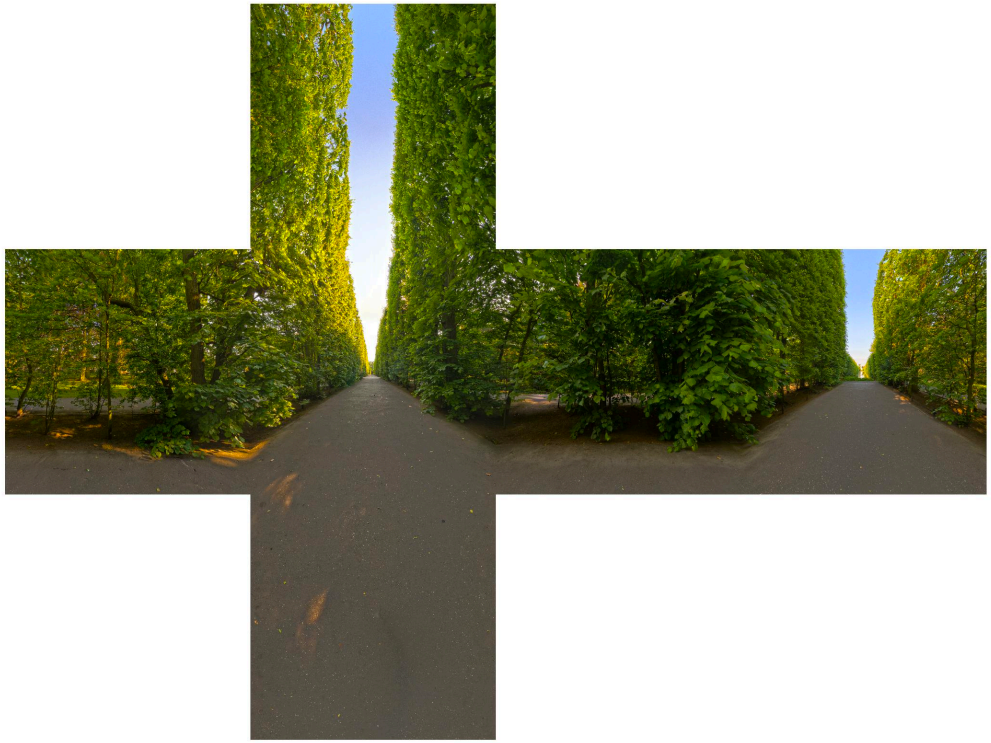
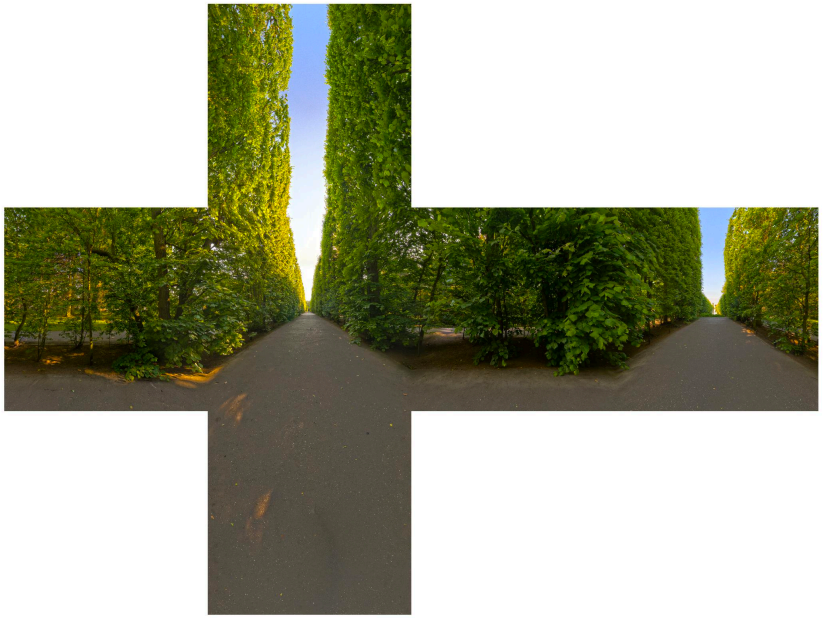


Image based lighting

Starting from an image (this motivates the name Image based lighting), and applying a step called *filtering*, the actual light received from each point in any direction can be determined.



Standard cubemap

Irradiance cubemap

Image based lighting

The approximation of the rendering equation remains the same as the one seen for the hemispheric lighting: only the light function changes to return the values computed in the filtering step.

