Module 00

Introduction

Definition:

The term "computer graphics" describes the use of computers to create and manipulate images. This involves creating, displaying, and interacting with images and animations on a digital screen.

Computer graphics can be used in a variety of applications, including video games, film and television, scientific visualization, and advertising. It involves a combination of programming, mathematics, and artistry to create realistic and visually appealing images. As technology continues to advance, the field of computer graphics is constantly evolving and expanding.

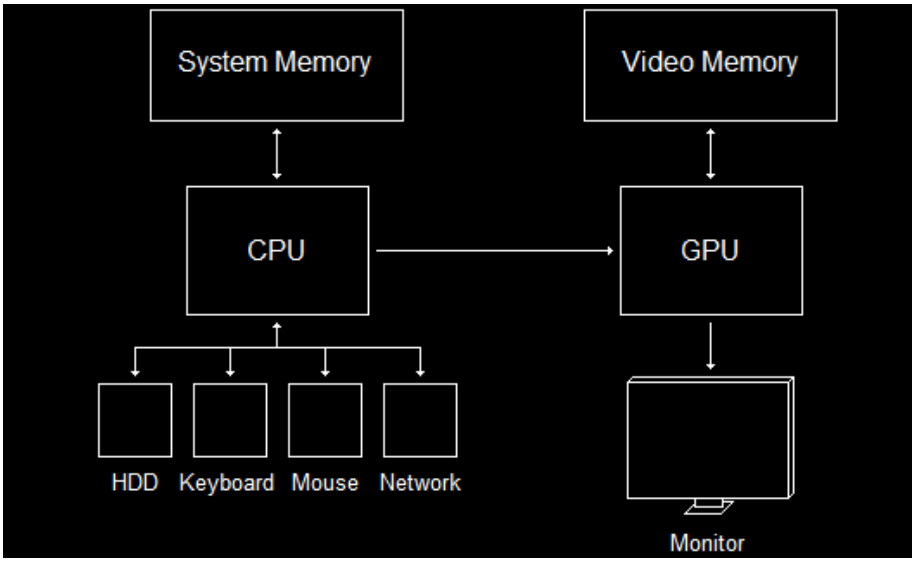
Types of CG:

- Non interactive Computer Graphics (offline rendering): This involves creating high quality images that do not require real time interactivity. The goal is to produce visually appealing and photo-realistic results by using complex rendering techniques, including global illumination, ray tracing, and shading algorithms.
- Interactive Computer Graphics (Real-time rendering): It focuses on generating images or animations in real time providing immediate feedback to user inputs or changes in the scene. Real-time rendering is commonly used applications such as video games, virtual reality, and computer-aided design (CAD) systems.

Graphics Hardware:

In graphics programming we are primarily interested in the GPU (graphics processing unit). It’s different from the CPU (central processing unit) in some structural ways and what is used for. The CPU is responsible of directing the entire computer unlike the GPU, it performs calculations on graphics and directs graphics output to the monitor.

In addition to having its own processor, Graphics programming works with a different portion of memory called video memory. It exists on the video card, so it can be quickly accessed by the GPU.



Graphics pipeline:

A graphics pipeline can be divided into 3 steps (Application, Geometry and Rasterization).

