DirectX Programming #1

Kang, Seongtae

Computer Graphics, 2009 Spring

VP LAB
Visual Processing Lab

Contents

- Installation and Settings
- Introduction to Direct3D 9 Graphics
- Initializing Direct3D
- Rendering Vertices

Installation

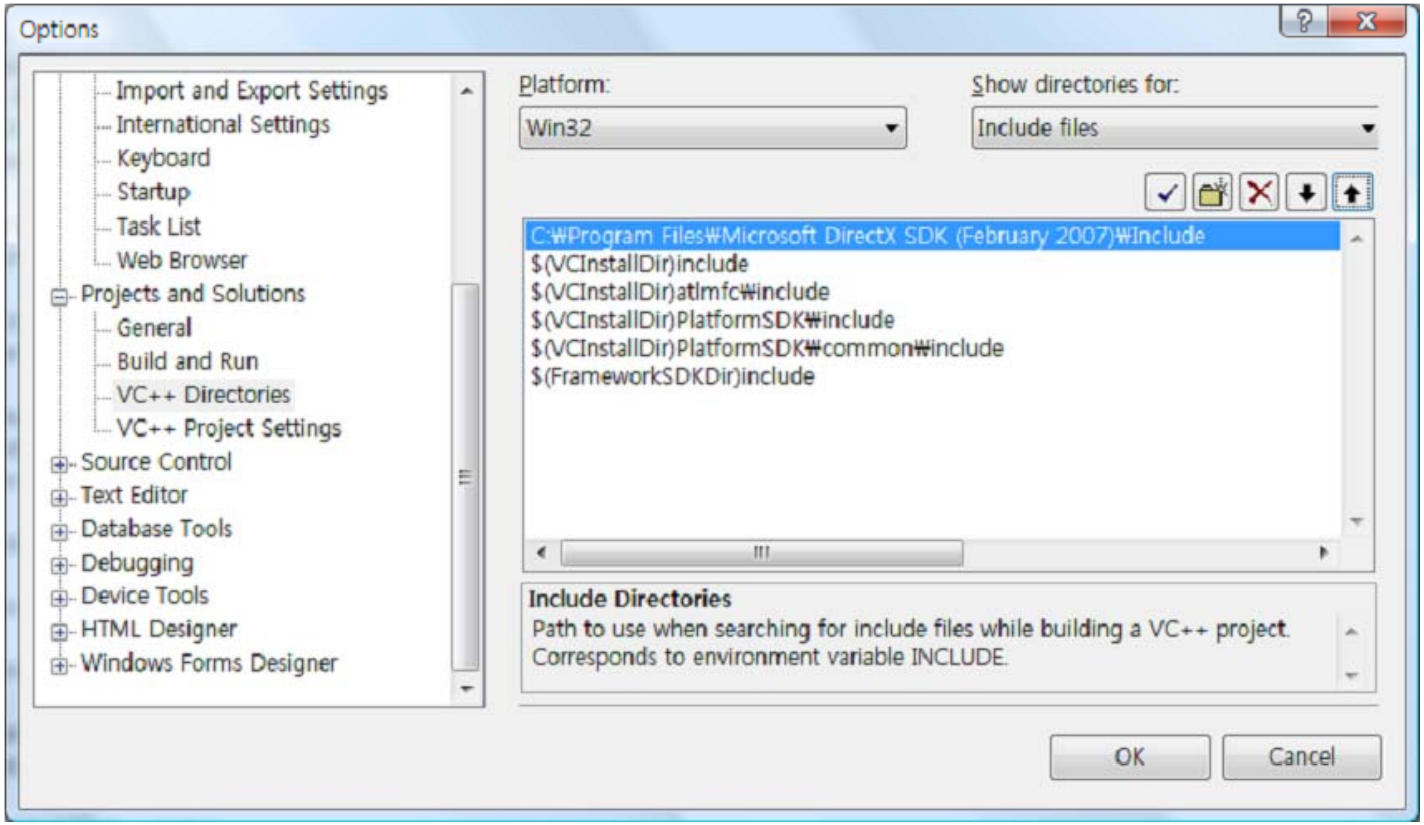
- DirectX SDK
  - You can download from Microsoft's homepage for free
    - Latest Version : March 2009
    - http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=24a541d6-0486-4453-8641-1eee9e21b282
  - You need not to get the latest version of SDK
    - April 2007 or later are OK

DirectX Layers

- Native DirectX SDK
  - COM-based API (C++ based)
  - Supports Visual C++ and Visual Basic
- Managed DirectX
  - Microsoft .Net wrapper for DirectX API
  - Supports .NET langueages (C#, VB.Net, C++/CLI, ...)
  - Obsoleted
    - Managed DirectX 9 for .NET framework 1.1 is the last version
    - No x64 support
- This lecture covers native DirectX APIs on the Win32 platform only
  - If you prefer, you can do your assignments with Managed DirectX and C#