In this case, the approximation includes both ambient and diffuse components of the light.

$$L(x,\omega_r) = \sum_l L\left(l,\overrightarrow{lx}\right) f_r\left(x,\overrightarrow{lx},\omega_r\right) + \boldsymbol{l}_A(x) f_A(x,\omega_r)$$

Image based lighting

Encoding function $\boldsymbol{l}_A(x)$ in an efficient way is not a simple task. The most common approaches are:

- Interpolating values stored in a table (*Cubic Mapping*)
- Spectral expansion (*Spherical Harmonics*)

We will present Cubic Mapping in a following lesson, and a complete description of Spherical Harmonics is outside the scope of this course.

Image based lighting

However, the simplest expansion using the *Spherical Harmonics*, can be briefly introduced, since it requires only four values: a basic color $\boldsymbol{l}_C$, plus three "deviation terms" along the three main axis $\Delta\boldsymbol{l}_x$, $\Delta\boldsymbol{l}_y$ and $\Delta\boldsymbol{l}_z$.

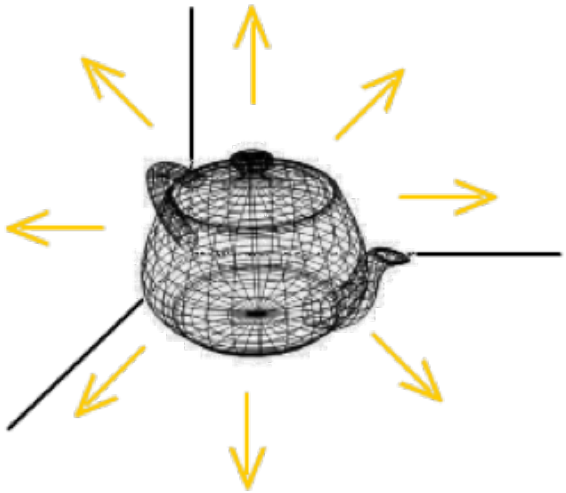
If we call respectively $(\boldsymbol{n}_x).x$, $(\boldsymbol{n}_x).y$ and $(\boldsymbol{n}_x).z$ the components of the (unitary) normal vector direction, we have:

$$\boldsymbol{l}_A(x) = \boldsymbol{l}_C + (n_x).x\ \boldsymbol{\Delta l}_x + (n_x).y\ \boldsymbol{\Delta l}_y + (n_x).z\ \boldsymbol{\Delta l}_z$$

Material emission

The *emission* term of a material accounts for small amounts of light emitted directly by an object.

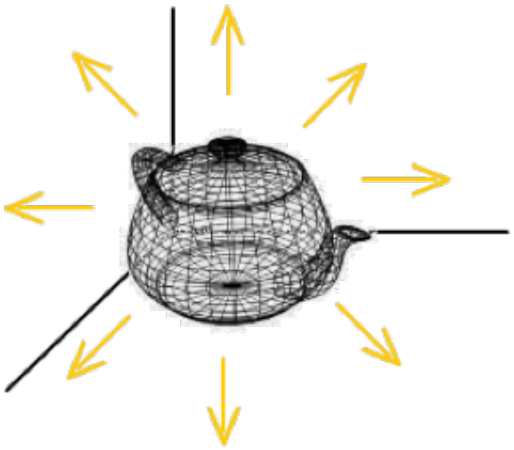
$$\boldsymbol{m}_E = (m_{ER}, m_{EG}, m_{EB})$$



Material emission

It corresponds to the emissive part of the rendering equation.

$$\boldsymbol{m}_E = (m_{ER}, m_{EG}, m_{EB})$$



$$L(x, \omega_r) = L_e(x, \omega_r) + \sum_l L_e\left(l, \overrightarrow{lx}\right) f_r\left(x, \overrightarrow{lx}, \omega_r\right)$$

Material emission

The material emission term is summed to the other parts of the rendering equation. The main difference with ambient light, is that it does not depend on the environment, but only on the considered object.

For example, if we consider direct light, Phong specular reflection, ambient light and emission, we have:

$$\boldsymbol{r}_x = 2\boldsymbol{n}_x \cdot (\boldsymbol{d} \cdot \boldsymbol{n}_x) - \boldsymbol{d}$$

$$L(x, \omega_r) = clamp(\boldsymbol{l}_D * (\boldsymbol{m}_D \cdot clamp(\boldsymbol{d} \cdot \boldsymbol{n}_x) + \boldsymbol{m}_S \cdot clamp(\omega_r \cdot \boldsymbol{r}_x)^\gamma) + \boldsymbol{l}_A * \boldsymbol{m}_A + \boldsymbol{m}_E)$$

Vertex attributes and Uniforms

As we have seen, *Shaders* are the pipeline components that compute the positions and the colors of the pixels on screen corresponding to points of the objects.

