Drawing Primitives

Rendering Objects

• Direct3D is geared towards 3D visualization.
  – Affects the type of primitives that can be rendered
  – Primitives should be easily rendered in 3D

• Issues
  – Accuracy of the computer
  – Consistency of the rendered data
    • Plane should stay a plane
    • Connected lines should stay connected
Primitive Objects

- Direct3D supports the following primitives
  - Points – 0D objects
  - Lines – 1D objects
  - Triangles – 2D objects

2D and 3D Objects
Direct3D Primitives

- Point lists
- Line lists
- Line Strip
- Triangle lists
- Triangles Strips
- Triangle fans

Point List

- Vertices
  - (20, 20)
  - (30, 110)
  - (40, 50)
  - (70, 80)
  - (80, 40)
  - (100, 60)
  - (100, 20)
  - (150, 60)
Line List

• Vertices
  – (20, 20)
  – (30, 110)
  – (40, 50)
  – (70, 80)
  – (80, 40)
  – (100, 60)
  – (100, 20)
  – (150, 60)

Doron Nussbaum
COMP 3501 Drawing Primitives

Line Strip

• Vertices
  – (20, 20)
  – (30, 110)
  – (40, 50)
  – (70, 80)
  – (80, 40)
  – (100, 60)
  – (100, 20)
  – (150, 60)
Triangle List

- Vertices
  - (20, 20)
  - (30, 110)
  - (40, 50)
  - (70, 80)
  - (80, 40)
  - (100, 60)
  - (100, 20)
  - (150, 60)

Triangle Strip

- Point list
  - (20, 120)
  - (25, 165)
  - (30, 135)
  - (45, 150)
  - (50, 130)
  - (60, 140)
  - (60, 120)
  - (85, 140)
Triangle Fan

- Vertices
  - (20, 20)
  - (30, 110)
  - (40, 50)
  - (70, 80)
  - (80, 40)
  - (100, 60)
  - (100, 20)
  - (150, 60)

Vertices

- Vertices are the fundamental building blocks
- Vertices are 0D objects but they implicitly define (topology)
  - 1D (lines)
  - 2D (triangles)
  - 3D (tetrahedron)
- The objects are made by a “decision” on how to connect the vertices
- Vertices are usually shared among a number of objects (triangles)
  - Magic number is 6
Vertex data

- Vertices are associated with different type of data
  - Spatial – position in space
  - Attributes – colour
  - Geometry
    - Normal
    - Texture coordinates

```c
struct myVertex {
    D3DVECTOR3 pos;
    D3DCOLOR color;
};

struct anotherVertex {
    D3DVECTOR3 pos;
    D3DVECTOR3 normal;
    D3DVECTOR2 TexCoords;
};
```

Doron Nussbaum  COMP 3501 Drawing Primitives  13

The two vertices contain the same data
Vertex Declaration

- A structure that describes the content of the vertex structure
- One structure for each field in the vertex structure

```
typedef struct D3DVERTEXELEMENT9 {
    WORD Stream;
    WORD Offset;
    BYTE Type;
    BYTE Method;
    BYTE Usage;
    BYTE UsageIndex;
} D3DVERTEXELEMENT9,*LPD3DVERTEXELEMENT9;
```

Output stream – Use 0
Offset – offset in bytes from the beginning of structure
Type – what is the field type (see D3DDECLTYPE)
Common types:
- D3DDECLTYPE_FLOAT1 a float
- D3DDECLTYPE_FLOAT3 a 3D float vector
- D3DDECLTYPE_D3DCOLOR – color as a 4D vector

Method – used in tessellation
Use D3DDECLMETHOD_DEFAULT

Usage – What does the field store (see D3DDECLUSAGE)
Common types
- D3DDECLUSAGE_POSITION
- D3DDECLUSAGE_NORMAL