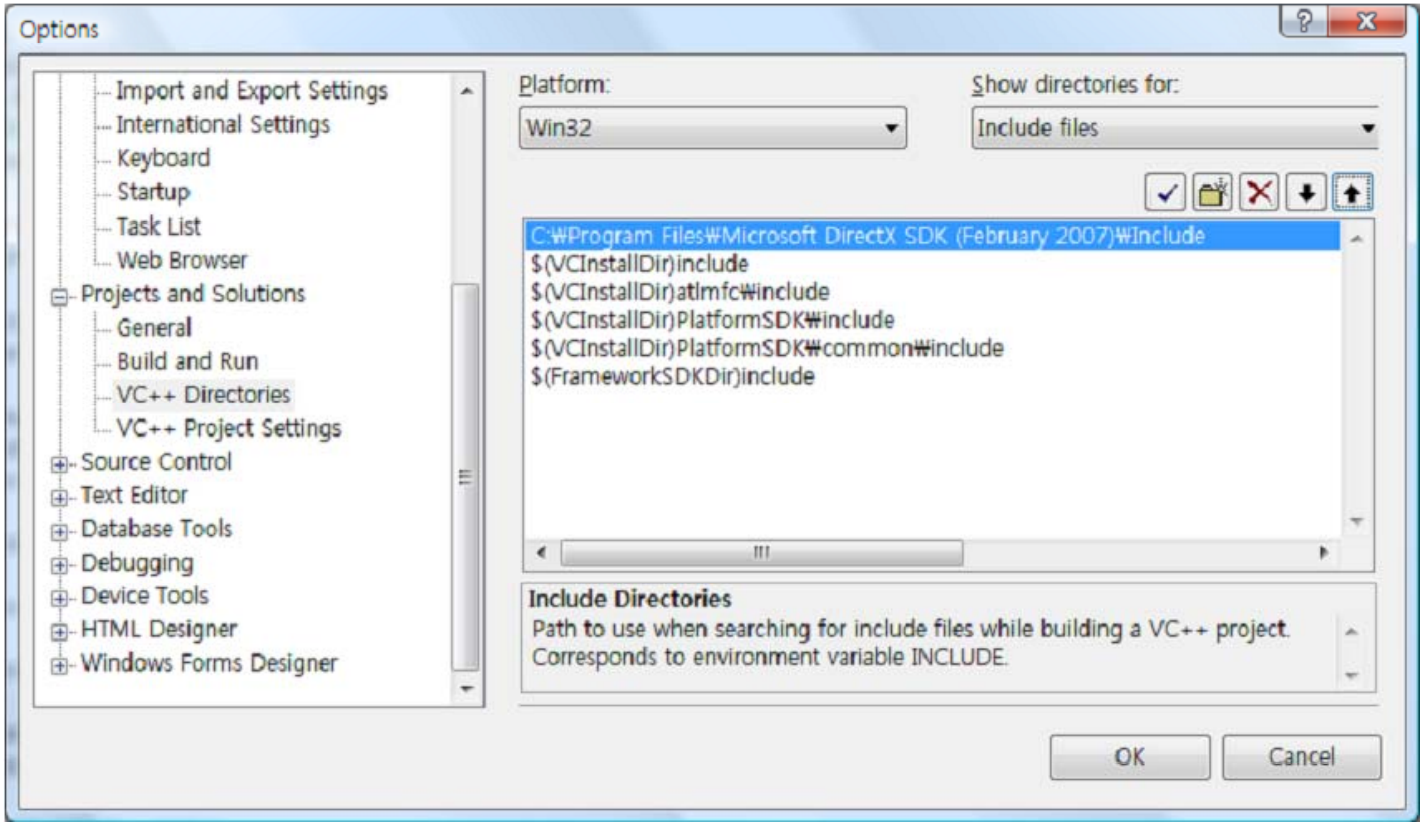
Visual Studio Settings

- Add DirectX include and library directories
  - <Menu> Tools → Options
    → Projects and Solutions → VC++ Directories



