POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

DE
IB

2024

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2024

Computer Graphics

- Pipelines

Pipelines

In order to transform a set of data representing a mesh to an image on screen, a sequence of operations needs to be performed.

This sequence of operations is called a *Pipeline*, since it resembles the classical pattern defined to exploit the parallel execution of different steps on a set of data.

Although we will focus on *Vulkan*, these concepts are very similar in other environments such as *OpenGL*, *Metal* or *Microsoft DirectX 12*.

Pipeline: the general concept

A pipeline is a structure where a stream of data needs to be processed into several steps. While one element is performing the second step, a new one can start in parallel with the first.

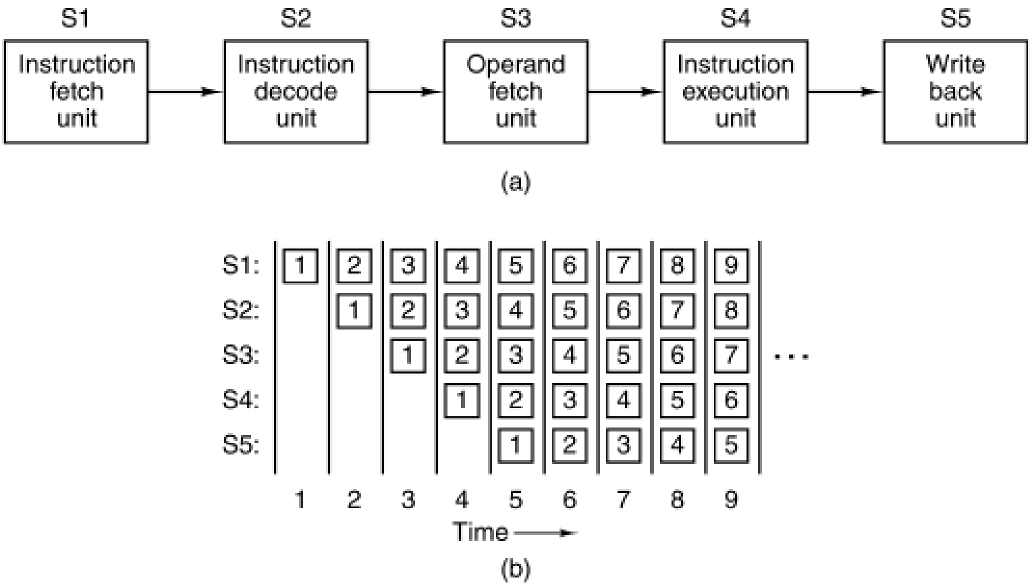
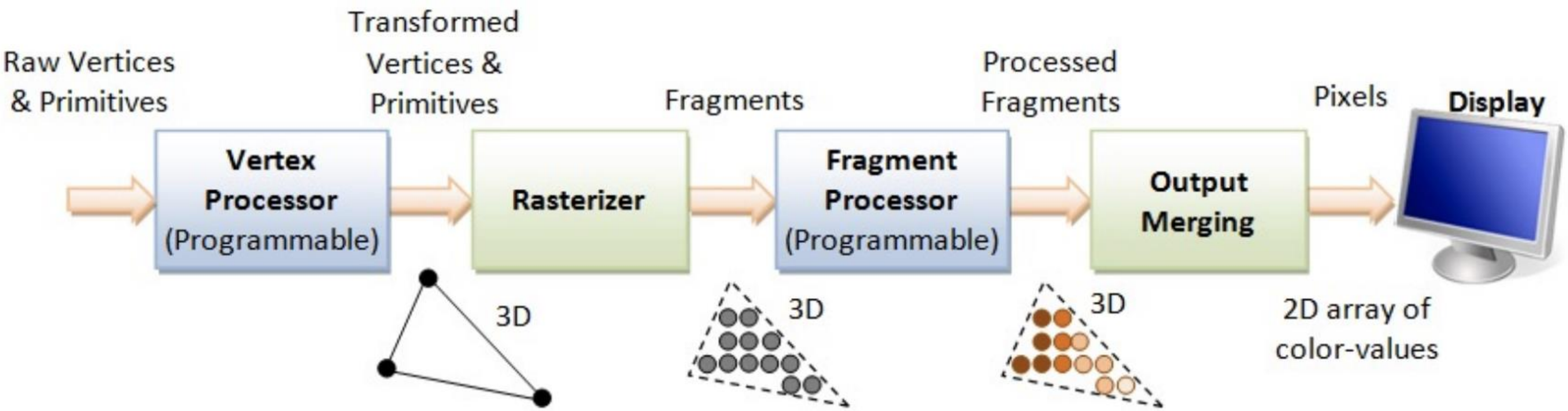


Figure 2-4. (a) A five-stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated.

Pipeline: the general concept

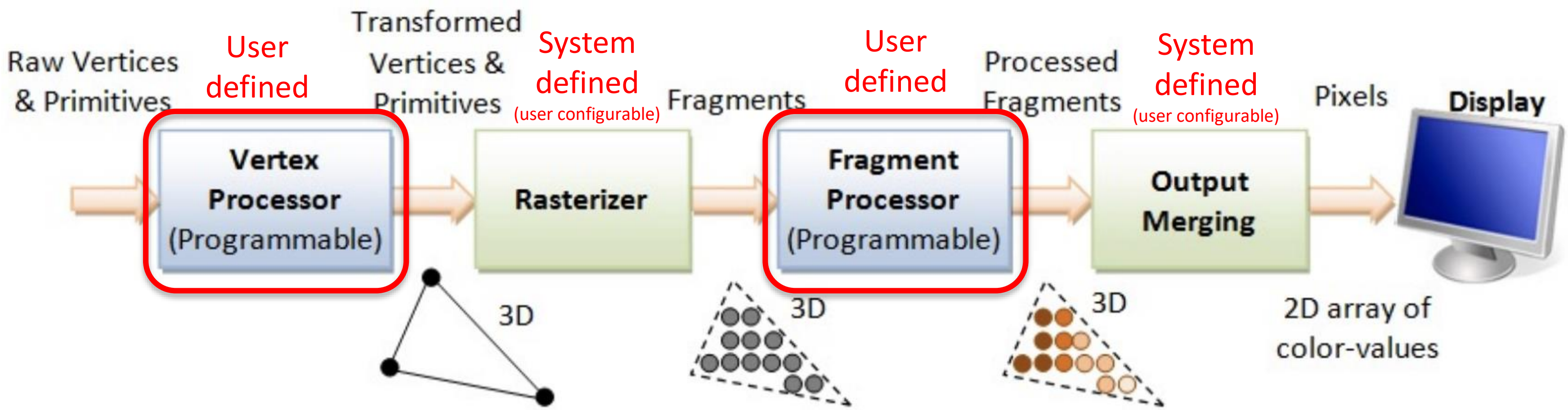
In *Vulkan*, and in CG in general, the process of creating an image on screen starting from the primitive description is accomplished through a set of steps than can be organized as a pipeline.



3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

Pipeline: the general concept

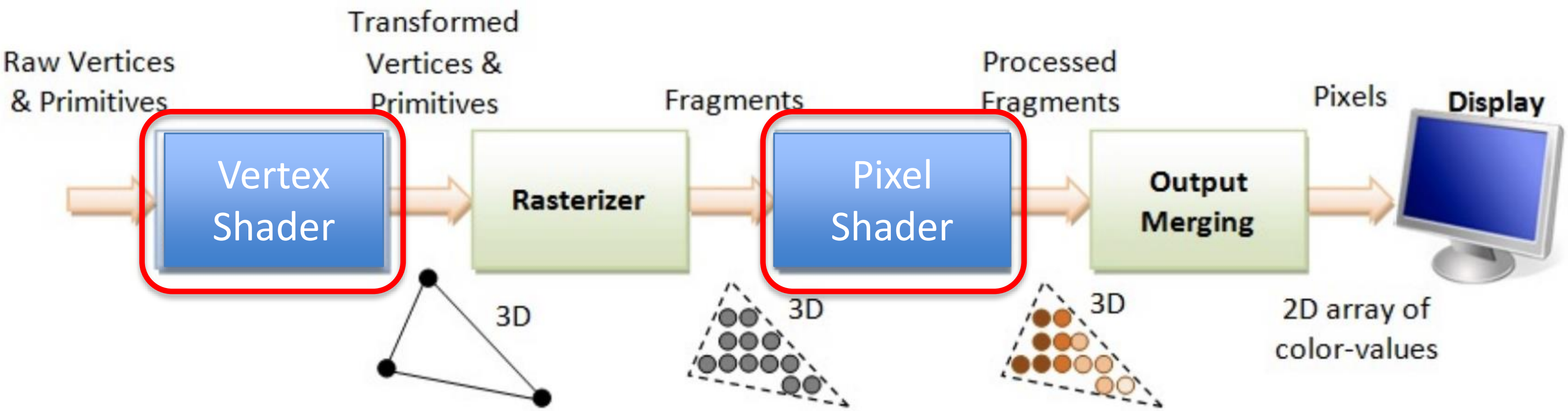
The actions taken in each stage of the pipeline can be either fixed and defined by the system (Vulkan, in our case), or programmed by the user.



3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

Pipeline: the general concept

For historical reasons, algorithms running in the programmable stages of the pipeline are called *Shaders*. In the following we will learn how to write shaders and control the graphics pipeline.



3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (*x*, *y*, *z*) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

Pipeline types

Different types of pipelines, with specific purposes, have been defined to handle the generation of 3D images.

Each pipeline type has its own set of fixed functions, input and output description, and programmable stages.

Creating a pipeline requires configuring all the parameters needed by its fixed functions, and connect it with the shaders that perform the user defined parts.

Pipeline types

The latest Vulkan versions supports up to four types of pipelines:

- Graphic pipelines
- Ray-tracing pipelines
- Mesh Shading pipelines
- Compute pipelines

The first three are meant to provide rendering of 3D meshes, while the last one is used for general computation purposes (i.e. GPGPU). *Ray-tracing* has been standardized only recently, and *Mesh Shading* is still in a very early stage: both of them might still have a limited support.

Rendering techniques

Several techniques have been introduced to approximate the rendering equation. We will briefly mention:

- *Scan-line rendering*
- *Ray casting*
- *Ray tracing*
- *Radiosity*
- *Montecarlo techniques*

These techniques are closely related to the types of pipelines that support them.

Scan-line rendering

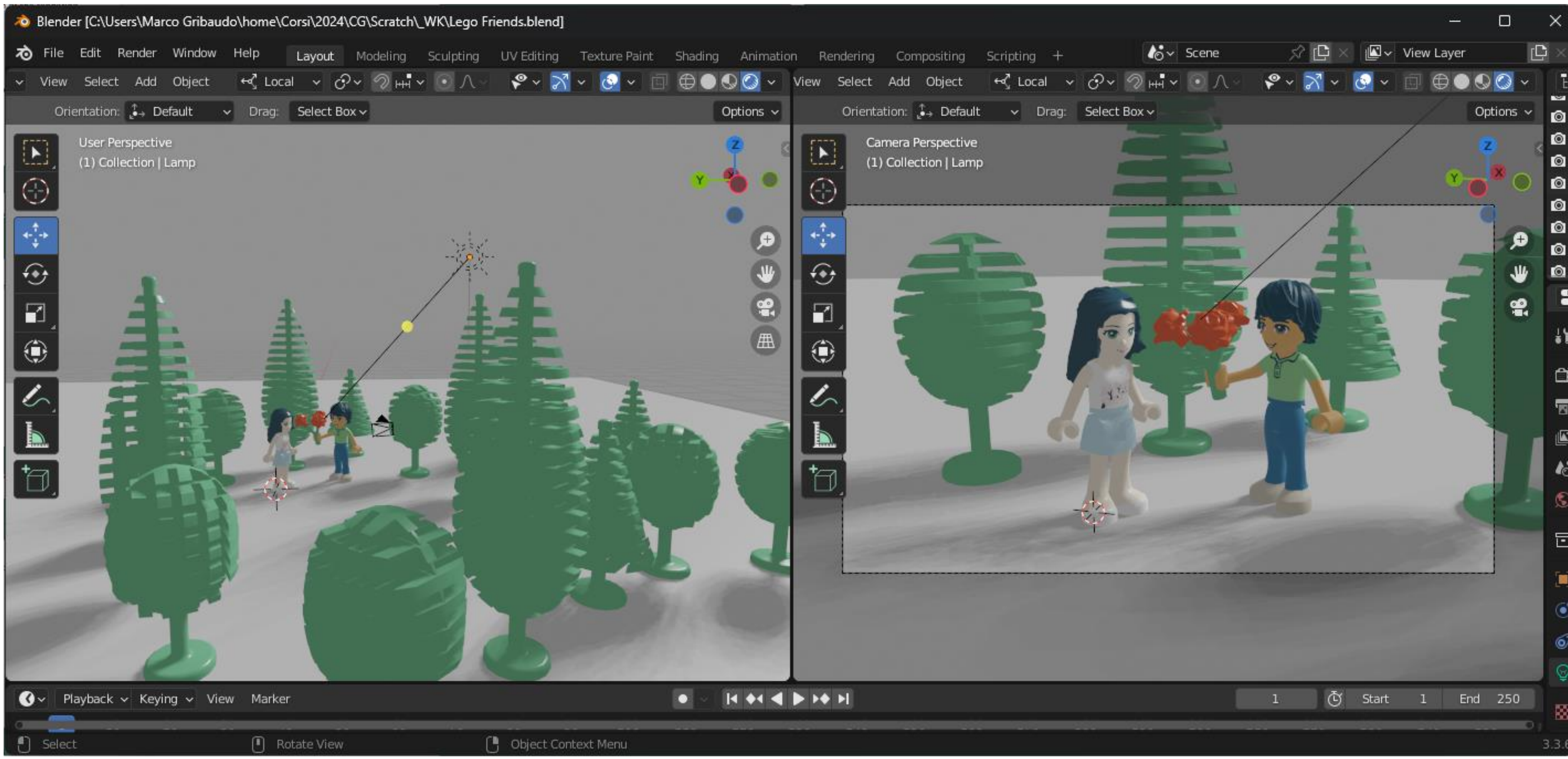
Scan-line rendering is the simplest approximation of the rendering equations.

