Computer Graphics

(Managed DirectX 9.0c - Tutorial)

Lecture 011

The Direct3D Device

- The root of all drawing in Direct3D is the device class. This is analogous to the actual graphics device (video card) in your computer.
- We initialize graphics by initializing the device, i.e. by using one of a number of device constructors.
- The parameters to the constructor include:
  - Adapter number: 0 is always the default
  - DeviceType: tells DirectX what type of device you want to create. For example a hardware device would indicate to DirectX that all processing will be performed by the card.
  - Forms.Control: Indicates to which window to bind to the device. Usually 'this' is used here.
  - Direct3D.CreateFlags: used to control aspects of the device's behaviour after creation. Ex. SoftwareVertexProcessing
  - Direct3D.PresentParameters: used to control how the device presents its data to the screen. Ex. Windowed mode

The basic loop !!

- After you have initialized your device you need to tell your DirectX what and when to draw to the device.

- The following is the basic drawing loop:
  - Clear the back buffer
    - (using Device.Clear(ClearFlage.Target, Color.White, ZBuffer 1.0f, StencilBuffer 0))
  - Ready Direct3D to begin drawing
    - (using Device.BeginScene())
  - Draw the scene
    - (next slide !!)
  - Indicate to Direct3D that we're done drawing
    - (using Device.EndScene())
  - Copy the back-buffer to the display
    - (using Device.Present())

The VertexBuffer and CustomVertex Classes

- A VertexBuffer is simply a vector that stores vertices.

- CustomVertex instances define the vertices. Note that there are a number of different vertex types. In general each instance will have an x,y,z coordinate + colour.

```
vb = new VertexBuffer(typeof(CustomVertex.PositionNormalTextured), 36, device, Usage.Dynamic
                                        | Usage.WriteOnly,
     CustomVertex.PositionNormalTextured.Format, Pool.Default);

CustomVertex.PositionNormalTextured[] verts = new CustomVertex.PositionNormalTextured[36];

// Front face
verts[0] = new CustomVertex.PositionNormalTextured(-1.0f, 1.0f, 1.0f, 0f, 0f, -1f, 0.0f, 0.0f);
verts[1] = new CustomVertex.PositionNormalTextured(-1.0f, -1.0f, 1.0f, 0f, 0f, -1f, 0.0f, 1.0f);
verts[2] = new CustomVertex.PositionNormalTextured(1.0f, 1.0f, 1.0f, 0f, 0f, -1f,1.0f, 0.0f);
verts[3] = new CustomVertex.PositionNormalTextured(-1.0f, -1.0f, 1.0f, 0f, 0f, -1f,0.0f, 1.0f);
verts[4] = new CustomVertex.PositionNormalTextured(1.0f, -1.0f, 1.0f, 0f, 0f, -1f,1.0f, 1.0f);
verts[5] = new CustomVertex.PositionNormalTextured(1.0f, 1.0f, 1.0f, 0f, 0f, -1f,1.0f, 0.0f);
```

Rendering to the screen

- First set the stream source i.e. inform DirectX of which vertices you are going to work with.
  - Use device.SetStreamSource(0,vb,0)
- Note that you might have more than one vertex buffer. Actually this is normally the case. For example you can have one vertex buffer for each object in the scene.
- Once you set the stream … you draw the primitives using
  - device.SetTexture(0, t7);
  - device.DrawPrimitives(PrimitiveType.TriangleList, 0, 2);
  - device.SetTexture(0, inside);
  - device.DrawPrimitives(PrimitiveType.TriangleList, 6, 4);
- At this point you may also want to transform some of the coordinates using the device.Transform.World matrix. Example:
- device.Transform.World = device.Transform.World *
  Matrix.RotationYawPitchRoll(_angle, _angle, _angle);

Setting up a camera …

- Recall that the viewing-reference coordinate system (VRC) is defined in terms of a view reference point (VRP) which represents the origin, a view-plane normal vector (VPN) and a view-up vector (VUP).

- ```
  private void SetupCamera() {
      device.Transform.Projection = Matrix.PerspectiveFovLH((float)Math.PI / 4,
      this.Width / this.Height, 1.0f, 150.0f);
      device.Transform.View = Matrix.LookAtLH(new Vector3(0,0, 20.0f), new
      Vector3(), new Vector3(0,1,0));
  }
  ```

- The projection transform gives us information about which projection is being applied. The view transform gives us information about the camera.