Visual Studio Settings

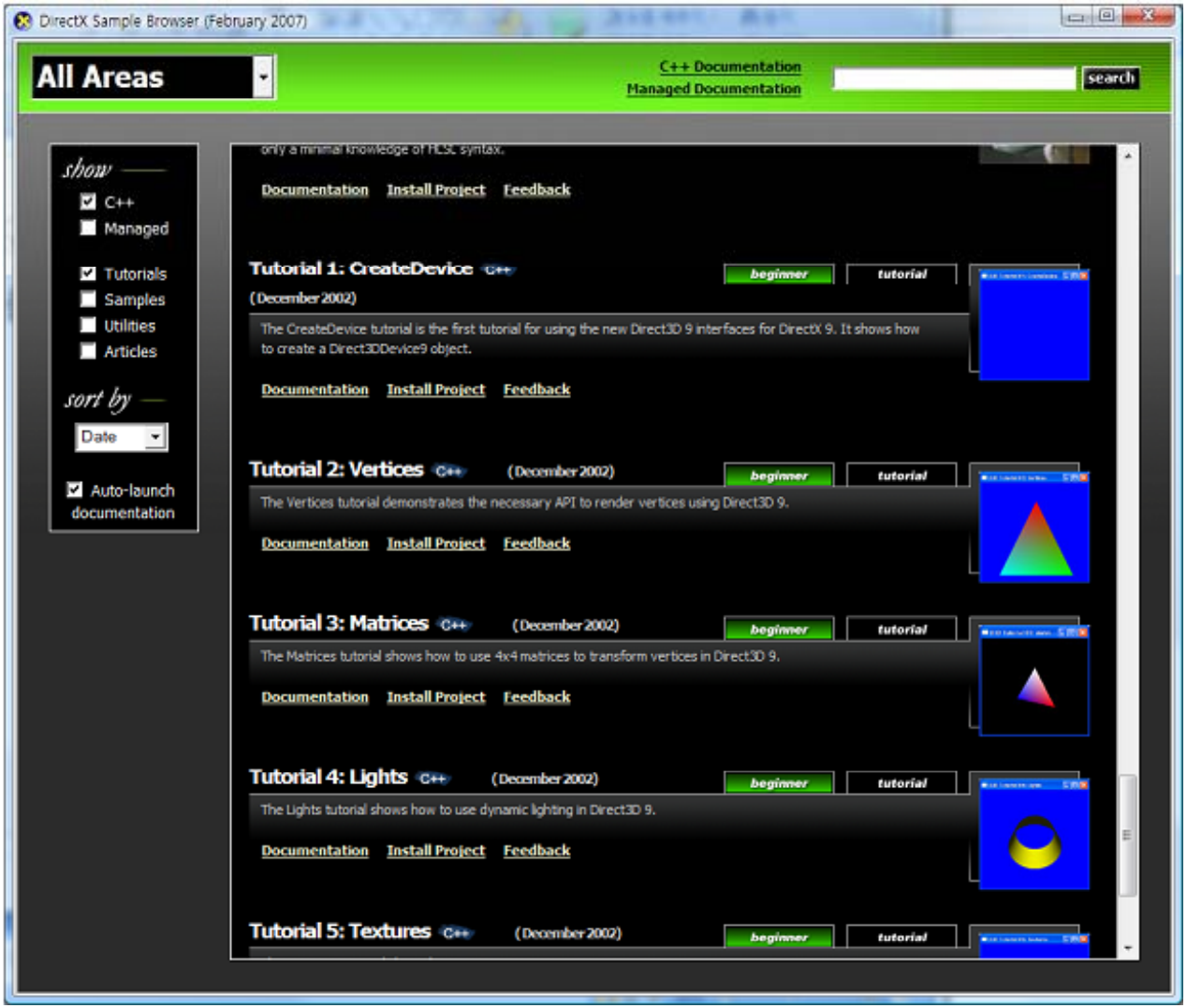
- Move the DirectX directories to the top
  - Visual Studio contains some old-version of DirectX headers and libraries, which may cause confliction with newer version



Sample and tutorial codes

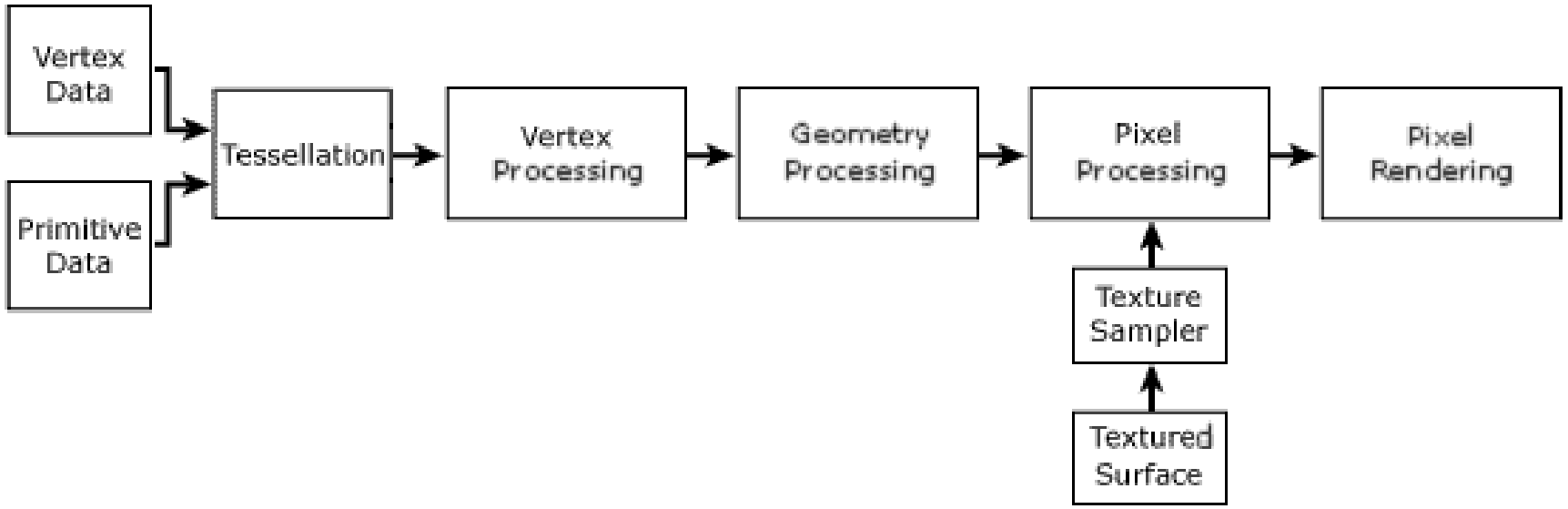
- Sample codes
  - [DX directory]/Samples/C++/Direct3D
- Tutorial Codes
  - [DX directory]/Samples/C++/Direct3D/Tutorials

