Pipelines and Shaders

Pipelines

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

This sequence of operations is called a *Pipeline*, since it resembles the classical system organization 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 *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.

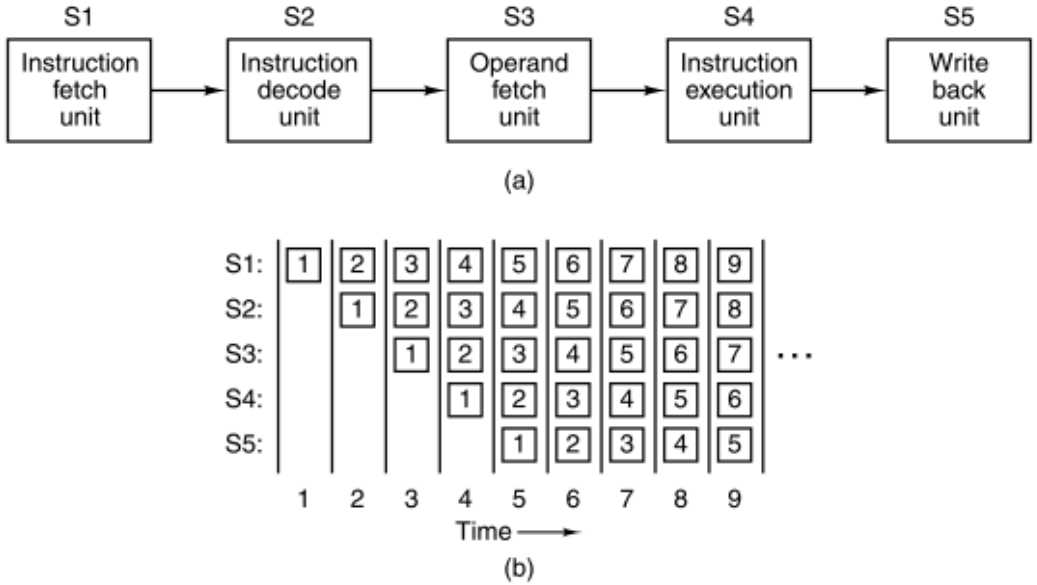
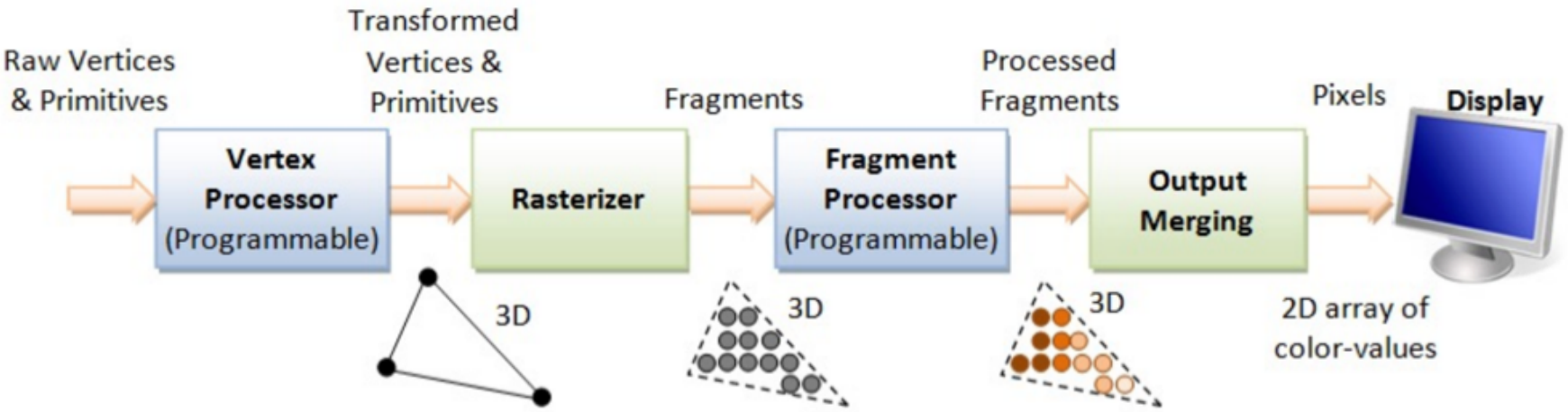


Figure 2-4. (a) A five-stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated.

Pipeline: the general concept

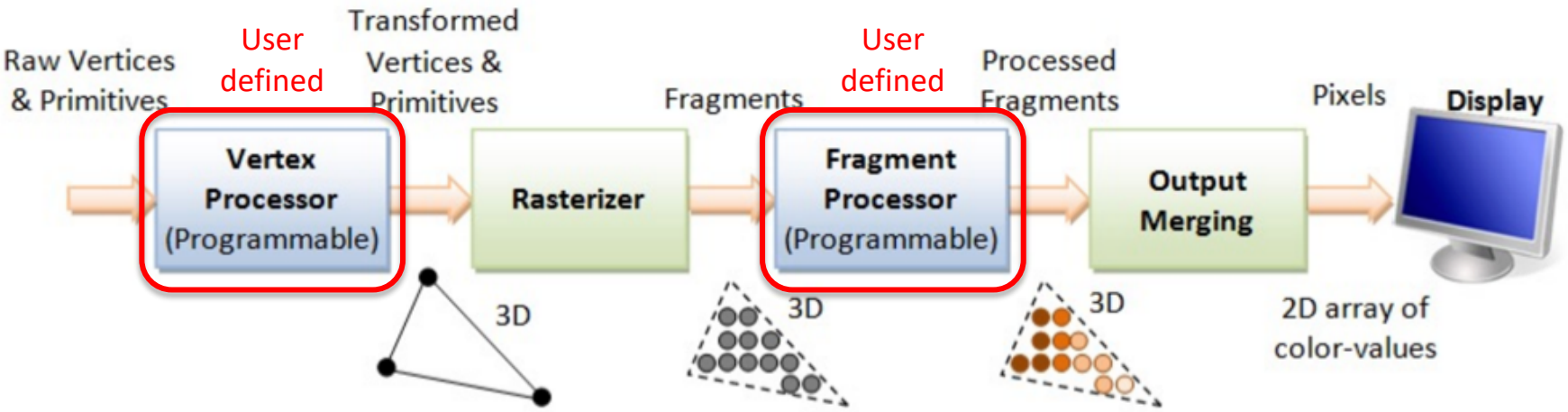
In Vulkan, and in CG in general, the process of creating an image on screen starting from the primitive description is accomplished through a pipeline.



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

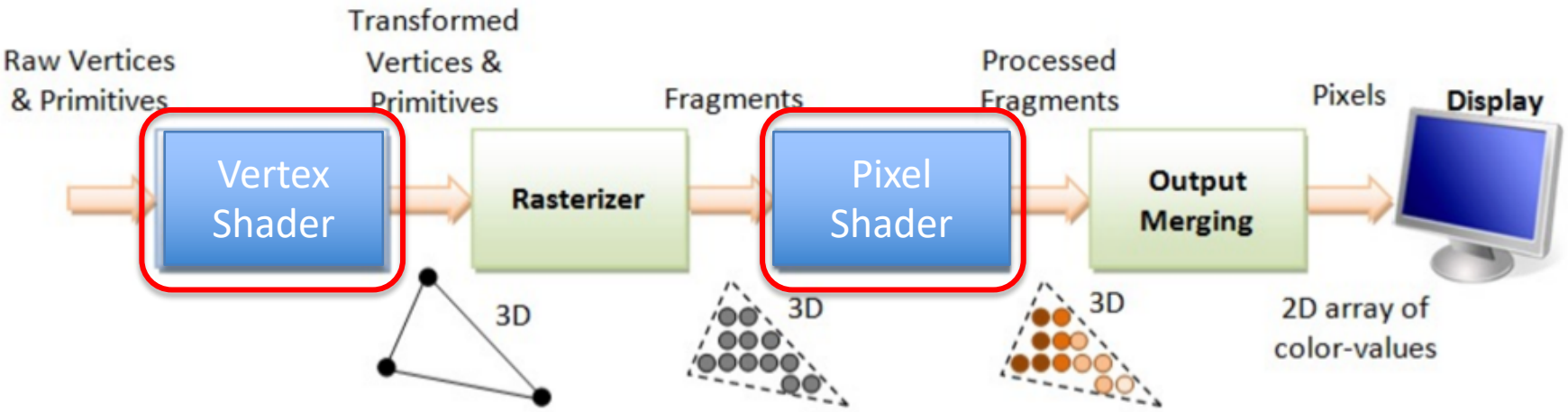
The actions taken in each stage of the pipeline can either be defined by the system (Vulkan, in our case), or programmed by the user.



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*.



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 different purposes, have been defined.

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 required by its fixed functions, and the connection with the shaders that performs its variable part.

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 just for general computation purposes (i.e. GPGPU). *Ray-tracing* and *Mesh Shading* are still in a very early stage, (especially the latter), and might have a very limited support.

Pipeline types

In this lesson, we will only focus on the first type:

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

The other pipelines will be very briefly introduced in future lessons.

Shaders

When creating a pipeline, the user has to:

- Specify parameters to control the fixed functions
- Provide shaders for the programmable stages

All shaders have very similar interfaces, and are connected in a common way.

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 GPU, so this process is usually a not very expensive task.

Shaders

Shaders are written in high level languages, such as:

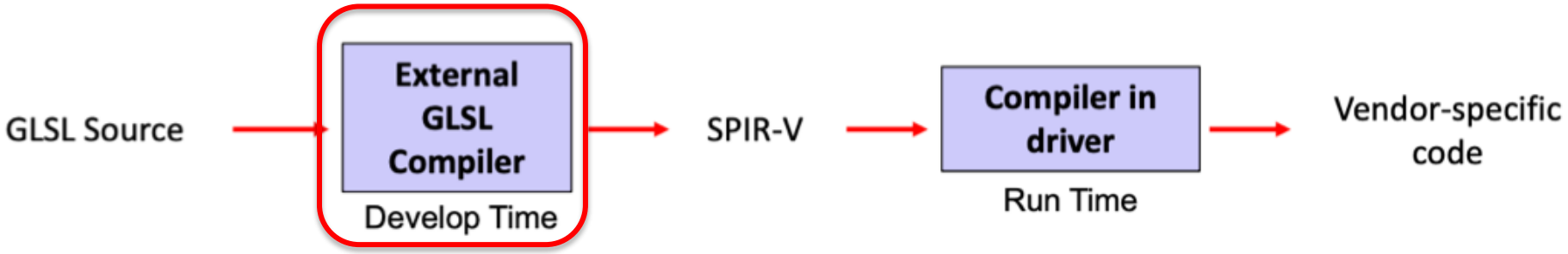
- GLSL (openGL Shading Language)
- HLSL (High Level Shading Language – Microsoft Direct X)

In later lessons, we will describe GLSL in depth.

Shaders

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

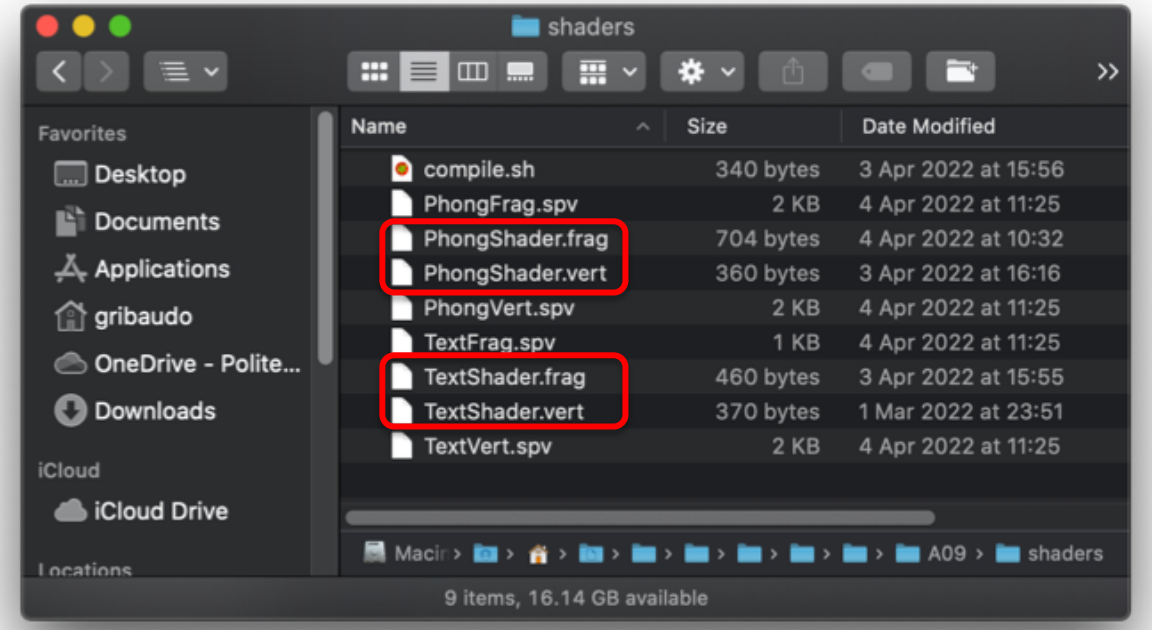
Although the main language supported by Vulkan is GLSL, compilers to transform HLSL shaders into SPIR-V exists, allowing developer to enjoy the two most popular alternatives.



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 Assignment 9:



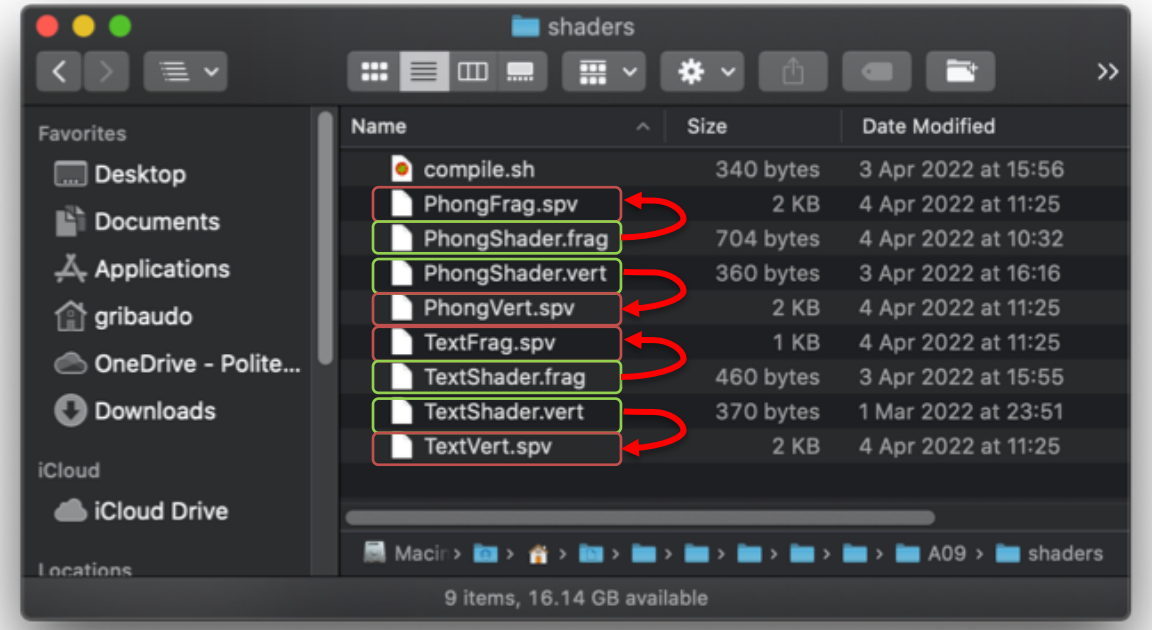
Compiling a shader into SPIR-V

The most popular extensions 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 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.

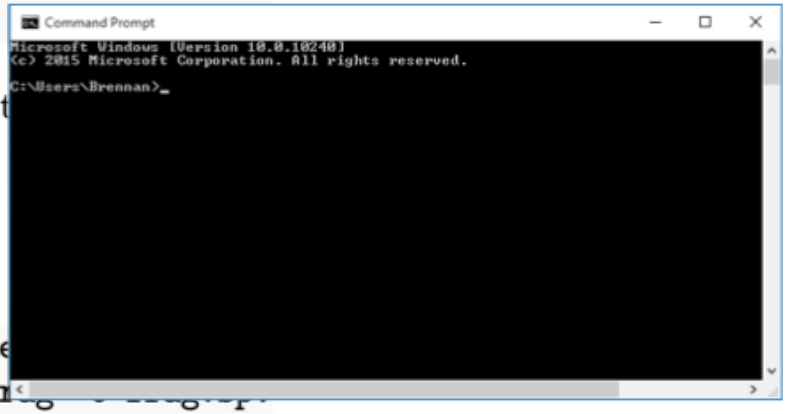
Compiling a shader into SPIR-V

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

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 option `-o` and the name of the `.spv` file that will contain the compiled version of the shader.

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.

The graphics pipeline

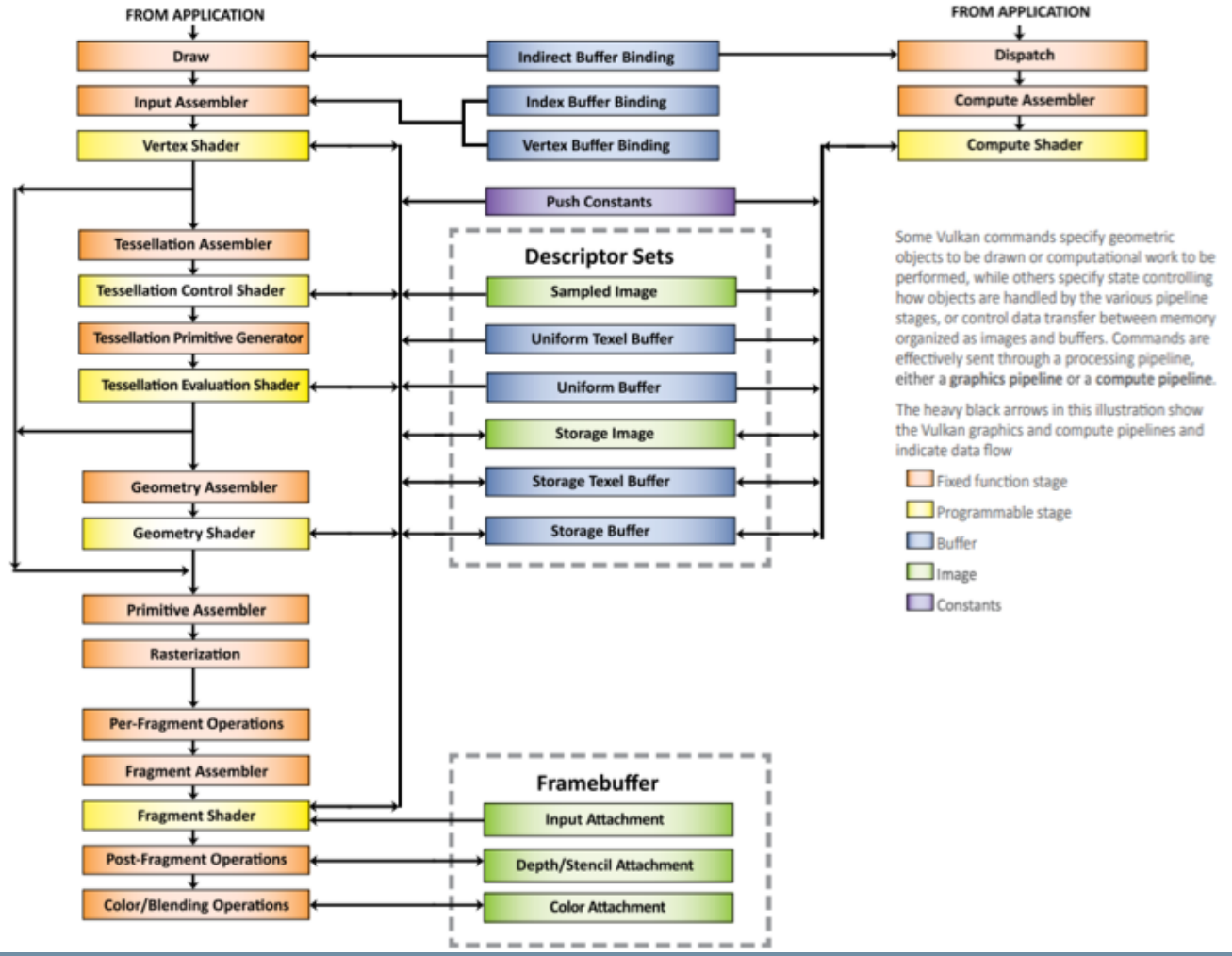
The most common way of rendering an image from a set of 3D primitives is using the graphics pipeline.

