CS 529

Fundamentals of Game Development

DirectX

1 Introduction

1.1 DirectX background

- First introduced in 1995.
- DirectX is a collection of multimedia APIs. (Application Programming Interface)
  - Graphics.
  - Input.
  - Network.
  - Sound.
  - Videos.
- Each API can be used independently.
- APIs are hardware accelerated.
  - Greater performance compared to software accelerated applications.
- Technically, DirectX is a set of libraries which communicate with the hardware.
- Hardware manufacturers create drivers that are compatible with DirectX, thus making changes almost seamless to programmers.
- DirectX provides a hardware abstraction layer (HAL), because programmers don't need to know what the underlying hardware is.

1.2 DirectGraphics

- Responsible for the rendering part of an application.
- Previously, 2 graphics APIs existed: DirectDraw and Direct3D.
  - DirectDraw handled 2D rendering.
  - Direct3D handled 3D rendering.
- Now they are both combined into a single API called DirectGraphics or Direct3D.
- DirectDraw's functions aren't used in general, because Direct3D is now capable of handling both 2D and 3D.

1.3 DirectInput

- Used to access input devices.
  - Keyboards, mice, game controllers, joysticks...
- DirectInput is used to determine the state of input devices.
  - Button/Key pressed or not pressed
  - Degree of rotation of an analog pad

1.4 DirectPlay

- Handles networking.
- Allows an application to communicate with other machines over a connection.

1.5 DirectSound

- Responsible for sound related functionalities
- DirectSound gives programmers a high-level of control over sound operations.
  - Easier, but you have less control.

1.6 DirectMusic

- Responsible for lower level audio functionalities.
- Flexible, low level control.
  - More control, but more difficult than DirectSound.

2 Requirements

- Install the latest DirectX SDK version
- Add the DirectX "include folder" to your application: $(DXSDK_DIR)\Include
- Add the DirectX "library folder" to your application: $(DXSDK_DIR)\Lib\x86
- Add these libraries to your application:
  - d3d9.lib
  - dinput8.lib
  - dxguid.lib
  - d3dx9d.lib
- Include these 2 header files in you application:
  - d3d9.h
  - d3dx9.h

3 Direct3D

3.1 Initialization

- Create a Direct3D interface object.
  - Create a pointer to the “IDirect3D9” class.
  - LPDIRECT3D9 is a "IDirect3D9 *" typedef
  - Create an instance of this class

```
LPDIRECT3D9 d3dObj = NULL;

if((d3dObj = Direct3DCreate9(D3D_SDK_VERSION)) == NULL)
    EXIT_APPLICATION();
```

- Before we create the Direct3D device, which will be used in most Direct3D related functions, we can (it's not required) obtain some information about the current display:
  - Desktop resolution
  - Display format
  - Monitor refresh rate

```
D3DDISPLAYMODE dm;

d3dObj->GetAdapterDisplayMode(D3DADAPTER_DEFAULT, &dm);
```

  - Structure of D3DDISPLAYMODE:

```
typedef struct D3DDISPLAYMODE
{
    UINT Width;
    UINT Height;
    UINT RefreshRate;
    D3DFORMAT Format;
} D3DDISPLAYMODE, *LPD3DDISPLAYMODE;
```

- Now it's time to create a Direct3D device.
  - Create a pointer to the “IDirect3DDevice9” class.
  - LPDIRECT3DDEVICE9 is a " IDirect3DDevice9 *" typedef
  - Create an instance of this class

```
LPDIRECT3DDEVICE9 pDevice;

if (FAILED(d3dObj->CreateDevice(D3DADAPTER_DEFAULT,
D3DDEVTYPE_HAL, hWnd, behaviour, &d3dpp, &pDevice)))
    EXIT_APPLICATION();
```

This is the function's definition:

```
HRESULT Idirect3D9::CreateDevice(UINT adapter, D3DDEVTYPE deviceType,
HWND focusWindow, DWORD behaviourFlags, D3DPRESENT_PARAMETERS
*presentationParameters, IDirect3DDevice9 **device);
```

- **Unit adapter**: Denotes the display adapter. Use D3DADAPTER_DEFAULT to use the primary display adapter.
- **D3DDEVTYPE:** Choose D3DDEVTYPE_HAL, which will allow the application to use hardware rasterization.
- **HWND:** A handle to the application's window. When creating the window, save a pointer to the window handle and use it later when creating the device instance.

```
//Global
HWND globalHwnd;

// Create the application's window
globalHwnd = CreateWindow("Win32", "Win32", WS_OVERLAPPEDWINDOW,
100, 100, 800, 600, NULL, NULL, wc.hInstance, NULL);
```

