POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

DE
IB

2024

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2024

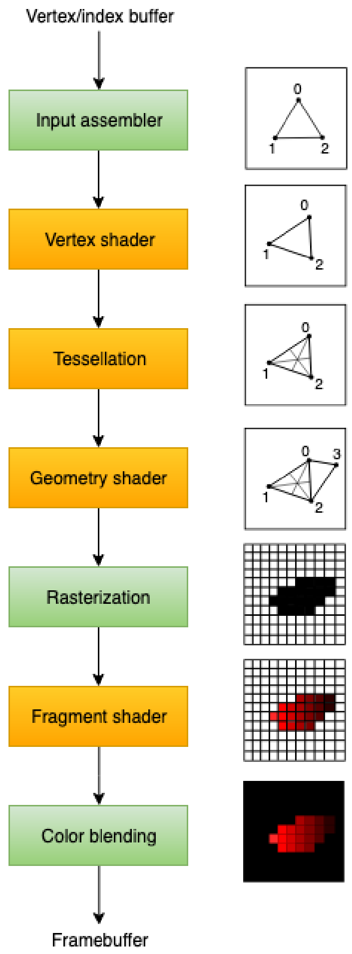
Computer Graphics

- Layouts – Part I

Pipelines

As we have seen, the Vulkan Graphics pipeline:

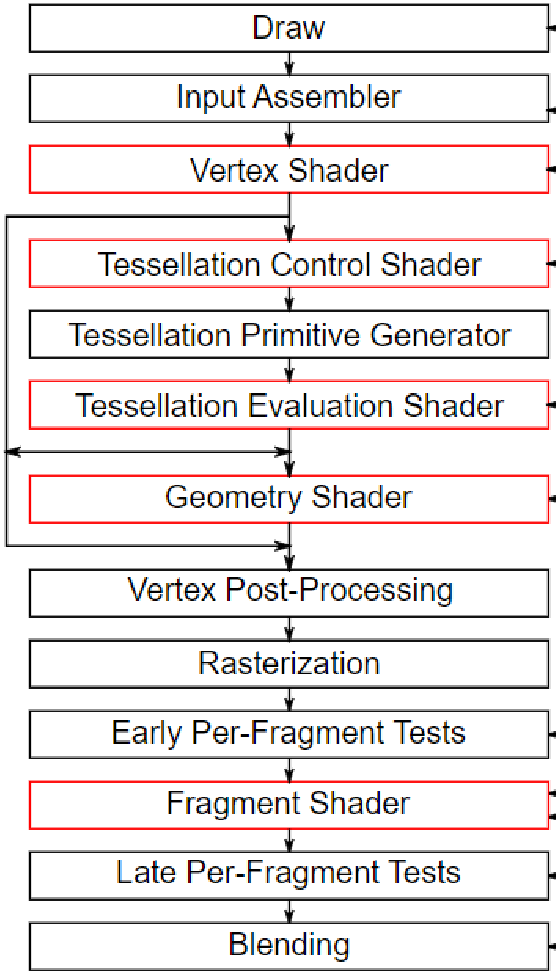
- It starts from a Mesh defined by a set of vertices and indices.
- It processes each vertex with a Vertex Shader that computes its normalized screen coordinates, plus other parameters to pass to the Fragment shader.
- For every Fragment (Pixel) on screen, it calls the Fragment shader to compute the final color of the pixel (for example, by implementing the corresponding BRDF).



Pipelines

The Graphic pipeline, although being based on a set of fixed functions, which can only be configured by the user, must support a large number of different use-cases, each one characterized by its own features.

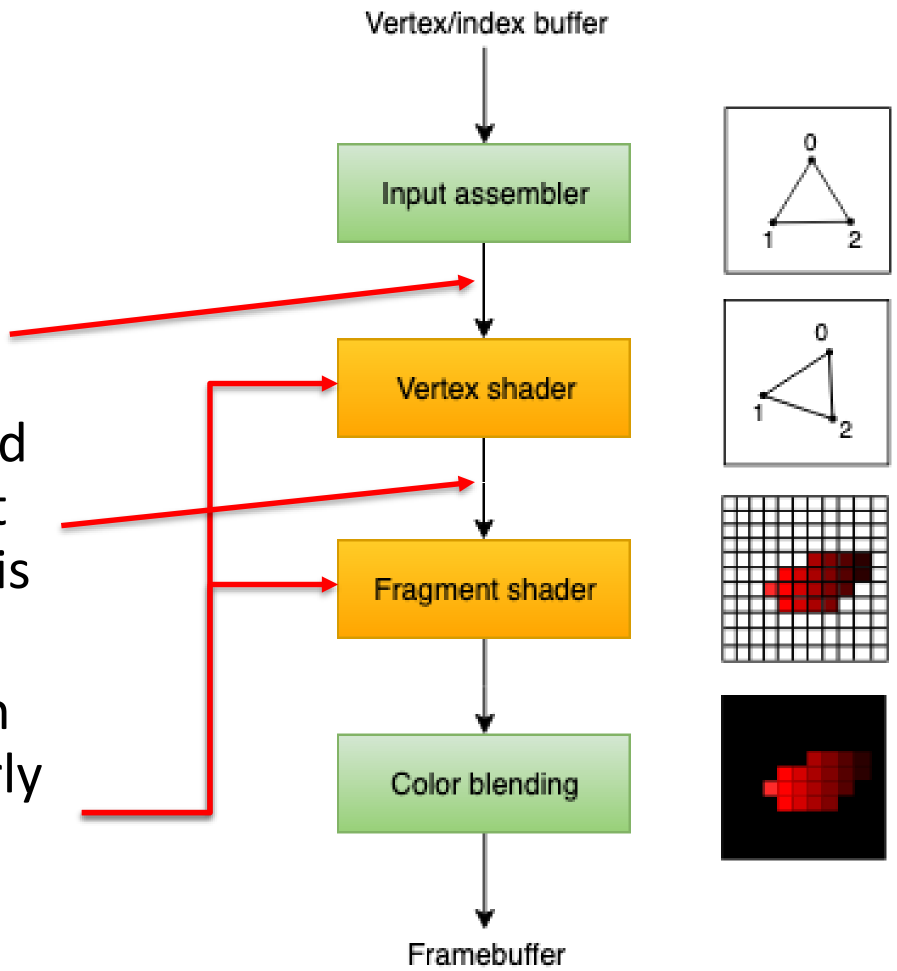
There cannot be a unique solution: Vulkan must allow the user to program what works better in each different case study.