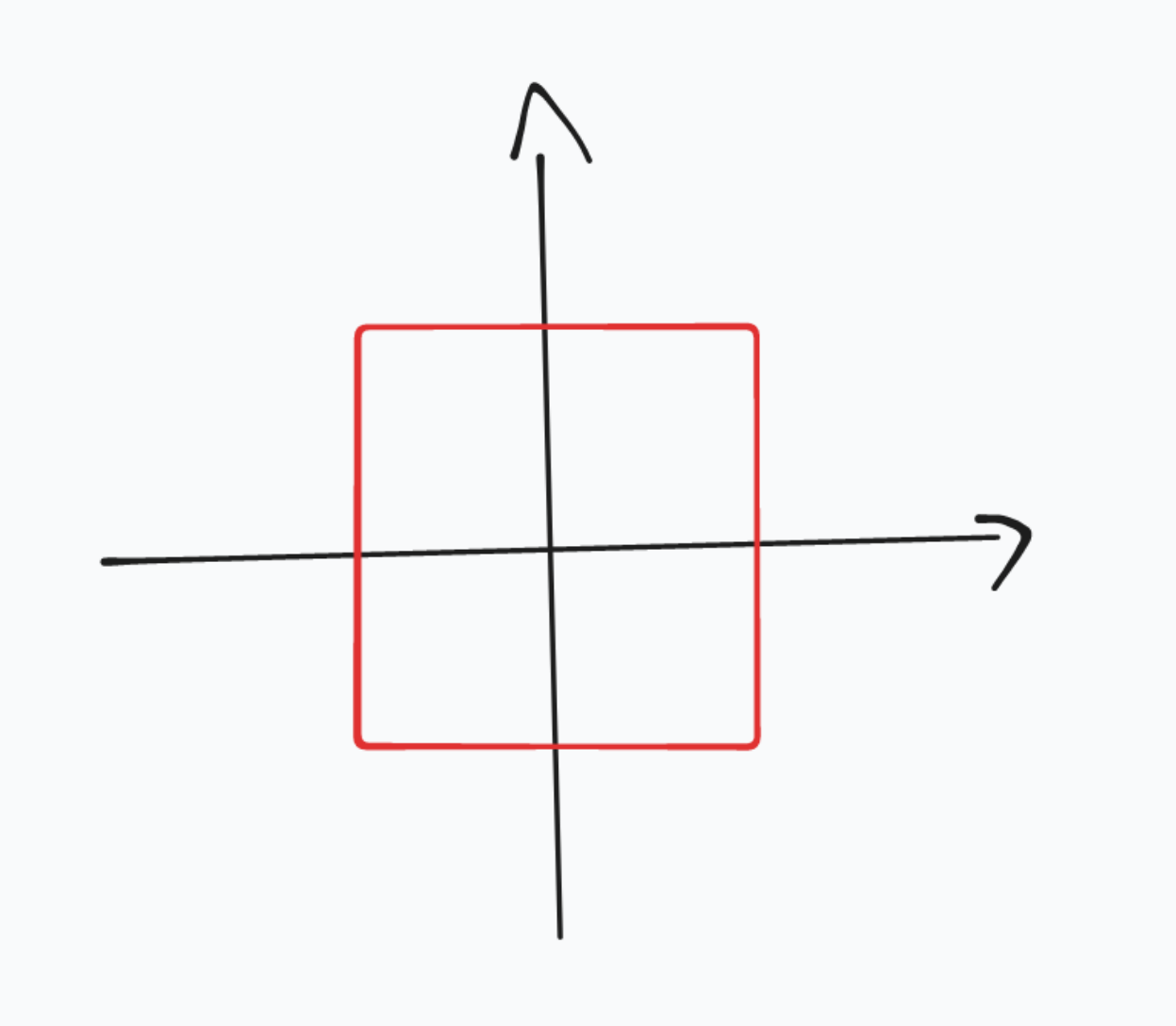
Application:

In the application stage, is where we typically provide the geometric data or models to be rendered. This stage involves specifying the vertices and attributes that define geometry of our objects.

In this stage, we define the position, color, texture coordinates, normals, and other relevant attributes for each vertex of our model. These attributes collectively make up the vertex data that will be processed and transformed in subsequent stages of the pipeline.

Overall, the application stage is where we provide the input geometry and associated data that will undergo further processing and rendering in subsequent stages of the graphics pipeline.

A model in its model space



Geometry:

In the geometry stage, the input geometry is transformed and processed to create the final output image. This stage involves operations such as transformation, projection, and culling to generate the final positions of the vertices in screen space.

- Modelling Transformations: This step involves transforming the scene objects and the virtual camera or viewer. The scene is transformed so that the camera is at its origin and facing along the Z axis. This transformation is called the view transformation.
- Illumination (shading): in this step, lighting calculations are performed to determine the illumination of objects in the scene. Light sources and material properties are taken into account to calculate the lighting effect for each vertex, interpolation is used to determine lighting values across the surface of triangles.

- Viewing Transformation (Perspective / Orthographic): The viewing transformation transforms the visible volume into a cube with specific coordinates. This transformation is often a projection, either perspective or orthographic. It maps the scene onto a 2D viewing plane and sets up clipping planes.
- Clipping: this step involves discarding primitives (objects) that are outside the visible volume. Primitives that are partially inside the frustum are clipped to fit within the visible volume.
- Projection (to screen space): The final transformation in the geometry step is the projection to screen space. This step maps the 3D coordinates to the viewport or window on the output device. It involves shifting and scaling the coordinates to fit withing the viewport.

