翻译：clayman
Clayman_joe@yahoo.com.cn

第一章 Direct3D 入门

创建设备

Device 类是 DirectX 里的所有绘图操作所必须的。可以把这个类假想为真实的图形卡。场景里所有图形对象都依赖于 device。一台计算机里可以有一个到几个 device，在 Mnaged DirctX3D 里，你可以控制任意多个 device。

Device 共有三个构造函数，我们现在只讨论其中的一个，但会在后边的内容里讨论其他的。先来看看具有如下函数签名的构造函数：

```
public Device(int adapter,DeviceType deviceType,Control renderWindow,CreateFlags behaviorFlags, PresentParameters[] presentationParameters);
```

（构造函数的第二种重载类似于上边这个，但它接受来自非托管（或者非 windows form）的窗口句柄作为 renderWindow。而只接受一个 IntPtr 参数的重载是非托管 com 组建指向 Idirect3Ddevice9 的接口。当你的代码需要和非托管的程序协作时则应用它）

好了，这些参数是什么意思，以及我们怎样来使用呢？呵呵，参数 adapter 表示我们将要使用哪个物理图形卡。计算机里的所有图形卡都有一个唯一的适配器标识符（通常是 0 到你的图形卡数量－1），默认的显卡总是标识为 0 的图形卡。

下一个参数，DeviceType，告诉了 DirectX3D 你要创建哪种类型的 device。这里最常用的值是 DeviceType.Hardware，表示你将创建一个硬件设备。另一个选项 DeviceType.Reference，这种设备允许你使用“参考光栅器”（reference rasterizer），所有的效果由 DirectX3D 运行时来实现，以很慢、很慢、很慢的速度运行^_^。应该仅在调试或测试你的显卡不支持的特性时使用这个选项。

（注意参考光栅器只包含在 DirectX SDK 里，so DirectX 运行时是不能使用这个特性的。最后一个为 DeviceType.Software 的值允许使用用户自定义的软件光栅器（custom software rasterizer）在不确定是否有这样一个光栅器存在时，忽略这个选项吧^_^。）

rendrWindow 表示把设备绑定到的窗口。因为 windows form 控件类都包含了一个窗口句柄（windows handle），所以很容易把一个确定的类作为渲染窗口。可以使用 form、panel 或其他任意的控件作为这个参数的值。但现在，我们只用 form。

下一个参数用来描述设备创建之后的行为。大部分 CreateFlags 枚举的成员都能组合起来使用，使设备具有多种行为。但有一些 flag 是相互排斥的，稍后讨论它们。我们现在只使用 SoftwareVertexProcessing 标志。这个标志适合于所有顶点处理都用 CPU 计算的情况。因此，这自然比所有点都用 GPU 处理要慢，因为我们不确定你的显卡是否支持所有特性。So，安全第一，假设你的 CPU 能完成现在的任务。

最后一个参数，它表示你的设备把数据呈现到显示器的方式。Presentation Parameter 类的外观都可以由这个类来控制。我们过后再来深入讨论它的构造函数，现在，我们只关心“Windowed”成员和“SwapEffect”成员。

Windowed 成员是一个布尔类型的值，决定设备是全屏还是窗口模式。

SwapEffect 成员用于控制缓存交换的行为。如果选择了 SwapEffect.Flip，运行时会创建额外的后备缓冲（back buffer），并且在显示时拷贝 front buffer。SwapEffect.Copy 与 Flip 相似，但要求你把后备缓冲设为 1。我们将要选择的 SwaoEffect.Discard，如果缓冲没有准备好被显示，则会丢弃缓冲中的内容（which simply discards the contents of the buffer if it isn’t ready to be presented）。

学了这么多，现在来创建一个设备吧。回到代码上，首先为我们的程序创建一个 device 对象：

（代码略，参见 DirectX sdk Tutorial 1: Create a Device）

现在让我们来重写 Paint（）函数：

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
    device.Clear(ClearFlags.Target, System.Drawing.Color.Blue, 1.0f, 0);
    device.Present();
}
```

我们使用Clear（）方法把窗口填充为实心的颜色。它的第一个参数指定了我们要填充的对象；在例子里，我们填充的即是目

标窗口。稍后再来讨论ClearFlags枚举的其它成员。第二个参数是我们所要填充的颜色。其他的两个参数先暂时忽略。在device被填充之后，我们必须更新显示：Present方法会为我们完成这个任务。这个方法也有几个重载的类型；上边使用的方法会显示device的整个区域。同样稍后再讨论。

看的有些枯燥了吗，好吧，现在来真正绘制一些图形

三维图形世界里最基本的图形就是三角形。使用足够的三角，我们可以呈现出任何东西，甚是是平滑的曲面。没有什么比画一个简单的三角形更好的了。为了使过程尽可能的简单，我们先避开“world space”以及各种变换（当然，我们马上就会提到他们），使用屏幕坐标来绘制一个简单的三角。再绘制我们迷人的三角前，我们必须做2件事。1，需要一些数据结构来保存三角的信息。2，告诉device来绘制它。

很幸运，DirectX已经有这样一个数据结构来保存三角了。Direct3D名称空间里叫做CustomVertex的类可以用来储存大多数Direct3D中用到的“顶点格式”数据结构（vertex format）。

一个顶点格式结构把数据保存为一种DirectX3D认识并可以使用的格式。我们将讨论很多这种结构，但先看看我们即将用来创建三角的TransformedColored结构。这个结构告诉DirectX3D运行时我们的三角不需要进行坐标变换（比如旋转或移动），因为我们已经指定了使用屏幕坐标系。它也包含了每一个点（顶点）的颜色的信息。回到重写的OnPaint方法添加如下代码：

```
CustomVertex.TransformedColored[] verts = new CustomVertex.TransformedColored[3];
Verts[0].SetPosition(new Vector4(this.Width/2.0f,50.0f,0.5f,1.0f);
Verts[0].Color = System.Drawing.Color.Aqua.ToArgb();
Verts[1]`````````
Verts[2]`````````
```

（参见DirectX sdk Tutorial 2: Rendering Vertices）

数组里的每一个元素表示三角的一个顶点，所以我们创建了3个元素。然后使用新创建的Vector4结构为每一个成员调用SetPositin方法。变换过的顶点坐标包含了在屏幕上x和y的坐标（相对于屏幕的（0，0）点而言），当然也包括z坐标和rhw成员（reciprocal of homogenous w三维齐次坐标）。先忽略后边两个参数。Vector4结构（注：Vector4其实就是（x,y,z,w）经过变换后成为（x/w,y/w,z/w））是保存这种信息最方便的方式。然后我们设置了点的颜色。注意，我们使用了标准颜色的ToArgb方法。DirectX3D假设所接收的颜色为32位int。

既然有了数据就可以告诉DirectX我们需要绘制这个三角形，并且绘制它。在重写的OnPaint里添加如下代码

```
device.BeginScene();
device.VertexFormat = CustomVertex.TransformedColored.Format;
device. DrawUserPrimitives (PrimitiveType.TriangleList,1， verts);
device.EndScene();
```

好了，这几行代码是什么意思呢？其实很简单。BeginScene方法告诉DirectX3D我们即将绘制一些东西，为绘制做好准备。现在已经告诉了DirectX3D要绘制一些东西，接下来就必须告诉它画什么。这就是VertexFormat属性的作用。它决定了DirectX3D运行时使用哪种“固定功能管道”（fixed function pipline）格式。在我们的例子里使用变换过的，着色过的顶点管道。

不用担心你现在不明白确定的功能管道是什么意思，我们会很快来讨论它。

DrawUserPrimitives函数是真正发生绘图的地方。So，他的参数是什么意思呢？第一个参数是我们要绘制的初等几何体的类型。有很多种可用的类型，but now，我们只是画一系列的三角形。所以选择了PrimitiveType.TriangleList类型。第二个参数是我们要绘制的三角形的数量。对于一个三角形的集合来说，这个值应该是你的顶点数量除以3。我们只画一个三角，所以设为1。最后一个参数则是DirectX3D用来绘图的数据。最后一个EndSence方法通知DirectX3D我们不再绘图了。你必须再每次调用BeginSence之后都调用这个方法。

如果现在编译运行程序，你会发现移动或重置窗口大小之后，并不会更新显示。原因是当我们需要重绘整个窗口时，Windows并不会每一次都计算窗口的收缩情况。因此，你只是移除了显示过的数据，但并没有删除已经显示的内容。很幸运，有个简单的方法解决这个问题，我们可以告诉Windows窗口总是需要被整个的重绘。在OnPaint的最后加上一下代码：

```
this.Invalidate();
```

呵呵，再试试看，哦，看起来我们破坏了程序！现在只能显示一片空白了，并且我们的三角还在不停的闪烁，尤其是当调整窗口大小时。我们都干了些什么呢？原来“聪明”的Windows总是尝试在Invalidate()方法后来绘制当前的窗口（即空白的这个窗口）。

在我们的OnPaint方法之外还存在其他的绘制过程！能容易的通过改变窗口的“style”属性来解决。在构造函数里加上如下代码

```
this.SetStyle(ControlStyles.AllPaintingInWmPaint | ConstolStyles.Opaque, true);
```

哦～，好了，终于erying works as expected。我们所做的就是告诉Windows一切绘图过程都在OnPaint里完成。

三维化三角形

再看看我们的程序，它看起来并不那么“三维”。而且我们所做的都能用GDI＋轻易完成。So，我们应该怎样在3维空间里绘图，并且给人留下深刻的印象呢？实际上，简单的修改就能达到这样的效果。

如果你还记得，先前在我们创建第一个三角形的时候，我们使用了一个叫做“经过变换的”（transformed）坐标系统。这种坐标是显示器的屏幕区所使用的坐标，也是最容易定义的。如果我们使用未变换过的坐标系统会怎样呢？实际上，未变换过的坐标系统被广泛的用于现代游戏场景。

与屏幕坐标（screem space）相比我们定义这些坐标时，还应在世界坐标（world space）里定义每一个顶点。你可以把世界坐标设想为一个无限大的三维笛卡儿坐标。可以把你的对象放到这个“世界”的任意位置。现在来修改我们的程序，绘制一个未经过世界坐标变换的三角形。

首先使用未变换顶点格式类型中的一种来改变三角形的数据。在这里我们只关心顶点的位置，以及颜色，因此使用CustomVertex.PositionColored。

```
CustomVertex.positionColored[] verts = new CustomVertex. positionColored[3];
Verts[0].SetPosition(new Vector3(0.0f,1.0f,1.0f));
Verts[0].Color = System.Drawing.Color.Aqua.ToArgb();
Verts[1].SetPosition(new Vector3(-1.0f, -1.0f, 1.0f));
Verts[1].Color = System.Drawing.Color.Black.ToArgb();
Verts[2].SetPosition(new Vector3(1.0f, -1.0f, 1.0f));
Verts[2].Color = System.Drawing.Color.Purple.ToArgb();
```

（参见DirectX sdk Tutorial 3: Using Matrices）

同样改变VertexFormat属性：

```
device.VertexFormat = CustomVertex.PositionColored.Format;
```

好了，现在运行程序：什么也没有发生，仅获得一个填充过的窗口。在讨论为什么之前，来看看我们作了些什么。如你看到的，我们选择了PositonColored结构来保存数据。这个结构用世界坐标保存了顶点的位置，也保存了它的颜色。因为为顶点是没有变换过的，所以我们使用Vector3类来代替Vector4类，没有变换过的顶点是没有rhw值的。Vector3结构的成员直接映射为世界坐标系里x,y,z的值。同时，我们需要确定DirectX3D知道所做的改变，所以我们通过更新VertexFormat属性来让固定功能管道使用新的未变换但填充过颜色的顶点。

So，为什么程序运行时没有正确的显示呢？问题在于，我们只是在世界坐标里绘图，但并没有给DirectX3D任何关于如何来显示它们的信息。我们需要为场景添加一个摄像机来确定如何观看我们的顶点。在经过变换的坐标系统里不需要摄像机的原因是：DirectX3D已经知道在屏幕的哪个位置来显示顶点。

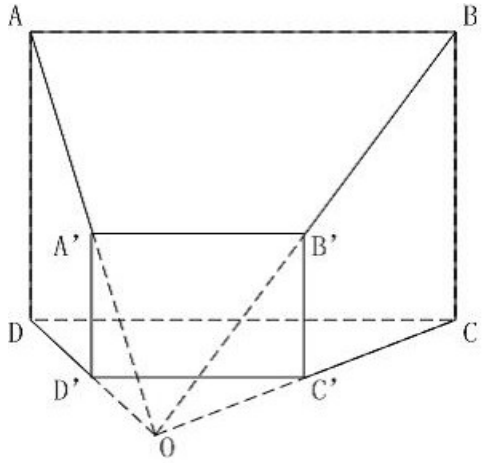
在device上通过两个不同的变换来控制摄像机。每一种变换都被定义为一个4×4的矩阵传递给DirectX3D。（each transform is defined as a 4*4 matrix that you can pass in to DirectX3D）

投影变换定义了场景被怎样投影到显示器。最简单的产生投影矩阵的方法就是使用Matrix类的PerspectiveFovLH方法。它将会使用左手坐标系创建一个正对场景的透视投影变换。（关于左右手坐标系的详细内容请参见sdk，或你的高等数学、高等物理教材^_^）DirectX3D通常使用左手坐标系。

以下是投影函数的签名：