- **DWORD behaviourFlags:** Used to determine if vertex processing is done in software or hardware. It can be D3DCREATE_HARDWARE_VERTEXPROCESSING or D3DCREATE_SOFTWARE_VERTEXPROCESSING.
  Nowadays, most, if not all video cards of capable of vertex processing, so input D3DCREATE_HARDWARE_VERTEXPROCESSING for this parameter.
- **D3DPRESENT_PARAMETERS**: Used to determine how your scene is presented. You don't have to fill all the parameters; you can set some of them to NULL. Here are the most important ones:

```
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory(&d3dpp, sizeof(d3dpp));
```

```
//Width and height of the back buffer
d3dpp.BackBufferWidth = 800;
d3dpp.BackBufferHeight = 600;

//Wait for v_blank? Or swap buffer as soon as we're done rendering?
d3dpp.PresentationInterval = D3DPRESENT_INTERVAL_IMMEDIATE;

//Windowed or full screen?
d3dpp.Windowed = TRUE;

d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;

//Format of the backbuffer
d3dpp.BackBufferFormat = D3DFMT_A8R8G8B8;

//Do we have z buffering?
d3dpp.EnableAutoDepthStencil                    = TRUE;

//Format of the z buffer
d3dpp.AutoDepthStencilFormat                    = D3DFMT_D16;


// setting the projection transformation
D3DXMATRIX matProj;
D3DXMatrixPerspectiveFovRH(&matProj, D3DXToRadian(90.0f), 4.0f/3.0f, 1.0f,
1000.0f);
pDevice->SetTransform(D3DTS_PROJECTION, &matProj);

//setting the view transformation
D3DXMATRIX matView;
D3DXVECTOR3 camPos(0, 0, 60), camLookAt(0, 0, 0), camUp(0, 1, 0);
D3DXMatrixLookAtRH(&matView, &camPos, &camLookAt, &camUp);
pDevice->SetTransform(D3DTS_VIEW, &matView);

//Turning off lights
pDevice->SetRenderState(D3DRS_LIGHTING, false);

//Culling clockwise triangles
pDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
```

- Now that we have initialized Direct3D, we can use it (Basically through the device LPDIRECT3DDEVICE9 pDevice) in order to change, enable, disable and set hardware flags, and obviously to render shapes.

3.2 Rendering using Direct3D

- This part assumes that a Direct3D object and device were successfully created and initialized.
- Each loop, we need to reset some buffers in order to avoid using what was collected during the previous loop.
  - The backbuffer should be cleared (The programmer chooses a default color for it).
  - The z buffer (which holds the z value for each pixel) should be cleared (Again, the programmer can choose a default value for it)
  - Clearing these buffers is similar to using memset to set each element to a certain value.
- After clearing the backbuffer, we should inform DirectX that we are about to render objects.
- After rendering everything, we should inform DirectX that we're done rendering for this loop.
- Remember that we're rendering to the backbuffer and not directly to the screen buffer.
  - This means that the shapes we rendered now are still hidden.

- In order to show what was just rendered on the backbuffer, we need to swap the backbuffer with the primary buffer

```
void Render(void)
{
        // Clear the backbuffer and the zBuffer
        //Default color is black, and default Z value is 1.0
        pDevice->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(0, 0, 0), 1.0f, 0);

        //Inform DirectX that we are about to render some shapes
        pDevice->BeginScene();

        //Inform DirectX that we are done rendering
        pDevice->EndScene();

        //Swap the backbuffer with the primary buffer
        pDevice->Present (0, 0, 0, 0);
}
```

3.3 Primitives

- **What is a primitive?**
  - A primitive is a basic shape used in computer graphics.
  - There are several types of primitives:
    - Point
    - Line
    - Triangle
    - Quad
    - Polygon
  - A group of primitives can be drawn together in order to form more complex shapes.
  - The "Triangle" primitive is the most commonly used one, because nowadays, hardware is optimized to draw triangles.
  - Note: Because the "triangle" primitive is the most used one, many books and documents refer to it as "polygon".
  - Usually, we will have to create a vertex buffer, where each vertex represents a single point.
  - The way these vertices are connected depends on which primitive type is selected upon drawing.