Pipelines

Beside offering the opportunity of writing shaders, Vulkan allows to:

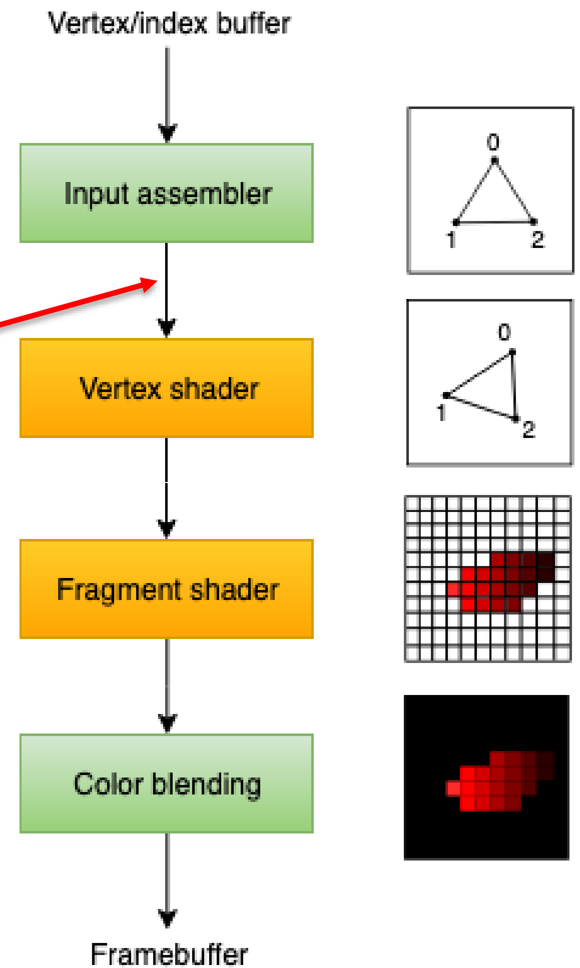
- Define which information is associated with vertices.
- Define which information is passed between the Vertex and Fragment Shaders, and whether and how it is interpolated.
- Define what other parameters can be passed to the shader to properly process vertices and compute fragments.



Vertex data

Data associated to vertices includes:

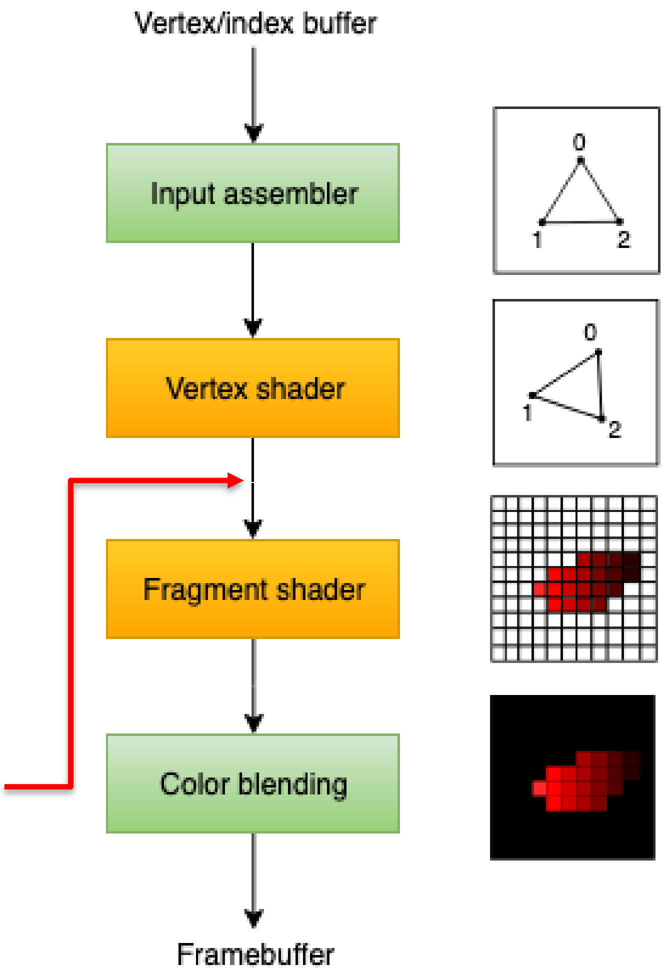
- Positions in the 3D or 2D space
- Normal vector directions (for Smooth Shading)
- UV coordinates
- Colors
- Tangents (for example in the Ward BRDF)
- … and many more!



Intra-Shaders data

Examples of data passed between the Shaders are:

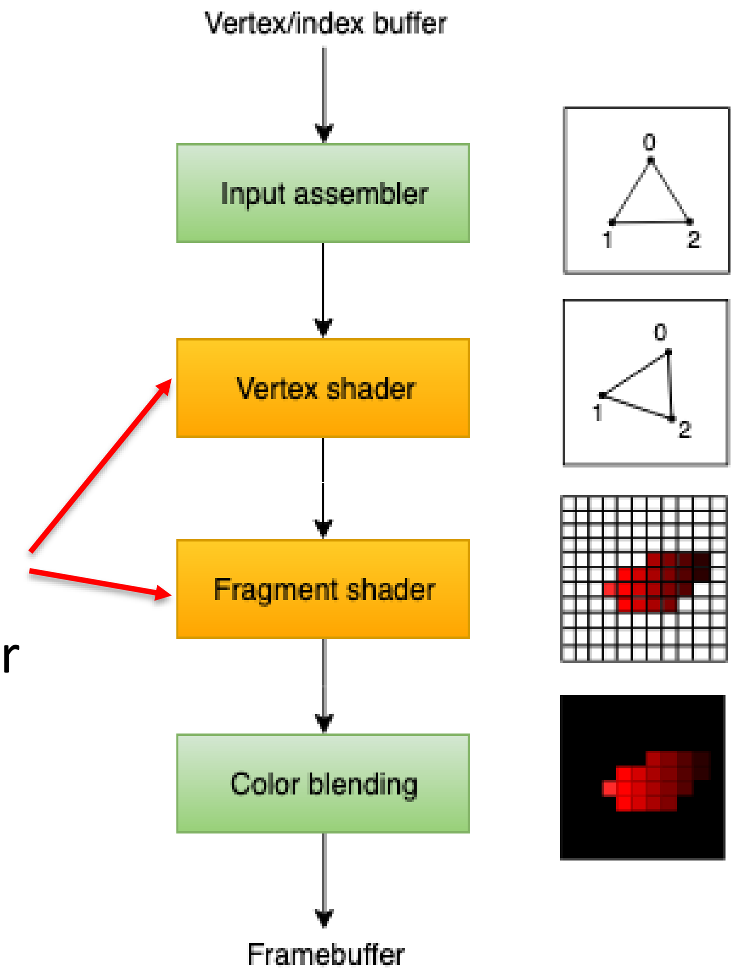
- Interpolated normal vector directions (Phong shading)
- Interpolated pixel color (Gouraud shading)
- Coordinates of the points in space (to compute the view vector)
- Interpolated UV coordinates (to implement textures)



CPU -> Shader data

Data passed to the Shaders can be:

- Transform Matrices (World, View, Projection, Normal vectors, …)
- Lights definitions (Positions, Directions, Colors, Ranges, …)
- Viewer positions
- Ambient and Indirect Light coefficients
- Material definitions (Diffuse and Specular color, roughness, specular power, …)
- Textures
- …



Layouts

In Vulkan, whatever data structure used to define the format and the type of information being used in any user-defined encoding is called *Layout*.