```
public static Matrix PerspectiveFovLH( float fieldOfViewY,float aspectRatio,float znearPlane,float zfarPlane);
```

投影变换描绘了场景的视见体（注：即可见部分）。视见体是由可视角度和前裁剪面（Near Plane）与后裁剪面（Far Plane）定义的一个平截头体（注：比如四棱锥横截面与底面之间的部分，上帝保佑，你还记得高中几何），在这个平截头体之内的即是可见部分。函数签名里的nearPlane和farPlane两个参数，描绘了锥体的边界：farPlane就是锥体的底面，而nearPlane则是横截面。



fieldOfView参数描绘了锥体的角度。aspectRatio类似于电视的高宽比，比如，宽银幕电视的高宽比是1.85。你可以用可视区域的宽度比上高度得出这个值。DirectX3D只绘制在这个平截头体中的物体。

既然我没从来没有进行过投影变换，也就根本不存在一个视见体，因此DirectX3D什么也没有绘制。但是，就算我们进行了投影变换，我们还没有进行包含了摄像机信息的**观察变换view transform**。可以用一下函数完成这个任务：

public static Matrix LookAtLH(Matrix pOut, Vector3 cameraPosition, Vector3 cameraTarget, Vector3 cameraUpVector);

仅仅通过各变量的名字你就可以知道如何使用这个函数。其中三个是用来描述摄像机的属性：它的位置、它观察点的位置以及一个被参考为“up”的方向。有了投影变换和观察变换的帮助，DirectX3D已经有足够的信息来绘制三角了。添加代码：（参见DirectX sdk Tutorial 3: Using Matrices中的SetupMatrices()函数）

再运行一次试试，哦，我们已经有一个三角了，不过它完全是黑色的！问题在哪呢？在没有经过变换的环境里，DirectX3D默认使用灯光来计算场景中几何体每一个像素的颜色，我们没有定义灯光，也没有额外的光照在三角上，So，它完全是黑色的。

既然我们已经为每一个点定义过了颜色，现在，可以安全并且简单的把场景里的灯关了。加上如下代码：

dev.RenderState.Lighting = false;

再试一次，终于，我们回到了未变换坐标前的样子。做了这么多改变到底有什么好处呢？和在屏幕上直接绘制相比最大的好处就在于获得了一个三维空间里的三角形——迈向伟大三维作品的第一步！^_^

既然有了三维空间里的三角，怎样做才能让他看起来确实是一个空间里的三角呢？最简单的事就是让它旋转起来。如何来做呢？很简单，只需更改世界坐标就可以了。

Device的世界坐标变换会把每一个用局部坐标定义的顶点位置转换为用世界坐标定义的顶点位置。（the world transform on the device is used to transform the objects being drawn from model space ,whice is where each vertex is defined with respect to the model, to world space,where each vertex is actually placed in the world.）Matrix对象的很多方法能完成这种变换：

device.Transform.World = Matrix.RotationZ((float)Math.PI/6.0f);

它告诉DirectX3D除非指定一个新的世界坐标变换，否则在这段代码之后所有绘制的对象都将进行这种变换。以上的世界坐标变换是根据所给的弧度旋转Z轴。注意这里的参数必须是弧度而不是角度。有规律的改变参数值就能让三角形平滑的转动起来了（以下代码略，参考sdk中的示例）。

我们旋转的三角并不能给人留下深刻的印象。来试试让他变得特别一点，同时旋转多个轴。很幸运，恰好有这样一个方法，好了，更新代码：

device.Transform.World = Matrix.RotationAxis(new Vector3（angle/((float)Math.PI*2.0f), angle/((float)Math.PI*4.0f), angle/((float)Math.PI*6.0f)）, angle/((float)Math.PI);

这里使用了RotationAxis函数，通过这个函数，我们先定义了旋转轴，并在每一维上用一个简单的式子不停改变轴的位置，然后再传入三角形围绕着轴旋转的角度，就像先前做的一样。

再次运行程序，哦，我们确实得到了一个围绕着一条旋转的轴转动的三角形，但似乎三角形会有规律的消失一阵，然后再显示出来。好了，还记得我们先前提到的背面剔除（back face culling）吗？这就是背面剔除在起作用的最好例子。当DirectX3D渲染物体的时候，如果它发现某一个面没有对着摄相机，就不会绘制它，这就叫做背面剔除。那么程序在运行时，又是怎样知道某一个特定的几何面是否对着摄像机呢？快速看看DirectX3D中的裁剪选项或许能给你一点提示。三种可用的剔除选项分别是：none，clockwise(顺时针)以及counterclockwise（逆时针）。在clockwise以及counterclockwise的情况下，当简单几何体的顶点排列顺序与剔除模式相反时，它就不会被绘制。

看看我们的三角形，它的顶点是按逆时针顺序来排列的（注：有关顶点的排列顺序，可参考 sdk 文档 Face and Vertex Normal Vectors）。DirectX3D 默认的剔除模式就是逆时针模式。

你可以简单在顶点集合中把第一个和第三个元素交换一下，看看会有什么不同。

现在我们知道背面剔除是怎样工作的，很显然，我们简单的程序并不需要剔除功能。有一个简单的 render state 来控制剔除模式，添加如下代码：

```
Device.RenderStates.CullMode = Cull.None;
```

再一次，erying works as expected，试试拖放窗口的大小会怎样？？

拖放窗口时自动重置 Device

任何曾经使用 C++或 VB 开发 DirectX3D 的人都知道，在改变窗口大小时，需要重新设置 devicd，否则，DirectX3D 会按原来的分辨率继续渲染场景，并且把结果拷贝到(通过拉伸)新的窗口。当通过 Windows Form 控件创建 device 时，聪明的 Mamaged DirectX 能发现你改变了窗口的大小，并且重置 device。毫无疑问，程序总是能在正常的行为下运行，同时，你也能方便的自己重置 device。在自动重置 device 之前，会引发一个叫做 DeviceResizing 的事件。捕获这个事件，把 EventArgs 类的 Cancel 成员设置为 true，就能回到默认的行为，在创建 device 之后加上如下代码

```
private void CancelResize(object sender, CancelEventArgs e)
{
    e.Cancel = true;
}
```

如你所见，这个方法只是简单的 say yes，我们确实想要取消这个操作。现在订阅事件处理程序，让 device 知道不进行这种操作 ：

```
device.DeviceResizing += new CancelEventHandler(this.CancelResize);
```

（注：CancelEventHandle 委托在 System.ComponentModel 名称空间）

运行程序，最大化窗口。三角的位置还和原来一样，不过这次看起来可怕极了。边缘都是锯齿，看起来糟糕透了。可以删除我们刚添加的代码了。Managed DirectX 默认操作已经帮我们完成了这个任务，可以直接利用它。

我说：“要有光”，于是就有了光

我们绘制了三角形并且让他转起来了，怎样才能让他更好呢？当然是灯光。在前面曾简要的提到过它，事实上，那个时候我们完全关闭了灯光。首先要做的就是先回到那个黑暗的场景：

```
deviceRenderState.Lighting = true;
```

其实你甚至可以把整行都删了，device 的默认行为是打开灯光的；只是为了让代码更清楚才保留它。现在获得了一个黑色的旋转三角。或许我们应该先定义一盏灯，再来打开它。你可能已经注意到有一个灯光数组连接到了 device 类上，并且这个数组的每一个元素都保存了有关灯光的大量属性。我们希望定义场景里的第一盏灯并且打开它，So，在 OnPaint 方法定义了三角形之后的地方（注：与 sdk 中有区别，不过都是一样的效果^_^）添加如下代码：

```
device.Lights[0].Type = LightType.Point;
device.Lights[0].Positon = new Vector3();
```

```
device.Lights[0].Diffuse = System.Drawing.Color.White;
device.Lights[0].Attenuation = 0.2f;
device.Lights[0].Range = 1000.0f;
device.Lights[0].Commit();
device.Lights[0].Enabled = true;
```

这些代码什么意思呢？首先申明了要创建的灯光类型，我们选择了一个在所有方向上辐射强度都一样的point light，创造了一个灯泡般的世界。当然，也有灯光沿着指定方向传播的direction light。direction light只会产生方向和颜色上的效果，忽略其他的灯光要素（比如光线的削弱（attenuation）和范围（range）），因此它也是计算量最小的灯光。最后一种能用的就是spot light了，类似于剧场里用来照亮舞台上人物的灯光。有许多的要素来描述spot light（位置，方向，角度，等等），所以它是系统里所需计算量最大的灯光。

在对灯光类型简单的讨论之后，我们继续。接下来设置灯光的位置。因为三角形的中心在（0，0，0），所以我们把灯光也放到那个位置。Vector3无参数的构造函数完成了这个任务。把灯光的散射颜色设置为白色，这样可以正常的照亮表面。接下来设置控制灯光强度在空间改变的削弱属性。范围是灯光能产生效果的最远距离。例子里的范围已经远远超过了我们所需要的。请查阅sdk寻找有关灯光的更多内容。

最后我们把灯光提交给了device，并使它可用。如果你浏览灯光的属性，会注意到一个叫做“Deferred”的布尔值。默认情况下，这个值是false，所以你需要在准备使用灯光之前调用Commit函数。把这个值设为true，可以取消对Commit的调用，但会带来一定的性能损失。在观看灯光的效果前一定要确定它是enable和committed的。

回到程序，你发现即使我们为场景定义了灯光，三角也还是黑色的！打开了灯，却看不到光，Direct3D一定没有照亮我们的三角形，事实上，它确实没有。只有在几何体的每一个面都有一条法线（normal）时，才会进行灯光的计算。知道了这点，我们来为三角添加法线吧，这样就能在场景里看到它了。最简单的方法就是把顶点格式改为一种包含了法线的格式。碰巧我们也有这样一个结构了，改变创建三角形的代码：

```
CustomVertex.PositionNormalColored[] verts = new CustomVertex.PositionNormalColored[3];
verts[0].SetPositon(new Vector3(0.0f,1.0f,1.0f));
verts[0].SetNormal(new Vector3(0.0f,0.0f,-1.0f));
verts[0].Color = Ststem.Drawing.Color.White.ToArgb();
verts[1]``````
````````
```

更新顶点格式来适应新的数据：

```
device.VertexFormat = CustomVertex.PositionNormalColored.Format;
```

这次最大的改变就是使用了一组包含法线的数据，并且把三角形的颜色改为白色。可以看到，我们把垂直于顶点指向外的方向定义为法矢量。因为点只是在Z平面内移动，所以沿着Z轴的负方向即是法线矢量的方向。现在程序就一切正常了。可以试着改变一下灯光的散射颜色，看看会有怎样的变化。

还有一件应该记住的事：灯光是按照每一个顶点来计算，所以在low polygon模型（就像我们简单的三角形）的情况下，灯光可能会不太真实。我们会在后边的章节里讨论一些高级灯光技术，比如per pixel linghting。这些灯光能创造一个真实的世界。

Device State and Transforms

至今为止，示例代码里还有两项没有讨论过：设备状态（device state）以及变换（transform）。对一个设备来说，有三种不同方式的设备状态：the render state，The sampler states，和 the texture state。我们仅仅使用过the render state中的几种类型；后边的两种类型是用来处理纹理的。不要担心我们很快就会谈到纹理。The render state类规定了Direct3D怎样来对场景进行光栅化。可以使用这个类来改变很多属性，包括我们已经使用过的灯光以及剔除。其他render state可用的选项有填充模式（fill mode） （比如wire frame mode）和各种雾化参数。我们也会来接下来的几章深入讨论。

前面提到过，变换就是用来把几何体位置从一个坐标系转到另一个坐标系的一系列矩阵。用于device上的三个主要变换就是world，view以及projection变换，但是也有一些其他的变换。比如用来控制texture stages的变换，就依赖于一个255的世界矩阵
```

(There are transforms thst are used to modify texture stages,as well as up to 255 world matrices??).

Swapchains and RenderTargets

Device到底作了些什么工作来绘制这些三角形呢？device有一些固定的方法来处理在哪绘制并且如何绘制对象。每一个device都有一个交换链（swap chain）以及一个渲染目标（render target）。

一条交换链实际上就是一系列被控制着用来渲染的缓冲区。所有绘图过程都是在交换链中的后备缓冲区发生。当使用SwapEffect.Flip来创建一条交换链时，后备缓冲区翻转（flipped）为真正被图形卡用于读取数据的前缓冲（front buffer）。同时，三号缓冲区变为新的后备缓冲，而先前的前缓冲变为未使用过的三号缓冲区。

真正的翻转操作是通过改变图形卡当前所读的数据区、刚读过的数据区以及后备缓冲区之间的地址来实现。只有在全屏模式下，才会发生真正的翻转操作。而在窗口模式，翻转实际上只是数据的拷贝而已，因为device并没有控制着整个显示器，仅仅是一小部分而已。虽然两种模式下结果都一样。全屏模式下，有一些驱动程序也会使用翻转操作来实现SwapEffect.Discard 或者SwapEffect.Copy。

如果使用SwapEffect.Copy或SwapEffect.Flip来创建交换链，可以确保presen()之后不会影响后备缓冲中的内容。运行时会在需要时强制创建额外的隐藏缓冲。建议使用SwapEffect.Discard来避免这种潜在的损失。这种模式允许驱动程序选择最高效的方法分配后备缓冲。使用SwapEffect.Discard时，不值得(???)在绘制新的图形前检查你是否清除了整个后备缓冲。调试模式下的运行时将会使用随机的数据来填充(刚刚使用过的)后备缓冲，让开发者检查是否忘了调用clear()。（it is worth nothing that when usuing SwapEffect.Discardyou will want to ensure that you clear the entire back buffer before starting new drawing operations. the runtime will fill the the back buffer with random data in the debug runtime so developers can see if they forget to call clear）（注：这一段内容看的不是太明白，所以把原文也给出来。Sdk中对SwapEffect枚举的解释也不是太清除。参考sdk：交换效果明确定义了调用present()之后，后备缓冲的状态。Flip交换链是一个循环的队列，可以有0～(n－1)块后备缓冲， discard交换链是一个队列, copy交换链只有一块后备缓冲。Flip中的后备缓冲在present()之后内容不会改变，所以系统需要额外内存作为后备缓冲，带来性能损失。既然后备缓冲中的内容不改变，如何构成循环队列来使用?? Discard后备缓冲中队列的长度以及怎样变化也没有明确说明，只有“The swap chain is essentially a queue where 0 always indexes the back buffer that will be displayed by the next Present operation and from which buffers are discarded once they have been displayed. An application that uses this swap effect should update an entire back buffer before invoking a Present operation that displays it.The debug version of the runtime overwrites the contents of discarded back buffers with random data, to enable developers to verify that their applications are updating the entire back buffer surface correctly.” 随机数据能帮助检查是否更新了整个后备缓冲区？？既然会丢弃数据还需要调用clear？？）

交换链的后备缓冲区也同样能作为渲染目标。毫无疑问，当创建了device，创建了交换链之后，渲染目标就被设置为链的后备缓冲。一个渲染目标就是能保存所执行的绘制任务的输出的表面（a surface that will hold the output of the drawing operations that you perform）。如果你创建了多个交换链的话，就必须确定预先更新了device的渲染目标。后边我们会稍后讨论这点。

第二章 选择正确的Device

The number of possible permutations when cretion a device is quite staggering. 如今，市场里有大量不同类型的显示卡，记住每种显卡所支持的特性几乎时不可能的。你应该询问device，让它告诉你它所支持的特性。我们接下来将讨论：

枚举系统里所有的适配器（adapter）

枚举每一个device所支持的格式

确定所列举的设备功能

枚举系统里的适配器

如今的大多数系统都支持多显示器。虽然这还不是主流配置，但多显示器确实很有用，并且变的越来越流行。在过去，这是高端图形卡专有的功能。但现在ATI，nVidia以及Matrox都支持让多台显示器共享一块显卡的多头显示技术。

Direct3D的device必须指定给每一个适配器。在这里，你可以把“适配器”理解为一块链接了特定显示器的显卡。比如ATI Radeon 9700的显卡只是一块物理适配器，但它有两个显示器接口（DVI和VGA），因此，在Direct3D里，它有两个适配器。也许你不知道选哪一个，甚至不确定有多少device在运行游戏的系统里，那么怎样来检测它们并且选择正确的一个呢？

在Direct3D里，一个叫做Manager的静态类可以简单的完成以上任务：枚举适配器和device的信息；获得系统里device所支持特性的信息。

Manager类最重要的属性就是适配器的列表。在许多地方都会用到这个属性。它有一个“count”成员储存了系统里适配器的数量。因此，可以直接用索引访问适配器（e.g. Manager.Adapters[0]），也可以枚举出系统里所有适配器。

用一个简单的程序测试一下这个功能，它将以树状结构显示出系统里的适配器，以及他们所支持的显示模式：

1. 创建新的C# Windows Formd工程；
2. 添加DirectX组件；
3. 创建一个TreeView控件，并且占满整个窗口：把Dock属性设置为fill

好了，现在该加入扫描每一个适配器，显示所支持的每一种显示模式的函数了：

