OpenGL and DirectX

Copyright © by V. Miszalok, last update: 20-11-2005

OpenGL and DirectX

OpenGL (= Open Graphics Language) is a software interface to the graphic hardware consisting of approx. 250 commands in 2 libraries `oglcore` and `oglutilities`. Developed since 1990 by SGI (Silicon Graphics Inc.) for hardware- and platform-independent graphics (aimed at graphic hardware manufacturers and programmers).
Links:
www.opengl.org
www.sgi.com/products/software/opengl
See the OpenGL/C++/MFC samples:
www.miszalok.de/C_3DC7/C1_OpenGL/C3DC7OpenGL_d.htm and
www.miszalok.de/C_3DC7/C2_Texture/C3DC7Texture_d.htm

DirectX is a collection of 10 libraries (see table below) for extreme hardware oriented programming in hardawre independent form. Permanent development by Microsoft since 1994 (formerly "Games SDK") in ascending version numbers (today: 9.0c). Goal: Windows as platform for multimedia. Nearly all graphic-, sound-, radio-, video-, TV-boards are bundeled with DirectX driver software.
The DirectX libraries are fundamentally differnt from all other Windows-APIs (Application Programming Interfaces). They do not guarantee any execution. The programmer has to find out whether the DirectX-programmed hardware exists on the target machine and if yes, what could be done with it.
In practice the quick-and-dirty programmer pins his hope on the DirectX hardware driver. The hope is that the driver is good enough not to simply deny an indigestible call via DirectX, but to dispose of a detour via normal Windows libs (see HEL below). Prof. Miszalok's **Course 3DCis** bases on this hope too.
Links:
www.microsoft.com/windows/directx/default.aspx
www.intel.com/cd/ids/developer/asmo-na/eng/57590.htm?page=1
www.it-academy.cc/content/article_browse.php?ID=732
Notice the DirectX/C++/MFC sample:
www.miszalok.de/C_3DC7/C3_DirectDrawMesh/C3DC7DirectDrawMesh_d.htm
and a lot of DirectX/C#/.NET tutorials and samples:
www.miszalok.de/C_3DCis/Index_of_Course.htm and
www.miszalok.de/C_3DCisCA/Index_of_Course.htm and
www.miszalok.de/C_3DCisMS/Index_of_Course.htm

OpenGL and Direct3D are on a par and rather similar. Please notice the following differences:

	OpenGL	DirectX
object oriented	no	yes
supports audio/video/game input devices	no	yes
operating systems	many	only Windows and its variants
drivers available for	high-end graphic boards	nearly all graphic boards
quality of drivers	often bad	often better than OpenGL-drivers
mainly used by	universities, research, CAD	game industry
docu, tutorials, samples, books	many	not as many as for OpenGL
new version	every 5 years (except "Extensions")	every 15 months
property of	Silicon Graphics Inc.	Microsoft

OpenGL Libraries and DirectX Namespaces

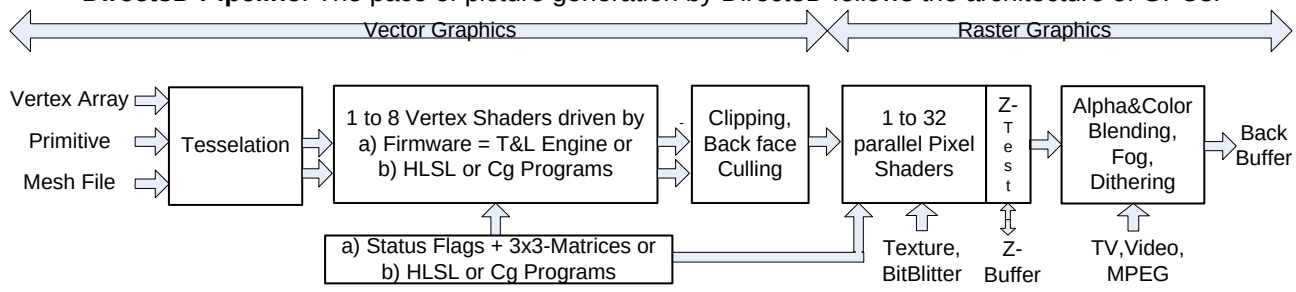
OpenGL consists of two graphic libraries whereas DirectX confederates 10 libraries which all bypass the operating system and access the hardware directly and dangerously. These libraries are wrapped by managed DirectX namespaces. The first four of them deal with graphics.

