**3D Game Engine**

**Window System**

Window class lets us initialize, show and update a window on the screen

**Graphics Engine**

Initialize DirectX resources that will allow us to execute DirectX drawing functions

**Swap Chain**

Used by the engine to show the rendered frames on the screen and for frame rate stabilization. A swap chain consists of a series of **frame buffers** (memory buffers containing data needed to represent all pixels in a frame).

The first framebuffer, the **screen/front buffer**, is the buffer that is rendered to the output of the video card. The remaining buffers are known as **back buffers**. Each time a new frame is displayed, the first backbuffer in the swap chain takes the place of the screenbuffer, this is called presentation or swapping.

The SwapChain of this project will have 2 frame buffers, allowing us to use the **DOUBLE BUFFERING** technique: when we call a draw function from DirectX, first we draw on the back buffer. After all drawing operations are completed, the back buffer is flipped, or copied into the front buffer that resides on video RAM (this process consists of simply swapping the pointer to back buffer with the pointer to the front buffer). While the frame is copied, it also gets rendered on the screen, since this copy is usually synchronized with the (monitor) rasterization of the frame.

Rasterization: the task of taking an image described in a vector graphics format (shapes) and converting it into a raster image (a series of pixels, dots or lines, which, when displayed together, create the image which was represented via shapes).

**Device Context**

The Device Context is an extension of the DirectX device, the purpose of which is mainly the one of generating rendering commands, such as DrawTriangle(…); These commands are sent to the *video driver* for execution. The driver will then redirect commands to the CPU and/or GPU for the elaboration and the final rendering on the screen (or better, on the render target). A **render target** can be viewed as a blank canvas in which we can, through the device context, draw anything we want/need. More precisely, it’s a memory buffer residing on video memory so it can easily be mainuplated by the GPU. An example of render target can be the back buffer of the swap chain, where we draw the scene that will be then transferred to the front buffer for final presentation on the monitor. We can create our own render targets for anything we want, so that specific parts of the scene can be drawn on specific target buffers (for example only drawing the UI on a specific target buffer).

The Device Context also allows us to set **pipeline states**, which influence the final results of our rendering commands.

There are two types of Device Context:

* IMMEDIATE context
  + Allows the immediate execution of (a single, or a list, of) rendering commands.
  + It sends the commands to the selected driver for elaboration, with no delay. Renders directly to the driver (e.g. with an Nvidia Card, it calls sub-routines of Nvidia Driver for rendering)
  + There is only ONE immediate context that we will get after the creation of DirectXDevice.
* DEFERRED context
  + Records the rendering commands launched by the user in a list, to be executed at another time.
  + Designed to exploit the multithreading system and so the simultaneous generation of rendering commands.
  + We can have zero or more deferred contexts.

In the end, we must use the immediate context to actually execute the command list generated by the various deferred contexts.

**Drawing a Triangle**

Triangle = polygon composed by 3 vertices.

Vertex = points in the space

In most graphics APIs, a vertex can be defined with a lot of different attributes, and it can be customized, adding attributes such as position, color, texture coordinates, etc. Different combinations of attributes create different **vertex types**.

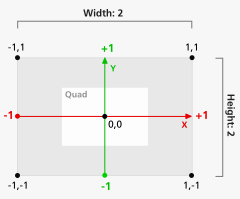
Vertices are used to render a triangle by adding them to the **vertex buffer** = a memory buffer containing a list of vertices of any size. Each vertex buffer can only contain vertices of the same type.

We need a way to tell our graphics API what attributes our vertex type is composed of.

This can be accomplished by the **Input Layout** object. In DirectX, an Input Layout is an object where we can add all necessary information that define the various attributes of our vertex type (names, bytes size, displacement inside the vertextype structure, and so on).

Most of the time, we will deal with vertices placed in a 3D space, and to allow the graphics API to process them and show the polygons on the screen we need to transfer them into screen space, through the vertex shader (future tutorials).

For now, we will directly place the vertices of our triangle directly in the Screen Space coordinates.

Screen Space= a 2D space in which there are 2 axes (x-axis = width, y-axis = height), having values in the range of [-1, 1].

After setting the vertices there is nothing to do but call the Draw() function to render the triangle.

Calling the Draw() function starts a process called **Graphics Pipeline**.

**The Graphics (Rendering) Pipeline**

A pipeline is a series of data processing units arranged as a chain, with the output of the one unit read as the input of the next. Pipelines are ideal for performing identical operations on multiple sets of data as is often the case with computer graphics.

The graphics pipeline implements the processing stages of the rendering process, which include vertex processing, clipping, rasterization and fragment processing. The purpose of the graphics pipeline is to process a scene consisting of objects, light sources and a camera, converting it to a two-dimensional image (pixel elements) in 5 stages:

* ***Input-Assembler***: in this stage we must set vertex buffers and input layouts so that all vertices can be collected and used.
* ***Programmable Shaders***: in this stage we must set shaders, such as the vertex shader, for the manipulation of vertices
* ***Rasterize***: stage where vector information (composed of shapes or primitives) is converted into a raster image, composed of pixels. In this stage, we must set at least one area (or **viewport**) of a specific render target where our rasterized image will be rendered.
* ***Pixel Shader***: stage in which the final pixel color is elaborated.
* ***Output-Merger***: we will indicate which render target we want to output the final rasterized image into. In practice, the final render target will be the back buffer of the swap chain.

17:50 ep 5