POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

DE
IB

2024

Dipartimento di Elettronica, Informazione e Bioingegneria

Computer Graphics

Milano, 2024

Computer Graphics

- Clipping, culling, depth-testing and screen synchronization

Introduction

In this set of slides, we will be presenting four techniques that are used in real-time computer graphics to address specific optimization and visualization problems. They are:

- Back-face culling
- Depth testing (and stencil buffer)
- Clipping
- Screen update synchronization

All these aspects can be configured in Vulkan, and affect some aspects of the visualization process. Knowing them in advance, can help in better understanding how Vulkan works.

Back-face culling

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

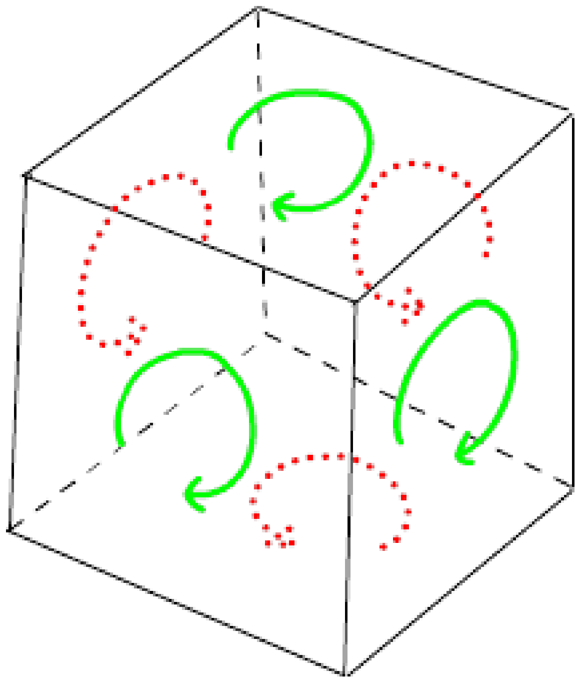
The check can be performed either before the projection of the triangle or using the normalized screen coordinates.

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.

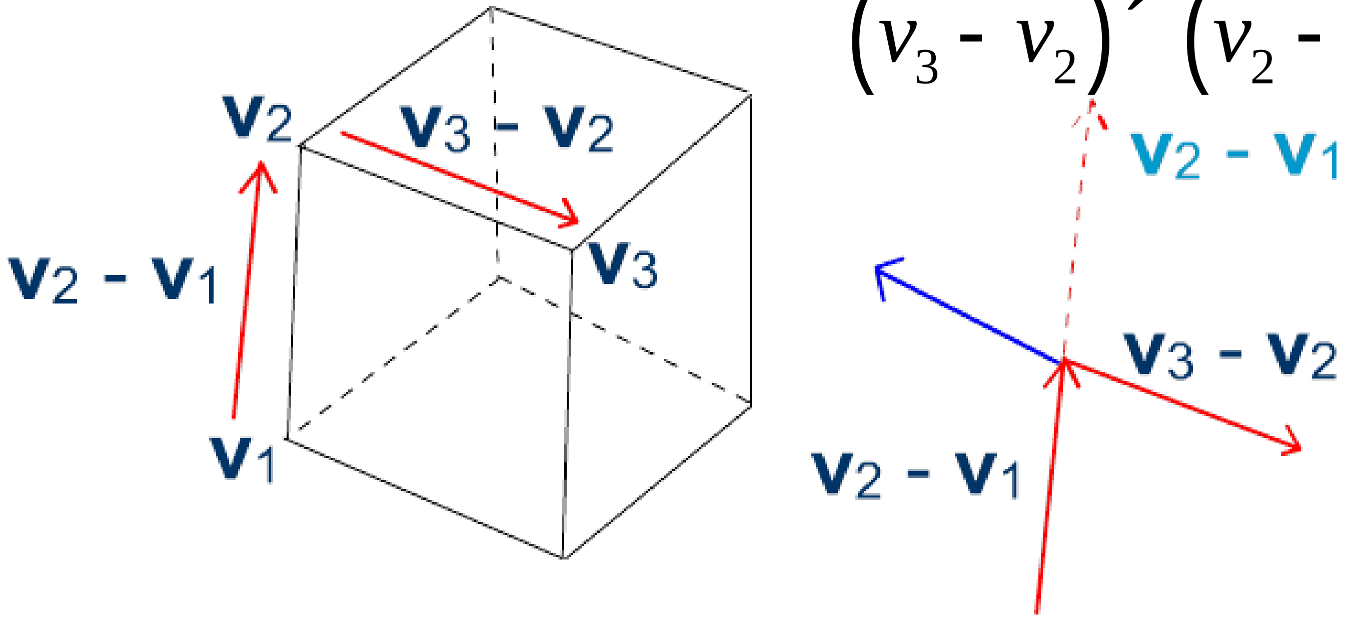
Once projected on screen, front faces will still be ordered clockwise, while back faces will be ordered in the opposite direction.



Back-face culling: implementation

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 the vertices are ordered clockwise.

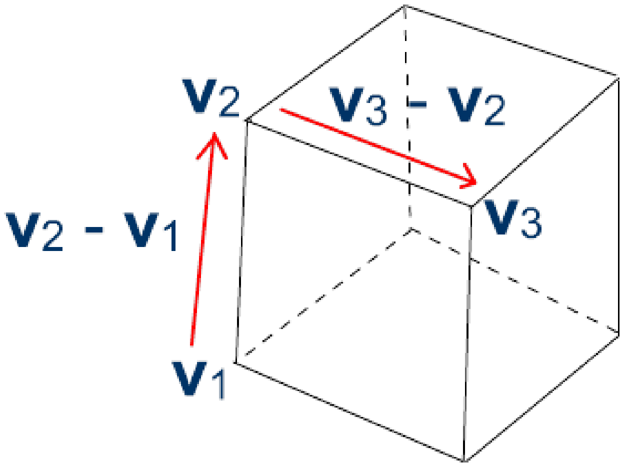
$$(v_3 - v_2) \times (v_2 - v_1)$$



Back-face culling: implementation

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

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



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

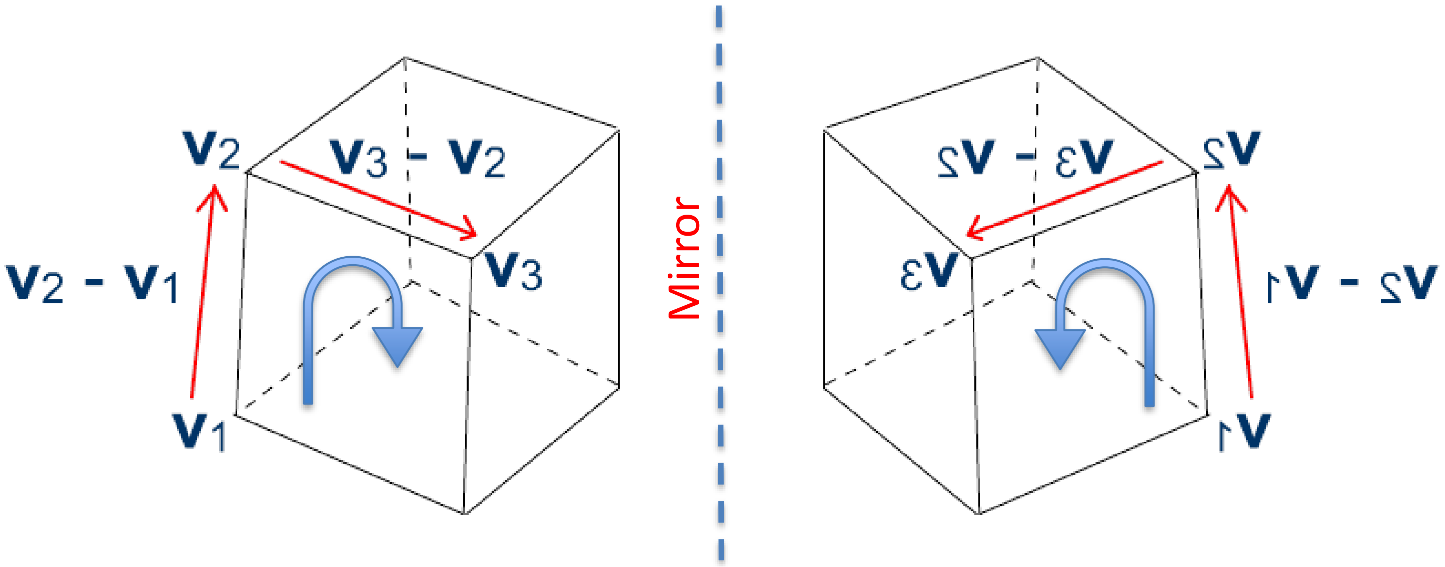
$$|u_x \quad u_y \quad u_z| \times |v_x \quad v_y \quad v_z| = |u_y v_z - u_z v_y \quad u_z v_x - u_x v_z \quad u_x v_y - u_y v_x|$$

Back-face culling: remarks

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

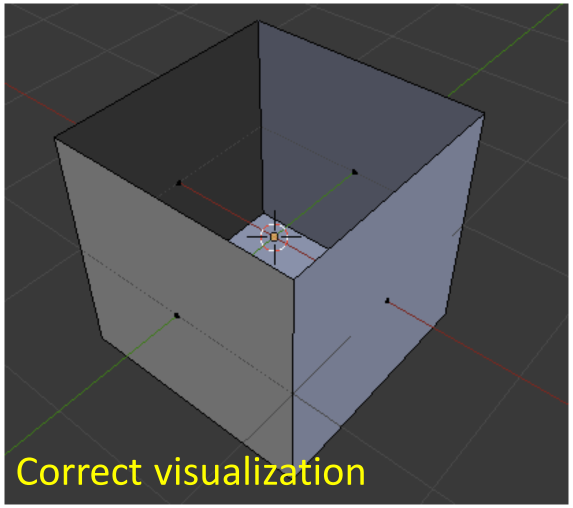
Back-face culling can improve the performances of an application, however there are some potential issues that can arise and that must be considered.

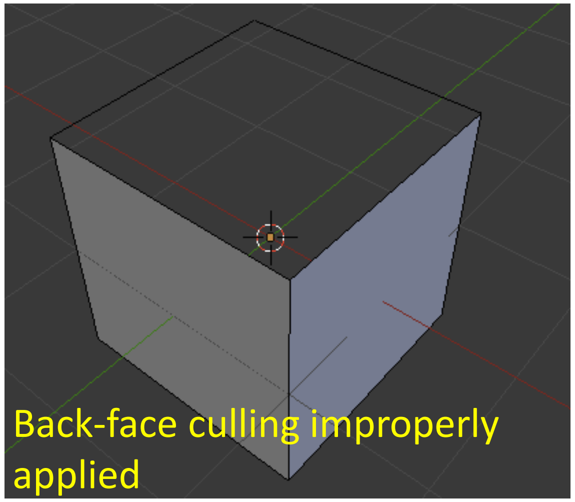
The first is that if a world matrix includes a scaling component with an odd number of negative coefficients (either one or all three), the acceptance test must be inverted.

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 which should be visible, could be delted.

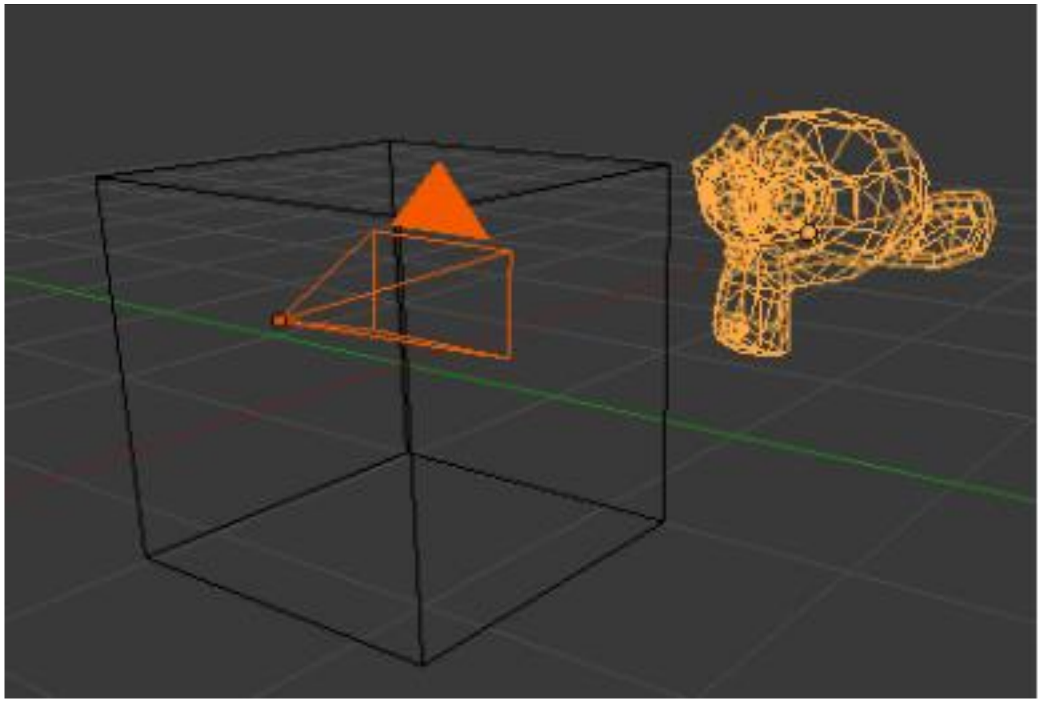


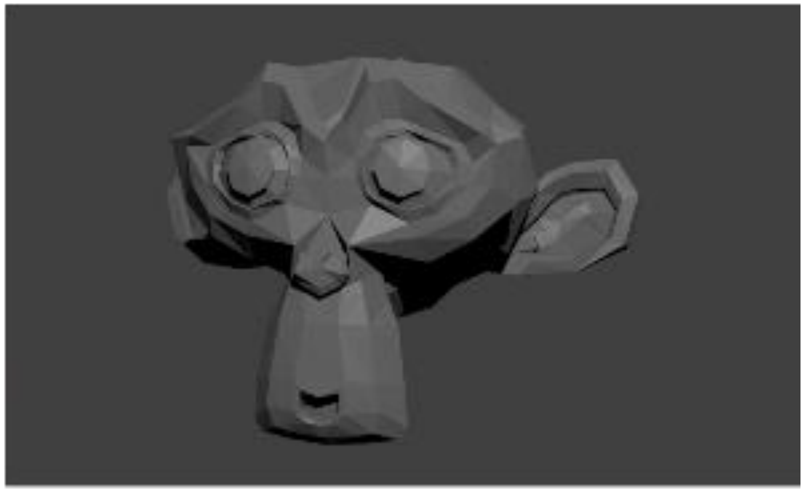


Transparent objects needs also their back faces to be drawn, since they are visible through their front faces.

Back-face culling: final remarks

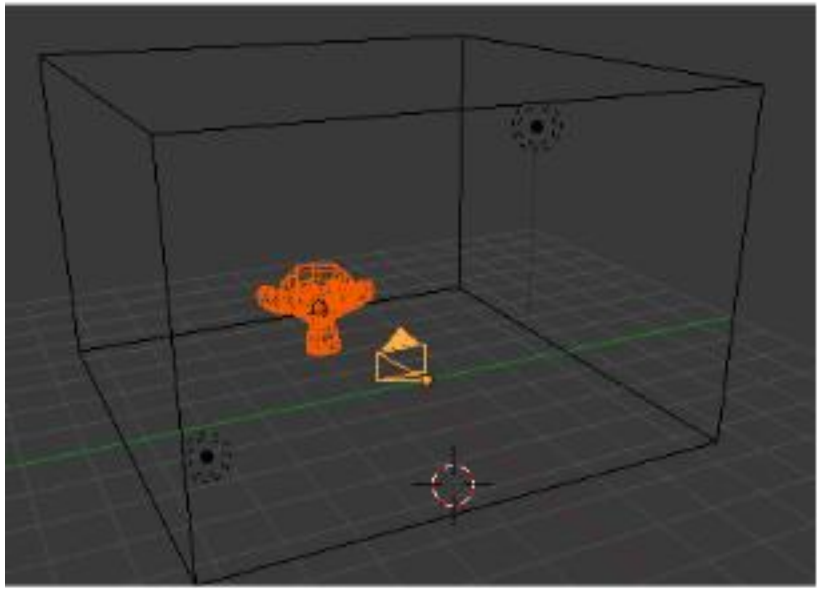
If a camera is placed inside an object for which back-face culling is enabled, its borders are not visible since they have the vertices oriented in the opposite direction.