It considers light sources and objects separately: the scene has a set of objects and a set of light sources.

No projected shadows or indirect lighting are produced.

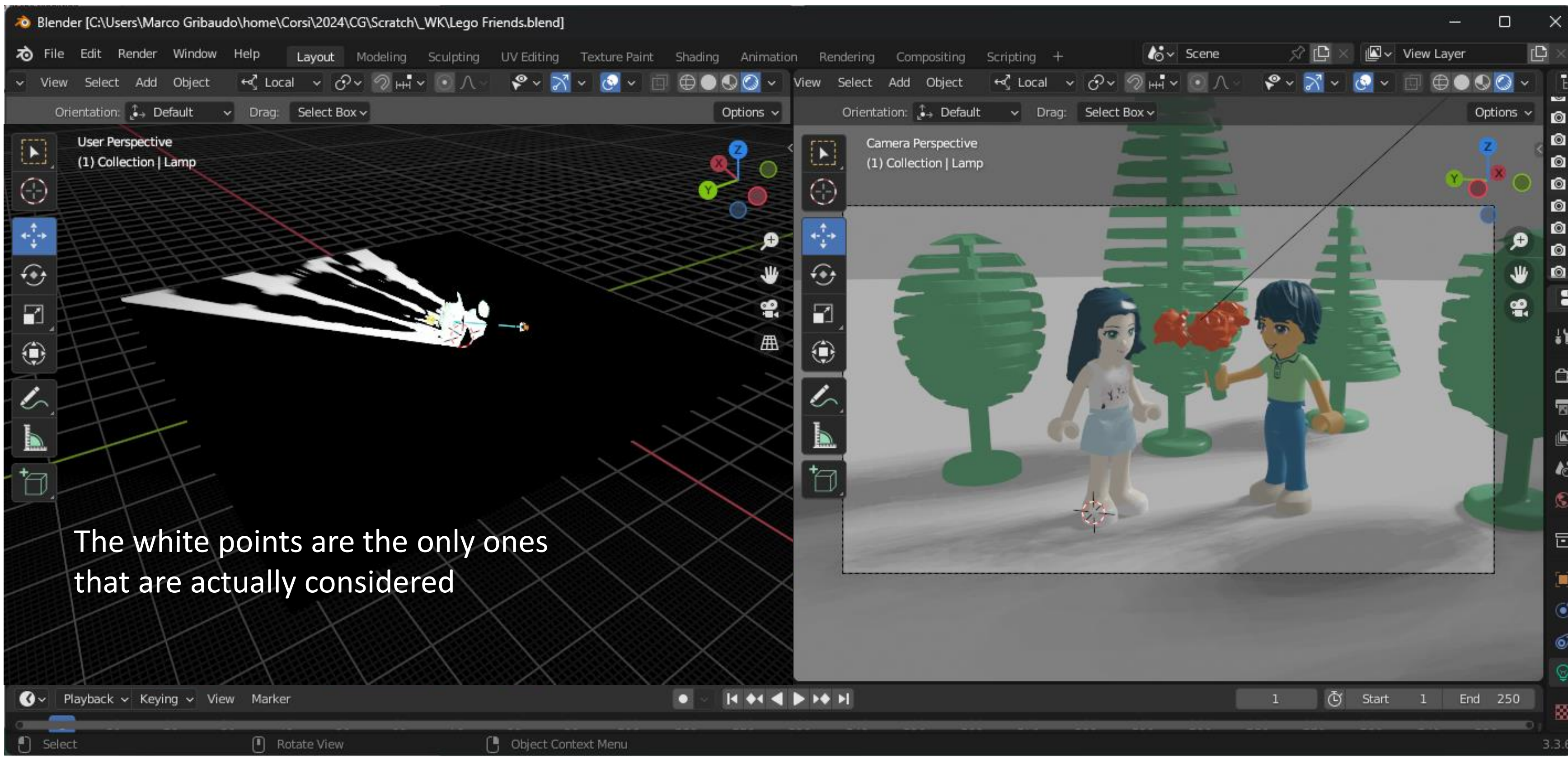
Scan-line rendering

Instead of computing the complete solution of the rendering equations…



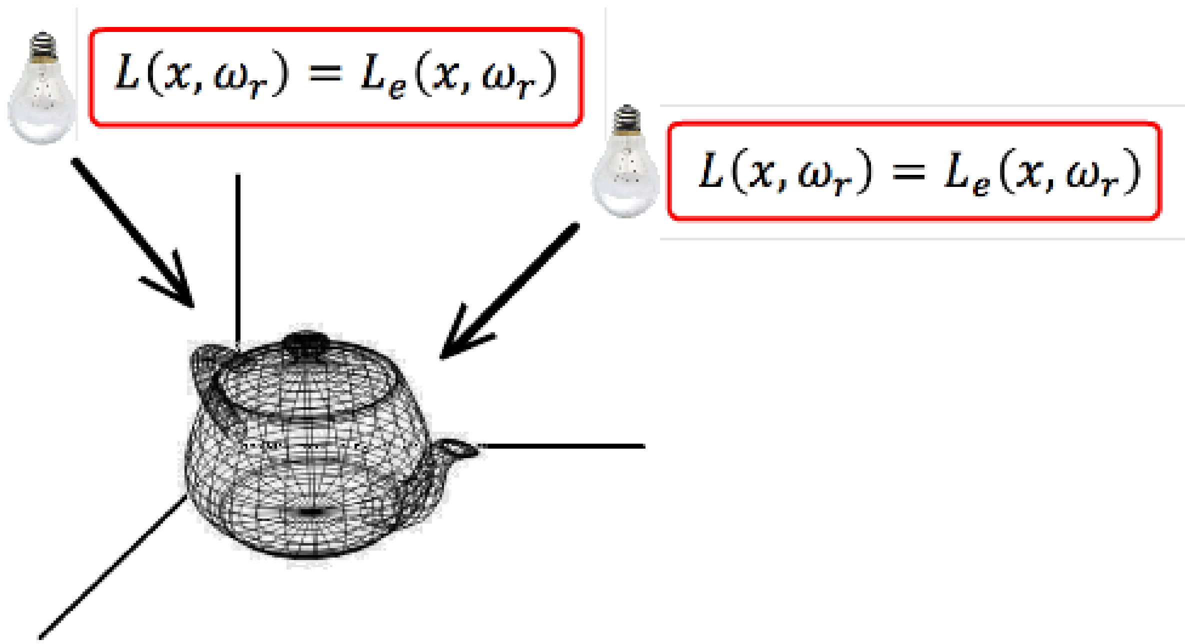
Scan-line rendering

The technique focuses only on the points currently visible by the camera.



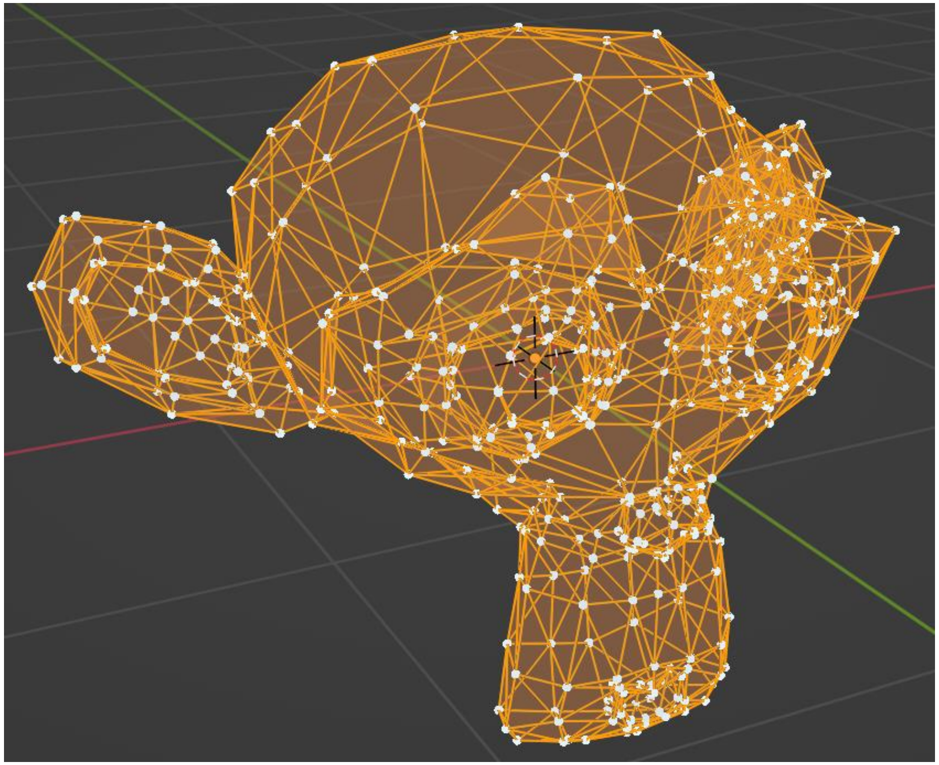
Scan-line rendering

Lights are characterized by having only the emission term in the rendering equation: this however can vary in position and direction.



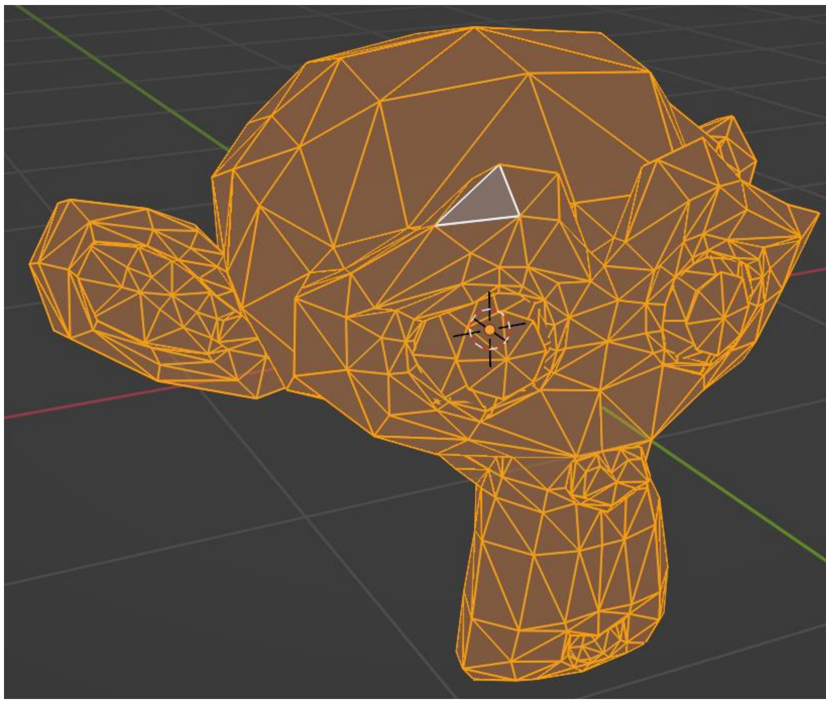
Scan-line rendering

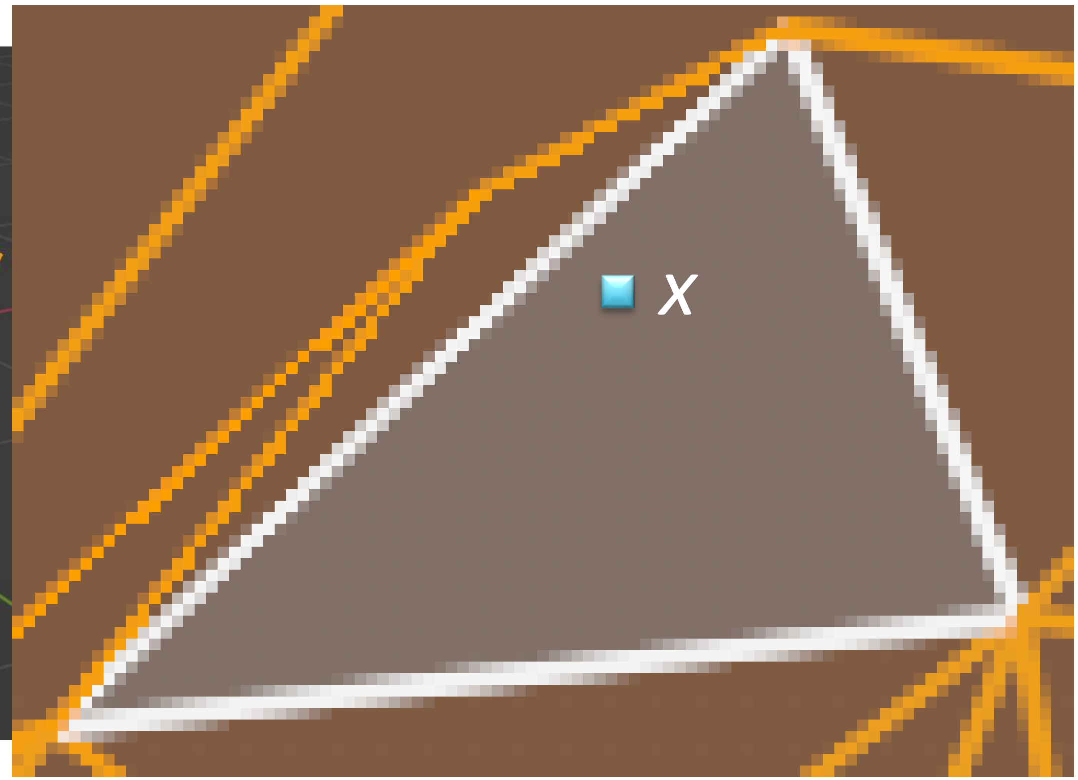
Points that define the vertices of the triangles belonging to a mesh are projected on screen, finding the corresponding hardware coordinates.



