DirectX: Eine Einführung

DirectX wurde 1995 erstmals von Microsoft unter dem Namen The Games SDK vorgestellt. Davor (zu DOS Zeiten) musste man umständlich auf Assembler zurückgreifen, wenn man irgendwas mit Grafik programmieren wollte. DirectX vereinfachte dies extrem und wurde von Version zu Version immer weiter entwickelt und auch immer besser. Derzeit stehen wir bei Version DirectX 9. DirectX ist eigentlich eine Sammlung von mehreren APIs. Diese sind DirectInput um Daten von Tastaturen, Mäuse, Joysticks etc. zu empfangen und zu verarbeiten, DirectSound/Music um Sounds/Musik abzuspielen, DirectPlay um Netzwerkprogramme wie Multiplayerspiele zu schreiben, DirectShow um Videos abzuspielen und zu guter Letzt Direct3D um auf dem Bildschirm zu zeichnen. Daneben gibt es noch DirectDraw, mit dem man ausschließlich 2D Objekte zeichnen kann(mit Direct3D kann man sowohl 3D als auch 2D Objekte zeichnen) und DirectSetup, ein Setup Generationsprogramm speziell für DirectX Applikationen. Dieses Tutorial befasst sich also nicht mit dem gesamten DirectX Multimedia Packet, sondern "nur" mit Direct3D. Es muss natürlich gesagt werden, dass jeder Unterbereich von DirectX schon allein ziemlich groß ist. So, nun wollen wir mal sehen, welche Möglichkeiten uns Direct3D bietet. Ebenso werden die nächsten Kapitel ziemlich Theorie beinhalten, mit nur relativ wenig Code. Aber durchhalten. Wir werden bald zu unseren ersten Zeichenoperationen kommen...

Direct3D: A closer look

Voll DirectX 9 kompatibel. Damit wird seit längerer Zeit von vielen Grafikkartenherstellern geworben. Die Grafikkartenhersteller meinen damit natürlich Direct3D (DirectInput auf der Grafikkarte? ^^).
Aber was macht Direct3D eigentlich genau?
Direct3D bietet uns die Möglichkeit auf dem Bildschirm Objekte zu zeichnen. Denkt einfach mal an ein Spiel wie Far Cry. Der ganze tolle Dschungel dort wird mit Direct3D gezeichnet (achja, wir haben noch einen langen Weg vor uns, bis wir sowas wie Far Cry machen können).
Aber wie sind diese Objekte jetzt aufgebaut? Die Antwort ist einfach: Aus Dreiecken. Denn aus Dreiecken kann man jedes beliebige Objekt zusammenstellen. Nehmt einfach einen Zettel und malt ein Rechteck drauf. Wie ihr vielleicht sehen könnt, besteht ein Rechteck aus zwei sich gegenüberliegenden rechtwinkeligen Dreiecken. Eine solche Aufteilung in Dreiecken lässt sich praktisch bei jedem Objekt machen. D.h. wir können also Dreiecke mit Direct3D zeichen (man kann aber auch Punkte und Linien zeichnen, aber am öftesten verwendet man Dreieckte). Aber Direct3D kann noch mehr. Es kann diese Dreiecke nun texturieren. Normalerweise kann man einem Dreieck nur eine Farbe zuweisen(blau, gelb, rot...), aber das ist natürlich langweilig. Deshalb erfand man eine Technik namens Texturieren d.h. dass man einfach ein Bild über das Dreieck legt. Damit sehen Objekte viel realitätsnäher aus. Des Weiteren kann man mit Direct3D auch Licht erzeugen (Lights). Das ist allerdings ziemlich komplex, daher werden wir das erst später besprechen. Zum Schluss kann Direct3D Dreiecke noch verschieben, vergrößern und verkleinern. Aber das werden wir auch noch später genauer kennen lernen.
So, jetzt haben wir uns mal einen groben Überblick über Direct3D verschafft. Jetzt

können wir näher auf die einzelnen Bereiche eingehen. Zuerst sehen wir mal, wie wir Direct3D dazu bringen, dass es uns erlaubt irgendwas zu zeichnen.