The Light and BRDF models seen in the previous lessons, which are the building blocks for computing approximations to the rendering equations, are the main algorithms with which the color of the pixel can be determined.

Such algorithms, can be parametrized to produce results that depends on the surface of the objects.

Vertex attributes and Uniforms

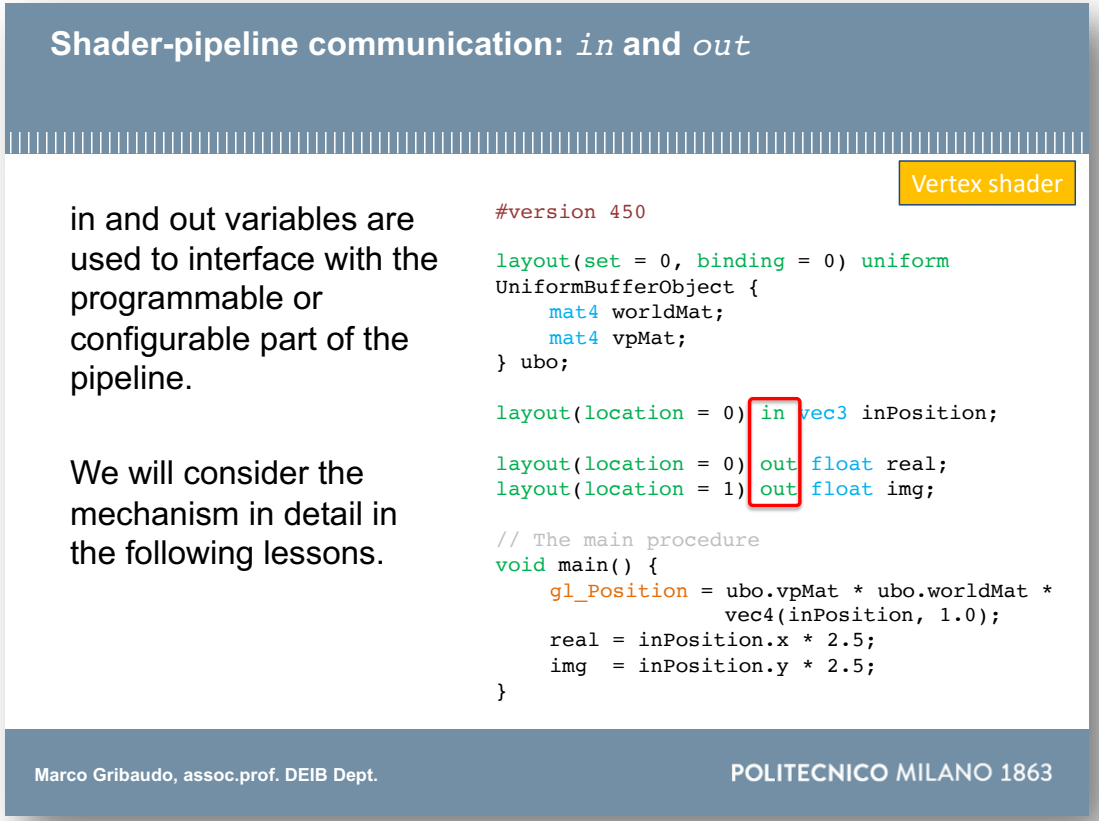
Object and environment parameters can be passed to the Shaders in two ways:

- Vertex attributes
- Uniform variables

In this lesson, we will investigate the former, and we will introduce the latter in a future presentation.

Vertex attributes: refresh

Let us remember that `in` and `out` variables are one of the way in which *Shaders* interfaces with the other components (either fixed or programmable) of the pipeline.



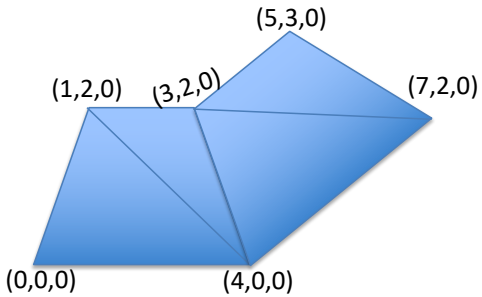
Vertex attributes: refresh

Let us also remember that graphics primitives are sent to the graphic pipeline as *triangles lists* or *triangle strips*, encoded with the values defining their vertices.

Encoding example

Example:

The following part of a geometrical primitive can be encoded with either a triangle strip or with a triangle list. Let us use both encodings, and let us compute the memory requirements supposing that each vertex occupies 12 Bytes.