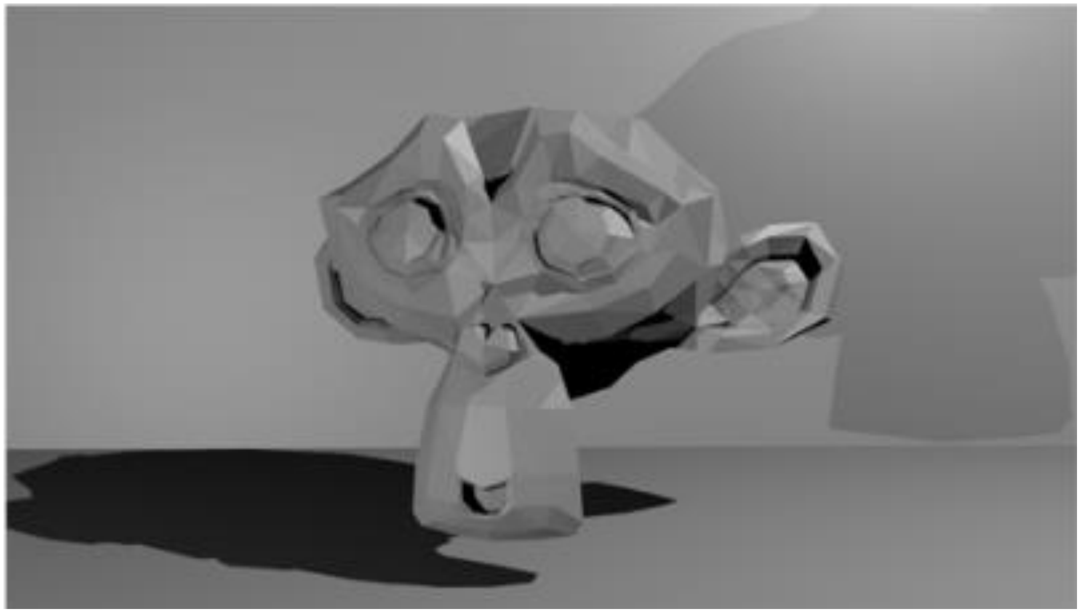




Back-face culling: final remarks

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.

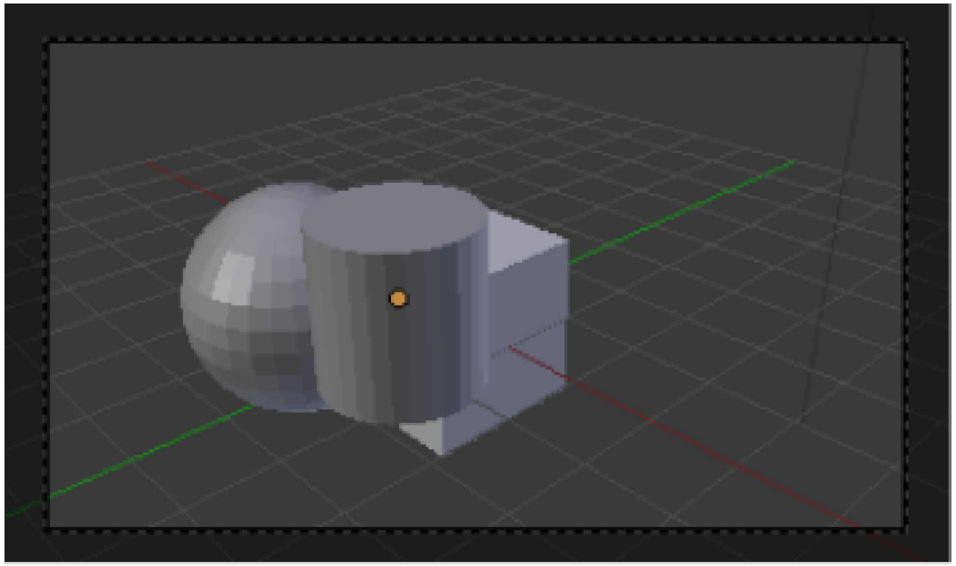
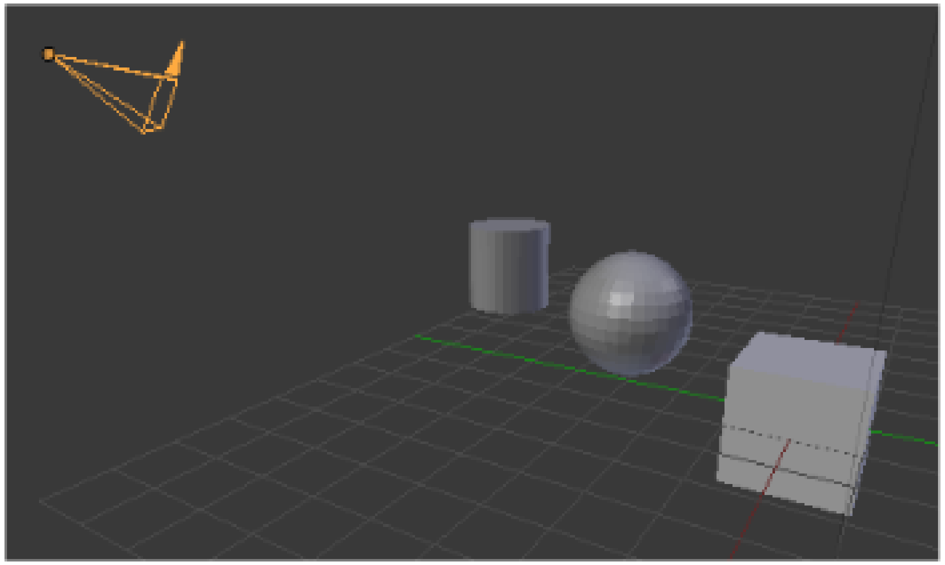




Depth testing

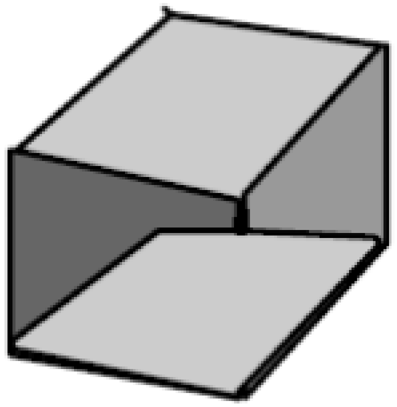
In a complex scene, objects may overlap.

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.

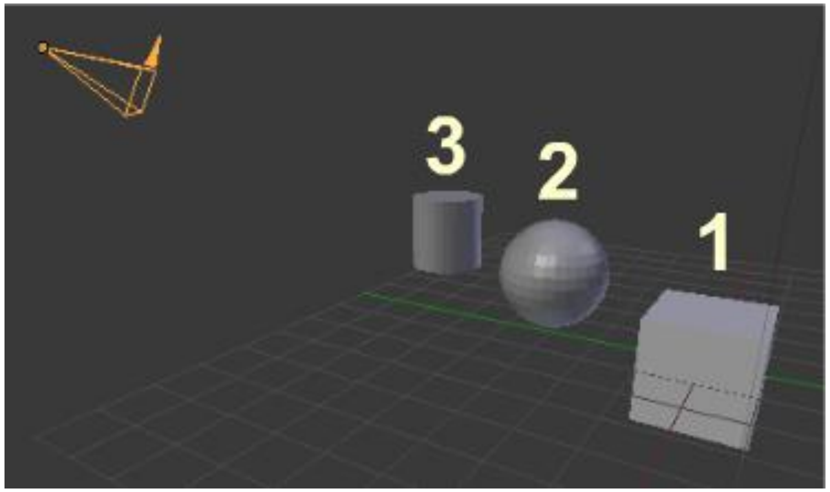


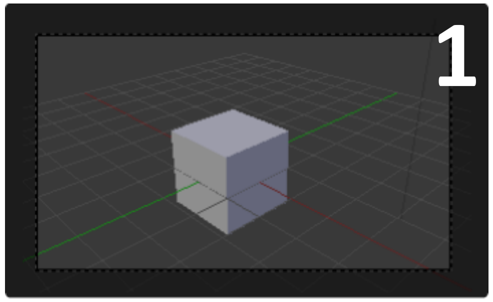
Respecting the proper order of visualization in a set of primitives is called *Hidden Surfaces Elimination*.

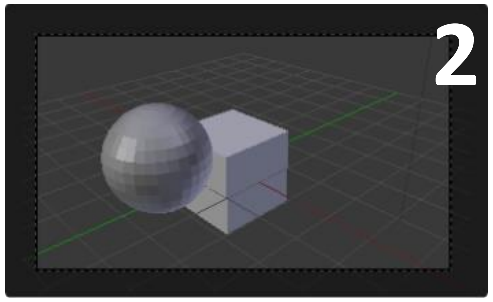
Depth testing

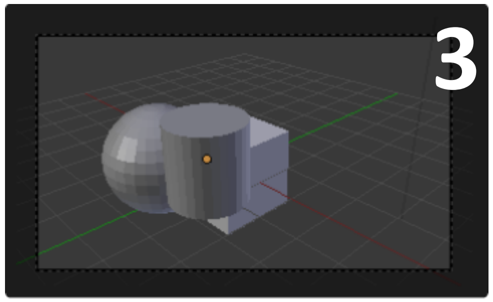
When dealing with non-transparent objects, usually a technique called the *Painter Algorithm* is applied.

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.



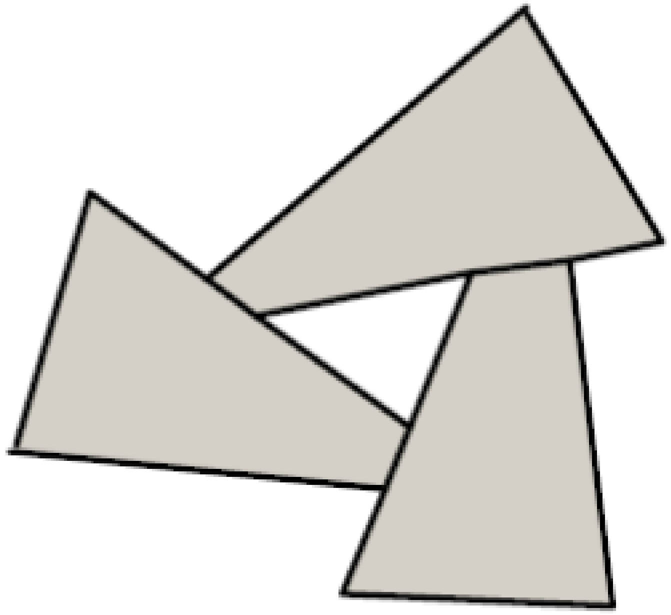






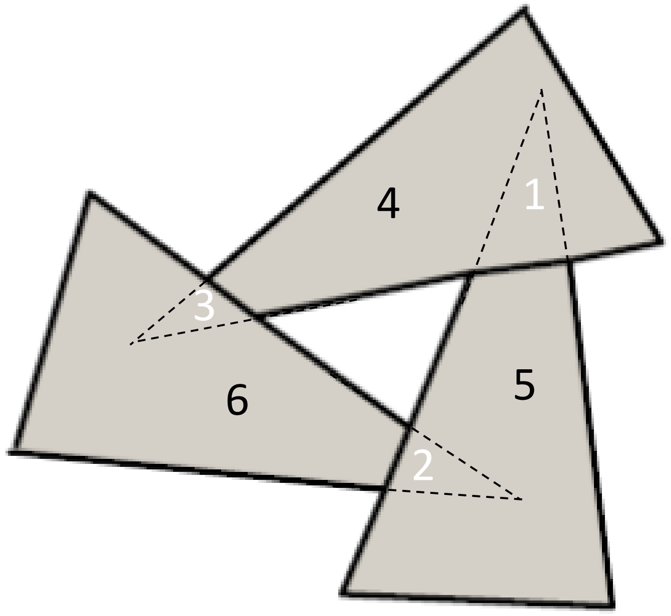
Depth testing

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



Depth testing

The solution in these cased, is splitting the primitives so that it is possible to find a proper order of the considered pieces.



Z-buffering

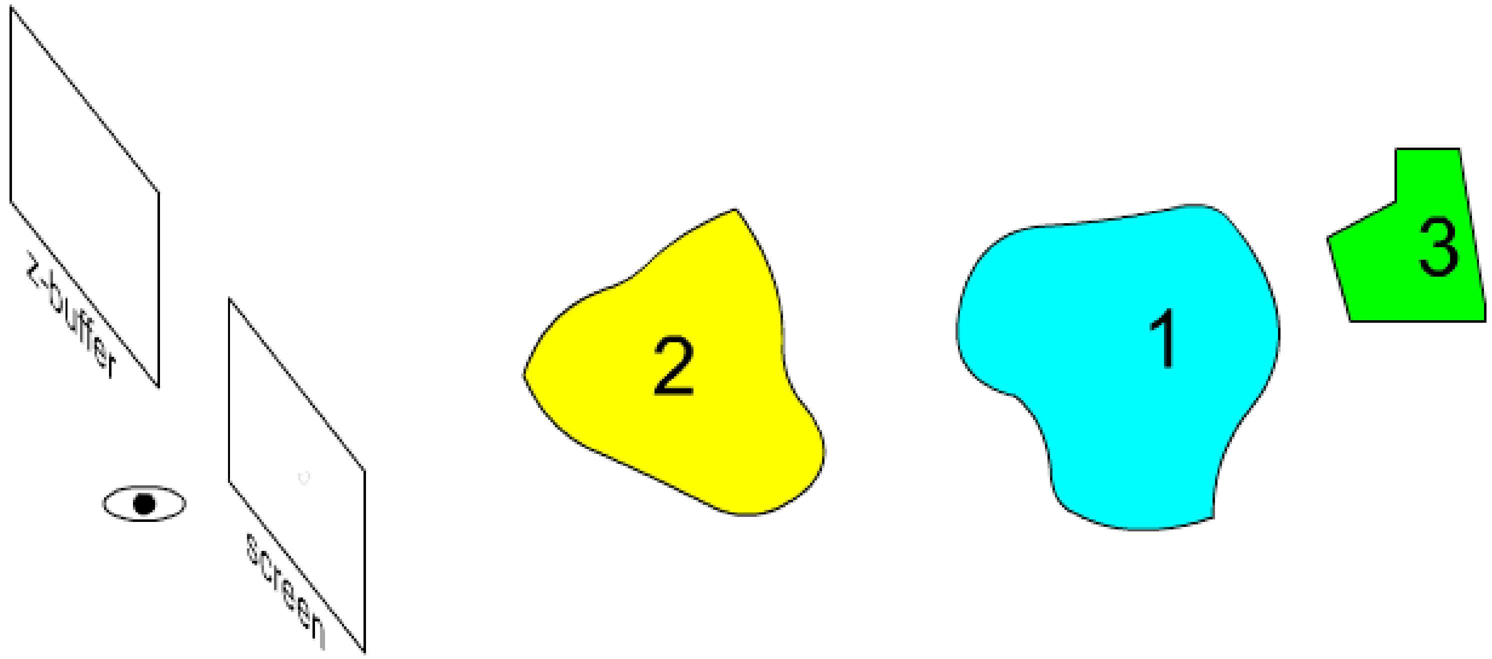
The *z-buffer* technique splits the primitives at the pixel level.

The distance from the projection plane is sometimes called “depth”: for this reason, the technique it is also known as *depth-testing*.