Rasterization:

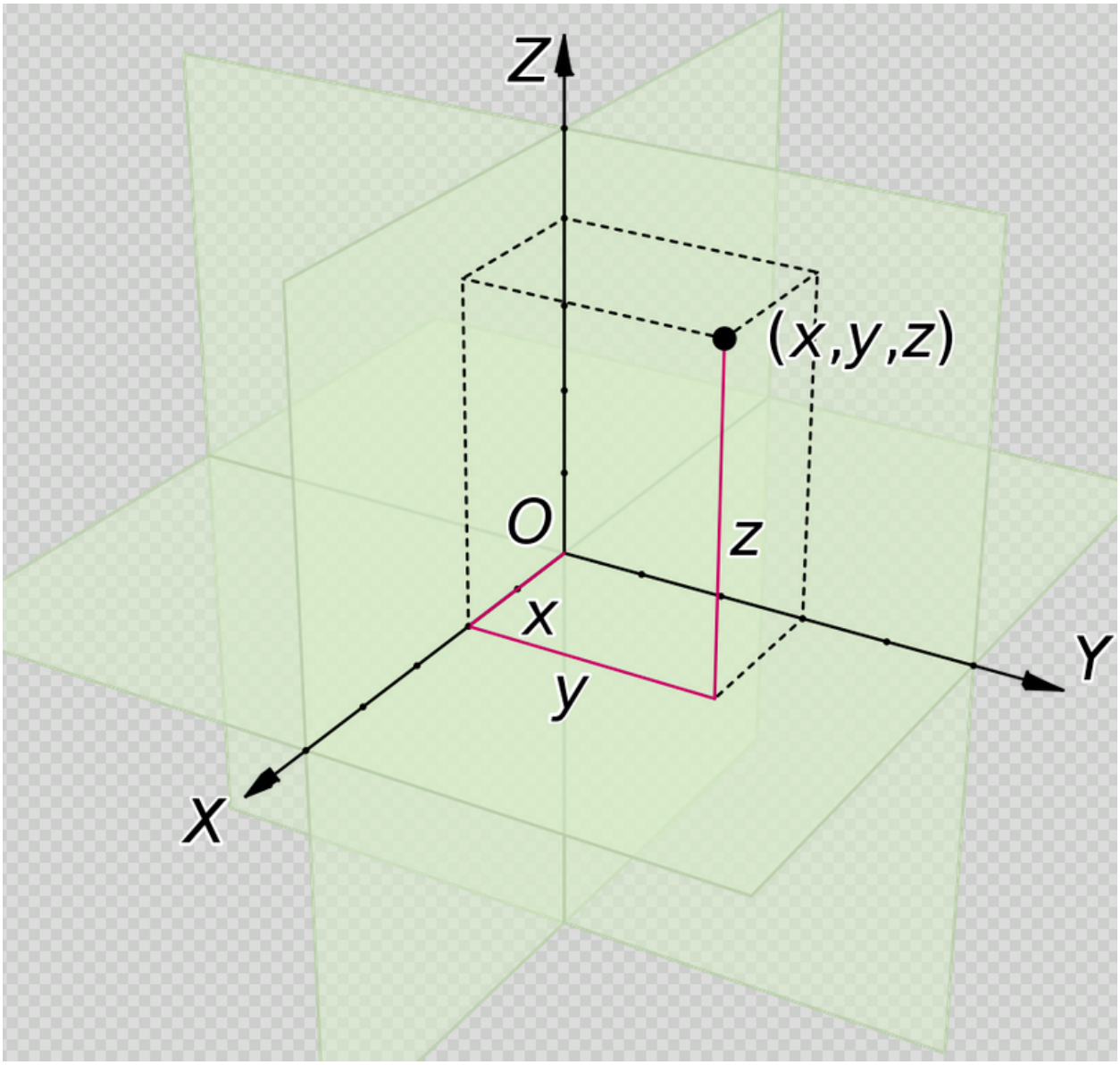
In the rasterization stage, the final image is generated by rendering the transformed geometry onto the screen. This involves converting the vertex data into pixels on the screen, and applying lighting and shading algorithms to generate the final output.

In this stage, the pixel data is generated by interpolating the attributes of each vertex across the surface of the object. This pixel data is then combined to produce the final output image, which is displayed on the screen.