Direct3D: Erste Schritte

Überlegen wir uns erstmal, was wir in diesem Kapitel erreichen wollen. Wir wollen ganz einfach ein Fenster erzeugen, das Blau(oder in irgendeiner anderen Farbe) gefärbt ist. Das ist sozusagen das "Hello World" von DirectX.
Was braucht man dazu? Ich sage mal ganz einfach, dass man mit einer IDE arbeiten sollte. Warum? Weil es das Leben vereinfacht. Ich selbst nutze Visual Studio .net 2005. Als nächstes braucht man die Standardtools, um mit C# programmieren zu können. Zu guter Letzt braucht man noch das so genannte DirectX SDK[1]. Das beinhaltet alles, was man braucht um mit DirectX zu programmieren. Nachdem das SDK installiert wurde, können wir weitermachen (irgendwie logisch...).
Zunächst referenzieren wir auf folgende DLLs:
System.dll
System.Drawing.dll
System.Windows.Forms.dll
Microsoft.DirectX.dll
Microsoft.DirectX.Direct3D.dll
Microsoft.DirectX.Direct3DX.dll
Nun binden wir folgende Namespaces ein, die eigentlich selbsterklärend sind:

```
using System;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
```

Für unsere erste DirectX Anwendung brauchen wir nur eine einzige Klasse. Ich nenne sie *MDXSampleApp.* Diese Klasse erbt, wie eine „normale" Windows Forms Anwendungen, von *System.Windows.Forms*, aber hat sonst kaum Ähnlichkeiten zu Windows Forms.
Unsere erste Anwendung kommt auch mit nur einer einzigen Variablen aus. Das DirextX Device:

```
public class MDXSampleApp : System.Windows.Forms.Form
        {
                private Device m_Device;
```

Ein Device ist die elementare Grundverbindung zwischen unserer Anwendung und Direct3D. Mit einem Device können wir alle oben genannten Dinge machen. Nun Implementieren wir eine Funktion *InitGfx()*, die uns dieses Device erstellt.

Die ersten Zeilen dieser Funktion sollten kein Problem darstellen:

```
public void InitGfx()
{
        try
        {
                this.ClientSize = new Size(800,600);
                this.Text = "Direct3D Tutorial 1";

                this.KeyPress += new
        KeyPressEventHandler(DXSampleApp_KeyPress);
```

Wir legen zuerst die Größe des Fensters fest, danach einen Fenstertitel und registrieren noch einen Eventhandler für das *KeyPress* Ereignis.

Kommen wir nun zu den PresentParameters.
Die PresentParamter (Präsentationsparameter) bestimmen wie sich das Device grundlegend verhalten. Für unsere Applikation brauchen wir zunächst nur diese beiden Parameter, es gibt aber noch viele mehr. Der erste Paramter zeigt an, dass wir eine Fensteranwendung haben wollen. Der Zweite gibt an, wie wir die Bilder auf den Bildschirm bringen wollen. Copy bedeutet einfach, dass wir immer den gesamten Bildschirm neu zeichnen wollen.

```
PresentParameters pp = new PresentParameters();

pp.Windowed = true;
pp.SwapEffect = SwapEffect.Copy;
```

Nun erstellen wir das Device:

```
m_Device = new Device(Manager.Adapters.Default.Adapter,
                      DeviceType.Hardware,
                      this,
                      CreateFlags.HardwareVertexProcessing,
                      pp);
```

Ok, gehen wir Parameter für Parameter durch:

Manager.Adapters.Default.Adapter: Gibt an, auf welchem Adapter (=Bildschirm) wir zeichnen wollen. Normalerweise haben die meisten Systeme einen Bildschirm. Und diesen Bildschirm repräsentiert Manager.Adapters.Default.Adapter.
DeviceType.Hardware: Das bedeutet, dass für alle möglichen Operationen von Direct3D die Grafikhardware verwendet wird. Man kann auch bestimmen, dass das die CPU(Software) machen soll, was aber ziemlich langsam ist.
this: Zeigt einfach an, dass das Devie dieses Fenster zum Zeichnen verwenden soll.
CreateFlags.HardwareVertexProcessing: Besagt ebenfalls grundsätzlich dasselbe wie DeviceType.Hardware, aber ist trotzdem ein wenig anders, außerdem gibts da noch ein paar andere Paramter, die aber für uns(derzeit) nicht interessant sind. Ich werde später noch genauer darauf eingehen.
pp: Zeigt auf die ausgefüllte PresentParameters Struktur.