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

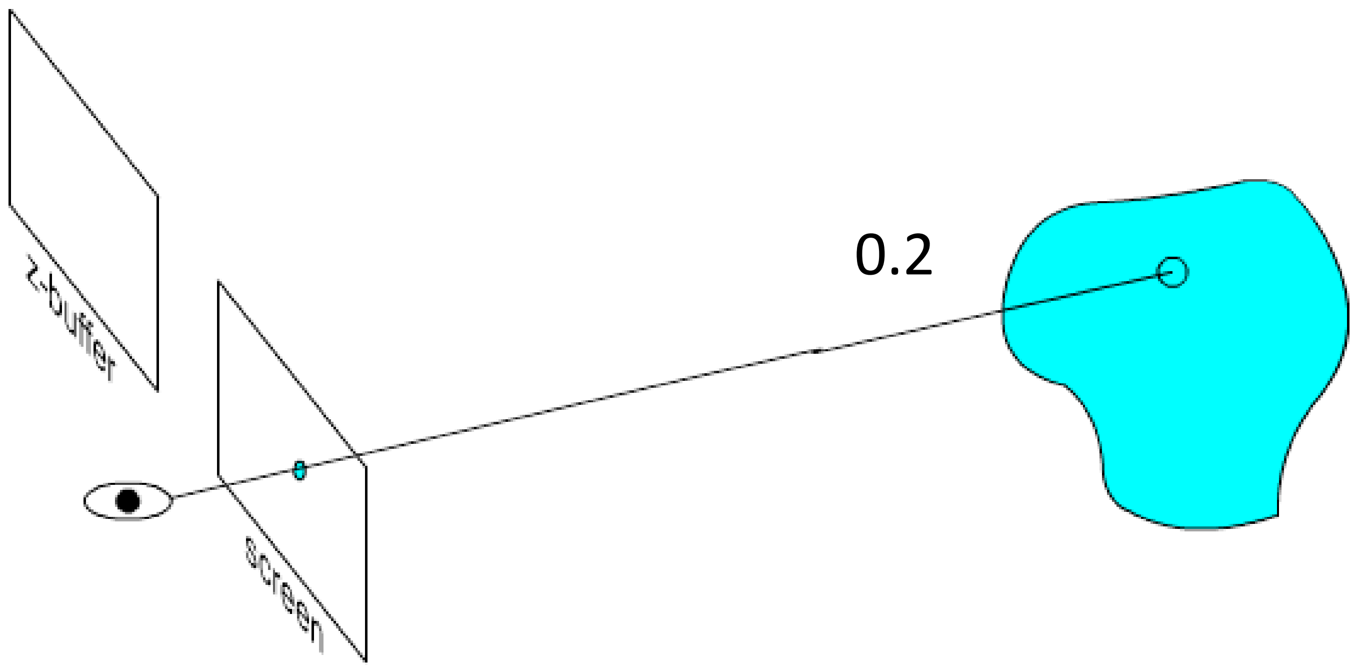
Z-buffering

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



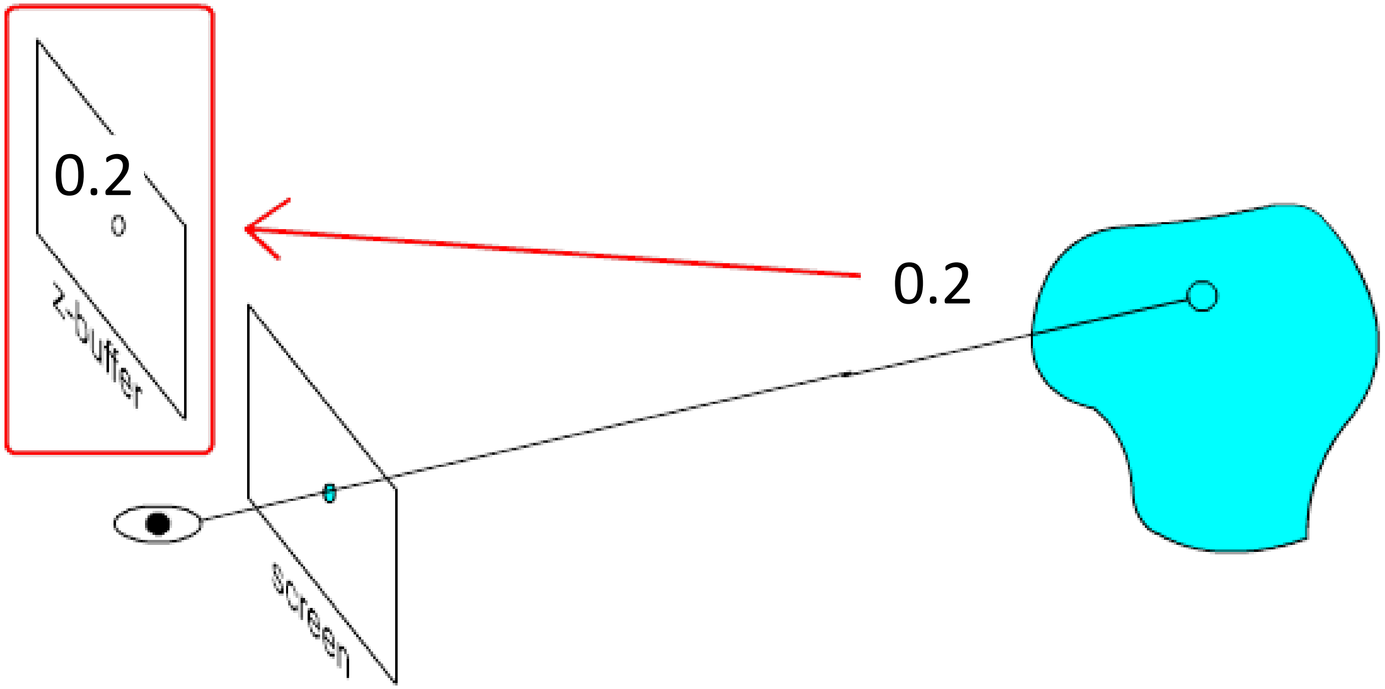
Z-buffering

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



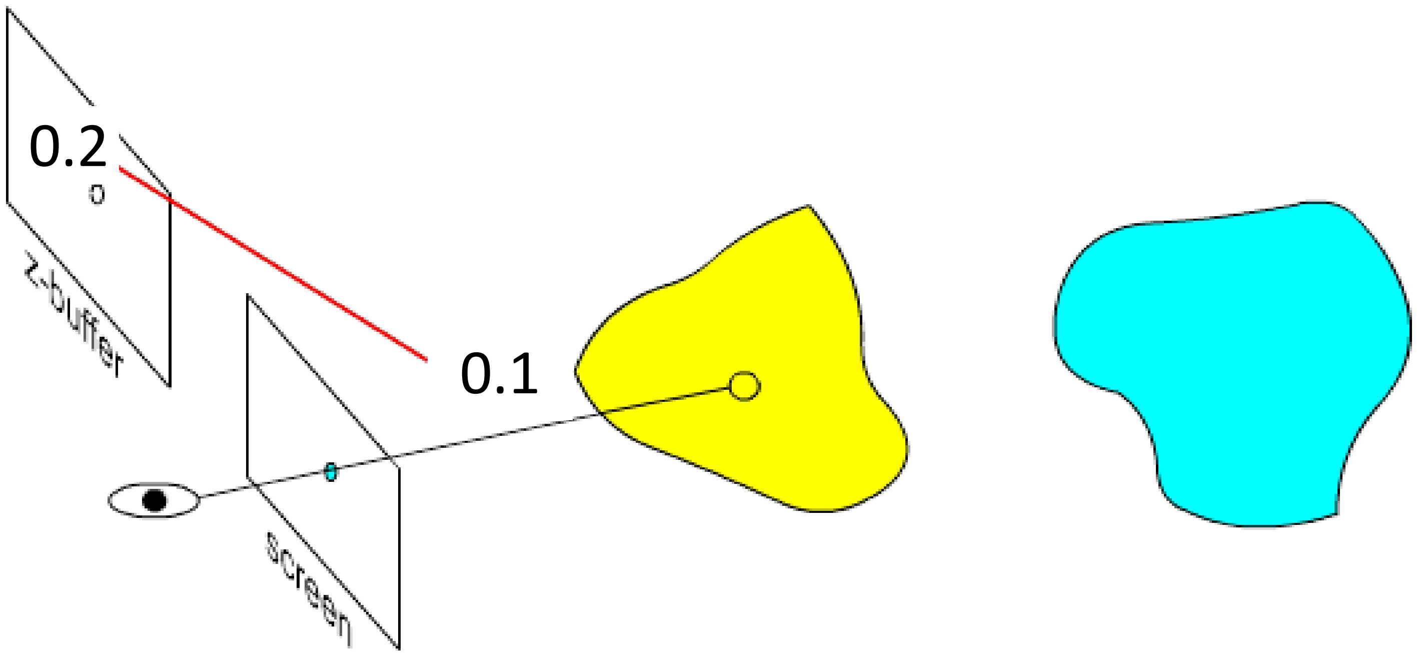
Z-buffering

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



Z-buffering

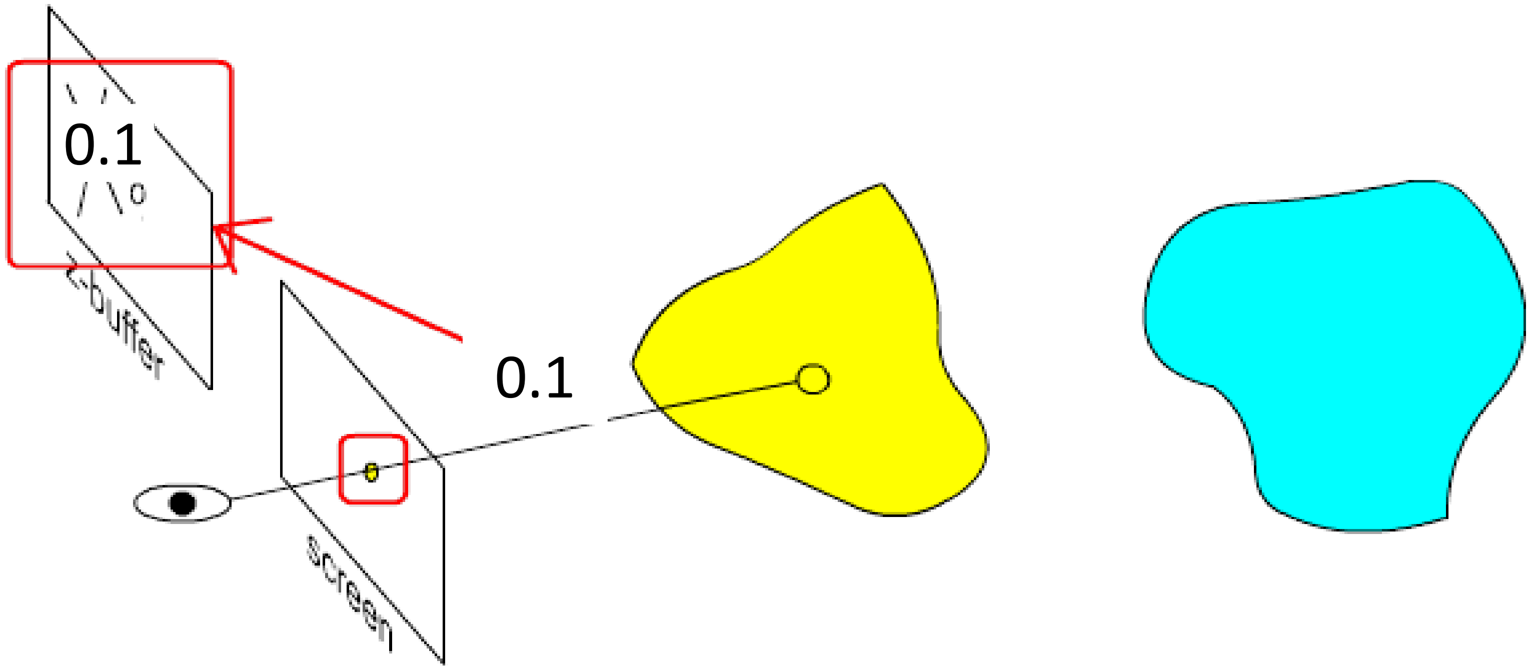
When a new pixel in the same position is written, its distance from the observer is compared against the value stored in the z-buffer.



Z-buffering

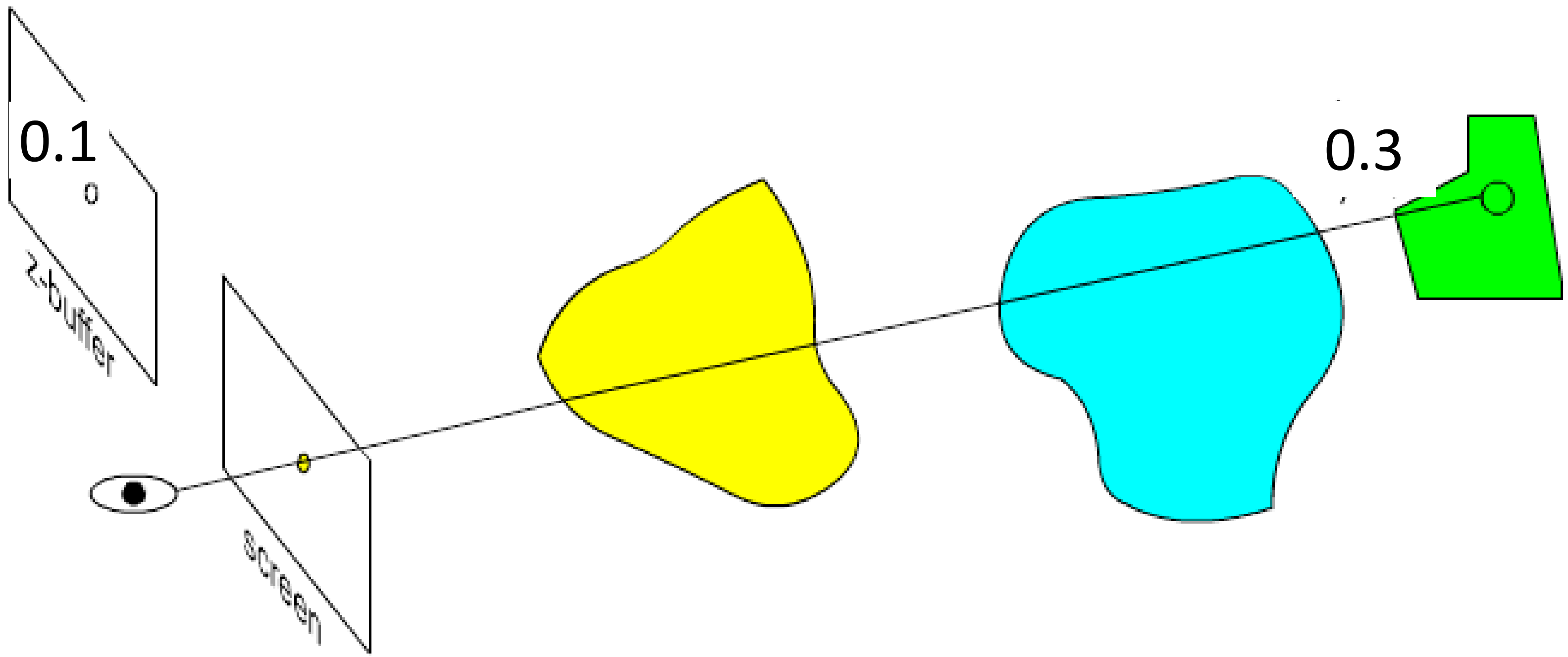
The new pixel is written on screen if the distance of the new pixel is less than the one contained in the z-buffer.

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.