```
Public void LoadGRaphics()
{
    foreach(AdapterInformation ai in Manager.Adapters)
    {
        treeNode root = new TreeNOde(ai.Information.Description);
        treeNode driverInfo = new TreeNode(string.Format(“Driver information:{0} – {1}”, ai.Information.DriverName, ai.Information.DriverVersion) );
        root.Node.Add(driverInfo);
        treeNode displayMode – new TreeNodeJ(string.Format(“Vurrent Display Mode:{0} × {1} × {2}”,
                                                ai.CurrentDisplayMode.Width,
                                                ai.CurrentDisplayMOde.Height,
                                                ai.CurrentDisplayuMode.Format) );
        foreach(DisplayMode dm in ai.SupportedDisplayModes)
        {
            treeNode supportedNode = new TreeNode(string.Format(“Supported:{0} × {1} × {2}”, dm.Width, dm.Height, dm.Format) );
            displayMode.Nodes.Add(supportedNode);
        }
        root.Nodes.Add(displayMode);
        treeView1.Node.Add(root);
```

```
        }
    }
```

虽然代码看起来有一点点多，但它所做的事情实际上是非常简单的。你可以先分开来看看我们都作了些什么。首先，我们枚举系统里的适配器。C#的Foreach迭代器使这个过程异常的简单。对每一个适配器来说，这个循环都只执行一次，并且用给定的适配器填充AdapterInformation结构。观察一下AdapterInformation结构，有以下几个成员

```
Public struct AdapterInformation
{
int adapter;
DisplayMode CurrentDisplayMode;
AdapterDetails Information;
AdapterDetails GetWhqlInformation();
DisplayModeEnumerator SupportedDisplayModes;
}
```

这里adapter成员指创建device时的适配器序数。序数是一个基于0的索引，并且序数的个数等于系统里适配器的个数。两个返回AdapterDetails结构的成员都使用同一个方法返回同样的结果。对Information成员来说，Windows Hardware Quality Labs（WHQL）并不返回细节，而GetWhqlInformation却可以。获得这些信息要花费一些代价及事件，所以我们把它分成了两部分。

AdapterDetails结构保存了适配器的大量信息，包括对适配器自身的描述以及驱动信息。虽然这不是一定会用到的，但应用程序却能依次作出对硬件类型的判断。

剩下的两个成员返回DisplayMode结构。这些结构包含了大量的显示模式，包括显示的高度和宽度，刷新率以及使用的格式。CurrentDisplayMode返回当前的显示模式，SupportedDisplayModes返回适配器所支持的模式的列表。

So，我们用从Information属性获得的对device的描述作为tree view的根节点。然后加入了一个表示驱动程序名字以及版本号的子节点。同样也加入了一个显示当前显示模式的子节点，并且在这个子节点下列出了所有支持的显示模式。

运行程序，可以看到包含了所有支持模式的列表。填充present parameter结构时，这些模式都能当作正确的后备缓冲格式。每一个枚举出来的模式后面都有一个以固定模式显示的字符串（e.g　X8R8G8B8）,字母和数字交替出现。字母表示了数据的类型，数字表示这种类型的数据所占的位数。下边是常见的字母：

A－－alpha　　B－－blue　　X---unused L----luminance　R----red　P----palette　　G---green

（虽然有很多种格式，但只有几种能正确的用于后备缓冲以及显示模式。可用于后备缓冲的模式包括：A2R10G10B10, A1R5G5B5, A8R8G8B8, X1R5G5B5, X8R8G8B8, R5G5B5; Display formats can be the same as the back buffer formats, with the exception of those that contain an alpha component . The only format that can be used for a display weith alpha is A2R10G10B10，and even then that’s only in full-screen mode.）

每种类型所占的位数加起来，就是这种格式的总大小。比如X8R8G8B8，就是32位的格式，红、绿、蓝各8位，还有8位没有使用。

至今为止，我们获得了要创建的适配器序数，要支持的后备缓冲格式，那么关于device构造函数的其他参数呢？很幸运，Manager类有我们所需的一切。

判断哪一个设备是可用的

manager类有许多方法可以用来检测你的适配器是否支持一个特定的功能。比如，你需要检测适配器是否支持一种特殊的格式，但又不想枚举所有可能的适配器以及格式，那你就可以用manager类来解决这个问题。使用如下的方法：

pubic static System.Boolean CheckDeviceType(int adapter, DeviceType checkType, Format DisplayFormat, Format backBufferFormat, bool windowed, int result)

这个方法可以快速的检测出device是否支持你将要使用的格式。第一个参数是你要检测的适配器序数；第二个是要检测的device类型，但这个值大多数情况下都被设置为DeviceType.Fardware。接着指定将使用的后备缓冲类型和显示格式，以及是否需要全屏显示。最后一个参数是可选的，如果使用的话他将返回关于这个方法的一个整数（即COM中的HRESULT）。如果这是一个有效的设

备，则方法返回true，否则为false。当你预先知道要使用的格式使，这个方法是很有用的。

（应该注意到，在窗口模式，后备缓冲的格式不一定要匹配于显示格式，只要你的硬件支持适当的颜色转换就可以了。不论你的硬件是否支持这种功能，CheckDeviceType方法都会返回适当的结果，应该使用manager类的CheckDeviceFormatConversion方法来判断是否支持这种转换。也可以在窗口模式下使用Format.Unknown。全屏模式下不需要这种转换。）

检测Device的功能（capabilities）

我们把每一个device能完全用硬件实现的功能都叫做“capability”，或简称做“Cap”。Direct3D有一个Caps结构可以列出device所支持的每一种可能的capabilities。创建了device之后，就可以使用device的Caps属性来检测他所支持的特性，但如果在创建设备之你就想知道device所支持的特性该怎么办呢？自然，Manager类也有一个方法能完成这个任务。

现在，先前的程序里加一点点代码来获得系统里每一种适配器的capabilities。我们将不再使用tree view来显示这些capabilities，应为这可能包含了数百种capabilities。最好的方法是使用一个text box。回到windows form的设计模式，把tree view的Dock属性改为“Left”，把宽度改为现在的一半；入text box控件，把Dock属性设置为“Fill”，Multiline设置为true，Scrollbars设置为“both”。

现在你可能想为程序添加一个钩子（hook），这样在选择了一个适配器之后，textbox里的数据也会更新。使用tree view的AfterSelect事件，添加如下代码：

```
private void treeView_1AfterSelect(object sender, System.Windows.Forms.TreeViewEcentArgs e)
{
    if (e.Node.Parent == null)
    {
        textBox1.Text = e.Node.Text + “Capabilities: \r\n\r\n” + Manager.GetDeviceCaps(e.Node.Indes, DeviceType.Hardware).ToString().Replace(“\n”, “\r\n”);
    }
}
```

如你所见，相当简单。运行一下看看结果吧。

第三章 使用简单的渲染技术

至今为止，我们的渲染工作效率都很低。每次渲染场景时，都要分配新的顶点列表，并且所有东西存储在系统内存里。现代显卡集成了足够显存，把顶点数据存放在显存可以获得大幅的新能提升：存放在系统内存里的数据，渲染每一帧时都要拷贝到显卡，这会带来极大的损失。只有移除每帧时的这种分配才能帮助我们提高性能。

使用顶点缓冲（Using Vertex Buffers）

Direct3D已经包含了这种机制：顶点缓冲（vertex buffer）。顶点缓冲，就像他名字的意思一样：一块储存顶点的内存。顶点缓冲的机动性能完美实现共享场景里变经过变换的几何体。如何让我们在第一章编写的三角形程序使用顶点缓冲呢？

创建顶点缓冲同样简单，有三个构造函数能完成这个任务，我们依次来看看：

```
public VertexBuffer( Device device, int sizeOfBufferInBytes, Usage usage, VertexFormats vertexFormat, Pool pool);
public VertexBuffer( Type typeVertexType, int numVerts, Device device, Usage usage,VertexFormats vertexFormat, Pool pool);
```

以下是各参数的意义：

- device——用来创建顶点缓冲的device，创建的顶点缓冲只能被这个device使用；
- sizeOfBufferInBytes——所创建的顶点缓冲大小，以字节为单位。使用带有这个参数的构造函数创建的顶点缓冲可以存放任何类型的顶点；
- typeVertexType——如果去要创建的顶点缓冲只储存一种类型的顶点，则使用这个参数。它的值可以是CustomVertex类中的顶点结构类型，也可以是自定义的顶点类型。且这个值不能为null；
- numVert——指定了顶点缓冲的储存类型之后，也必须指定缓冲储存的顶点数量最大值。这个值必须大于0；
- usage——定义如何使用顶点缓冲。并不会是所有Usage类型的成员都能使用，只有一下几个是正确的参数：DoNotClip,Dynamic, Npatches, Points, PTPatches, SoftwareProcessing, WriteOnly;
- vertexFormat—— 定义储存在顶点缓冲中的顶点格式。，如果创建的为通用缓冲的话，则使用VertexFormat.None；
- pool——定位顶点缓冲使用的内存池位置，可以指定一下几个内存池位置：Default， Managed， SystemMemory， Scratch。

观察第一章中的程序，把三角形的数据移动到顶点缓冲里应该很容易。首先，申明顶点缓冲变量：

```
private Device device = null;
private VertexBuffer vb = null;
```

接着添加创建三角形的代码：

```
device = new (0,DeviceType.Hardware, this.CreatFlags.softwreVertexProccessing, presentParams);
CustomVertex.positionColored[] verts = new CustomVertex. positionColored[3];
Verts[0].SetPosition(new Vector3(0.0f,1.0f,1.0f));
Verts[0].Color = System.Drawing.Color.Aqua.ToArgb();
Verts[1]`````````
Verts[2]`````````
vb = new VertexBuffer(typeof(VustomVertex.PositionColored),2,device,Usage.Dynamic| Usage.WriteOnly,
CustomVertex.PositionColored.Format, Pool.Default);
vb.SetData(vets,0,LockFlags.None);
```

唯一的改变就是定义了三角形之后的两行代码。首先，创建用来保存三个顶点的顶点缓冲。出于性能上的考虑，创建的缓冲是动态、只读的并且位于默认的内存池。接下来，我们把三角形的顶点放到缓冲内，使用简单的SetData方法。这个方法接收任何类型

的对象作为第一个参数，第二个参数是顶点缓冲中所要放置数据地址的便宜量。我们打算填充所有的顶点缓冲，所以设置为0。最后一个参数描述了当写入数据时，如何锁定缓冲。我们将稍后讨论锁存机制；现在，不用关心他是怎样锁定的。

现在编译程序，很自然，得到了一个编译错误：因为OnPaint方法里的DrawUserPrimitives需要获得verts变量。需要有一个方法告诉Direct3D，我们要绘制顶点缓冲里的内容，而不是先前所申明的数组。调用device的SetStreamSource让Direct3D绘图的时候读取顶点缓冲。这个方法有以下两种重载：

```
public void SetStreamSource(int streamNumber, VertexBuffer streamData, int offsetInBytes, int stride);
public void SetStreamSource( int streamNumber, VertexBuffer streamData, int offsetInBytes);
```

两个函数的不同之处在于其中一个多了表示（数据）流步幅大小（stride size of the stream）的参数。第一个参数是这段数据所使用流的数量。现在，把它设置为0即可；我们会在下一章讨论使用多个流。第二个参数是作为数据源的顶点缓冲，第三个则是顶点缓冲里需要DirectX绘制的数据的偏移量（以字节为单位）。stride则是缓冲里每一个顶点的大小。如果是用特定类型创建的顶点缓冲，则不需要这个参数。

现在修改绘图的方法：

```
device.SetStreamSource(0, vb, 0);
device.DrawPrimitives(PrimitiveType.TriangleLise, 0, 1);
```

正如刚才所描述的，我们把顶点缓冲作为数据流0，同时把偏移量设置为0，使用所有数据。值得注意的是，我们同时也改变了真正绘图的函数。既然所有数据都在顶点缓冲里了，就不需要调用DrawUserPrimitives方法。因为DrawUserPrimitives只是用于绘制直接传递给它的用户定义数据。更加通用的DrawPrimitives将会绘制来自数据流源里的几何体。DrawPrimitives有三个参数，第一个我们已经讨论过了。第二个表示流里的起始顶点，最后一个表示所要绘制的几何体个数。

就连这个仅绘制一个三角形的小样在使用了顶点缓冲之后都带来了10%的性能提升（基于画面更新率，即帧频frame rate）。我们会在稍后几张来讨论有关性能及帧频。不幸的是，当你尝试改变窗口大小的时候，三角形会立即消失。（注：偶在实际测试时三角形并米有消失）

有几种情况会导致这种行为，其中的两种我们先前已经讨论过了。回想一下上一章，我们知道在改变窗口大小的时候，设备会自动重置。但当所创建的资源位于默认的内存池时（比如顶点缓冲），重置设备会释放缓冲。所以当改变窗口大小的时候，重置了device，释放了顶点缓冲。Managed DirectX有一个极好的特性就是在重置device之后会自动的重建顶点缓冲。但是，这时顶点缓冲里已经没有了数据，所以没有任何东西被绘制出来。

我们可以捕获顶点缓冲一个叫做“created”的事件，它会在重建顶点缓冲，准备好填充数据的时候发生。现在是使用这个事件更新我们程序的时候了，修改代码如下：

```
private void OnVertexBufferCreate(object sender, EventArgs e)
{
    VertexBuffer buffer = (VertexBuffer)sender;
    CustomVertex.positionColored[] verts = new CustomVertex. positionColored[3];
    Verts[0].SetPosition(new Vector3(0.0f,1.0f,1.0f));
    Verts[0].Color = System.Drawing.Color.Aqua.ToArgb();
    Verts[1]`````````
    Verts[2]`````````
    buffer.SetData(verts,0,LockFlags.None);
}
```

订阅事件处理程序：

```
vb.Created += new EventHandleer(this.OnVertexBufferCreate);
OnVertexBufferCreate(vb,null);
```

这段代码为顶点缓冲订阅了事件处理程序，并且保证无论在什么情况下创建顶点缓冲，都会调用OnVertexBufferCreate方法。因为第一次创建顶点缓冲的时候，还没有订阅过处理程序，所以需要手动调用一次。

好了，通过使用video memory和顶点缓冲，我们已经把原来缓慢的小样改变为了一个高效的程序。当然，它还是相当的枯燥。那么，接下来让我们创造一个盒子吧。

三维场景里的所有几何体都是由三角形组成，那么如何来渲染一个盒子或一个立方体呢？Well，每个立方体由六个正方形构成，而两个三角形可以构成一个正方形（呵呵，这个都要讲，看来老外的数学真的不行）实际上，我们只需要获得立方体8个顶点的坐标就可以了。添加代码：

```
CustomVertex.PositionColored[] verts = new CustomVertex.PositionColored[36];
// Front face
verts[0] = new CustomVertex.PositionColored(-1.0f, 1.0f, 1.0f, Color.Red.ToArgb());
verts[2] =````, verts[3] , verts[4], verts[5] =`````````
// Back face (remember this is facing *away* from the camera, so vertices should be clockwise order)
verts[6] = new CustomVertex.PositionColored(-1.0f, 1.0f, -1.0f, Color.Blue.ToArgb());
 verts[7] , verts[8], verts[9], verts[10], verts[11]=````````
```

(注：详见附件中的源码，注意顶点申明的顺序)

正如前面提到的，盒子由12个三角形组成，每个三角形有三个顶点，构成一个顶点集合。还有几个需要修改的地方