The term is also used to define the pixel color encoding being used (i.e. how many bits per pixel are used, which color space is selected, the presence of an alpha-channel and so on).

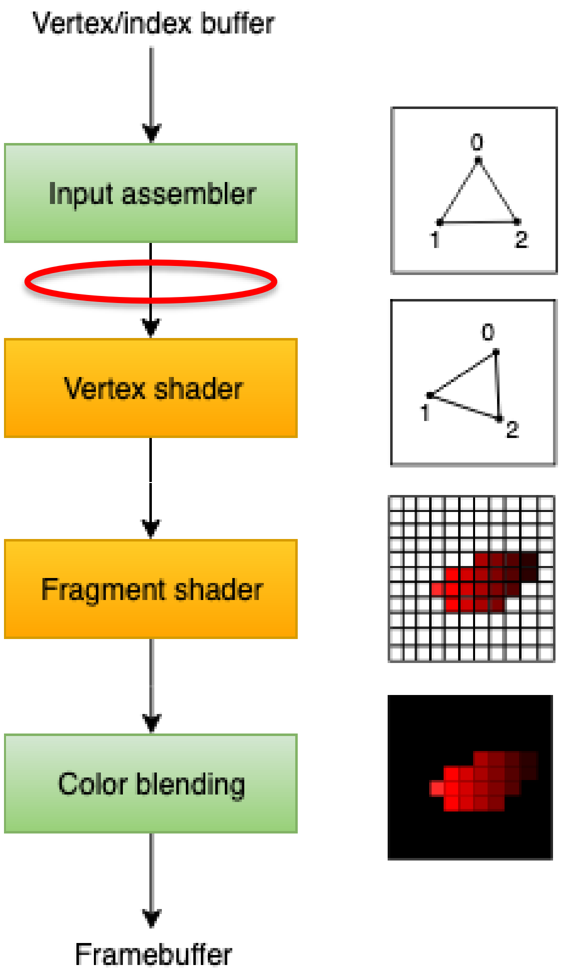
Unfortunately, this term is a little bit abused in the Vulkan documentation, and its context is not always clear, generating a lot of confusion when reading related documentation and tutorials.

Layouts

Layouts are handled in a way that allows them to be re-used whenever possible.

Vulkan also focuses on interoperability between shaders, vertices and different blocks of information, making it possible to mix-and-match whatever uses the same type of data and produces the same set of results.

Vertex data



Vertex/index buffer
Input assembler
Vertex shader
Fragment shader
Color blending
Framebuffer
0
1
2
0
1
2

Vertex attributes: refresh

In GLSL, `in` and `out` global variables are used to interface *Shaders* with the other components of the pipeline (either fixed or programmable) .

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

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

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

Vertex shader

```
#version 450

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

layout(location = 0) in vec3 inPosition;

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

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

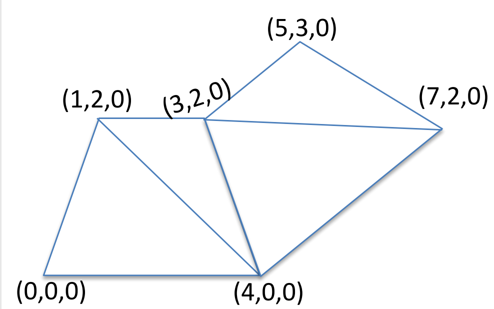
Vertex attributes: refresh

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.

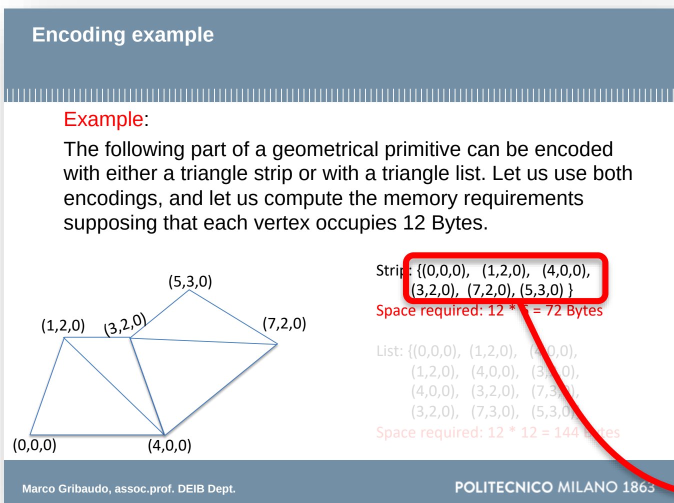


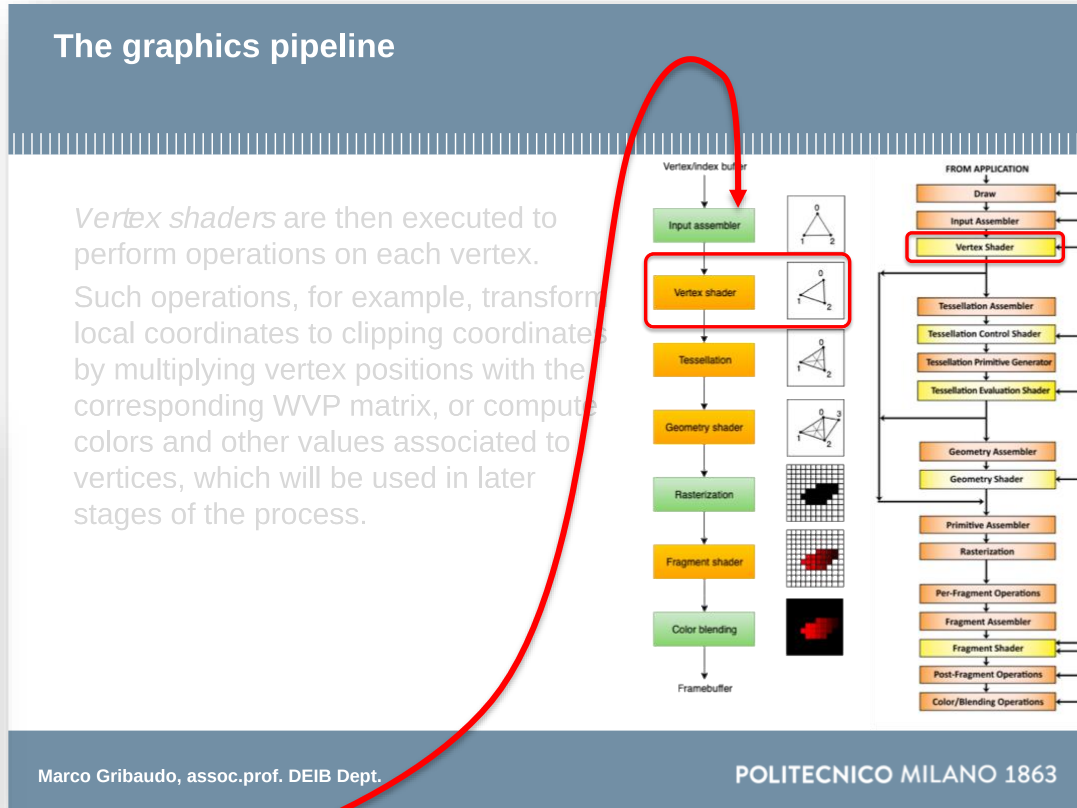
Strip: {(0,0,0), (1,2,0), (4,0,0),
(3,2,0), (7,2,0), (5,3,0) }
Space required: 12 * 6 = 72 Bytes

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.





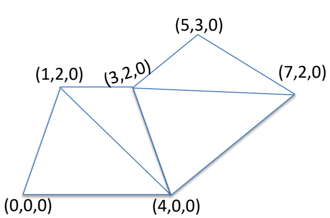
Vertex attributes: refresh

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),
(3,2,0), (7,3,0), (5,3,0)}

Space required: 12 * 12 = 144 Bytes

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;
}
```