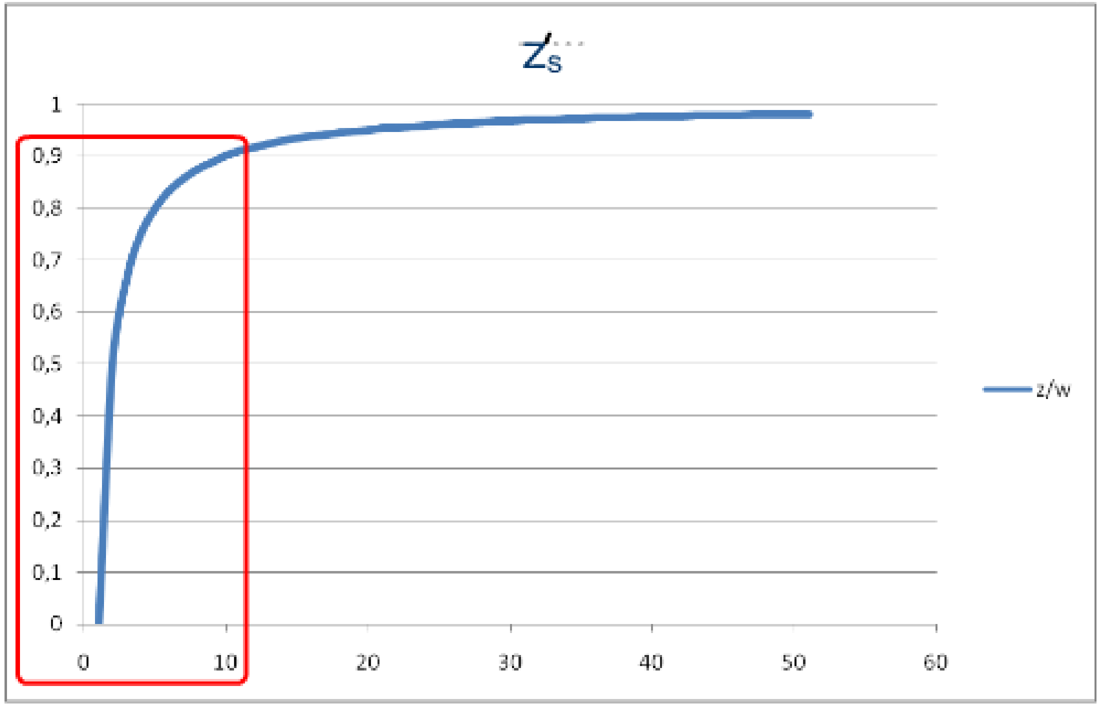
Z-buffering: issues

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.

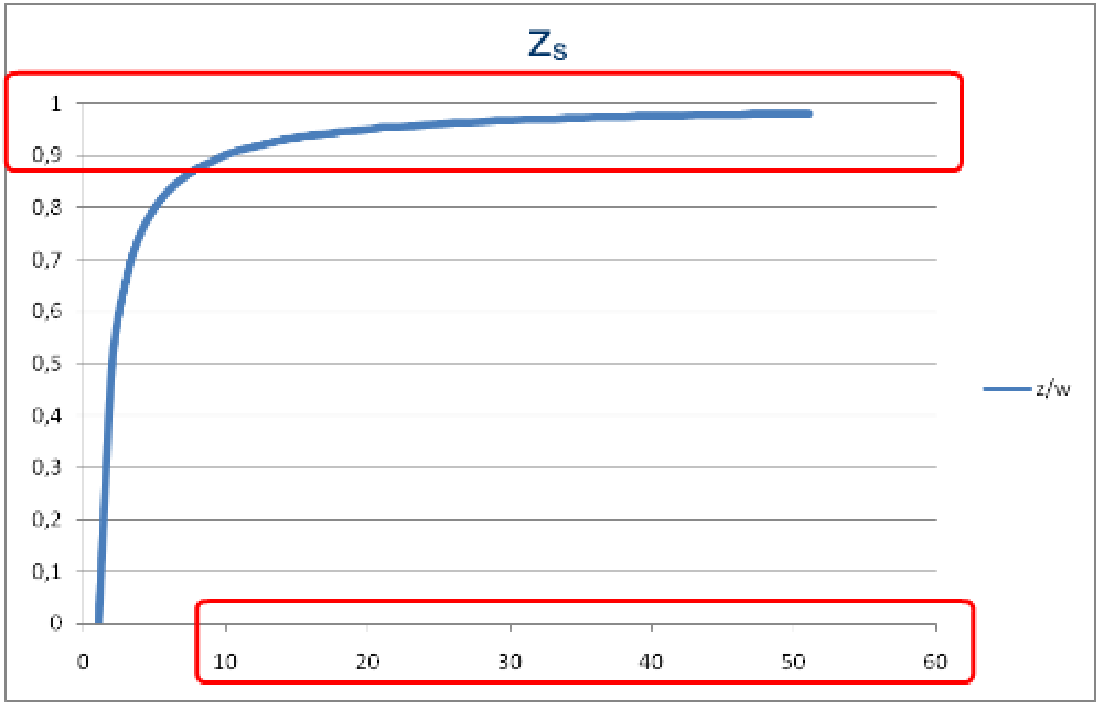
Z-buffering: issues

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).



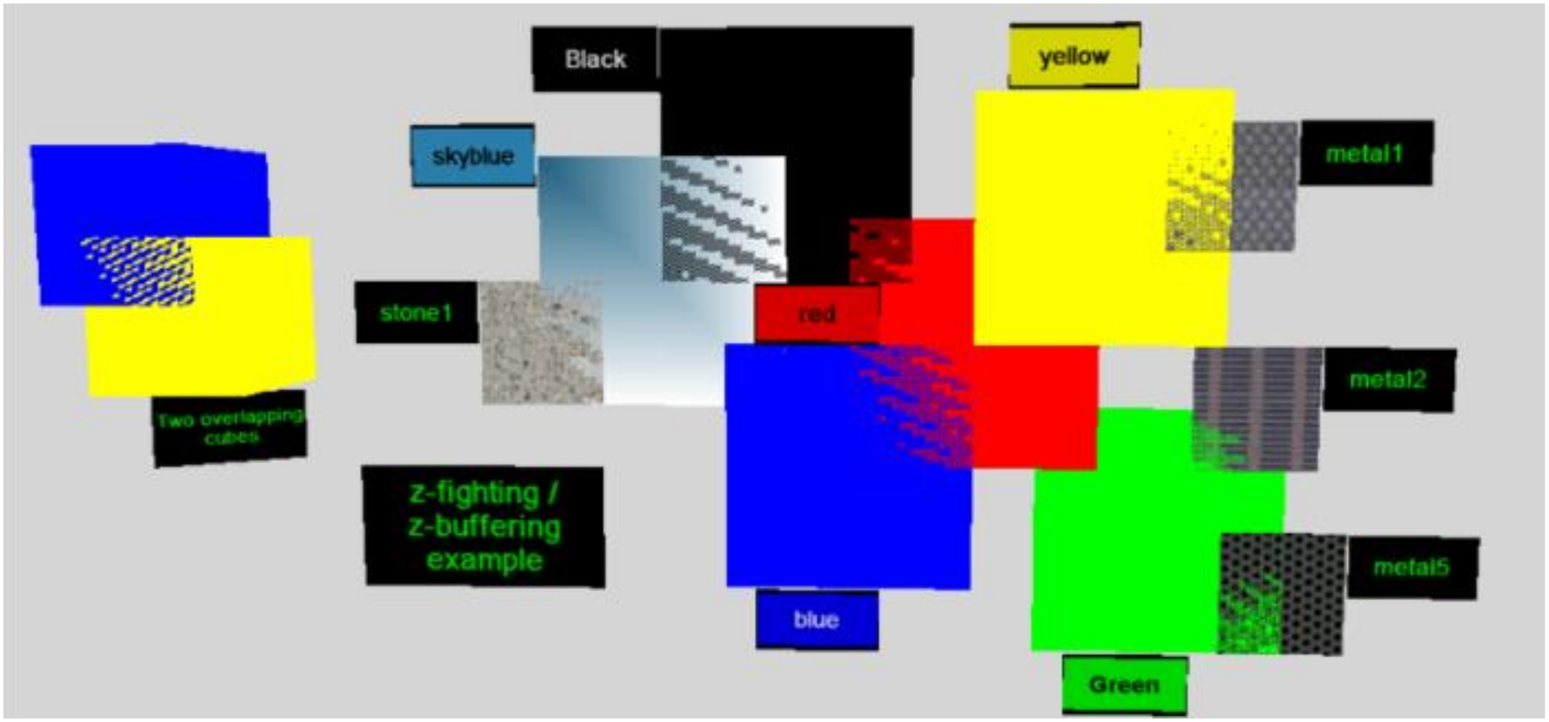
Z-buffering: issues

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



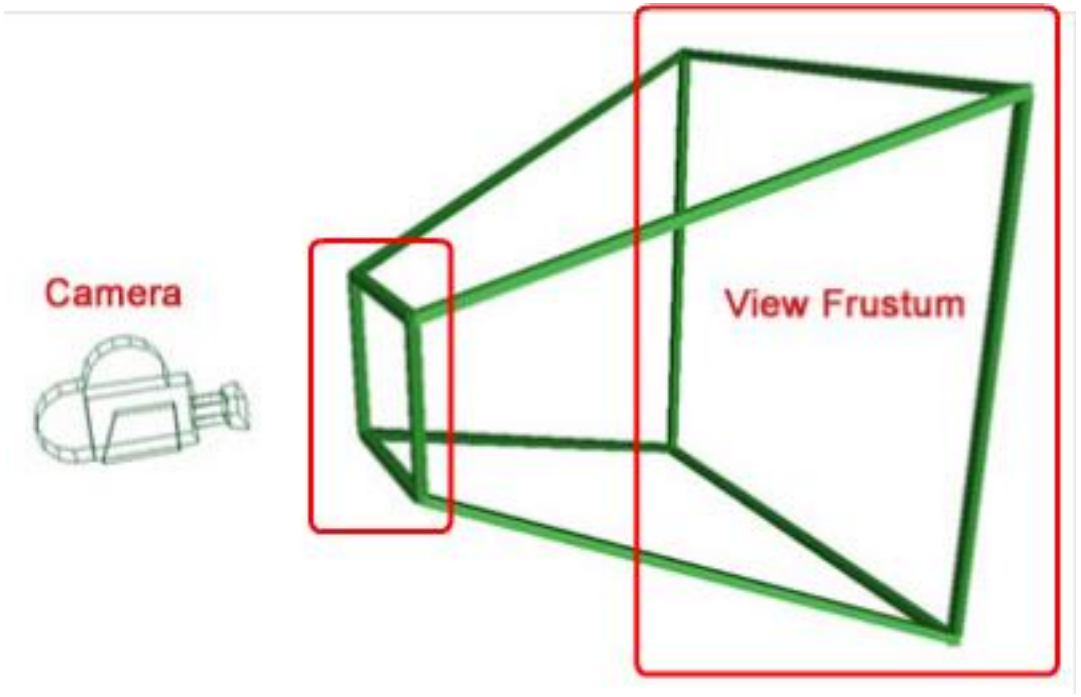
Z-buffering: issues

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* and *large*: they should always be appropriate for the considered scene.



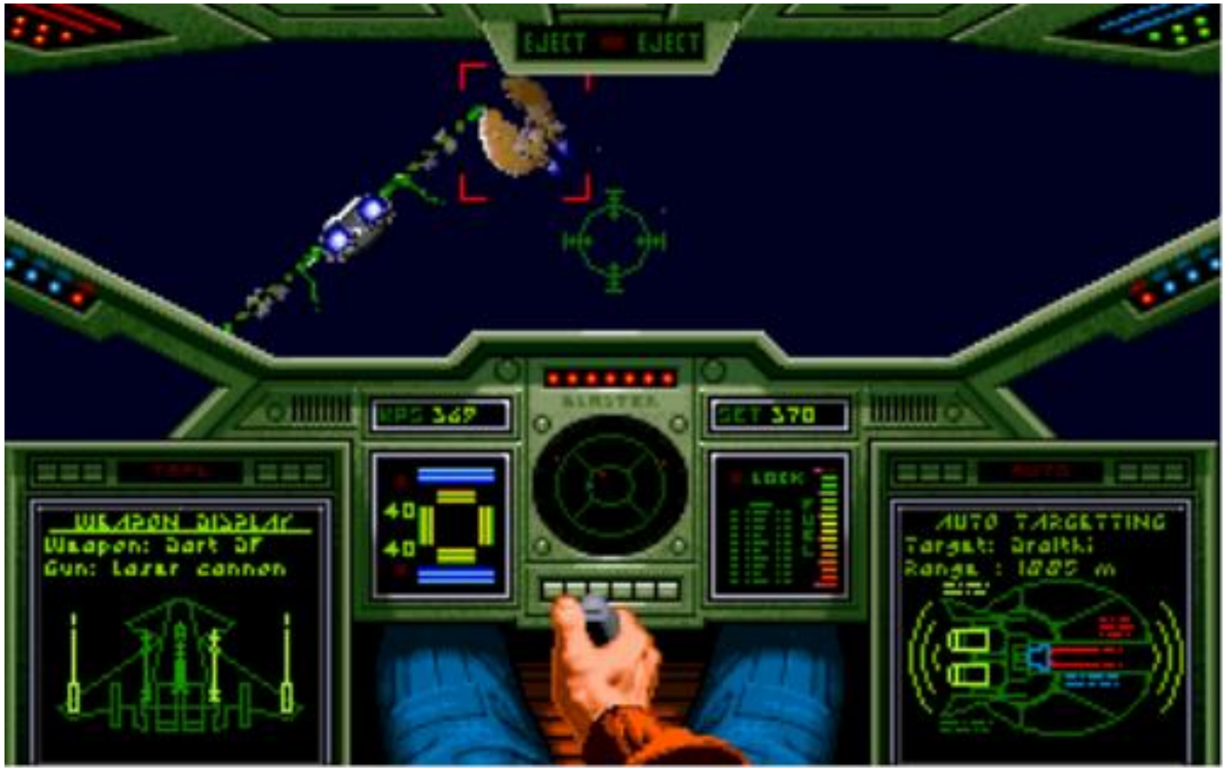
Stencil buffer

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

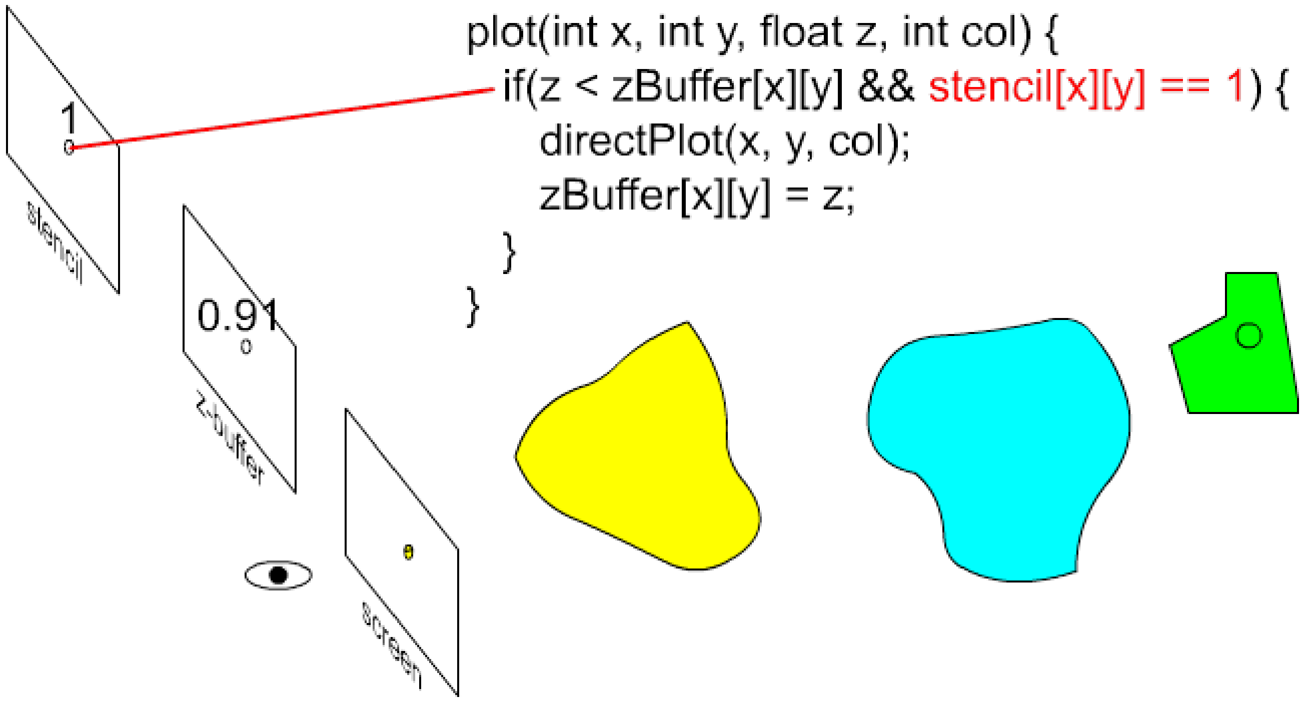
Stencil

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.