```
vb = new VertexBuffer(typeof(CustomVertex.PositionColored),36,device,Usage.Dynamic |
Usage.WriteOnly,CustomVertex.PositionColored.Format,Pool.Default);
evice.Transform.World = Matrix.RotationYawPitchRoll(angle/(float)Math.PI, angle/(float)Math.PI*2.0f,
angle/(float)Math.PI);
device.DrawPrimitives(PrimitiveType.TriangleList,0,12);
```

这里最大的改变就是重新定义了顶点缓冲的大小。同时，我们也改变了盒子的旋转角度，让他转的更疯狂一点。最后改变所要渲染的图元数量。实际上，既然盒子完全是三维的，就没有必要看到他的背面。使用Direct3D里的默认剔除模式（逆时针）：删除前面声明剔除模式的代码。在运行程序。

非常了不起，我们现在有了一个在屏幕中疯狂旋转的彩色盒子。但是如果需要渲染一系列盒子的话，没有人希望申明一系列顶点缓冲吧。有一个简单的方法可以做到这一点。

现在我们要肩并肩的绘制三个盒子。由于现在的摄像机设置让第一个盒子占慢了整个屏幕，我们需要把他摄像机稍稍往后移一点：

```
device.Transform.View = Matrix.LookAtLH(new Vector3(0,0,18.0f),new Vector3(),new Vector3(0,1,0));
```

如你所见，我们只是把他往后移了一点点就可以看到更多场景。为了绘制更多的盒子，我们可以再次利用现有的顶点缓冲，只需要告诉Direct3D再次绘制同样的顶点就可以了。在device.DrawPrimitives之后添加一下代码：

```
device.Transform.World = Matrix.RotationYawPitchRoll(angle/(float)Math.PI, angle/(float)Math.PI/2.0f,
angle/(float)Math.PI*4.0f) *Matrix.Translation(5.0f,0.0f,0.0f);
device.DrawPrimitives(PrimitiveType.TriangleList,0,12);
device.Transform.World = Matrix.RotationYawPitchRoll(angle/(float)Math.PI, angle/(float)Math.PI*4.0f,
angle/(float)Math.PI/2.0f)*Matrix.Translation(-5.0f,0.0f,0.0f);
device.DrawPrimitives(PrimitiveType.TriangleList,0,12);
```

好了，这次我们又作了些什么呢？因为绘制第一个盒子时已经设置过VertexFormat属性，所以Direct3D知道将要绘制的顶点类型。同样，它也知道在哪里获得数据。那么绘制第二个盒子Direct3D还需要知道什么呢？只需要绘制的位置和绘制什么就可以了。

设置world transform可以把数据从局部坐标（object space）“移动”到世界坐标（world space），那么把什么用作变换矩阵呢？首先，使用类似SetupCamera函数里的方法；做一点点改变，让盒子以不同的角度旋转。然而 world transform里的另一半则

是新内容：把一个Matrix.Translation与现有的旋转矩阵相乘。变换矩阵可以把空间中的一个点移动到另一个位置。我们的变换矩阵把第二个盒子向坐移动了5个单位，第三个盒子则向右移动了5个单位。

需要注意的是两个变换矩阵相乘得到的累积效果，是由相乘时矩阵的顺序来决定的。在这里，我们的先旋转盒子，然后再移动。如果先移动再选旋转，那么结果将有很大区别。记住变换时的顺序是很重要的。

为对象添加纹理

虽然使用颜色和灯光来渲染很有趣，但仅使用这样的技术，对象看起来并不真实。在非三维的程序里“纹理（texture）”通常用来描述对象的粗糙程度（roughness of an object）。三维场景里的纹理就是一张用来模拟几何图元纹理的2D位图。Direct3D可以同时为每一个图元渲染8层纹理，但现在，我们只解决每个图元一张纹理的情况。因为Direct3D使用普通的位图作为它的纹理格式，任何加载的位图都能当作纹理对象。 如何把2D的纹理映射到3D的对象上呢？绘制到场景中的每个对象都有一个可以在光栅化时把每个texel映射到屏幕特定位置的纹理坐标。textl是texture element的缩写，或者表示纹理中每个address的特定颜色值。Address可以想象为一个表示行和列的数字，分别称为U，V坐标。一般来说，这些值都是标量，取值范围从0.0到1.0 。（0，0）表示纹理的左上角，（1,1）表示右下角，中央的坐标为（0.5,0.5）。

为了使用纹理来渲染盒子，必须改变盒子的顶点格式，以及传递给图形卡的数据。使用纹理坐标来代替顶点数据中的“color”元素。虽然同时使用颜色和纹理都是有效的，当作为练习，我们只用纹理来定义图元的颜色。修改代码：

```
CustomVertex.PositionTextured[] verts = new Microsoft.DirectX.Direct3D.CustomVertex.PositionTextured[36];
verts[0] = new CustomVertex.PositionTextured(-1.0f,1.0f,1.0f,0.0f,0.0f);
vert[1]````````````(略)
```

显然，最大的改变就是储存顶点集合的数据类型。每个顶点中的最后两个float值储存了渲染图元所用的纹理U、V值。应为盒子每个面和纹理都为正方形，所以直接把纹理映射到每个面就可以了。注意，图元的左上角映射纹理的（0，0）textl，右下角映射到（1，1）textl。同时，我们还必须修改创建顶点缓冲的地方：

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

有如此多的重复代码，现在让我们用一个简单的方法来绘制盒子，添加一个函数完成这个任务。

```
private void DrawBox(float yaw, float potch, float roll, float x, float y,float z,texture t)
{
    ngle += 0.01f;
    device.Transform.World = Matrix.RotationYawPitchRoll(yaw,pitch,roll)*Matrix.Translation(x,y,z);
    device.SetTexture(0,t);
    device.DrawPrimitives(PrimitiveType.TriangleList,0,12);
}
```

前六个参数和我们之前使用的一样，最后一个新的参数表示渲染时所使用的纹理。我们还调用了SetTexture方告诉Direct3D渲染时使用哪个纹理。它的第一个参数是这张纹理的“层（stage）”。还记得先前我提过可以为一个图元渲染8层纹理吗，这个参数就是这些纹理的索引。因为只有一张纹理，我们使用第一个索引，0。同时应该注意到，我们修改了angle变量以及world transform，可以把SetupCamera里同样的几行删了。

在调用新方法渲染之前，先要申明一些将要使用的纹理，源码里附带了一个包含三张纹理的资源文件。分别为puck.bmp,ground.bmp,banana.bmp。添加如下代码：

```
private Texture tex = null;
private Texture tex1 = null;
private Texture tex2 = null;
```

这是我们即将使用的三张纹理。但是，还需要真正“装配”起作为资源嵌入的三张位图。在创建顶点缓冲之后添加如下代码：

```
tex = new Texture(device,new Bitmap(this.GetType(),"puck.bmp"),0,Pool.Managed);
tex1 =• • • • • （略）
```

Texture的构造函数接受四个参数。第一个是用于渲染纹理的device。场景里所有的资源（纹理，顶点缓冲，等等）都要和device发生联系。下一个参数Bitmap是我们获取纹理数据的地方。第三个参数Usage，先前已经讨论过它。最后一个参数是储存纹理的内存池位置。方便起见，现在使用托管的内存池。Texture其他的构造函数包括：

（注：此处略去一个在DX9c中已经不存在的构造函数）

```
public Texture( Device device, int width, int height, int numLevels,Usage usage, Format format, Pool pool);
public Texture( Device device, Stream data, Usage usage,Pool pool);
```

第一个方法允许我们从“空白”开始创建一张纹理，可以指定它的高度、宽度，细节程度（number of levels of detail）。最后一个和我们使用的很相似，但使用流而不是位图对象。当然，流中的数据要能被转换为位图。TextureLoad类中还有一些关于加载位图的有趣方法，我们将在下一张讨论。

好了，现在已经定义且加载了位图，是更新绘图代码的时候了，使用如下代码代替先前的绘图代码：

```
DrawBox(angle / (float)Math.PI, angle / (float)Math.PI * 2.0f, angle / (float)Math.PI / 4.0f, 0.0f, 0.0f, 0.0f, tex);
DrawBox(angle / (float)Math.PI, angle / (float)Math.PI / 2.0f, angle / (float)Math.PI * 4.0f, 5.0f, 0.0f, 0.0f, tex1);
DrawBox(angle / (float)Math.PI, angle / (float)Math.PI * 4.0f, angle / (float)Math.PI / 2.0f, -5.0f, 0.0f, 0.0f, tex2);
```

第四章 更多渲染技术

在讨论过了基础渲染方法之后,我们应该把注意力放到一些能提高性能,并且让场景看起来更好的渲染技术上来：

渲染各种图元类型

至今位置，我们只渲染过一种类型的图元，称为三角形集合。实际上，我们可以绘制很多种不同类型的图元，下边的列表描述了这些图原类型：

PointList－－这是一个自我描述的图元类型，它把数据作为一系列离散的点来绘制。不能使用这种类型绘制indexed primitives。

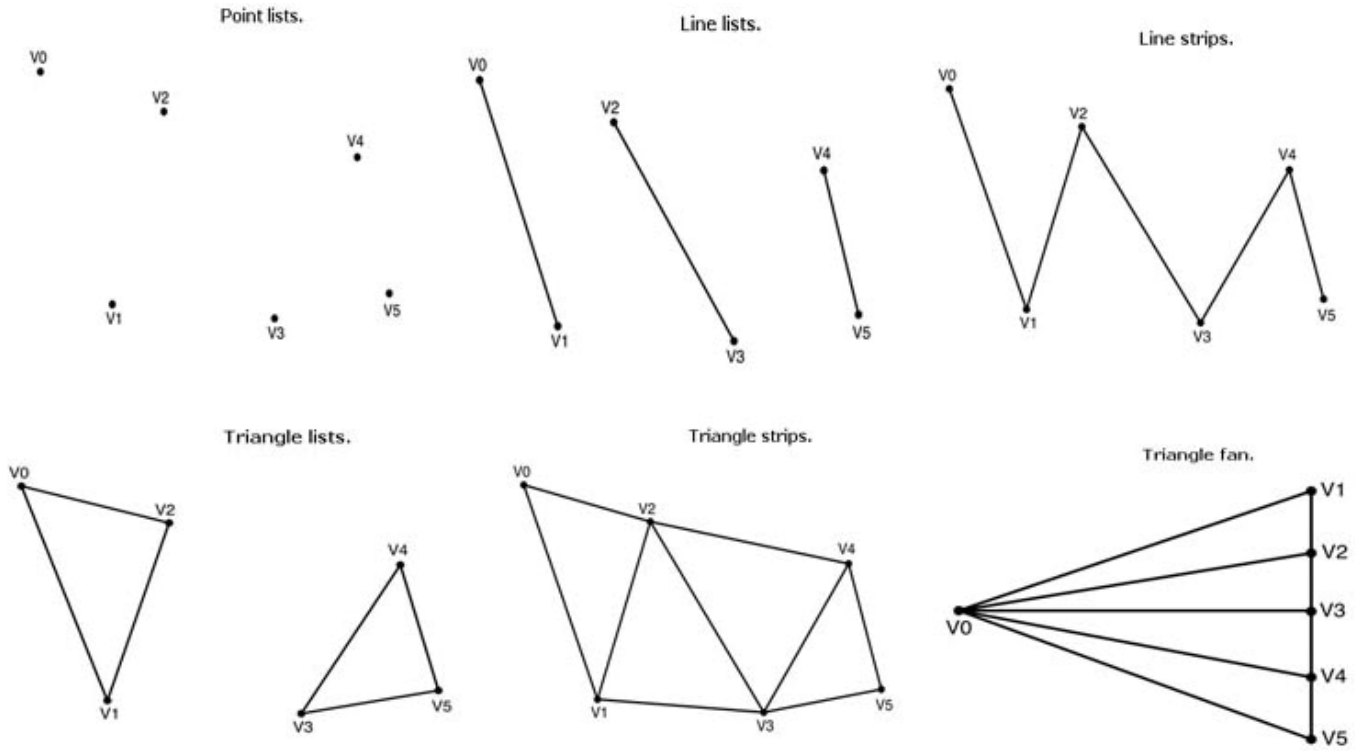
LineList——把每一对点作为单独的直线来绘制。使用时至少需要有两个顶点。

LineStrip——把顶点绘制为一条折线。至少需要两个顶点。

TrangleList——这就是我们一直在使用的类型。每三个顶点被绘制为一个单独的三角形。通过当前的剔除模式来决定如何进行背面剔除。

TrangleStrip——三角形带是一系列相连的三角形，每两个相邻的三角形共享两个顶点。剔除模式会自动翻转所有偶数个三角形（flipped on all even-numbered triangles），因为相邻的三角形共享两个顶点，他们会被翻到反方向。这也是复杂的3D对象使用的最多的图元类型。

TrangleFan——与三角形带相似，不过所有的三角形都共享一个顶点。



可以使用同样的数据来绘制任意类型，任意数量的图元。Direct3D会根据给定的图元类型来绘图。写一点来嘛来绘制一下这几种图元吧。

修改我们创建顶点缓冲时的代码。因为不需要移动顶点，可以把SetupCamera里的world transform删除了，同样所有引用到angle成员的代码也可以删除了。添加一下代码：

```
private const int NumberItems = 12;
```

12虽然是随便挑选的数字，但也有一定的原因。太多的顶点会让屏幕太拥挤，同时，顶点的数量要同时能被2和3整除。这样无

论那种图元都能都被正确的渲染。接下来修改创建顶点缓冲的代码：

```
vb=new VertexBuffer(typeof(CustomVertex.PositionColored), NumberItems, device, Usage.Dynamic | Usage.WriteOnly,
CustomVertex.PositionColored.Format,Pool.Default);
CustomVertex.PositionColored[] verts = new CustomVertex.PositionColored[NumberItems];
for(int i=0;i<NumberItems;i++)
{
    float xPos = (float)(Rnd.NextDouble()*5.0f) - (float)(Rnd.NextDouble()*5.0f);
    ・・・・・・（详见源码）
    verts[i].SetPosition(new Vector3(xPos,yPos,zPos));
    verts[i].Color = RandomColor.ToArgb();
}
```

这里没有什么特别的地方，我们修改了顶点缓冲大小来保存足够多的顶点。接下来，修改了创建顶点的方法，用一种随机的方式来填充顶点。你可以在源码中找到关于Rnd和RandomColor的声明。

现在需要修改绘图方法了。不停的滚动显示几种类型的图原，可以简单的展示出他们之间的联系。我们每两秒钟显示一种类型。可以根据开机时到现在为止的相对时间（in ticks）来计时。添加一下两个成员变量的声明：

```
private bool needRecreate = false;
private static readonly int ImitialTickCount  = System.Environment.TickCount;
```

第一个布尔变量控制着在每个“周期”开始的时候重新创建顶点缓冲。这样，就不必每次都显示同样的顶点。用一下代码代替简单的DrawPrimitives方法：

（见源码中带有switch的部分）

这基本上是一段可以自我解释的代码。根据一个周期中的不同时刻，调用DrawPrimitives来绘制相应的图原。注意，由于图原类型的不同，相同数量的顶点能绘制的图原数也是不同的。运行程序，将按照PointList，Linelist，LineStrip，TragleList,TangleStrip的顺序显示图原。如果你觉得显示PointList时“点”太小看不清楚，可以通过调整render state把它稍稍放大一点：

```
device.RenderStare.PointSize = 3.0f;
```

使用索引缓冲（Index Buffer）

还记得我们创建盒子时的带码吗，我们一共创建了36个顶点。实际上，我们只使用了8个不同的顶点而已，即正方形的8个顶点。在这样的小程序里把相同的顶点储存许多次并不会出什么大问题。但在需要储存大量数据的大得多的程序里，减少数据的重复来节约空间就显得很重要了。很幸运，Direct3D里一种成为索引缓冲的机制能让同一个图原共享他的顶点数据。

就像他的名字暗示的那样，索引缓冲就是一块保存了顶点数据索引的缓冲。缓冲中的索引为32位或16位的整数。比如，你使用索引0，1，6来绘制一个三角形时，会通过索引映射到相应的顶点来渲染图像。使用索引来修改一下绘制盒子的代码吧，首先修改创建顶点的方法：