Coordinate Systems:

A coordinate system is a system that uses one or more numbers, or coordinates to determine the position of the points or other geometric elements on a manifold such as Euclidean space.

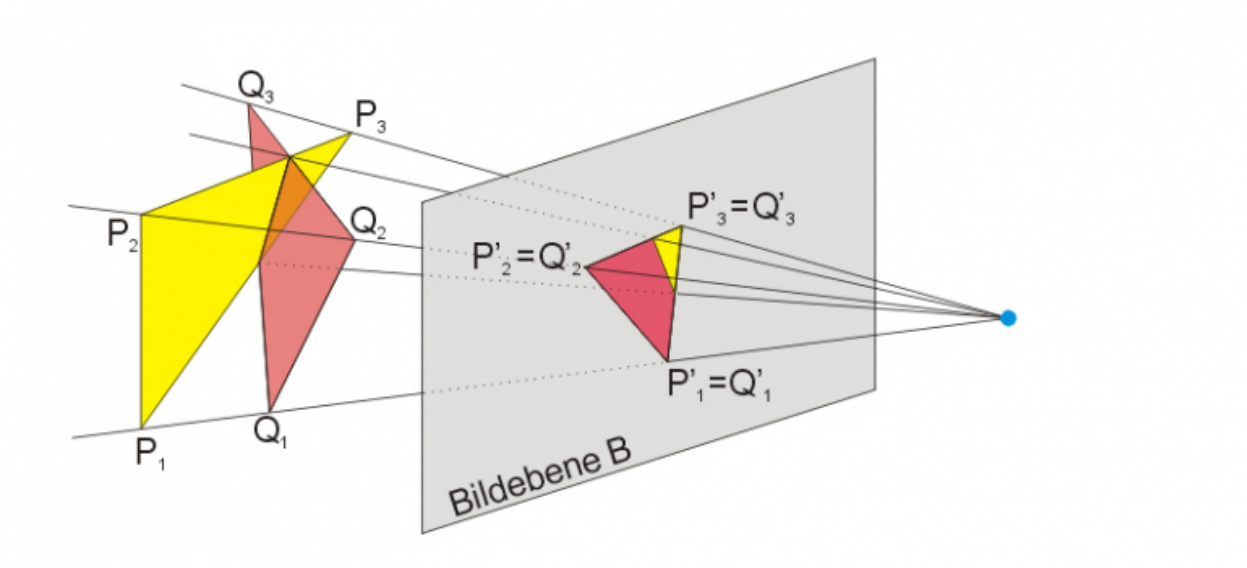
> source Wikipedia



Cartesian coordinates:

In Computer Graphics the most commonly used coordinate system is the Cartesian coordinate system which uses the (X, Y and Z) to define positions in 3D space. Each axis is perpendicular to other, forming a right handed coordinate system.

Homogeneous coordinates:



When working with 3D, we usually think in terms of Euclidean geometry, using coordinates in three-dimensional space (*X*, *Y*, and *Z*). However, there are situations where it is useful to think in terms of projective geometry instead. Projective geometry introduces an extra dimension, *W*, in addition to the *X*, *Y*, and *Z* dimensions. This four-dimensional space is called "projective space," and coordinates in projective space are called "homogeneous coordinates."

- explained more by 2D analogy here:

@https://www.tomdalling.com/blog/modern-opengl/explaining-homogenous-coordinates-and-projective-geometry/

Uses of Homogeneous coordinates in Computer Graphics:

- Translation Matrices for 3D coordinates: In order to do translation transformation, the matrices need to have at least four columns. This why transformations are often 4x4 matrices. A matrix with 4 columns can’t be multiplied with a 3D vector, due to the rules of matrix multiplication. A 4 column matrix can only be multiplied with 4 element vector that’s why we use homogeneous 4D vectors.
- Perspective Transformation: is an important concept that is utilized in 3D computer graphics to create realistic and breathtaking images. This technique involves the use of a transformation matrix to alter the W element of each vertex. By doing so, the perspective of the image can be changed, creating the illusion of depth and distance. Applying the camera matrix to each vertex determines the distance from the camera. The larger Z is, the more the vertex should be scaled down. The W dimension affects the scale, while the projection matrix modifies the W value based on the Z value.
- Positioning Directional Lights: Homogeneous coordinates have the unique property of allowing for points at infinity (infinite length vectors), which is not possible with 3D coordinates. Points at infinity occur when W = 0. However, if you attempt to convert a W = 0 homogeneous coordinate

into a normal W = 0 coordinate, it results in division by zero operations. Directional lights can be thought of as point lights that exist at an infinite distance. When a point light is infinitely far away, the rays of light become parallel, and all of the light travels in a single direction. This is the definition of a directional light.