Vertex attributes: ID

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

```
layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
    fragColor = colors[gl_VertexIndex];
}
```

From https://vulkan-tutorial.com/

Vertex attributes: user defined values

In addition, it can have an arbitrary set of user defined attributes, each one characterized by one of the supported GLSL types. Vertices may also be *empty*: i.e. have no user defined attributes.

Types [4.1]

Transparent Types

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

Floating-Point Opaque Types

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

Signed Integer Opaque Types

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

Continue ↑

Signed Integer Opaque Types (cont'd)

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

Unsigned Integer Opaque Types

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

Continue ↑

Unsigned Integer Opaque Types (cont'd)

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

Implicit Conversions

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

Aggregation of Basic Types

Arrays	float[3] foo; float foo[3]; int a [3][2]; // Structures, blocks, and structure members // can be arrays. Arrays of arrays supported.
Structures	struct *type-name* { *members* } *struct-name*[]; // optional variable declaration
Blocks	in/out/uniform *block-name* { // interface matching by block name *optionally-qualified members* } *instance-name*[]; // optional instance name, optionally an array

```
#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

In a 2D game application, for example, a set of `vec2` normalized screen coordinates can be used to denote directly the position where an element should be put.

Super Mario Wonder (Nintendo - 2023)

```
#version 450

layout(location = 0) in vec2 inPosition;

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

This is for example the case of the vertex shader used to render the text in the Assignments

Vertex attributes: user defined values

A classical 3D scene, uses at least a `vec3` element to store the positions in the 3D local space.

Star Wars (Atari - 1983)

```
#version 450

layout(set = 0, binding = 0) uniform
    UniformBufferObject {
     mat4 mMat;
     mat4 mvpMat;
     mat4 nMat;
} 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 mMat;
     mat4 mvpMat;
     mat4 nMat;
} 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.

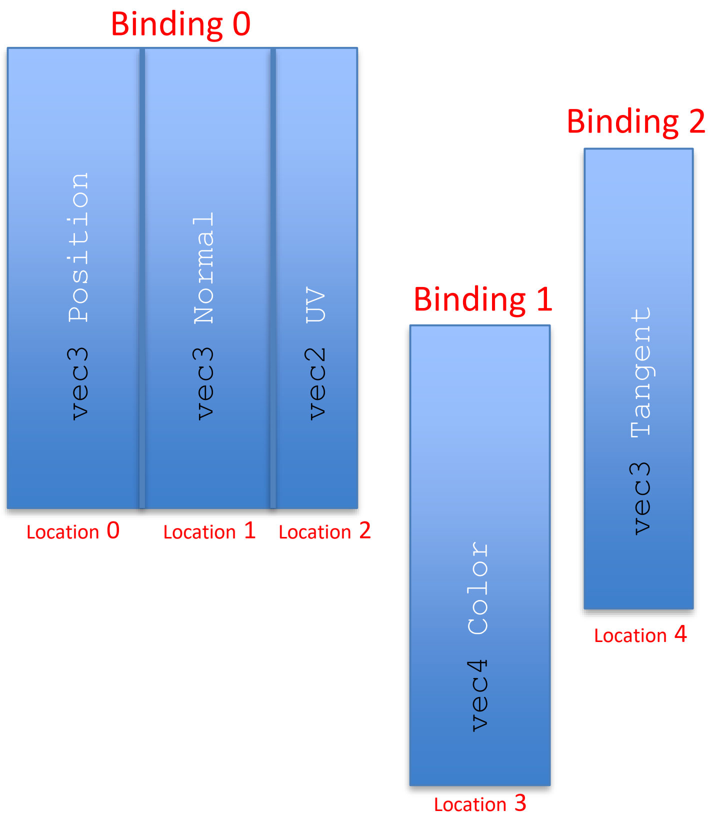
Different meshes might be characterized by different vertices formats (however this will require the creation of different pipelines).

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;
}
```

gribaudo › Documents › Corsi › 2022 › CG › Assignments › A00 - Compile and run › WK › shaders › PBRShader.vert

Characters: 732 · Words: 90

Vertex attributes: Vertex Input Descriptors

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

Its fields contain the binding *id* and the size of the object in bytes.