```
vb=new  VertexBuffer(typeof(CustomVertex.PositionColored),  8,  device,  Usage.Dynamic  |  Usage.WriteOnly,
CustomVertex.PositionColored.Format,Pool.Default);
CustomVertex.PositionColored[] verts = new CustomVertex.PositionColored[8];
verts[0] = new CustomVertex.PositionColored(-1.0f, 1.0f, 1.0f, Color.Red.ToArgb());
・・・・・（见源码OnVertexBufferCreate方法）
```

如你所见，我们戏剧性的减少了顶点的数量，仅储存正方形的8个顶点。既然已经有了顶点，那36个绘制盒子的索引应该是什么样子呢？看一下先前的程序，依照36个顶点的顺序，列出适当的索引：

```
private static readonly short[] indices =
```

```
{
    0,1,2,   //front face
    1,3,2,   //front face
    ・・・・・
}
```

为了便于阅读，索引分为3个一行，表示一个特点的三角形。第一个三角形使用顶点0，1，2第二个使用1，3，2；以此类推。仅仅有索引列表是不够的，还需要创建索引缓冲：

```
private IndexBuffer ib = null;
```

这个对象就是储存并且让Direct3D访问索引的地方。它与创建顶点缓冲的方法也很相似。接下来初始化对象，填充数据：

```
ib = new VertexBuffer(typeof(short),indices.Length,device,Usage.WriteOnly,Pool.Default);
ib.Created += new EventHandler(ib_Created);
OnIndexBufferCreate(ib,null);

private void ib_Created(object sender, EventArgs e)
{
    IndexBuffer buffer = (IndexBuffer)sender;
    buffer.SetData(indices,0,LockFlags.None);
}
```

除了参数的约束条件以外，索引缓冲的构造器简直和顶点顶点缓冲的一样。与前面提到的一样，只能使用16位或32位的整数作为索引。我们订阅了事件处理程序，并且在程序第一次运行时手动调用他。最后为索引缓冲填充了数据。

现在，需要修改渲染图像的代码来使用这个数据了。如果你还记得，我们以前使用了一个叫“SetStreamSource”的方法来告诉DirectX渲染的时候使用哪一快顶点缓冲。同样，对于索引缓冲来说也有这样一种机制，不过它仅仅只是一个属性而已，因为同一时间只可能使用一种类型的索引缓冲。在SetStreamSource之后，设置如下属性：

```
device.Indices = ib;
```

这下Direct3D知道顶点缓冲的存在了，接下来修改绘图代码。目前，我们的绘图方法尝试从顶点缓冲绘制12个图原，可是这必然不会成功，因为现在顶点缓冲里只有8个顶点了。添加DrawBox方法：

```
private void DrawBox(float yaw,float pitch,float roll,float x,float y,float z)
{
    angle += 0.01f;
    device.Transform.World = Matrix.RotationYawPitchRoll(yaw,pitch,roll) * Matrix.Translation(x,y,z);
    device.DrawIndexedPrimitives(PrimitiveType.TriangleList,0,0,8,0,indices.Length /3);
}
```

这里，我们把DrawPrimitives改为了DrawIndexedPrimitives。来看看这个方法的原型吧：

```
public void DrawIndexedPrimitives(PrimitiveType primitiveType,int baseVertex ,int minVertexIndex,int numVertices,
int startIndex, int primCount);
```

第一个参数和上一个方法的一样，表示要绘制的图原类型。参数baseVertex表示从索引缓冲起点到要使用的第一个顶点索引的偏移量。MinVertexIndex是这几个顶点中最小的顶点索引值。很显然，numVertices指的就是所要使用的顶点数量。startIndex表示从数组中的哪一个位置开始读取顶点。最后一个参数则是要绘制的图原数量。

现在通过索引缓冲中的8个顶点，就可以绘制出了构成立方体的12个图原了。接下来用DrawBox方法代替原来的DrawPrimitives方法。

```
DrawBox(angle / (float)Math.PI, angle / (float)Math.PI * 2.0f, angle / (float)Math.PI / 4.0f, 0.0f, 0.0f, 0.0f);
```

（略，详见源码）

再次运行程序，可以看到颜色非常鲜艳的盒子在旋转。我们的每一个顶点都有不同的颜色，因此，真实的反映了使用索引缓冲共享顶点的缺点。当多个图原共享顶点的时候，所有的顶点数据都是共享的，包括颜色，法线数据等等。当决定是否共享顶点时，必须确定共享数据不会带来灯光或颜色上的错误（因为灯光的计算依赖于法线）。可以看到立方体每个面的颜色都是由顶点颜色插值计算出来的。

使用深度缓冲（Using Depth Buffer）

深度缓冲（depth buffer）（也就是通常所说的z-buffer或w-buffer）是Direct3D在渲染时储存“深度”（“depth”一般指方向为从屏幕指向观察者的z轴的窗口坐标）。深度信息用于在光栅化时决定象素之间的替代关系（注：度通常用视点到物体的距离来度量,这样带有较大深度值的象素就会被带有较小深度值的象素替代,即远处的物体被近处的物体遮挡住了）。至今为止，我们的程序都没有使用过深度缓冲，所以光栅化时没有象素被遮挡住。除此之外，我们甚至还没有会相互重叠的象素，那么，现在来绘制一些会与已有的立方体重叠的的立方体吧。

在已有的DrawBox方法调用后添加如下代码：

```
DrawBox(angle / (float)Math.PI,angle / (float)Math.PI*2.0f, angle / (float)Math.PI /
4.0f,0.0f,(float)Math.Cos(angle),(float)Math.Sin(angle));
```

• • •（略）

我们在添加了三个旋转的立方体到原来中间一排的立方体上。运行程序，可以看到重叠的立方体，却不能分清两个立方体重叠部分的边界，看起来不过是一块普通的斑点而已。这就需要通过深度缓冲来处理了。

添加深度缓冲实在是一个简单的任务。记得我们传递给device构造函数的presentation parameters参数吗？well，这将是我们添加深度缓冲的地方。创建一个包含深度缓冲的device，需要用到两个新的参数：

```
public Mircosoft.DirectX.Direct3D.DepthFormat AutoDepthStencilFormat [ get, set ]
public bool EnableAutoDepthStencil [get,set]
```

把EnableAutoDepthStencil设置为true就可以为device打开深度缓冲，使用DepthFormat来指定AutoDepthStencilFormat成员。DepthFormat枚举中，可使用的值列在下表中：

D16	A 16-bit z-buffer bit depth.
D32	A 32-bit z-buffer bit depth.
D16Lockable	A 16-bit z-buffer bit depth that is lockable.
D32Flockable	A lockable format where depth value is represented by a standard IEEE floating point number.
D15S1	A 16-bit z-buffer bit depth using 15 bits for depth channel, with the last bit used for the stencil channel (stencil channels will be discussed later).
D24S8	A 32-bit z-buffer bit depth using 24 bits for depth channel, with the remaining 8 bits used for the stencil channel.
D24X8	A 32-bit z-buffer bit depth using 24 bits for depth channel, with the remaining 8 bits ignored.
D24X4S4	A 32-bit z-buffer bit depth using 24 bits for depth channel, with 4 bits used for the stencil channel, and the remaining 4 bits ignored.
D24FS8	A non-lockable format that contains 24 points of depth (as a floating point) and 8 bits for the stencil channel.

深度缓冲越大，能储存的深度数据也越多，但这是以牺牲性能为代价的。除非你确定需要使用很大的深度缓冲，否则使用最小的值就可以了。大部分现代的图形卡都支持最小 16－bit 的深度缓冲，so，添加代码：

```
presentParams.AutoDepthStencilFormat = DepthFormat.D16;
presentParams.SwapEffect = SwapEffect.Discard;
```

Perfect，现在device获得了深度缓冲。来看看有什么不同吧，运行程序。哇，结果并不是我们期盼的那样，程序被破坏了。这些立方体发生了什么？为什么加入了深度缓冲之后导致渲染被破坏了呢。呵呵，原因是深度缓冲从来没有被“cleared”，所以它一直处于一种不正确的状态。应该在clear device的同时clear深度缓冲，修改代码如下

```
device.Clear(ClearFlags.Target | ClearFlags.ZBuffer, Color.CornflowerBlue, 1.0f, 0);
```

Ok，一切正常了，休息一下来欣赏我们的作品吧^_^。

第五章 Rendering with Meshes

定义Mesh

虽然有很多时候，你需要手动创建顶点和索引数据，但更普遍的情况是从外部的资源加载已有的顶点数据，比如从一个文件。通常我们使用.X文件来保存这些信息。在前一章里，代码的大部分都用来创建几何体了。对于简单的三角形和立方体来说这似乎是完全可行的，但设想假如用相同的方式来创建拥有上万个顶点的物体将，所花费的时间和努力都将是很可怕的。

幸运的是，Managed DirectX里有一个可以封装并且加载顶点和索引数据的对象，这就是**Mesh**。Mesh可以用来储存任何类型的图形数据，但主要用来封装复杂的模型。Mesh类同样也有一些用来提高渲染物体性能的方法。所有的mesh对象都包含了一个顶点缓冲和一个索引缓存，除此之外，他还包含了一个**属性缓冲（attribute buffer）**——我们将会在这一章的后面讨论它。

真正的mesh对象包位于Direct3D扩展库（D3DX Direct3D Extensions library）中。添加对Direct3DX.dll程序集的引用，我们将尝试着使用mesh来创建一个旋转的立方体。首先，在声明顶点缓冲和索引缓冲成员之前添加mesh成员：

```
private Mesh mesh = null;
```

mesh类有三个构造函数，但现在还不需要用到其中的任何一个。Mesh类有几个静态方法可以用来创建或加载不同的模型。首先需要注意的就是“Box”方法，就像它的名字一样，它将创建包含了一个立方体的mesh。想想看，我们立刻就能渲染这个立方体，简直完美极了^_^（注：呵呵，可以删除之前所有与顶点缓冲、索引缓冲有关的代码了）。在创建device之后添加以下代码：

```
mesh = Mesh.Box(device,2.0f,2.0f,2.0f);
```

这个方法创建了一个包含顶点和索引的mesh，并且可以渲染为一个长、宽、高都为2的立方体。它和之前用顶点缓冲手动创建的立方体大小一样。我们已经把创建物体的代码减少为一行了，不能再简单了。虽然已经创建了mesh，但可以用原来的方法来渲染它吗，还是需要另辟途径？之前，在渲染时，我们需要调用SetStreamSource来告诉Direct3D从哪一块顶点缓冲读取数据，同样还必须设置索引以及顶点格式的属性。对于渲染mesh来说，这些都是不需要的。

（tips：mesh已经内置了所有顶点缓冲、索引缓冲以及顶点格式的信息。渲染时会自动设置stream source、索引和顶点格式的属性）

那么如何渲染mesh呢？Mesh会被分为一系列的子集（subsets）（依据属性缓冲的大小来分配），同时使用一个叫做“DrawSubset”的方法来渲染。修改DrawBox方法：

```
private void DrawBox(float yaw,float pitch,float roll,float x,float y,float z)
{
    angle += 0.01f;
    device.Transform.World = (Matrix.RotationYawPitchRoll(yaw,pitch,roll) *
    Matrix.Translation(x,y,z));
    mesh.DrawSubset(0);
}
```

这里把DeawIndexedPrimitives方法改为了DrawSubset。使用Mesh类创建的普通图元总是只有一个基于0的子集。好了，这就是要让程序再次运行所作的所有改动了，出乎意料的简单。运行看看吧。

Well，再次得到了九个（在源码中是12个）旋转的盒子，但是全部变为了白色对不对？观察一下mesh中顶点的顶点格式（可以通过mesh的VertexFormat属性查看），会发现只有顶点的位置和法线数据储存在mesh中。Mesh中没有关于颜色的数据，灯光也米有打开，自然一切都是白色的。

还记得第一张中提到过，只要顶点数据包含了法线的信息，就可以使用灯光吗，既然盒子有法线数据，也许我们应该吧灯光打开。默认情况下灯光是打开的，现在可以把关闭灯光的代码删了或者设置为true。

呵呵，我们成功把白色的盒子变为黑色了-.-#。 希望你已经猜到了这是因为场景中并没有光源，所以一切都是黑色的。对于指定特定的光源而言，创建一盏能照亮整个场景的灯光将会很不错。欢迎来到**环境光（ambient lighting）**。

环境光为场景提供了均衡（constant）的光源。场景中所有物体都按同样的方式被照亮，因为环境光并不依赖于其它几种光源需要的因素（比如位置、方向、衰减）。甚至不需要法线数据就可以使用环境光。环境光是最高效的灯光类型，但却不能创造出真

实的“世界”。但就现在而言，他就能达到我们满意的效果。在设置RendrState的地方添加如下代码：

```
device.RenderState.Ambient = Color.Red;
```

环境光完全是由ambient render state来定义的，接受一个颜色参数。这里，我们希望全局灯光是红色的，这样可以看到明显的效果。运行程序，你希望可以看到9个红色的旋转盒子，不幸的是，它们仍然为黑色。还遗漏了些什么呢？

使用材质和灯光（Using Materials and Lighting）

这里和我们以前使用灯光有什么不同呢？最大的不同点（除了使用的是mesh之外）在于顶点数珠中没有关于颜色的信息。这导致了光照失败。

为了让Direct3D正确的计算3D物体中特定点的颜色，除了灯光的颜色之外，还需要知道物体如何反射灯光的颜色。在真实的世界里，如果把红色的灯光照在淡蓝色的表面，那么它会呈献出柔和的紫色。你还需要描述我们的“表面”（我们的盒子）是如何反光的。

在Direct3D里，**材质（materials）**描述了这种属性。你可以指定物体如何反射环境光以及散射（diffuse）光线，镜面高光（Specular Highlights）（少后会讨论它）看起来是什么样，以及物体是否完全反射（emit）光线。在DrawBox中添加如下代码（在DrawSubset方法前）：

```
Material boxMaterial = new Material();
boxMaterial.Ambient = Color.White;
boxMaterial.Diffuse = Color.White;
device.Material = boxMaterial;
```

这里创建了一个新的材质，它的**环境颜色（ambient color）**（注：环境颜色和环境光的颜色是不同的^_^）和散射颜色值都被设置为白色。使用白色表示它会反射所有的光线。接下来，我们把材质赋予了device的Material属性，这样Direct3D就知道渲染时使用那种材质数据。

运行程序，现在可以看到正确的结果了。修改环境光的颜色可以改变所有盒子的颜色。修改材质的环境颜色元素可以改变灯光如何照亮物体（注：后悔当年没有好好听光学课啊555～～，maya完全手册中是这样说的：环境色（ambient color），当其为黑色时，表示（环境光）不会影响材质的颜色，当环境色变浅时，它就会照亮材质，并将两种颜色混和起来，从而影响材质的颜色。如何场景中有环境光，那么这些光的颜色和亮度就会控制环境色对于最终材质颜色的影响程度）。把材质改为没有红色成分的颜色（比如绿色）会使物体再次变为黑色（注：因为此时物体不会反射红色，红色的光线被物体全部吸收了），改为含一些红色成分的颜色（比如灰色gray）会使物体呈现深灰色。

先前说过，使用这种方式渲染出来的物体不会太真实。甚至看不到每个立方体的“倒角”，好像是一些红色的类立方体斑纹一样。这是因为环境光以同样的方法来计算所有顶点。我们需要一盏真实一点点的灯，在创建环境光之后添加如下代码：