Let's see in detail which are its stages, and what do they means.

In a following lesson, we will see how these stages are configured in the source code of a Vulkan application.

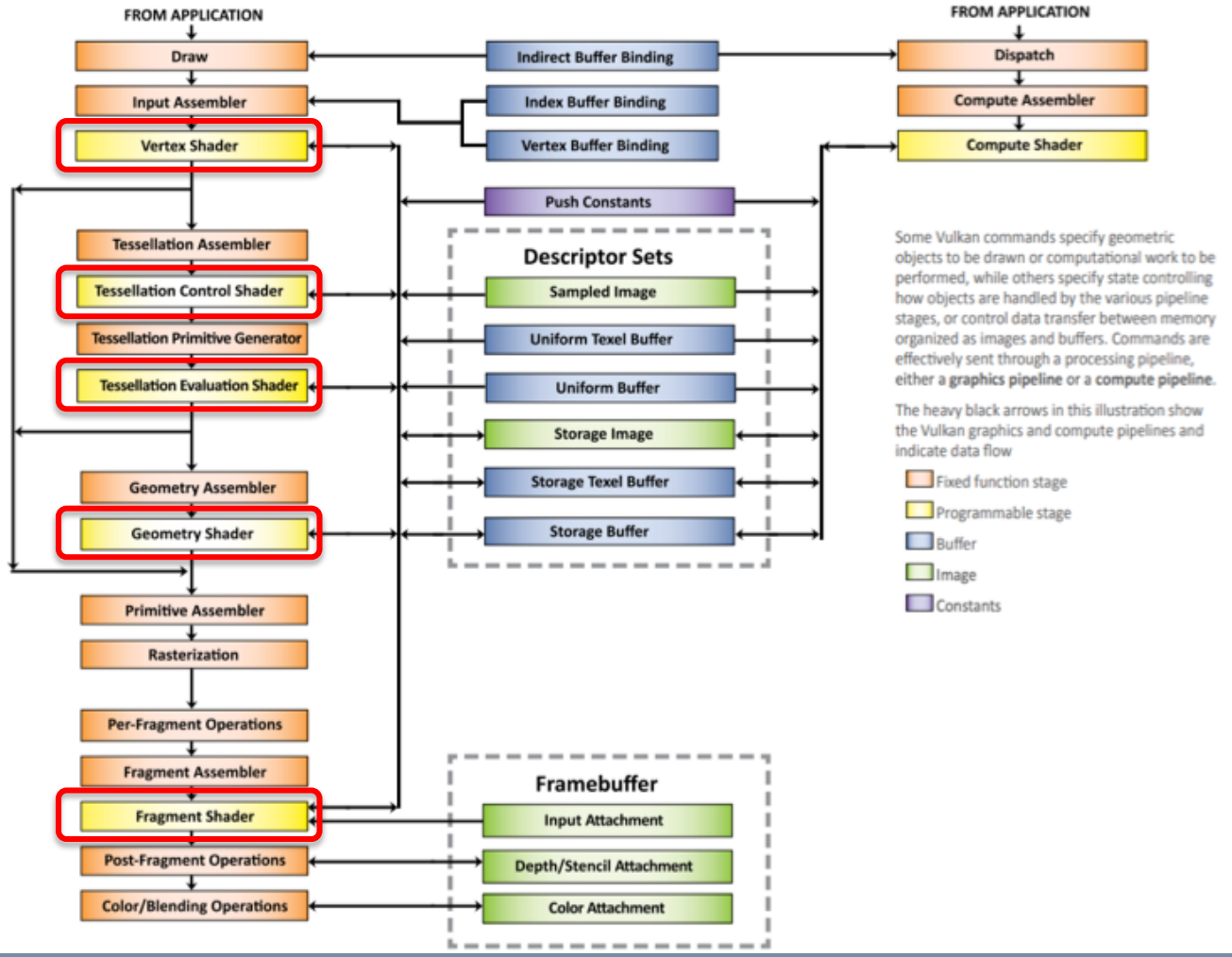
The graphics pipeline

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



The graphics pipeline

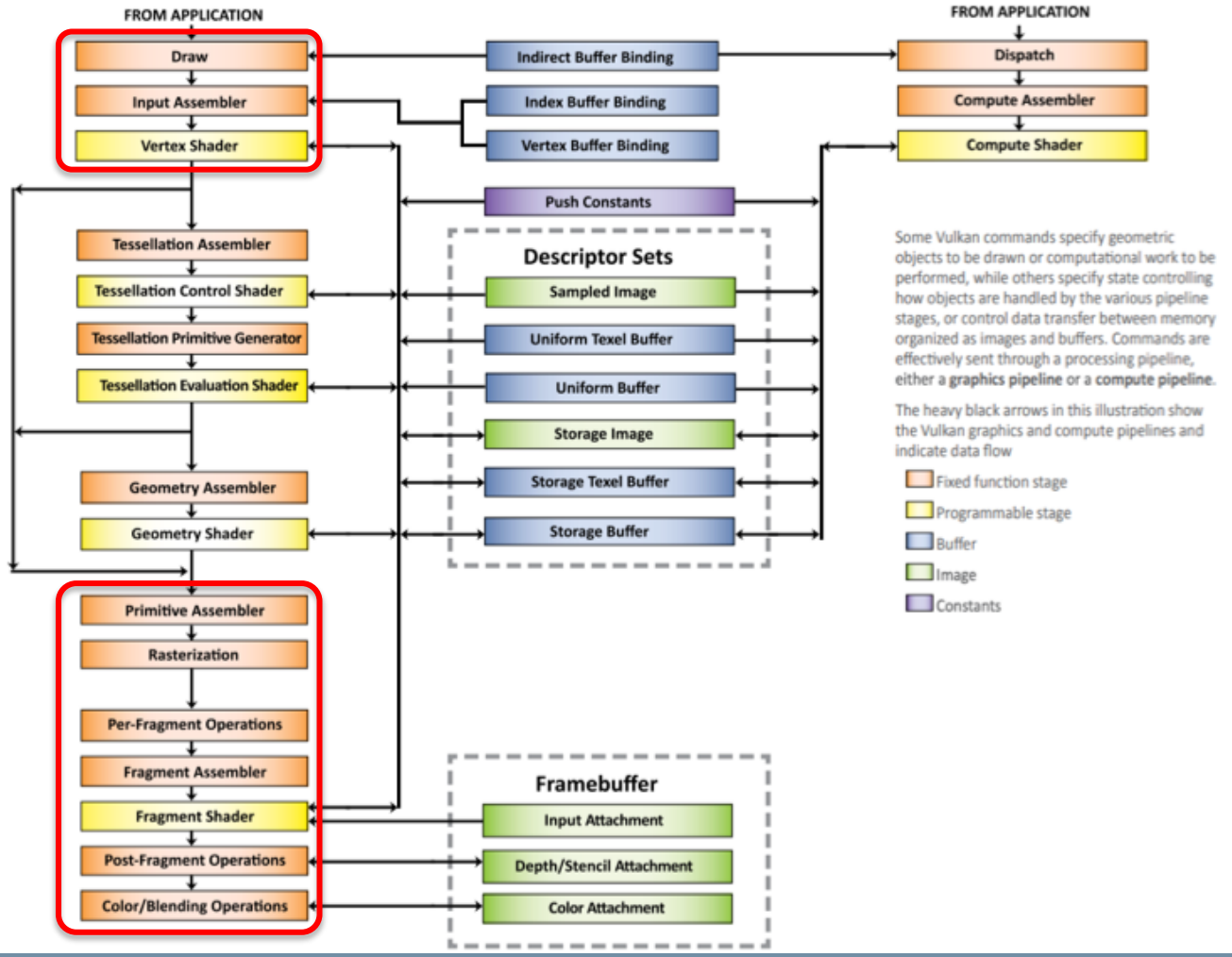
Up to five different types of shaders can be used to define the functions of the programmable stages of the pipeline.



The graphics pipeline

Only the initial and final stages are generally required.

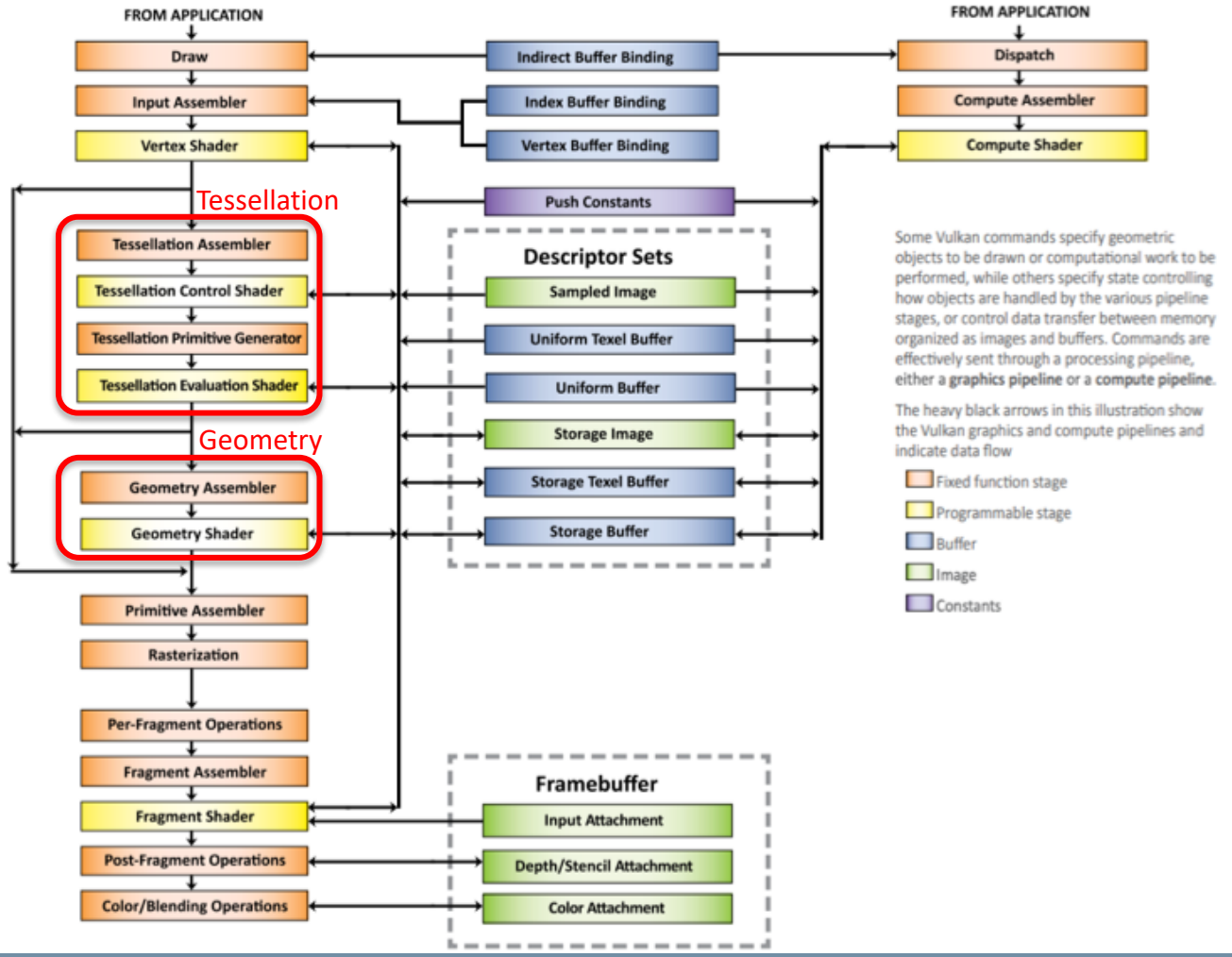
This means that in most of the cases, only the *Vertex* and the *Fragment* shaders are required to generate an image.



The graphics pipeline

Tessellation and geometry stages are optional.

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

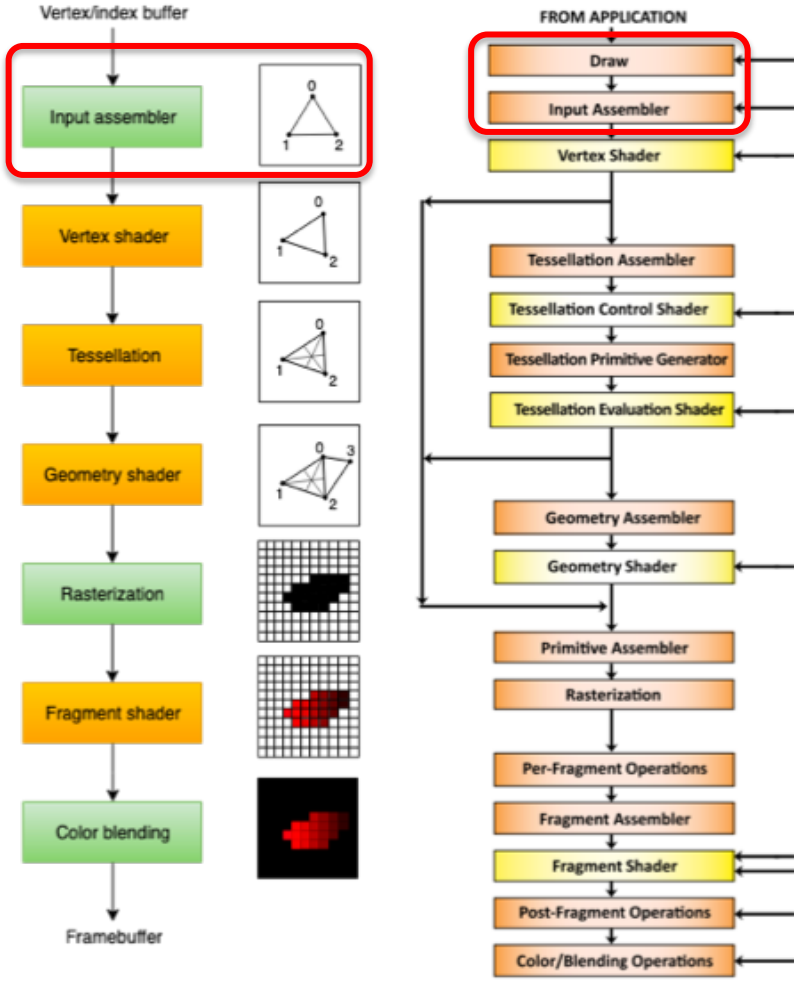


The graphics pipeline

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

If several instances (copies – we will return on this later) of the same object are required, 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.

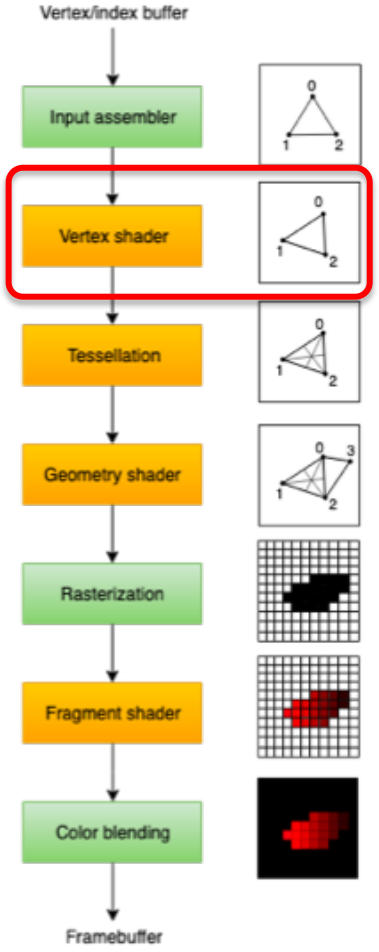


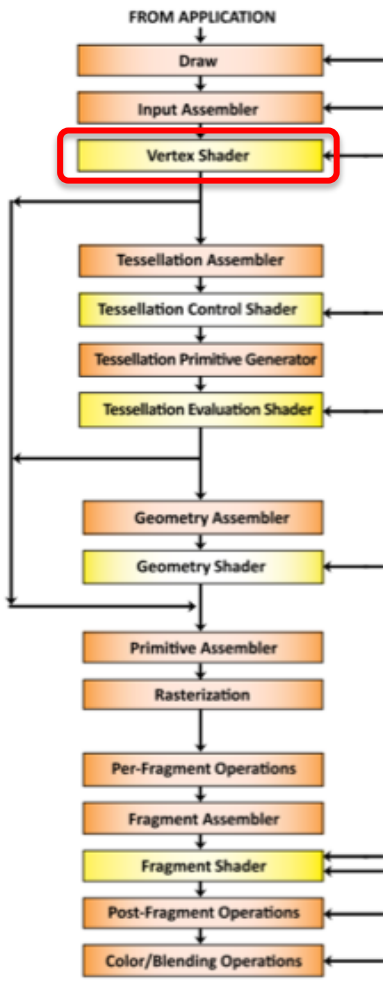
From: https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction

The graphics pipeline

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

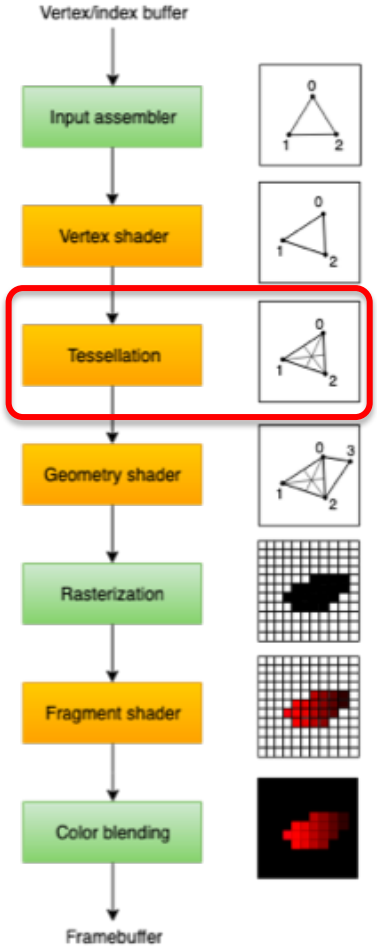
Such operations, for example, transform local coordinates to clipping coordinates by multiplying vertex positions with the corresponding WVP matrix, or compute colors and other values associated to vertices, which will be used in later stages of the process.