The `inputRate` field can be used for *instanced rendering*, that will be briefly introduced in a following lesson.

```
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

Please note however that not all formats are supported on all platforms.
The four above should work almost everywhere!

Vertex attributes: Vertex Input Descriptors

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

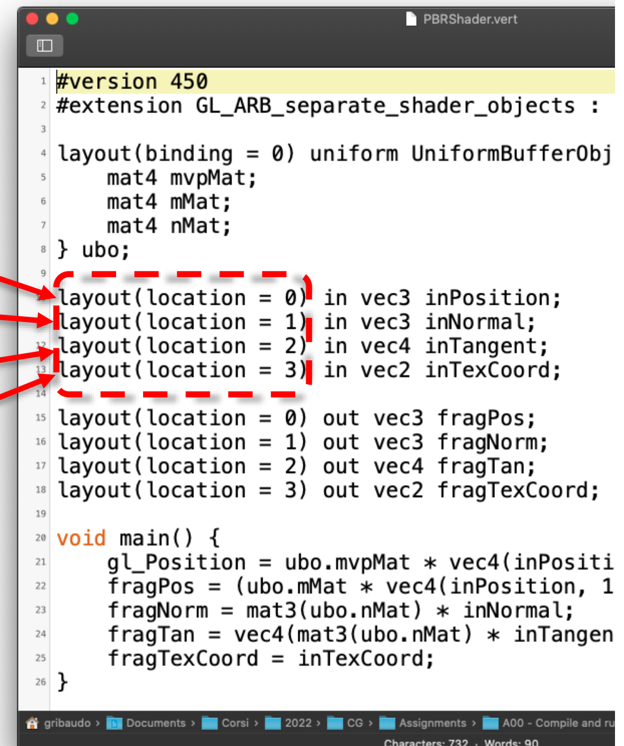
This can be computed using the C++ `offsetof()` macro.

```
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);
```

Vertex attributes: Vertex Input Descriptors

The vertex shade use the layout(location = …) directive to join global variables with the corresponding vertices attributes.

```
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);
```



PBRShader.vert

```
#version 450
#extension GL_ARB_separate_shader_objects :

layout(binding = 0) uniform UniformBufferObj
    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(inPositi
    fragPos = (ubo.mMat * vec4(inPosition, 1
    fragNorm = mat3(ubo.nMat) * inNormal;
    fragTan = vec4(mat3(ubo.nMat) * inTangen
    fragTexCoord = inTexCoord;
}
```

gribaudo › Documents › Corsi › 2022 › CG › Assignments › A00 - Compile and ru

Characters: 732 · Words: 90

Vertex Layouts in Starter.hpp

The vertex attributes must then be defined in three steps:

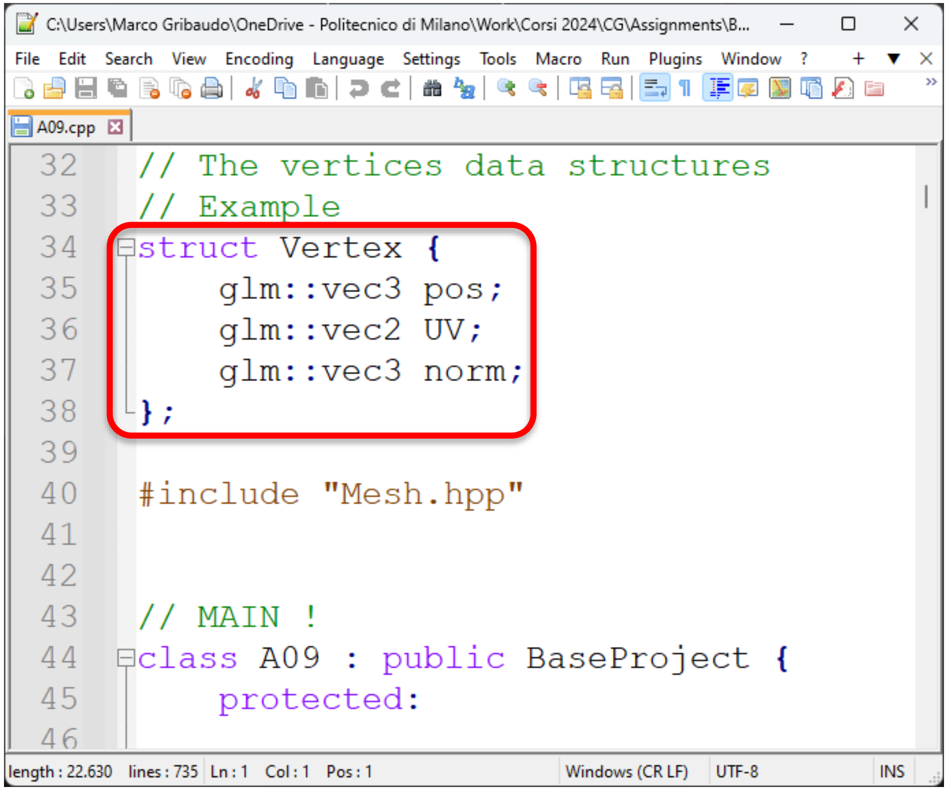
- Create a CPP data structure (CPP code)
- Create the vertex description Vulkan structure (CPP code)
- Define the corresponding in variable with the proper locations in the vertex shader (GLSL code).

Vertex Layouts in Starter.hpp

I - Create the CPP data structure (CPP code)

The CPP structure for a vertex is defined in the usual way, generally as a global statement.

This example shows the vertex used in Assignment A09, which uses position, UV coordinates, and normal vector direction.



Vertex Layouts in Starter.hpp

II - Create the vertex description Vulkan structure (CPP code)

`Starter.hpp` has a structure called `VertexDescriptor`, which is used to handle the creation of the vertex layout in Vulkan.