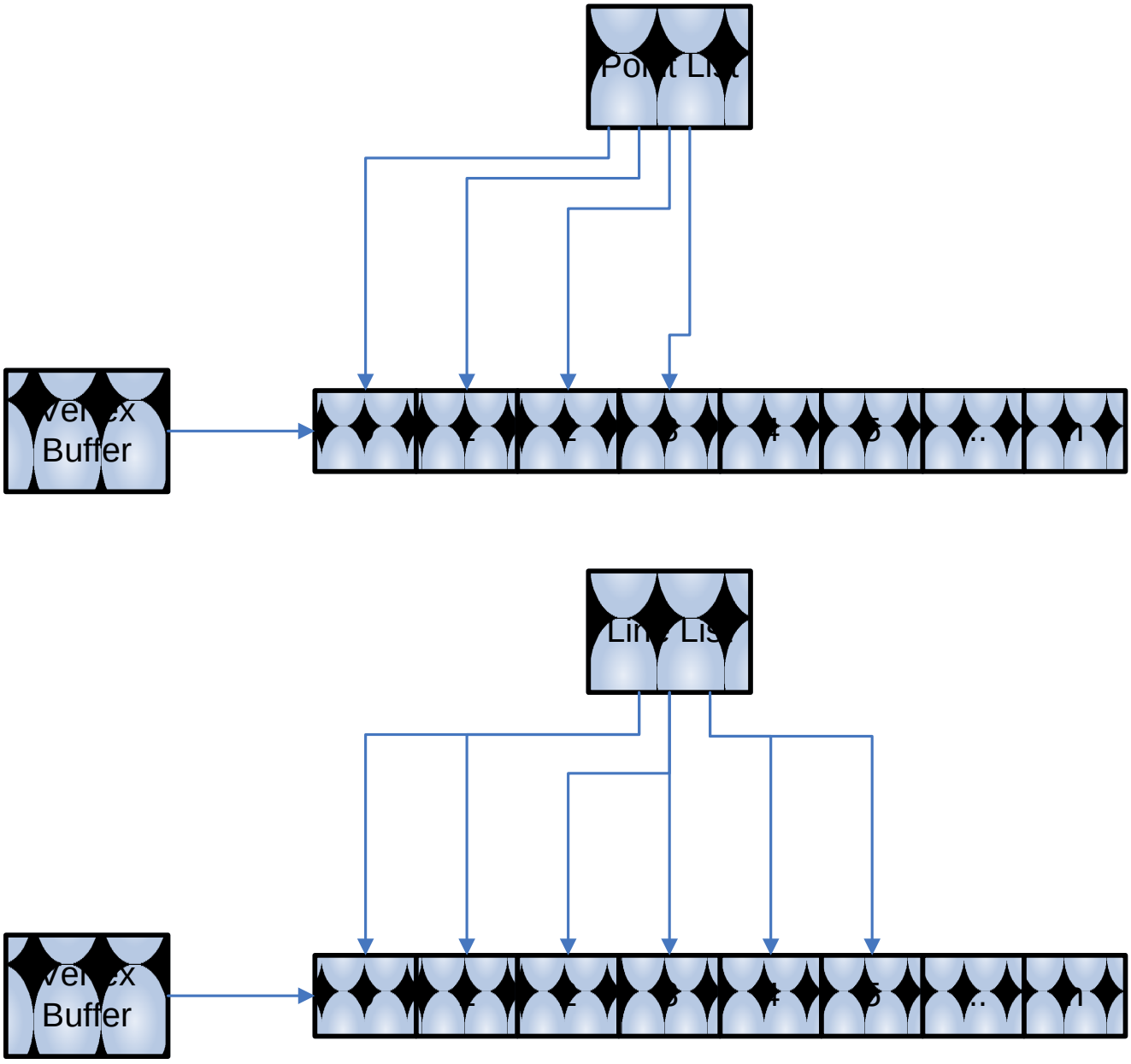
  - Primitive types in Direct3D:

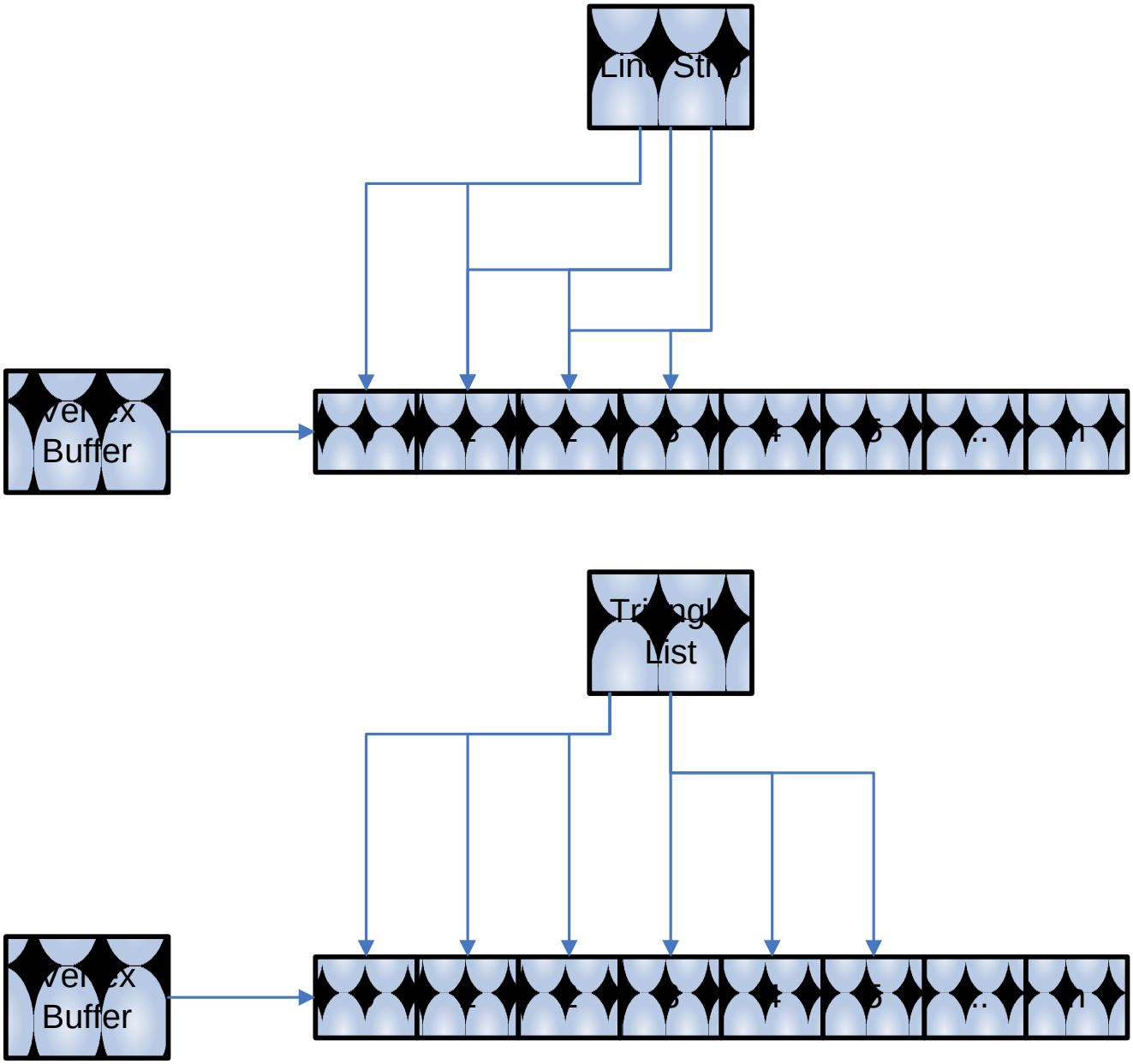
```
typedef enum _D3DPRIMITIVETYPE
{
    D3DPT_POINTLIST         = 1,
    D3DPT_LINELIST          = 2,
    D3DPT_LINESTRIP         = 3,
    D3DPT_TRIANGLELIST      = 4,
    D3DPT_TRIANGLESTRIP     = 5,
    D3DPT_TRIANGLEFAN       = 6,
    D3DPT_FORCE_DWORD       = 0x7fffffff, /* force 32-bit size enum */
} D3DPRIMITIVETYPE;
```