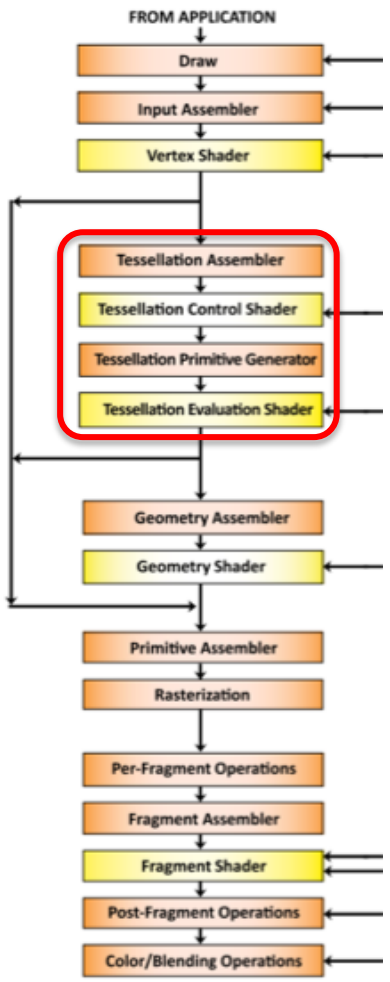




The graphics pipeline

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

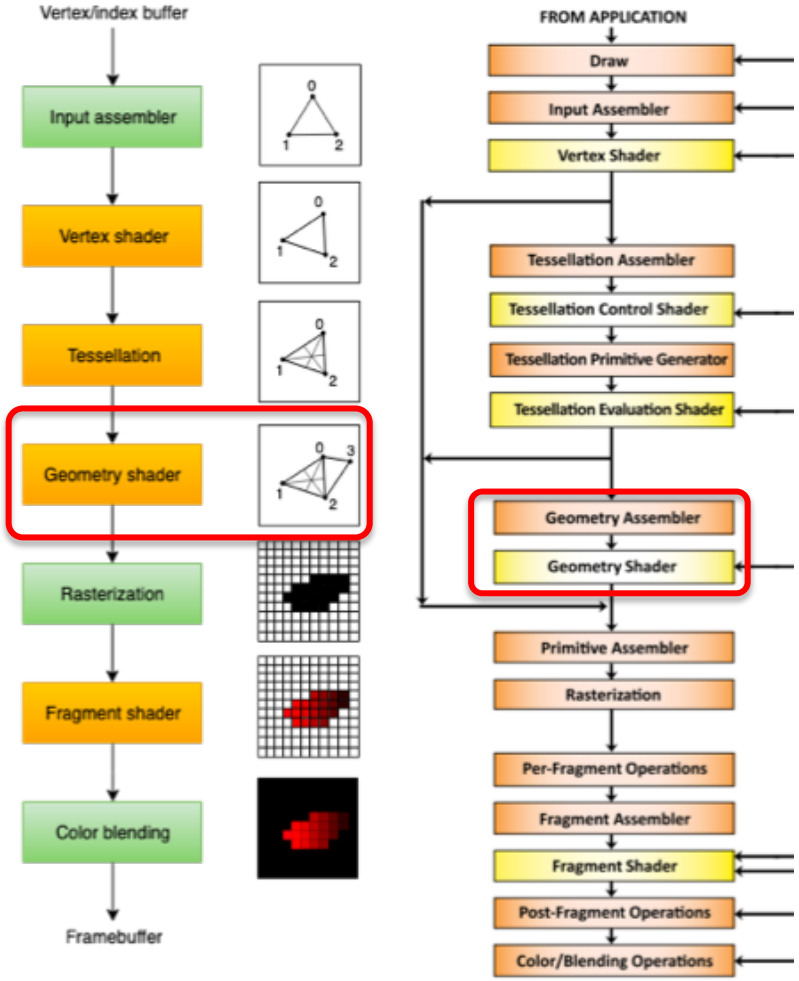




The graphics pipeline

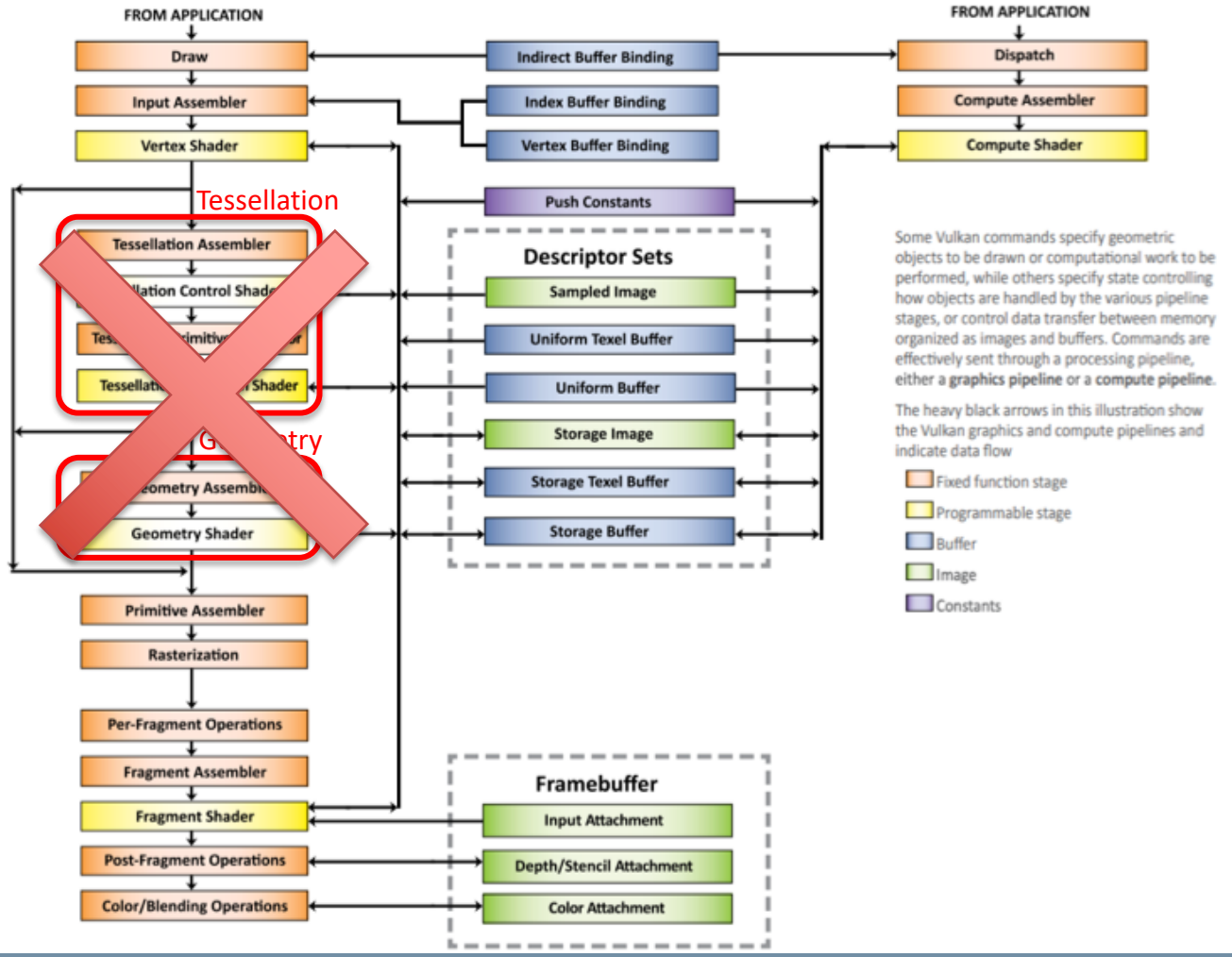
Geometry shaders can remove or add primitives to the stream, starting from the previously generated elements.

In principle, it could perform the same tasks as Tessellation stages: however, due to its generality, it would result in more complex code to be written and slower performance.



The graphics pipeline

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

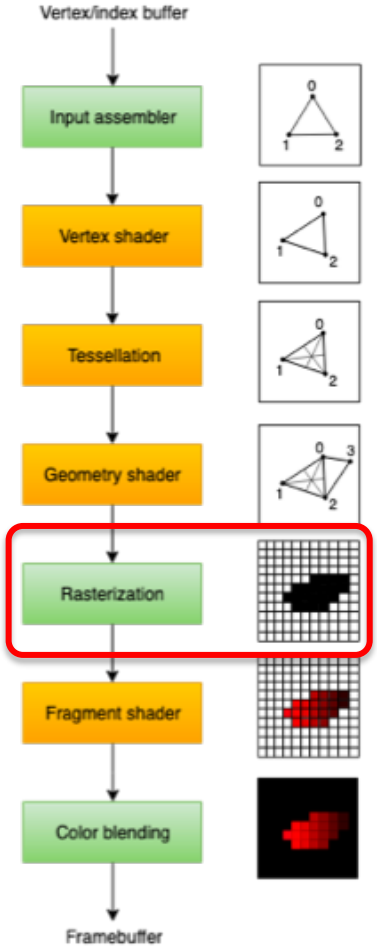


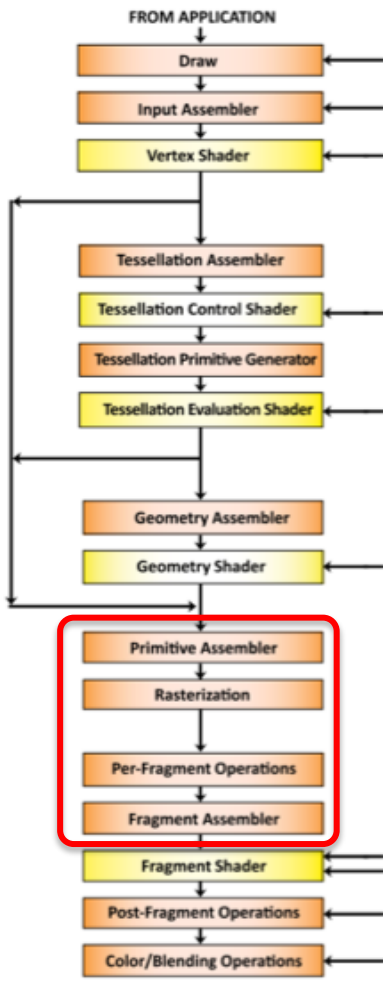
The graphics pipeline

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

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

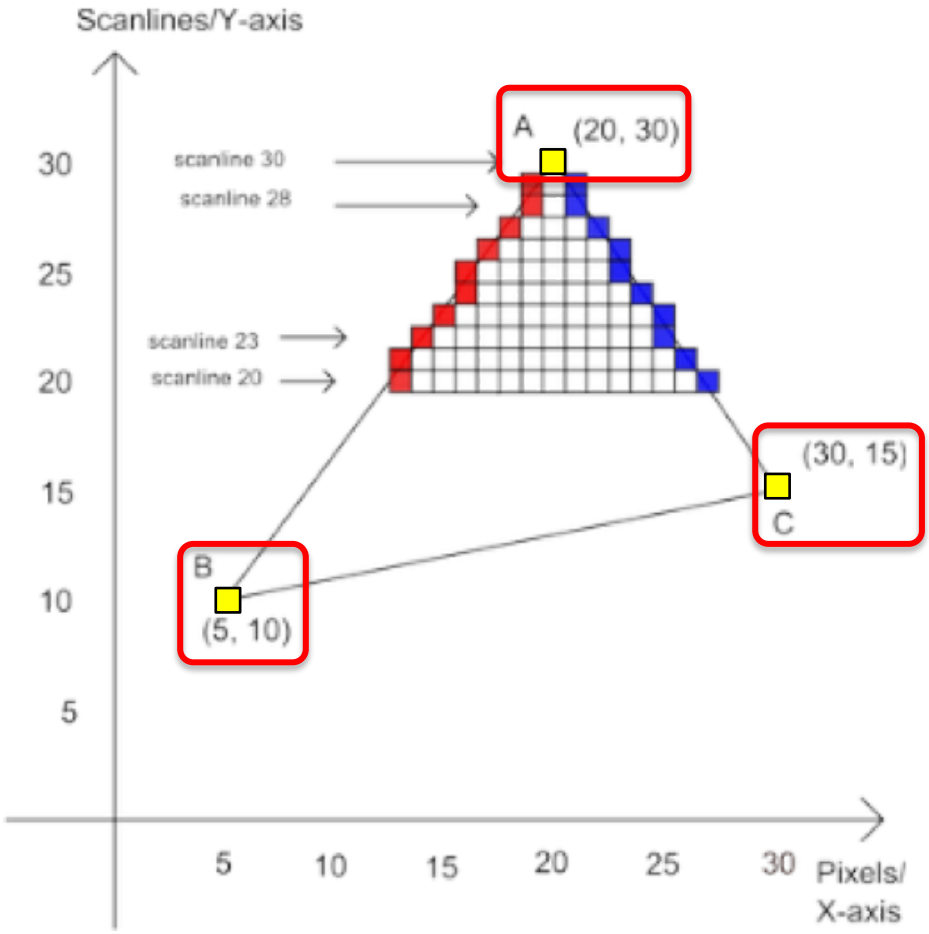
In these stages, the “division by w” to transform clipping coordinates into normalized screen coordinates is also performed.





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.

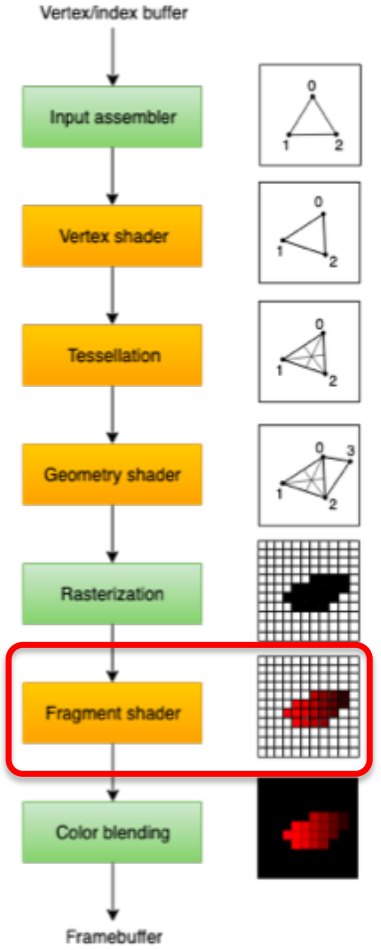


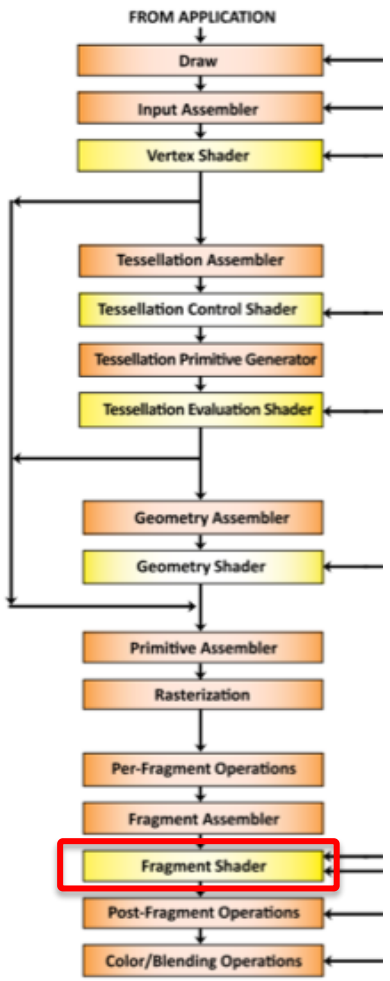
The graphics pipeline

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 technique to produce either realistic or effective images.

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

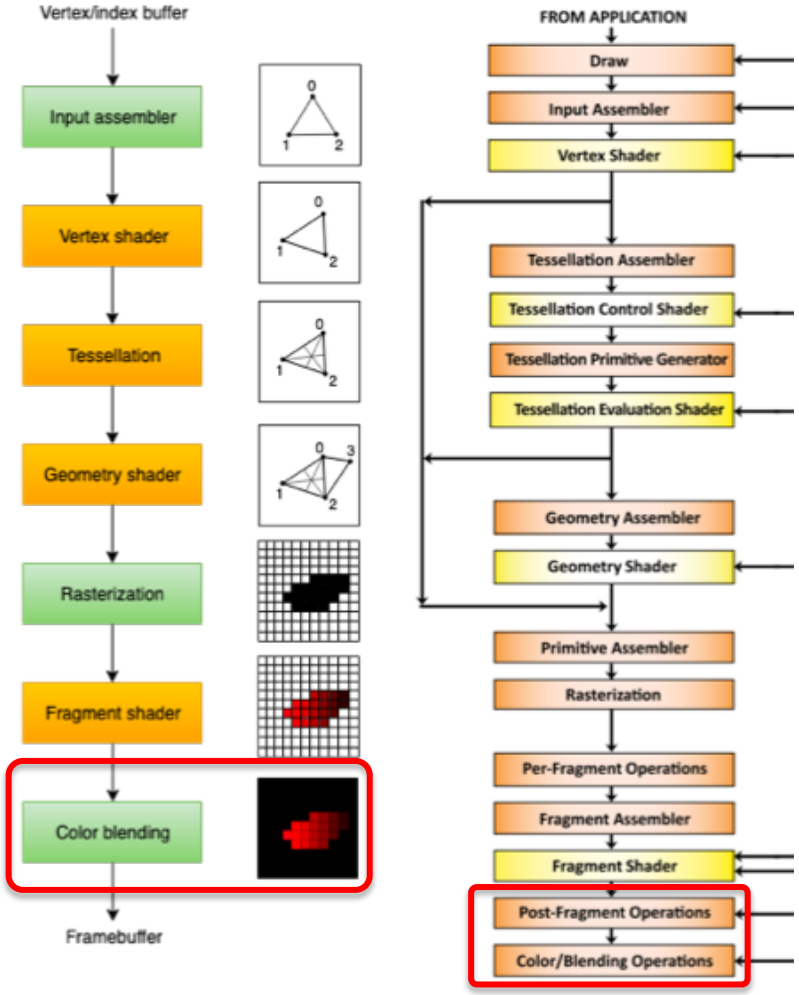




The graphics pipeline

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

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



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.

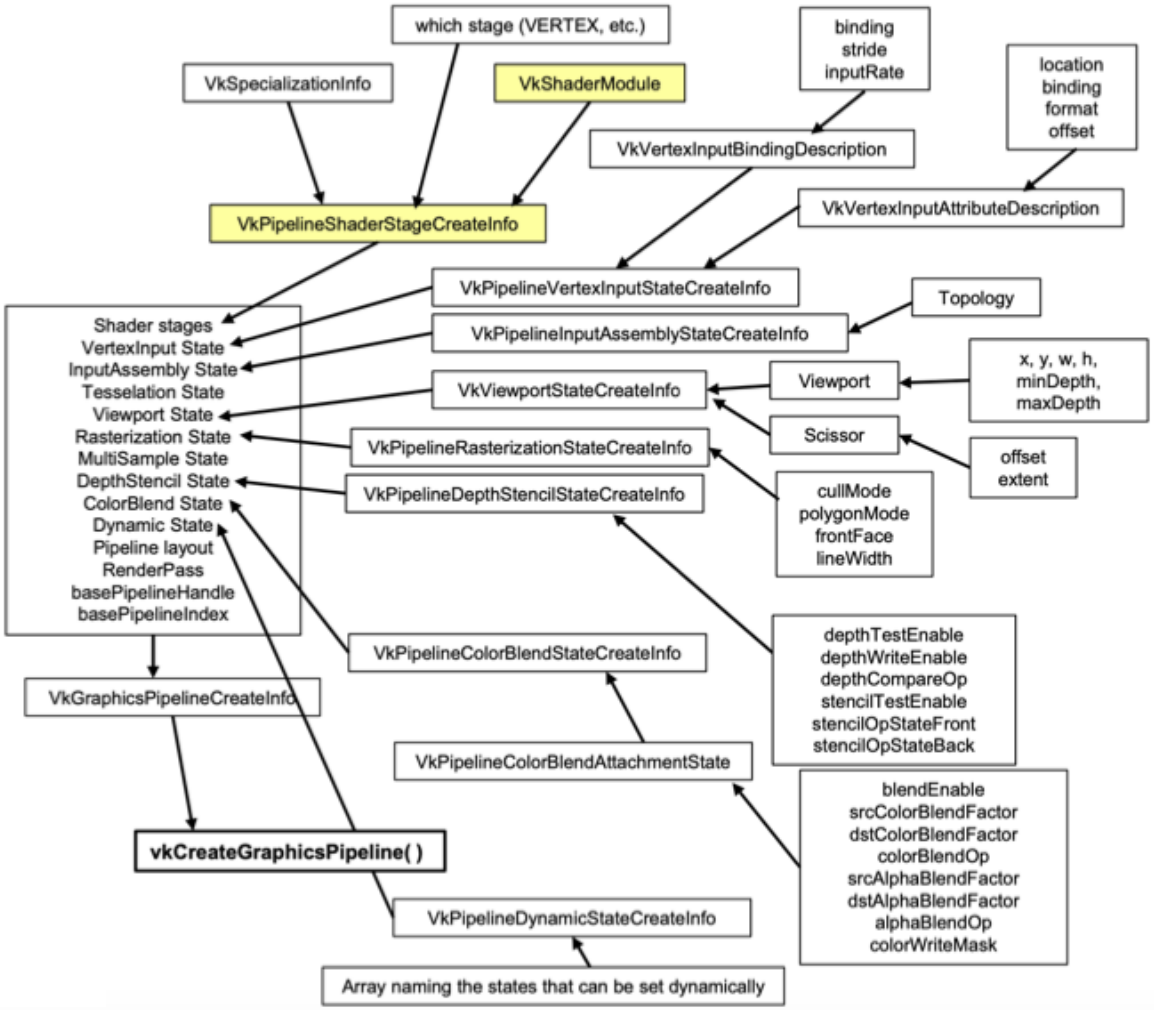
For example, Assignment 0 has two pipelines (one for the scene, and one for the background), and Assignment 3 has four (the previous two, plus one for the wireframe rendering of their starships, and one for the text overlay).