Und jetzt noch der Rest der Funktion (falls irgendein Fehler auftreten solle fangen wir diesen durch einen try-catch Block ab und zeigen den Fehler an)

```
        }
        catch(DirectXException e)
        {
                MessageBox.Show(e.Message);
        }
}
```

Gut. Nun kommen wir zum Zeichnen. Das wird als Rendern bezeichnet. Deshalb implementieren wir einfach mal eine Funktion namens Render.

```
public void Render()
{
        m_Device.Clear(ClearFlags.Target,Color.Blue,0.0f,0);
        m_Device.BeginScene();
        // rendern
        m_Device.EndScene();
        m_Device.Present();

}
```

Der erste Funktionsaufruf ist nicht sonderlich interessant (derzeit...) und besagt, dass das Fenster (Target) mit einem blauen Hintergrund gerendert werden soll.
Danach kommen wir zum eigentlichen rendern. Das rendern geschieht **immer** zwischen einem BeginScene und einem EndScene. Doch das werden wir später auch noch genauer besprechen. Das Present veranlsst Direct3D nun, alles auch wirkich zu rendern.
Diese Renderfunktion wird von der Main aus ständig aufgerufen und zeichnet nun also unser Fenster.

Nun brauchen wir noch eine Funktion, die Direct3D herunterfährt und noch die Main Funktion. Schließlich implementieren wir noch die OnKeyPress.

```
public void Shutdown()
{
        m_Device.Dispose();
}



[STAThread]
static void Main()
{
        MDXSampleApp example = new MDXSampleApp();
        example.InitGfx();
        example.Show();

        while(example.Created)
        {
                example.Render();
                Application.DoEvents();
        }
        example.Shutdown();

}

private void OnKeyPress(object sender, KeyPressEventArgs e)
```

```
{
        if((int)e.KeyChar == (int)Keys.Escape)
                this.Close();
}
```

So, das war es. Insgesamt hatten wir nur rund 100 Zeilen Code. Im nächsten Teil werden wir uns das Rendern noch mal genauer ansehen.

Rendern von Dreiecken

Dieses Kapitel wird viele grundlegende Dinge beinhalten, also gut aufpassen.
Zunächst wollen wir einmal sehen, wie ein Dreieck aufgebaut ist: Ein Dreieck besteht aus 3 Eckpunkten, die miteinander verbunden werden. Und für jedes Dreieck, das wir zeichnen wollen, müssen wir diese 3 Eckpunkte definieren.
Aber wie müssen wir diese 3 Eckpunkte definieren, damit DirectX sie zeichnen kann? Nun, dass ist relativ einfach: Für Direct3D ist der Bildschirm nichts weiter als ein Koordinatensystem. Wir müssen nur die Koordinaten für die einzelnen Eckpunkte an Direct3D schicken und Direct3D zeichnet uns dann ein Dreieck.
Wie ist dieses Koordinatensystem nun aufgebaut?
Ich denke mal, dass jeder weiß, was ein (kartesisches) Koordinatensystem ist. Für Direct3D existieren 3 Achsen. Die x-Achse, die y-Achse und die z-Achse. Die x-Achse verläuft auf der Horizontalen, die y auf der Vertikalen. Die z-Achse geht in den Bildschirm hinein.
Da wir ja jetzt erstmal nur 2D Objekte rendern wollen, brauchen wir uns um die z-Achse nicht zu kümmern.
Zu sagen bleibt hierzu eigentlich nur noch, dass die englische Bezeichnung für Eckpunkt Vertex lautet, im Plural Vertices.
Um solche Vertices zu speichern, bietet uns Direct3D eine ganze Reihe von Strukturen. Diese sind alle im Namespace *Microsoft.DirectX.Direct3D.CustomVertex* zusammengefasst. In diesem Namespace sind ziemlich viele unterschiedliche Strukturen zum Speichern von Vertices, für uns ist jedoch nur die Struktur CustomVertex.TransformedColored (erstmal) interessant. Das Transformed bedeutet das wir 2D Koordinaten angeben wollen und das Color das wir jeden Eckpunkt eine Farbe zuweisen wollen. Wenn wir jetzt jeden Eckpunkt eine eigene Farbe zuweisen, wird der unmittelbare Bereich zu diesem Eckpunkt mit der Farbe gefärbt. An den Übergängen zwischen den einzelnen Farben, werden die Farben *interpoliert*. Beim interpolieren wird einfach ein Mittelwert zwischen den einzelnen Farben berechnet und dadurch kommt ein Farbverlauf zwischen den Farben zustande.
Jetzt müssen wir noch wissen, wie wir die Vertices speichern. Dazu müssen wir zunächst mal ein Array von CustomVertex.TransformedColored erstellen und mit unseren Daten befüllen. Dazu fügen wir unserer Klasse aus dem 1. Kapitel eine weitere private Variable *verts* hinzu:

```
private CustomVertex.TransformedColored[] verts;
```