```
evice.Lights[0].Type = LightType.Directional;
device.Lights[0].Diffuse = Color.White;
device.Lights[0].Direction = new Vector3(0,-1,-1);
device.Lights[0].Commit();
device.Lights[0].Enabled = true;
```

这里创建了一盏白色的方向光，照向摄像机相同的方向。现在可以看到不同方向上光影的变化了。

创建mesh的时候，有一系列物体可以使用。使用以下一种方法来创建mesh（这些方法都要求device作为第一个参数）：

(以下均使用左手坐标系)

```
mesh = Mesh.Box(device,2.0f,2.0f,2.0f);
```

Width、Height、Depth分别表示盒子在X、Y、Z轴上的尺寸

```
mesh = Mesh.Cylinder(device,2.0f,2.0f,2.0f,36,36);
```

Radius1,Radius2 表示圆柱体的下底面和上底面半径，必须为非负；Length 表示圆柱体在Z方向的高度；Slices 表示沿中心轴的片段数量，Stacks 表示沿主轴的“堆数量。（注：类似于由经、纬线分成的水平和垂直方向上的块数）

```
mesh = Mesh.Polygon(device,2.0f,8);
```

Length 表示多边形每一边的长度，Sides表示有多少条边

```
mesh = Mesh.Sphere(device,2.0f,36,36);
```

Radius表示球体的半径，Slices和Stacks的含义与Cylinder的相同。

```
mesh = Mesh.Torus(device,0.5f,2.0f,36,18)
```

InnerRadius 圆环的内径，OutterRadius 圆环的外径，Sides横截面上的面数，Rings横截面上的环数，前两个值必须为非负数，后两个必须大于等于三。

```
mesh = Mesh.Teapot(device)
```

创建一个茶壶（对一个茶壶，你没有看错^_^）。

以上每一个方法都有一个能返回邻接信息（adjacency information）的重载,每个面用三个整数来做为邻接信息，指定了相邻的三个面（Adjacency information is returned as three integers per face that specify the three neighbors of each face in the mesh）。

使用Mesh渲染复杂模型

渲染茶壶虽然很有意思，但游戏里不可能只需要渲染茶壶。大量的mesh是通过艺术家使用专业的建模软件来创造的。如果你的建模软件可以导出.X文件那么恭喜你，你很幸运（Direct SDK里包含了常用建模软件的导出转换器）。

可以通过加载x文件里储存的几种数据类型来创建mesh。当然顶点和索引数据是渲染物体的最基本要求。mesh的每个子集都会关联到一种材质。每一个材质组也同样能包含纹理信息。还可以同时使用x文件和High Level Shader Language(HLSL)文件来创建mesh。HLSL是一门高级技术，我们会在后边的内容里深入讨论。

和创建“简单”图原类型的静态方法一样,Mesh类还有两个主要的静态方法可以加载外部模型。这两个方法分别是Mesh.FormFile和Mesh.FromStream。两个方法本质上来说都是一样的，stream方法有更多的重载以适应不同大小的流。最常用的重载方法如下：

```
public static Mesh FromFile(string filename, MeshFlags options,Device device,out GraphicsStream adjacency,out ExtendedMaterial materials,out EffectInstance effects);
public static Mesh FromStream(Stream stream,int readBytes, MeshFlags options,Device device,out GraphicsStream adjacency,out ExtendedMaterial materials, out EffectInstance effects);
```

第一个参数是加载为mesh的数据源。对于FromFile方法来说，他是所要加载的文件名；对于FromStream方法来说，它是所使用的流以及要读取的数据字节数。如果使用整个流的话，只要使用没有readBytes参数的重载就可以了。MeshFlags参数控制着去哪里以及如何加载数据。这个参数的值可以是通过以下值组合而来：

Mesh Flags Enumeration Values

PARAMETER	VALUE
MeshFlags.DoNotClip	Use the Usage.DoNotClip flag for vertex and index buffers.
MeshFlags.Dynamic	Equivalent to using both IbDynamic and VbDynamic.
MeshFlags.IbDynamicUse	Usage.Dynamic for index buffers.
MeshFlags.IbManaged	Use the Pool.Managed memory store for index buffers.
MeshFlags.IbSoftware	ProcessingUse the Usage.SoftwareProcessing flag for index buffers.
MeshFlags.IbSystemMem	Use the Pool.SystemMemory memory pool for index buffers.
MeshFlags.IbWriteOnly	Use the Usage.WriteOnly flag for index buffers.
MeshFlags.VbDynamic	Use Usage.Dynamic for vertex buffers.
MeshFlags.VbManaged	Use the Pool.Managed memory store for vertex buffers.
MeshFlags.VbSoftwareProcessing	Use the Usage.SoftwareProcessing flag for vertex buffers.
MeshFlags.VbSystemMem	Use the Pool.SystemMemory memory pool for vertex buffers.
MeshFlags.VbWriteOnly	Use the Usage.WriteOnly flag for vertex buffers.
MeshFlags.Managed	Equivalent to using both IbManaged and VbManaged.
MeshFlags.Npatches	Use the Usage.NPatches flag for both index and vertex buffers. This is required if the mesh will be rendered using N-Patch enhancement.
MeshFlags.Points	Use the Usage.Points flag for both index and vertex buffers.

MeshFlags.RtPatches	Use the Usage.RtPatches flag for both index and vertex buffers.
MeshFlags.SoftwareProcessing	Equivalent to using both IbSoftwareProcessing and VbSoftwareProcessing.
MeshFlags.SystemMemory	Equivalent to using both IbSystemMem and VbSystemMem.
MeshFlags.Use32Bit	Use 32-bit indices for the index buffer. While possible, normally not recommended.
MeshFlags.UseHardwareOnly	Use hardware processing only.

下一个参数是渲染mesh的device。因为资源必须关联到一个device，这是个必选参数。adjacency参数是一个“out”参数，着表示在这个方法结束后adjacency会被分配并且传递出去，它将返回聆接信息。ExtendedMaterial类保存了普通的Direct3D材质和一个加载为纹理的字符串。这个字符串通常是使用的纹理或资源文件名，因为加载纹理是由程序来进行的，它也可以是任何用户提供的字符串。组后，EffectInstance参数描述了用于mesh的HLSL材质文件和值。可以根据需要选择具有不同参数的方法重载。

这里讨论了大量关于加载和渲染mesh的细节，但实际上并没有那么复杂。一开始你可能会有些担心，但看到实际代码之后，确实很简单。现在就来试试吧。首先，要确保有可以用来为不同的子集储存材质和纹理的变量成员。在声明了mesh之后添加如下代码：

```
private Material[] meshMaterials;
private Texture[] MeshTextures;
```

因为mesh中可能有许多不同的子集，所以需要分别创建一个材质和纹理的数组以满足每一个子集的需要。好了现在来添加一些真正加载mesh的方法吧，创建一个名为“LoadMesh”的函数，代码如下：

```
private void LoadMesh(String file)
{
    ``````(此处代码较多，详见源码)
}
```

好啦，虽然看起来比我们之前所作的简单工作吓人一点，但实际上却不是这样。首先，我们我们声明了用于保存mesh子集信息的ExtendenMaterial数组。然后，调用FromFile方法加载mesh。我们现在并不关心adjacency或HLSL参数，所以选用了不含这两个参数的重载。

加载mesh之后，需要为大量的子集储存材质和纹理信息。确定了是否有不同的子集之后，我们最终使用子集的大小为材质和纹理成员分配大小。接下来，使用循环把ExtenedMaterial中的数据拷贝到meshMaterials中。如果子集中还包含纹理信息的话，使用TextureLoader.FromFile方法来创建纹理。这个方法接受两个参数，device以及作为纹理的文件名，这个方法可要比以前使用的System.Drawing.Bitmap快**许多**。

为了绘制mesh，还需要添加如下方法：

```
private void DrawMesh(float yaw,float pitch,float roll,float x,float y,float z)
{
    angle += 0.01f;
    device.Transform.World = Matrix.RotationYawPitchRoll(yaw,pitch,roll)*Matrix.Translation(x,y,z);
    for(int i=0;i<meshMaterials.Length;i++)
    {
        device.Material = meshMaterials[i];
        device.SetTexture(0,meshTextures[i]);
        mesh.DrawSubset(i);
    }
}
```

你可能已经注意到，这个方法保留了DrawBox方法的签名部分。接下来为了绘制mesh,迭代所有材质，并且执行一下步骤：

1， 把保存的材质赋予device；

2， 把纹理赋予device。这里，在没有纹理的情况下，即使值为null也不会出错。

3， 根据子集的ID调用DrawSubset方法

perfect，现在我们已经完成了加载和渲染mesh的工作了。我已经制作了一个名为tiny.x的模型。添加如下代码来加载这个模型吧：
```

```
this.LoadMesh(@"..\..\tiny.x");
```

还需要调整一下摄像机的位置，应为只是模型看起来像除了tiny之外的任何东西。由于模型非常的大，摄像机需要退后一点，修改以下方法：

```
device.Transform.Projection = Matrix.PerspectiveFovLH((float)Math.PI / 4, this.Width / this.Height, 1.0f, 1000.0f);
device.Transform.View = Matrix.LookAtLH(new Vector3(0,0, 580.0f), new Vector3(), new Vector3(0,1,0));
```

我们重修调增加了到后裁剪平面的距离，平且把摄像机移动的相当靠后，好了最后的任务：在渲染部分调用DrawMesh方法：

```
this.DrawMesh(angle / (float)Math.PI,angle / (float)Math.PI*2.0f,angle/(float)Math.PI/4.0f,0.0f,0.0f,0.0f);
```

最后，你还可以调整一下灯光的颜色试试。

我们又向前迈进了一大步，这可比总看着立方体旋转要有趣多了。

第六章 使用Managed DirectX编写游戏

选择游戏

虽然很多关于3D游戏编程的高级主题还没有讨论，但我们已经有足够的背景知识来写一个简单游戏了。这一章，我们将使用至今学过的知识，再加上一点点新的东西来创建游戏。

真正开始写游戏之前，最好先拟一份计划。我们需要确定写什么类型的游戏，它将有哪些最基本的特性，等等。考虑到目前的技术限制，自然不能写太复杂的游戏。这将是一个简单的游戏。在MS－DOS环境下，曾经有一个叫做“Donkey”的游戏，玩家控制着车不能撞到路上的donkey。听起来足够简单吧，我们将创建一个三维版本，并且用普通的障碍物来代替donkey。这个游戏叫做“躲避者（Dodger）”。

开始编码之前，需要花一点时间来策划和设计游戏。我们需要怎样的游戏，玩的时候来控制。Well，显然，要有一个Car类来控制交通工具。接下来，使用另一个类来控制障碍物将会很不错。除此之外，主要的游戏引擎类必须完成所有的渲染操作并把所有对象组织起来。

如果尝试商业游戏，那么大部分时间将会花在游戏创意上。游戏创意将会写成详细的文档，包括了游戏主题和特性的各种细节。本书的着重于讨论游戏的实际开发工作，而不是游戏发行和创意，所以我们将略过这一步。

通常开之发写还必须写完整的技术文档（technical specification）（简称为spec）。它包以适当的细节列出了所以类，以及需要实现的各种方法、属性。通常还包括表示对象之间关系的UML图。这份文档的目的是让你在编码前坐下来认真考虑程序的设计。由于本书聚焦于代码的编写，我们同样略过这一步。需要说明的是，强烈建议你在写任何代码前花点时间撰写技术文档。

编写游戏

现在可以打开VS创建项目了。创建一个名为Dodger的windows应用程序。使用DodgerGame代替代码中所有出现Form1的地方。添加对DirectX程序集的引用。创建私有的device成员，如下修改构造函数：

```
public DodgerGame()
{
    this.Size = new Size(800,600);
    this.Text = “Dodger Game”;
    this.SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.Opaque,true);
}
```

这将会把窗口设置为800×600（注：实际代码中我将会创建一个全屏的游戏，另外如果现在运行程序，会发现我们创建了一个透明的窗口），设置窗口标题和样式（style），这样渲染代码才会正常工作。接下来修改程序的入口点：

```
static void Main() {详见源码}
```

这个应该很熟悉了吧，基本上就是之前每一章用来启动程序的代码。创建窗体、初始化图形引擎，运行窗体。在initializeGraphics内做如下改动：

```
private void InitializeGraphics() {详见源码};
```

创建了presentation parameters结构之后，确保有它有深度缓冲。这里有什么新内容呢？首先，保存了默认的适配器的序数号，接下来保存了creation flags，并把它的默认值设为software vertex processing。但是，现代图形卡都在硬件层实现了vertex processing。何必把宝贵的CPU资源用在显卡可以完成的任务上呢？答案是不需要这样做，但你不知道是否真的支持这种特性，于是有了接下来的代码。在真正创建device之前，需要先保存显卡的功能（capabilities，简称Caps），这样可以用来决定使用那一种flags创建device。因为你只是创建一个硬件设备，所以只储存这几个Caps就可以了。关于检查适配器所有Caps的内容回忆一下第二章吧。

还记得使用顶点缓冲时需要在重置设备之后重建缓冲吗？我们为device订阅了created事件。当device重置之后，设定device的所有默认状态，添加如下代码：

```
private void OnDeviceReset(object sender,EventArgs e) {详见源码};
```

（注意：类似于这里的代码，你可能会使用一个层（layer）来检查支持的灯光。这种情况下，先检查是否支持一盏灯，如果可以，则创建它。然后再用类似的方法检测是否支持第二盏灯。这样即使最差的情况你也能获得一盏灯光）

这里和前面学过的代码也很类似，通过projection fransform和view transform来设置摄像机。对于这个游戏来说，我们的摄像机不需要移动，所以只需要在重置设备之后设置一次就可以了（与设备相关的状态都会在重值之后丢失）。

环境光不是最好的选择，我们已经知道他不能产生真实的光影效果，所以方向光将是不错的选择。但并不能确定设备是否支持这种光源。创建了设备之后，就不需要再使用先前的Caps结构了，device会为你保留着这些信息。如果device支持方向光，而且支持一盏以上的灯光，你应该使用它；否则，使用默认的环境光。它虽然不真实，但总比黑色的场景要好吧。最后，重载OnPaint方法，：

```
protected override void OnPaint(PaintEventArgs e){详见源码};
```

这里没有什么新内容，当然你可以把背景改为任何你喜欢的颜色。现在已经为加载模型做好了准备。创建变量来储存.X文件中的赛道模型吧。

```
private Mesh roadMesh = null;
private Material[] roadMaterials = null;
private Texture[] roadTextures = null;
```

接下来修改一下前一章里的load mesh方法。最大的改变是将把它改为静态方法，因为不止一个类会调用它，同样把所有的材质和纹理作为参数来传递，而不是作为类成员来访问。添加如下代码：

```
public static Mesh LoadMesh(Device device,string file,ref Material[] meshMaterials,ref Texture[] meshTextures){详见源码};
```

这个方法前面已经深入讨论过了。使用这个方法来加载赛道模型，还需要在重置设备的事件里添加它，在OnDeviceReset最后加上如下代码：

```
roadMesh = LoadMesh(device,@"..\..\road.x",ref roadMaterials,ref roadTextures);
```