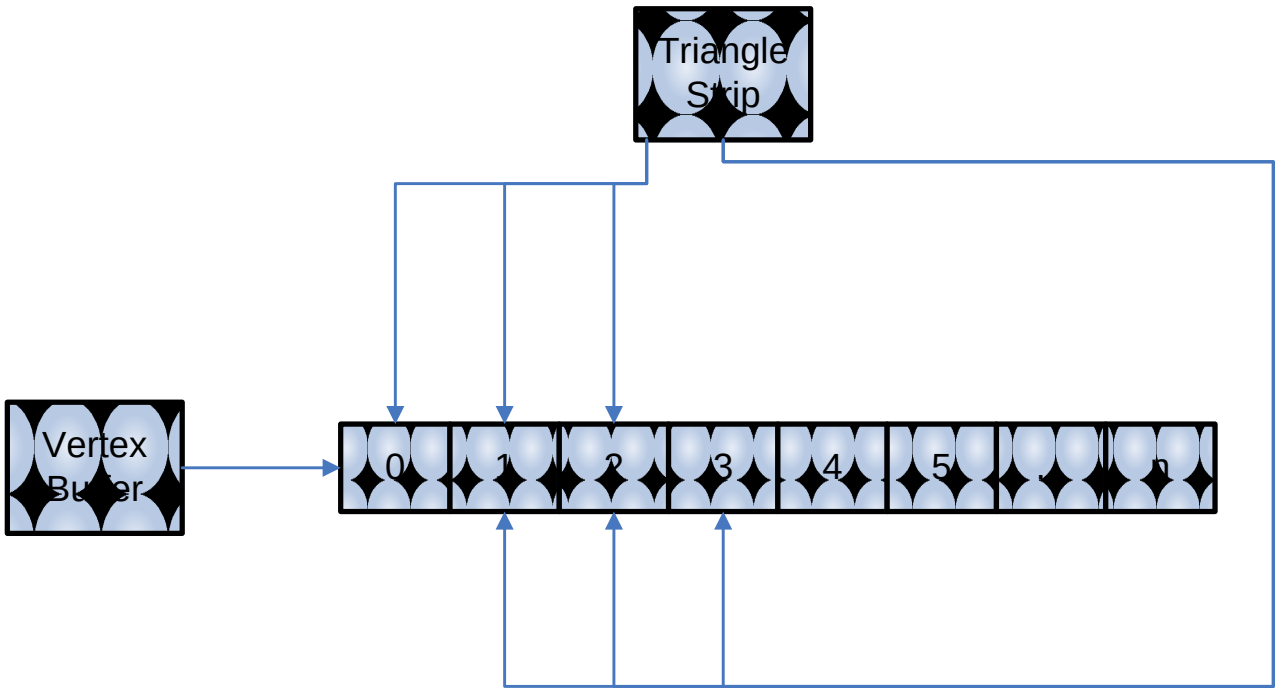
    - D3DPT_POINTLIST: The points will be rendered separately without any connection among them.
    - D3DPT_LINELIST: The points will be connected 2 by 2. Points {0,1} will form a line, points {2,3} will form a line and so on...
    - D3DPT_LINESTRIP: The points will be connected 2 by 2 without any discontinuation. Points {0,1}will form a line, points {1,2}, points {2,3} will form a line and so on...
    - D3DPT_TRIANGLELIST: Each 3 points will form a triangle. Points {0,1,2} will form a triangle, points {3,4,5} will form a triangle and so on...
    - D3DPT_TRIANGLESTRIP: Each 3 points will form a triangle without any discontinuations. Points {0,1,2} will form a triangle, points {1,2,3} will form a