The creation of one pipeline is also the most complex parts of a 3D application: in the assignments, it requires 185 lines of code to 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.



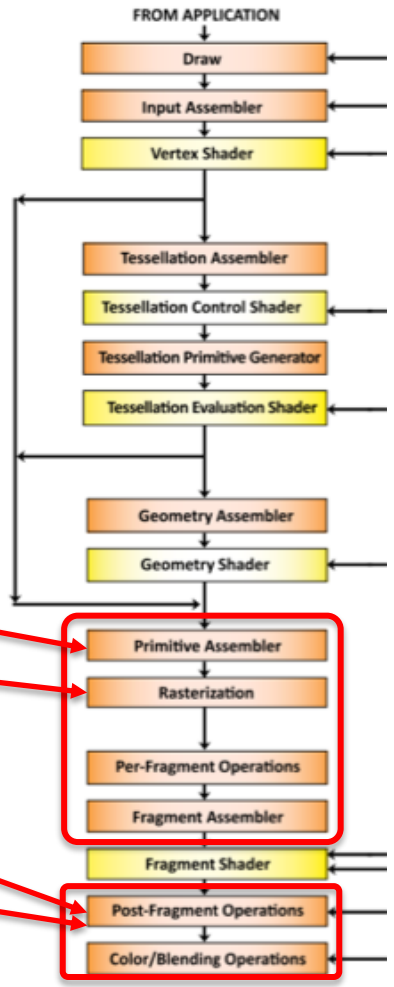
From: https://web.engr.oregonstate.edu/~mjb/vulkan/

Fixed functions

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

We will enter in detail in the following functions:

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



Clipping

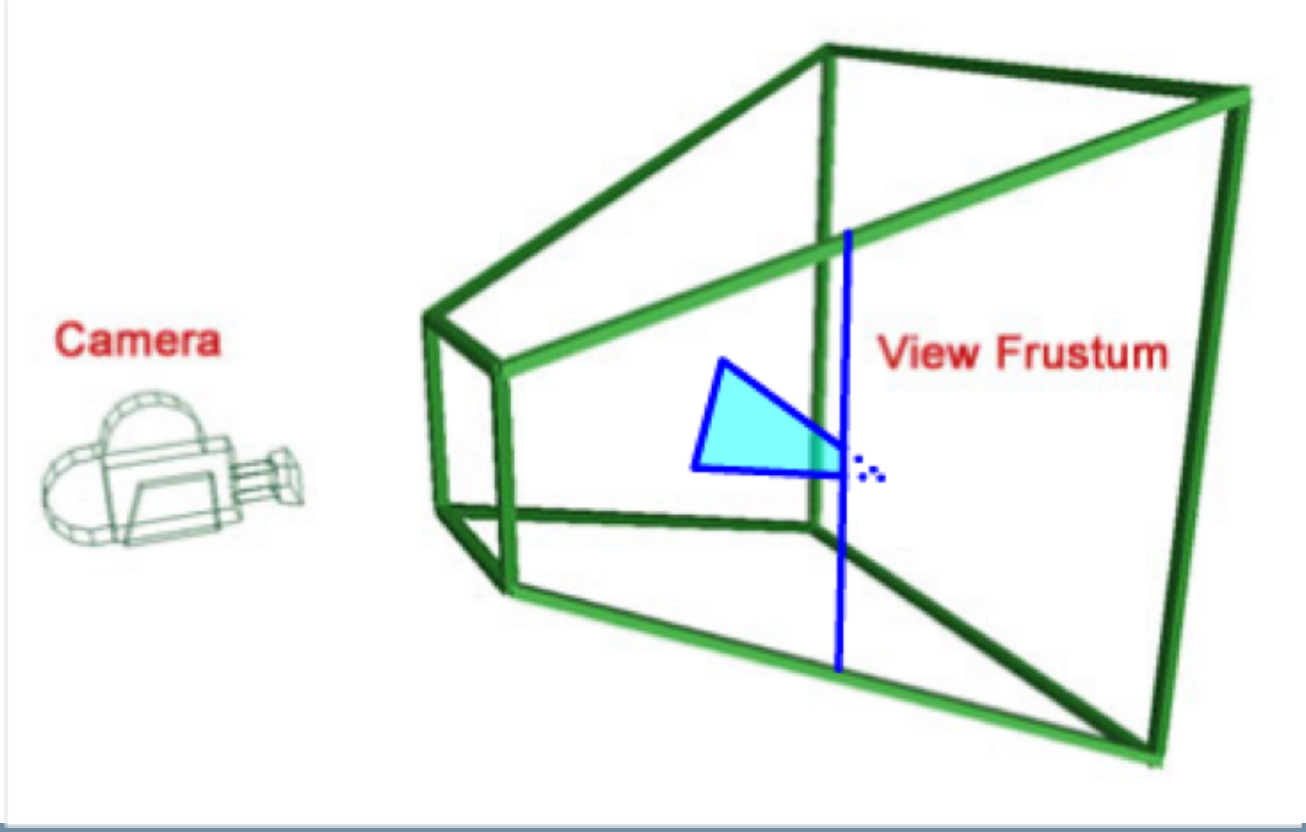
The triangles of a mesh can intersect the screen boundaries and can be only partially shown.

As we have briefly introduced, *clipping* is the process of truncating and eliminating parts of graphics primitives to make them entirely contained on the screen.

Clipping is usually performed after the projection transform, but before normalization (division by w). For this reason, the space in which it is performed is called *Clipping Coordinates*.

Clipping

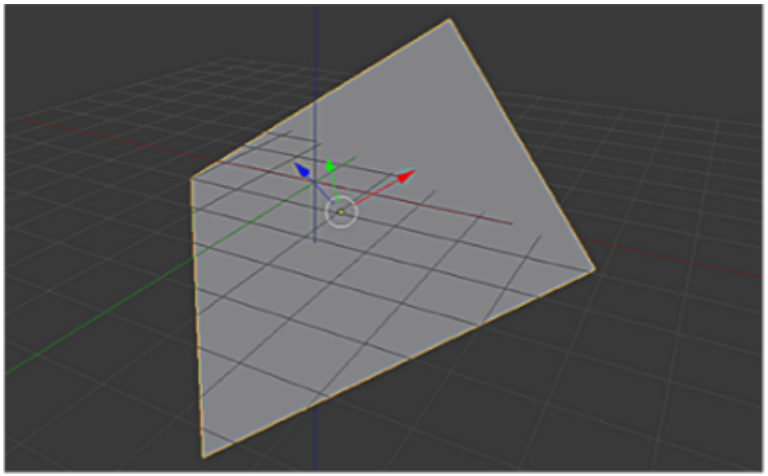
In 3D, clipping is performed against the viewing frustum.



Clipping: half spaces

Let us recall (from the geometry course) the equation of a plane:

$$n_x \cdot x + n_y \cdot y + n_z \cdot z + d = 0$$

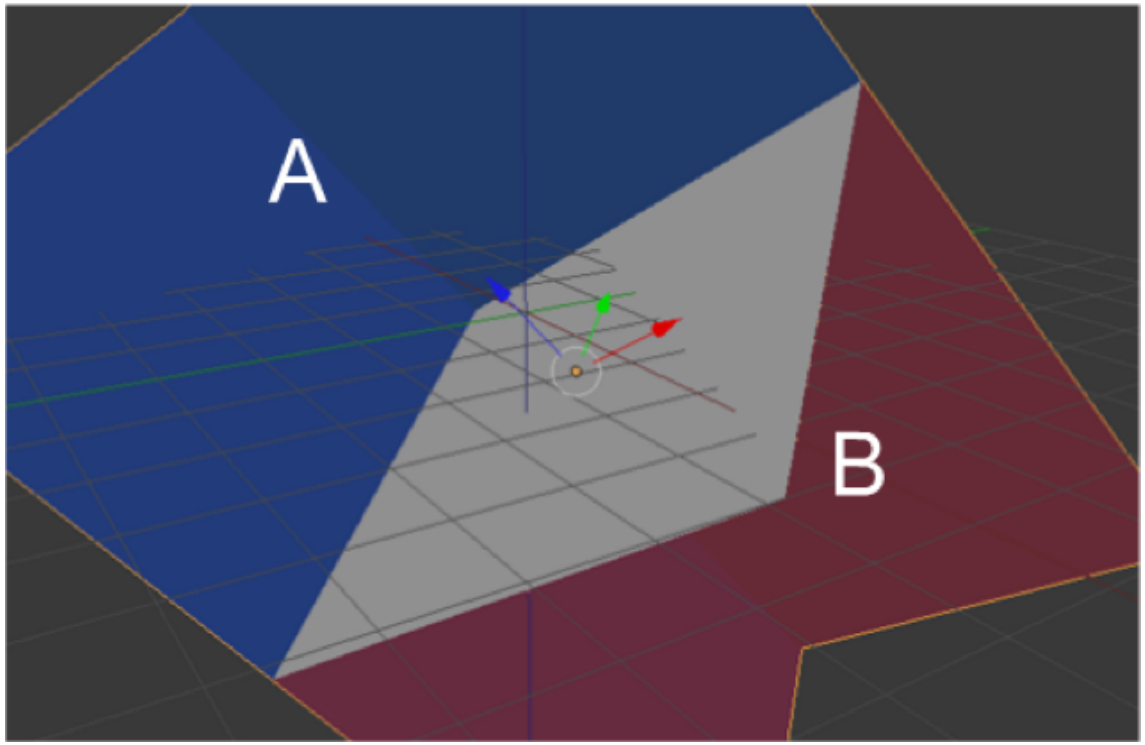


Components n_x, n_y, n_z represent the direction of the normal vector to the plane.

The constant term *d*, defines the distance from the origin (which is zero if the plane passes through it).

Clipping: half spaces

The equation divides the space into two regions, called *half-spaces*.



Clipping: half spaces

If we use homogeneous coordinates, we can identify a plane with a four components vector *n*.

In this way, the plane equation becomes a scalar product between the homogeneous coordinates of the point and the vector representing the plane.

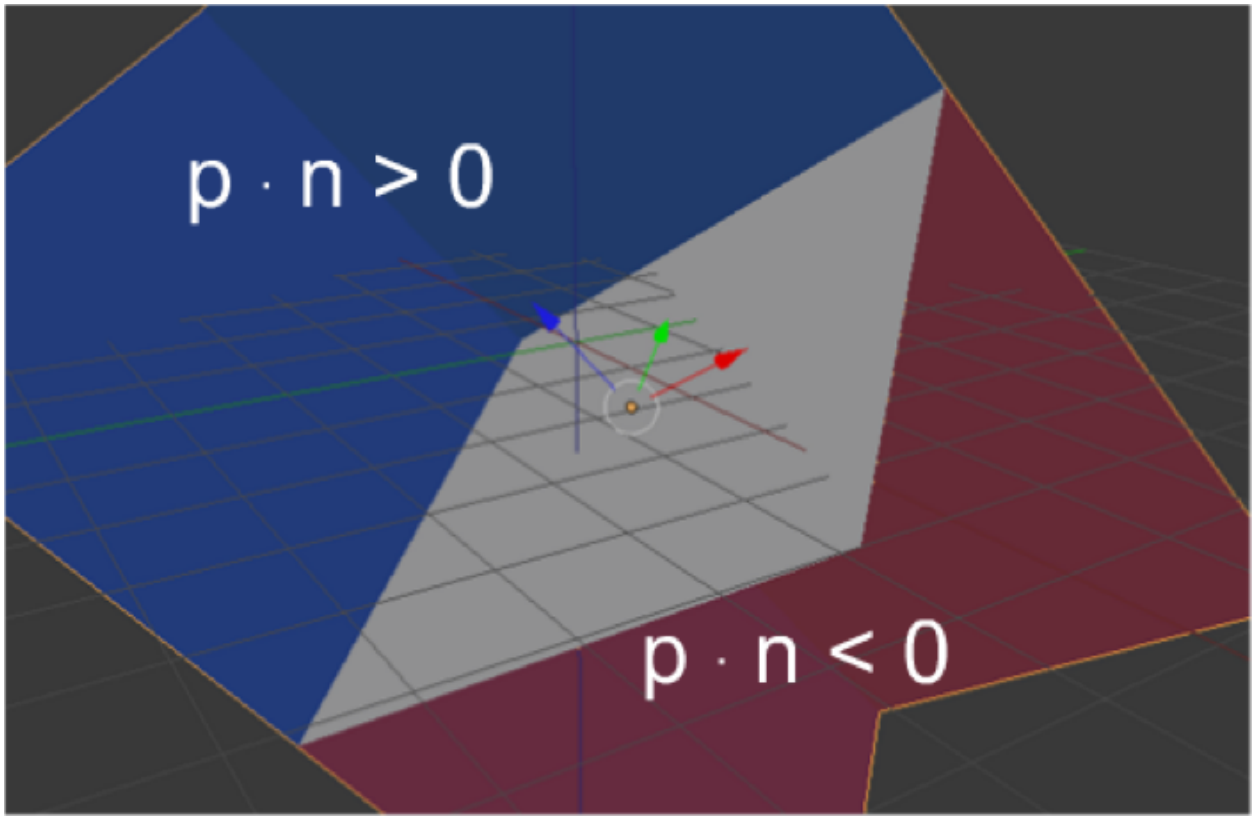
$$n = (n_x, n_y, n_z, d)$$

$$p = (x, y, z, 1)$$

$$n_x \cdot x + n_y \cdot y + n_z \cdot z + d = 0 \quad \Rightarrow \quad \boxed{n \cdot p = 0}$$

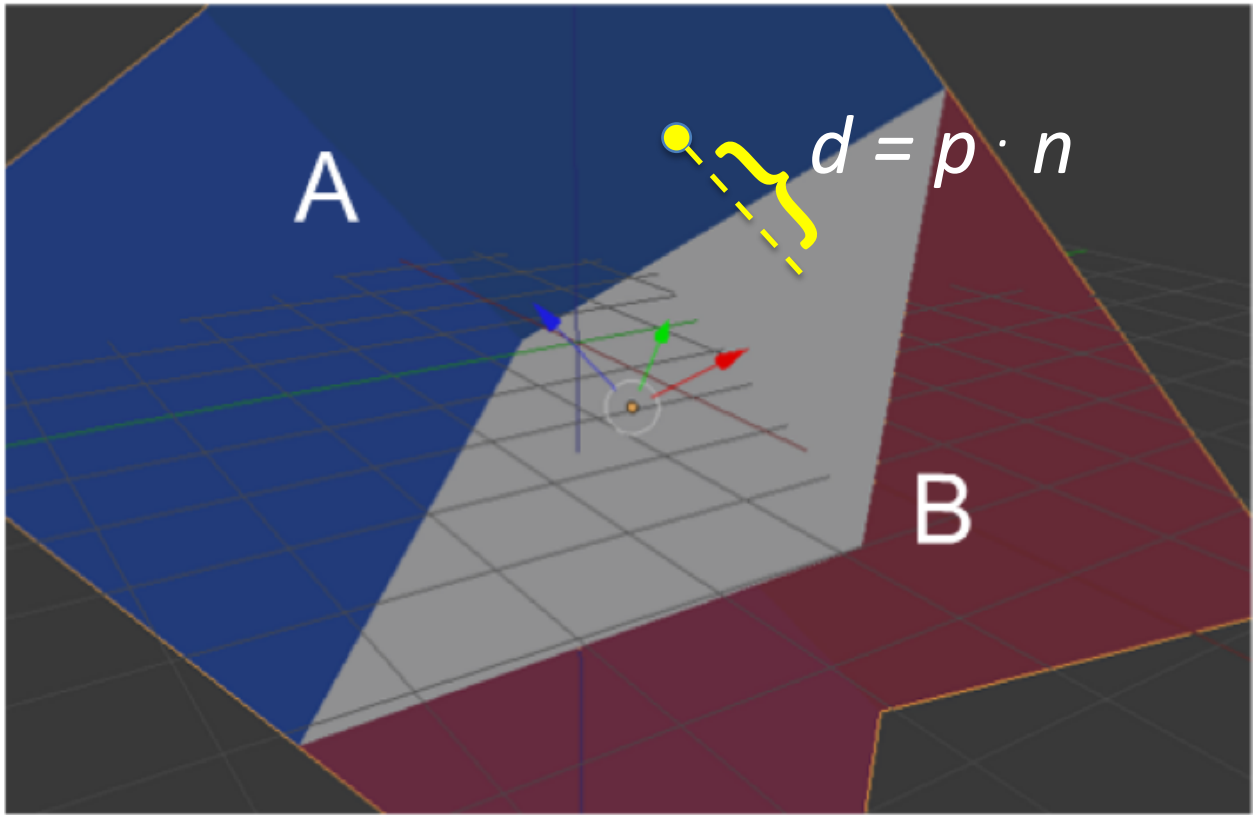
Clipping: half spaces

A point is in either of the two half-spaces depending on the sign of the scalar product of *p* and *n*.



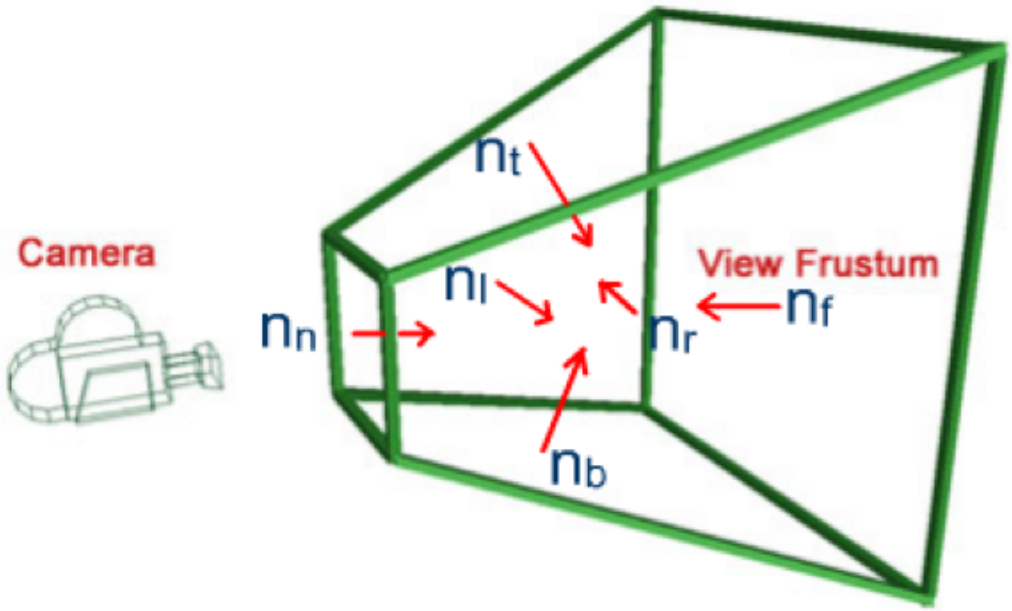
Clipping: half spaces

Note also the the result of the product is the signed distance of the point from the plane dividing the half spaces.