OpenGL lib	covers **DirectX** functionality from
oglcore	Microsoft.DirectX, Microsoft.DirectX.DirectDraw, Microsoft.DirectX.Direct3D
oglutilities	Microsoft.DirectX.Direct3DX
.NET Namespace	**API** = Application Programming Interface
Microsoft.DirectX	common basic functions
Microsoft.DirectX.DirectDraw	subset of Direct3D-lib: basic 2D functions, bitmaps, window management
Microsoft.DirectX.Direct3D	API for 3D graphics: wireframes, textures, light, Vertex and Pixel Shaders
Microsoft.DirectX.Direct3DX	3D utilities library, Mesh class and scene graph
Microsoft.DirectX.DirectPlay	network support for multiplayer games, host administration for DirectPlay sessions
Microsoft.DirectX.DirectSound	contains DirectMusic, API for real time multichannel mixer, 3D sound
Microsoft.DirectX.DirectInput	API for keyboard, mouse, joystick, trackball, touchpad, gamepad, wheel, force feedback
Microsoft.DirectX.AudioVideoPlayback	API for simple sound and video
Microsoft.DirectX.Diagnostics	system diagnostics API
Microsoft.DirectX.Security	system security API

OpenGL & Direct3D Pipeline

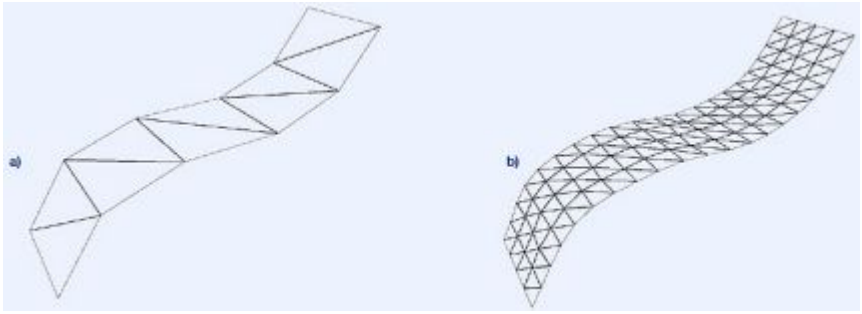
Modern graphic chips contain several cascading autonomous graphic processors which are arranged in form of a pipeline (= bucket chain). The first half of the processor chain is occupied with vector graphics, the second with raster graphics.
The command chain of OpenGL and Direct3D mirrors the processor chain of the graphic chips = GPUs. Thus about half of the OpenGL and Direct3D functions are vector functions und half are raster functions and some are mixed. The fundamental differences between vector- and raster graphics are veiled and hidden in order to spare the programmer the problems of vector to raster transformation. The programmer is also shielded from all problems of division of labor between CPU and graphic board, as from all technical differences between the various types of graphic chips. With all this comfort, beginners need just poor hardware and raster graphics know how.

Direct3D Pipeline: The pace of picture generation by Direct3D follows the architecture of GPUs.



In DirectX you can switch off the complete vector graphics part of the GPU with the flag `CreateFlags.SoftwareVertexProcessing` in the constructor of `Device`. In this case and/or if the graphic board or the mother board have no or not a complete GPU, OpenGL/DirectX simulates the pipeline inside the CPU. Then the terms vertex shader, T&L engine, HSSL/Cg programs give no sense any more and should be replaced by CPU-based graphics, see below chapter HEL and HAL.