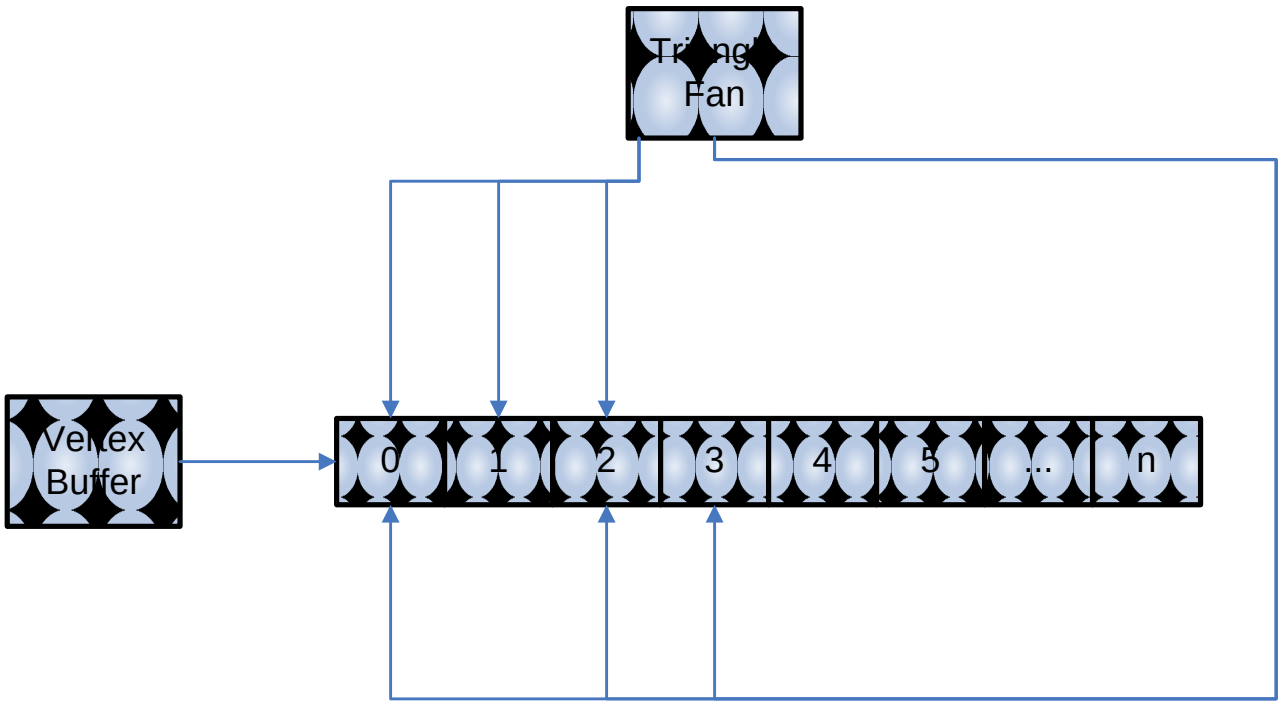
triangle, points {2,3,4} will form a triangle and so on...

- D3DPT_TRIANGLEFAN: Each 2 points will form a triangle along with point 0. Point {0,1,2} will form a triangle, points {0,2,3} will form a triangle, points {0,3,4} will form a triangle and so on...









Creating the appropriate FVF (flexible vertex format)

- A vertex can hold a lot of information:
  - Position
  - Color
  - Texture coordinates (Can be more than 1)
  - Normal
  - ...

- A shape might require its vertices to have positions and colors only, while other shape might require more data.
- For each **unique** vertex structure, we should create the appropriate FVF

```
//This FVF has a position and a color for each vertex
#define VERTEX_FVF_POS_COLOR (D3DFVF_XYZ | D3DFVF_DIFFUSE)

//The vertex structure
struct VtxPosCol
{
	float x, y, z;
	int c;
};

//Creating an array of vertices
```



```
VtxPosCol vtx[3];

vtx[0].x = -5.0f; vtx[0].y = -5.0f;   vtx[0].z = 0.0f;   vtx[0].c = 0.0f;
vtx[1].x = 5.0f;  vtx[1].y = -5.0f;   vtx[1].z = 0.0f;   vtx[1].c = 0.0f;
vtx[2].x = 0.0f;  vtx[2].y = 5.0f;    vtx[2].z = 0.0f;   vtx[2].c = 0.0f;
```

- **Drawing primitives**

  - Direct3D should be informed of the current shape's FVF before rendering it.
    pDevice->SetFVF(VERTEX_FVF_POS_COLOR);

  - There are 2 main ways to draw primitives.
    - Send primitives from the user memory in order to be rendered. (Slow but easy)
    - Create primitives on the graphics memory. (Fast, but requires an initialization process).

  - Sending primitives from the main memory

    - This method is slow, because it involves sending data from user memory to graphics memory each time the function is called.