Clipping: half spaces

Since a frustum is a convex solid, it can be defined as the intersection of the half-spaces delimited by its six faces.



Clipping: half spaces

Clipping is performed in “clipping space”, and coordinates are meant to be inside the frustum if between -1 and 1, the six vectors are the following:

$$\frac{x}{w} > -1, \quad x > -w, \quad x + w > 0, \quad n_l = |1 \quad 0 \quad 0 \quad 1|$$

$$\frac{x}{w} < 1, \quad x < w, \quad -x + w > 0, \quad n_r = |-1 \quad 0 \quad 0 \quad 1|$$

$$\frac{y}{w} > -1, \quad y > -w, \quad y + w > 0, \quad n_b = |0 \quad 1 \quad 0 \quad 1|$$

$$\frac{y}{w} < 1, \quad y < w, \quad -y + w > 0, \quad n_t = |0 \quad -1 \quad 0 \quad 1|$$

$$\frac{z}{w} > 0, \quad z > 0, \quad n_n = |0 \quad 0 \quad 1 \quad 0|$$

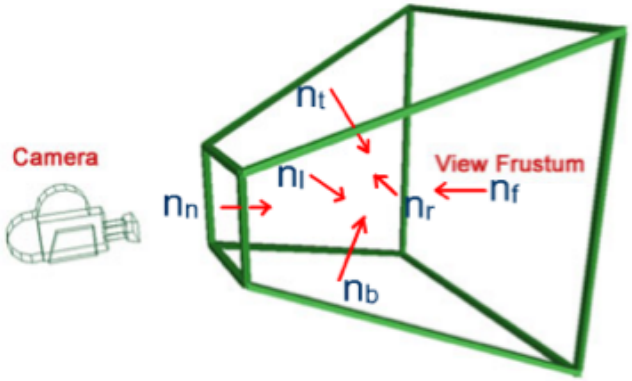
$$\frac{z}{w} < 1, \quad z < w, \quad -z + w > 0, \quad n_f = |0 \quad 0 \quad -1 \quad 1|$$

Clipping: points

We can determine if a point is inside the frustum by performing the scalar products of its (homogeneous) coordinates with the six normal vectors.

In particular, the point is inside the frustum if all the products are positive.

A clipping algorithm can thus discard a point *p* if the product with at least one of the normal vectors to the frustum is negative.



$$p \cdot n_v > 0$$

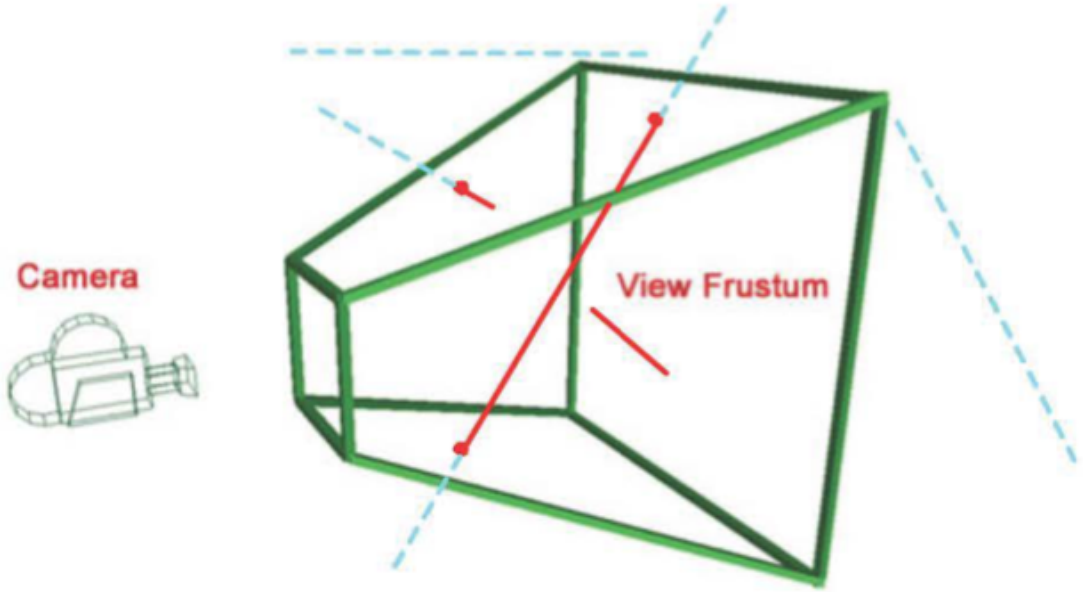
$$\forall v \in \{l, r, t, b, n, f\}$$

Clipping: lines

A line can either be totally inside, totally outside or intersecting the frustum.

The first two cases are trivial, and are solved either considering the segment as is, or discard it completely.

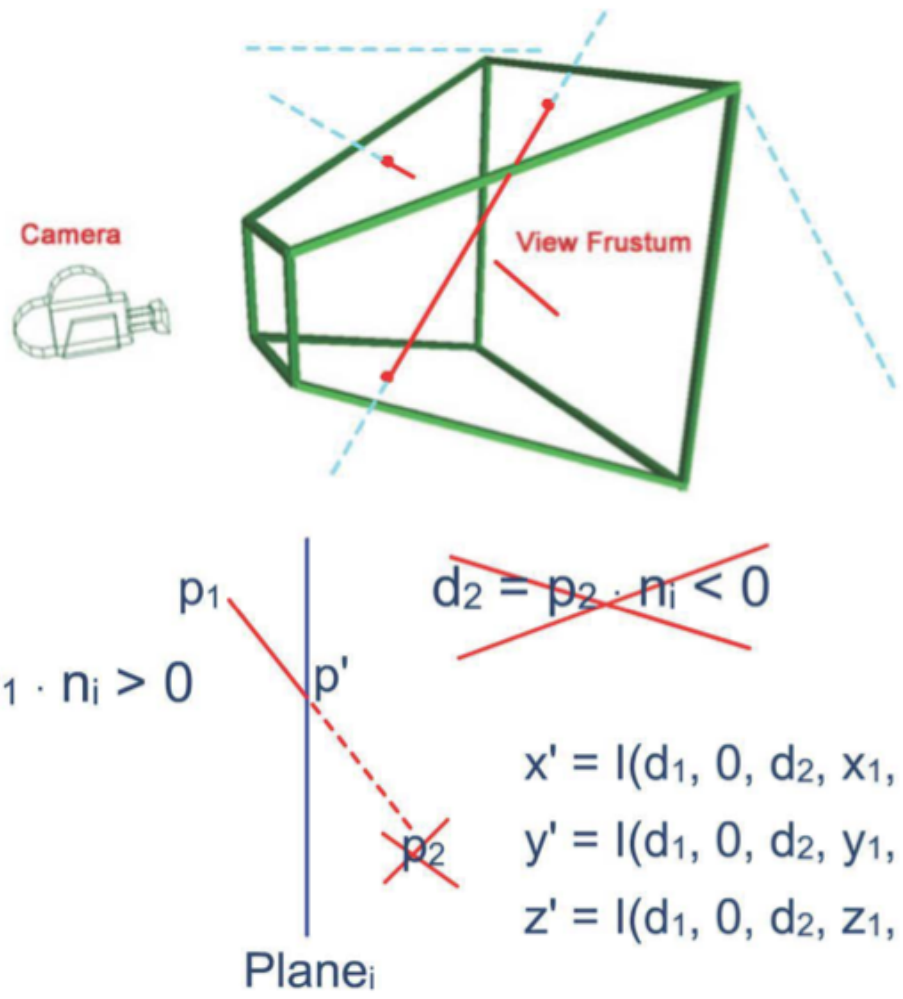
Intersecting lines should instead be “cut” to transform them in segments entirely contained in the frustum.



Clipping: lines

Intersecting a line, means changing one or both of its starting and ending points, with one at the intersection with the frustum.

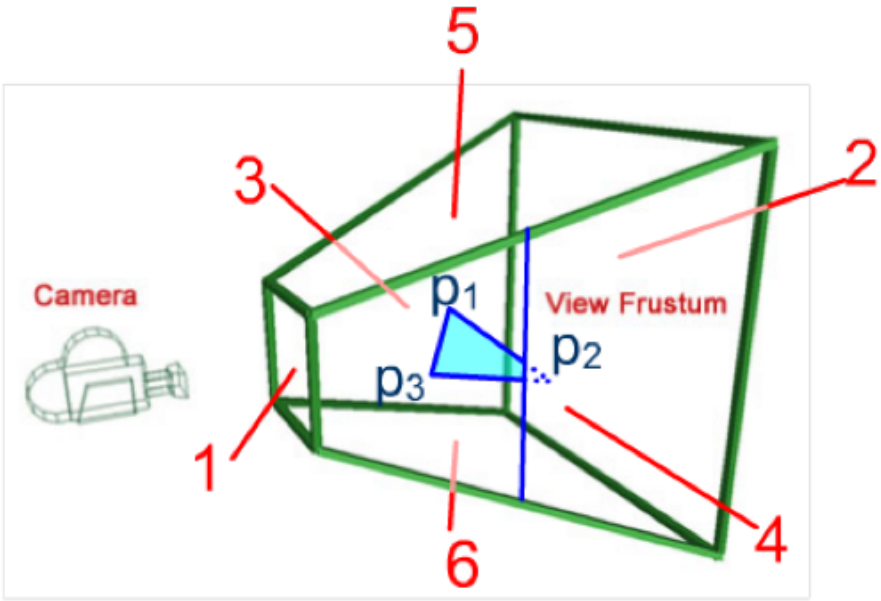
This can be done using the distance of the points from the clipping plane and performing interpolatior



Clipping: triangles

For triangles things are slightly more complex, and many different algorithms have been proposed.

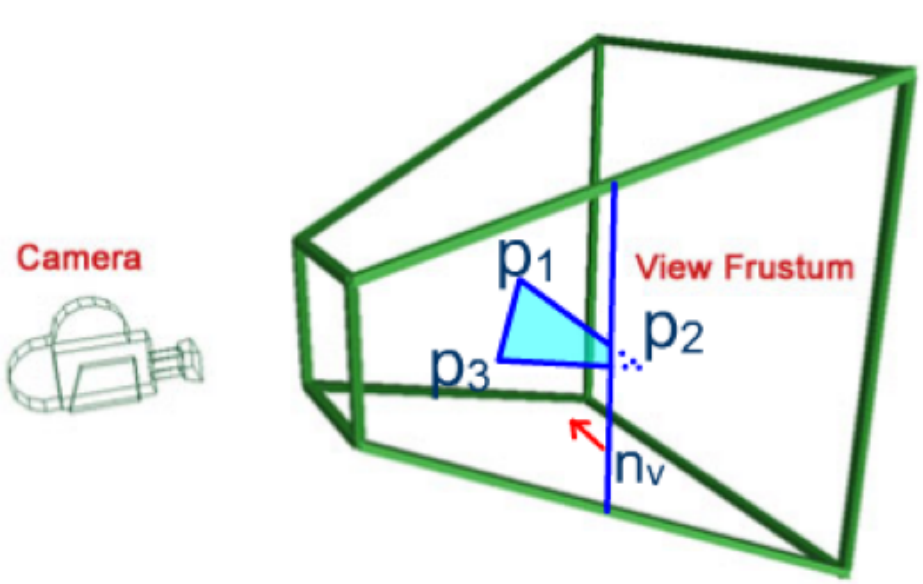
We will briefly introduce one of the most popular.



Clipping: triangles

Let us focus on a side whose normal vector is n_v.

The distance from the plane d_1, d_2 and d_3 are computed by performing the scalar product of the three coordinates of vertices p_1, p_2 and p_3 with the normal vector to the plane n_v.



$$d_1 = p_1 \cdot n_v$$
$$d_2 = p_2 \cdot n_v$$
$$d_3 = p_3 \cdot n_v$$

Clipping: triangles

We have a *trivial reject*, or a *trivial accept for the side* if the three distances have all the same sign.

In case of reject (all negative), the algorithm stops, since the triangle is outside the frustum.

For accept (all positive), the iteration continues with the next plane.

$$d_1 = p_1 \cdot n_v > 0$$
$$d_2 = p_2 \cdot n_v > 0$$
$$d_3 = p_3 \cdot n_v > 0$$

Trivial accept (for the side)

$$d_1 = p_1 \cdot n_v < 0$$
$$d_2 = p_2 \cdot n_v < 0$$
$$d_3 = p_3 \cdot n_v < 0$$

Trivial reject

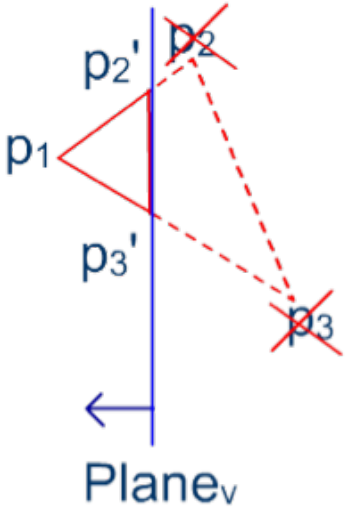
Clipping: triangles

If two points are outside the frustum, say p_2 and p_3, then two intersections p_2' and p_3' are computed with interpolation.

The distance from the plane d_2 and d_3 are used as interpolation coefficients.

The two vertices p_2 and p_3 are replaced by p_2' and p_3' .

$$d_1 = p_1 \cdot n_v > 0$$
$$d_2 = p_2 \cdot n_v < 0$$
$$d_3 = p_3 \cdot n_v < 0$$

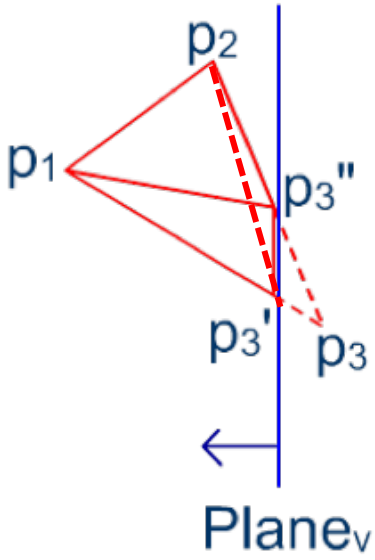


Clipping: triangles

If just one point is outside, say p_3, two intersections p_3' and p_3'' on the segment that connects it to p_1 and p_2 are computed.

In this case two triangles are produced. Two alternatives are possible, and usually the choice is arbitrary.

$$d_1 = p_1 \cdot n_v > 0$$
$$d_2 = p_2 \cdot n_v > 0$$
$$d_3 = p_3 \cdot n_v < 0$$

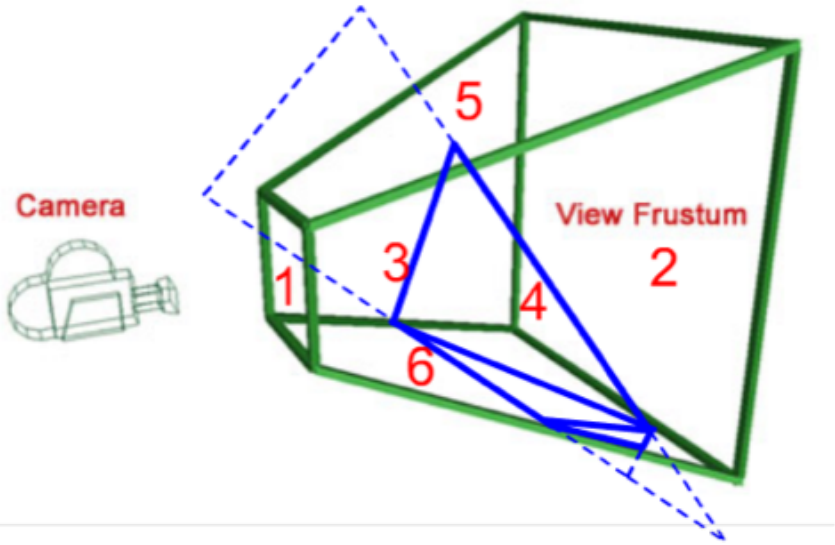


Clipping: triangles

If the triangle is not rejected, it is clipped against the next plane.

If two triangles have been produced, they are considered separately.

The algorithm terminates either when all triangles have been checked with all the sides of the frustum, or when all the triangles have been rejected.



Clipping: triangles

The algorithm is very simple, but it can produce a large number of triangles, since potentially they can double at every check.

This has also implications on the data structure required to store the triangles, since it must be able to accommodate a variable number of figures.

Moreover, computing the intersection is usually very complex, since it must take into account many parameters assigned to vertices that will be detailed in the following lessons (e.g. normal vectors, colors and UV coordinates).

Back-face culling

Back-face culling can exclude the faces that belong to the backside of a mesh simply by checking whether triangle vertices are ordered clockwise or counterclockwise.

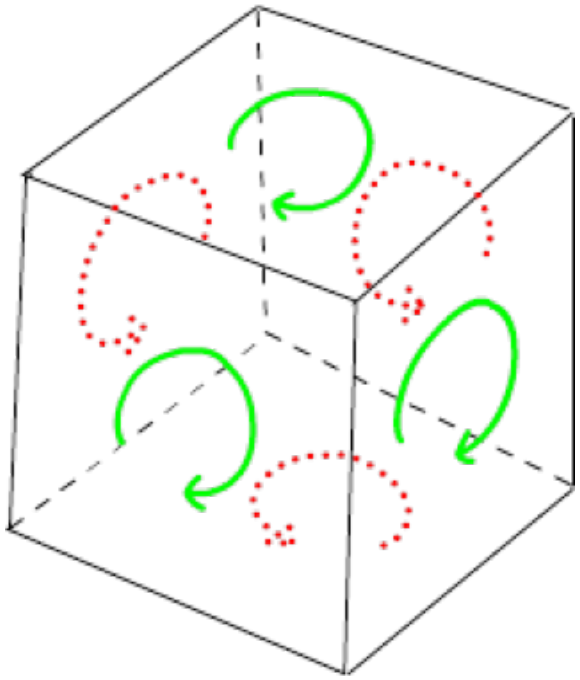
The check can be performed either before the projection of the triangle or using the normalized screen coordinates.

Vulkan implements back-face culling in normalized screen coordinates: we will thus focus only on this implementation.

Back-face culling

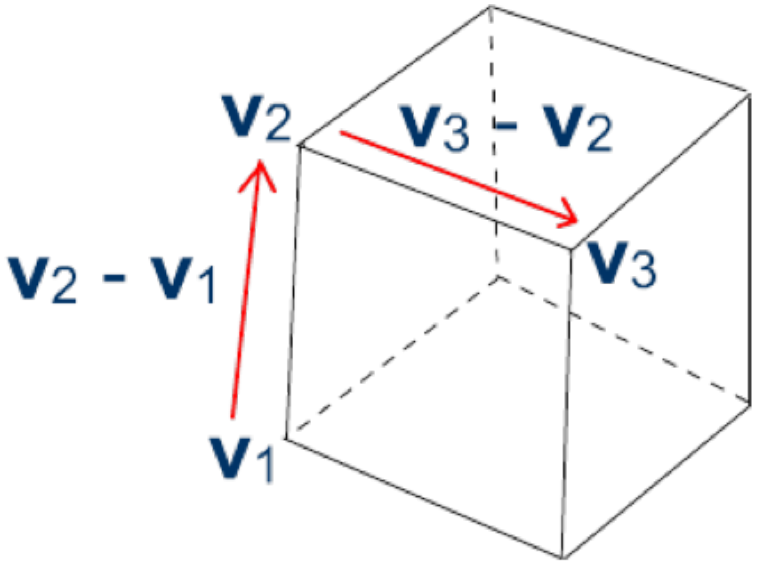
Let us suppose that all the triangles of a mesh are encoded using a consistent orientation, for example clockwise.

Once projected on screen, front faces will still be ordered clockwise, while back faces will be ordered in the opposite direction.