Strip: {(0,0,0), (1,2,0), (4,0,0),
(3,2,0), (7,2,0), (5,3,0) }
Space required: 12 * 6 = 72 Bytes

List: {(0,0,0), (1,2,0), (4,0,0),
(1,2,0), (4,0,0), (3,2,0),
(4,0,0), (3,2,0), (7,3,0),
(3,2,0), (7,3,0), (5,3,0)}
Space required: 12 * 12 = 144 Bytes

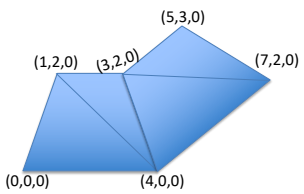
Vertex attributes: refresh

In the graphics pipeline, values associated with the vertices are sent directly to the *Vertex Shader* by the *Input Assembler* pipeline component.

Encoding example

Example:

The following part of a geometrical primitive can be encoded with either a triangle strip or with a triangle list. Let us use both encodings, and let us compute the memory requirements supposing that each vertex occupies 12 Bytes.



Strip: {(0,0,0), (1,2,0), (4,0,0), (3,2,0), (7,2,0), (5,3,0) }

Space required: 12 * 6 = 72 Bytes

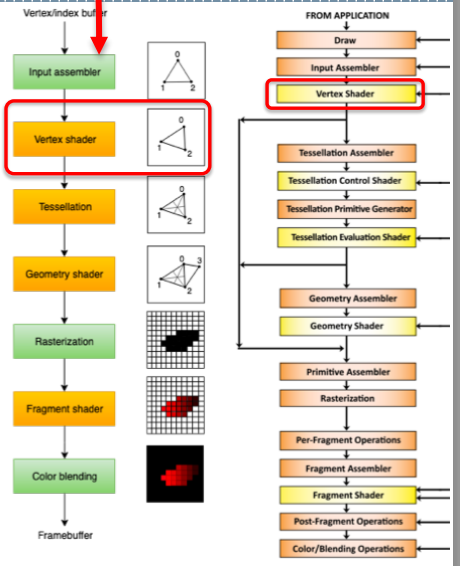
List: {(0,0,0), (1,2,0), (4,0,0), (1,2,0), (4,0,0), (3,2,0), (4,0,0), (3,2,0), (7,3,0), (3,2,0), (7,3,0), (5,3,0)}

Space required: 12 * 12 = 144 Bytes

The graphics pipeline

Vertex shaders are then executed to perform operations on each vertex.

Such operations, for example, transform local coordinates to clipping coordinates by multiplying vertex positions with the corresponding WVP matrix, or compute colors and other values associated to vertices, which will be used in later stages of the process.



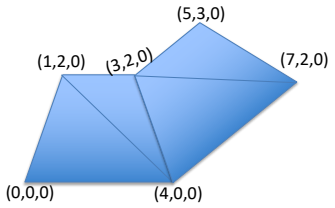
Vertex attributes: refresh

In particular, vertex coordinates are sent into `in` variables of the Vertex Shader.

Encoding example

Example:

The following part of a geometrical primitive can be encoded with either a triangle strip or with a triangle list. Let us use both encodings, and let us compute the memory requirements supposing that each vertex occupies 12 Bytes.



Strip: {(0,0,0), (1,2,0), (4,0,0), (3,2,0), (7,2,0), (5,3,0) }

Space required: 12 * 6 = 72 Bytes

List: {(0,0,0), (1,2,0), (4,0,0), (1,2,0), (4,0,0), (3,2,0), (4,0,0), (3,2,0), (7,3,0), (5,3,0)}

Space required: 12 * 12 = 144 Bytes

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;
}
```

Vertex attributes: ID

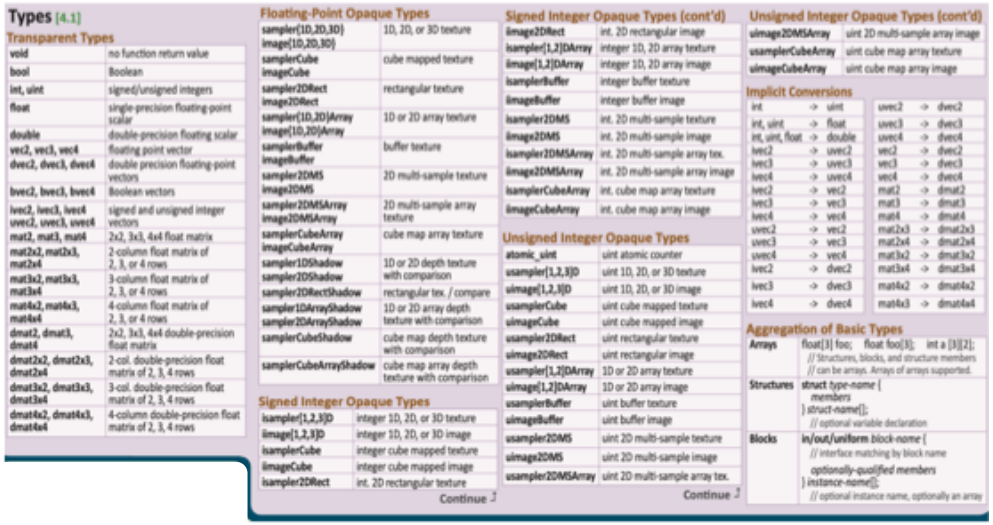
Each vertex has an implicit integer value that represents its index. It is contained in global variable `gl_VertexIndex`.