Usage Index – differentiates between multiple similar fields

Examples

```
struct myVertex {
    D3DVECTOR3 pos;
    D3DVECTOR3 normal;
} ;
D3DVERTEXELEMENT9 decl[] = {
    {0,0, D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0}
} D3DECL_END();
```

```
struct anotherVertex {
    D3DVECTOR3 pos;
    D3DVECTOR3 normal0;
    D3DVECTOR3 normal1;
} ;
D3DVERTEXELEMENT9 decl[] = {
    {0,0, D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0}
    {0,12, D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 0}
    {0,24, D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 1}
} D3DECL_END();
```
Vertex Declaration

- Store the vertex declaration structure in an object that can be invoked by Direct3D

```c
struct myVertex {
    D3DVECTOR3 pos;
    D3DVECTOR3 normal;
} ;

D3DVERTEXELEMENT9 decl[] = {
    {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
    {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 0}
} ;

IDirect3DVertexDeclaration9* myVertexDecl = NULL;
d3dDev->CreateVertexDeclaration(decl, &myVertexDecl);
```

Alternative Flexible Vertex Format

- Each vertex has some properties
  - Location
  - Colour
    - Diffuse
    - Specular
  - Texture information
- Not all data is required all the time
  - Waste of space
Flexible Vertex Format

- Select only attributes that are needed
- #define D3DFVF_MYVERTEX (D3D_XYZ | D3DFVF_DIFFUSE)

- D3DFVF_XYZ
  - the location of the point
  - Type – three float values
- D3DFVF_DIFFUSE – a color for the vertex

Possible attributes of FVF

- D3DFVF_XYZ
  - 3 floats
- D3DFVF_XYZRHW
  - Location of vertex (in screen coordinates)
  - 4 floats
- D3DFVF_DIFFUSE
  - Colour of diffuse lighting
  - 32 bits color
- D3DFVF_SPECULAR
  - Colour of specular lighting
  - 32 bits color
- D3DFVF_TEX0 - D3DFVF_TEX8
  - Coordinates for texture mapping
  - 2 floats each
#define MYVERTEXFVF (D3D_XYZ | D3DFVF_COLOR)

- The corresponding data structure
  - Onus is on the programmer to declare the structure

struct MyVertex{
  D3DVECTOR3 v; // the vertex location
  DWORD colour; // the colour of the vertex
}

Constructing Triangles

- Quad
  - Quad vertex quad[6] {
    v0, v1, v2; // Triangle 1
    v0, v2, v3; // Triangle 2
  }

- Coarse Circle
  - Coarse circle vertex circle [24] {
    v8, v0, v1; // Triangle 1
    v8, v1, v2; // Triangle 2
    v8, v2, v3; // Triangle 3
    v8, v3, v4; // Triangle 4
    v8, v4, v5; // Triangle 5
    v8, v5, v6; // Triangle 6
    v8, v6, v7; // Triangle 7
    v8, v7, v0; // Triangle 8
  }
Using Indices

- Using vertices is problematic
  - Each vertex appears 6 times (expected number)
  - Waste of space –
    - E.g., Position 16B/vertex (four floats)
  - Redundancy –
    - Update is time consuming
    - Expensive book keeping
  - Waste of computation resources
    - 6 times the amount of work

- Solution
  - Store the vertices in one location
  - Use indirect access to the vertex information - indices

- Advantages
  - Indirect access
    - Independent of the vertex data – can be associated with multiple sets of vertices
  - No redundancy
    - Vertex data appears only once
    - Changing of vertex data is easy – occurs only once
  - Simple book keeping
  - Saves space – “short” or “long” as indices (2B or 4B)
  - Speeds computation resources
    - Each vertex is process only once

- Disadvantage
  - Requires two data structures
Defining Triangles using Indices

- **Quad**
  
  Vertex v[4] = {v0,v1,v2,v3}
  
  long indexQuad[6] {
    0, 1, 2, // Triangle 1
    0, 2, 3 // Triangle 2
  }

- **Coarse Circle**
  
  
  long indexCircle[24] {
    8, 0, 1 // Triangle 1
    8, 1, 2, // Triangle 2
    8, 2, 3 // Triangle 3
    8, 3, 4 // Triangle 4
    8, 4, 5 // Triangle 5
    8, 5, 6 // Triangle 6
    8, 6, 7 // Triangle 7
    8, 7, 0 // Triangle 8
  }