Using Sample Browser



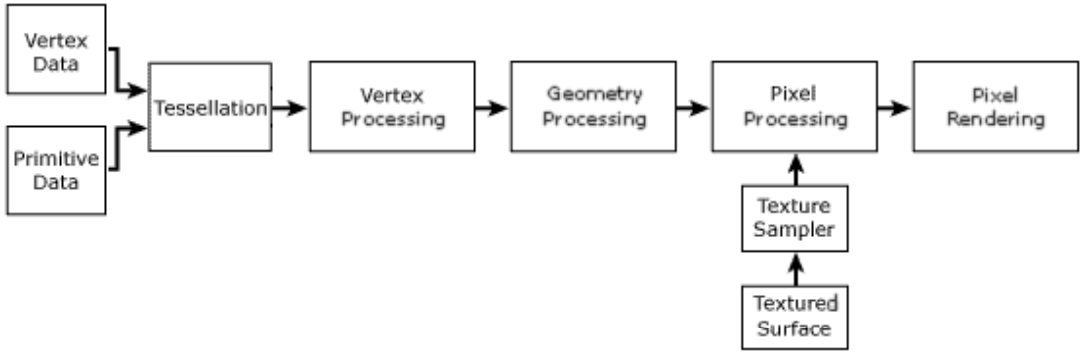
- Some versions of DirectX SDK do not contain the sample browser.

Direct3D 9 Architecture



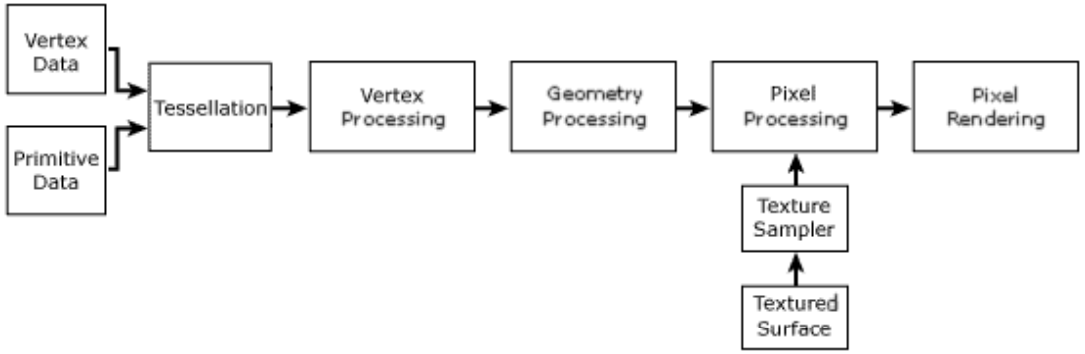
Direct3D 9 Pipeline Components

- Vertex Data
  - Untransformed model vertices are stored in vertex buffer
- Primitive Data
  - Geometric primitives, including points, lines, triangles, and polygons, are referenced in the vertex data with index buffer
  - When index buffer is not defined, the pipeline build primitives using vertices in vertex buffer sequentially



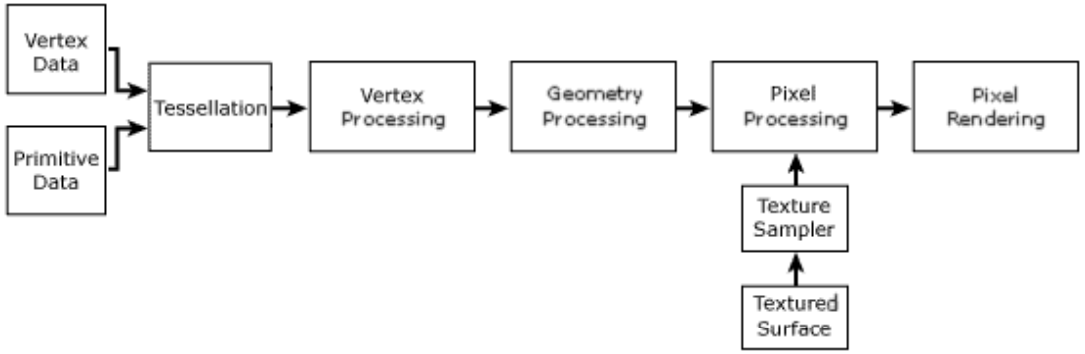
Direct3D Graphics Pipeline Stages

- Vertex processing
  - Vertex transformation : world, view, projection transformation
- Geometry processing
  - Clipping, back face culling, attribute evaluation
  - Rasterization
- Texture Sampler
  - Level-of-detail filtering for texture sampling with input texture values from D3D texture surfaces