```
1 layout(location = 0) out vec3 fragColor;
2
3 void main() {
4     gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
5     fragColor = colors[gl_VertexIndex];
6 }
```

From https://vulkan-tutorial.com/

Vertex attributes: user defined values

In addition, it can have other an arbitrary set of user defined attributes, each one characterized by one of the supported GLSL types. Vertices may also have no user defined attributes.




```
#version 450

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

layout(location = 0) in vec3 inPosition;

void main() {
     gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
}
```

Vertex attributes: user defined values

A bi-dimensional scene, can use for example, just a set of `vec2` normalized screen coordinates to denote the position of the elements to connect.

```
#version 450

layout(location = 0) in vec2 inPosition;

void main() {
     gl_Position = vec4(inPosition, 0.0, 1.0);
}
```

Vertex attributes: user defined values

A classical 3D scene, such as the one in *Assignment 09*, uses a single `vec3` element to store the positions in the 3D local space.

```
#version 450

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

layout(location = 0) in vec3 inPosition;

layout(location = 0) out vec3 fragPos;

void main() {
     gl_Position = ubo.mvpMat * vec4(inPosition, 1.0);
     fragPos = (ubo.mMat * vec4(inPosition, 1.0)).xyz;
}
```

Vertex attributes: user defined values

Other pipelines might require for each vertex a `vec3` position (measured in the 3D local coordinates), and a `vec3` color to have the diffuse reflection varying over the surface of the objects.

```
#version 450

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

layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

void main() {
     gl_Position = ubo.mvpMat * vec4(inPosition, 1.0);
     fragColor = inColor;
}
```

Vertex attributes: user defined values

All the vertices of one mesh must have the same vertex format, i.e. the same attributes. The fixed functions of the pipeline pass such values to the Vertex Shaders.

Different meshes might be characterized by different vertices formats (however this will require the creation of different pipelines).

In the following lessons we will see several examples of vertices formats, each one aimed at supporting a different rendering effect.

Communication between Vertex and Fragment shaders

Other components of the pipeline, such as the *Fragment Shaders*, cannot directly access the attributes associated with a vertex.

The Vertex Shader must pass such values to other components of the pipeline using the `out` variables.

```
#version 450

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

layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

void main() {
     gl_Position = ubo.mvpMat * vec4(inPosition, 1.0);
     fragColor = inColor;
}
```

Communication between Vertex and Fragment shaders

The `in` and `out` variables used for the shaders communication, is implemented with a set of *slots*, each one identified by a `location` number.

Location numbers starts from zero, and are limited by a hardware dependent constant (usually large enough to support standard applications).

Communication between Vertex and Fragment shaders

Whenever an `in` or `out` variable is defined, the user provides the location id of the slot used for communication in a `layout` directive.

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;
}
```

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;
		}
…
}
```

Communication between Vertex and Fragment shaders

Vulkan relies on the developer to use the correct locations values when defining `in` and `out` variables. An error in these settings creates unexpected and unpredictable results.

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;
}
```

