Course 3D_MDX: 3D-Graphics with Managed DirectX 9.0
Chapter C4: Comments to Standard Meshes = Primitives

---

namespaces

`using System;` //Home of the base class of all classes "`System.Object`" and of all primitive data types such as `Int32, Int16, double, string`.
`using System.Windows.Forms;` //Home of the "`Form`" class (base class of `Form1`) and its method `Application.Run`.
`using System.Drawing;` //Home of the "`Graphics`" class and its drawing methods such as `DrawStirng, DrawLine, DrawRectangle, FillClosedCurve` etc.
`using Microsoft.DirectX;` //Utilities including exception handling, simple helper methods, structures for matrix, clipping, and vector manipulation.
`using Microsoft.DirectX.Direct3D;` //Graphics application programming interface (API) with models of 3-D objects and hardware acceleration.
For DirectX see: **http://msdn.microsoft.com/library/default.asp** → Win32 and COM Development → Graphics and Multimedia → DirectX → SDK Documentation → DirectX SDK Managed → DirectX SDK → Namespaces.

Entry to start our .NET Windows program: `public class Form1 : Form`

//We derive our window `Form1` from the class `Form`, which is contained in the `System.Windows.Forms` namespace.

`static void Main() { Application.Run( new Form1() ); }` //Create an instance of `Form1` and ask the operating system to start it as main window of our program.

`static Device device = null;` //The global device object must be static since we need it inside the static Timer event handler.

`static float xAngle, yAngle, zAngle;` //Global movements around the main 3 axes.

`static Mesh meshPolygon, meshBox, meshSphere, meshTorus, meshCylinder, meshTeapot, meshText;` //declarations of the mesh identifiers
//The following lines give each 3D-object a specific position on stage.
These x,y,z-shiftings are expressed as 3D-vectors and the vectors are transposed into a general 4x4 matrix-format. The resulting matrices will be used in the rendering loop (see `OnTimer(...)`-function below) where they will be multiplied at run time with the current rotation matrix `m`.
`static Matrix mPol = Matrix.Translation( new Vector3(    0f,    0f, -1.5f ) ); //polygon` //in front of all others
`static Matrix mBox = Matrix.Translation( new Vector3(  1.5f,    0f,    0f ) ); //box` //right side
`static Matrix mSph = Matrix.Translation( new Vector3( -1.5f,    0f,    0f ) ); //sphere` //left side
`static Matrix mTor = Matrix.Translation( new Vector3(    0f,  1.5f,    0f ) ); //torus` //top side
`static Matrix mCyl = Matrix.Translation( new Vector3(    0f, -1.5f,    0f ) ); //cylinder` //bottom side
`static Matrix mTea = Matrix.Translation( new Vector3(    0f,    0f,  1.5f ) ); //teapot` //in the mid but behind the polygon
`static Matrix mTex; //text` //The 3D-text has no specific position.

`Timer myTimer = new Timer();` //This Timer sends messages at fixed time intervals to `Form1`, that trigger `Form1` to execute its `OnTimer(..)` method.

Constructor `public Form1()` inside `public class Form1`

`Text = "Primitive Meshes";` //Title in the blue title bar of `Form1`.

`myTimer.Tick += new EventHandler( OnTimer );` //Obligatory definition of an event handler for the Timer event.
`myTimer.Interval = 1;` //1 millisecond intervals means: as fast as possible. The operating system will raise as many events as possible (normally 1000[msec] divided by monitor refresh[≈80Hz] ≈ 13 msec).

`ClientSize = new Size( 400, 300 ); //Calls OnResize( ... )` //This statement raises an `OnResize(...)` event which leads to the first time initialization of a DirectX-`Device`.

Overridden event handler `protected override void OnResize( System.EventArgs e )` inside `public class`

```
Form1
//Whenever the window changes we have to initialize Direct3D from scratch.
```

`myTimer.Stop();` //Stop the timer during initialization. It may disturb DirectX-initialization.