确定你已经把赛道模型和纹理文件复制到了源文件的目录下。这段代码将会加载模型以及纹理，并储存纹理、裁制以及模型。每一帧道路mesh都需要渲染很多次，因该创建一个方法来完成渲染工作。添加如下代码：

```
private void DrawRoad(float x, float y ,float z) {详见源码};
```

你应该还记得这个方法吧，它和我们之前使用的方法如此类似。把mesh变换为正确的位置然后渲染每一个子集。我们需要每次渲染两段赛道mesh：一段是赛车现在行驶的赛道，一段是即将行驶到的赛道。实际上我们的赛车并没与移动，而是赛道在移动。这样做的原因有两个：如果每一帧都移动赛车，那么还必须同时移动摄像机来跟上它。这些而外的计算实际上是不必要的。还有一个更重要的原因：如果赛车向前移动，而且玩家很厉害，那么赛车的位置可能会超出浮点值的范围，甚至导致溢出。因为我们的游戏世界并没有边界（游戏不会有终点），所以让赛车停留在原地，移动赛道。

自然，需要一些变量来控制赛道。添加如下代码：

```
public const float RoadLocationLeft = 2.5f;
public const float RoadLocationRight = -2.5f;
private const float RoadSize = 100.0f;
private const float MaxRoadSpeed = 250.0f;
private const float RoadSpeedIncrement = 0.5f;
private float RoadDepth0 = 0.0f;
private float RoadDepth1 = -100.0f;
private float RoadSpeed = 30.0f;
```

作为mesh的赛道模型是已知的，长宽各为100个单位。RoadSize常量就是赛道的长度，两个location常量标记了赛道两边的中点。最后两个常量用来控制游戏操作。最大速度让游戏每秒移动250个单位，每次加速多移动0.5个单位。

最后，设置两段赛道的深度。把地一段赛道设置为0，第二段紧跟着上一段赛道。添加绘制赛道的代码，使用这几个变量来绘制赛道。在BeginScene方法之后添加如下代码：

```
DrawRoad(0.0f,0.0f,RoadDepth0);
DrawRoad(0.0f,0.0f,RoadDepth1);
```

现在运行程序，可以看到已经正确的绘制了赛道，但是这条沥青的赛道看起来极度可怕。这种结果是由Direct3渲染计算像素

的方式引起的。当一个texel要覆盖屏幕中的多个像素时，这些像素需要通过一个放大过滤器来补偿（magnify filter to compensate）。当几个texel需要被绘制为一个像素时，他们会通过一个缩小过滤器。两种情况下的默认过滤器是一个名为Point的过滤器，它将会使用texel最接近的颜色作为像素的颜色，因此导致了这种情况。

有很多种方法来过滤纹理，但是，device不一定支持。我们只需要一个可以在texel之间插值计算，让赛道纹理看起来比较平滑的过滤器就可以了。在OnDeviceReset方法里添加如下代码：

详见private void OnDeviceReset(object sender,EventArgs e)中的代码

如你所见，先检查设备在放大（magnification）和缩小（minification）上是否支持各向异性（anisotropic）的过滤器。如果可以，就使用它。不行的话，再检测是否支持线性（linear）过滤器。如果两者都不可用，那么只能什么都不作，保留这种粗糙的效果。假设你的图形卡能支持其中一种过滤器，那么现在可以看到效果要好多了。

赛道以及处在了屏幕的中间，但还没有移动。还需要一个方法来更新游戏状态，完成移动赛道，进行碰撞检测。应该再OnPaint方法一开始就调用这个方法（再clear方法前）：

```
OnFrameUpdate();
```

以下则是这个方法的代码：

```
private void OnFrameUpdate(){详见源码}
```

整个游戏编写完之后会有很庞大的代码，但现在，我们所需的只是让路动起来而已。先忽略elapsedTime，这段代码所作的只是移动路面而已。最后还需要添加一个变量：

```
private float elapsedTime = 0.0f;
```

特别提示：

为什么需要使用时间呢？为了方便讨论，假设我们每一帧都把赛道移动相同的距离。也许在你的电脑上它运行的很完美，但在其他的系统上呢？找一台比你的系统配置低的系统运行看看吧，赛道看起来会运行的相当缓慢。同样换到配置较高的系统上，赛道又会移动的快很多。原因在于你的计算是基于帧速率（frame rate）。假设在你的系统上，每秒可以跑60帧，那么所有的计算过程都是依赖于这个静态的帧速率而来的。因此，在每秒可以跑40帧或80帧的系统中，自然会得到不同的计算结果。让你的程序在**每一个**系统下运行都得到同样的结果是我们的基本目标之一，因此无论如何都应该避免基于帧速率的计算。

解决这个问题一个比较好的方法就是根据时间来计算位移。比如，赛道的最大速度定义为每秒250个单位。首先，我们需要获得自上一次“更新”过后过去的时间间隔。.net运行时内建的一个属性（tick count）可以用来获得系统的tick count。但它它并不完美：这个计时器的精度太低。它的值大约每15毫秒才更新一次，因此，在一个高帧速率的系统中（每秒60帧以上），赛道的移动将是不连续的，因为所用的时间不是平滑的。

如果你的系统支持的话，在DirectX SDK包含了一个高精度（通常精度为1毫秒）的计时器类DirectXTimer。但如何你的系统不支持，那么则只能使用tick count了。本书都将使用这个计时器来计算时间。(注：这里的DirectXTimer实际上是作者通过P/Invoke自己实现的一个计时器，代码在Utility.cs文件中，书上没有具体讲解实现方法，但大家应该都能看明白吧^_^)

为场景添加一辆可移动的赛车吧

好了，现在已经有了渲染好的、并且可以沿着场景移动的赛道了，接下来应该添加实际与玩家交互的对象了：一辆赛车。可以简单的再添加赛道的那个主要的类里加上一些关于赛车的变量和常量就可以了，但这样的代码将不是模块化的。你应该把关于赛车的代码分离出来，成为一个独立的类。为工程添加一个名为“Car”的新类吧。

Car类应该完成些什么任务呢？因为当其他物体移动的使用它仍然是静止不动的，不需要向前，也不需要向后。但为了让赛车能躲避路上的障碍物，它应该能够左右移动，同样，它还需要能渲染自身。好了，有了这些信息，就可以为类添加成员了：

详见源码

这些变量已经足够用于控制赛车了。Height和depth都为静态的常量。赛车向两旁的移动速度的增量也是常量。使用最后一个常量的原因是赛车模型的大小刚好比赛道大，所以需要把它缩小一点点。

其他的成员基本上一看名字就知道它的用途了。有赛车当前的位置数据，默认情况下赛车位于赛道的左边。赛车的直径（Diameter），稍后会使用它来进行碰撞检测。有赛车的侧滑速度。当然，还有用来检测赛车在向哪个方向移动的两个布尔变量。最后，是有关mesh的变量。

Car类的构造函数需要完成两个任务：创建mesh对象（包括与它相关的结构）以及计算赛车的直径。添加一下构造函数：

public Car(){详见源码}

创建car mesh的方法和创建road mesh的方法基本上一样。接下来计算直径的新的代码则是比较有趣的。这里实际上是在计算赛车的边界球体（bounding sphere，mesh的所有的顶点都包含在这个球体内）。Geometry类包含了这个方法，只要把需要计算边界的顶点作为参数传给这个方法就可以了。

这里所需的就是从mesh获得顶点。你已经知道顶点保存在顶点缓冲内的，因此直接使用这块顶点缓冲。为了读取顶点缓冲中的数据，必须调用lock方法。在下一章中，会学到更多来自于VertexBuffer类的lock方法重载。现在，只需要知道这个方法会使用一个流返回所有顶点数据。还可以使用ComputeBoundingSphere方法获得这个mesh的“中心”以及边界球体的半径。因为我们并不需要关心mesh的中心，所以只需要把半径乘2获得直径就可以了。但是，模型经过了缩放，所以直径也需要缩放同样的比例。最后（在必不可少的finally块中），确定解锁并且释放了顶点缓冲。

接下来，添加绘制赛车的方法。Car类已经保存了赛车的位置，只需要获得device对象就可以绘图了。这个方法几乎和DrawRoad方法一样，区别在于变量不同以及在变换前需要缩放mesh，添加如下代码：

public void DrawCar(Device device) {详见源码}

在使用Car类之前，还需要让外部可以访问类的私有成员，添加如下公共属性：

{详见源码}

现在，应该在主要的游戏引擎类里添加成员来使用Car类了。在DogerGame类里添加如下代码：

private Car car = null;

由于car类的构造函数需要device作为变量才能初始化，所以只有在创建了device之后才能调用它。在OnDeviceReset方法里创建car是个不错的主意，在创建了road mesh之后添加如下代码：

car = new Car(device);

创建了赛车之后，就可以更新渲染部分的代码了。在OnPaint中两个DrawRoad方法之后添加以下代码：

car.Draw(device);

可以看到，已经在路上正确的渲染了赛车。可是，如何才能控制赛车左右移动呢？先忽略鼠标的存在，假设玩家拥有键盘，并且将使用键盘来控制游戏。使用键盘上的4个方法键来控制游戏是不错的选择。重载OnKeyDown方法：

protected override void OnKyeDown(KeyEventArgs e) {详见源码}

这里没有什么特别的内容。如果按下了ESC则游戏结束同时关闭窗口。按下左键或者右键，则把相应的moving变量设置为true，另一个则设为false。现在运行程序，按下按键可以正确更新赛车的两个moving变量。但赛车本身并不会移动，还需要为赛车添加一个函数更新它的位置：

public void Update(float elapsedTime)

这个方法接受逝去的时间值作为参数，所以无论在任何系统上，都会得到相同的结果。这个方法本身很简单，哪一个moving变量的值为true，则向那个方向移动移动相应的距离（根据所经过的时间长短）接下利检查是否已经移动到了边界，如果是的话则完成移动。但是，这个方法是不会自己调用自己的，还需要更新OnFrameUpdate方法，加入以下代码：

car.Update(elapsedTime);

（注：如果你是按着教程一步一步来，没有偷看最后源码的话，会发现此时赛车根本不会移动，郁闷吧，呵呵，原因是根本没有启动计时器。在初始化图形设备的InitializeGraphics()方法中加上如下代码吧 Utility.Timer(DirectXTimer.Start); ）

添加障碍物

恭喜，这就是你创建的第一个3D互动程序了。已经完成了模拟赛车的移动。虽然实际上是赛道在移动，但显出的效果确实是赛车在移动。至此，游戏已经完成大半。接下来是添加障碍物的时候了。与添加Car类一样，添加一个名为Obstacle的类。

我们将使用不同颜色形状的mesh作为障碍物。通过mesh类创建stock对象可以改变mesh的类型，同时，使用材质来改变障碍物的颜色。添加如下的变量和常量：

{详见源码}

第一个常量表示将会有5种不同类型的mesh（球体、立方体、圆环、圆柱以及茶壶）。其中大多数的物体都有一个长度或半径的参数。我们希望所有障碍物都有同样的尺寸，所以应该把这些参数都设置为常量。很多种mesh类型都有一个而外的参数可以控制

mesh中的三角形数量（stacks,slices,rings等等）。最后一个常量就是用来控制这些参数的。可以增大或减小这个参数来控制mesh的细节。

接下来的color数组用来控制mesh的颜色。我只是随即的选择了一些颜色而已，也可以把它们改为任何你喜欢的颜色。应该注意到这个类里既没有任何的材质数组，也没有纹理数组。你应该知道默认的mesh类型只包含了一个没有材质和纹理的子集，因此，额外的信息是不需要的。

由于障碍物需要放置在路面之上，并且实际上是路在移动，所以必须保证它们是和路面同时移动的。需要position属性来保证在路面移动时障碍物会同时更新。最后由于在创建茶壶时不能控制它的大小，需要检查创建的是否为茶壶，平且对它进行相应的缩放。为Obstacle类添加如下构造函数：

注意到这里我们使用了来自utility的Rnd属性。它的具体实现非常简单位于utitity.cs文件中，只是用来返回一个随即的时间而已。Obstacle默认的构造函数保存了障碍物的默认位置，而且默认的为一个“非茶壶的”mesh。接下来选择创建某个类型的mesh。最后，选择一个随机的颜色作为材质颜色。

在把障碍物添加到游戏引擎之前，还有一些额外的工作需要完成。首先，添加一个方法来和赛道同步更新障碍物的位置。，添加如下代码：

```
public void Update(float elapsedTime,float speed)
```

再一次使用elapsed time作为参数来保证程序在任何系统都能正常工作。同时，把当前赛道的速度也作为参数，这样物体就好像是“放置”在赛道上一样。接下来，还需要一个方法渲染障碍物：

```
public void Draw(Device device) {详见源码}
```

因为茶壶没有经过正确的缩放，因此如果渲染的是茶壶，那么应该先对他进行缩放，再移动到正确的位置。之后，设置材质颜色，把纹理设置为null，绘制mesh。

显然，同一时间赛道上需要多个障碍物。你需要一个方法来简单的在游戏引擎里添加或者移除障碍物。使用数组是一个可行的方法，却不是最好的：数组不能重置大小。集合是一个不错的选择，为obstacles添加一个集合来储存障碍物：

```
public class Obstacles : Ienumerable {详见源码}
```

当然，别忘了对System.Collections名称空间的引用。这个类包含了可以直接访问集合成员的索引器，可以让foreach方法正确工作的迭代器，以及三个值得注意的方法：add,remove以及clear。obstacle文件有了这些基本的方法之后，可以为游戏引擎添加障碍物了。

首先，需要一个变量来储存当前场景里的障碍物。为DodgerGame类添加如下变量：

```
private Obstacles obstacles;
```

接下来，需要一个方法用新的障碍物填充即将出现的一段赛道，添加如下代码:

```
private void AddObstacles(float minDepth) {详见源码}
```

这个方法是把障碍物添加到游戏的起点。首先，计算需要添加到这段赛道的障碍物数量。同时，还必须保证在障碍物之间有足够的距离让赛车躲避，否则，对玩家而言很不公平。接下来，就把障碍物随机的添加到路上。同时，把它添加到当前的obstacles集合中。注意到在创建障碍物时使用了一个名为ObstaclesHeight的常量，以下是它的声明：

在障碍物出现在场景之前，还有3件事要做：you need to add a call into our obstacle addition method somewhere,你需要取保为场景中的每一个障碍物都调用了update方法，你最后还需要渲染障碍物。因为在开始游戏前，需要把所有成员变量都重置为默认状态。是添加一个新方法的时候了，使用这个方法来初始化AddObstacles。添加如下代码：

```
private void LoadDefaultFameOptions()
{
    RoadDepth0 = 0.0f;
    RoadDepth1 = -100.0f;
    RoadSpeed = 30.0f;
    car.Location = RoadLocationLeft;
    car.Speed = 10.0f;
    car.IsMovingLeft = false;
```

```
        car.IsMovingRight = false;
        foreach(Obstacle o in obstacles)
        {
            o.Dispose();
        }
        obstacles.Clear();
        AddObstaxles(RoadDepth1);
        Utility.Timer(DirectXTimer.Start);
    }
```

这个方法重置了大量我们关心的成员变量。同时依次对集合中的对象进行dispose操作，并且在重新填充集合之前删除所有元素。最后，启动计时器。应该在InitializeGraphics方法中创建了device之后的地方调用这个方法。千万**不要**把这个方法添加到OnDeviceReset方法中；只需要在每次游戏开始的时候调用一次就足够了。

```
    LoadDefaultGameOptions();
```