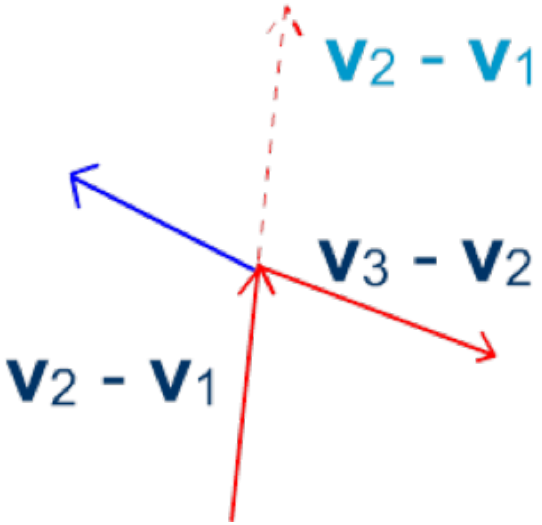


Back-face culling

The orientation of the vertices of a triangle in normalized screen coordinates can be computed with a simple cross-product: if the result vector is oriented toward the viewer *(positive z component)*, then the vertices are ordered clockwise.



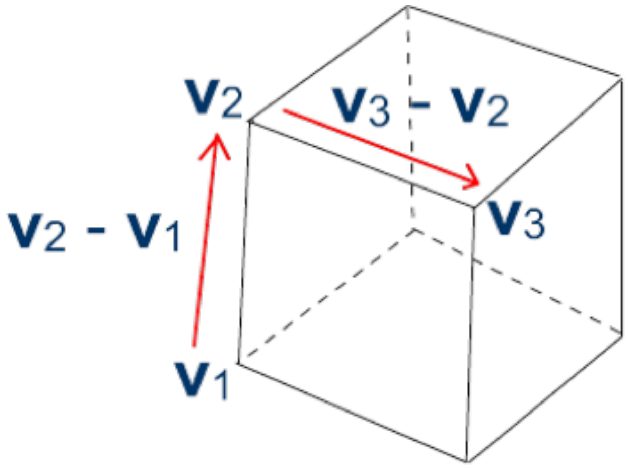
$$(v_3 - v_2) \times (v_2 - v_1)$$



Back-face culling

Since only the sign of the *z* component is required, the test can be performed in a very efficient way.

$$(v_{3.x} - v_{2.x})(v_{2.y} - v_{1.y}) \geq (v_{3.x} - v_{2.x})(v_{2.y} - v_{1.y}) \Longrightarrow CW$$



$$(v_3 - v_2) \times (v_2 - v_1)$$

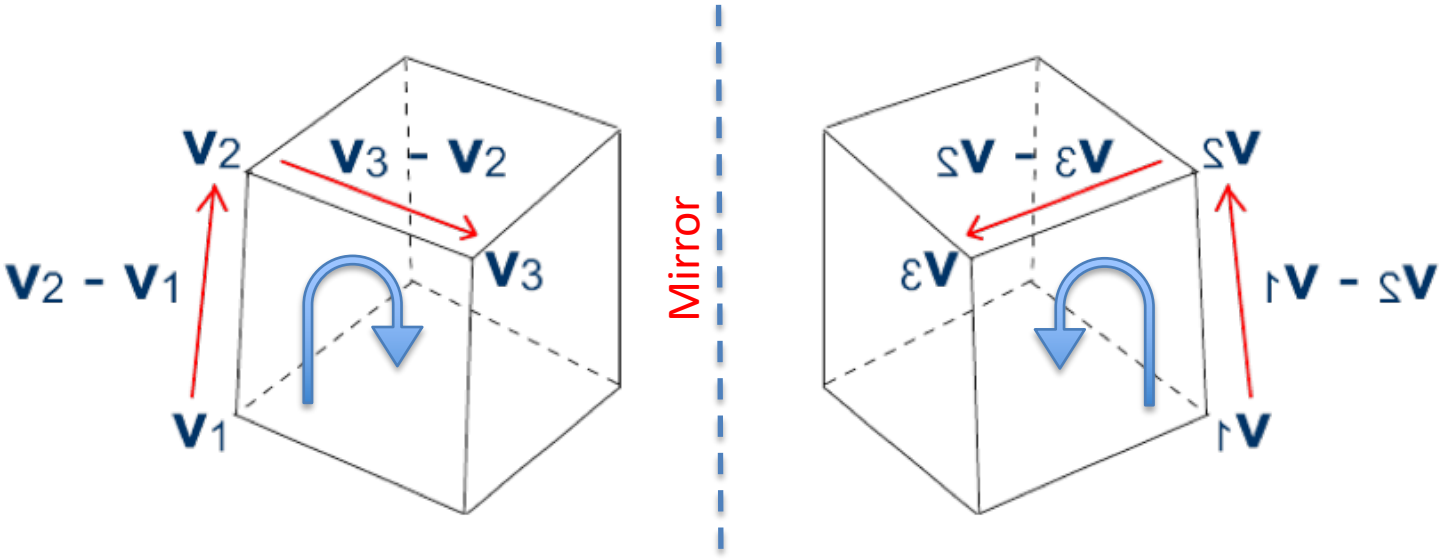
$$|u_x \quad u_y \quad u_z| \times |v_x \quad v_y \quad v_z| = |u_y v_z - u_z v_y \quad u_z v_x - u_x v_z \quad u_x v_y - u_y v_x|$$

Back-face culling

There are however transformations that changes the ordering of the vertices (i.e. mirroring).

Moreover, not all software that produces 3D assets use the same convention for front faces.

For these reasons, the user must be able to specify whether to show faces with vertices ordered clockwise, or in the opposite direction.



Back-face culling: final remarks

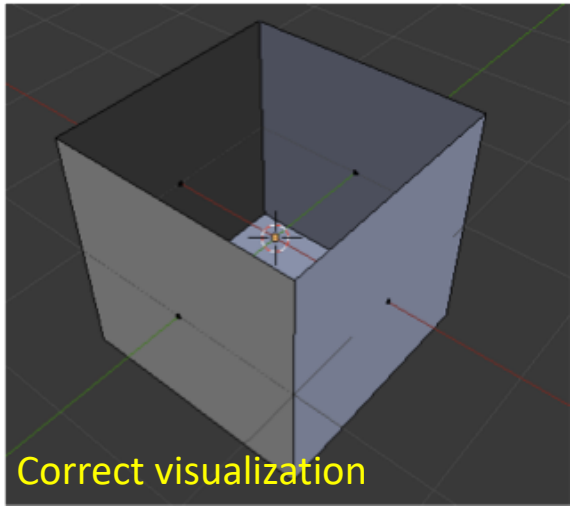
Back-face culling can improve the performances of an application, however there are some potential issues that can arise and that must be considered.

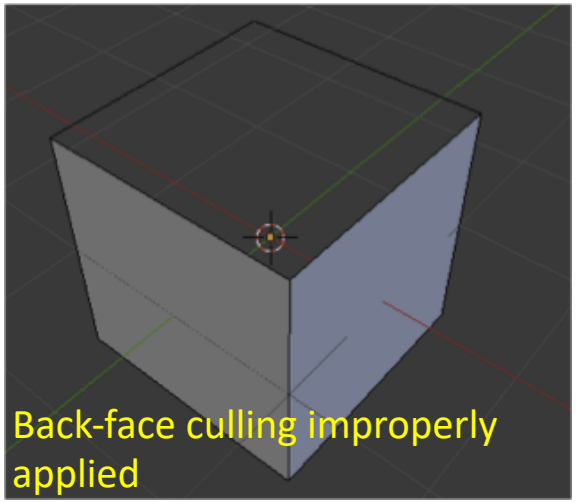
The first is that if a world matrix includes a scaling component with an odd number of negative coefficients (either one or all three), the acceptance test must be inverted.

Back-face culling: final remarks

Back-face culling cannot be always applied.

For example in non 2-manyfold objects, with holes or lamina faces, some polygons belonging to the back faces could be visible.

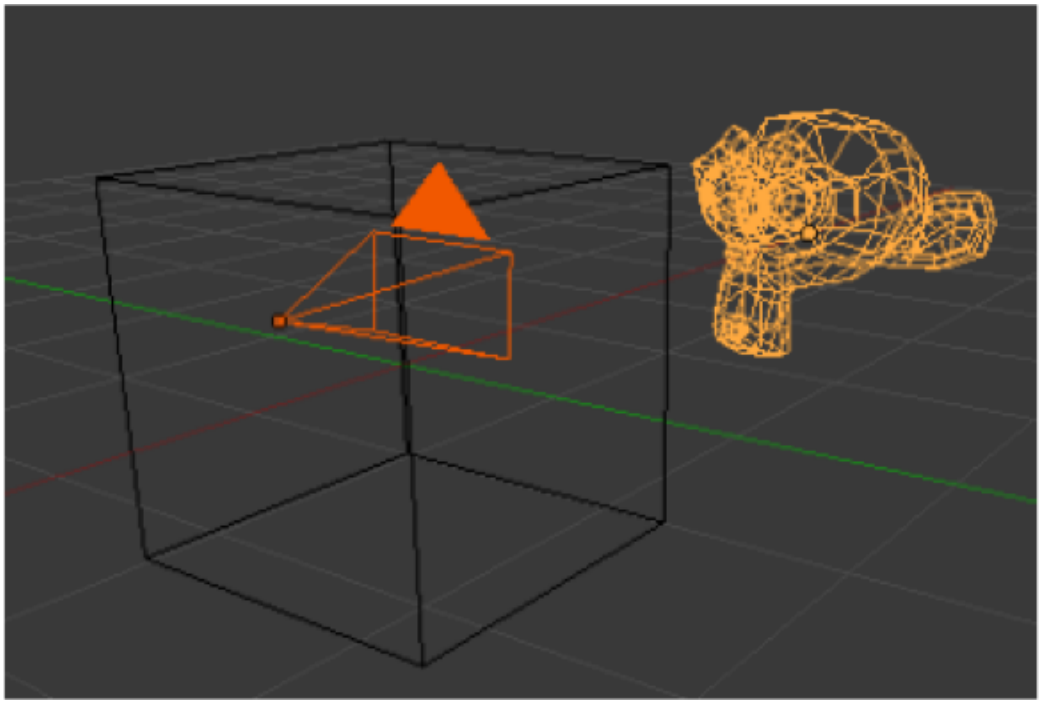


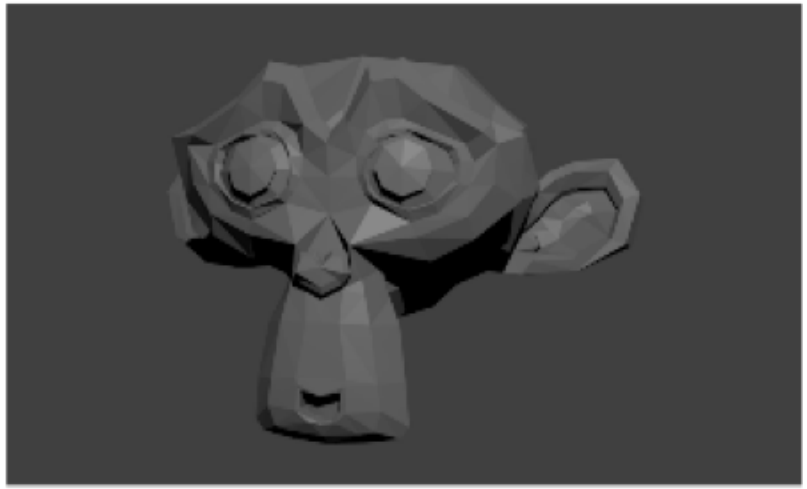


Transparent objects needs also their back faces to be drawn, since they are visible through their front faces.

Back-face culling: final remarks

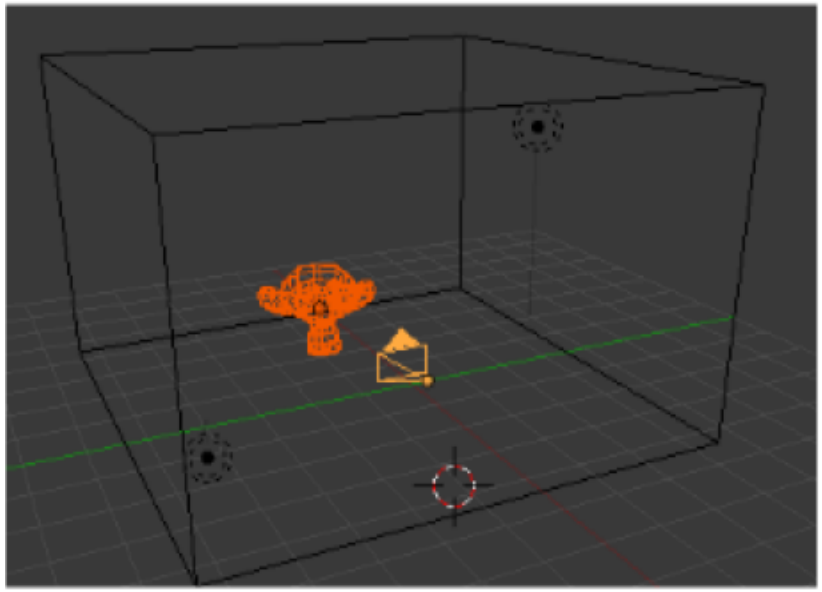
If a camera is placed inside an object for which back-face culling is enabled, its borders are not visible since they have the vertices oriented in the opposite direction.

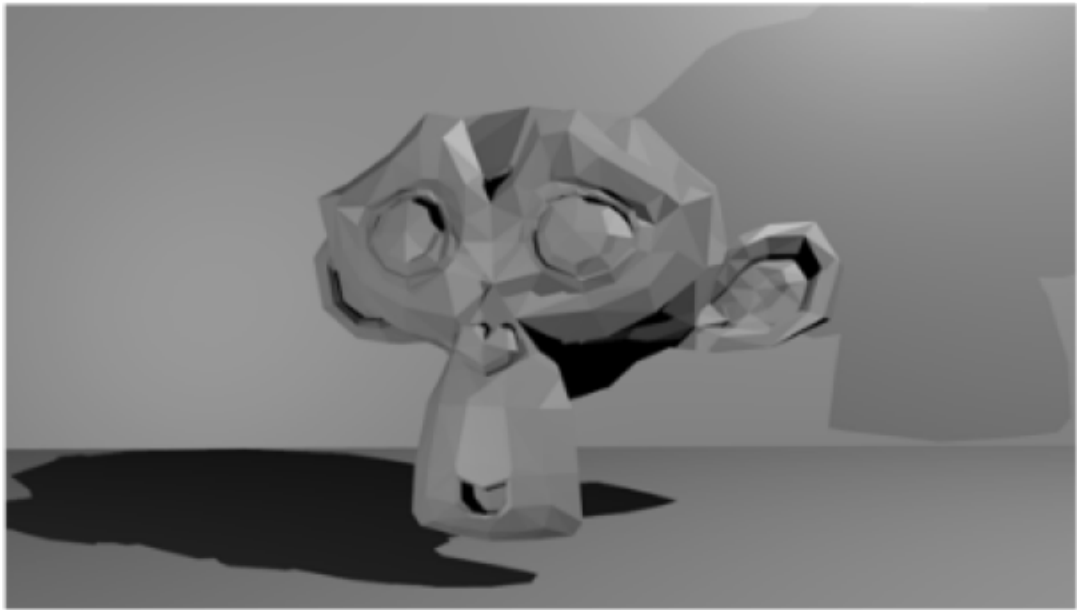




Back-face culling: final remarks

For this reason, an object representing the borders of the area where the 3D world is contained (e.g. a room or a *skybox*), should be created with the vertices ordered in the opposite direction.

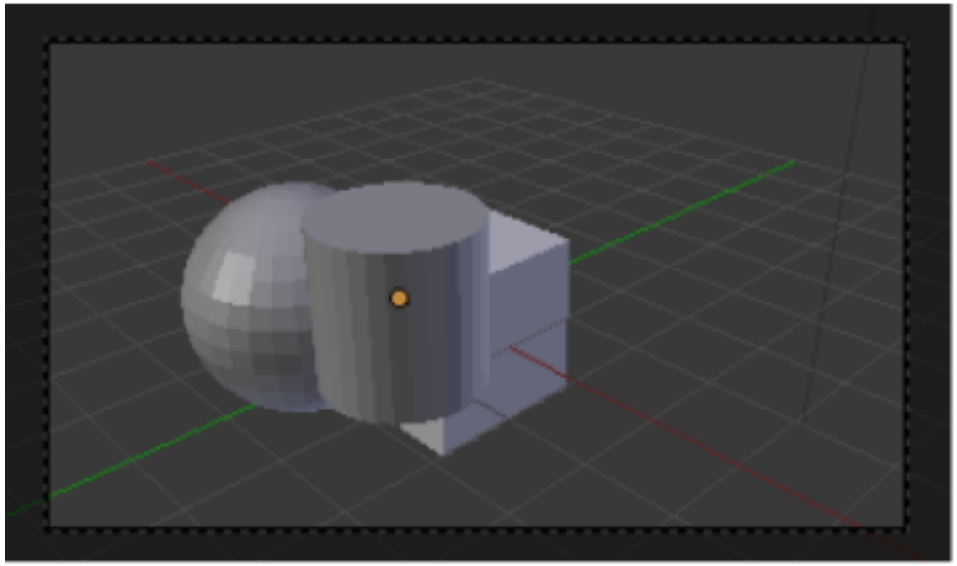


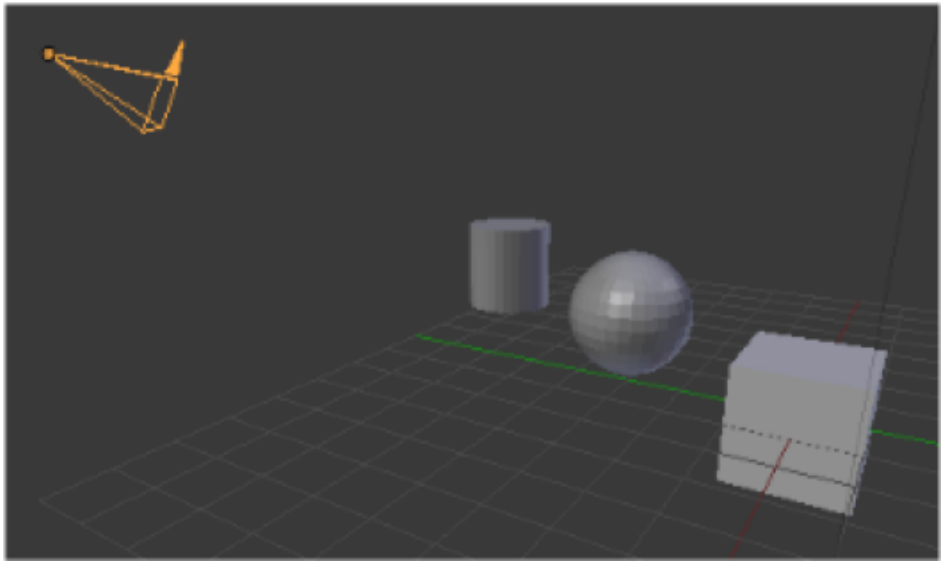


Depth testing

In a complex scene, objects may overlap.

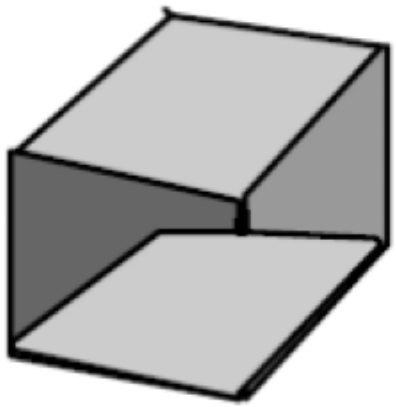
It is important that the polygons closer to the observer cover the objects behind them.





Depth testing

As briefly outlined when presenting 3D normalized screen coordinates, unrealistic figures will appear if faces are not drawn in the proper order.