- The parameters to the LookAtLH method indicate
  - The camera position in world space
  - The position in the world space we want the camera to look at
  - The last argument is the direction that will be considered ‘up’

- You can clearly move the camera using this method by modifying the first parameter before every frame is drawn.
  ```

The Matrix Class (and 3D Transformations)

- All the 3D transformations discussed earlier in class can be carried out with the Matrix class. The main transformations are rotation and translation.

- To rotate you would write down:
  - device.Transform.World = device.Transform.World *

    Matrix.RotationAxis (rotationAxis, angle);

- To translate you would write down:
  - private Matrix cube1 = Matrix.Translation(-2.0f,-2.0f,2.0f);

Lighting

- If there are no lights in the scene then all objects are black !!

- We therefore have to introduce some light in the scene. We get four different sorts of light in Direct3D
  - Point lights - Point lights emanate light equally in all directions from a particular point in space.
  - Directional lights - don't emanate from a particular point in space. Rather, they are assumed to be located an infinitely far distance away
  - Spot lights – Like point lights but are constrained to point in a particular direction
  - Ambient lights – DirectX models ambient light as a light with no source that illuminates everything evenly. Simplest light to use.

- protected void SetupLights() {

```
    // Use a white, directional light coming from over our right shoulder device.
    Lights[0].Diffuse = Color.White;
    device.Lights[0].Type = LightType.Directional;
    device.Lights[0].Direction = new Vector3(-3, -1, 3);
    device.Lights[0].Update();
    device.Lights[0].Enabled = true;

    // Add a little ambient light to the scene
    device.RenderState.Ambient = Color.FromArgb(0x40, 0x40, 0x40);
}
```

Textures

- A texture is simply an image – say from a BMP file – that is “glued” or “tacked” onto a triangle or series of triangles. We need to introduce a texture coordinate system in order to be able to discuss textures. These 2D coordinates are usually referred to as u and v.

- The PositionNormalTextured class is used were for each vertex we have to specify
  - 3D coordinates
  - Normal Vector
  - Texture coordinates

- (u,v) = (0,0) indicates the upper left corner of the image
  = (1,0) indicates the upper right corner of the image
  = (0,1) indicates the lower left corner of the image
  = (1,1) indicates the lower right corner of the image

- Texture t = TextureLoader.FromFile(device, "texture.bmp");
- Before drawing the primitives change the texture by calling
  - device.SetTexture(0, texture);

Z-Buffer

- We don't have to bother about the order we draw our shapes in the scene, because we can use the implementation of Z-Buffer in DirectX to calculate which surfaces are visible from a particular point of projection.

- // Tell Direct3D to create a ZBuffer for us
  pres.EnableAutoDepthStencil = true;
  pres.AutoDepthStencilFormat = DepthFormat.D16;

- The first line actually informs DirectX that we want to use a z-buffer. The second line indicates what format we want to use. In this case we'll be using a 16-bit buffer format. This is usually enough.

- Note however that now for every time we draw a frame we also need to clear the z-buffer and not only the display to a particular colour. We do this by modifying slightly our call to the device.clear() method.

- device.Clear(ClearFlags.Target | ClearFlags.ZBuffer, Color.Black, 1.0F, 0);

Meshes

- A mesh is simply a predefined set of vertices and a set of textures and materials that go with them.
- So instead of (directly) using the VertexBuffers and DrawPrimitives() we use a higher level of abstraction. We will use the Mesh class.
- Every mesh is split into a number of subsets. Each subset contains the vertices which use the same texture and material. Recall that texture and material (ex. Specular, matt, etc) are global properties of the device. In Direct3D the different subsets are loaded separately. Here's how:
  - ExtendedMaterial[] exMaterials; //will store the textures and materials which each subset has applied to it.
  - mesh = Mesh.FromFile(path, MeshFlags.SystemMemory, device, out exMaterials); // loads the mesh (.x file)
  - We then populate the Material[] and Texture[] arrays from exMaterials
  - Finally loop through the Texture[] array and render the vertices using Mesh.DrawSubset(i)