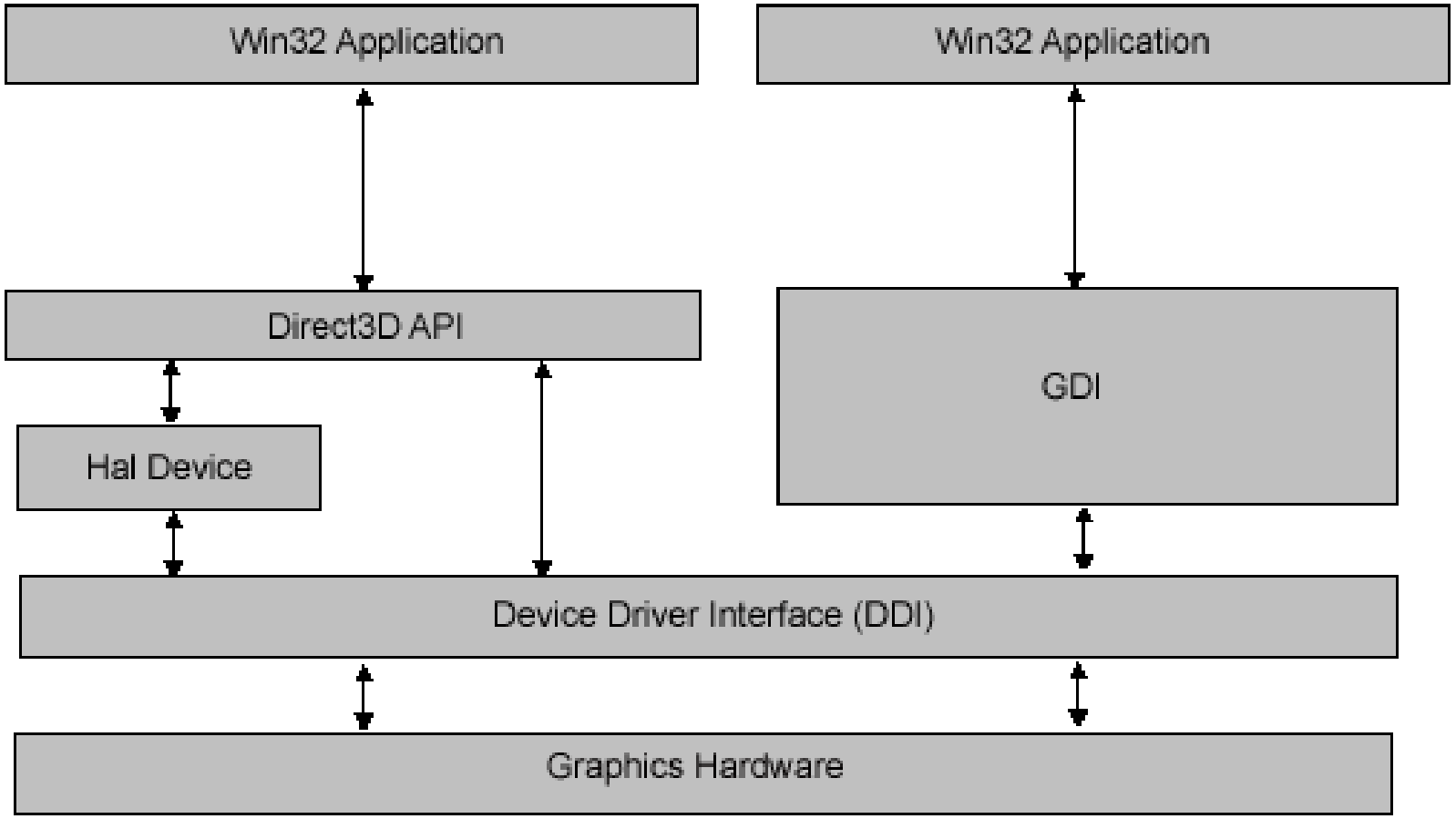


Direct3D Graphics Pipeline Stages

- Pixel processing
  - Modify input vertex and texture data using geometry data
  - Output pixel color values
- Pixel Rendering
  - Modify pixel color values finally
  - Alpha blending
  - Depth, stencil, alpha test
  - Fog blending



Direct3D System Integration



Tutorial 1 : Creating Devices

- Brief Structure of Win32 D3D Program

- Create a window
- Create Direct3D bound to the window
- Create a device bound to Direct3D
- Define fixed data and parameters
  - Vertices and indices, textures, primitive types …
- Setting per-render parameters
  - Viewing parameters, lights, …
- Begin rendering
  - Draw primitives
- End rendering
- Present

Rendering Loop

Creating Devices

- Initializing Direct3D
  - After the window is created – You learned this before
  - Creating Direct3D
  - Creating device
    - Setting presentation parameters

```
LPDIRECT3D9     g_pD3D = NULL;
LPDIRECT3DDEVICE9  g_g_pd3dDevice;

if( NULL == ( g_pD3D = Direct3DCreate9( D3D_SDK_VERSION ) ) )
        return E_FAIL;

D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;

if( FAILED( pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                        &d3dpp, &g_g_pd3dDevice ) ) )
                                        :
```