    - It's done using the Direct3D device function:
      HRESULT DrawPrimitiveUP(D3DPRIMITIVETYPE *PrimitiveType***,**
      UINT *PrimitiveCount***,**
      CONST void* *pVertexStreamZeroData***,**
      UINT *VertexStreamZeroStride***)**;

      D3DPRIMITIVETYPE *PrimitiveType*: Which primitive type? It should be a member of the "D3DPRIMITIVETYPE" enumeration. (Check **What is a primitive?)**

      UINT *PrimitiveCount*: How many primitives? Not to be confused with the number of vertices. If you are rendering 1 triangle using 3 vertices, then the number of primitives is 1 and not 3.

      CONST void* *pVertexStreamZeroData*: Pointer to the array of vertices

      UINT *VertexStreamZeroStride*: Size of the vertex structure.

  - Example (Sending vertices from the main memory):
    ```
    //This FVF has a position and a color for each vertex
    #define VERTEX_FVF_POS_COLOR (D3DFVF_XYZ | D3DFVF_DIFFUSE)
    ```

```
//The vertex structure
struct VtxPosCol
{
	float x, y, z;
	int c;
};

//Creating an array of vertices
VtxPosCol vtx[3];

vtx[0].x = -5.0f; vtx[0].y = -5.0f;	vtx[0].z = 0.0f;	vtx[0].c = 0.0f;
vtx[1].x = 5.0f;  vtx[1].y = -5.0f;	vtx[1].z = 0.0f;	vtx[1].c = 0.0f;
vtx[2].x = 0.0f;  vtx[2].y = 5.0f;	vtx[2].z = 0.0f;	vtx[2].c = 0.0f;

//Setting the FVF of the current shape
pDevice->SetFVF(VERTEX_FVF_POS_COLOR);

//Drawing the current shape
//Parameter1: Primitive type. A member of _D3DPRIMITIVETYPE. Check
//primitive section.
//Parameter2: Number of primitives to render. (Not the number of vertices)
//Parameter3: Array of vertices
//Parameter4: Size of the vertex structure.
pDevice->DrawPrimitiveUP(D3DPT_TRIANGLELIST, 1, vtx, sizeof(VtxPosCol));
```

- Creating a vertex buffer on the video card

  - This is the most used method.
  - It doesn't involve sending vertices from the main memory each time they're being rendered.
  - Vertices are created and placed on the video card memory.
  - Many vertex buffers can be created and placed (as long as there is free memory left)
  - When rendering, the user has to just point out which vertex buffer needs to be rendered, along with the correspondent FVF (flexible vertex format).
  - Creating a vertex buffer on the video card is done using this Direct3D device function:

    HRESULT CreateVertexBuffer(UINT *Length***,**
    DWORD *Usage***,**
    DWORD *FVF***,**
    D3DPOOL *Pool***,**
    Idirect3DVertexBuffer9** *ppVertexBuffer***,**
    HANDLE* *pSharedHandle***);**

UINT *Length*: Size of the vertex buffer (Size of 1 vertex * number of vertices)

DWORD *Usage***:** Determine the usage of this vertex buffer (See D3DUSAGE). Example: D3DUSAGE_DONOTCLIP, D3DUSAGE_WRITEONLY... Usually this value is set to 0.

DWORD FVF**:** The FVF of the vertex buffer (Flexible vertex format).

D3DPOOL Pool**:** Defines the memory class of this vertex buffer. Usually D3DPOOL_MANAGED is chosen, which will backup the vertices in case the device is lost (therefore you don't need to recreate them manually in case the device was lost and needs to be reinitialized).

**IDirect3DVertexBuffer9 ****ppVertexBuffer*: Address of a pointer to IDirect3DVertexBuffer9. This is the buffer that will be created.

HANDLE* pSharedHandle**:** Reserved and never used. Always set to 0.

The function return D3D_OK if it succeeds.

- Now that we have created a vertex buffer, we have to initialize it. This means we have to set a position and a color (and any other characteristics in the vertex structure) for each vertex in that buffer.
- Remember that the vertex buffer we created is located on the graphics memory (VRAM), and not the user memory (RAM).
- This buffer has to be locked first, which will give us access to it.
- After assigning new values to the vertices of that particular buffer, we should unlock it back.

- Example (Creating a vertex buffer in the graphics memory)