Nun, diese Vertices liegen jetzt aber im Hauptspeicher (dem RAM). Wenn wir nun unser Dreieck rendern wollen, muss Direct3D diese erst über den AG Port (AGP) oder

den PCI Express Port zur Grafikkarte transportieren. Bei ein paar Vertices, wird sich das sicher nicht bemerkbar machen, aber wenn wir jetzt mehrere tausend Vertices haben? Deshalb wäre es ja von Vorteil, wenn die Vertices direkt auf dem Speicher der Grafikkarte liegen würden.(Video RAM). Der Vorteil des Video RAM ist, dass er eine extrem kurze Zugriffszeit hat und dass wir nicht bei jedem Rendervorgang, die Daten über den AG Port transportieren müssen. Ein Nachteil ist jedoch, dass der Video RAM von der Größe her relativ begrenzt und teuer ist. Aber darum müssen wir uns nicht kümmern.
Also wie speichern wir jetzt unsere Vertices im Video RAM? Ganz einfach: Mit einem VertexBuffer. Ein VertexBuffer ist nichts anderes als ein Speicherbereich im Video RAM, der unsere Vertices beinhaltet. Ein VertexBuffer wird durch die Klasse *Direct3D.VertexBuffer* repräsentiert.

```
private VertexBuffer m_VertexBuffer;
```

Nun müssen wir unserer Vertices noch definieren und nebenbei den VertexBuffer mit diesen Daten befüllen. Dies geschieht logischerweise in der *InitGfx* Methode. Zuerst definieren wir die Vertices:

```
verts = new CustomVertex.TransformedColored[3];

verts[0].X=150;verts[0].Y=50;verts[0].Z=0.5f; verts[0].Rhw=1;verts[0].Color
= Color.Yellow.ToArgb();

verts[1].X=250;verts[1].Y=250;verts[1].Z=0.5f;
verts[1].Rhw=1;verts[1].Color = Color.Bisque.ToArgb();
verts[2].X=50;verts[2].Y=250;verts[2].Z=0.5f; verts[2].Rhw=1;
;verts[2].Color = Color.White.ToArgb();
```

Wir sehen also, dass wir jedem Vertex eine x-Koordinate, eine y-Koordinate, eine z-Koordinate und eine Farbe zuweisen. Wie gesagt, müssen wir uns eigentlich nicht um die z-Koordinate kümmern, jedoch habe ich sie auf 0.5 gesetzt, da ein 0.0 bei manchen (älteren) Grafikkarten zu Problemen führen könnte.
Nun erstellen wir den VertexBuffer und zwar mit den folgenden Konstruktor:

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

Der erste Parameter bestimmt, welchen VertexType wir für unseren VertexBuffer verwenden wollen. Hier setzen wir einfach typeof(CustomVertex.TransformedColored) ein.
Der zweite Parameter bestimmt die Anzahl der Vertices, die wir in unserem VertexBuffer speichern wollen. Da wir nur ein Dreieck speichern wollen, setzen wir hier eine 3 ein.
Der vierte Parameter bestimmt das Device für welches der VertexBuffer erstellt

werden soll: m_Device.
Der fünfte Parameter bietet uns einen Satz von Konstanten an, die bestimmen, wie der VertexBuffer intern aufgebaut ist. Hier setzen wir Usage.WriteOnly ein, was besagt, das wir nur in den VertexBuffer schreiben wollen, ihn jedoch nicht auslesen (das kann zu kleineren Performancegewinnen führen).
Der vorletzte Parameter ist das VertexFormat:
CustomVertex.TransformedColored.Format
Der letzte Parameter bestimmt schließlich, wo wir unseren VertexBuffer speichern wollen. Hier können wir Pool.Default einsetzen, was besagt, dass Direct3D die Vertices einfach in den Video RAM platziert. Pool.Managed übergibt die Kontrolle des VertexBuffers an den Grafikkartentreiber. Der kann dann z.B., wenn der Platz auf dem Video RAM knapp wird, unsere Vertices wieder in den RAM verschieben. Pool.SystemMemory besagt schlieslich, dass wir unsere Vertices im System RAM plazieren wollen, was die Vorteile eines VertexBuffers jedoch wieder zunichte macht.
In der Praxis sieht das nun so aus:

```
m_VertexBuffer = new VertexBuffer(typeof(CustomVertex.TransformedColored),
3, m_Device, Usage.WriteOnly, CustomVertex.TransformedColored.Format,
Pool.Default);
```

Nun müssen wir unsere Vertices noch in den VertexBuffer hineinschreiben. Dazu müssen wir ihn zuerst sperren (locken), um sicherzustellen, dass während unseres Schreibvorgangs niemand anders auf den Buffer zugreift. Dann schreiben wir unsere Daten in den VertexBuffer und schlieslich entsperren wir ihn wieder(unlocken).Das ganze sieht so aus:

```
GraphicsStream stream = m_VertexBuffer.Lock(0,0,0);

stream.Write(verts);
m_VertexBuffer.Unlock();
```

Der Code bedarf eigentlich keiner großartigen Erklärungen, außer das die Parameter von Lock angeben, dass wir den ganzen VertexBuffer sperren wollen.

Nun kommen wir zum rendern und springen in die Render Methode.
Bevor wir unser Dreieck renden können, müssen wir unserem Device noch sagen, in welchem Format(CustomVertex.xxx) unsere Vertices gespeichert sind:

```
m_Device.VertexFormat = CustomVertex.TransformedColored.Format;
```

Jetzt haben wir zwei Möglichkeiten. Entweder wir rendern direkt aus dem SystemMemory heraus (unser Array verts liegt ja immer noch da drinnen) oder wir rendern vom VertexBuffer heraus. Ich zeig zuerst mal die Variante aus dem Systemmory, dazu nutzen wir die Funktion Device.DrawUserPrimitives

```
public void DrawUserPrimitives(
PrimitiveType primitiveType,
Int primitiveCount,
Object vertexStreamZeroData
```

```
);
```

Der erste Parameter bestimmt den Primitive Type. Hier können wir entweder Punkte, Linien oder Dreiecke einsetzen (ein Primitiv ist nichts anderes als ein Punkt, Linie oder Dreieck). Wir setzen natürlich PrimitiveType.TriangleStrip ein. Es gibt mehrere Varianten von jedem Primitives. TriangleStrip besagt, dass, falls wir evtl. mehrere Dreiecke haben, die Dreiecke in einem Band miteinander verbunden gerendert werden sollen.
Der nächste Parameter bestimmt die Anzahl der Primitive, die gezeichnet werden soll. Wir können unser verts Array z.B. mit 6 Elementen befüllen, um dann zwei Dreiecke zu zeichnen. Da wir aber nur ein Dreieck zeichnen wollen, setzen wir hier eine 1 ein.
Der letzte Parameter bestimmt nun das Array, in dem unsere Vertices liegen: verts
Also hier der Code:

```
m_Device.DrawUserPrimitives(PrimitiveType.TriangleStrip,1,verts);
```