Scan-line rendering

All pixels belonging to a triangle are then enumerated. Each pixel becomes a point *x* for which the rendering equation is solved.





Scan-line rendering

Objects can only reflect light. They might emit some light, but they cannot illuminate other objects.

Inter-reflection between objects is not considered: the integral becomes a summation over all the light sources.

The geometric term is generally included in the BRDF:

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

With $f_{r'}(\ldots) = f_r(\ldots) G(\ldots)$

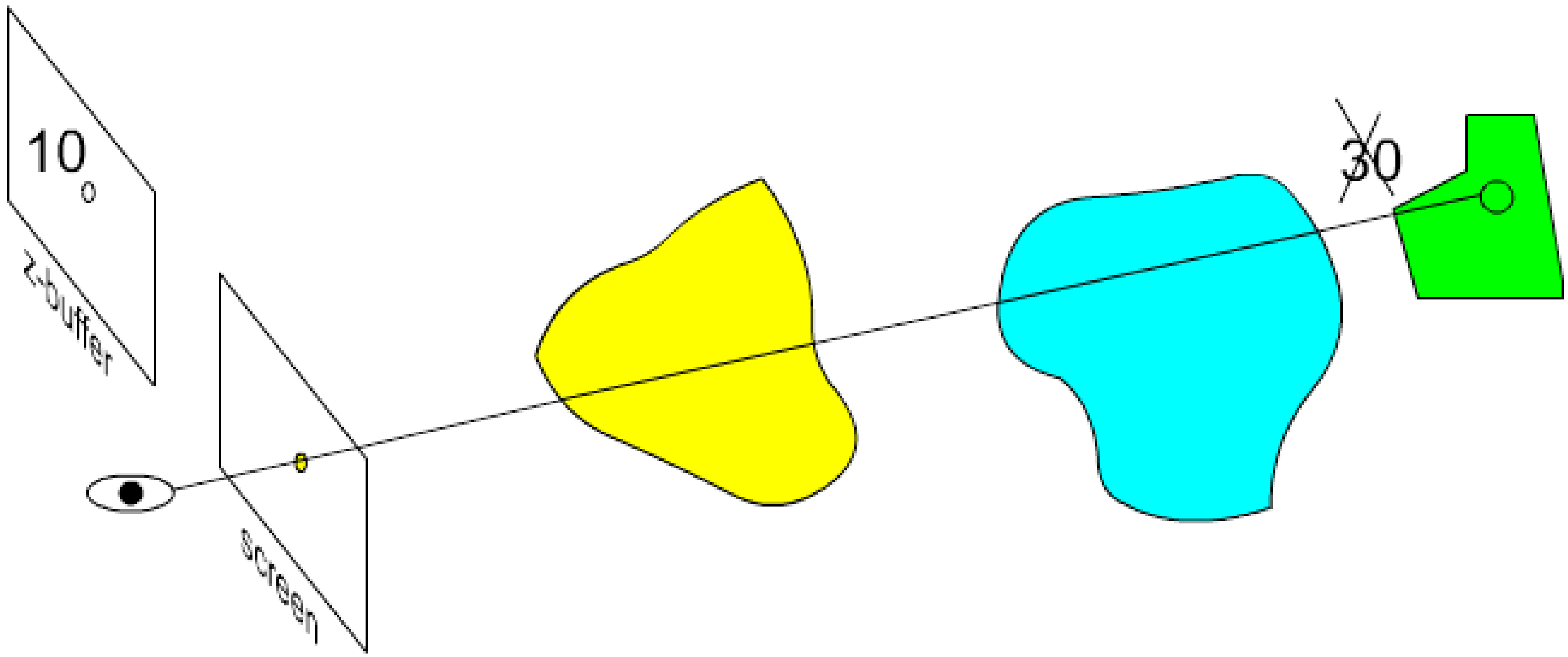
This is the emission term

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

The summation is over all the direct lights *l* in the scene

Scan-line rendering

Visibility is considered only with respect to the observer, by means of the *z-buffer* algorithm.



Scan-line rendering

Since term *V()* of the rendering equation is not considered for lights, scan-line rendering does not generate projected shadows.

Neither it does include light emitted by other objects in the scene, and thus it does not produces reflection, refraction or indirect illumination.

However, it considers different types of BRDF functions that can describe the materials composing the objects in a detailed way.

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

The BRDF is used to reproduce materials

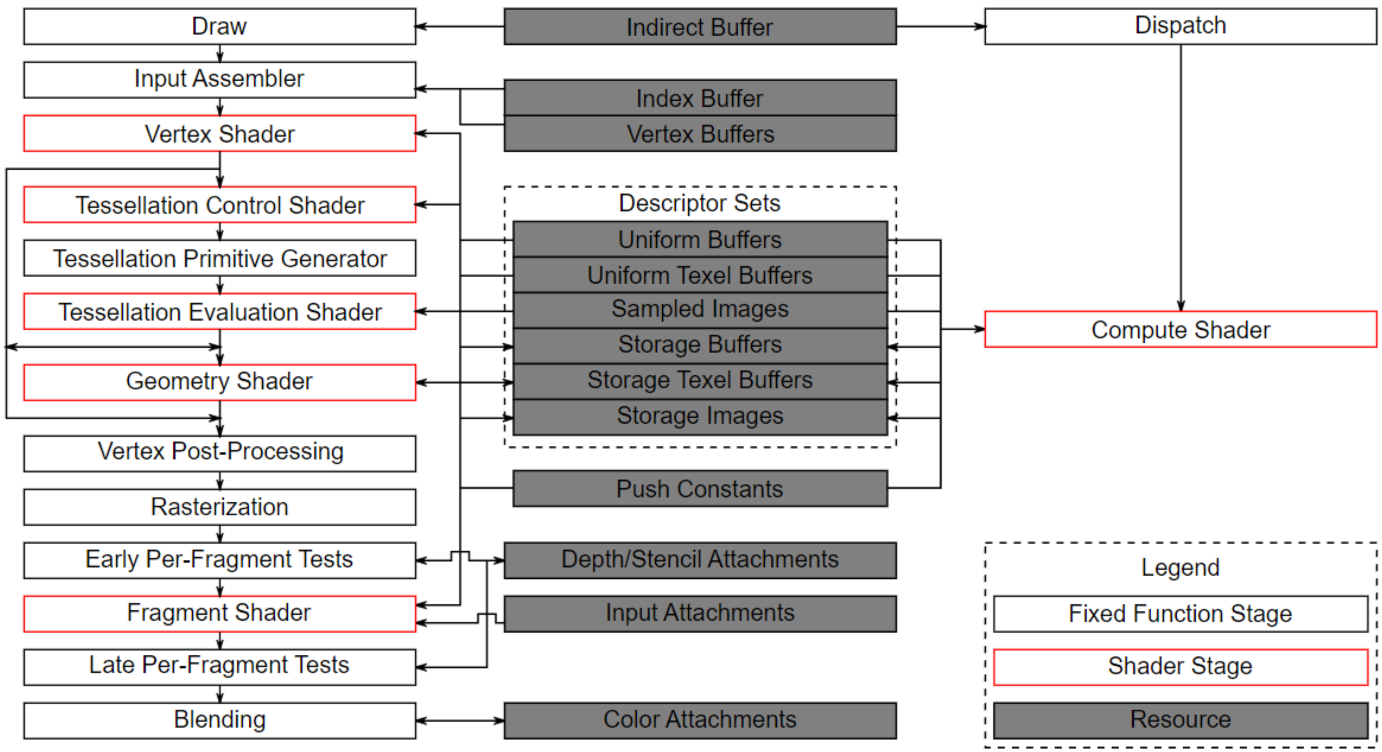
The graphics pipeline

Vulkan supports scan-line rendering, with a specific type of pipeline, called “*the graphics pipeline*”.

Let’s see in detail which are its stages, and what is their function.

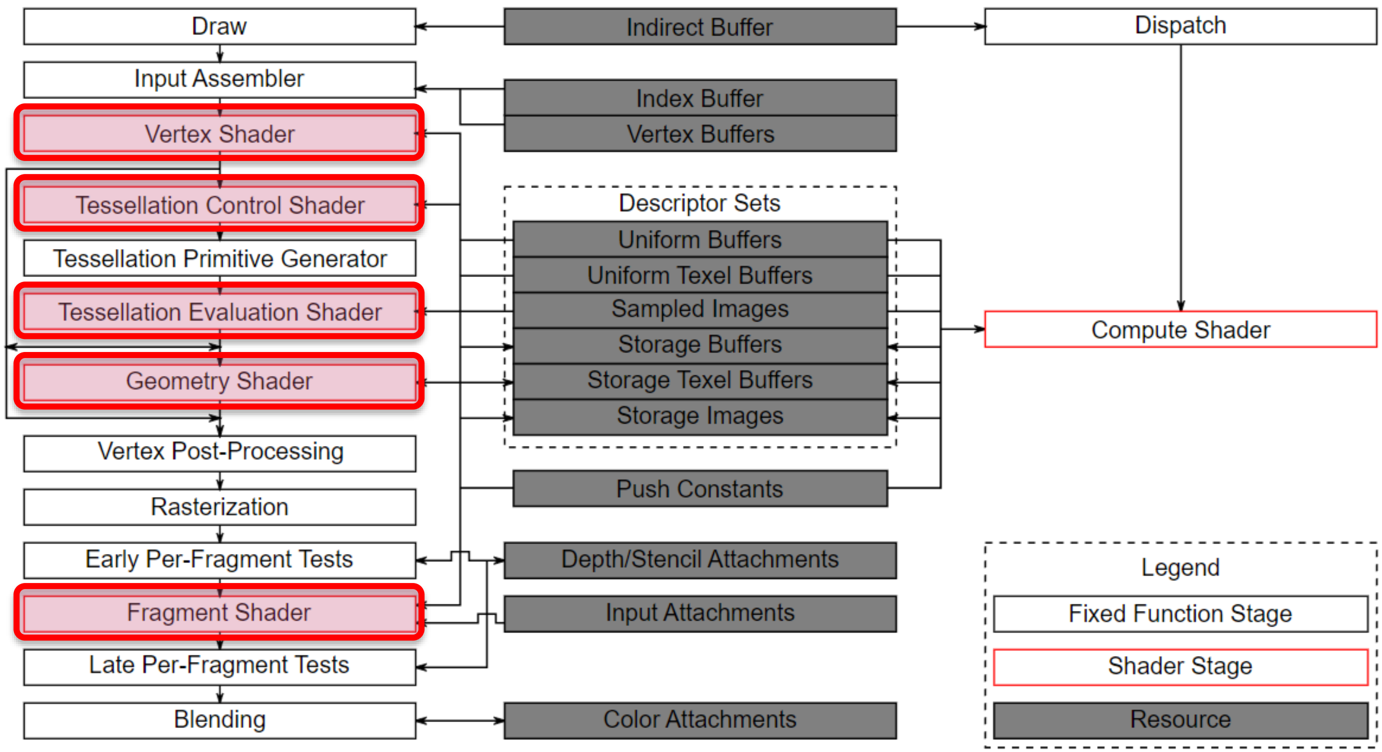
The graphics pipeline

According to the Vulkan documentation, the graphics pipeline has the following structure:



The graphics pipeline

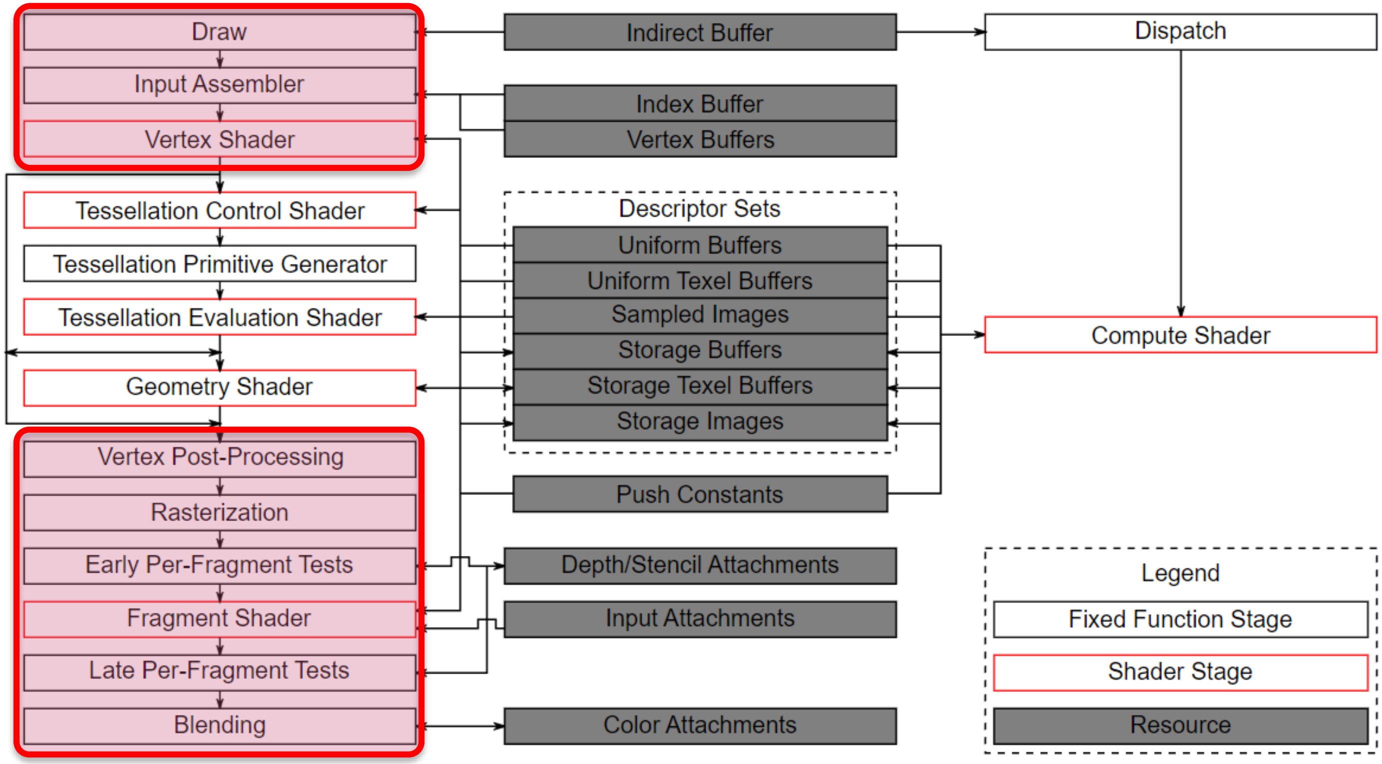
Up to five different types of shaders can be used to define the functions of the programmable stages of the pipeline.



The graphics pipeline

Only the initial and final stages are generally required.

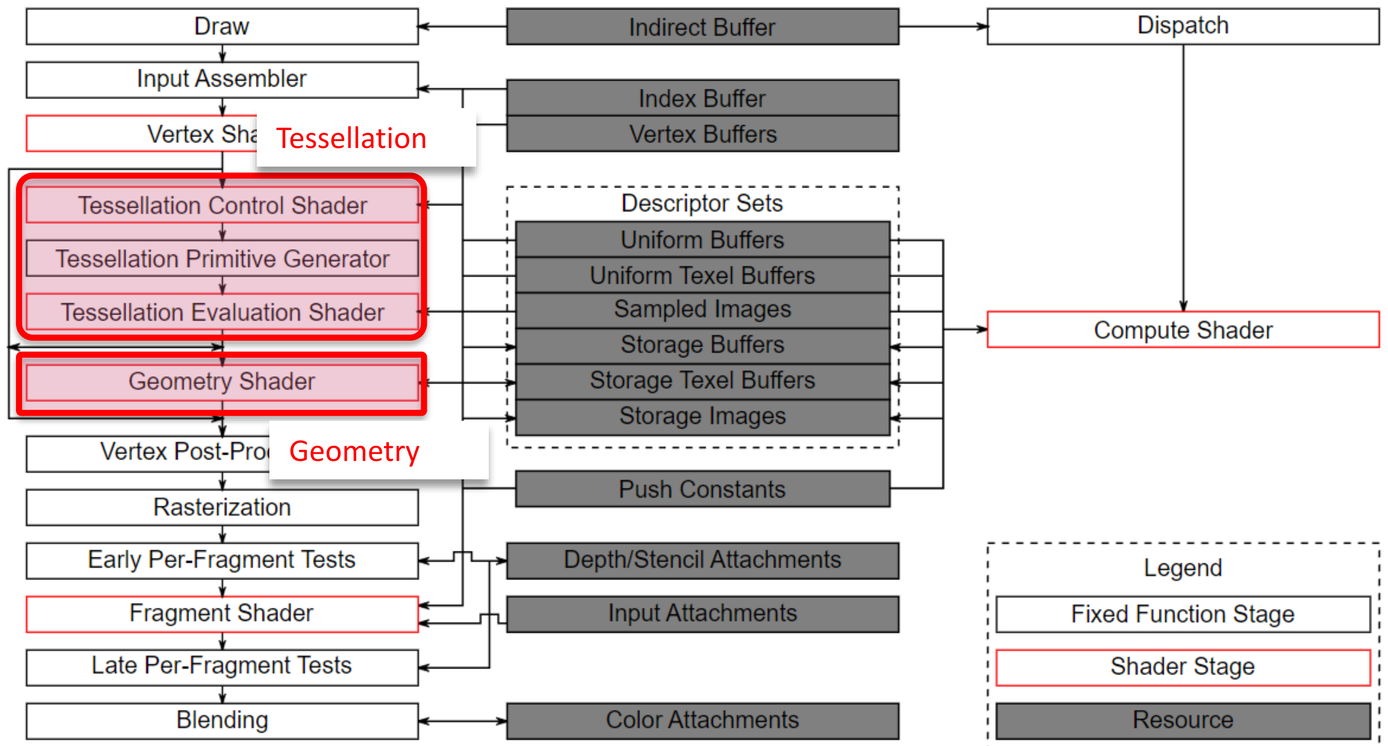
This means that in most of the cases, only the *Vertex* and the *Fragment* shaders are required to generate an image.



The graphics pipeline

Tessellation and geometry stages are optional.

If not present, the pipeline ignores such functions and continues to the following stages.

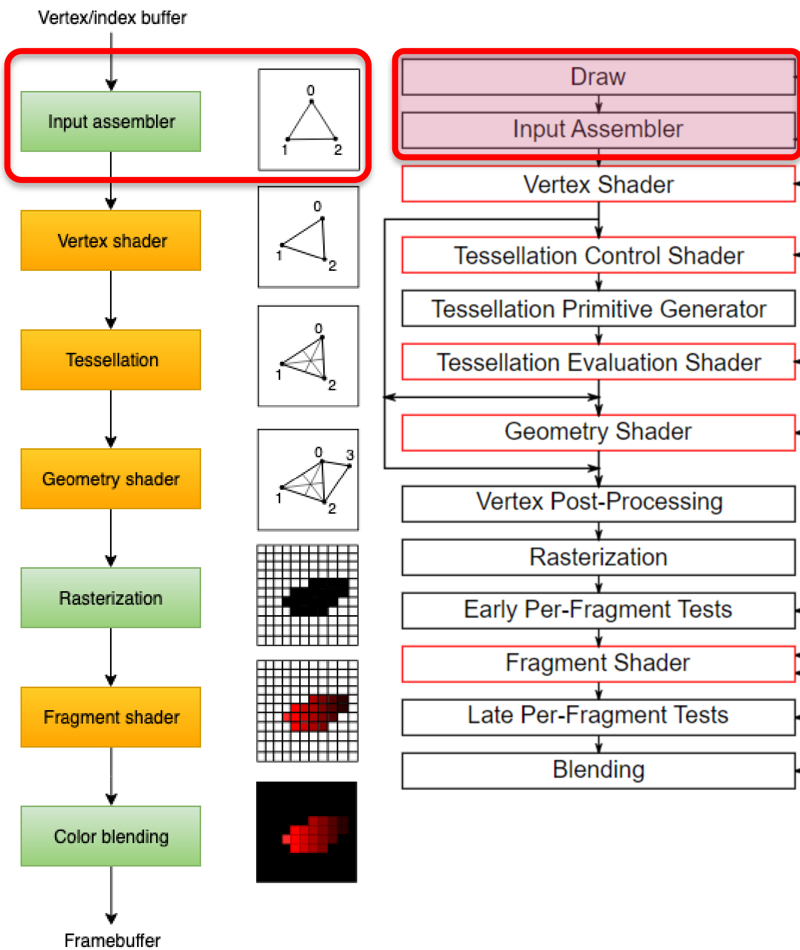


The graphics pipeline

Whenever a draw command is issued, Vulkan creates the vertices by combining all parameters that describe them.

If several instances (copies – we will return on this later) of the same object are used, vertices are replicated as many times as required.

This stage also decides if we are drawing point, lines or triangles, using lists or other strip-based approaches.

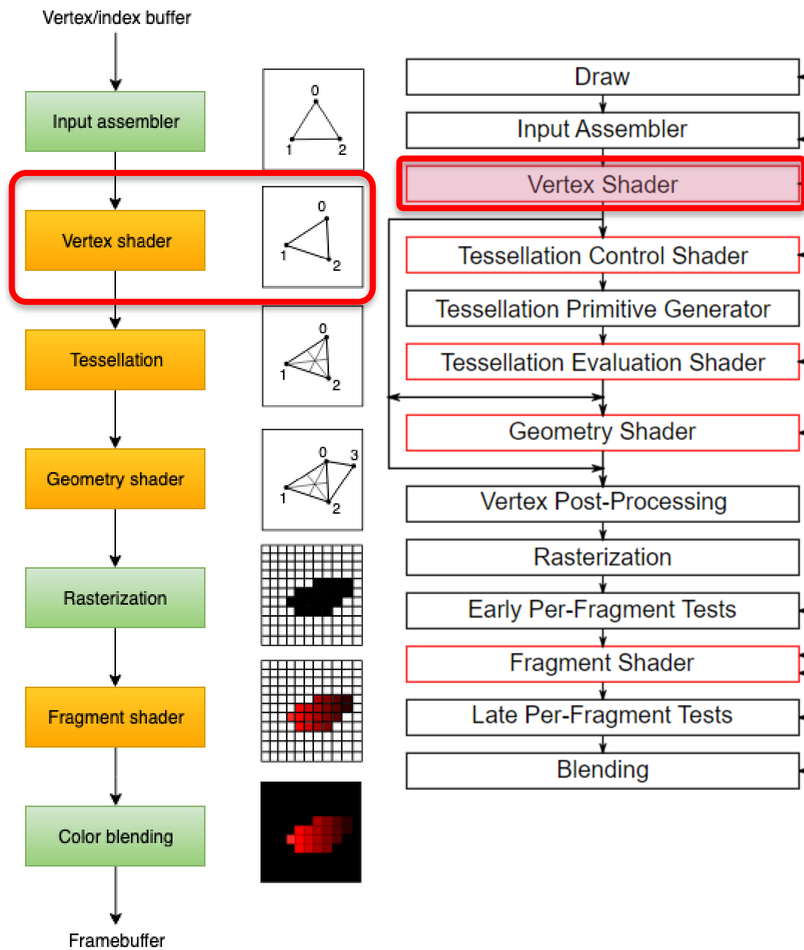


From: https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction

The graphics pipeline

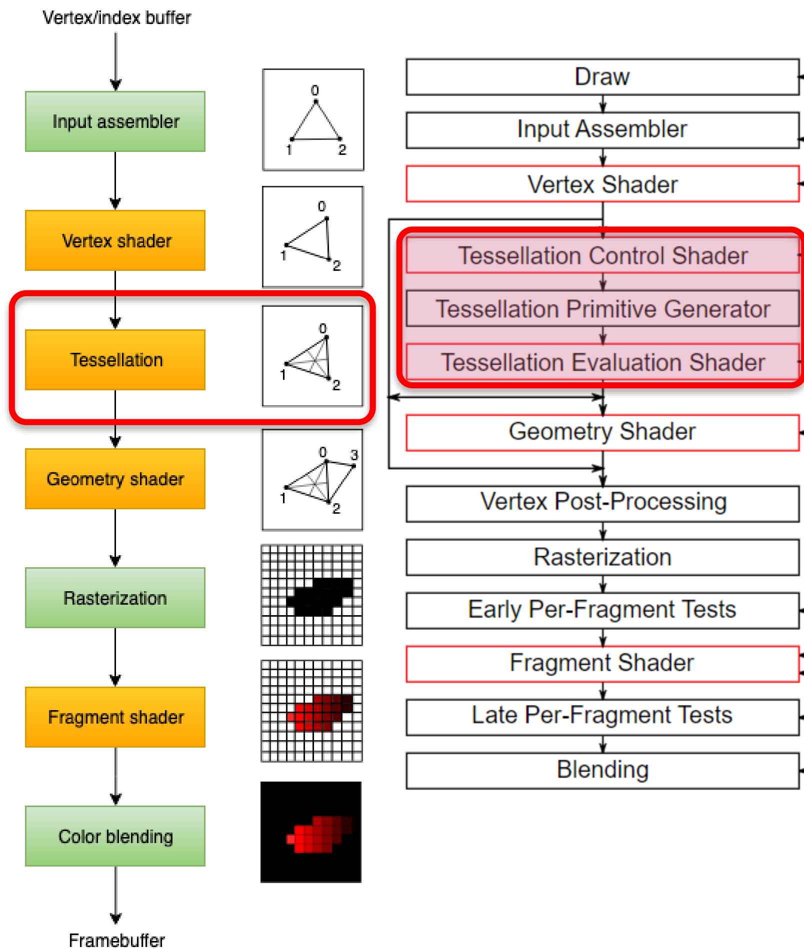
Vertex shaders are then executed to perform operations on each vertex.