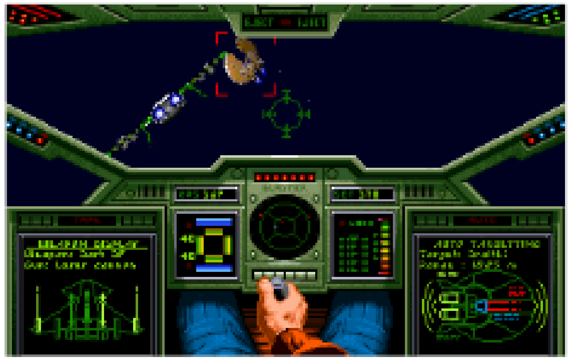
During rendering, stencil buffer data can be used to perform specific tasks on the corresponding pixels.

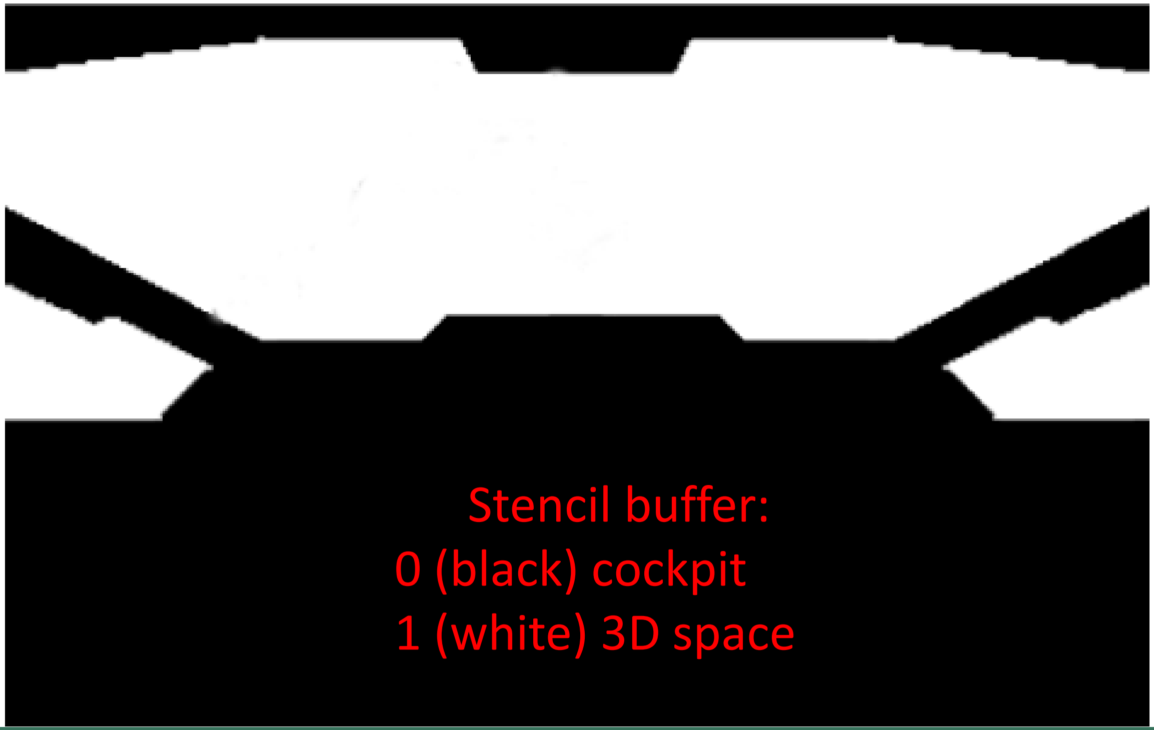


```
plot(int x, int y, float z, int col) {
  if(z < zBuffer[x][y] && stencil[x][y] == 1) {
    directPlot(x, y, col);
    zBuffer[x][y] = z;
  }
}
```

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.





Clipping

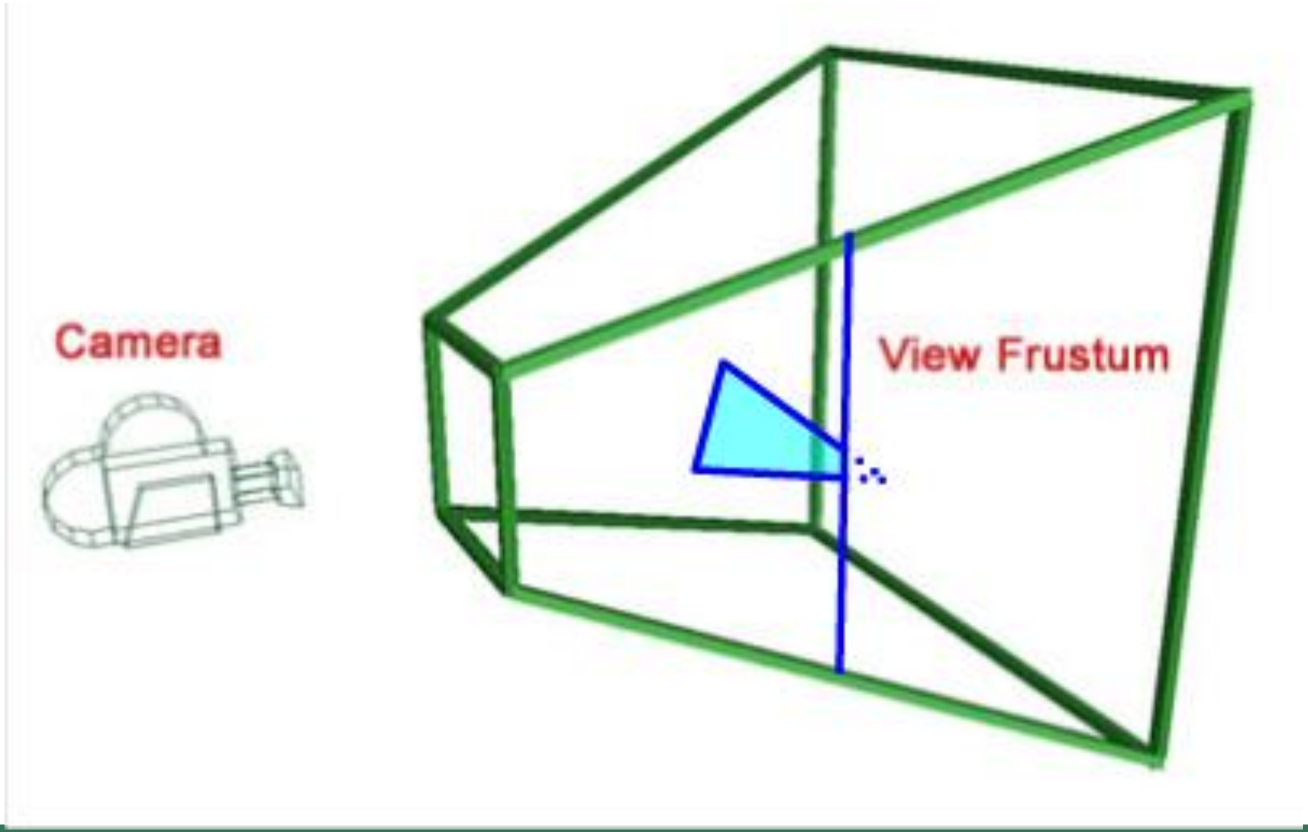
The triangles of a mesh can intersect the screen boundaries and can be only partially shown.

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

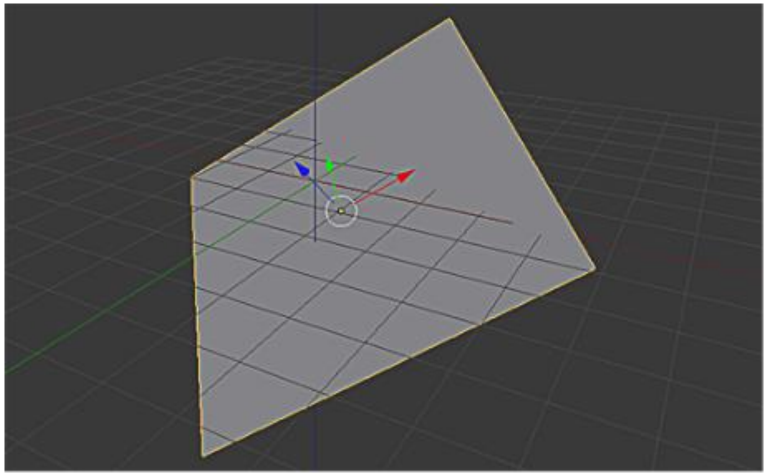
In 3D, clipping is performed against the viewing frustum.



Clipping: half spaces

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

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



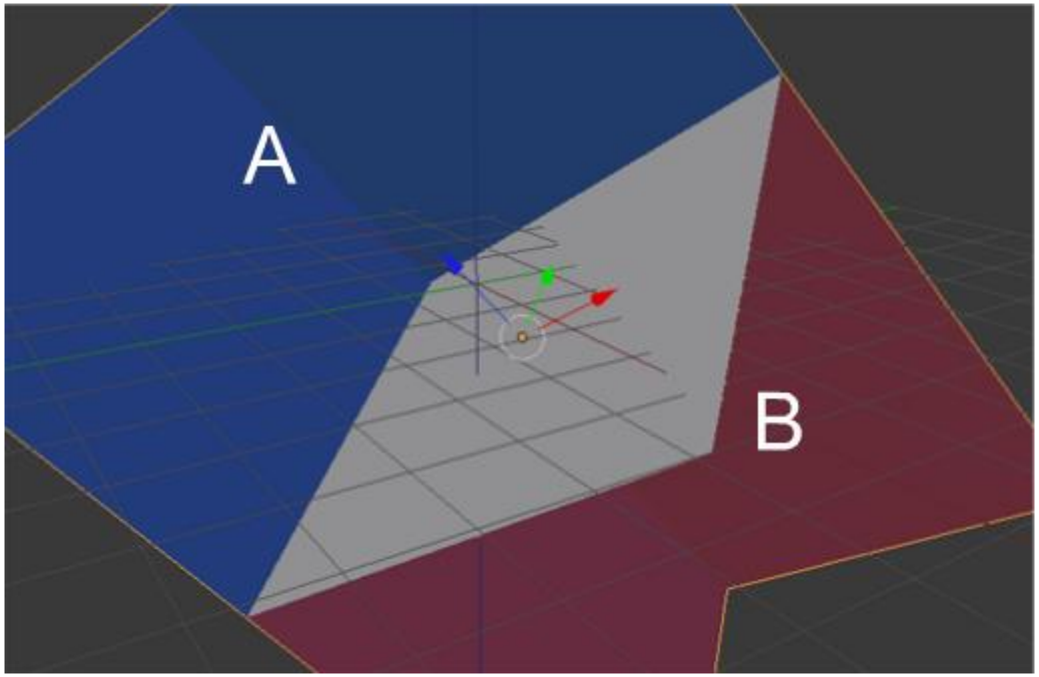
Components n_x, n_y, n_z represent the direction of the normal vector to the plane.

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

Clipping: half spaces

When expressed as an inequality, the equation divides the space into two regions, called *half-spaces*.

$$n_x \times x + n_y \times y + n_z \times z + d > 0$$



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.