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;
		}
…
}
```

Communication between Vertex and Fragment shaders

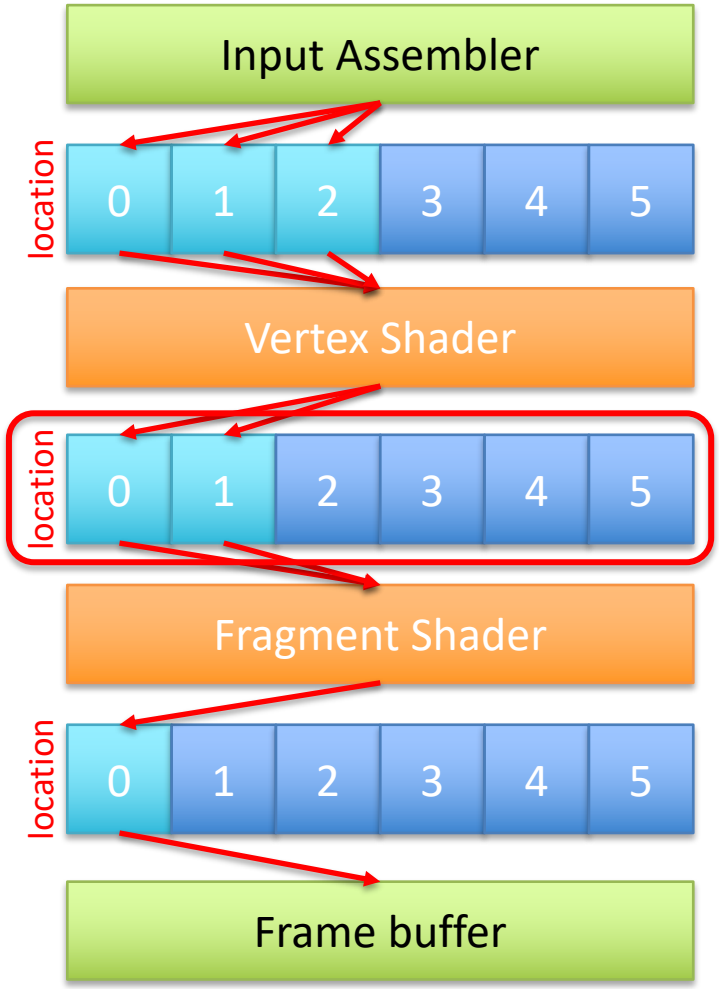
The slots used by the *Input Assembler*, which will be available inside `in` variables of the *Vertex Shader*, are configured in the pipeline creation.

The configuration of the pipeline, also defines the `out` variables that *the Fragment Shader* will write to set the final color of the considered pixel (fragment).

Communication between Vertex and Fragment shaders

Communication between the Vertex and Fragment shader is controlled by their GLSL specification.

As we will see, the fixed functions of the pipeline interpolate the values of the `out` variables emitted by the *Vertex Shader*, according to the position of the corresponding pixels on screen, before passing their values to the *Fragment Shader*.

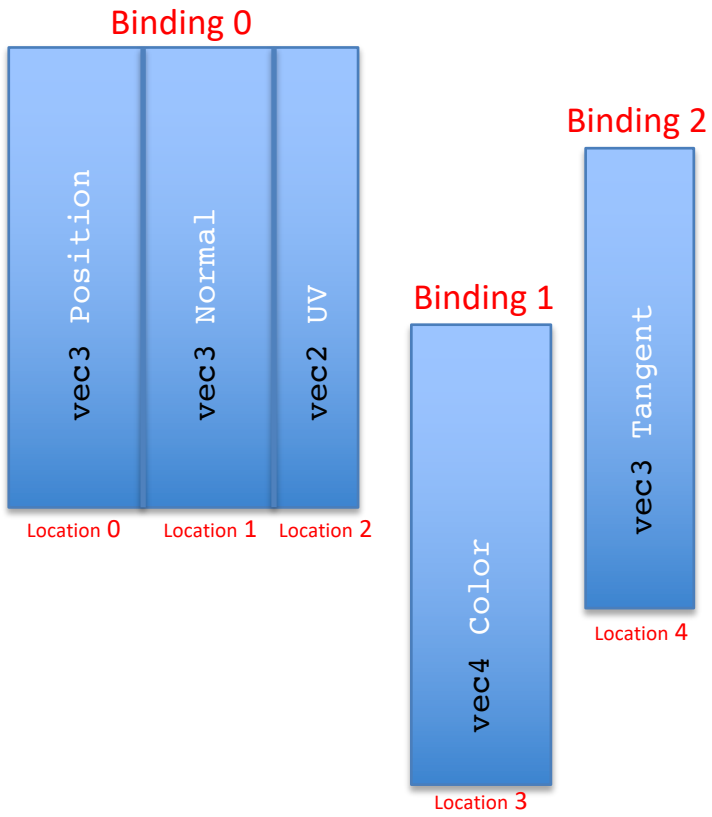


Vertex attributes: Vertex Input Descriptors

Vulkan is very flexible in configuring the pipeline for specifying the vertex attributes to send to the *Vertex Shader*.

In particular, it allows to split vertex data into different arrays, each one containing some of the attributes.

Each of these array is known as a *binding*, and it is characterized by a progressive `binding` id.



Vertex attributes: Vertex Input Descriptors

The most common approach, however, is to use a single binding.

In particular, we create a C++ structure containing all the vertex attributes, using the GLM types that match the ones defined in the corresponding *Vertex Shader*.