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.



The graphics pipeline

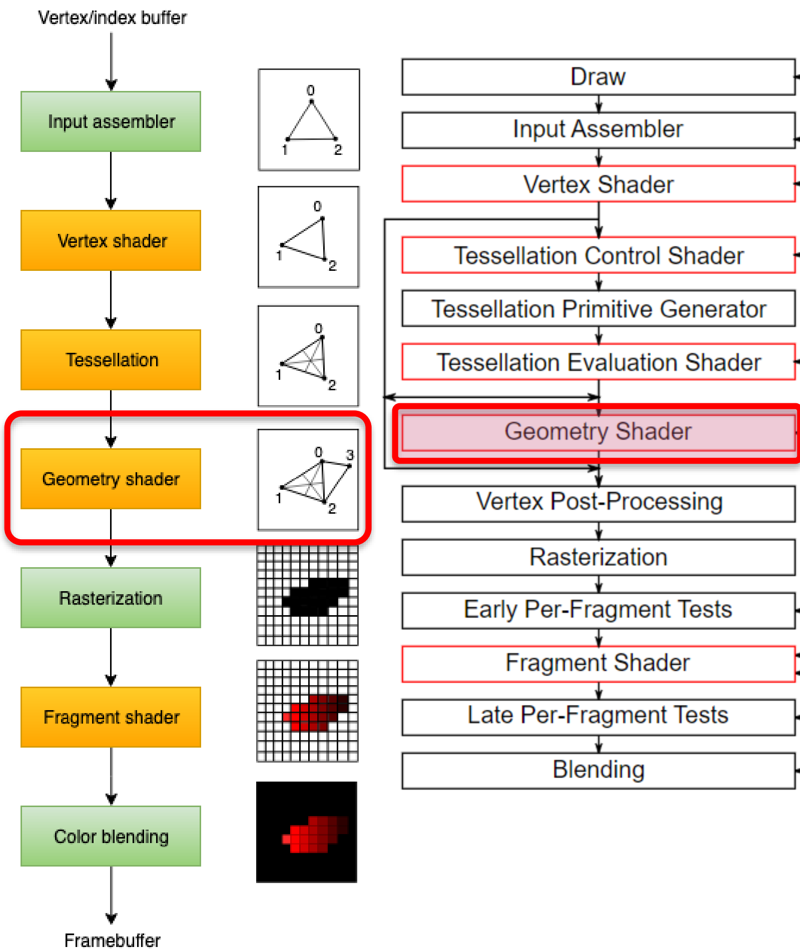
Tessellation is used to increase the resolution of an object: in this way, for example, a sphere can be approximated by few triangles when it is far from the viewer, or with a very high number of subdivisions when seen from a close distance.



The graphics pipeline

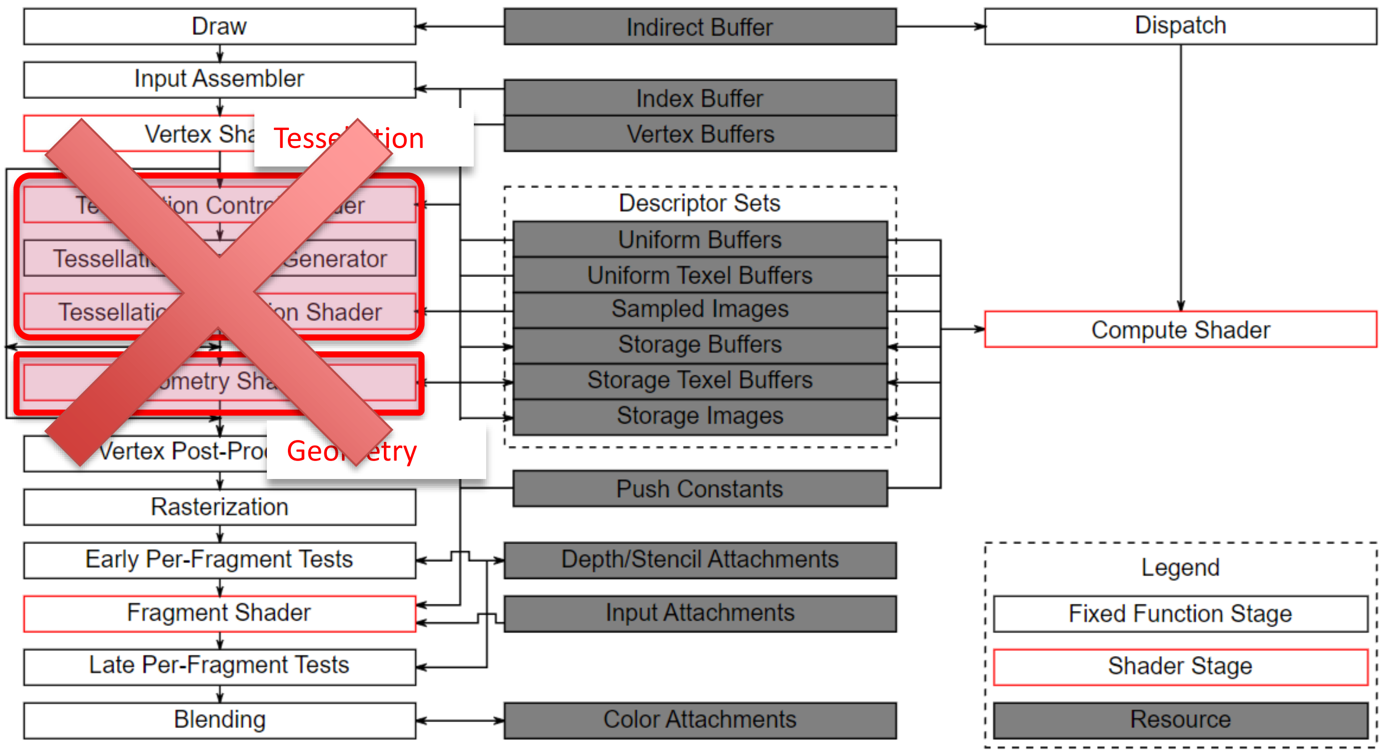
Geometry shaders can remove or add primitives to the stream, starting from the previously generated elements.

In principle, it could perform the same tasks as the tessellation stages: however, due to its generality, implementing these functions in geometry shaders would require more complex code, and will deliver a slower performance.



The graphics pipeline

Due to time constraints, in this course we will not cover tessellation and geometry shaders.

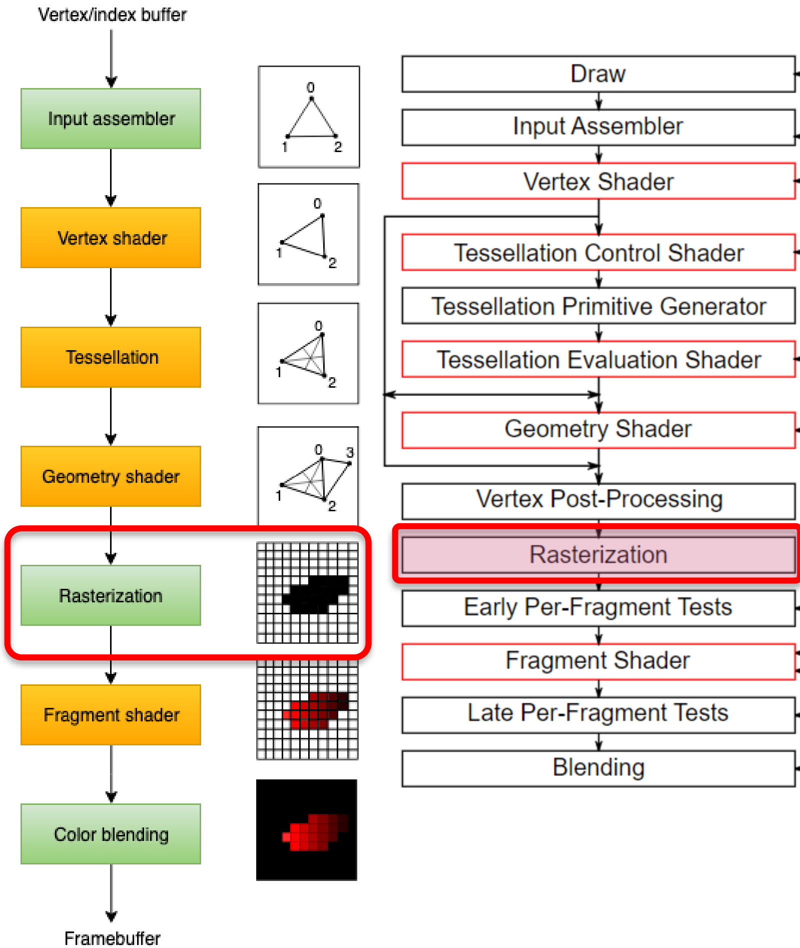


The graphics pipeline

Rasterization determines the pixels in the frame-buffer occupied by each primitives.

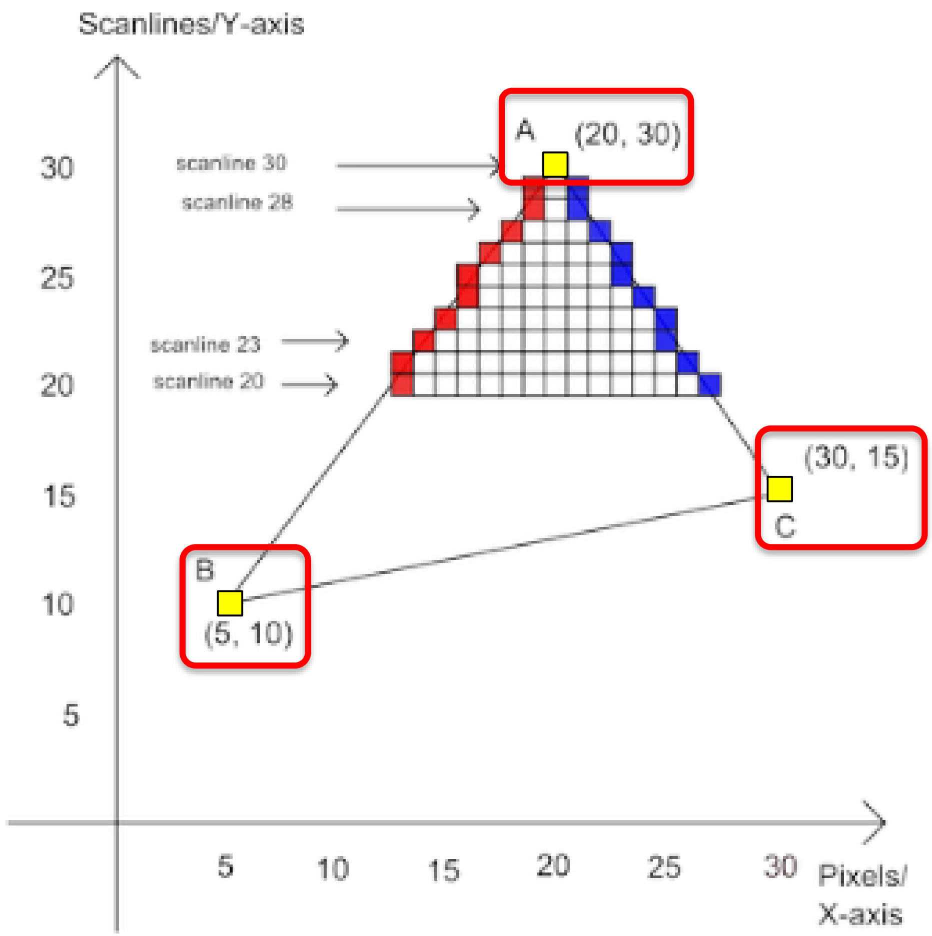
They are called *fragments* and not pixels, since a single pixel on screen can be computed by merging several fragments to increase the quality of the final image (the so called *anti-aliasing*: we will briefly return on this in a future lesson).