`try` //All the following things crash when DirectX is not properly installed. In this case the `try-catch` clause offers a civilized exit.

```
//Get information from the operating system about its current graphics properties.
PresentParameters presentParams = new PresentParameters(); //This structure is an obligatory parameter for creating a new Device. It carries several flags such as Windowed = true; and SwapEffect.Discard; = status flags controlling the behavior of the Device.
//we have to set four flags
presentParams.Windowed = true; //We want a program in a window not a full screen program.
presentParams.SwapEffect = SwapEffect.Discard; //This flag tells the graphic board how to handle the backbuffer(s) after front-back flipping. Many graphic boards need this flag, but I do not really know why. See:
```
http://msdn.microsoft.com/library/.../D3DSWAPEFFECT.asp
```
presentParams.EnableAutoDepthStencil = true; //with depth buffer //We want a Z-buffer on the graphics board.
presentParams.AutoDepthStencilFormat = DepthFormat.D16; //16 bit depth //Z-buffer just needs limited resolution (short integers). Other possible formats see:
```
http://msdn.microsoft.com/archive

```
//Create a new D3D-device that serves as canvas.
if ( device != null ) device.Dispose(); //Free the old canvas if any.
device = new Device( 0, DeviceType.Hardware, this, CreateFlags.SoftwareVertexProcessing,
presentParams );
//1. parameter = 0 = default device. (The computer can have different devices f.i. two graphic boards.)
//2. parameter = DeviceType.Hardware allows rasterization by the graphic board (HAL=first choice), software (HEL) or mixed.
//3. parameter = this Pointer to our Form1-Control being the target of any graphical output.
//4. parameter = CreateFlags.SoftwareVertexProcessing is a flag that switches off the vector graphics part of the graphic board to avoid any risk from old graphic boards and/or old DirectX-drivers = all vector graphics via HEL. Disadvantage: Waste of the powerful HAL vector pipelines of a modern graphic board.
//5. parameter = presentParams is a structure of status flags describing the behavior of a graphic board.
//see:
```
../../Lectures/L05_OpenGL_DirectX

```
//The following if-statements do not matter in first time initialization but they are necessary in any later OnResize-event, when the old Device does not exist anymore. Being connected with a dead Device the old meshes are useless and must be thrown away in order to free their memory.
if ( meshPolygon  != null ) meshPolygon .Dispose(); //free the old mesh if any
if ( meshBox      != null ) meshBox     .Dispose(); //free the old mesh if any
if ( meshSphere   != null ) meshSphere  .Dispose(); //free the old mesh if any
if ( meshTorus    != null ) meshTorus   .Dispose(); //free the old mesh if any
if ( meshCylinder != null ) meshCylinder.Dispose(); //free the old mesh if any
if ( meshTeapot   != null ) meshTeapot  .Dispose(); //free the old mesh if any
if ( meshText     != null ) meshText    .Dispose(); //free the old mesh if any
//Any mesh (except Teapot) needs parameters of relative lengths, sizes, etc..
meshPolygon  = Mesh.Polygon ( device, 0.3f, 8 ); //line length + no of vertices
0.3f = lengths of edges of a regular polygon, 8 = no. of vertices
meshBox      = Mesh.Box ( device, 0.5f, 0.5f, 0.5f ); //xSize, ySize, zSize
0.5f = length, width, height of a rectangular solid
meshSphere   = Mesh.Sphere ( device, 0.5f, 20, 20 ); //radius, no slices, no stacks
0.5f = radius, 20 = no. of latitudes, 20 = no. of longitudes
meshTorus    = Mesh.Torus ( device, 0.2f, 0.4f, 20, 20 );//in+out radii, slices+stacks
0.2f = inner radius, 0.5f = outer radius, 20 = no. of latitudes, 20 = no. of longitudes
meshCylinder = Mesh.Cylinder( device, 0.5f, 0.2f, 0.8f, 20, 20 ); //front+back radii,
length, slices+stacks
0.5f = bottom radius, 0.2f = top radius, 0.8f = height, 20 = no. of latitudes, 20 = no. of longitudes
meshTeapot   = Mesh.Teapot ( device );
```

```
String text = "Size: " + ClientSize.Width.ToString() + "/" + ClientSize.Height.ToString();
```
//String = "`Size: 400/300`". The figures 400 and 300 change after a `OnResize`-event.

`GlyphMetricsFloat[] gly = new GlyphMetricsFloat[text.Length];` //Array of character size descriptions.
See: **http://windowssdk.msdn.microsoft.com/library/default.asp?url=/library/en-us/OpenGL/ntopnglr_6i44.asp**
See: **http://pluralsight.com/wiki/default.aspx/Craig.DirectX/FontBasicsTutorial.html**

```
meshText = Mesh.TextFromFont( device, new System.Drawing.Font( FontFamily.GenericSerif, 12 ), text, 0.01f, 0.25f, out gly ); //string, smooth, thick, per char info
```
`0.01f` = how detailed, `0.25f` = how thick
See: **http://pluralsight.com/wiki/default.aspx/Craig.DirectX/FontBasicsTutorial.html**