```
struct Vertex {
      glm::vec3 pos;
      glm::vec3 normal;
      glm::vec4 tangent;
      glm::vec2 texCoord;
}
```

PBRShader.vert

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 0) uniform UniformBufferObject {
    mat4 mvpMat;
    mat4 mMat;
    mat4 nMat;
} ubo;

layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec3 inNormal;
layout(location = 2) in vec4 inTangent;
layout(location = 3) in vec2 inTexCoord;

layout(location = 0) out vec3 fragPos;
layout(location = 1) out vec3 fragNorm;
layout(location = 2) out vec4 fragTan;
layout(location = 3) out vec2 fragTexCoord;

void main() {
    gl_Position = ubo.mvpMat * vec4(inPosition, 1.0);
    fragPos = (ubo.mMat * vec4(inPosition, 1.0)).xyz;
    fragNorm = mat3(ubo.nMat) * inNormal;
    fragTan = vec4(mat3(ubo.nMat) * inTangent.xyz, inTangent.w);
    fragTexCoord = inTexCoord;
}
```

Vertex attributes: Vertex Input Descriptors

The binding, is defined inside a `VkVertexInputBindingDescription` structure.

Its field contains the binding *id* and the size of the object in bytes.

The `inputRate` field can be used for instanced rendering – we will not consider it in this course, and we will keep it to the value shown here.

```
struct Vertex {
    glm::vec3 pos;
    glm::vec3 normal;
    glm::vec4 tangent;
    glm::vec2 texCoord;
}
VkVertexInputBindingDescription bindingDescription{};
bindingDescription.binding = 0;
bindingDescription.stride = sizeof(Vertex);
bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
```

The size of the object in bytes, which must be set in the `stride` field, can be easily computed with the `sizeof()` macro.

Vertex attributes: Vertex Input Descriptors

Single attributes are defined inside the element of an array of `VkVertexInputAttributeDescription` structures.

```
struct Vertex {
      glm::vec3 pos;
      glm::vec3 normal;
      glm::vec4 tangent;
      glm::vec2 texCoord;
}
std::array<VkVertexInputAttributeDescription, 4>
                        attributeDescriptions{};

attributeDescriptions[0].binding = 0;
attributeDescriptions[0].location = 0;
attributeDescriptions[0].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[0].offset = offsetof(Vertex, pos);

attributeDescriptions[1].binding = 0;
attributeDescriptions[1].location = 1;
attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[1].offset = offsetof(Vertex, normal);

attributeDescriptions[2].binding = 0;
attributeDescriptions[2].location = 2;
attributeDescriptions[2].format = VK_FORMAT_R32G32B32A32_SFLOAT;
attributeDescriptions[2].offset = offsetof(Vertex, tangent);

attributeDescriptions[3].binding = 0;
attributeDescriptions[3].location = 3;
attributeDescriptions[3].format = VK_FORMAT_R32G32_SFLOAT;
attributeDescriptions[3].offset = offsetof(Vertex, texCoord);
```

Vertex attributes: Vertex Input Descriptors

Each attribute definition contain the specification of both its `binding` and its `location` ids.

```
struct Vertex {
      glm::vec3 pos;
      glm::vec3 normal;
      glm::vec4 tangent;
      glm::vec2 texCoord;
}
std::array<VkVertexInputAttributeDescription, 4>
                        attributeDescriptions{};

attributeDescriptions[0].binding = 0;
attributeDescriptions[0].location = 0;
attributeDescriptions[0].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[0].offset = offsetof(Vertex, pos);

attributeDescriptions[1].binding = 0;
attributeDescriptions[1].location = 1;
attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[1].offset = offsetof(Vertex, normal);

attributeDescriptions[2].binding = 0;
attributeDescriptions[2].location = 2;
attributeDescriptions[2].format = VK_FORMAT_R32G32B32A32_SFLOAT;
attributeDescriptions[2].offset = offsetof(Vertex, tangent);

attributeDescriptions[3].binding = 0;
attributeDescriptions[3].location = 3;
attributeDescriptions[3].format = VK_FORMAT_R32G32_SFLOAT;
attributeDescriptions[3].offset = offsetof(Vertex, texCoord);
```