In these stages, the "division by w" to transform clipping coordinates into normalized screen coordinates is also performed.



The graphics pipeline

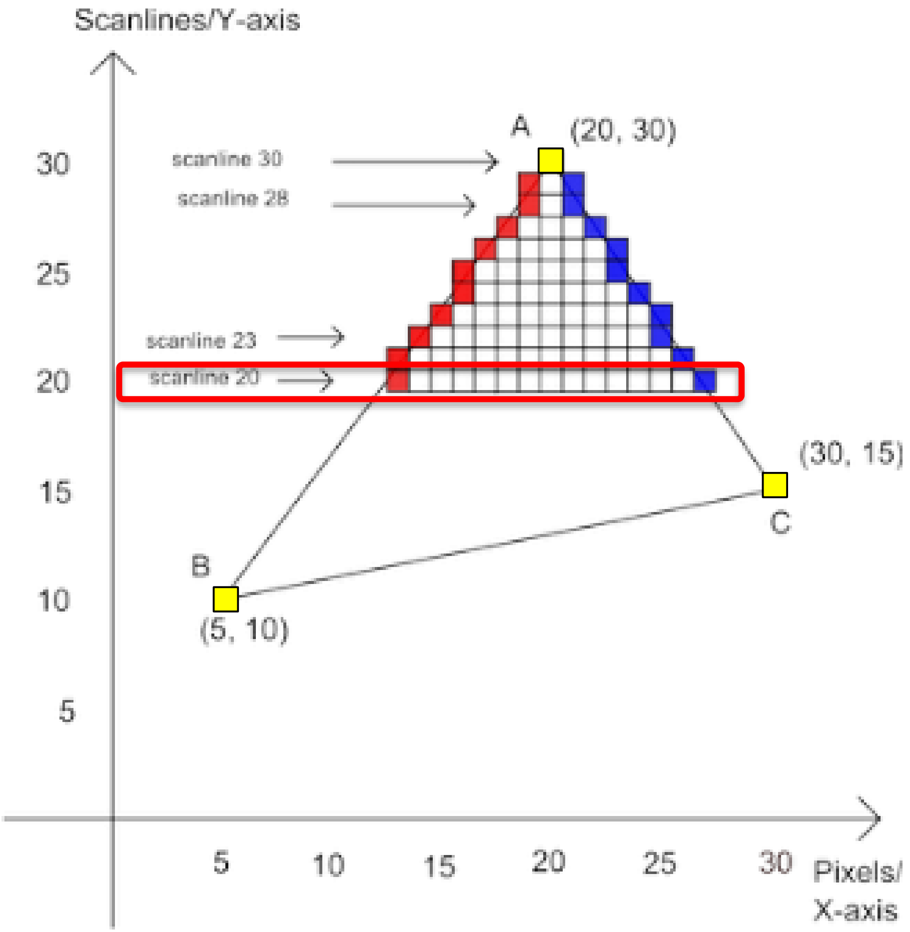
For example, if the considered basic primitive corresponds to a triangle, the rasterization stage will generate at least a fragment for all the pixels connecting the screen projections of its three vertices.



The graphics pipeline

Fragments are usually generated per line, left to right, with respect to the corresponding triangle.

This feature is what motivates the *“scanline rendering”* name of this technique.



The graphics pipeline

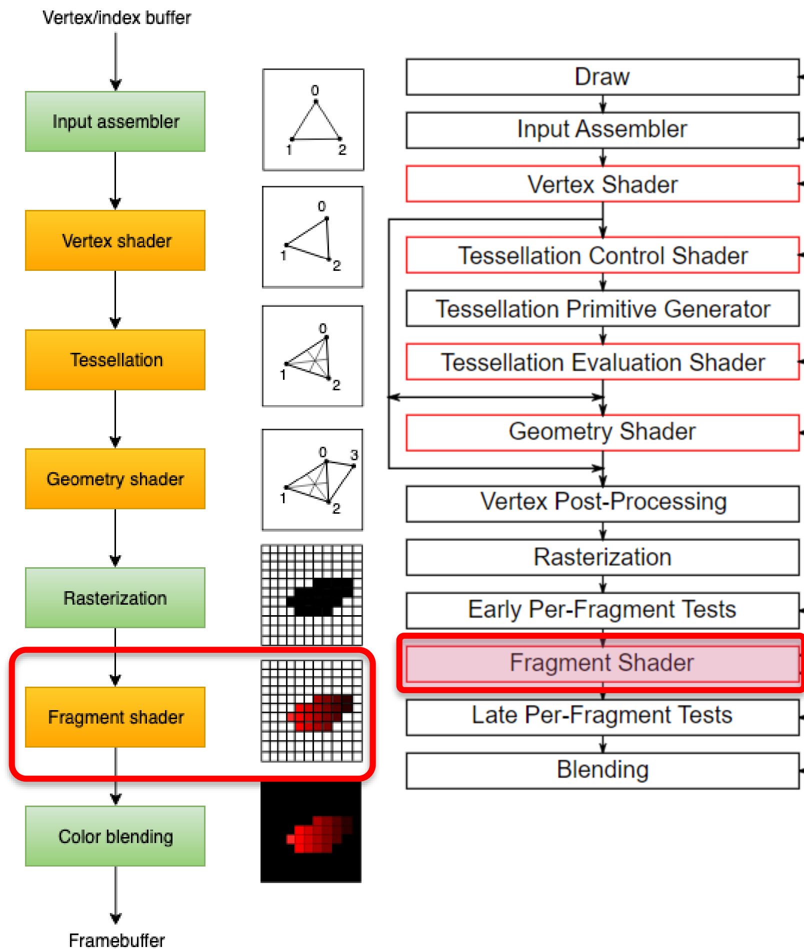
The final color of each fragment is determined by a user defined function contained in the *Fragment shader*.

This section will use either physically based models, or other artistic techniques to produce either realistic or effective images.

In other words, it will compute the approximate solution of the rendering equation for the considered pixel.

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

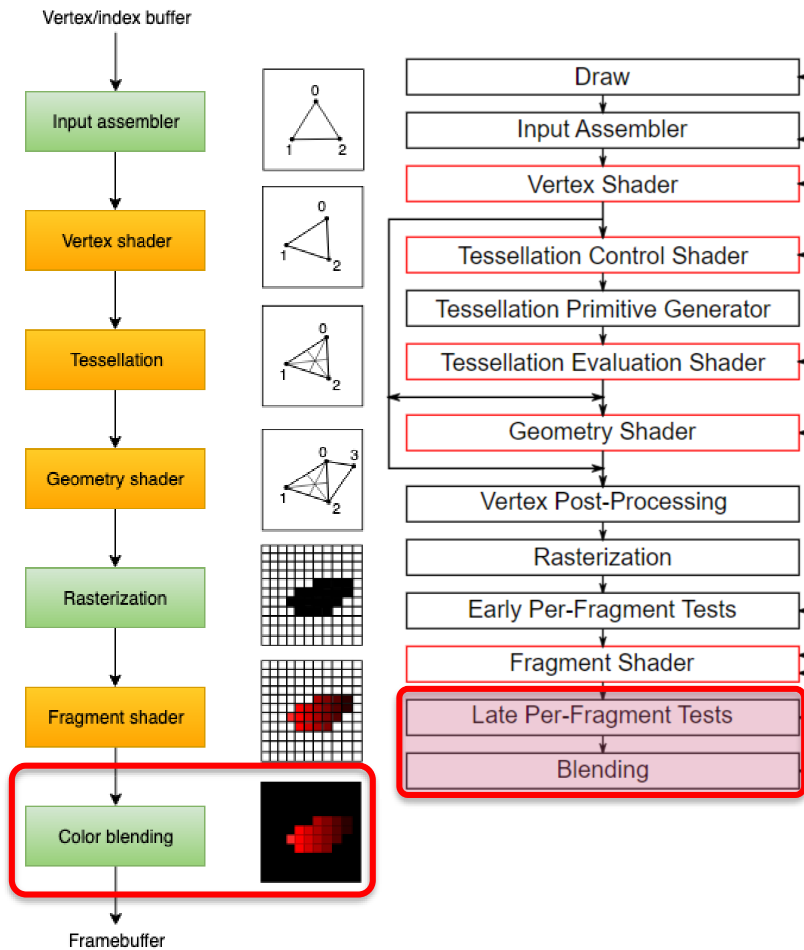
We will consider vertex and fragment shaders functions in depth during the future lessons.



The graphics pipeline

Finally, the computed colors might either replace the ones already present in the same position, or be combined with them.

The latter can be used to implement transparency, or other blending effects.

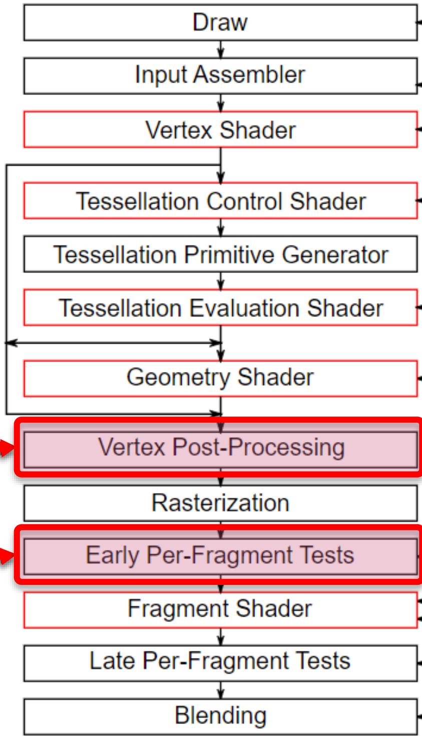


Fixed functions

Several important actions occurs in the final fixed sections of the pipeline.

In particular, that is were the a few of the functionality previously introduced take place:

- Primitives clipping
- Back-face culling
- Depth testing (*z-buffer*)
- Stencil



Scan-line rendering implementation

The pseudo-code of a scan-line rendering algorithm is thus the following:

```
for each mesh object A in the scene do
  Determine the screen coordinates of each vertex of t
  for each visible (passes the back-face culling and clipping) triangle t of A do
    for each pixel x of t on screen do
      if pixel x is visible (passes Z-buffer test) then
        Set the pixel color C = L_e(x, ω_r) (emission and ambient light)
        for each light l in the scene do
          Set C = C + L(l,lx)*fr(x,lx,ω_r)
                      (contribution of light l to the color of x)
        end
      end if
    end
  end
end
```

Scan-line rendering implementation

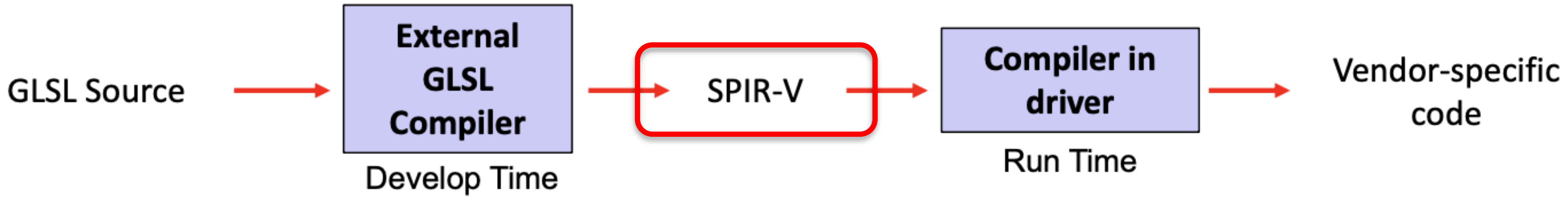
When performing real-time rendering with Vulkan, the user implements the parts in red with the *Fragment Shader*, the one in blue with the *Vertex Shader*, while Vulkan does the parts in green.

```
for each object A in the scene do
  Determine the screen coordinates of each vertex of t
  for each visible (passes back-face culling and clipping) triangle t of A do
    for each pixel x of t on screen do
      if pixel x is visible (passes Z-buffer test) then
        Set the pixel color C = L_e(x,ω_r)
        for each light l in the scene do
          Set C = C + L(l,lx)*fr(x,lx,ω_r)
        end
      end if
    end
  end
end
```

Shaders

In Vulkan, shaders are defined by **SPIR-V** code blocks.

SPIR stands for *Standard Portable Intermediate Representation*, and it is a binary format for specifying instructions that a GPU can run in a device independent way.



Shaders

Every Vulkan driver converts the SPIR-V code into the binary instructions of their corresponding GPU.

SPIR-V has been created with the goal of being efficiently converted into instructions for the most popular GPUs, so this process is usually not very expensive from a computational point of view.