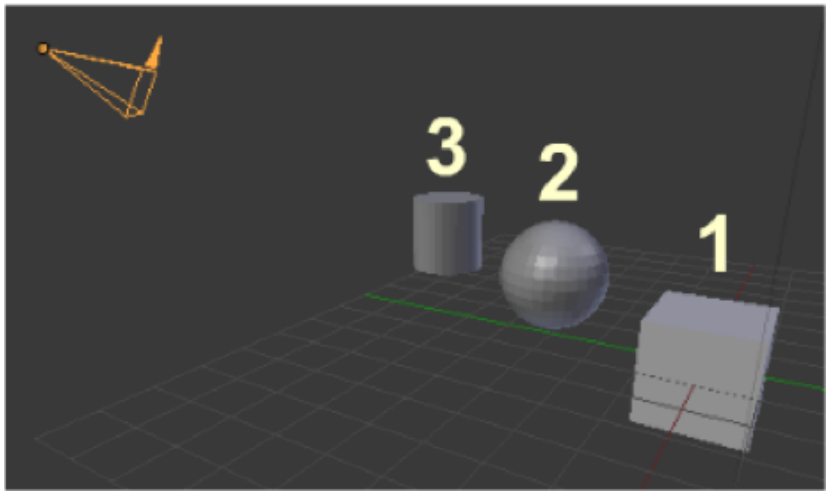


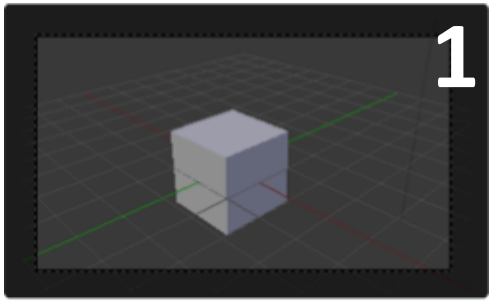
Respecting the proper order of visualization in a set of primitives is called *Hidden Surfaces Elimination*.

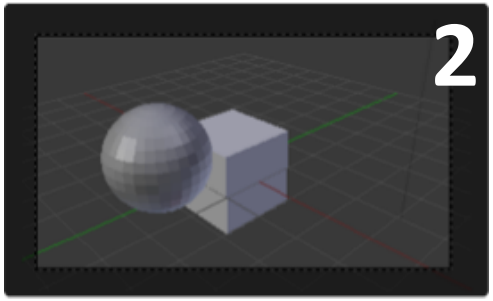
Depth testing

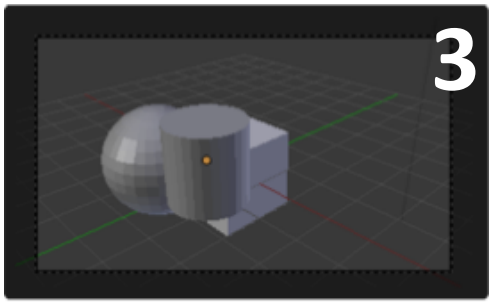
When dealing with non-transparent objects, usually a technique called the *Painter Algorithm* is applied.

Primitives are drawn in reverse order with respect to the distance from the projection plane: in this way, objects closer to the view cover the ones further apart.



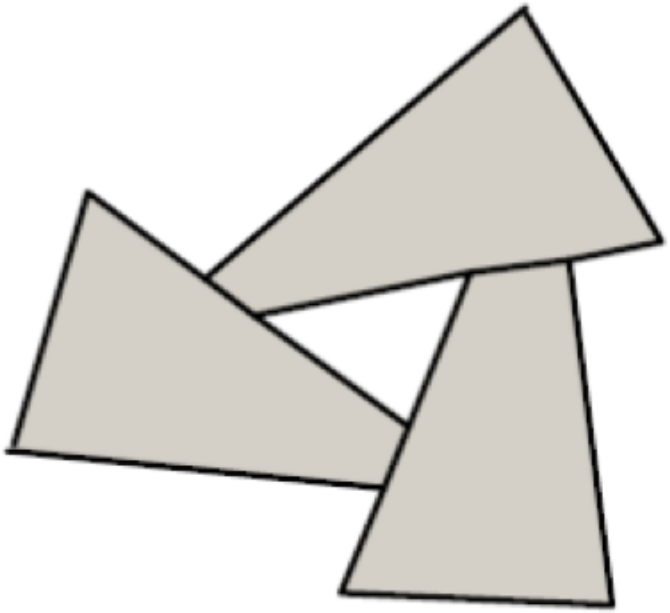






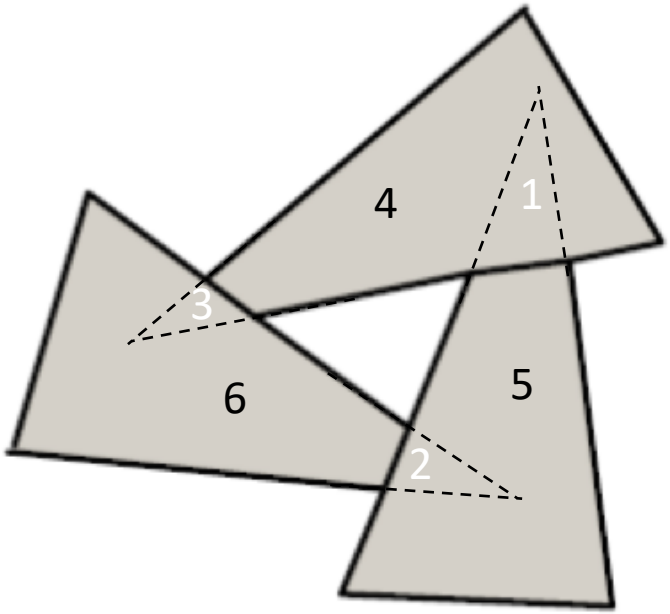
Depth testing

There are cases however in which a correct order cannot be determined and the painter algorithm cannot be applied by simply sorting the objects.



Depth testing

The solution in this case relies on splitting the primitives so that it is possible to find a proper order of the considered pieces.



Z-buffering

The *z-buffering* technique splits the primitives at the pixel level.

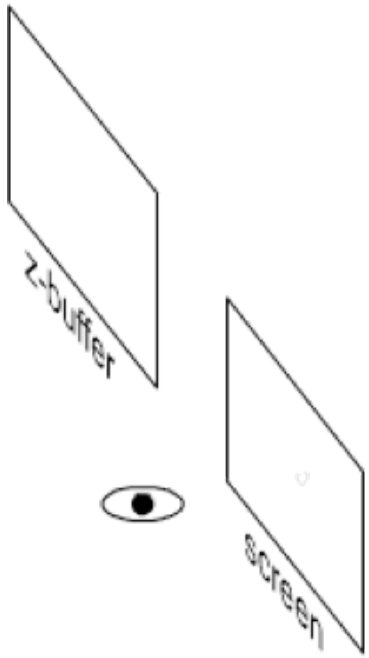
In particular, it applies the painter algorithm to the pixels resulting from the complete projection sequence, rather than to the surfaces of the objects being drawn.

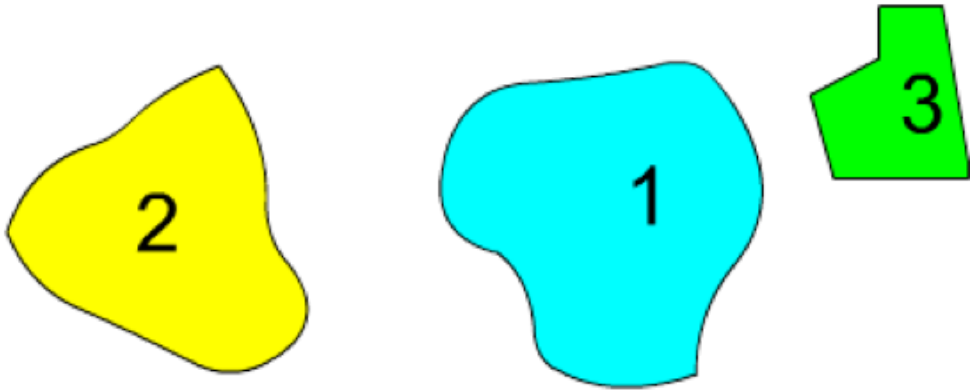
This technique requires a special memory area that stores additional information for every pixel on the screen, which is called the *z-buffer*.

Since the technique uses the distance from the projection plane, sometimes called “depth”, it is also known as depth-testing.

Z-buffering

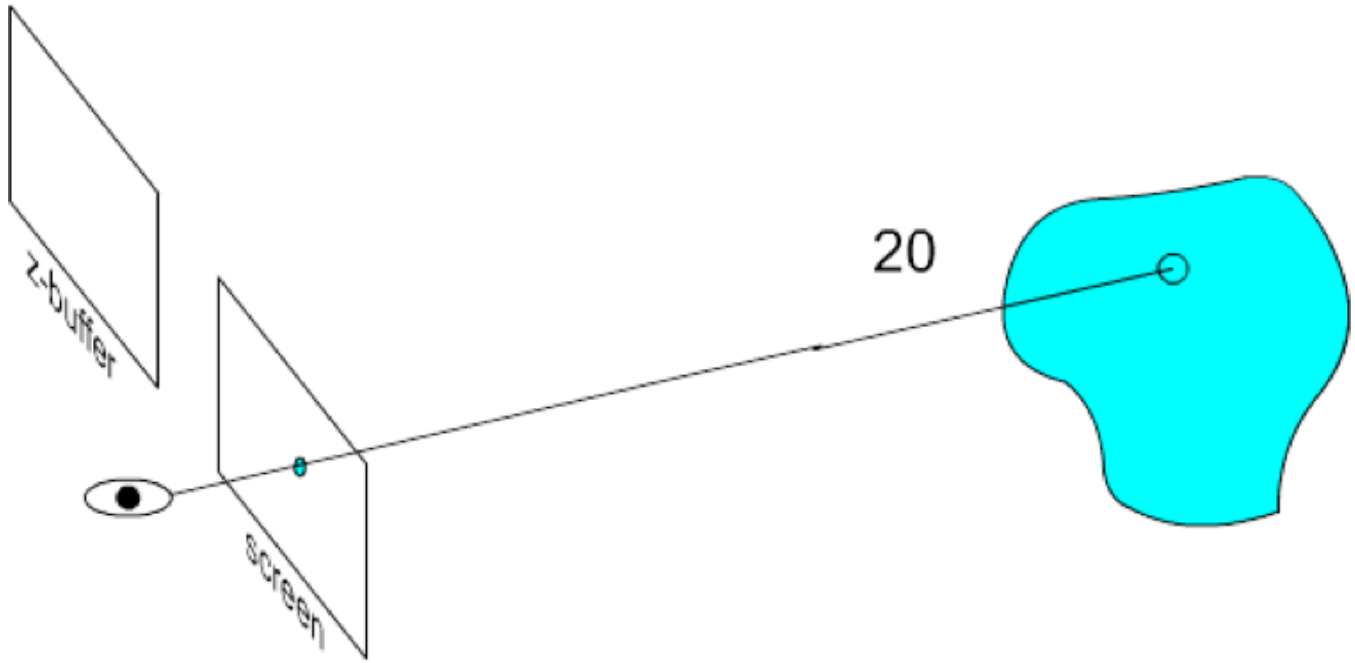
The algorithm draws all the primitives in the scene, and tests whether to draw their corresponding pixels on screen.





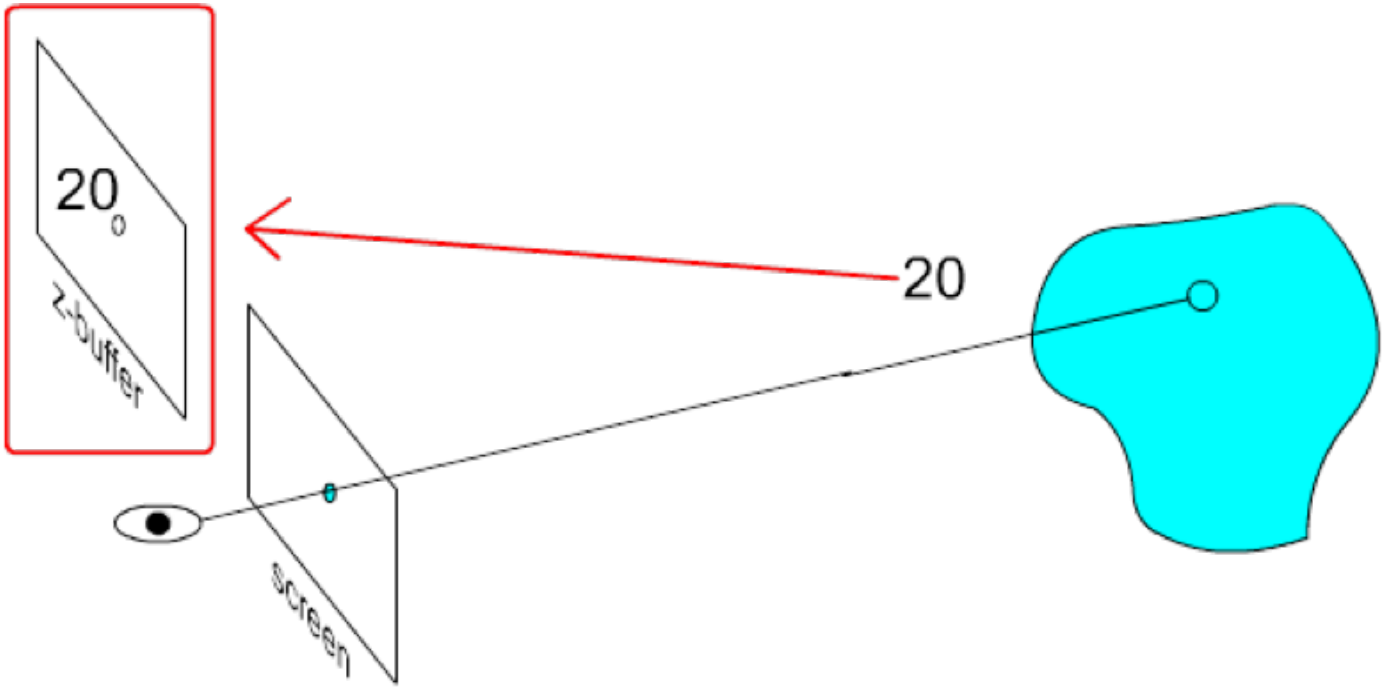
Z-buffering

For each pixel, both the color and the distance from the observer are computed.



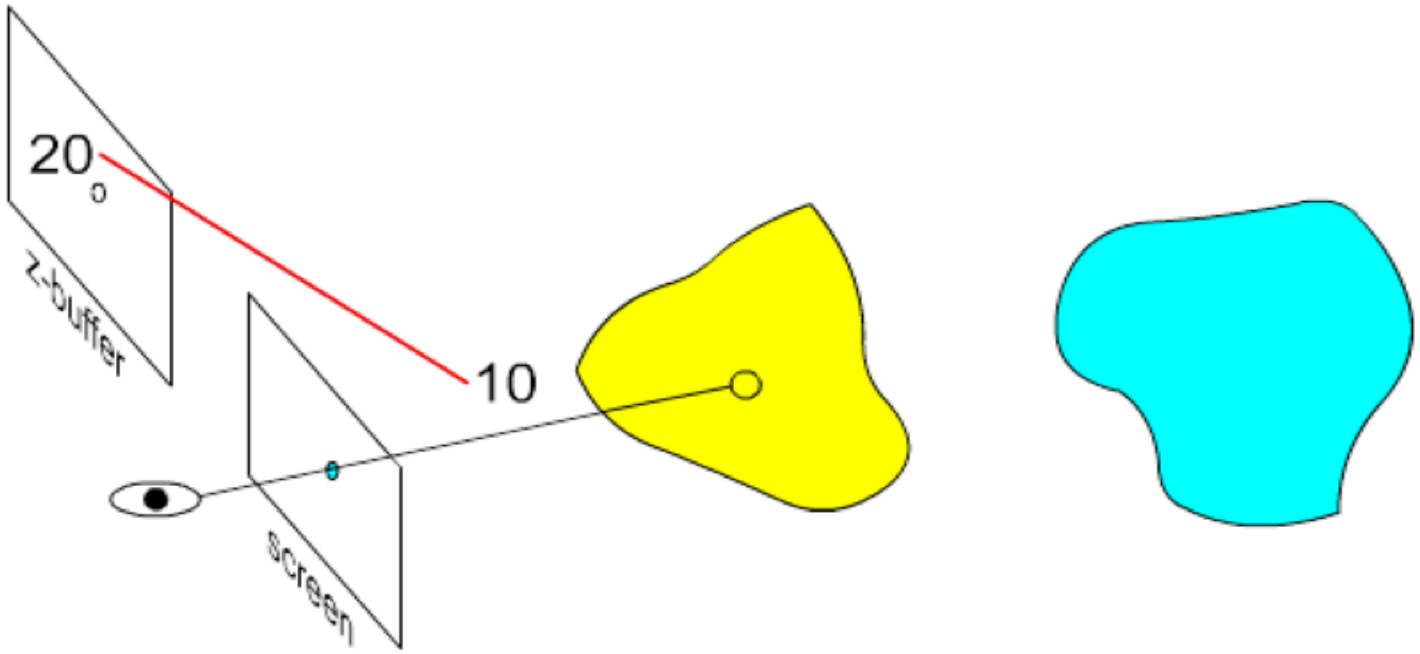
Z-buffering

The z-buffer stores the distance from the observer (i.e. the normalizes screen *z* coordinate) for each pixel on the screen.



Z-buffering

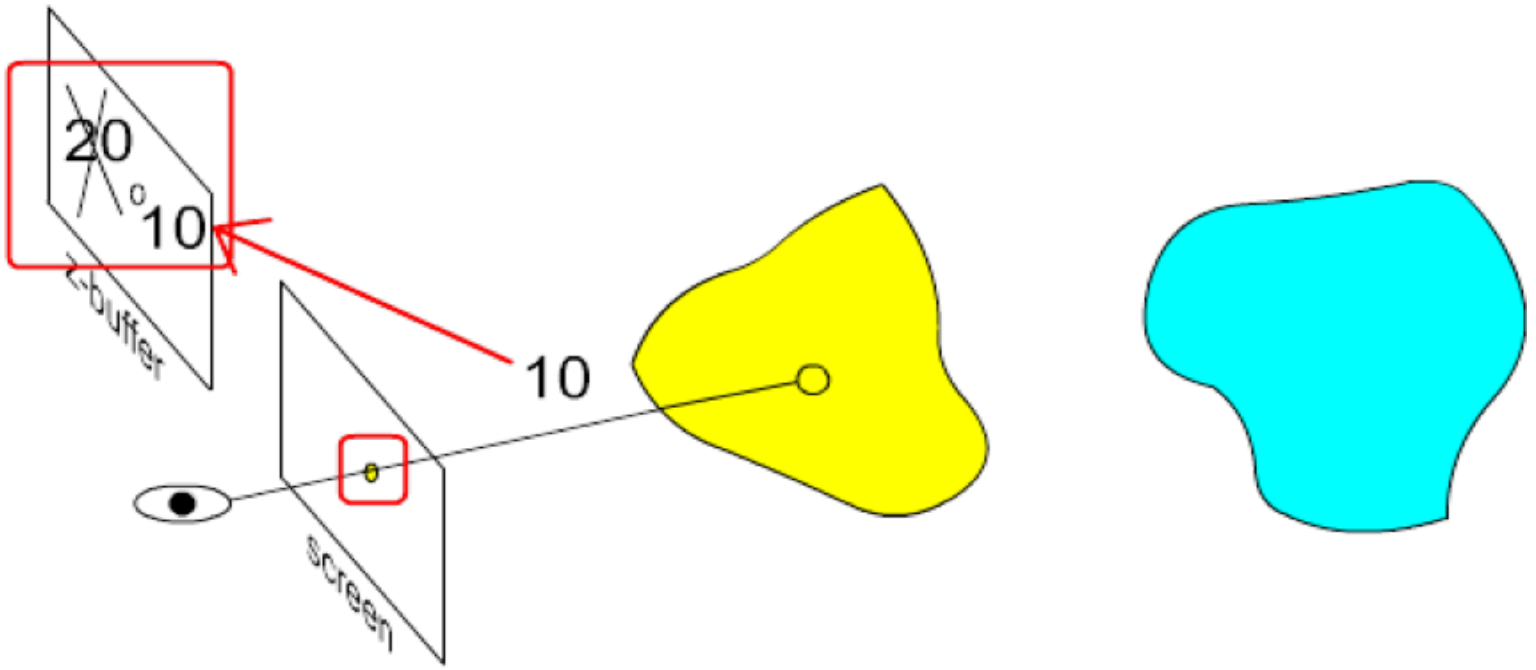
When a new pixel in the same position is written, its distance from the observer is compared against the value stored in the z-buffer.