Doron Nussbaum

COMP 3501 Drawing Primitives
Vectors in DirectX

• Provides basic vector type
  – D3DVECTOR3 vec(x,y,z)
  – D3DVECTOR4 vec(x,y,z,w) // homogenous

• Provides basic manipulation of vectors
  – Basic operations - +, -, *, /
  – Dot product – prod = D3DXVec3Dot(&inV1, &inV2)
  – Cross product – pOutV1 = D3DXVec3Vec(&outV, &inV1, &inV2)
  – Normalize – pOutV1 = D3DXVec3Normalize(&outV, &inV1)
  – Magnitude/length – len = D3DXVec3Length(&inV1)

Matrices in DirectX

• Basic type
  – D3DMATRIX - a 4x4 matrix

• Provides basic manipulation of vectors
  – Basic matrix-matrix operations – +, -, *
  – Basic matrix scalar operations – *, /
  – Identity – D3DXMatrixIdentity(*outMat)
  – Transpose – D3DXMatrixTranspose(*outMat, *inMat)
  – Inverse – D3DXMatrixInverse(*outMat, *inDet,*inMat)
  – Multiply – D3DXMatrixMultiply(*outMat, *inMat1,*inMat2)
  – Transpose – D3DXMatrixTranspose(*outMat, *inMat)
Object graphics initialization

- Declare the vertex structure
- Allocate memory for the vertices of the object
- Initialize the vertices
- Allocated memory for the vertices in the video memory
- Load the vertices to the video memory

Create Vertex Buffer

- A device function
- Purpose: creates a placeholder/container for the vertices
- Receive a handle to the vertex buffer

```c
struct MyVertex{
    float x, y, z;  // position
    DWORD colour;  // colour
};

struct MyVertex vtx[6];

vtx[0].x = vtx[0].y = vtx[0].z = 10.0;
Vtx[0].colour = D3DCOLOR_XRGB(255,0,0)

CreateVertexBuffer( )
```

```c
struct myVertex {
    D3DVECTOR3 pos;
};

IDirect3DVertexBuffer9 *vBuf;

d3dDev->CreateVertexBuffer(3 * sizeof(struct myVertex), // size
0, // usage
D3DFVF_XYZ | D3DFVF_DIFFUSE, // flexible vertex format
D3DPOOL_MANAGED, // video memory or managed resource
&vBuf, // address of handle
NULL) // set to NULL
```
Memory pools

• Purpose: state where to store the resource

Options
• D3DPOOL_DEFAULT
  – Let Direct3D choose the best location for the resource (e.g., system mem, video mem)
• D3DPOOL_MANAGED
  – Direct3D manages the resource (moving to and from as needed automatically).
  – Backup copy is kept by Direct3D in system memory
• D3DPOOL_SYSTEMMEM
  – Asks the resource to be in system memory
• D3DPOOL_SCRATCH
  – Asks the resource to be in system memory
  – Device cannot access this resource
  – This resources can be copied to and from each other

Usage

• Purpose – specifies how the buffer is used

• D3DUSAGE_DYNAMIC
  – Buffer will be modified during the program execution
  – Usually placed in the Accelerated Graphics Port (AGP) memory
  – Can be updated fast
  – Memory must be copied to the video card
• D3DUSAGE_POINTS
  – Buffer will hold point primitives (particle systems)
• D3DUSAGE_SOFTWAREPROCESSING
  – Vertex processing is done by software
• D3DUSAGE_WRITEONLY
  – Application will only write to this location
  – Will be placed in a location that supports fast writing_
Notes

- Writing to and reading from video memory and or AGP is very slow
  - Keep a copy of the geometry locally

- Memory that is not declared dynamic is static
  - Used to store data that does not change (e.g., streets, houses, terrain, etc.)
  - Stored in the video memory