Tesselation: Produces triangle grids from polygons and refines a coarse triangle grid.
Modern: Refinement dependent on the viewer distance = adaptive refinement = depth-adaptive tesselation = Level Of Detail based Tesselation = LOD based Tesselation = coarse triangles when the object is far away and small and fine triangles in short distance to the eye point. image source: www.hartware.net



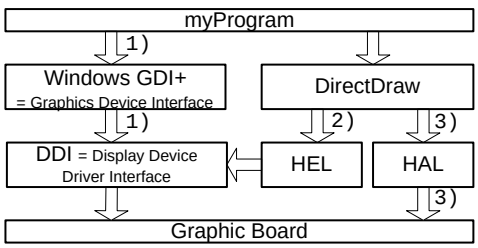
Vertex Shader = Pipeline of micro processors inside the GPU. Modern GPUs contain up to 8 such pipelines in parallel. The meaning is ambiguous: A program written in **HLSL** or **Cg** to be fed into a Vertex Shader is called Vertex Shader also.
T&L Engine = Transform & Light Engine = Fixed Vector Pipeline = means 1 to 8 parallel Vertex Shaders, driven by prefabricated firmware offering poor freedom. You have to use property flags and 3x3 matrices to enter commands into this firmware:
a) property flags (f.i. `device.Lights[0].Enabled = true;`) and
b) 3x3 matrices (f.i. `device.Transform.View = Matrix.LookAtLH( new Vector3( 0f, 0f,-4f ), new Vector3( 0f, 0f, 0f ), new Vector3( 0f, 1f, 0f ) );`).
Clipping = Cutting lines and convex polygons which overlap the image border using the Cohen-Sutherland-Algorithm
Back Face Culling: approx. 50% of the triangles show their back sides. Removing them accelerates all following raster operations to double speed.
Pixel-Shader = rasterizer = special processors downstream of the vertex shaders, specialized on raster graphics = textures and rendering of singular pixels, programmable with **HLSL** or **Cg**, graphic chip contains up to 32 parallel pixel shaders.
Information and tools about shader programming:
http://developer.nvidia.com/object/fx_composer_home.html
Texture = deformation = distorts a rectangular image in such a way that it fits onto a mesh
BitBlitter = abbrev. Bit Block Transfer = externally rendered characters, lines, rectangles, ellipses etc.
Z-Test = Depth Test = compare current z-coordinate with corresponding Z-buffer content and throw the pixel away if hidden
Alpha & Color Blending = masked superimposing of transparent pixels or pixels with special colors
Fog = fog depending on distance
Dithering = smoothing of stepped color transitions of 4-, 8- und 16-bit images by sliding mixture

HEL and HAL

With the installation of a driver of a graphics board, sound board, joystick etc. the driver embodies itself in the operating system in form of a specific Device Driver Interface DDI. With the help of the appropriate DDIs each DirectX library initializes a Hardware Emulation Layer HEL and a call-identical Hardware Abstraction Layer HAL. HEL contains the low-level calls of basic functions and CPU-code, HAL the calls of external, autonomous micro programs of the graphic board, sound board etc. HAL has priority in front of HEL, but all library calls are executed via HEL in case HAL does not work. HEL animations, HEL audios, HEL videos etc. used to be awfully slow.
But CPU manufacturers as Intel or AMD fighting against the graphic- and multimedia boards improve the graphic and sound power of their CPUs and the architecture of busses and empower HEL vs. HAL. They have limited success but many today users just playing simple games and simple multimedia do not necessarily need dedicated hardware. Modern on-board graphic chips = computers without graphic boards, video memory and HAL execute DirectX programs at sufficient speed for normal office applications.

Example: Draw with GDI+ or with DirectDraw HEL/HAL
There are three ways to draw something:
`1)` normal Windows instruction without DirectX uses GDI+ and DDI.
Sample: `graphics.DrawLine(mypen,0,0,100,100);`
`2)` using DirectDraw, HEL and DDI
`3)` using DirectDraw und HAL

`3)` is faster than `2)` and `2)` is faster than `1)`. If call `3)` exists, `2)` is closed.
Its possible to mix GDI+ and DirectDraw statements in any order.
GDI+ Info: **GDIPlus.asp**



Caution: Not fully developed and buggy graphic drivers often install an incomplete DDI and/or HAL which depreciates a good graphic hardware. The class **Device** just has the driver as only knowledge source about the hardware and just uses the features described by DDI and/or HAL. Recommended: Check in the Internet with the manufacturer of your graphic board whether there is a newer driver and install it.