Z-buffering

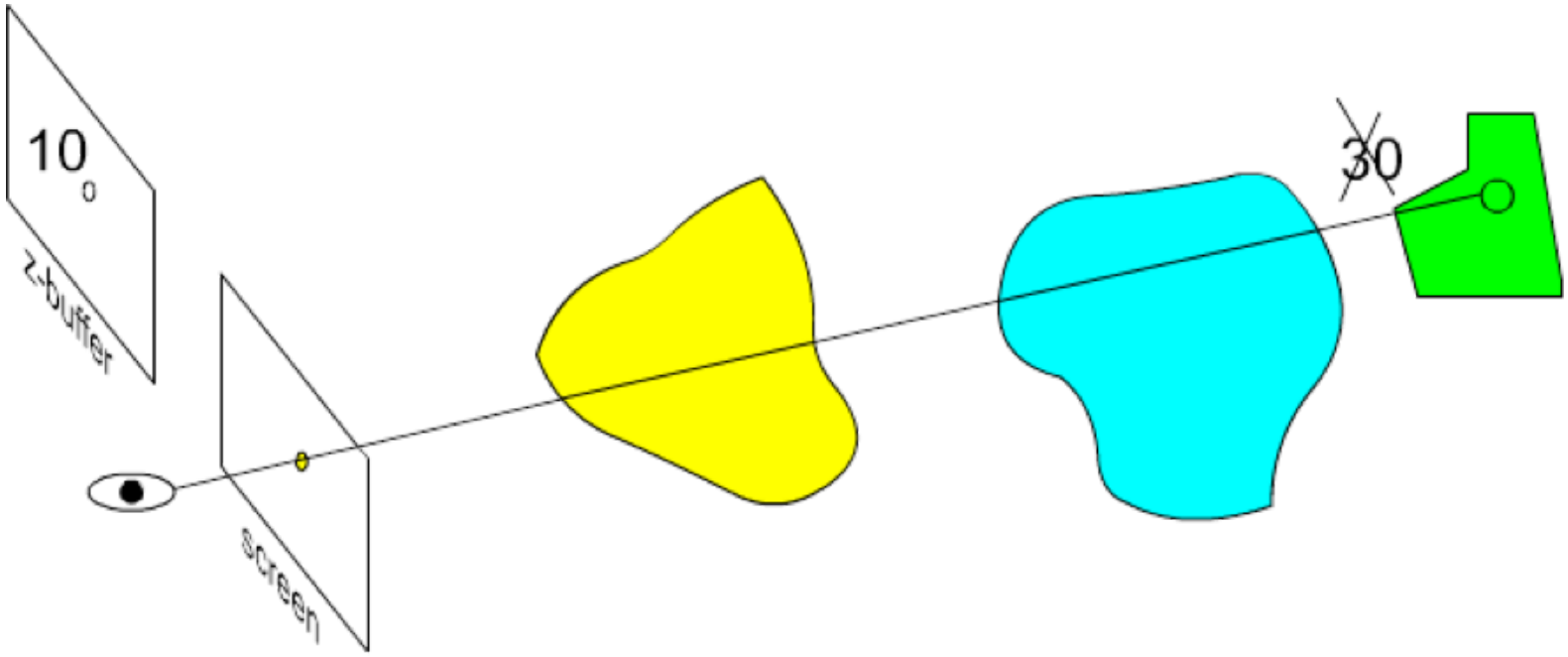
The new pixel is written on screen if the distance of the new pixel is less than the one contained in the z-buffer.

The corresponding value in z-buffer is also updated with the new distance.



Z-buffering

If instead the distance of the new pixel is grater than the value stored in the z-buffer, the new pixel is discarded (since it corresponds to an object behind the one already shown) and no update is performed.



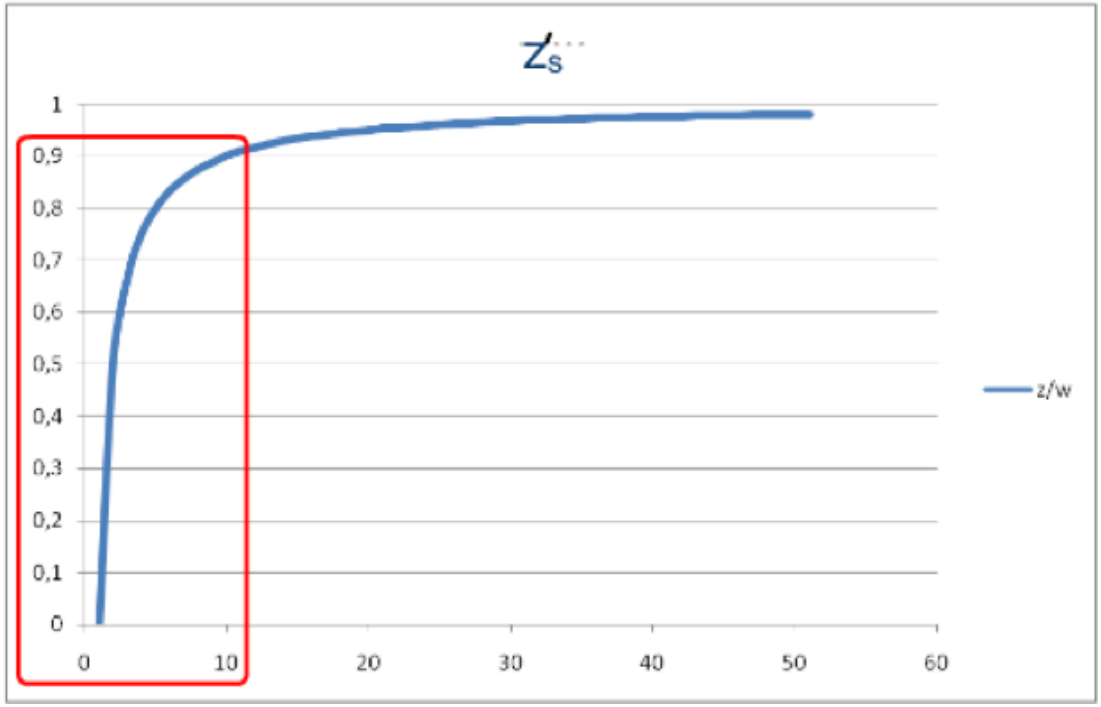
Z-buffering: issues

The z-buffering technique is very simple, but it requires an extra memory area that can store the distance information for all the pixels: in Vulkan, this memory area must be created by the programmer, making the use of z-buffer more complex than in other environments.

Moreover, it requires the generation of all the pixels belonging to the primitives displayed, even if they are completely covered by other objects.

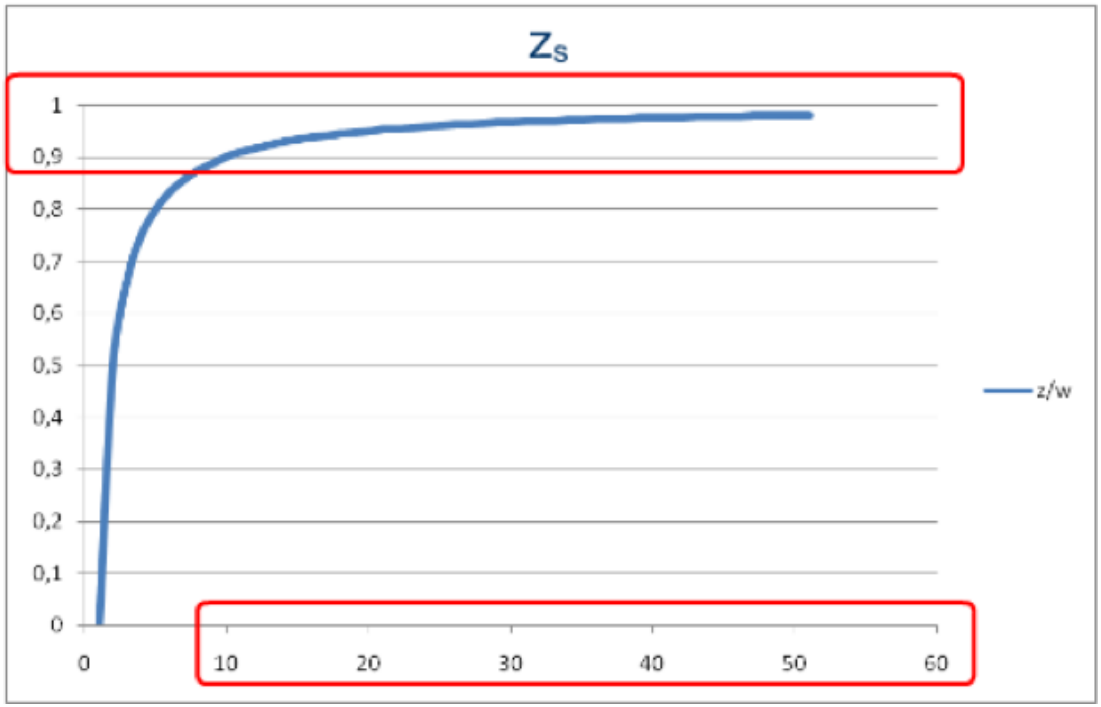
Z-buffering: issues

The worst issue is the numerical precision: the largest part of the *[0,1]* range of the z_s coordinates, is used for the points that are very close to the projection plane (and that are generally not much used in a scene).



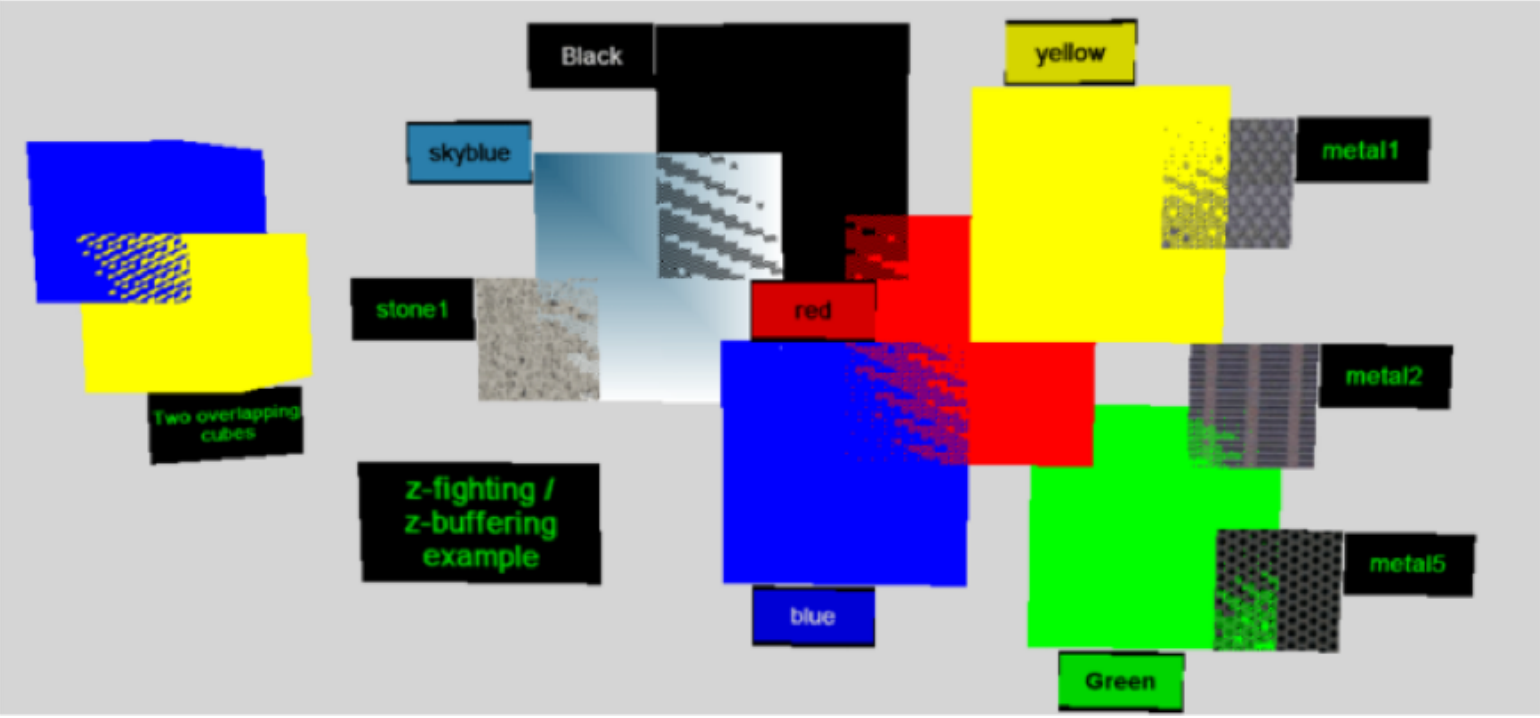
Z-buffering: issues

Since values are discretized, we need a sufficient precision to store z_s for the distances that are further away from the projection plane.



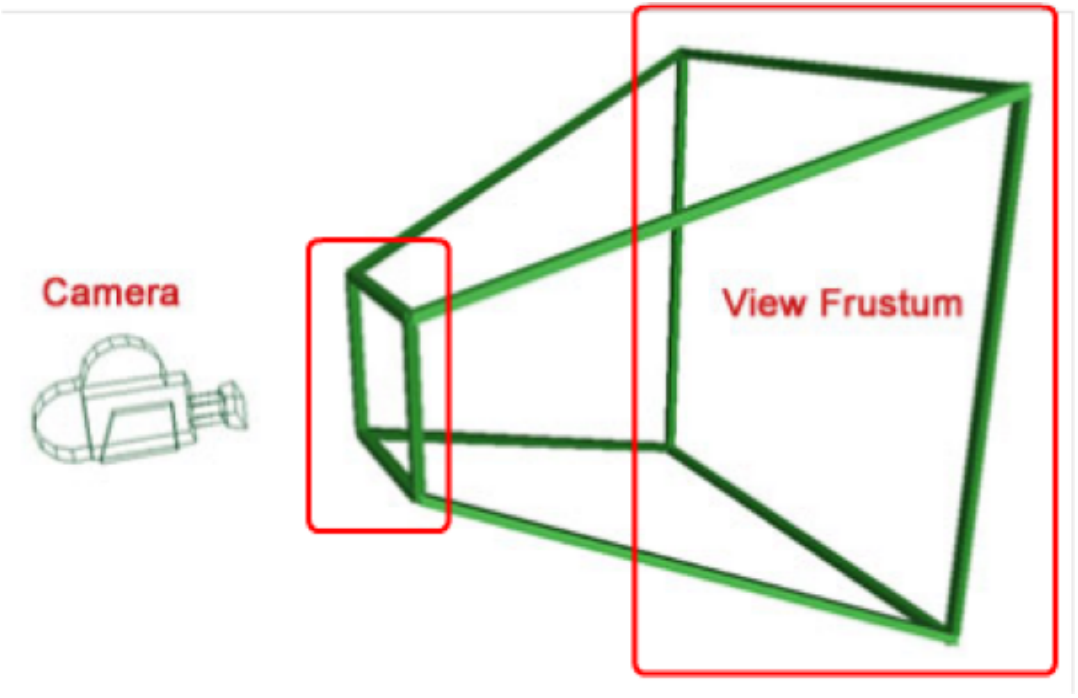
Z-buffering: issues

Otherwise a problem known as "*Z Fighting*" may occur: when two almost co-planar figures are rendered, the final color is determined by the round-off of the two distances.



Z-buffering: issues

Since the z_s coordinate are normalized with respect to the position of the near and far planes, these two parameters cannot be set arbitrarily small / large, and they should always be appropriate for the considered scene.



Stencil buffer

Stencil buffer is a technique similar to z-buffer, usually adopted to prevent an application from drawing in some region of the screen.

Like z-buffering, it is implemented by storing additional information for every pixel on the screen in a special memory area called the *stencil buffer*.

Stencil buffer

A typical application of stencil buffer is to allow the rendering area to be of arbitrary shapes: for example, reserving the space to draw the cabin of a ship, or the HUD (Head-Up Display).

Wing Commander
(Origin System Inc.) 1990

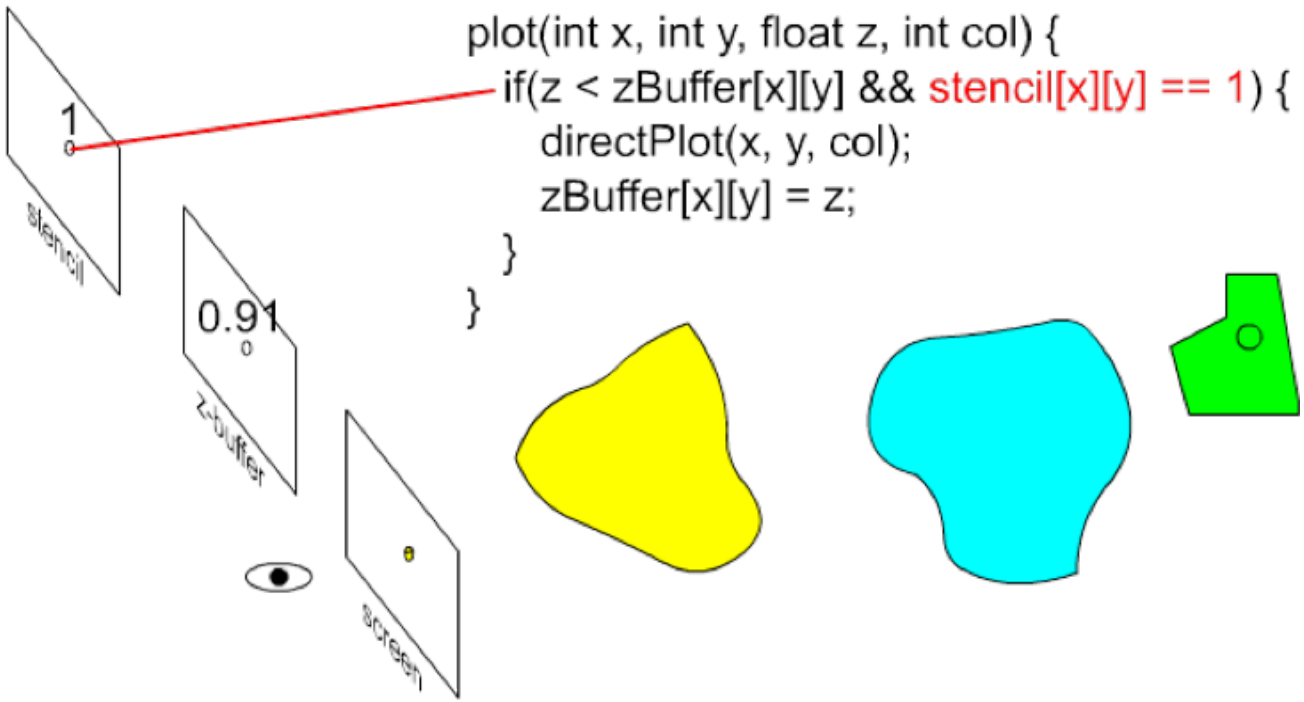
Stencil

More complex applications use stencils to draw shadows, reflections or contours in multi-pass rendering techniques.

Stencil

The stencil buffer is a memory area that associates an integer information to each pixel, which is usually encoded at the bit level.

During rendering, stencil buffer data can be used to perform specific tasks on the corresponding pixels.



Stencil

Usually it stores a single bit of information (1 - the pixel must be drawn, 0 - the pixel can be skipped), but more complex functions can be accomplished.