Shaders

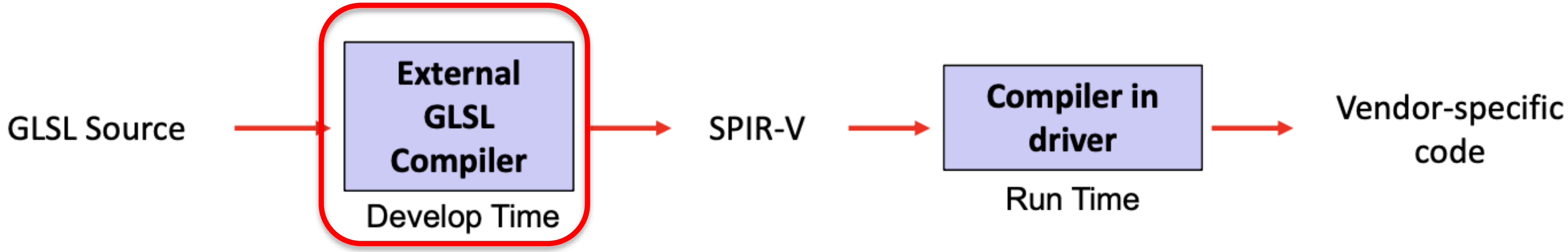
Shaders are written in high level languages, such as:

- GLSL (openGL Shading Language)
- HLSL (High Level Shading Language – Microsoft Direct X)

In this course, we will focus on GLSL.

Shaders

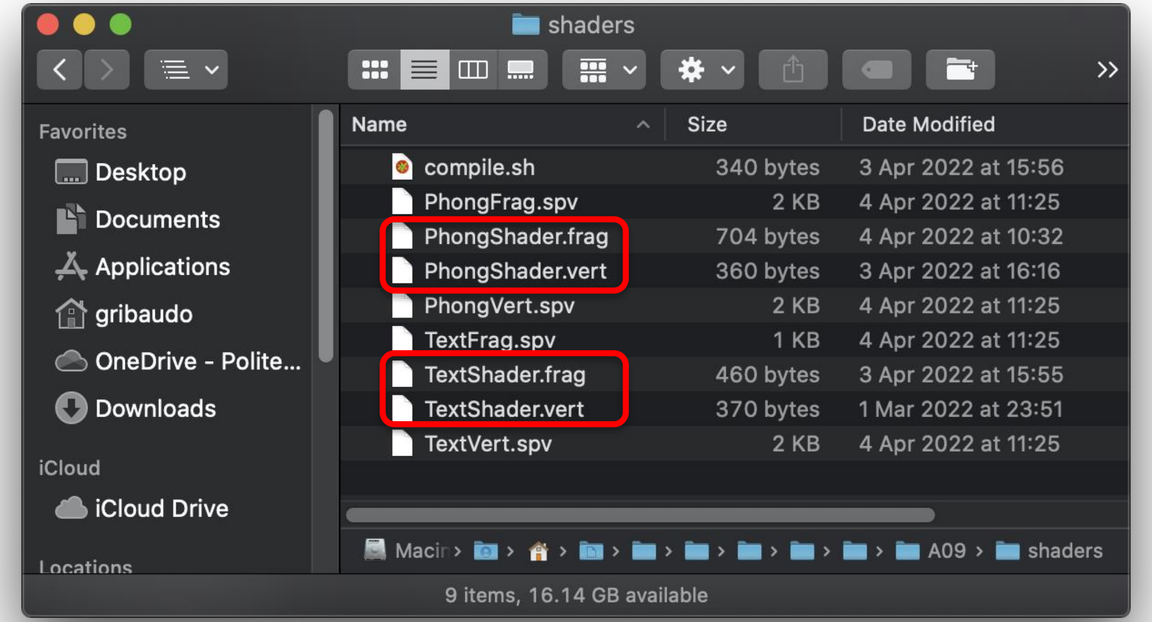
At development time, the shaders are compiled from their original language to SPIR-V.



Compiling a shader into SPIR-V

In general, depending on the shader type, the file containing the its source code has a different extension.

For example, the following are the ones corresponding to the shaders used in many or our *Assignments*:



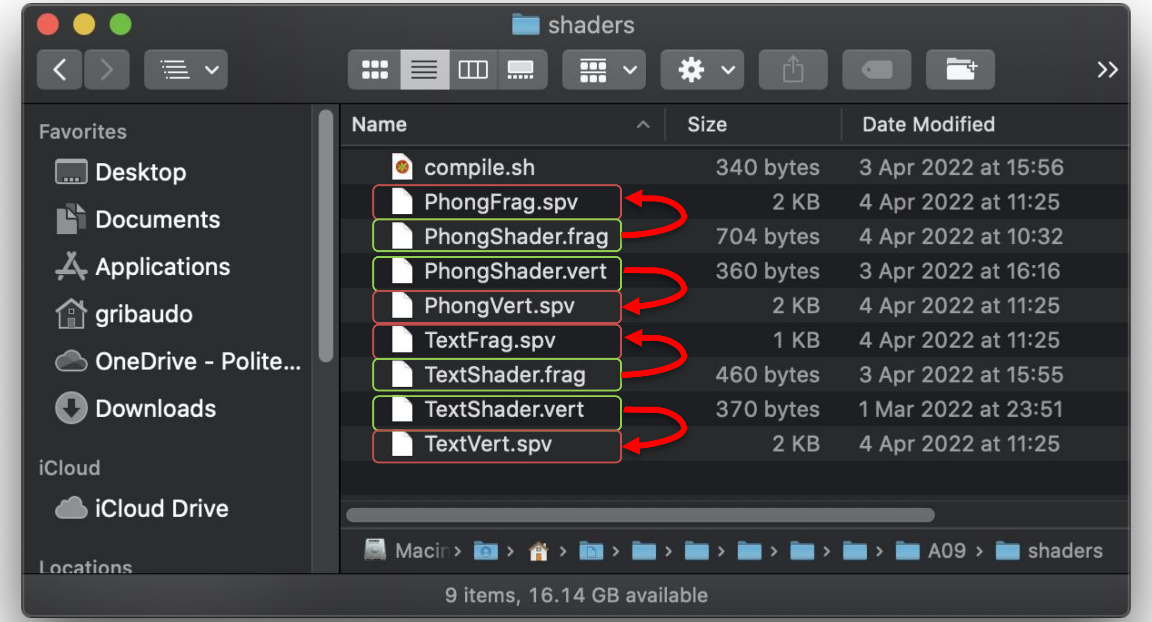
Compiling a shader into SPIR-V

The most popular extensions for GLSL shaders source code files are the following:

```
.vert   for a vertex shader
.tesc   for a tessellation control shader
.tese   for a tessellation evaluation shader
.geom   for a geometry shader
.frag   for a fragment shader
.comp   for a compute shader
.mesh   for a mesh shader
.task   for a task shader
.rgen    for a ray generation shader
.rint    for a ray intersection shader
.rahit   for a ray any hit shader
.rchit   for a ray closest hit shader
.rmiss   for a ray miss shader
.rcall   for a ray callable shader
```

Compiling a shader into SPIR-V

Compiled shaders into SPIR-V files have instead the *.spv* extension.



Compiling a shader into SPIR-V

Shaders can be compiled using the `glslc` tool, which is included in the Vulkan SDK.

Windows

Create a **compile.bat** file with the following contents:

```
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe shader.vert -o vert.spv
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe shader.frag -o frag.spv
pause
```

Replace the path to **glslc.exe** with the path to where you installed the Vulkan SDK. Double click the file to run it.

Linux and MacOS

Create a **compile.sh** file with the following contents:

```
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslc shader.vert -o vert.spv
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslc shader.frag -o frag.spv
```

Replace the path to **glslc** with the path to where you installed the Vulkan SDK. Make the script executable with **chmod +x compile.sh** and run it.

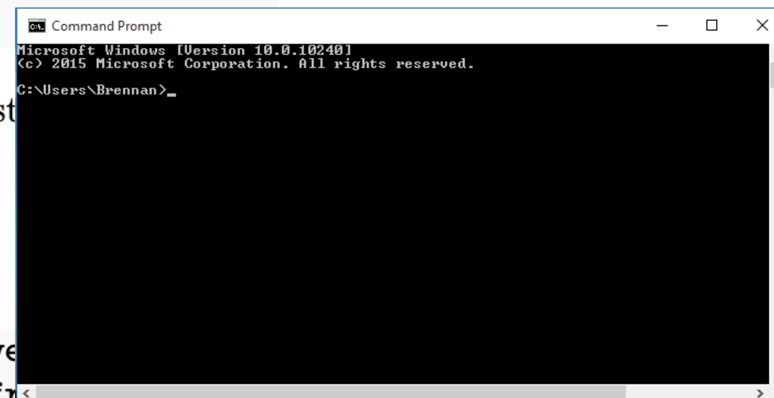
Compiling a shader into SPIR-V

The most difficult part is locating the tool in the SDK folder, and manually call it from a *command line window*.

Windows

Create a **compile.bat** file with the following contents:

```
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe shader.vert -o vert.spv
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe shader.frag -o frag.spv
pause
```



Replace the path to **glslc.exe** with the path to where you inst
SDK. Double click the file to run it.

Linux and MacOS

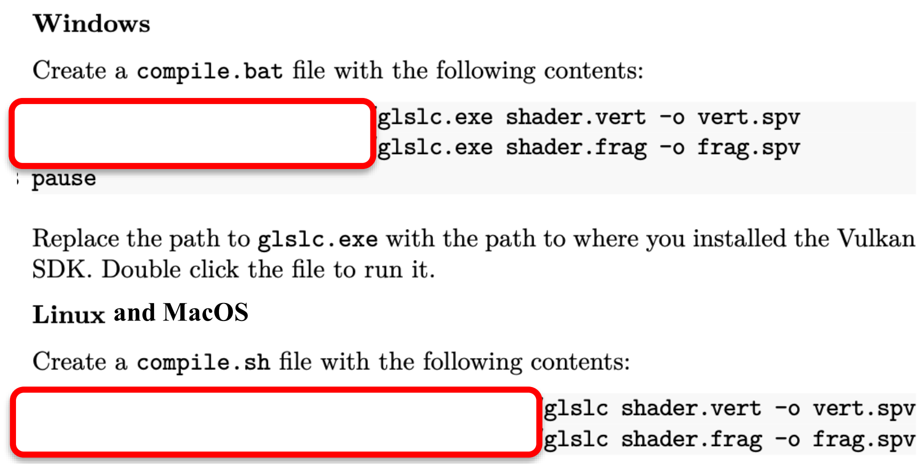
Create a **compile.sh** file with the following contents:

```
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslc shader.ve
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslc shader.fr
```

Replace the path to **glslc** with the path to where you installed the Vulkan SDK. Make the script executable with **chmod +x compile.sh** and run it.

Compiling a shader into SPIR-V

Many (correct) Vulkan installations, however, makes `glslc` directly available in the main command path, simplifying a lot the procedure.



Windows

Create a `compile.bat` file with the following contents:

```
glslc.exe shader.vert -o vert.spv
glslc.exe shader.frag -o frag.spv
pause
```

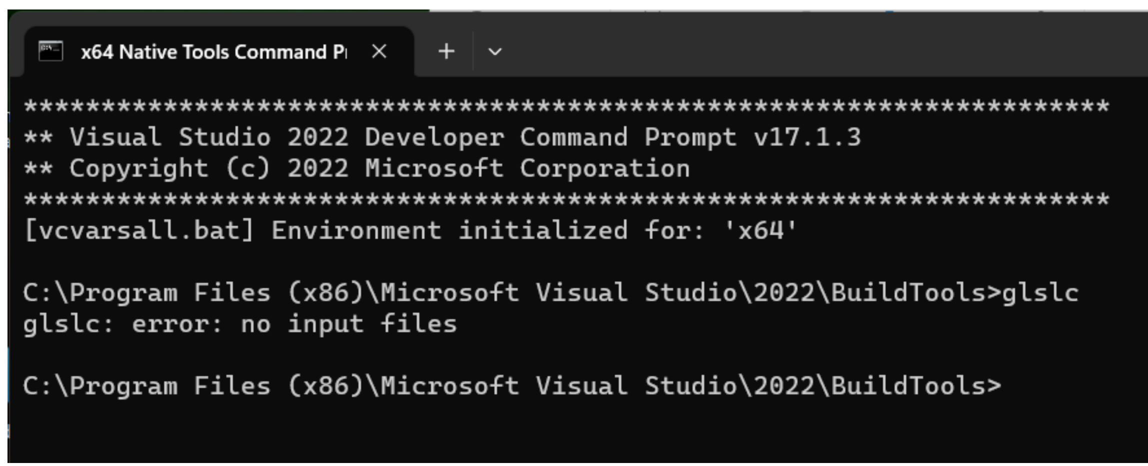
Replace the path to `glslc.exe` with the path to where you installed the Vulkan SDK. Double click the file to run it.

Linux and MacOS

Create a `compile.sh` file with the following contents:

```
glslc shader.vert -o vert.spv
glslc shader.frag -o frag.spv
```

Just open a terminal window, write `glslc` and press enter: if you get the "`glslc: error: no input files`", this means the path was set successfully, and you can invoke the compiler directly.