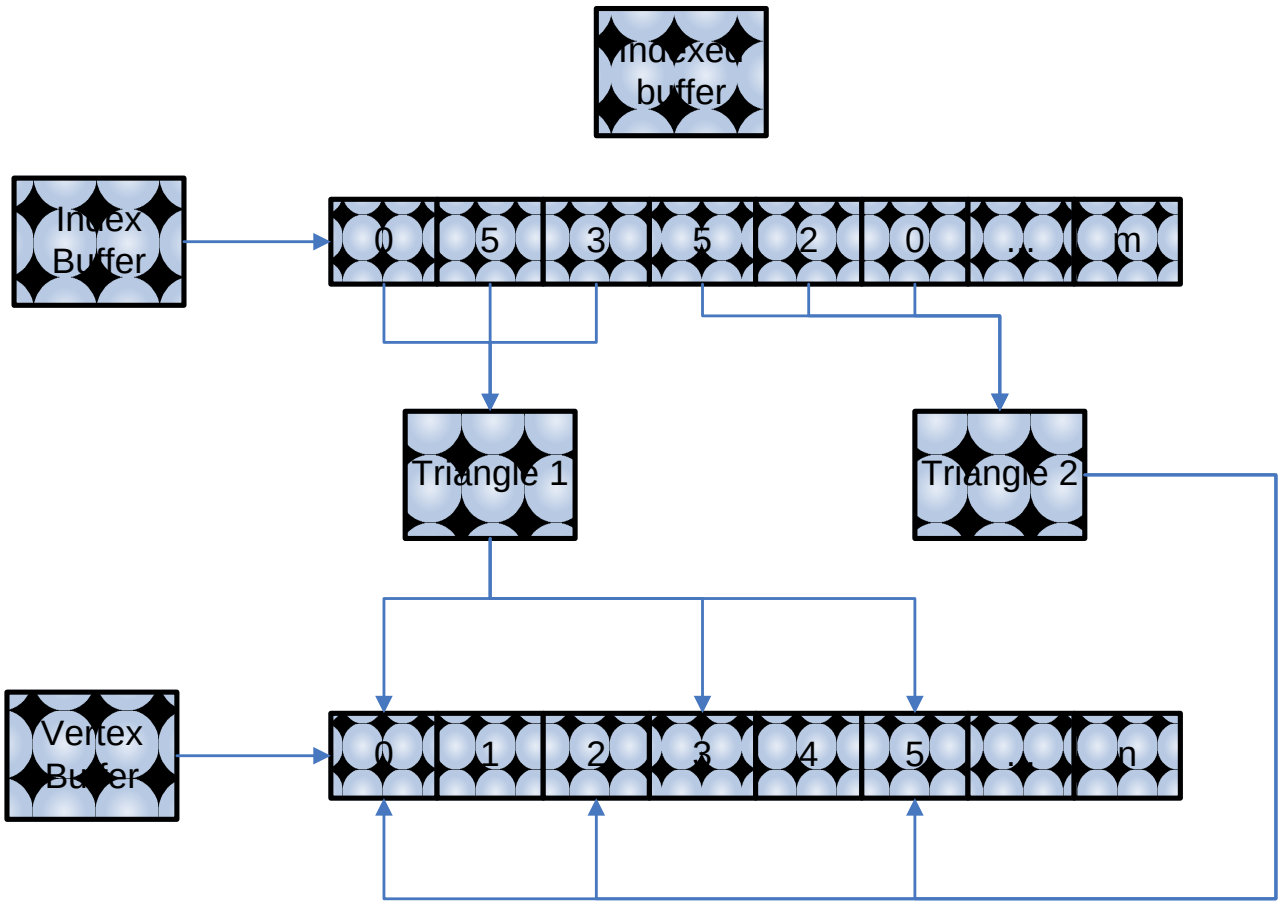
```
//This FVF has a position and a color for each vertex
#define VERTEX_FVF_POS_COLOR (D3DFVF_XYZ | D3DFVF_DIFFUSE)

//The vertex structure
struct VtxPosCol
{
      float x, y, z;
       int c;
};

//Creating an array of vertices (in RAM)
```

```
VtxPosCol vtx[3];

vtx[0].x = -5.0f; vtx[0].y = -5.0f;     vtx[0].z = 0.0f;     vtx[0].c = 0.0f;
vtx[1].x = 5.0f;  vtx[1].y = -5.0f;     vtx[1].z = 0.0f;     vtx[1].c = 0.0f;
vtx[2].x = 0.0f;  vtx[2].y = 5.0f;      vtx[2].z = 0.0f;     vtx[2].c = 0.0f;

//Creating a pointer to the vertex buffer structure
IDirect3DVertexBuffer9 *pVB;

//Creating the vertex buffer on the video memory
if(FAILED(pDevice->CreateVertexBuffer(3*sizeof(VtxPosCol), 0,
VERTEX_FVF_POS_COLOR, D3DPOOL_MANAGED, &pVB, NULL)))
    return E_FAIL;

VtxPosCol *pVertices;
//Locking the vertex buffer, which will allow us to change the values of its vertices
//Parameter1: UINT offset. Set 0 to begin from the first element of the buffer.
//Parameter2: UINT size to lock. What's the size of the elements we're locking?
//Parameter3: VOID **. Address of the vertex array which will contain the retrieved
//data.
//Parameter4: Flags like read-only, do not wait... See D3DLOCK.
if(FAILED(pVB->Lock(0, 3*sizeof(VtxPosCol), (void**)&pVertices, 0)))
    return E_FAIL;

//Changing the values of the vertices. You can set value manually, read them from a
file, or copy them from another array like the example below
memcpy(pVertices, vtx, 3*sizeof(VtxPosCol));

//Unlocking the vertex buffer (we can't change its vertices' values anymore, unless
we lock it back)
pVB->Unlock();

//Rendering the vertex buffer

//Choosing which vertex buffer to render
//Parameter1: Stream number. Usually it's 0, unless you want to apply more than
//one vertex shader
//Parameter2: Pointer to the vertex buffer which will be rendered.
//Parameter3: Offset in bytes, in case we do not want to start from the beginning of
//the stream. Usually it's set to 0.
//Paramater4: Size of the vertex structure.
pDevice->SetStreamSource(0, pVB, 0, sizeof(VtxPosCol));

//Setting the FVF of the selected vertex buffer
pDevice->SetFVF(VERTEX_FVF_POS_COLOR);
```

```
//Drawing the vertex buffer
//Parameter1: Primitive type. A member of _D3DPRIMITIVETYPE (Check
//primitive section)
//Parameter2: Index of the first vertex. Usually it's 0.
//Parameter3: Number of primitives (Not number of vertices)
pDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
```