Direct3D Device

is the most important Direct3D class, it mirrors and manages the graphic board hardware and the current software canvas. The most visible method `Device.Present` shows the scene on the monitor by flipping the BackBuffer of the graphic board to FrontBuffer.
The class contains properties/methods for vector graphics (i.e. Viewport, VertexFormat, Transform) and for raster graphics (i.e. Material, Texture, adresses and lengths of the output buffers).
At first any Direct3D program has to instanciate this class in order to obtain ressources and access rights to video memory. Unfortunately both are not of duration, they can be lost at any moment and have to be initialized from scratch. The first symptom of the loss of `Device` is the `DeviceLostException` thrown by `Device.Present` which does not work anymore. `Device` is lost when:
1) The user changes the window size of the program with the mouse: `OnResize`-Event.
2) A screen saver takes over the graphic board exclusivly.
3) The Windows operating system takes over the graphic board exclusivly.
4) CPU or graphic board change to standby power saving mode.
5) The cover of the notebook is opened or closed.
The samples of **Course3DCis** solve problem 1) by initializing `Device` inside the `OnResize()`-EventHandler.
Caution: For the sake of simplicity problems 2) till 5) remain unsolved. Professional solutions can be found under **http://pluralsight.com/wiki/default.aspx/Craig.DirectX/DeviceRecoveryTutorial.html** and **www.jkarlsson.com/Articles/devicelost.asp**

Important **Properties**	of Direct3D class "**Device**"
DeviceCaps	Gets a struct representing the capabilities of the hardware; this is the property to query when you want to know whether the hardware supports a particular feature that your application may require.
Viewport	Gets/sets the rectangular rendering region on the device canvas.
Material	Gets/sets the material to use in rendering.
Lights	Gets the collection of lights that can be activated for rendering.
RenderState	Gets the collection of render states that are used to control the different stages of the Direct3D pipeline.
VertexDeclaration	Gets/sets a description of the vertex format being used with a vertex shader.
VertexFormat	Gets/sets a description of the vertex format being used with the fixed vector pipeline.
VertexShader, PixelShader	Gets/sets the vertex/pixel shader to use for rendering

Important **Methods**	of Direct3D class "**Device**"
BeginScene	Prepares the device to render a frame of primitives.
EndScene	Tells the device that all the primitives have been rendered for a frame.
DrawPrimitives	Renders a primitive.
Clear	Clears the viewport in preparation for another frame to render.
Present	Prepares and renders the next buffer; Present is called after EndScene and before the next BeginScene (for the next frame)
GetTransform, SetTransform	Gets/sets the world, view, projection or other transform; transforms are applied to vertex positions and normals, and/or to texture coordinates
GetTexture, SetTexture	Gets/sets the texture associated with a given texture stage

Sample before initialising "Device": Query of available pixel formats and refreh rates

```
1: StringBuilder s = new StringBuilder();
2: AdapterInformation ai = Manager.Adapters(0);
3: foreach DisplayMode dm in ai.SupportedDisplayModes
4:   s.Append( dm.Format + " " + dm.RefreshRate + "\r\n" );
```

line 2: Take graphic board no. 0. (There can be more than one on the bus !)
line 3: Enumerate all available `SupportedDisplayModes`.
line 4: Append a line to text `s`: `Format`-string + blank + `RefreshRate`-string + carriage return + new line.

Sample initialising "Device":
Such initialising is necessary upon any window resize (`OnResize` event) because `Device` is lost.

```
1: presentParams = new PresentParameters();
2: presentParams.Windowed = true;
3: presentParams.SwapEffect = SwapEffect.Discard;
4: Device device = new Device( 0, DeviceType.Hardware, this,
            CreateFlags.SoftwareVertexProcessing, presentParams );
```

line 1: Memory space for structure `PresentParams`
line 2: Do not use full screen.
line 3: Switch off `SwapEffect`.
line 4: new `Device`: order the necessary memory space and fill it with prededefined and some self defined properties.

Sample using "Device":

```
1: device.Clear( ClearFlags.Target, Color.Blue, 1.0f, 0 );
2: device.BeginScene();
3:   (Mesh.Teapot( device )).DrawSubset( 0 );
4: device.EndScene();
5: device.Present();
```

line 1: Clear the BackBuffer canvas.
line 2: Open bracket
line 3: Draw the teapot on the BackBuffer canvas.
line 4: Close bracket
line 5: Flip BackBuffer and FrontBuffer.