Introduction to graphics APIs (OpenGL, DirectX)

OpenGL:

Open Graphics Library is a specification of a cross-platform and cross-programming language programming interface for the development of 2D and 3D computer graphics applications.

OpenGL (Hello Window):

- GLFW (Graphics Library Framework: is an open source, multi-platform library for creating windows, contexts, and handling input events in computer graphics applications. It is primarily used as a helper library in graphics programming and is commonly used alongside OpenGL and Vulkan.
- GLAD: is a lightweight, header-only library that generates and manages OpenGL function pointers for various versions of the OpenGL API. It is often used with GLFW. The process of loading OpenGL functions can vary across different operating systems and graphics drivers GLAD simplifies this process.

CODE:

```
int glfwInit( void );
```

A function provided by the GLFW library that initializes the GLFW library itself. when it gets called it performs the necessary initialization tasks to set up the GLFW library and prepares it for use.

if successful it returns GLFW_TRUE which is 1.

if failed it returns GLFW_FALSE which is 0.

```
void glfWindowHint( int hint, int value );
```

A function provided by the GLFW library that allows you to configure various hints or parameters for the creation of a GLFW window. These hints modify the behavior and properties of the window that will be created.

```
GLFWwindow *glfwCreateWindow( int w, int h, const char *title, GLFWmonitor *monitor, GLFWwindow *share ); )
```

A function that creates a window with specified dimensions, title and monitor.

The function returns a pointer to **GLFWwindow** object which represents the created window if the creation fails the functions returns NULL.

```
void glfwMakeContextCurrent( GLFWwindow *window );
```

A function that sets the specified window as the current OpenGL or Vulkan rendering context for the calling thread.

By setting a window as the current context, you establish a binding between the rendering context and the thread that OpenGL or Vulkan calls. This means the subsequent used by OpenGL or Vulkan commands issued by the thread will affect the specified window.

```
int gladLoadGLLoader( GLADloadproc loader );
```

A function provided by GLAD library which is a OpenGL extension loader. It is used to initialize the OpenGL function pointers after creating an OpenGL context.

When we work with OpenGL, we need to load the available OpenGL functions dynamically at runtime, because the implementation of OpenGL functions can vary across different platforms and graphics drivers.

```
void glViewport( GLint x, GLint y, GLsizei width, GLsizei height );
```

The function is used to set the viewport transformation. The viewport defines the portion of the window or framebuffer where the rendering output will be displayed.

- x and y coordinates represent the pixel location where the viewport starts.
- width and height specify the dimensions of the viewport in pixels. The values of width and height define the width and height of the rectangle that will be used for rendering.

```
int glfwWindowShouldClose( GLFWwindow *window );
```

The function is part of GLFW library, the function takes a parameter window, which is a pointer to a GLFW window object. It returns an integer value that indicates whether the window should be closed or not.

```
void glfwSwapBuffers( GLFWwindow *window );
```

The function is part of GLFW library. It is used to swap the front and back buffers of a window, which is necessary for double buffered rendering.

When rendering with double buffering, the graphics card maintain two buffers: the front buffer and the back buffer. The front buffer is the one currently displayed on the screen while the back buffer is where you render your next frame. This results in a smooth and flicker-free animation or image.

```
void glfwPollEvents( void );
```

This function processes all pending events in the event queue and then returns. This includes handling user input events, such as key presses, mouse movements and mouse button clicks, as well as window related events like resizing or closing window.

OpenGL (Hello Triangle):

1. Creating Vertex data: store the coordinates of the triangle in an array. Each point is represented by three floats (x, y, z).
2. Generate two objects: a Vertex Array Object (VAO) and a Vertex Buffer Object (VBO). The VAO stores information about how the vertex data is organized and the VBO stores the actual vertex data in the GPUs memory
3. Binding and setting up the buffers: The VAO and VBO are bound to make them active for further operations. The VBO is then populated with the vertex data using `` `glBufferData` ``. The `GL_STATIC_DRAW` hint indicates that the data will not change frequently.
4. Specifying vertex attribute: The `` `glVertexAttribPointer` `` function is used to specify how OpenGL should interpret the vertex data. After setting up the pointer you need to enable it to signal to OpenGL that this attribute should be used in rendering with `` `glEnableVertexAttribArray` ``
5. Unbinding Buffers: After setting up the VAO and VBO, they are unbound to prevent any accidental changes to them.
6. Shader compilation: Compile and link the vertex and fragment shaders used for rendering the triangle. The vertex shader sets the position of the vertices, and the fragment shader sets the color of the triangle.
7. Rendering the triangle: set the shader program with `` `glUseProgram` `` that will be used to render the triangle. Then the VAO is bound again and `` `glDrawArrays` `` is called to draw the triangle using the VBO.