Basic Rendering Routine

- Clear
  - Clear back buffer, depth buffer, and stencil buffer
- BeginScene / EndScene pair
  - Compose a rendering block
- Present
  - Presents the back buffer

```
    g_g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255), 1.0f,
0 );

    if( SUCCEEDED( g_g_pd3dDevice->BeginScene() ) )
    {
        // Render here
        g_g_pd3dDevice->EndScene();
    }

    g_g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
```

Locating the Rendering Routine

- WM_PAINT handler
  - When the rendered scene is static
  - Minimum load for rendering
  - Refersh by WM_PAINT message
    - InvalidateRect

```
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
   switch( msg )
   {
                                                    :
                                                    :
    case WM_PAINT:
          Render();
          ValidateRect( hWnd, NULL );
          return 0;
                                                    :
                                                    :
```

Locating the Rendering Routine

- Message Loop
  - When the rendered scene changes continuously
    - E.g. time-series animation
  - Infinite loop
    - If the rendering routine is heavy, the application drains system resource

```
MSG msg = {0};
do      // message loop
{
        if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))        // if there's a delivered message
        {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
        else                                                // render when there's no delivered message
        {
                Render();
        }
} while(WM_QUIT != msg.message);      // until 'quit the application' message is delivered
```

Finalization

- When the D3D application ends
  - Usually located in WM_DESTROY message handler
- Clear objects
  - Release all created objects

```
case WM_DESTROY:
  if( g_g_pd3dDevice != NULL)
     g_g_pd3dDevice->Release();
  if( g_pD3D != NULL)
     g_pD3D->Release();
```

Result : Tutorial 1

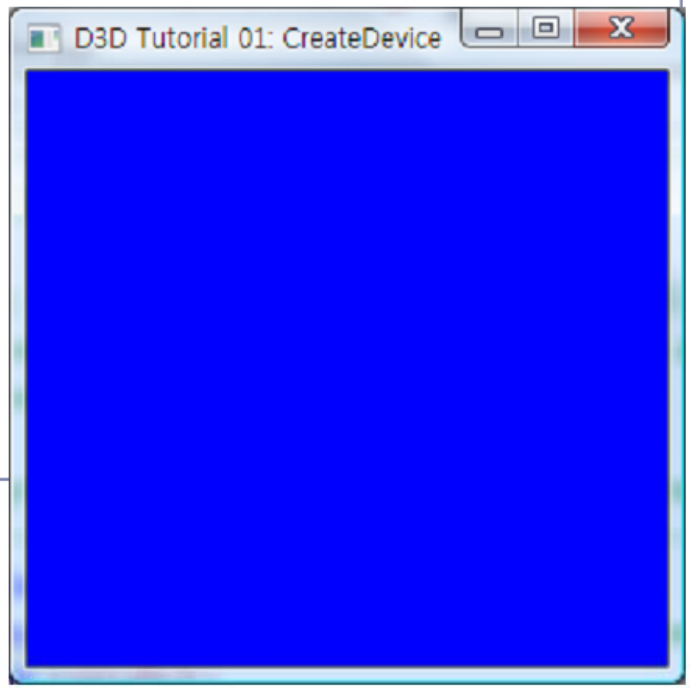
```
VOID Render()
{
    if( NULL == g_pd3dDevice )
        return;

    // Clear the backbuffer to a blue color
    g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255), 1.0f, 0 );

    // Begin the scene
    if( SUCCEEDED( g_pd3dDevice->BeginScene() ) )
    {
        // Rendering of scene objects can happen here

        // End the scene
        g_pd3dDevice->EndScene();
    }

    // Present the backbuffer contents to the display
    g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
}
```



Tutorial 2 : Rendering Vertices

- Definition of the vertex type
  - Vertex declaration object
    - Flexible declaration
    - Complicated code
  - FVF
    - Combination of pre-defined type templates
    - Fixed order
    - Simple

Vertex Type Definition : FVF

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)
```

- FVF
  - Description of structure for a vertex
  - OR combination of predefined FVF templates
  - Element order
    - position – normal – diffuse, specular – texture coordinates
  - e.g.
    - D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_DIFFUSE
      - 3D position, normal and diffuse
    - D3DFVF_XYZ | D3DFVF_TEX2
      - 3D position and two 2D texture coordinates
    - D3DFVF_XYZ | D3DFVF_TEX1 | D3DFVF_TEXCOORDSIZE3(0)
      - 3D position with one 3D texture coordinates

Vertex Type Definition : C++ Struct

- Vertex structure
  - Just for comfortable coding
    - FVF components have fixed order and size
  - You can even use just byte array
- Size of elements
  - Position and texture coordinate : (# of dim.) x FLOAT
  - Diffuse/specular color : DWORD (D3DCOLOR_ARGB)
- Element ordering
  - Defining order of elements in the structure must be matched to FVF element ordering

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)

struct CUSTOMVERTEX
{
    FLOAT x, y, z, rhw; // The transformed position for the vertex.
    DWORD color;        // The vertex color.
};
```

Creating the Vertex Buffer

- Vertex Buffer
  - Holds vertices to render
  - Reside in the video memory

- Creating a vertex buffer

```
HRESULT CreateVertexBuffer(UINT Length, DWORD Usage, DWORD FVF, D3DPOOL Pool,
                           IDirect3DVertexBuffer9** ppVertexBuffer,  HANDLE* pSharedHandle);
```