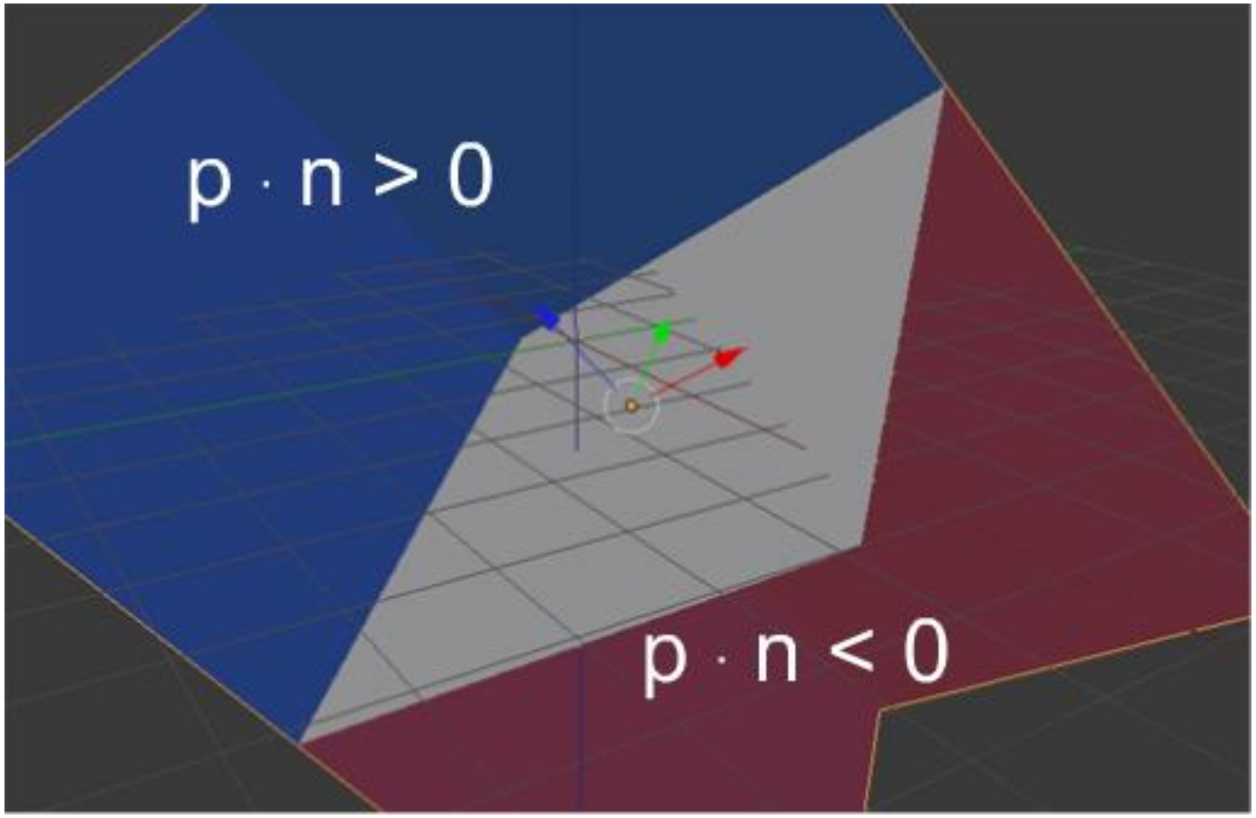
$$n = (n_x, n_y, n_z, d)$$

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

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

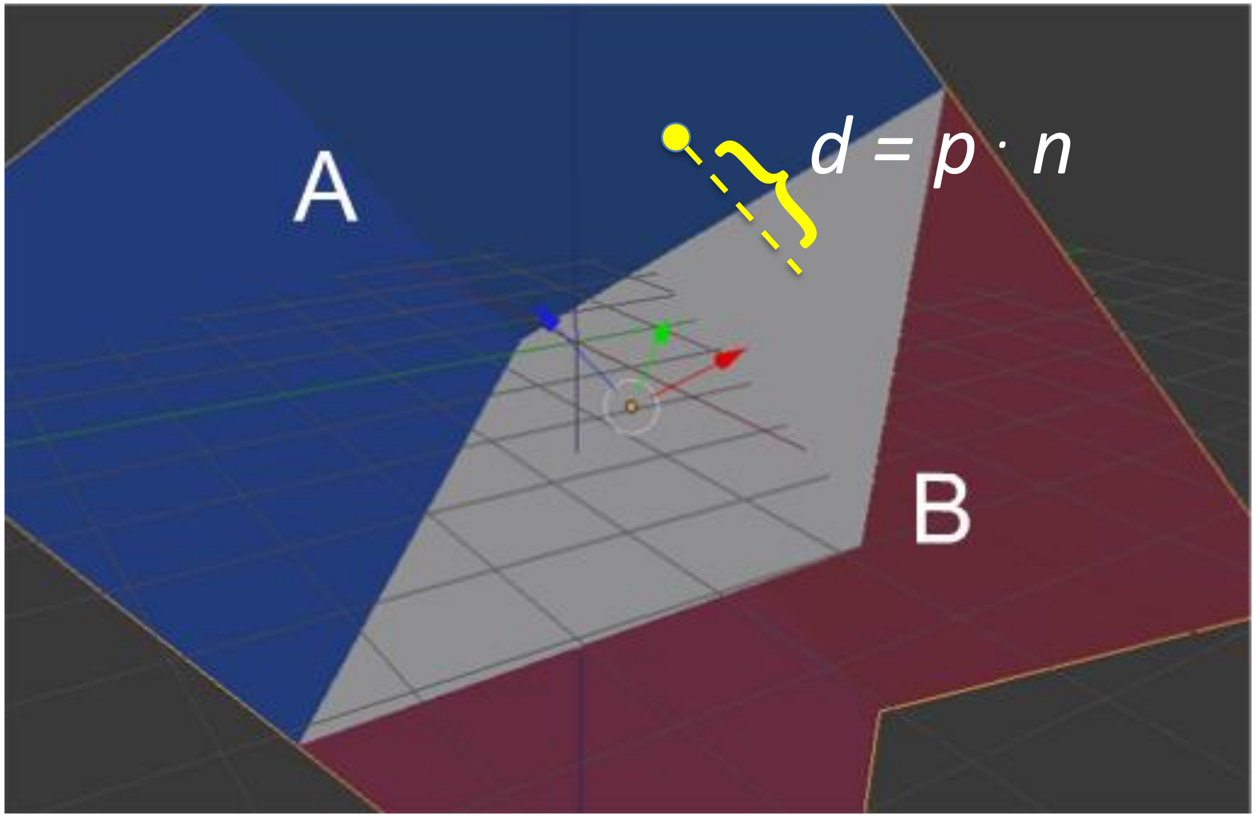
Clipping: half spaces

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



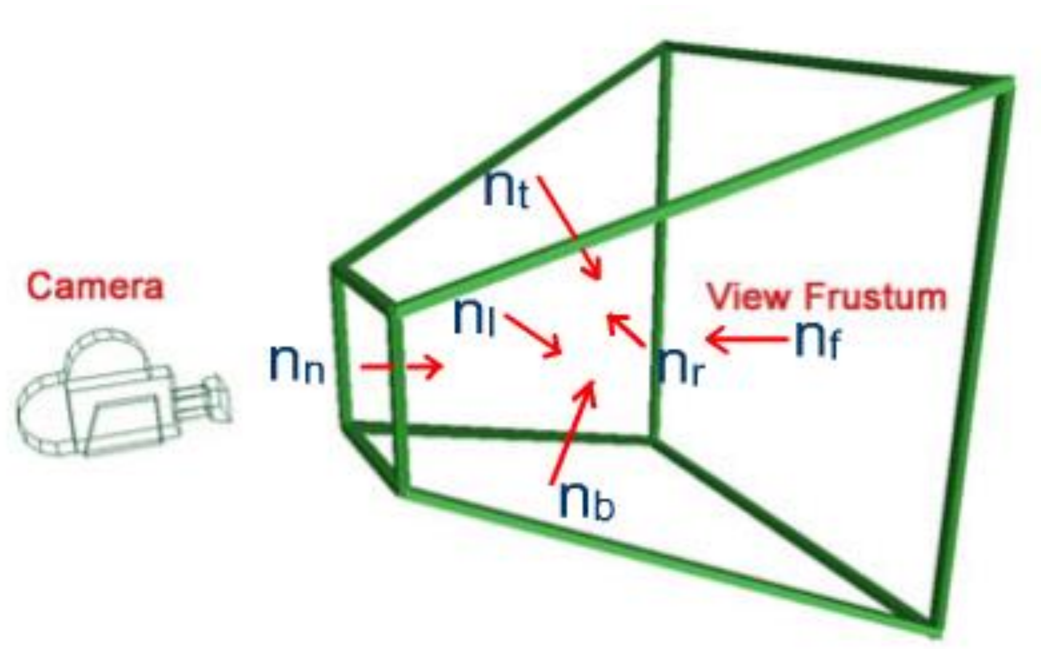
Clipping: half spaces

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



Clipping: half spaces

Since a frustum is a convex solid, it can be defined as the intersection of the six half-spaces that correspond to 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* (*x* and *y*), or *0* and *1* (*z*), the six vectors are the following:

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

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

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

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

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

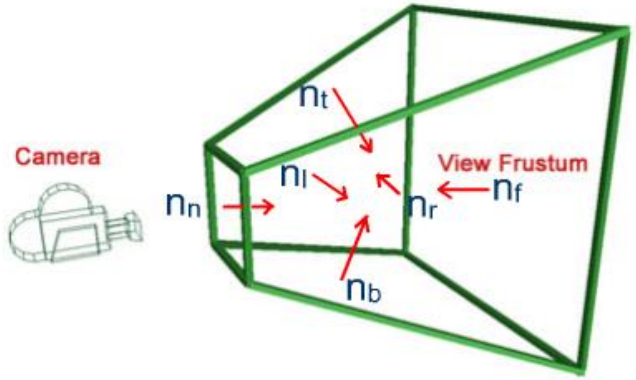
$$\frac{z}{w} < 1, \qquad z < w, \qquad -z + w > 0, \qquad n_f = |0 \quad 0 \quad -1 \quad 1|$$

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.

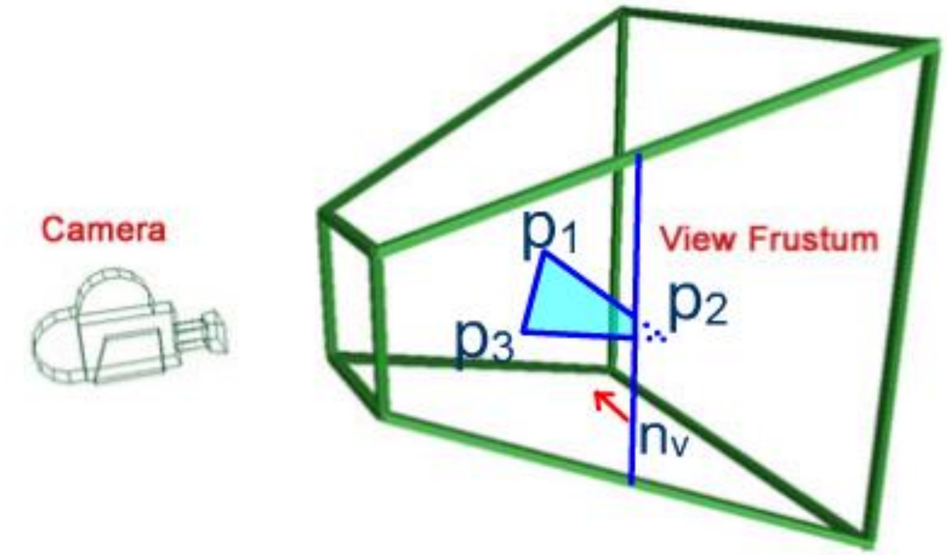


$$p \times n_v > 0$$
$$\forall v \in \{l, r, t, b, n, f\}$$

Clipping: triangles

When considering triangles, things are a more difficult.
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.



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

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.

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

Trivial accept (for the side)

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

Trivial reject

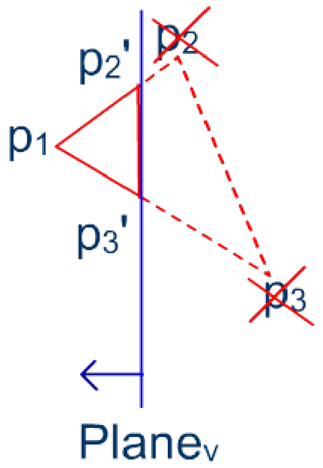
Clipping: triangles

If two points are outside the frustum, say p_2 and p_3, then two intersections p_2' and p_3' of the segments connecting them to p_1 with the plane 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' .

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

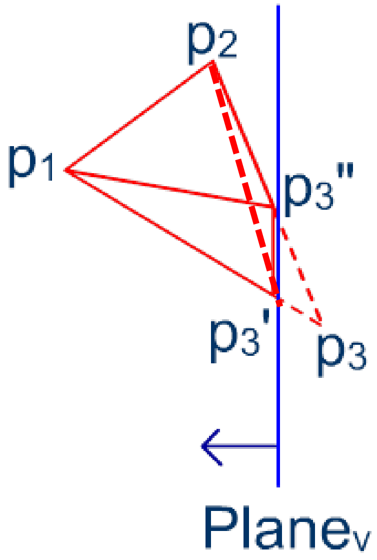


Clipping: triangles

If just one point is outside, say p_3, two intersections p_3' and p_3'' on the segments that connect 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.

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

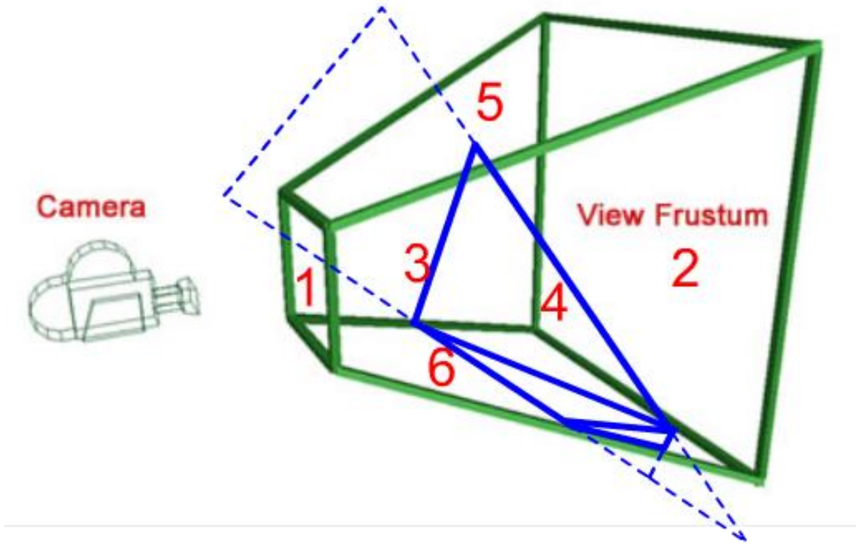


Clipping: triangles

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 all the parameters assigned to vertices which, as we will see in the following lessons, can be a lot (e.g. normal vectors, colors and UV coordinates).

Clipping: final remarks

Although clipping is generally performed by the drivers in way that is transparent to the user, knowing how it works can help to understand one of the important steps that occurs when drawing something on screen.

Moreover, the techniques just presented can be adapted to solve many intersection problems that can occur in an application.

Extremely advanced users can control the clipping distance of triangles to obtain interesting effects: this opportunity is however outside the scope of this course.

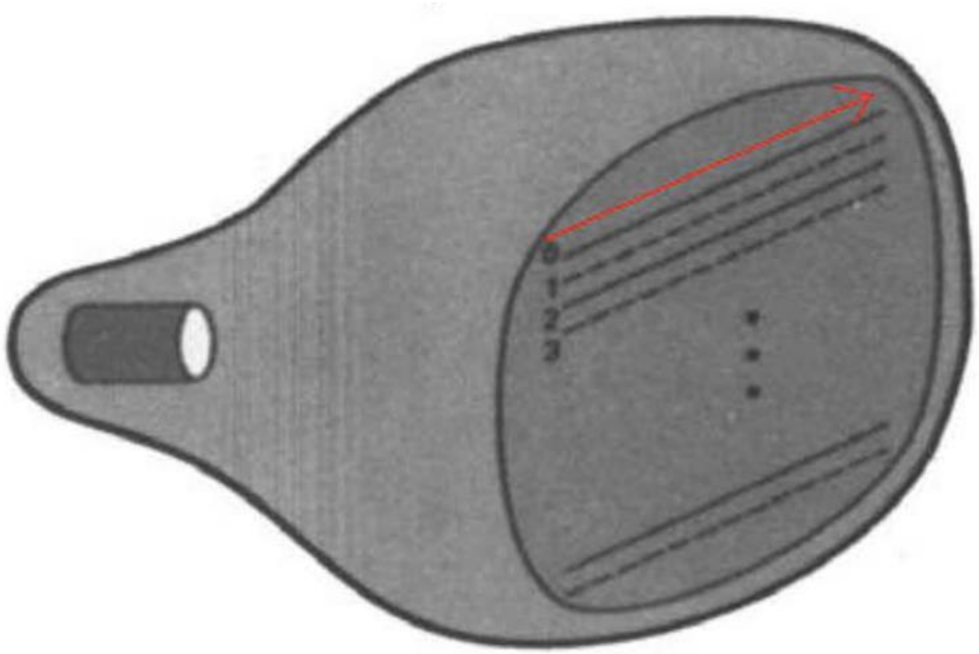
Screen Synchronization

The CPU, the GPU and the screen update process occur at different and independent speeds.

If the program is not written properly, this may cause flickering in animations.

Screen Synchronization

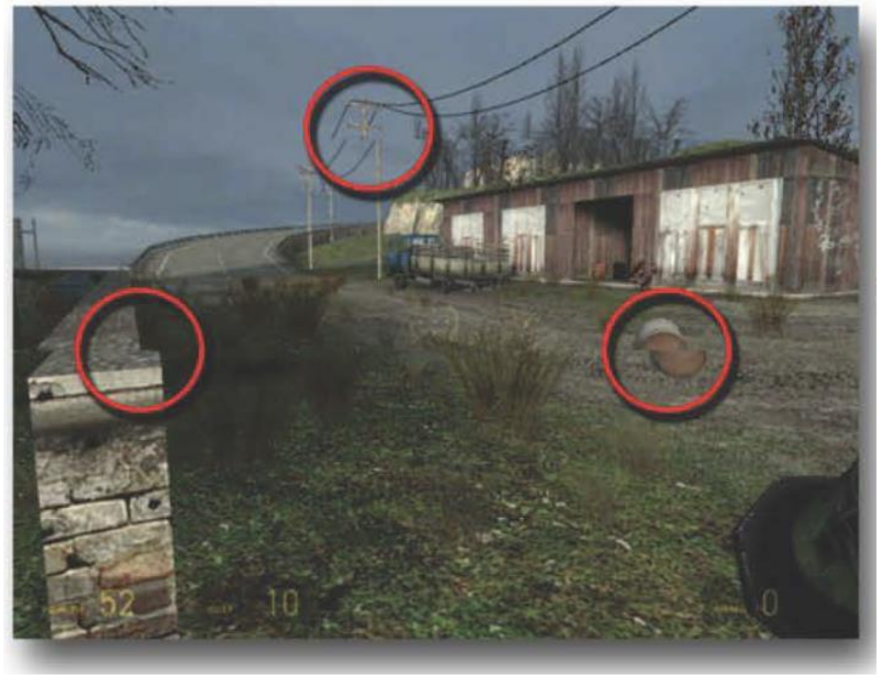
Monitors and display devices compose the images by updating their pixels in a predefined order (usually, scanning horizontally left to right, and top to bottom).



Screen Synchronization

Flickering may result in a "break" into the animation **(tearing)**.

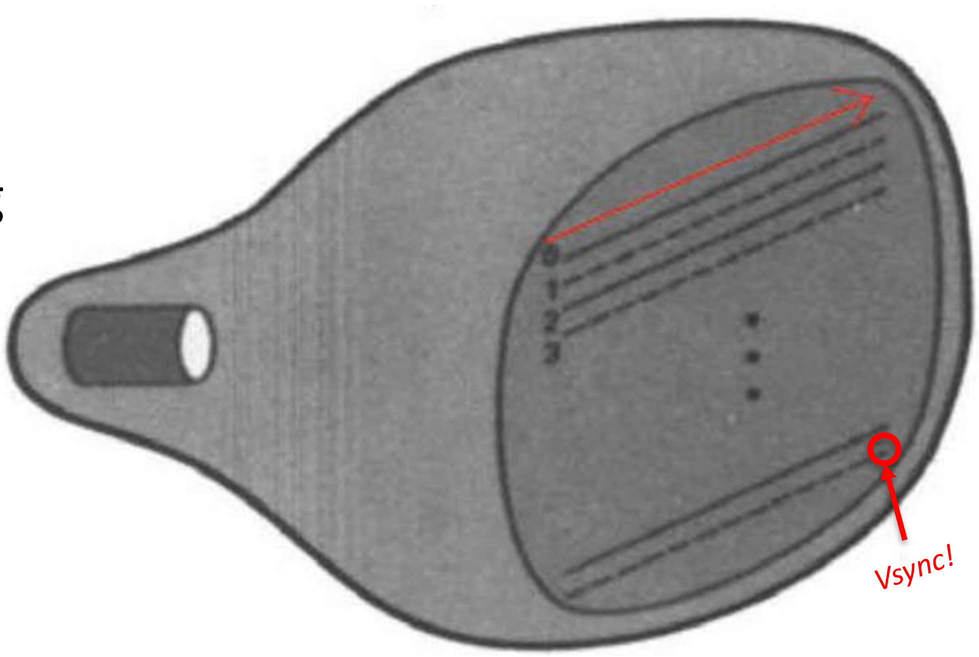
This happens because the graphic adapter reads the video memory when the program has not finished yet composing the image.