Accessing the Buffer’s Memory

- Buffer contains a number of fields (e.g., size, memory type, usage…)
- One of the structure fields is the geometry
- Two functions are used to obtain access to the buffer geometry section
  - IDirect3dVertexBuffer::Lock()
  - IDirect3dVertexBuffer::Unlock()
The Lock function

- **Purpose**
  - Locks a portion of the geometry buffer
- **Offset** - number of bytes from the start of buffer
- **Size** – the number of bytes to lock (0 == lock the array)
- **ppBufAddress** – address of the locked section
- **Flags** –
  - **D3DLOCK_DISCARD**
    - Use only with dynamic memory mode
    - Buffer is used only once
  - **D3DLOCK_NOOVERWRITE**
    - Append mode
    - Allows device to keep rendering
    - Used only with dynamic memory mode
  - **D3DLOCK_READONLY**
    - Locking for reading
    - No writing

Load Vertex Buffer

1. Declare a void pointer
   ```c
   void* pVtx;
   ```
2. Ask the system for an exclusive write privilege to a video memory location
   ```c
   g_pVB->Lock(0,sizeof(MyVertex)*6, &pVtx,0));
   ```
3. Copy the vertices to the video memory
   ```c
   memcpy(pVertices, vtxBuf, sizeof(MyVertex)*6);
   ```
4. Release the write privilege
   ```c
   g_pVB->Unlock();
   ```
Setting the Vertex Buffer

```c
struct myVertex {
  D3DVECTOR3 pos;
} vbuf[3];
IDirect3DVertexBuffer9 *gBuf; // the graphics memory
void *vBuf;
Vertex* vertices;

vBuf = gBuf->Lock(0, 3*sizeof(Vertex), (void**)&vBuf, 0);
memcpy(&vBuf[0*sizeof(Vertex)], &vertices[0], sizeof(struct MyVertex));
memcpy(&vBuf[1*sizeof(Vertex)], &vertices[1], sizeof(struct MyVertex));
memcpy(&vBuf[2*sizeof(Vertex)], &vertices[2], sizeof(struct MyVertex));
gBuf->Unlock();
```

Presenting the primitives

A two step action
a. Tell the system which vertex buffer to use
b. Tell the system what to do with the buffer

```c
IDirect3DDevice9::SetStreamSource(
  UNIT StreamNumber,
  IDirect3DVertexBuffer9 *pStreamData,
  UNIT OffsetInBytes,
  UNIT Stride);
```
Drawing the primitives

A two step action
a. Tell the system which vertex buffer to use
b. Tell the system what to do with the buffer

What to draw – points, triangles,…

The first vertex index

How many primitives

Example

Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(r, g, b), 1.0f, 0);
BeginScene(); // begin the 3D scene
SetStreamSource(0, g_pVB, 0, sizeof(MyVertex)); // which vtx buffer to use
SetVertexDeclaration(myVertexDecl);
DrawPrimitive(D3DPT_LINESTRIP, 0,3);
DrawPrimitive(D3DPT_TRIANGLEFAN, 2, 2);
EndScene(); // ends the 3D scene
Present(NULL, NULL, NULL, NULL); // displays the created frame on the screen
The index buffer

- Purpose: to reuse the vertices
- Idea: store the vertices once and access them many times

- Reuse the vertices
  - Single copy of data –
    - no redundancy
    - Error in data can be easily fixed
  - Saves space
  - Easy to manipulate

```c
IDirect3DIndexBuffer9* Indexbuf;
d3dDevice->CreateIndexBuffer(
    buffer size,       // how the buffer is used
    Usage,            // how the memory is managed
    format,           // indices format (e.g., 16-bit)
    &IndexBuf,        // return address of buffer
    0);               // reserved, not used
```

The index buffer

```c
IDirect3DIndexBuffer9* Indexbuf;
d3dDev->CreateIndexBuffer(
    30*sizeof(short),   // buffer size
    D3DUSAGE_WRITEONLY, // usage
    D3DFMT_INDEX16,     // 16-bit indices
    D3DPOOL_MANAGED,    // indicate buffer in video mem
    &IndexBuf,          // the buffer
    0);                 // reserved, not used
```
Index Buffer - Drawing Primitives