- **Drawing indexed primitives**

  - So far we covered rendering non-indexed primitives.
  - This means that vertices are saved in a consecutive order inside the vertex buffer.
    - A single vertex can be used for 1 or maybe 2 primitives.
  - But what if we want to draw a triangle using vertex #0, vertex#11 and vertex #307?
  - These 3 vertices are obviously not consecutive in the vertex buffer, but they are way apart.
  - This is very common in 3D models which are made of hundreds of connected triangles.
  - One solution would be to create duplicate data, but obviously it would be a great waste of resources.
  - In order to render non-consecutive vertices as a single primitive, we need to create an index buffer.
  - The index buffer, as its name says, will index the primitives 1 by 1.
    - Now instead of going through the vertex buffer to determine which vertices to draw, the graphics card will go through the index buffer to determine which vertices to draw.



- We will have to create an index buffer (and the previous vertex buffer too).
- **HRESULT CreateIndexBuffer(UINT** *Length*,
  DWORD *Usage*,
  D3DFORMAT *Format*,
  D3DPOOL *Pool*,
  IDirect3DIndexBuffer9** *ppIndexBuffer*,
  HANDLE* *pSharedHandle*);

- After creating the index buffer, we should initialized it.
- Similarely to the process of initializing the vertex buffer, we have to lock it, set new values then unlock it.
- Finally, make sure to select the index buffer before rendering the vertices.

- Example: Drawing an indexed vertex buffer (This part assumes that we have already


```
created a vertex buffer. Check previous section)

//Creating a pointer to the index buffer structure
IDirect3DIndexBuffer9 *pIB;

//Creating the index buffer
//Parameter1: Size of the index buffer. The size of each element is 2 bytes, therfore
//the total size is: Number of indices*2, which is Number of Triangles * 3 * 2
//Parameter2: Usage Determine the usage of this vertex buffer (See D3DUSAGE).
//Example: D3DUSAGE_DONOTCLIP, D3DUSAGE_WRITEONLY... Usually
//this value is set to 0.
//Parameter3: Format of the index, usually it's 2 bytes, so choose
//D3DFMT_INDEX16
//Parameter4: The memory pool of this index buffer. Choose
//D3DPOOL_MANAGED so that you don't have to worry about recreating this
//buffer in case the device was lost.
//Parameter5: Address of a pointer to the index buffer structure. This is the index
//buffer that will be created it.
//Parameter6: Not used, set to 0.
pDevice->CreateIndexBuffer(3*2*sizeof(unsigned char), 0, D3DFMT_INDEX16,
D3DPOOL_MANAGED, &pIB, 0);


unsigned short *pIndices;
//Locking the index buffer, which will allow us to change its values (the indices)
//Parameter1: UINT offset. Set to 0 to begin  from the first element of the buffer.
//Parameter2: UINT size to lock. What's the size of the elements we're locking?
//Parameter3: VOID **. Address of the index array which will contain the
//retrieved data.
//Parameter4: Flags like read-only, do not wait... See D3DLOCK
pIB->Lock(0, 3*sizeof(unsigned short), (void **)&pIndices, 0);

//Filling the retrieved index buffer manually (we can do a memcpy from another
//array, or get data from a file...)
pIndices[0] = 0;
pIndices[1] = 3;
pIndices[2] = 1;

//Unlocking the index buffer (we can't change its values anymore, unless we lock
//it back)
pIB->Unlock();


//Rendering
```



```
//Choosing which vertex buffer to render
//Parameter1: Stream number. Usually it's 0, unless you want to apply more than one
//vertex shader
//Parameter2: Pointer to the vertex buffer which will be rendered.
//Parameter3: Offset in bytes, in case we do not want to start from the beginning
//of the stream. Usually it's set to 0.
//Paramater4: Size of the vertex structure
pDevice->SetStreamSource(0, pVB, 0, sizeof(VtxPosCol));

//Setting the correspondant FVF
pDevice->SetFVF(VERTEX_FVF_POS_COLOR);

//Setting the index buffer
pDevice->SetIndices(pIB);

//Drawing the vertex buffer using the selected index buffer
//Parameter1: Primitive type. A member of _D3DPRIMITIVETYPE (Check
//primitive section)
//Parameter2 & 3: Offset and starting indices in the vertex buffer. Usually set to 0
//Parameter4: Number of vertices you want to render.
//Parameter5: Start index in the index buffer
//Parameter6: Number of primitives.
pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0, 3, 0, 1);
```