  - Length : Size of the vertex buffer, in bytes
  - Usage : Usage of the resource; Usually 0 for vertex buffer
  - FVF : FVF to use for this vertex buffer
  - Pool : description of the memory class that holds the buffer. See D3DPOOL
    Usually D3DPOOL_DEFAULT or D3DPOOL_MANAGED for vertex buffer
  - ppVertexBuffer : pointer of the vertex buffer object
  - pSharedHandle : Not used

Creating the Vertex Buffer

- Locking and unlocking
  - Lock
    - Lock the buffer and obtains a pointer to the memory
    - CPU can access the locked resource buffer
    - GPU memory → CPU memory (download)
    - When locking for writing, setting D3DLOCK_DISCARD flag helps performance (no downloading)
  - Unlock
    - CPU memory → GPU memory (upload)
    - For reading only, locking with D3DLOCK_READONLY flag helps unlocking performance (no uploading)

```
HRESULT Lock(UINT OffsetToLock, UINT SizeToLock, VOID ** ppbData, DWORD Flags);
```

- OffsetToLock : Offset into the vertex data to lock, in bytes; 0 for locking the entire buffer
- SizeToLock : Size of the vertex data to lock, in bytes; 0 for locking the entire buffer
- ppbData : VOID* pointer to a memory buffer
- Flags : Locking flags; See D3DLOCK on the SDK document

Creating the Vertex Buffer : Tutorial 2

```
LPDIRECT3DVERTEXBUFFER9 g_pVB;

CUSTOMVERTEX vertices[] =
{
   { 150.0f,  50.0f, 0.5f, 1.0f, 0xffff0000, }, // x, y, z, rhw, color
   { 250.0f, 250.0f, 0.5f, 1.0f, 0xff00ff00, },
   {  50.0f, 250.0f, 0.5f, 1.0f, 0xff00ffff, },
};

if( FAILED( g_pd3dDevice->CreateVertexBuffer( 3*sizeof(CUSTOMVERTEX),
        0 /*Usage*/, D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT, &g_pVB, NULL ) ) )
   return E_FAIL;

VOID* pVertices;
if( FAILED( g_pVB->Lock( 0, sizeof(vertices), (void**)&pVertices, 0 ) ) )
   return E_FAIL;

memcpy( pVertices, vertices, sizeof(vertices) );

g_pVB->Unlock();
```

Binding the Vertex Buffer

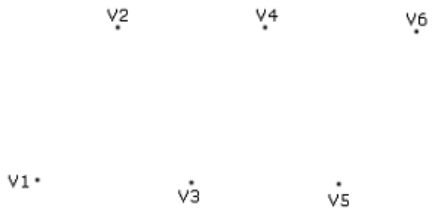
- Vertex buffers need to be attached to the pipeline

```
HRESULT SetStreamSource(UINT StreamNumber, IDirect3DVertexBuffer9 * pStreamData,
                                                    UINT OffsetInBytes, UINT Stride);
```

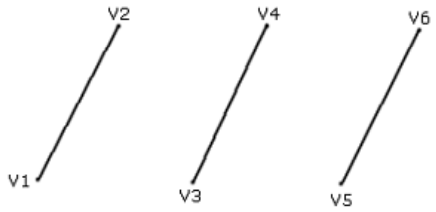
- StreamNumber : Specifies the data stream
- pStreamData : Pointer to a vertex buffer
- OffsetInBytes : Offset from the beginning of the stream to the beginning of the vertex data, in bytes
  Usually 0; (Actually, many hardwares don't support VB offset features)
- Stride : Stride, i.e. size of a component, in bytes

Primitives

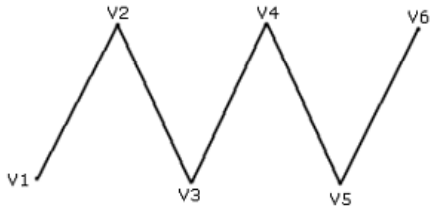
- Primitive types determine how the pipeline interpret incoming vertex stream to build primitives