**Example**

```c
D3DDevice->SetStreamSource(0, vBuf, 0, sizeof(myVertex));
D3DDevice->SetIndices(ibuf);
D3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0, 8, 0, 12);
```

**Indexed Vertex Structure**

```c
v_buffer->Lock(0,0,(void*)&v, 0);
v[0] = VertexPos(-1.0f, -1.0f, -1.0f);
v[1] = VertexPos(-1.0f, 1.0f, -1.0f);
v[2] = VertexPos(1.0f, 1.0f, -1.0f);
v[3] = VertexPos(1.0f, -1.0f, -1.0f);
v[4] = VertexPos(-1.0f, -1.0f, 1.0f);
v[5] = VertexPos(-1.0f, 1.0f, 1.0f);
v[6] = VertexPos(1.0f, 1.0f, 1.0f);
v[7] = VertexPos(1.0f, -1.0f, 1.0f);
v_buffer->Unlock();
```

```c
Index_buffer->Lock(0, 0, (void**)&k, 0);
k[0] = 0; k[1] = 1; k[2] = 2; // Front face
k[15] = 4; k[16] = 1; k[17] = 0;
k[18] = 3; k[19] = 2; k[20] = 6; // Right face
k[21] = 3; k[22] = 6; k[23] = 7;
k[24] = 1; k[25] = 5; k[26] = 6; // Top face
k[27] = 1; k[28] = 6; k[29] = 2;
k[30] = 4; k[31] = 0; k[32] = 3; // Bottom face
k[33] = 4; k[34] = 3; k[35] = 7;
Index_buffer->Unlock();
```
Example- draw two triangles

```
gd3dDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0, 6, 50, 2);
```

Direct3D Transformation
Transformation Pipeline

Transformations maps one coordinate system into another coordinate system

Model Space (3D) ➔ Model Transformation ➔ (x, y, z) ➔ 3D object coordinates
Model Space (3D) ➔ Model Transformation ➔ 3D object coordinates
World Space (3D) ➔ World Transformation ➔ 3D world coordinates
World Space (3D) ➔ World Transformation ➔ 3D world coordinates
View Space (3D) ➔ View Transformation ➔ 3D camera coordinates
View Space (3D) ➔ View Transformation ➔ 3D camera coordinates
Camera Space (3D) ➔ Projection Transformation ➔ 2D view plane coordinates
Camera Space (3D) ➔ Projection Transformation ➔ 2D view plane coordinates
"Camera" Space (2D) ➔ View port Transformation ➔ 2D image coordinates
"Camera" Space (2D) ➔ View port Transformation ➔ 2D image coordinates
Image Space (2D) ➔ (x', y')

Transformation Matrices

- **Translation**
  
  D3DXMatrixTranslation(D3DXMATRIX *Out, dx, dy, dz);

- **Scaling**
  
  D3DXMatrixScaling(D3DXMATRIX *Out, sx, sy, sz);

- **Rotation**
  
  D3DXMatrixRotationX(D3DXMATRIX *Out, float angle);
  D3DXMatrixRotationY(D3DXMATRIX *Out, float angle);
  D3DXMatrixRotationZ(D3DXMATRIX *Out, float angle);
  D3DXMatrixRotationAxis(D3DXMATRIX *Out, D3DXVECTOR3 *v, float angle);
Setting up the transformations

- Telling Direct3D which matrix to use for each step in the transformation pipeline

```cpp
IDirect3DDevice9::SetTransform(
  D3DTRANSFORMSTATETYPE State,
  CONST D3DMATRIX *pMatrix);
```