Screen Synchronization

Every graphic adapter sends an interrupt to the processor called *Vsync* whenever it finishes tracing the screen.

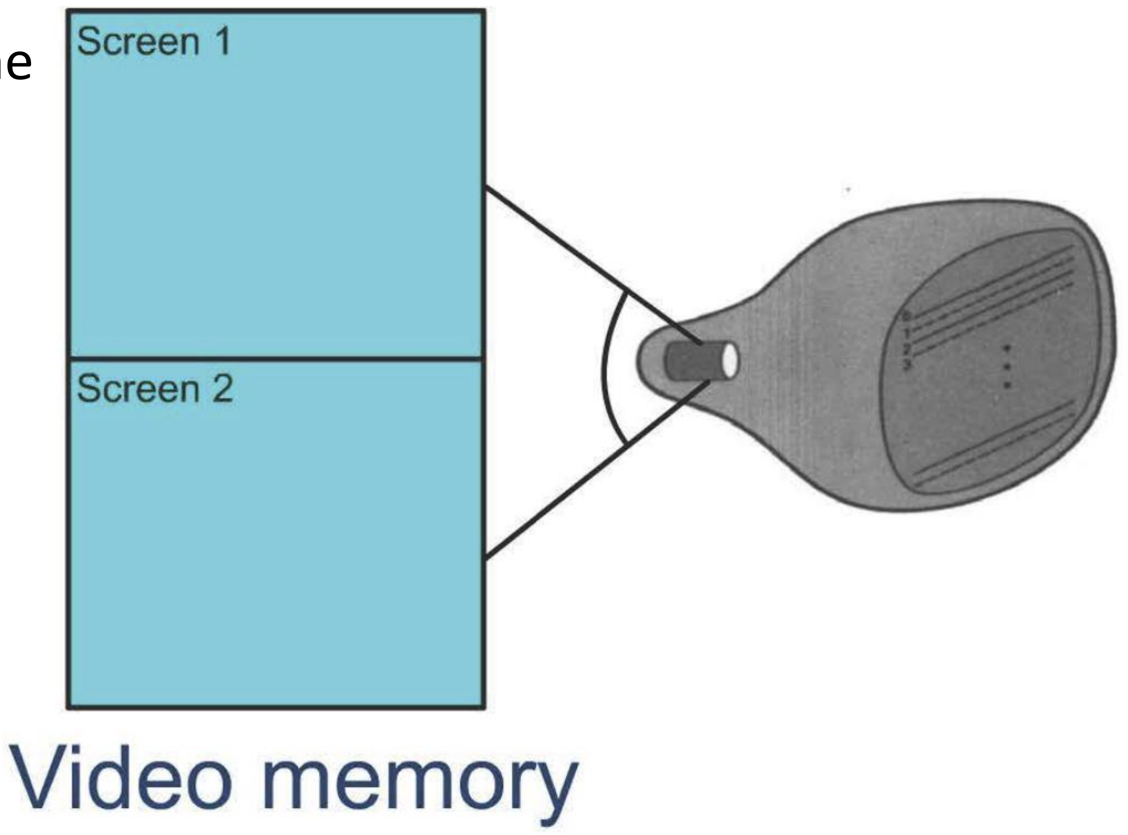
The application can intercept this interrupt, and start updating the background only when the *Vsync* signal is received.



Double Buffering

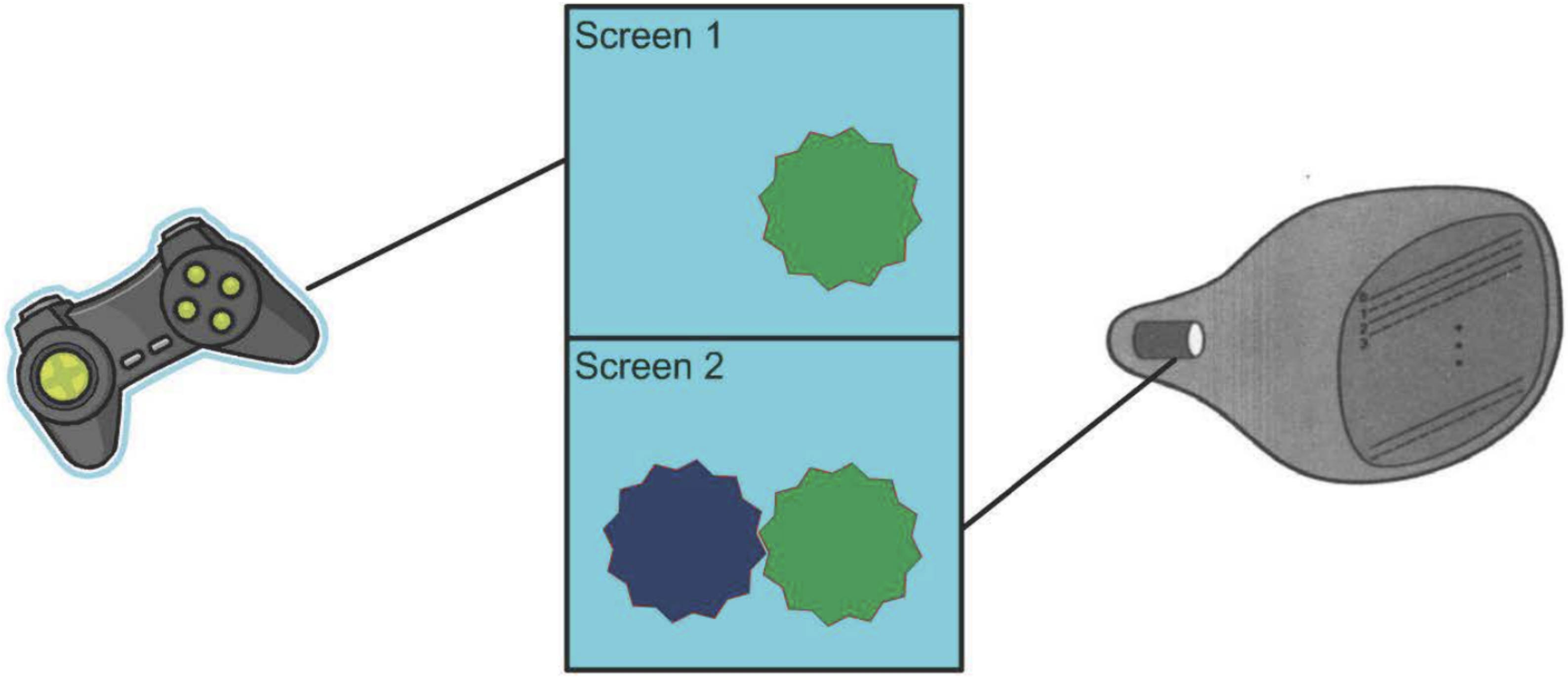
With *Double Buffering*, the video memory has two frame buffers: the *front buffer* and the *back buffer*.

While the video adapter is showing the content of one buffer, the application can compose the image in the other.



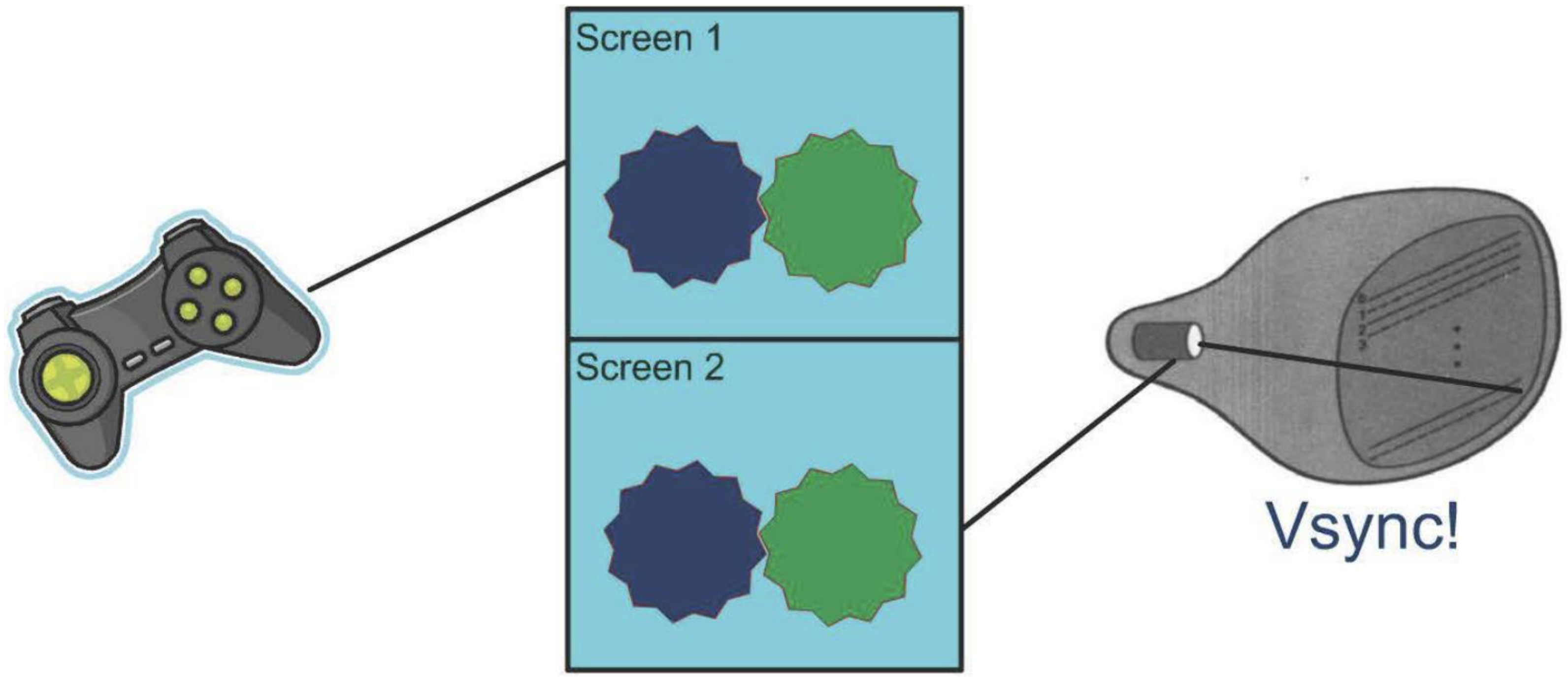
Double Buffering

In the beginning the application works on one screen (the *back buffer),* while the video adapter is showing the other one (the *front buffer).*



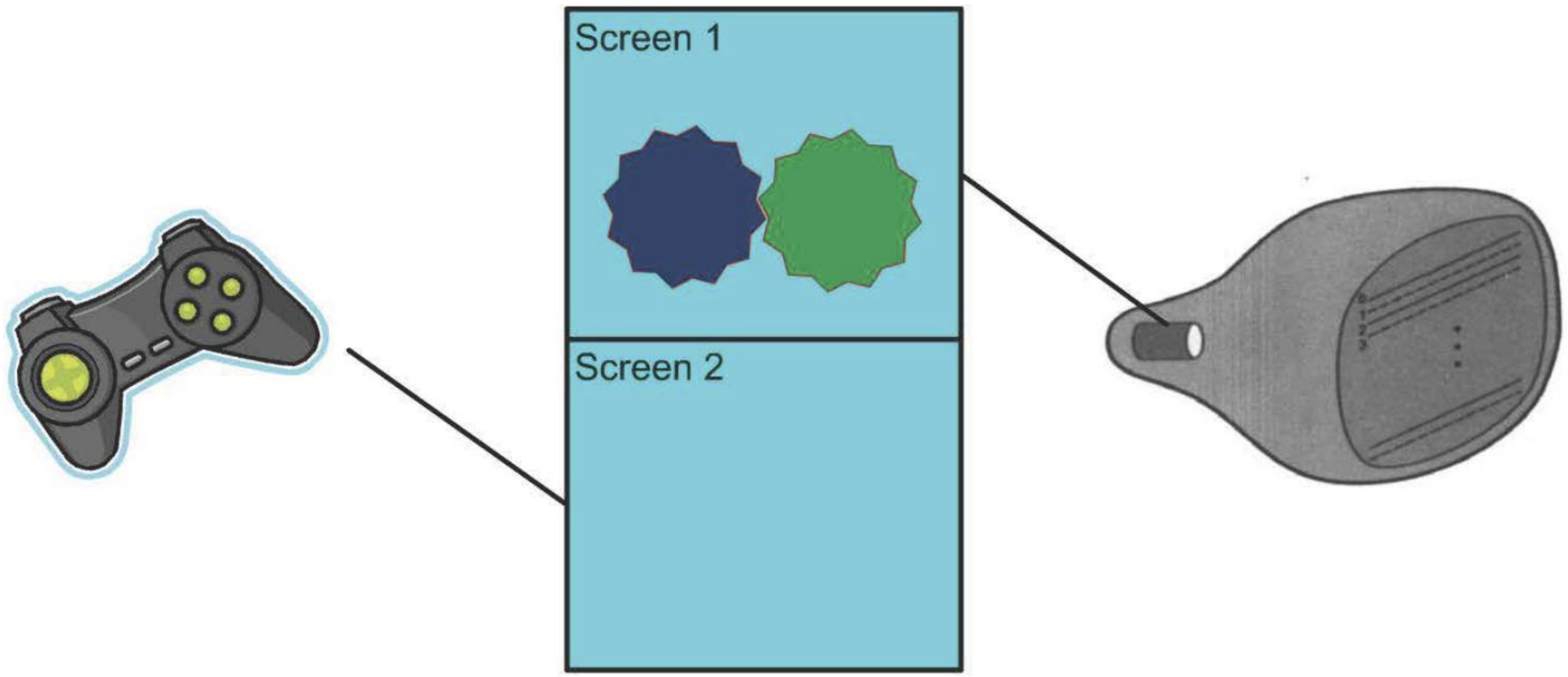
Double Buffering

As soon as the new frame has been composed, the application waits for the Vsync signal from the monitor.



Double Buffering

It then swaps the front with the back buffer, and starts composing the new frame while the monitor is showing the one just finished.



Triple Buffering

Double Buffering, although being very effective, has several drawbacks.

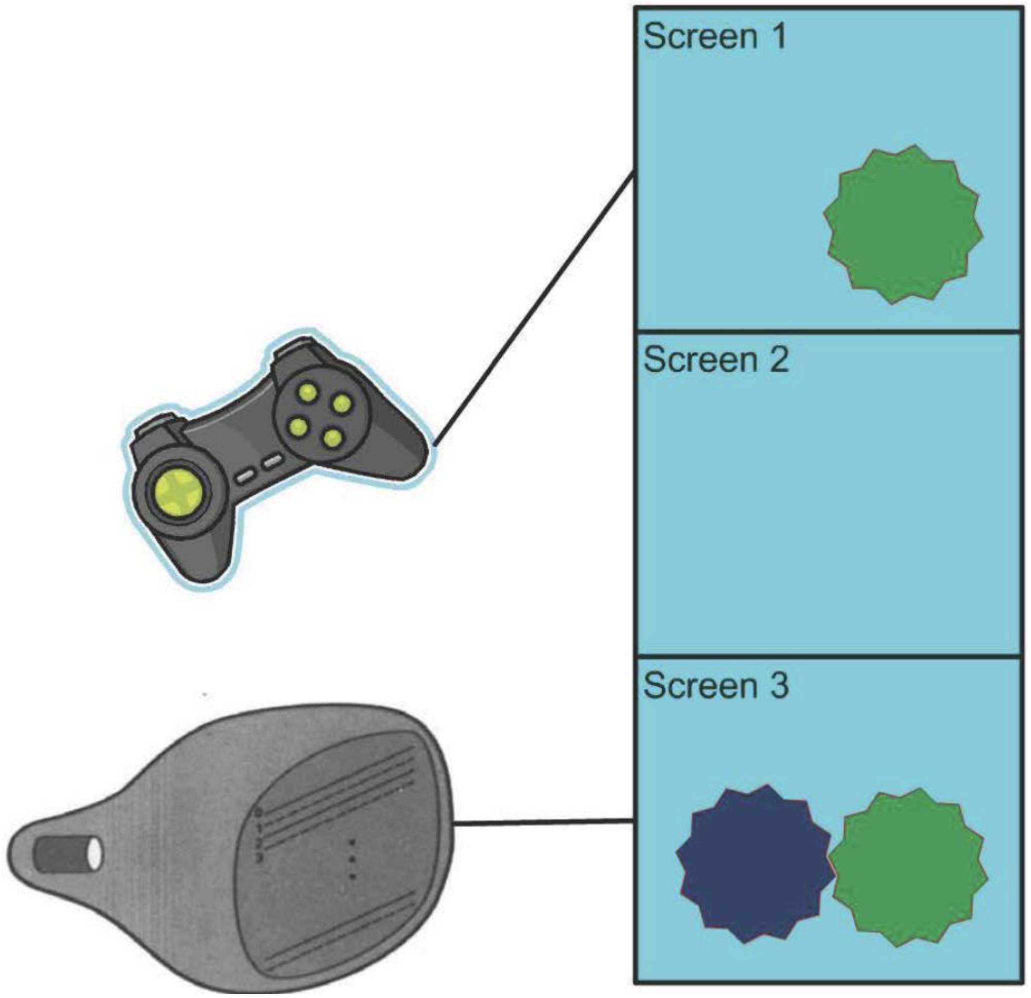
The application must stop composing the image as soon as a screen is completed, and wait for the *Vsync* interrupt to continue.