//Loop over each element in the `GlyphMetricsFloat` array, accumulating data as it goes, to compute a length and width for the Mesh. Each `GlyphMetricsFloat` object contains data about the horizontal distance from the previous letter to this one (`CellIncX`), and about the height of each letter (`BlackBoxY`). By simply summing the former and finding the max of the latter, we compute the overall height and width of the text object.

```
float x=0f, y=0f; //width and height of text
for ( int i=0; i < text.Length; i++ ) //for all chars
{ x += gly[i].CellIncX; //x=text.Width = sum of char.Width //sum up all widths
  if ( gly[i].BlackBoxY > y ) y = gly[i].BlackBoxY; //y=text.Height= max of char.Height //find out the heightest character
} //end of for
```
See: **http://pluralsight.com/wiki/default.aspx/Craig.DirectX/FontBasicsTutorial.html**

```
mTex = Matrix.Translation( new Vector3( -x/2f, -y/2f, 2f ) );
```
//Calculate a translation matrix based on the bounds we figured out earlier. The reason is that we' d like to rotate the text around its center. Because the origin of the `mTex`-mesh is at the beginning of the first character, rotating without primary translation will make it spin around one corner of the text...not the effect we want.

```
//set up material with diffuse and ambient white color
Material myMaterial = new Material();
myMaterial.Diffuse = myMaterial.Ambient = Color.White;
```
//Since all material properties are white, the meshes will reflect any sort of light.

```
device.Material = mtrl;
```
//Copy the material properties to the device.

```
//turn on some blue directional light coaxial to the camera
device.Lights[0].Type = LightType.Directional;
```
//See: **http://msdn.microsoft.com/archive**

```
device.Lights[0].Diffuse = Color.DarkBlue;
device.Lights[0].Direction = new Vector3( 0, 0, 5 );
```
//Light source = centrally in front of the monitor = roughly from the spectators nose.
Recommended experiments: Change to upper left = `-1,1,5`; to lower left = `-1,-1,5`; to backside = `1,1,-5` etc.

```
device.Lights[0].Enabled = true; //We have to set the D3DRS_LIGHTING renderstate to enable lighting.
```

```
//set up the transformation of world coordinates into camera or view space
device.Transform.View = Matrix.LookAtLH(
  new Vector3( 0f, 0f,-5f ),  // eye point 5.0 in front of the canvas
  new Vector3( 0f, 0f, 0f ),  // camera looks at point 0,0,0
  new Vector3( 0f, 1f, 0f ) );  // worlds up direction is the y-axis.
```
See: **http://msdn.microsoft.com/archive**

```
//set up the projection transformation using 4 parameters:
//1.: field of view = 45 degrees; 2.: aspect ratio = height / width = 1 = square window;
//3.: near clipping distance = 0; 4.: far clipping distance = 10;
device.Transform.Projection = Matrix.PerspectiveFovLH((float)Math.PI/4, 1f, 1f, 100f );.
```
//Describe the truncated viewing pyramid = frustum:
1. is the viewing angle in radians (PI/4=45^{o}),
2. is the ratio height / width,
3. is the z-value of the front plane of the viewing volume and
4. the z-value of its back plane.

//See: **http://msdn.microsoft.com/archive**
//See: **www.lighthouse3d.com/opengl/viewfrustum/** Please mail me if this link is dead.
Experiment 1: Enlarge `Math.PI/4` to `Math.PI/2 = 90`^{o}. The scene will appear shifted away.
Experiment 2: Distort the ratio to a) 0.5 and b) to 2.0.
Experiment 3: Shift the front plane away from You towards the cylinder in steps of 0.5.

```
device.RenderState.CullMode = Cull.None;
```
//Culling is a method to accelerate rendering by excluding (mostly back-) surfaces from the render process.

```
device.RenderState.Lighting = true;
```
//Switch on the directional and the ambient light.

```
xAngle = yAngle = zAngle = 0; //start angles
```

```
myTimer.Start(); //start the timer again
```
//It has been stopped by the first statement of this function

---

```
catch (DirectXException) { MessageBox.Show( "Could not initialize Direct3D." ); return; }
```
//Emergency exit when DirectX 9.0 was not found and/or `new Device` crashed. End of the `try`-clause = 2nd statement of this function.

Event handler `protected static void OnTimer( Object myObject, EventArgs myEventArgs )` inside `public class Form1`

`if (device == null) return;` //Emergency exit if the DirectX initialization has gone wrong.

```
//throw the old image away
device.Clear( ClearFlags.Target | ClearFlags.ZBuffer, Color.Gray, 1f, 0 );
```
//Erase any former content from the canvas and the Z-buffer.
Recommended experiment: Kick out this `Clear`-statement and observe what happens.

`//rotate with 3 angular velocities` //The meshes rotate around three main axes with the same velocity of 0.02/step ≈ 3.6°/step.

```
Matrix m = Matrix.RotationYawPitchRoll( yAngle += 0.02f, xAngle += 0.02f, zAngle += 0.02f
); //
```

```
//draw on the canvas
device.BeginScene(); //Open the render clause
  device.Transform.World = m * mPol; meshPolygon .DrawSubset( 0 ); //rotate + translate
  device.Transform.World = m * mBox; meshBox     .DrawSubset( 0 ); //rotate + translate
  device.Transform.World = m * mSph; meshSphere  .DrawSubset( 0 ); //rotate + translate
  device.Transform.World = m * mTor; meshTorus   .DrawSubset( 0 ); //rotate + translate
  device.Transform.World = m * mCyl; meshCylinder.DrawSubset( 0 ); //rotate + translate
  device.Transform.World = m * mTea; meshTeapot  .DrawSubset( 0 ); //rotate + translate
  device.Transform.World = mTex * m; meshText    .DrawSubset( 0 ); //translate + rotate
device.EndScene(); //Close the render clause
device.Present(); //show the canvas // = Command to flip the front and the back buffer of the graphic board.
```