Compiling a shader into SPIR-V

The command then requires the name of the GLSL source file to compile…

Windows

Create a `compile.bat` file with the following contents:

```
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe shader.vert -o vert.spv
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe shader.frag -o frag.spv
pause
```

Replace the path to `glslc.exe` with the path to where you installed the Vulkan SDK. Double click the file to run it.

Linux and MacOS

Create a `compile.sh` file with the following contents:

```
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslc shader.vert -o vert.spv
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslc shader.frag -o frag.spv
```

Replace the path to `glslc` with the path to where you installed the Vulkan SDK. Make the script executable with `chmod +x compile.sh` and run it.

Compiling a shader into SPIR-V

… followed by the `-o` option, and the name of the `.spv` file that will be generated by the compiler.

Windows

Create a **compile.bat** file with the following contents:

```
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe shader.vert -o vert.spv
C:/VulkanSDK/x.x.x.x/Bin32/glslc.exe shader.frag -o frag.spv
pause
```

Replace the path to **glslc.exe** with the path to where you installed the Vulkan SDK. Double click the file to run it.

Linux and MacOS

Create a **compile.sh** file with the following contents:

```
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslc shader.vert -o vert.spv
/home/user/VulkanSDK/x.x.x.x/x86_64/bin/glslc shader.frag -o frag.spv
```

Replace the path to **glslc** with the path to where you installed the Vulkan SDK. Make the script executable with `chmod +x compile.sh` and run it.

Notes on pipelines and their creation

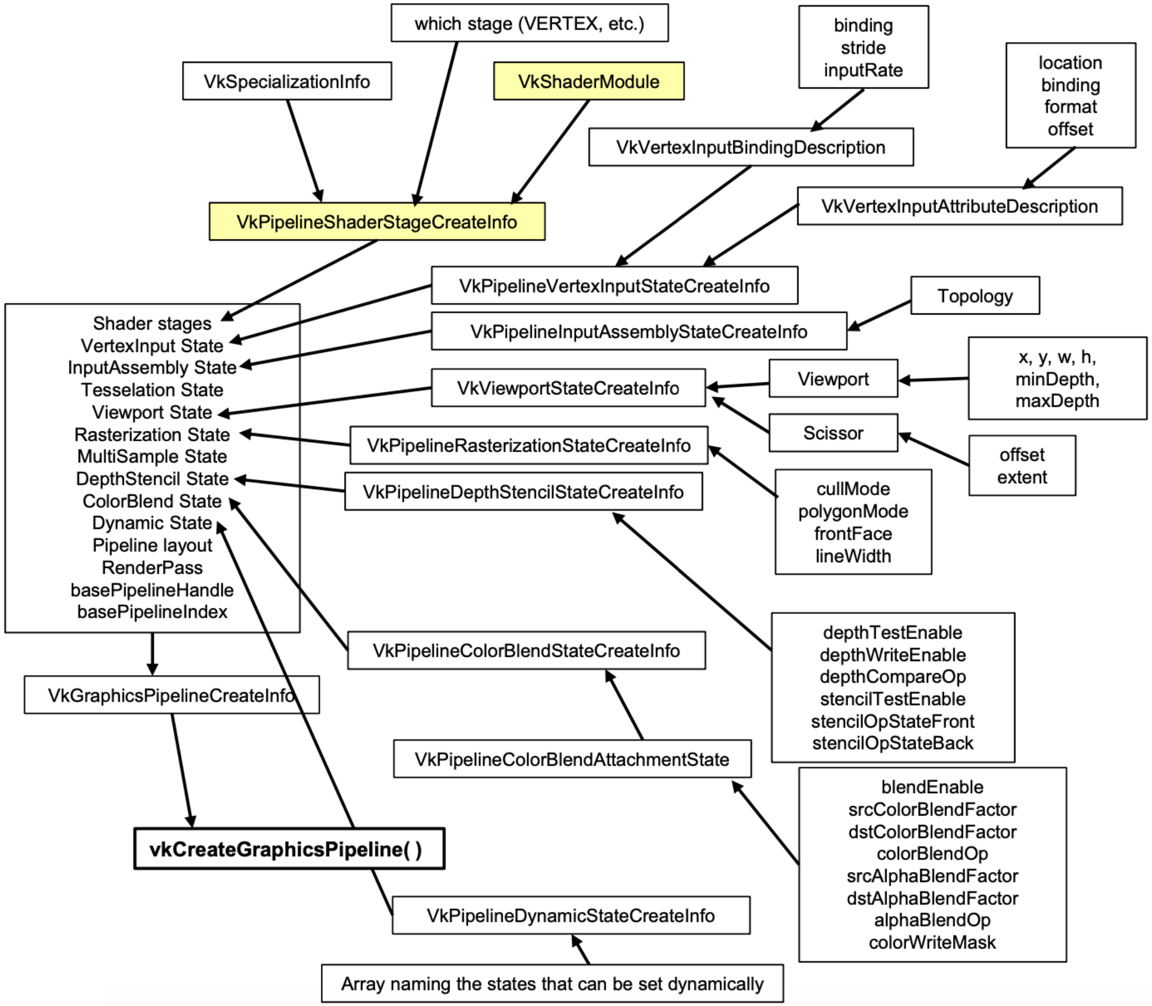
A 3D application generally requires several pipelines to compose the final image, each one characterized by its own parameters and shaders.

The creation of one pipeline is also the most complex part of a 3D application: in the `Starter.hpp` header used in the assignments to wrap Vulkan interactions, this process requires around 185 lines of code to load the shaders and setup all the parameters of the fixed functions.

Notes on pipelines and their creation

In the next lessons, we will focus on the most important features of pipeline creation.

This figure, summarizes all the parameters that can be configured in a graphics pipeline.



From: https://web.engr.oregonstate.edu/~mjb/vulkan/

Ray casting

Ray casting is an extension of the scan-line rendering that computes the visibility function for all the triangle points / scene light couples in the scene.

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

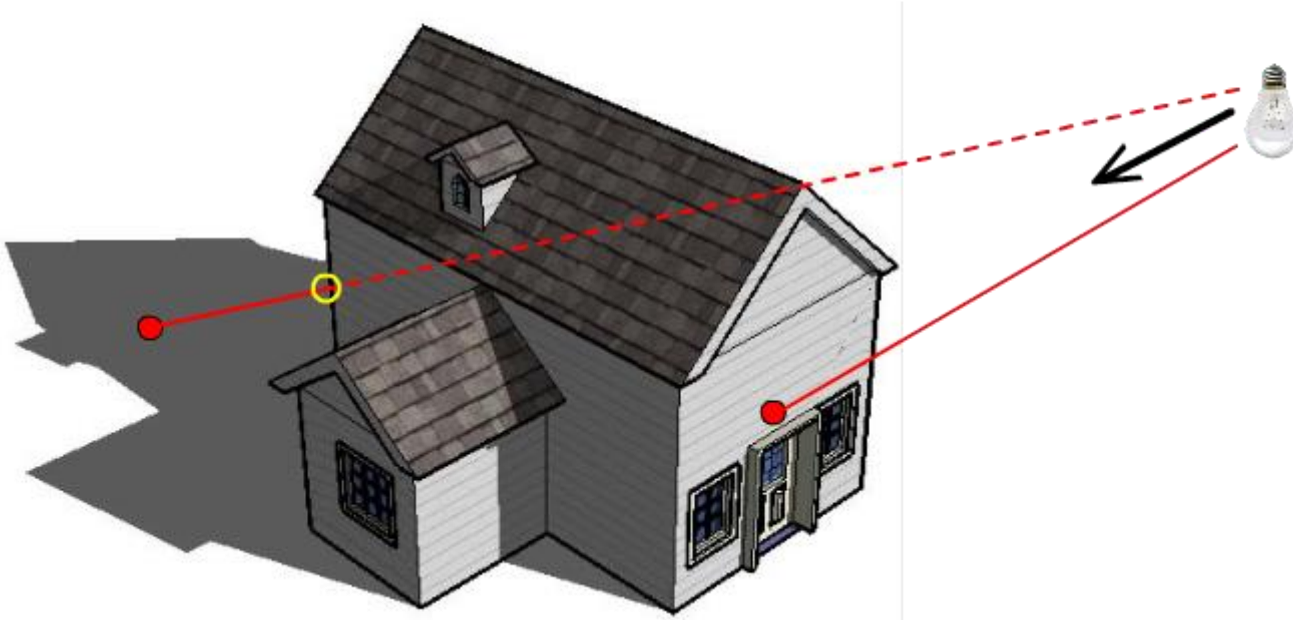
$$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L(l,\overrightarrow{lx}) f_{r'}(x,\overrightarrow{lx},\omega_r) \boxed{V(x,l)}$$

$L(x,\omega_r) = L_e(x,\omega_r) + \sum_l L_e(l,\overrightarrow{lx}) f_{r'}(x,\overrightarrow{lx},\omega_r)$ ← For comparison, this was the scan-line rendering equation

Ray casting

Ray casting allows the inclusion of projected shadows.

The visibility function is computed by casting a ray that connects the considered points with each light source: if the ray intersects an object, the light is occluded and its effect is not considered in the rendering equation.



Ray casting

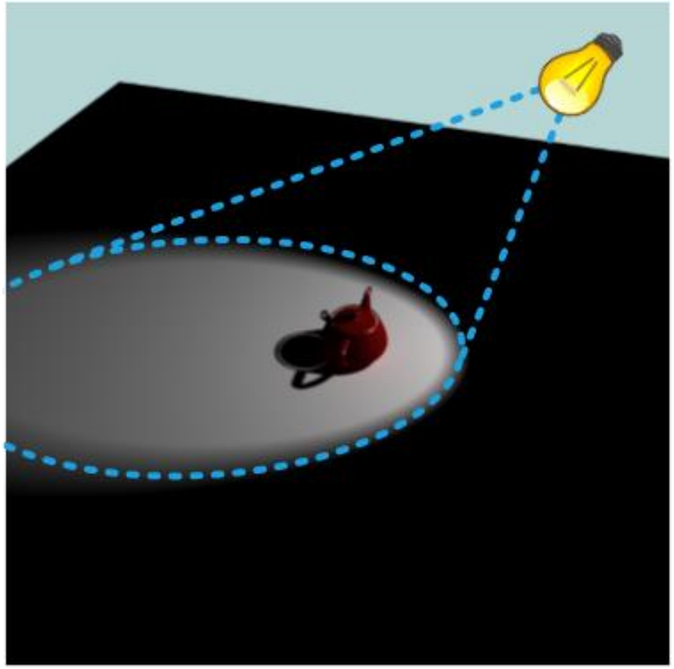
The pseudo-code of a ray casting rendering algorithm is the following:

```
for each object A in the scene do
  for each visible (pass back-face culling and clipping) triangle t of A do
    for each pixel x of t on screen do
      if pixel x is visible (pass Z-buffer test) then
        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 + L(l,lx)*fr(x,lx,ω_r)
          end if
        end
      end if
    end
  end
end
```

Ray casting

One of the typical techniques to perform ray-castin in real-time, is the use of a *Shadow Map*:

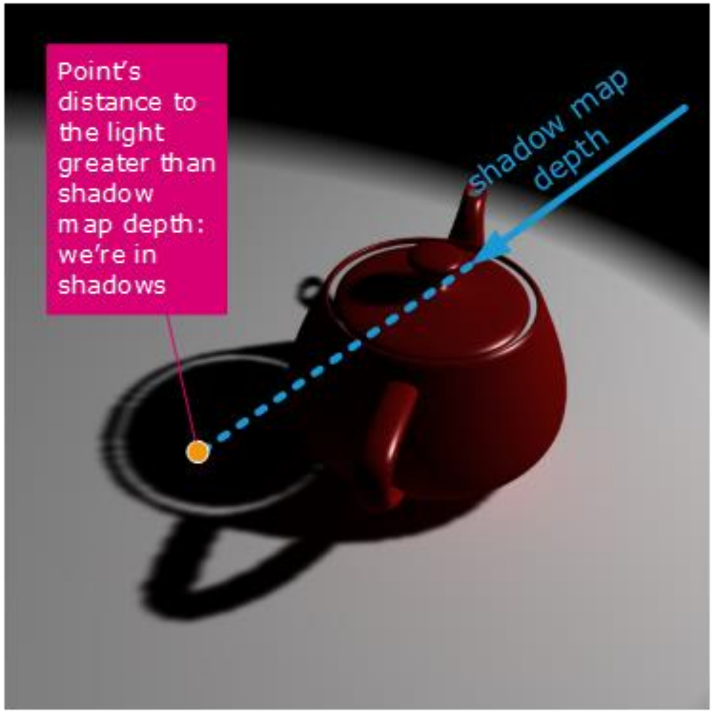
- an image rendered from the position of the light source, where the color of each pixel accounts for the distance of the point from the light.



Ray casting

The information stored in the image is used to determine if a pixel is hit by the light or not.

The shader computes the distance from the light source, and compares it with the one in the map: if it greater, then the light is not considered.



Ray casting is generally implemented in *the graphics pipeline*, by executing several passes: first the shadow maps for each light are computed, and later they are used to determine if a light hits a point.

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)