现在需要在OnFrameUpdate方法中添加一个方法来更新障碍物。因为每一帧都需要更新所有障碍物，所以应该迭代他们。在OnFrameUpdate方法中，car.Update(elapsedTime)之前，添加如下代码：

```
    foreach(Obstacle o in obstacles)
    {
        o.Update(elapsedTime,roadSpeed);
    }
```

把障碍物添加到游戏引擎的一步就是渲染他们了。在OnPaint方法中，紧跟在绘制赛车的代码之后，添加我们熟悉的方法来渲染障碍物：

```
    foreach(Obstacle o in obstacles)
    {
        o.Draw(device);
    }
```

尝试着运行程序吧！(注：哈哈，程序抛出异常了吧，记住，还要在OnDeviceReset的最后添加 obstacles = new Obstacles();)。当你沿赛道行驶了一段距离，避开了几个障碍物之后，发生了什么？看起来在避开了最初的几个障碍物之后，就不在有新的障碍物出现了。回想一下先前的代码，你只是在第一段赛道添加了障碍物，之后，不停的移动这段赛道。但对于新的“赛道段”，你并没有调用任何方法填充障碍物。在添加新赛道段的地方添加如下代码吧：

```
    if(roadDepth0 > 75.0f)
    {
        roadDepth0 = roadDepth1 - 100.0f;
        AddObstacles(roadDepth0);
    }
    if(roadDepth1 > 75.0f)
    {
        roadDepth1 = roadDepth0 - 100.0f;
        AddObstacles(roadDepth1);
    }
```

不错，现在看起来好多了。你获得了一辆在赛道上缓缓行驶并且可以穿越（至少现在可以）障碍物的赛车。障碍物看起来有些呆板。应该让他动起来，让障碍物在赛车经过的时候旋转起来。首先，需要添加一些新的成员变量让obstacle类能控制旋转：

```
    private float rotation = 0;
    private float rotationspeed = 0.0f;
    private Vector3 rotationVector;
```

障碍物的旋转速度和转轴都应该是随机的，这样他们才会看起来与众不同。在obstacle类构造函数的最后添加以下两行代码就可以实现这个功能：

```
rotationspeed = (float)Utility.Rnd.NextDouble() * (float)Math.PI;
rotationVector = new
Vector3((float)Utility.Rnd.NextDouble(),(float)Utility.Rnd.NextDouble(),(float)Utility.Rnd.NextDouble());
```

为了使障碍物旋转起来，还有两个需要修改的地方。首先，需要把实现旋转的代码添加到Update函数中：

```
Rotation += (rotationspeed * elapsedTime);
```

这里没什么特别的，只是根据时间和随机的旋转速度来增加旋转角度而已。最后，真正改变通过world transform来实现旋转，这样，渲染出来的物体才是旋转的。更新以下代码：

```
if(isTeapot)
{
    device.Transform.World = Matrix.RotationAxis(rotationVector,rotation)
    *Matrix.Scaling(ObjectRadius,ObjectRadius,ObjectRadius)*Matrix.Translation(position);
}
else
{
    device.Transform.World = Matrix.RotationAxis(rotationVector,rotation) * Matrix.Translation(position);
}
```

再次运行程序，可以看到当赛车经过的时候，障碍物已经在随机旋转了。下一步该干什么了呢？Well，让赛车真正的能和障碍物碰撞将会是很酷的，同时可以记录下总分来，看看你到底开了多远。当添加计分系统的同时，你也应该实时的记录游戏状态。在游戏引擎的DodgerGame类中，添加如下变量：

```
private bool isGameOver = true;
private int fameOverTick = 0;
private bool hasGameStarted = false;
private int score = 0;
```

所有关于游戏的信息都储存在这里。你可以知道游戏是否结束了，是否是第一次开始游戏，上一次和当前的游戏得分。这些都是不错的特性，不过怎么才能实现他呢？首先从计分系统开始吧，毕竟这才是玩家最关心的内容。玩家每经过了一个障碍物，就获得一定的分数。当然，你也希望游戏更具挑战性：加快赛道的速度，这样障碍物也会来到更快。另外，很重要的一点是：在LoadDefaultGameOptions方法中添加一行代码来重置总分，这样在新游戏开始的时候玩家才不会获得额外的分数。

```
sorce = 0;
```

接下来，在OnFrameUpdate方法中，在移动障碍物之前，添加如下代码：

```
Obstacles removeObstacles = new Obstacles();
foreach(Obstacle o in obstacles)
{
    if(o.Depth > car.Diameter - (Car.Depth * 2))
    {
        removeObstacles.Add(o);
        roadSpeed += RoadSpeedIncrement;
        if(roadSpeed >= MaxRoadSpeed)
        {
            roadSpeed = MaxRoadSpeed;
        }
        car.IncrementSpeed();
        score += (int)(roadSpeed * (roadSpeed / car.Speed));
```

```
            }
        }
        foreach(Obstacle o in removeObstacles)
        {
            obstacles.Remove(o);
            o.Dispose();
        }
        removeObstacles.Clear();
```

使用这段代码获得了玩家已经经过的障碍物的列表（这个“表”同一时间只应该包含一个元素）。列表中每次增加一个障碍物，就相应的增加总分，增加赛道的速度提升难度，增加赛车的速度（虽然赛车的速度没有赛道增加的快）。完成了这些操作之后，把障碍物从列表中移除。注意观察，我们还使用了一个公式来根据赛道的速度计算得分，因此，开的越远，得分就越多。同时你可能已经注意到我们使用了一个还没有实现的方法来增加赛车的速度。不用多说，添加代码：

```
    public void IncrementSpeed()
    {
        carSpeed += SpeedIncrement;
    }
```

现在，需要添加一个新的方法来判断赛车是否撞到了障碍物。在obstacle类中，添加如下代码：

```
    public bool IsHittingCar(float carLocation,float carDiameter)
    {
        if(position.Z > (Car.Depth - (carDiameter / 2.0f)))
        {
            if((carLocation < 0) && (position.X < 0))
                return true;
            if((carLocation > 0) && (position.X > 0))
                return true;
        }
        return false;
    }
```

这里都是些很简单的东西；首先检查赛车和障碍物的深度（depth）是否相同，能否发生碰撞，如果相同，并且位于路的同一边，那么返回ture，否则赛车和障碍物不会碰撞，，返回false。有了这些代码，就可以在游戏引擎中实现碰撞检测了。更新OnFrameUpdate方法：

```
    foreach(Obstacle o in obstacles)
    {
        o.Update(elapsedTime,roadSpeed);
        if(o.IsHittingCar(car.Location,car.Diameter))
        {
            isGameOver = true;
            gameOverTick = System.Environment.TickCount;
            Utility.Timer(DirectXTimer.Stop);
        }
    }
```

每次更新了障碍物之后，检查是否发生了碰撞。如果发生了，那么游戏结束。设置游戏状态，并且停止计时器。

最后一步

至今为止，我们还没有使用过那些状态变量。你应该先完成游戏的逻辑设计。你将要求玩家通过按下任意键来启动游戏。游戏结束之后，将会有一瞬间停顿（大约一秒左右），接下来再次按下任意键就可以重新启动游戏。你首先要确定的是一旦游戏结束了，其它状态就不应该再更新了。因此，OnFrameUpdate方法的第一行应该是这样的：

```
if((isGameOver) || (!hasGameStarted))
    return;
```

接下来解决通过按下任意键启动游戏，在OnKeyDown方法重载的最后一行，添加如下的逻辑部分：

```
if(isGameOver)
{
    LoadDefaultGameOptions();
}
isGameOver = false;
haGameStarted = true;
```

好了，这就是我们所要求的行为。当游戏结束了，玩家按下任意键，一个恢复为默认设置的新游戏就开始了。如果你愿意，可以从InitializeGraphics方法中删除LoadDefaultGameOptions的调用了，因为在每次按下任意键启动游戏的时候就会调用它。但是，我们还没有添加碰撞后让画面短暂停留一瞬间的代码。同样也在OnKeyDown方法中来实现；而且因该在检查是否按下了ESC键之后的添加代码：

这将会在游戏结束后的一秒之内忽略所有击键（除了可以按下ESC退出游戏）。现在可以尝试着玩玩我们的游戏了！虽然它们并不完整。我们记录了得分，却并没有显式的把这个得分告诉玩家。现在来完善这一步吧。Direct3D名称空间下有一个称为Font的类可以用来绘制文本。注意，在System.Drawing名称空间里也有一个Font类，而且如果没有前缀修饰的情况下使用“Font”，那个这两个类会发生冲突。幸运的是，可以使用using语句来做如下申明：

```
Using Driect3D = Microsoft.DirectX.Direct3D;
```

你所创建的每个字可以是不同的颜色，但最好使用相同的大小以及字体。对这个游戏来说，你需要两种不同的文本类型，自然，也需要2种不同的字体。为DodgerGame类添加如下变量：

```
private Direct3D.Font scoreFont = null;
private Direct3D.Font gameFont = null;
```

你只有在创建了device之后才能初始化这些变量。在创建了device之后的地方添加代码。它们并不需要在OnDeviceReset事件中初始化，应为这些类会自动处理重置设备时的事件。在InitializeGraphics的最后一行添加如下代码：

```
scoreFont = new Microsoft.DirectX.Direct3D.Font(device, new System.Drawing.Font("Arial",12.0f,FontStyle.Bold));
gameFont = new Microsoft.DirectX.Direct3D.Font(device, new System.Drawing.Font("Arial",36.0f,FontStyle.Bold | FontStyle.Italic));
```

这里创建了两种不同大小的Arial字体。之后，更新进行渲染的方法来绘制字体。字体是最后才需要绘制的内容，因此在绘制赛车的代码之后添加如下代码：

```
if(hasGameStarted)
{
    scoreFont.DrawText(null,string.Format("Current score:{0}",score),
    new Rectangle(5,5,0,0), DrawTextFormat.NoClip,Color.Yellow);
}
if(isGameOver)
{
    if(hasGameStarted)
    {
        gameFont.DrawText(null,"You crashed. The game is over.",
        new Rectangle(25,45,0,0),DrawTextFormat.NoClip,Color.Pink);
    }
```

```
        if((System.Environment.TickCount - gameOverTick) >= 1000)
        {
            gameFont.DrawText(null,"Press any key to begin.", new Rectangle(25,100,0,0),
            DrawTextFormat.NoClip, Color.WhiteSmoke);
        }
    }
```

我们将在后面的章节里详细讨论DrawText方法。现在只需要知道它能完成他的名字所表示的功能。好了，现在可以看到当游戏已开始，就可以看到当前的分数了。除此而外，当游戏结束之后，你告诉玩家他撞车了。最后，游戏结束了一秒钟之后，提醒玩家按任意键可以可以重新开始游戏。

Wow，至今为止，你已经完成了一个完整的游戏了。试玩一下吧。还有什么遗漏的吗？把最高分保存起来将是不错的尝试^_^。

（注：以下部分作者演示了如何把最高分和玩家的姓名作为一个结构保存到注册表中，由于大部分是代码，且这部分内容基本与图形无关，就不翻译了）

First off, we will need a place to store the information for the high scores. We will only really care about the name of the player as well as the score they achieved, so we can create a simple structure for this. Add this into your main games namespace:

```
    public struct HighScore
    {
        private int realScore;
        private string playerName;
        public int Score { get { return realScore; } set { realScore = value; } }
        public string Name { get { return playerName; } set {
        playerName = value; } }
    }
```

Now we will also need to maintain the list of high scores in our game engine. We will only maintain the top three scores, so we can use an array to store these. Add the following declarations to our game engine:

```
    Private HighScore[] highScores = new HighScore[3];
    private string defaultHighScoreName = string.Empty;
```

All we need now is three separate functions. The first will check the current score to see if it qualifies for inclusion into the high score list. The next one will save the high score information into the registry, while the last one will load it back from the registry. Add these methods to the game engine:

```
    private void CheckHighScore()
    {
        int index = -1;
        for (int i = highScores.Length - 1; i >= 0; i--)
        {
            if (score >= highScores[i].Score) // We beat this score
        {
           index = i;
        }
     }

   // We beat the score if index is greater than 0
   if (index >= 0)
```

```
    {
        for (int i = highScores.Length - 1; i > index ; i--)
        {
            // Move each existing score down one
            highScores[i] = highScores[i-1];
        }
        highScores[index].Score = score;
        highScores[index].Name = Input.InputBox("You got a highscore!!","Please enter your name.",
defaultHighScoreName);
    }
}

     private void LoadHighScores()
     {
          Microsoft.Win32.RegistryKey key
=Microsoft.Win32.Registry.LocalMachine.CreateSubKey("Software\\MDXBoox\\Dodger");
          try
          {
               for(int i = 0; i < highScores.Length; i++)
               {
                    highScores[i].Name = (string)key.GetValue(string.Format("Player{0}", i),  string.Empty);
                    highScores[i].Score = (int)key.GetValue(string.Format("Score{0}", i),  0);
               }
               defaultHighScoreName = (string)key.GetValue("PlayerName", System.Environment.UserName);
          }
          finally
          {
               if (key != null)
               {
                    key.Close(); // Make sure to close the key

               }
     }

     /// <summary>
     /// Save all the high score information to the registry
     /// </summary>
     public void SaveHighScores()
     {
          Microsoft.Win32.RegistryKey key =
Microsoft.Win32.Registry.LocalMachine.CreateSubKey("Software\\MDXBoox\\Dodger");
          try
          {
               for(int i = 0; i < highScores.Length; i++)
```

```
            {
                key.SetValue(string.Format("Player{0}", i),highScores[i].Name);
                key.SetValue(string.Format("Score{0}", i),highScores[i].Score);
            }
            key.SetValue("PlayerName", defaultHighScoreName);
        }
        finally
        {
            if (key != null)
                key.Close(); // Make sure to close the key
        }
    }
```

I won't delve too much into these functions since they deal mainly with built-in .NET classes and have really nothing to do with the Managed DirectX code. However, it is important to show where these methods get called from our game engine.

The check for the high scores should happen as soon as the game is over. Replace the code in OnFrameUpdate that checks if the car hits an obstacle with the following:

```
if (o.IsHittingCar(car.Location, car.Diameter))
{
    isGameOver = true;
    gameOverTick = System.Environment.TickCount;
    Utility.Timer(DirectXTimer.Stop);
    CheckHighScore();
}
```

You can load the high scores at the end of the constructor for the main game engine. You might notice that the save method is public (while the others were private). This is because we will call this method in our main method. Replace the main method with the following code:

```
using (DodgerGame frm = new DodgerGame())
{
    frm.Show();
    frm.InitializeGraphics();
    Application.Run(frm);
   frm.SaveHighScores();
}
```

The last thing we need to do is actually show the player the list of high scores. We will add this into our rendering method. Right before we call the end method on our game font, add this section of code to render our high scores:

```
gameFont.DrawText(null, "High Scores: ", new Rectangle(25,155,0,0),
DrawTextFormat.NoClip, Color.CornflowerBlue);
for (int i = 0; i < highScores.Length; i++)
{
    gameFont.DrawText(null, string.Format("Player: {0} : {1}",highScores[i].Name, highScores[i].Score), new
Rectangle(25,210 + (i * 55),0,0), DrawTextFormat.NoClip,
            Color.CornflowerBlue);
```

```
}
```

再花一点时间来复习一下所完成的工作吧，这是你的第一个游戏。