@https://learnopengl.com/Getting-started/Hello-Triangle

DirectX:

DirectX is a collection of application programming interfaces (APIs) for handling tasks related to multimedia, especially game programming and video, on Microsoft platforms. It includes components

for Direct3D, DirectInput, DirectPlay, DirectSound, and DirectMusic. DirectX is widely used in the development of video games for Windows and Xbox consoles.

DirectX (Hello Window):

- **Win32 Console Programs and Windows Programs**

In **Win32 Console Programs** similarly to the kind of programs you might run in a command prompt or terminal, the program starts with a function called `main()`. However a **Windows program** is more advanced and can interact with the graphical user interface GUI, of the Windows operating system. It has two main functions: `WinMain()` and a special message handling function.

- **The `WinMain()` Function**

Just like `main()` in regular programs, `WinMain()` is the starting point for Windows programs, it's like the door to your program's world.

```
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow);
```

1. **WINAPI**: This is a method of passing information to functions. It's like a special way Window talks to your program. You don't need to worry too much about it.
2. **HINSTANCE hInstance**: This is like a special number that helps Windows tell one program from another. Every time you run your program, Windows gives it a unique number.
3. **HINSTANCE hPrevInstance**: This used to be important before Windows 95, when multiple copies of the same application were forced to share the same memory space.
4. **LPSTR lpCmdLine**: Imagine this as a note that tells your program how it was started. For example if you start a game with `MyGame.exe-mode safe` this parameter helps your program know it's in safe mode.
5. **int nCmdShow**: This is like telling Windows how to show your program's window. big, normal, small etc …

CODE (windows programming):

@http://www.directxtutorial.com/LessonList.aspx?listid=11

```
LRESULT CALLBACK WindowProc(
  HWND hWnd,
  UINT message,
  WPARAM wParam,
  LPARAM lParam
);
```

A function that handles messages sent to the window. For example the quit message.

```
WNDCLASSEX windowClass;
```

A structure describing the window’s properties, like its appearance and behavior.

```
ATOM RegisterClassEx(const WNDCLASSEX *param);
```

Registers the window class so that the operating system knows how to create windows of this type.

```
HWND CreateWindowExW(
  [in]           DWORD     dwExStyle,
  [in, optional] LPCWSTR   lpClassName,
  [in, optional] LPCWSTR   lpWindowName,
  [in]           DWORD     dwStyle,
  [in]           int       X,
  [in]           int       Y,
  [in]           int       nWidth,
  [in]           int       nHeight,
  [in, optional] HWND      hWndParent,
  [in, optional] HMENU     hMenu,
  [in, optional] HINSTANCE hInstance,
  [in, optional] LPVOID    lpParam
);
```

Creates an actual window instance based on the registered class.

```
BOOL AdjustWindowRect(
  [in, out] LPRECT lpRect,
  [in]      DWORD  dwStyle,
  [in]      BOOL   bMenu
);
```

Adjusts the window size to account for borders and menu bars.

```
BOOL ShowWindow(HWND hWnd, int nCmdShow);
```

Displays the window on screen.

```
BOOL PeekMessageA(
  [out]          LPMSG lpMsg,
  [in, optional] HWND  hWnd,
  [in]           UINT  wMsgFilterMin,
  [in]           UINT  wMsgFilterMax,
  [in]           UINT  wRemoveMsg
);
```

Checks if there’s a message in the queue, like a button click.

```
BOOL TranslateMessage(const MSG *msg);
```

Converts low level messages to higher level messages.

```
LRESULT DispatchMessage(const MSG *msg);
```

Sends a message to the `WindowProc()` function for handling.

```
WM_QUIT
```

A message indicating that the application should quit.

```
msg.wParam
```

The exit code returned when the program ends.