```cpp
struct VertexBindingDescriptorElement {
    uint32_t binding;
    uint32_t stride;
    VkVertexInputRate inputRate;
};

enum VertexDescriptorElementUsage {POSITION, NORMAL, UV, COLOR, TANGENT, OTHER};

struct VertexDescriptorElement {
    uint32_t binding;
    uint32_t location;
    VkFormat format;
    uint32_t offset;
    uint32_t size;
    VertexDescriptorElementUsage usage;
};

struct VertexComponent {
    bool hasIt;
    uint32_t offset;
};

struct VertexDescriptor {
    BaseProject *BP;

    VertexComponent Position;
    VertexComponent Normal;
    VertexComponent UV;
    VertexComponent Color;
    VertexComponent Tangent;

    std::vector<VertexBindingDescriptorElement> Bindings;
    std::vector<VertexDescriptorElement> Layout;

    void init(BaseProject *bp, std::vector<VertexBindingDescriptorElement> B,
    std::vector<VertexDescriptorElement> E);
    void cleanup();

    std::vector<VkVertexInputBindingDescription> getBindingDescription();
    std::vector<VkVertexInputAttributeDescription>
                        getAttributeDescriptions();
};
```

Vertex Layouts in Starter.hpp

The init() method requires both the definitions of the *bindings* and of the *locations*.

C:\Users\Marco Gribaudo\OneDrive - Politecnico di Milano\Work\Corsi 2024\CG\Assignments\Backup\A09\A09.cpp - Notepad++

A09.cpp | Starter.hpp

```
    // Vertex formats
    VertexDescriptor VD;

        // Vertex descriptors
        VD.init(this, {
                    {0, sizeof(Vertex), VK_VERTEX_INPUT_RATE_VERTEX}
                }, {
                    {0, 0, VK_FORMAT_R32G32B32_SFLOAT, offsetof(Vertex, pos),
                           sizeof(glm::vec3), POSITION},
                    {0, 1, VK_FORMAT_R32G32_SFLOAT, offsetof(Vertex, UV),
                           sizeof(glm::vec2), UV},
                    {0, 2, VK_FORMAT_R32G32B32_SFLOAT, offsetof(Vertex, norm),
                           sizeof(glm::vec3), NORMAL}
                });
```

Bindings

Locations

Vertex Layouts in Starter.hpp

The bindings contain:

1. The ID of the binding (starting from 0)
2. The size of each vertex in Bytes, inside this memory block (the *stride*). This is usually computed using the CPP `sizeof()` macro.
3. The Vulkan constant specifying the *input rate*, as previously introduced.

```
VD.init(this, {
        {0, sizeof(Vertex), VK_VERTEX_INPUT_RATE_VERTEX}
    }, {
```

```
struct VertexBindingDescriptorElement {
    uint32_t binding;
    uint32_t stride;
    VkVertexInputRate inputRate;
};
```

Vertex Layouts in Starter.hpp

```
struct Vertex {
    glm::vec3 pos;
    glm::vec2 UV;
    glm::vec3 norm;
};
```

```
, {
 {0, 0, VK_FORMAT_R32G32B32_SFLOAT, offsetof(Vertex, pos),
        sizeof(glm::vec3), POSITION},
 {0, 1, VK_FORMAT_R32G32_SFLOAT, offsetof(Vertex, UV),
        sizeof(glm::vec2), UV},
 {0, 2, VK_FORMAT_R32G32B32_SFLOAT, offsetof(Vertex, norm),
        sizeof(glm::vec3), NORMAL}
});
```

For each field of the vertex structure, the locations contains contain:

1. The ID of the binding (starting from 0)
2. The ID of the location (starging from 0)
3. The Vulkan constant specifying its *format*, as previously introduced.

```
struct VertexDescriptorElement {
    uint32_t binding;
    uint32_t location;
    VkFormat format;
    uint32_t offset;
    uint32_t size;
    VertexDescriptorElementUsage usage;
};
```

Vertex Layouts in Starter.hpp

```
struct Vertex {
    glm::vec3 pos;
    glm::vec2 UV;
    glm::vec3 norm;
};
```

```
, {
  {0, 0, VK_FORMAT_R32G32B32_SFLOAT, offsetof(Vertex, pos),
         sizeof(glm::vec3), POSITION},
  {0, 1, VK_FORMAT_R32G32_SFLOAT, offsetof(Vertex, UV),
         sizeof(glm::vec2), UV},
  {0, 2, VK_FORMAT_R32G32B32_SFLOAT, offsetof(Vertex, norm),
         sizeof(glm::vec3), NORMAL}
);
```

4. The offset in byte of the property in the memory area of each vertex. Usually this is computed using the C++ `offsetof()` macro.
5. The size in bytes of the property. Usually this is obtained with the `sizeof()` macro.

```
struct VertexDescriptorElement {
    uint32_t binding;
    uint32_t location;
    VkFormat format;
    uint32_t offset;
    uint32_t size;
    VertexDescriptorElementUsage usage;
};
```

Vertex Layouts in Starter.hpp

6. A special enumeration constant that tells if this element corresponds to a standard feature of a vertex. This value is not used by Vulkan, but from the 3D model loading procedure of `Starter.hpp`. If the vertices of the model in a file being loaded has the corresponding property, the loader procedure will put their value in the corresponding place.

```
, {
  {0, 0, VK_FORMAT_R32G32B32_SFLOAT, offsetof(Vertex, pos),
         sizeof(glm::vec3), POSITION},
  {0, 1, VK_FORMAT_R32G32_SFLOAT, offsetof(Vertex, UV),
         sizeof(glm::vec2), UV},
  {0, 2, VK_FORMAT_R32G32B32_SFLOAT, offsetof(Vertex, norm),
         sizeof(glm::vec3), NORMAL}
});
```

```
struct VertexDescriptorElement {
    uint32_t binding;
    uint32_t location;
    VkFormat format;
    uint32_t offset;
    uint32_t size;
    VertexDescriptorElementUsage usage;
};
```

Vertex Layouts in Starter.hpp

For the moment, the known properties are:

- POSITION in a 3D space
- NORMAL vector direction
- UV coordinates
- COLOR of a vertex
- TANGENT to the vertex

If a vertex uses non-standard features, they can be generically assigned value OTHER.

```
enum VertexDescriptorElementUsage {POSITION, NORMAL, UV, COLOR, TANGENT, OTHER};
```

Vertex Layouts in Starter.hpp

III - Define the corresponding in variables with the proper locations in the vertex shader.

Finally, for each property, a global `in` variable, with the corresponding type and location should be added in the vertex shader code.