D3D World Transformation

```cpp
d3dDev->BeginScene();

... 
D3DXMatrixTranslation(&matT, 10.0f, 10.0f, 5.0f);
D3DXMatrixRotationX(&matRX, D3DXToRadian(alpha));
D3DXMatrixRotationY(&matRY, D3DXToRadian(beta));
D3DXMatrixRotationZ(&matRZ, D3DXToRadian(gama));
D3DXMatrixScaling(&matScale, 1.5f, 1.5f, 1.5f);
worldMat = matRZ * matRY * matRX * matScale * matT

D3DTS_WORLD
```
D3D View Transformation

• Build a Left-Handed, Look-at Matrix

D3DXMatrixLookAtLH(
   D3DXMATRIX *Out,
   D3DXVECTOR3 *Eye,
   D3DXVECTOR3 *pAt,
   D3DXVECTOR3 *pUp);

• Setup

D3DTS_VIEW

D3DXMatrixLookAtLH(
   &matView,
   &D3DXVECTOR3(4.0f, 4.0f, -3.0f), // camera loc
   &D3DXVECTOR3(2.0f, 2.0f, 1.0f),  // look-at target
   &D3DXVECTOR3(0.0f, 1.0f, 0.0f));

d3dDev-&SetTransform(D3DTS_VIEW, &matView);

d3dDev-&BeginScene();
   .
   .
   .
   .
d3dDev-&DrawPrimitive(...);
   d3dDev-&EndScene();
   d3dDev-&Present(0, 0, 0, 0);

Doron Nussbaum  COMP 3501 Drawing Primitives  52
D3D Perspective Projection Transformation

- Build a Left-Handed, Look-at Matrix

\[
\text{D3DXMatrixPerspectiveFovLH}(\text{Out}, \text{fo}, \text{Aspect}, \text{near}, \text{far});
\]

- Setup

\[
\text{IDirect3DDevice9::SetTransform}(\text{State}, \text{pMatrix});
\]

D3DXPerspectiveProjectionFovLH(\text{matProjection}, 45, \text{Width/Height}, 1.0f, 1000.0f);

\begin{align*}
\text{gd3dDev->SetTransform(D3DTS_PROJECTION, } & \text{&matProjection);} \\
\text{d3dDev->DrawPrimitive(…); } & \\
\text{d3dDev->EndScene(); } & \\
\text{d3dDev->Present(0, 0, 0, 0); }
\end{align*}

Note that aspect ratio changes when the window is resized
D3D Orthographic Projection Transformation

• Build a Left-Handed, Look-at Matrix

\[
D3DXMATRIX * D3DXMatrixOrthoLH( \\
D3DXMATRIX *pOut, \\
float width, \\
float height, \\
float near, \\
float far );
\]

• Setup

IDirect3DDevice9: :SetTransform( \\
D3DTRANSFORMSTATETYPE State \\
CONST D3D MATRIX *p Matrix );

D3D Projection Transformation

```c
D3DProjectionTransformation

d3dDev->BeginScene();
.
.
.
D3DXMatrixOrthoLH(&matProjection, // out matrix
800, // width
600, // height
1.0f, 1000.0f); // near and far planes
gd3dDev->SetTransform(D3DTS_PROJECTION, &matProjection);
.
.
d3dDev->DrawPrimitive(...);
d3dDev->EndScene();
d3dDev->Present(0, 0, 0, 0);
```
View Port

- Sets the screen coordinates to be used
  - Convert from projection coordinates to screen coordinates
- By default the view port size is the back buffer
- Can be used to divide the screen into a number of sections
  - Each section of the screen is being drawn independently
  - Divide the screen into quadrants each with a different display

D3D View Port Transformation

```c
typedef struct D3DVIEWPORT9 {
    DWORD X;  // upper left corner of the view port on the rendered surface
    DWORD Y;  // upper left corner of the view port on the rendered surface
    DWORD Width;  // width of the view port
    DWORD Height;  // height of the view port
    float MinZ;  // minimum z value (of z buffer) normally 0
    float MaxZ;  // maximum z value (of z buffer) normally 1
} D3DVIEWPORT9, *LPD3DVIEWPORT9;

D3DVIEWPORT9 vp={50,50,300,300,0,1};
md3dDev->SetViewport(&vp); d3dDev->BeginScene();
```