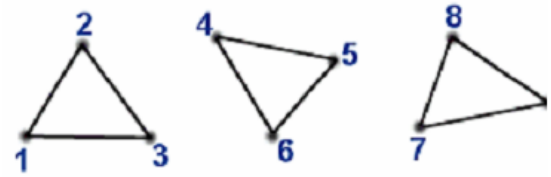
D3DPT_POINTLIST



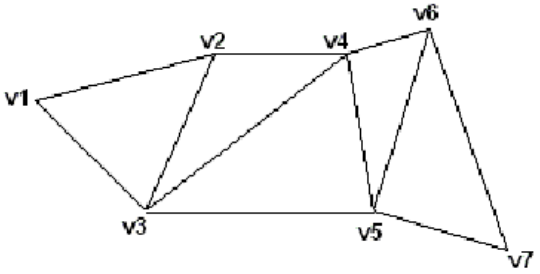
D3DPT_LINELIST



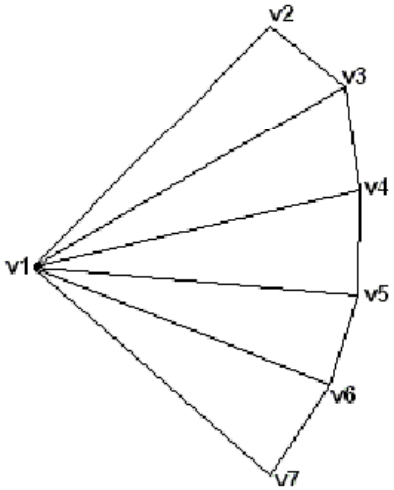
D3DPT_LINESTRIP



D3DPT_TRIANGLELIST



D3DPT_TRIANGLESTRIP



D3DPT_TRIANGLEFAN

Drawing Primitives

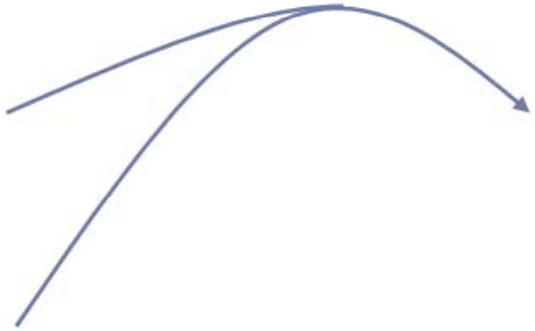
HRESULT DrawPrimitive(D3DPRIMITIVETYPE PrimitiveType, UINT StartVertex, UINT PrimitiveCount);

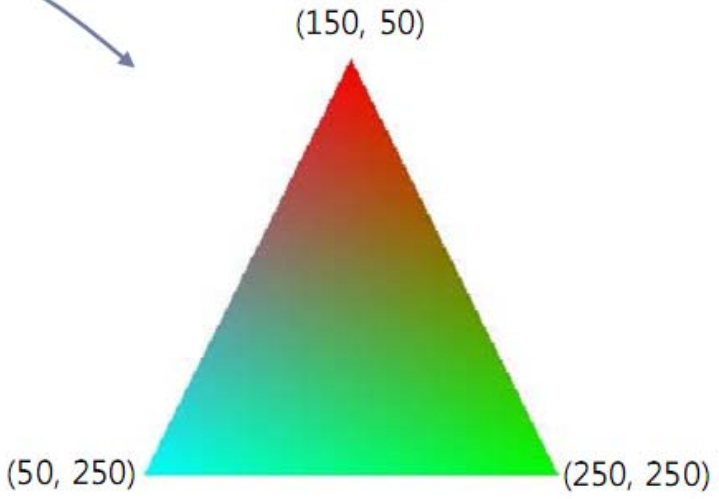
- PrimitiveType : primitive type
- StartVertex : Index of the first vertex to load
- PrimitiveCount : Number of primitives to render

DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);

D3DFVF_XYZRHW|D3DFVF_DIFFUSE

X	Y	Z	rhw	Diffuse
150.0	50.0	0.5	1.0	0xFFFF0000
250.0	250.0	0.5	1.0	0xFF00FF00
50.0	250.0	0.5	1.0	0xFF00FFFF





Result : Tutorial 2

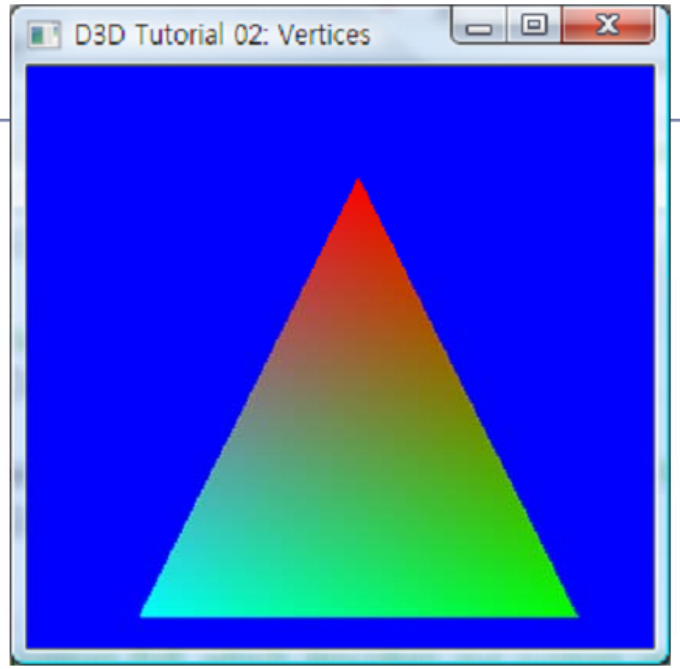
```
g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255), 1.0f, 0L );
g_pd3dDevice->BeginScene();

g_pd3dDevice->SetStreamSource( 0, g_pVB, 0, sizeof(CUSTOMVERTEX) );

g_pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );

g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );

g_pd3dDevice->EndScene();
g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
```



Practice Assignments

- Install DirectX
- Compile and run Tutorial 1 and 2
  - Draw two or more triangles
  - Try other primitives
- Review 3D Geometric Transformations