```cpp
struct Vertex {
    glm::vec3 pos;
    glm::vec2 UV;
    glm::vec3 norm;
};
```

```cpp
, {
 {0, 0, VK_FORMAT_R32G32B32_SFLOAT, offsetof(Vertex, pos),
        sizeof(glm::vec3), POSITION},
 {0, 1, VK_FORMAT_R32G32_SFLOAT, offsetof(Vertex, UV),
        sizeof(glm::vec2), UV},
 {0, 2, VK_FORMAT_R32G32B32_SFLOAT, offsetof(Vertex, norm),
        sizeof(glm::vec3), NORMAL}
);
```

C:\Users\Marco Gribaudo\OneDrive - Politecnico di Milano\Work\Corsi 2024\CG\Assignments\...

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

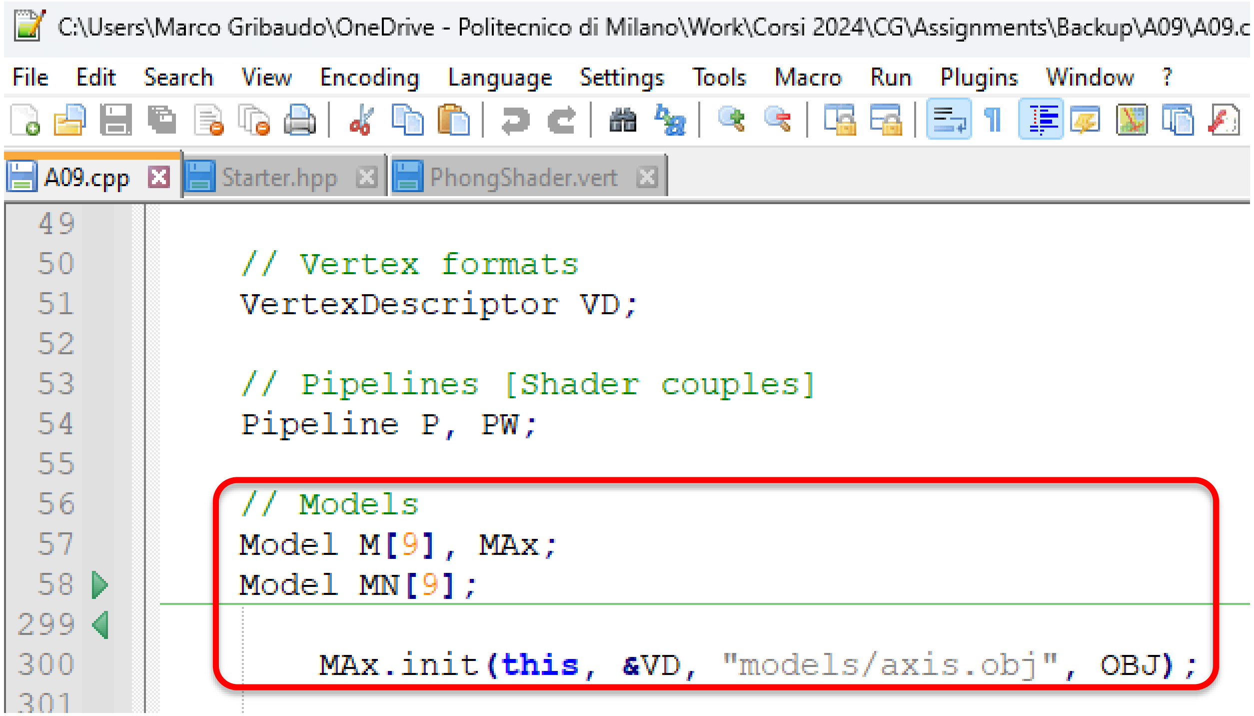
A09.cpp | Starter.hpp | PhongShader.vert

```glsl
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 0) uniform UniformBufferObject {
    mat4 mvpMat;
    mat4 mMat;
    mat4 nMat;
    vec4 color;
} ubo;

layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec2 inUV;
layout(location = 2) in vec3 inNorm;

layout(location = 0) out vec3 fragPos;
layout(location = 1) out vec3 outNorm;

void main() {
    gl_Position = ubo.mvpMat * vec4(inPosition, 1.0);
    fragPos = (ubo.mMat * vec4(inPosition, 1.0)).xyz;
    outNorm = (ubo.nMat * vec4(inNorm, 0.0)).xyz;
}
```

length : 531 lines : 22 Ln : 1 Col : 1 Pos : 1 Unix (LF) UTF-8 INS

Loading models

`Starter.cpp` also includes a data structure called `Model` to include meshes, and a procedure to load them from an external file.



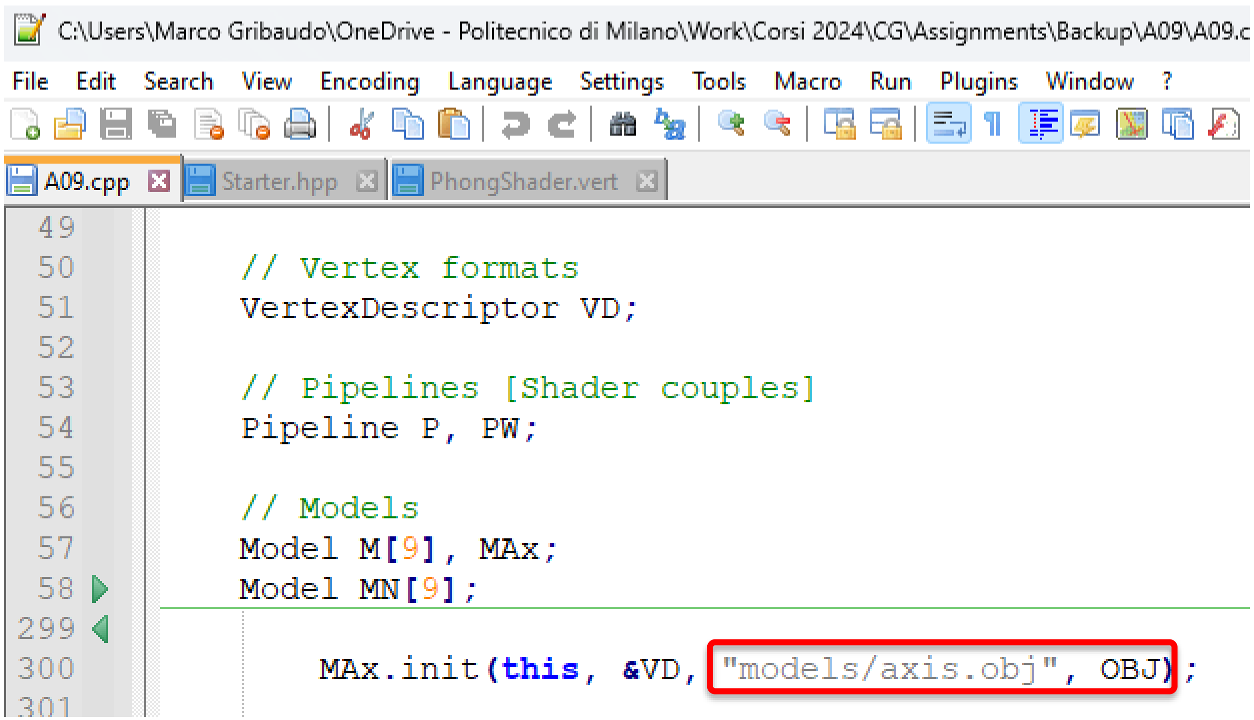
Loading models

The `init()` procedure, requires the definition of the format of the vertices to be created by means of a point to a `VertexDescriptor`.