This limits the frame rate to the one of the monitor, and creates locks in the application, reducing the utilization of both the CPU and the GPU.

Triple Buffering solves this issue by allowing complete independence between the application and the presentation.

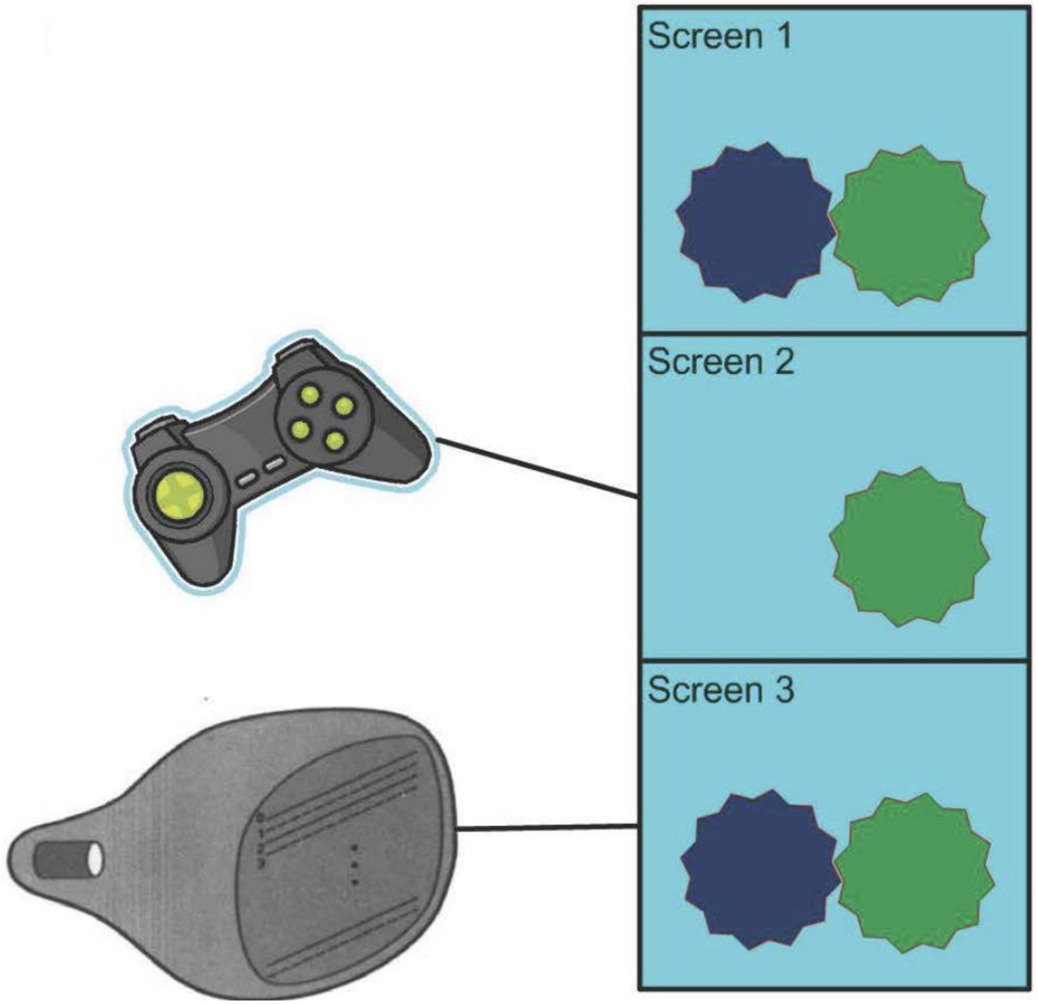
Triple Buffering

Initially, the application works on one frame buffer, while the system is displaying another.



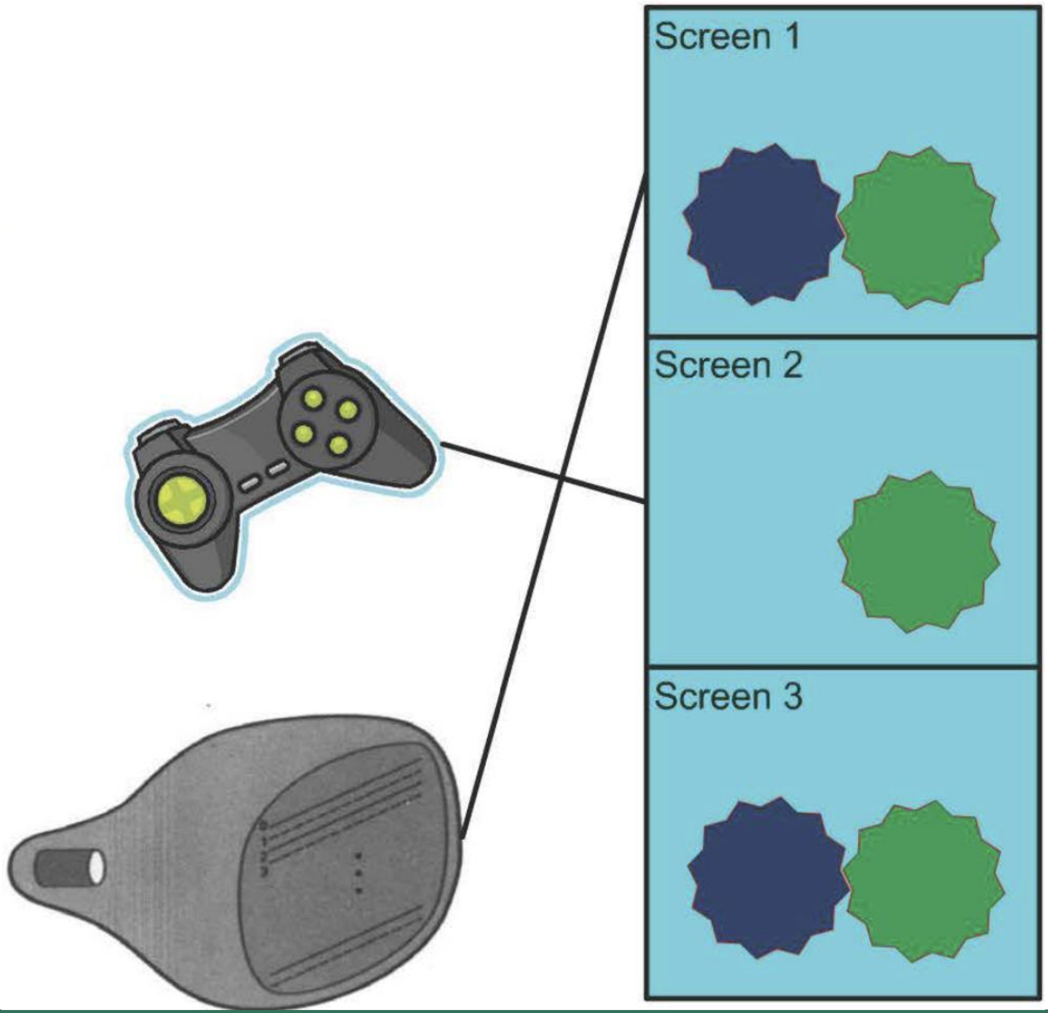
Triple Buffering

As soon as the application has finished composing the screen, it starts working on the next one in the buffer currently not used by the presentation.



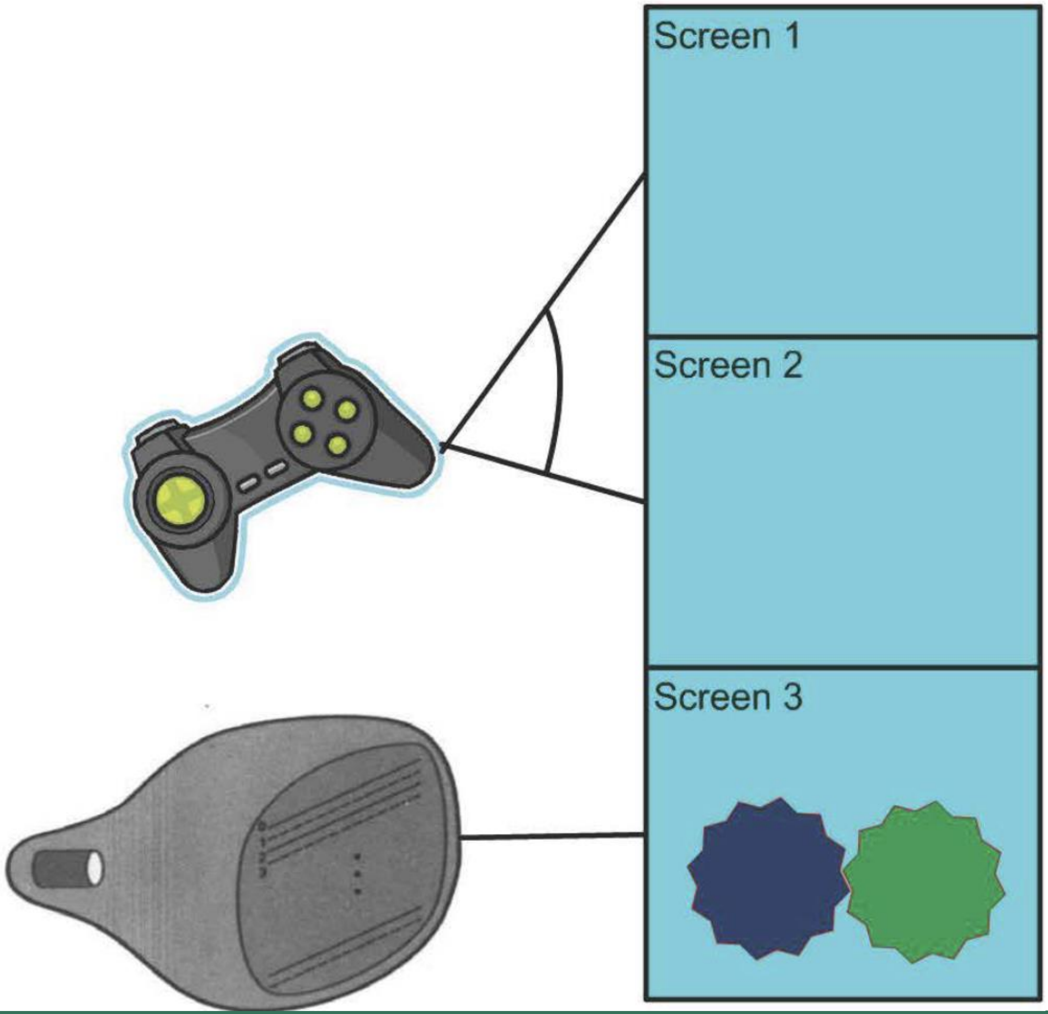
Triple Buffering

At the next *Vsync* signal, the presentation shows the last frame fully composed (the one where the application is currently not working).



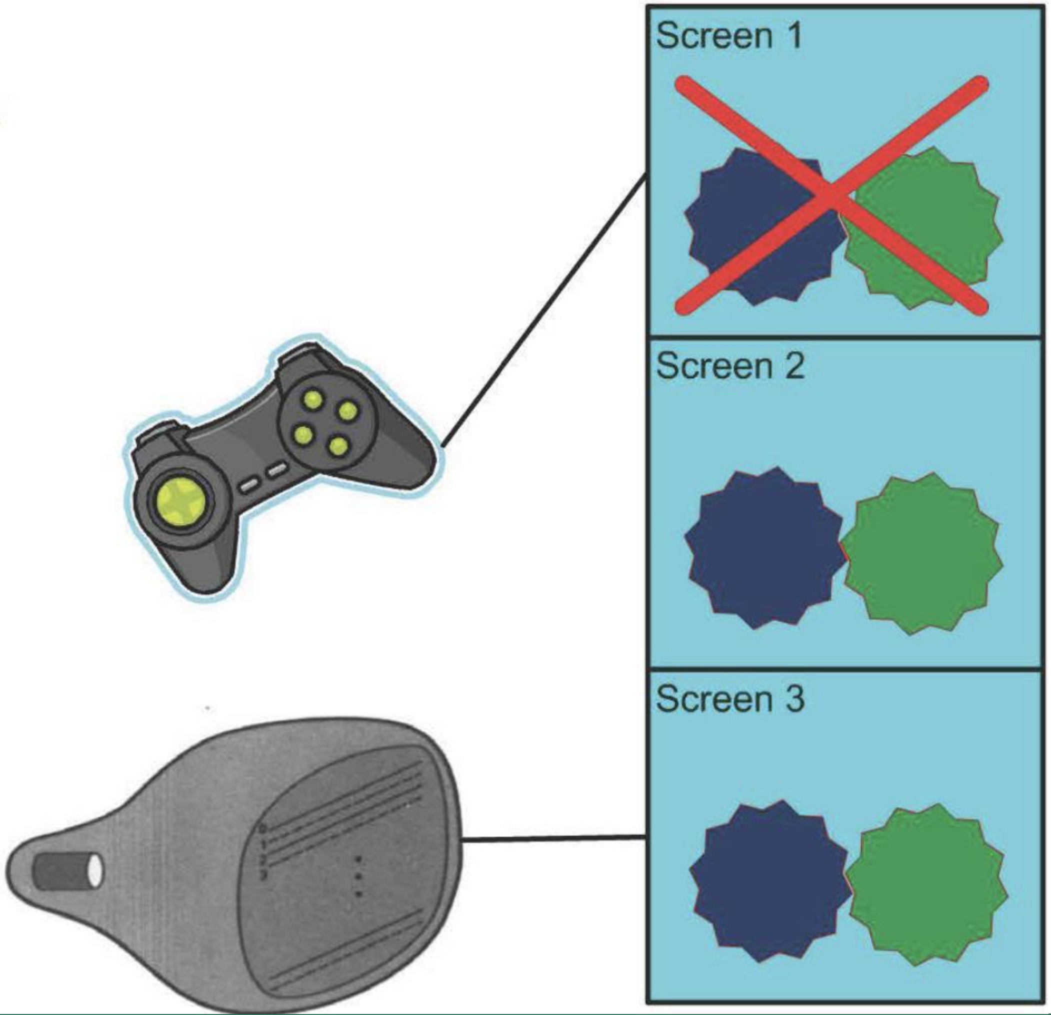
Triple Buffering

The application can switch as many times as needed between the two back buffers not currently shown, while waiting for the *vSync.*



Triple Buffering

If a frame is completed before the *vSync* is received, the screen previously generated is *skipped* (it is never displayed).



Triple Buffering

Frame skipping allows having frame rates higher than the one of the display, by automatically discarding frames not used.

Swapping at the Vsync, allows for smooth animations and prevents the tearing effect.

The drawbacks of Triple Buffering are the *memory requirements* (at least three frame buffers are needed), and the *unwise resource utilization* that might cause both the CPU and the GPU wasting computation time (and energy) on a lot of frames that will be discarded.

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)