Vertex attributes: Vertex Input Descriptors

A constant specifying its data type (format) is then required.

```
struct Vertex {
      glm::vec3 pos;
      glm::vec3 normal;
      glm::vec4 tangent;
      glm::vec2 texCoord;
}
std::array<VkVertexInputAttributeDescription, 4>
                        attributeDescriptions{};

attributeDescriptions[0].binding = 0;
attributeDescriptions[0].location = 0;
attributeDescriptions[0].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[0].offset = offsetof(Vertex, pos);

attributeDescriptions[1].binding = 0;
attributeDescriptions[1].location = 1;
attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[1].offset = offsetof(Vertex, normal);

attributeDescriptions[2].binding = 0;
attributeDescriptions[2].location = 2;
attributeDescriptions[2].format = VK_FORMAT_R32G32B32A32_SFLOAT;
attributeDescriptions[2].offset = offsetof(Vertex, tangent);

attributeDescriptions[3].binding = 0;
attributeDescriptions[3].location = 3;
attributeDescriptions[3].format = VK_FORMAT_R32G32_SFLOAT;
attributeDescriptions[3].offset = offsetof(Vertex, texCoord);
```

Vertex attributes: Vertex Input Descriptors

The most common formats are the following:

`float`	`VK_FORMAT_R32_SFLOAT`
`vec2`	`VK_FORMAT_R32G32_SFLOAT`
`vec3`	`VK_FORMAT_R32G32B32_SFLOAT`
`vec4`	`VK_FORMAT_R32G32B32A32_SFLOAT`

A complete list can be found here:
https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VkFormat.html

Vertex attributes: Vertex Input Descriptors

Finally, the offset in byte inside the data structure for the considered field must be provided.

This can be computed using the `offsetof()` macro.

```
struct Vertex {
      glm::vec3 pos;
      glm::vec3 normal;
      glm::vec4 tangent;
      glm::vec2 texCoord;
}
std::array<VkVertexInputAttributeDescription, 4>
                        attributeDescriptions{};

attributeDescriptions[0].binding = 0;
attributeDescriptions[0].location = 0;
attributeDescriptions[0].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[0].offset = offsetof(Vertex, pos);

attributeDescriptions[1].binding = 0;
attributeDescriptions[1].location = 1;
attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[1].offset = offsetof(Vertex, normal);

attributeDescriptions[2].binding = 0;
attributeDescriptions[2].location = 2;
attributeDescriptions[2].format = VK_FORMAT_R32G32B32A32_SFLOAT;
attributeDescriptions[2].offset = offsetof(Vertex, tangent);

attributeDescriptions[3].binding = 0;
attributeDescriptions[3].location = 3;
attributeDescriptions[3].format = VK_FORMAT_R32G32_SFLOAT;
attributeDescriptions[3].offset = offsetof(Vertex, texCoord);
```

Input Assembler configuration

Next, both binding and attributes descriptors are collected into a `VkPipelineVertexInputStateCreateInfo` structure.

```
VkPipelineVertexInputStateCreateInfo vertexInputInfo{};
vertexInputInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;

vertexInputInfo.vertexBindingDescriptionCount = 1;
vertexInputInfo.vertexAttributeDescriptionCount = 4;
vertexInputInfo.pVertexBindingDescriptions = &bindingDescription;
vertexInputInfo.pVertexAttributeDescriptions = attributeDescriptions.data();

VkPipelineInputAssemblyStateCreateInfo inputAssembly{};
inputAssembly.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
inputAssembly.primitiveRestartEnable = VK_FALSE;
```

Input Assembler configuration

The structure requires the pointer to both the binding and attribute description arrays, and the count of the corresponding elements.

```
VkPipelineVertexInputStateCreateInfo vertexInputInfo{};
vertexInputInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;

vertexInputInfo.vertexBindingDescriptionCount = 1;
vertexInputInfo.vertexAttributeDescriptionCount = 4;
vertexInputInfo.pVertexBindingDescriptions = &bindingDescription;
vertexInputInfo.pVertexAttributeDescriptions = attributeDescriptions.data();

VkPipelineInputAssemblyStateCreateInfo inputAssembly{};
inputAssembly.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
inputAssembly.primitiveRestartEnable = VK_FALSE;
```

Input Assembler configuration

The configuration of the *Input Assembler* stage of the graphic pipeline is competed filling up a `VkPipelineInputAssemblyStateCreateInfo` structure with the type of primitives being drawn – i.e. triangle lists or triangle strips (with or without restart).

```
VkPipelineVertexInputStateCreateInfo vertexInputInfo{};
vertexInputInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;

vertexInputInfo.vertexBindingDescriptionCount = 1;
vertexInputInfo.vertexAttributeDescriptionCount = 4;
vertexInputInfo.pVertexBindingDescriptions = &bindingDescription;
vertexInputInfo.pVertexAttributeDescriptions = attributeDescriptions.data();

VkPipelineInputAssemblyStateCreateInfo inputAssembly{};
inputAssembly.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
inputAssembly.primitiveRestartEnable = VK_FALSE;
```