```
// Vertex formats
VertexDescriptor VD;

// Pipelines [Shader couples]
Pipeline P, PW;

// Models
Model M[9], MAx;
Model MN[9];

    MAx.init(this, &VD, "models/axis.obj", OBJ);
```

Loading models

It requires the name of the file to be loaded, and a constant defining its format.

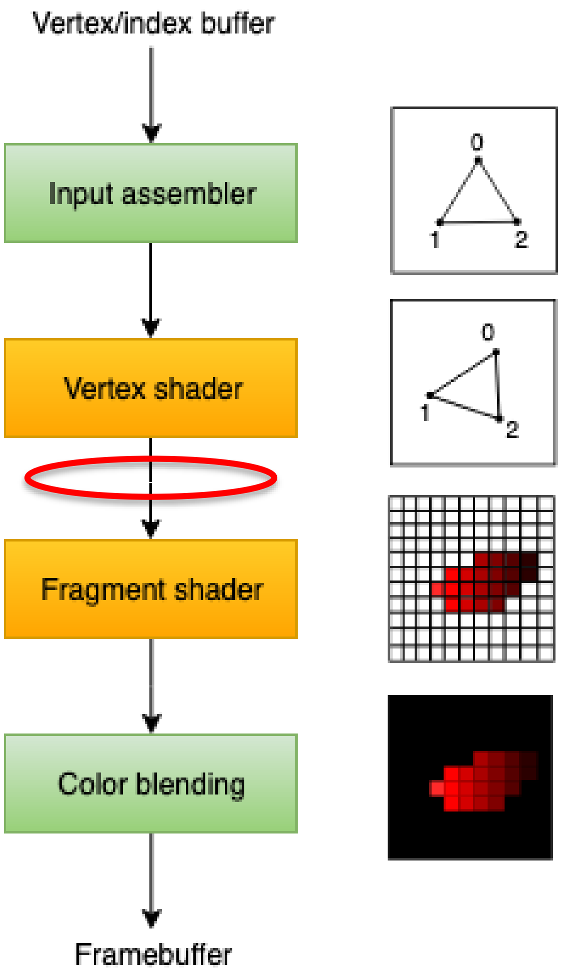


Loading models

Currently, only two formats are supported:

FORMAT	Extensions	Supported features
OBJ	.obj	Position, Normal vector, a single UV channel, Vertex color
GLTF	.gltf	Position, Normal vector, multiple UV channels, Vertex color, Tangent vector

Communication between Vertex and Fragment shaders



Communication between Vertex and Fragment shaders

Only the vertex shader can access attributes.

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

As for the vertices attributes, The `in` and `out` variables, used for the shaders communication, are 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

Please note that only slot numbers are used. The corresponding global variable can change name between the shaders.

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 c_r;
layout(location = 1) in float c_i;

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 + c_r;
		m_img = 2.0 * m_real * m_img + c_i;
		m_real = temp;
	}
…
}
```

Communication between Vertex and Fragment shaders

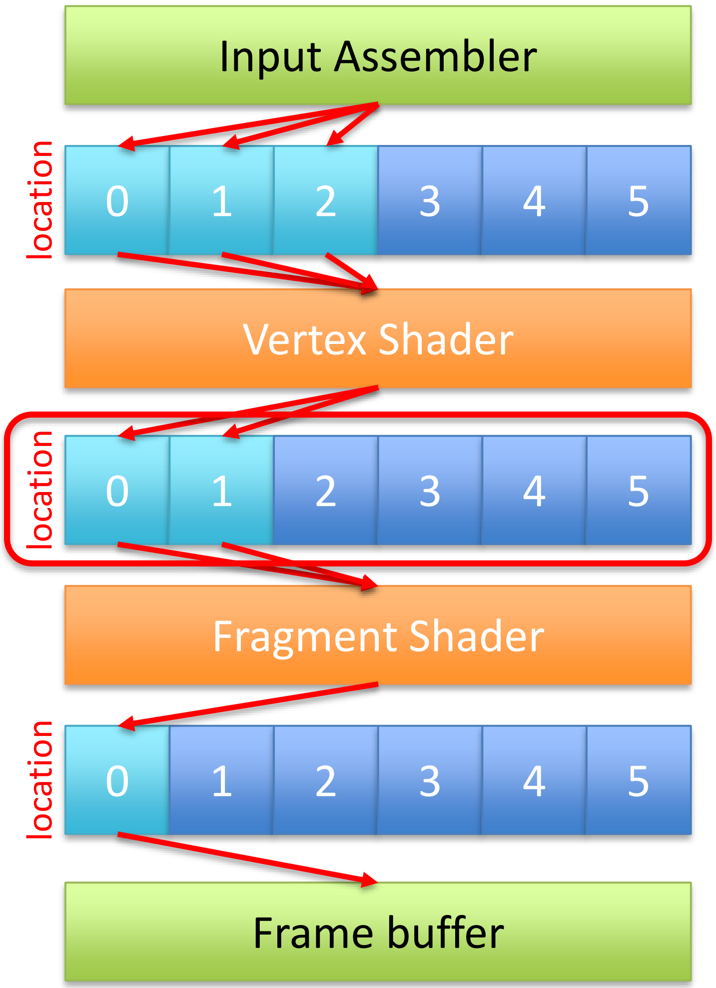
The slots used by the *Input Assembler*, which will be available inside `in` variables of the *Vertex Shader*, are configured in the pipeline creation.

The configuration of the pipeline, also defines the `out` variables that *the Fragment Shader* will write. Usually there is just one: the final color of the pixel (fragment). However, more advanced applications can compute more values in the Fragment Shader.

Communication between Vertex and Fragment shaders

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

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*.



Interpolation between Vertex and Fragment shader

The default interpolation between Vertex and Fragment shader is via *Perspective Correct* interpolations.

However it can be controlled with the `flat` and `noperspective` directives before the `in` and `out` variables.

Vertex shader

```
layout(location = 0) out vec3 fragPos;
layout(location = 1) out vec3 fragNorm;
layout(location = 2) noperspective out vec2 fragUV;
```

Fragment shader

```
layout(location = 0) in vec3 fragPos;
layout(location = 1) in vec3 fragNorm;
layout(location = 2) noperspective in vec2 fragUV;
```

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)