Nun kommen wir zur Methode mit den VertexBuffer. Bevor wir aus einem VertexBuffer heraus rendern können, müssen wir unserem Device sagen, aus welchem wir überhaupt rendern wollen (ja, wir können mehrere VertexBuffer haben...). Dazu nutzen wir die Funktion Device.SetStreamSource:

```
m_Device.SetStreamSource(0,m_VertexBuffer,0);
```

Der erste Parameter bestimmt welchen Kanal (Vertex Stream) wir nutzen wollen. Einfach 0 einsetzen, denn das ist der 1. Vertex Kanal, auf dem die Vertices fließen.
Der zweite Parameter bestimmt jenen VertexBuffer, von dem die Daten kommen sollen.
Der letzte Parameter gibt den Offset vom Beginn des Streams bis zu den ersten Vertex Daten an. Hier setzen wir einfach wieder 0 ein.
Nun steht uns nichts mehr im Wege, aus dem VertexBuffer zu rendern. Und das machen wir via Device.DrawPrimitives:

```
m_Device.DrawPrimitives(PrimitiveType.TriangleStrip,0,1);
```

Die Parameter unterscheiden sich nicht allzu sehr von den Parametern von DrawUserPrimitives, bis auf den 2., der den Startvertex angibt, also den Vertex, ab dem gerendert werden soll (einfach 0). Der Letzte gibt wieder an, wie viele Primitive wir zeichnen wollen.
Dieses Kapitel war jetzt schon etwas länger, aber wir sehen ja auch mehr am Bildschirm.

Renderstates

Was ist ein Renderstate? Nun, ein Renderstate beschreibt, **wie** Direct3D unsere Dreiecke zeichnen soll. Es gibt sehr viele verschiedene Renderstates, daher kann ich jetzt nicht auf alle eingehen (abgesehen davon, könnten wir die meisten Renderstates mit unserem bisherigen Wissen ohnehin nicht verstehen). Daher will ich mal das grundlegende Konzept der Renderstates zeigen, damit wir später keine Probleme haben.
Also zuerst mal, beeinflusst ein Renderstate wie ein Dreieck gezeichnet wird. Ein einfacher Renderstate wäre z.B. der FillMode. Der sagt Direct3D, ob unser Dreieck, entweder solid(was der Standard Wert ist und mit dem haben wir auch in den bisherigen Anwendungen gearbeitet), ob nur die Eckpunkte oder nur die Linien gezeichnet werden sollen. Wir können disen RS vor dem Rendern unseres Dreiecks einsetzen:

```
m_Device.RenderState.FillMode = FillMode.WireFrame;
m_Device.SetStreamSource(0,m_VertexBuffer,0);
m_Device.DrawPrimitives(PrimitiveType.TriangleStrip,0,1);
```

Mit dem WireFrame Modus werden von unserem Dreieck nur die Linien gezeichnet. Daneben gibt es noch FillMode.Solid, der der Standardwert ist. Und dann gibt es noch FillMode.Point, der nur die Eckpunkte zeichnet (Bemerkung: Wenn wir den DrawPrimitive Aufruf auf m_Device.DrawPrimitives(PrimitiveType.PointList,0,3); abändern, hätte wir den selben Effekt wir mit dem Renderstate FillMode auf Point.).

Natürlich gibt es noch viel mehr RSs als nur FillMode, daher will ich hier noch ein paar vorstellen.

Der nächste ist DitherEnable. Beim Dithering wird da, wo mehrere verschiedene Farben aufeinandertreffen, ein Mittelwert aus diesen berechnet und dieser dann eingesetzt. Auf heutigen Grafikkarten macht es fast keinen Unterschied, ob Dithering eingeschalter ist oder nicht, daher schalten wir es ein:

```
m_Device.RenderState.DitherEnable = true;
```

Und noch einer ist das Culling. Das Culling entscheided, welche Objekte sichtbar sind und deshalb gezeichnet werden und welche nicht sichtbar sind und nicht gezeichnet werden.
Direct3D unterstützt bereits von Haus aus 2 einfache Arten von Culling, die in unseren Beispielanwendungen vollkommen ausreichen werden. Bei Spielen/Grafikdemos würde man natürlich zum Standard Culling noch sein eigenes implementieren.
Direct3D bietet uns das *clockwise(CW) culling* und das *counterclockwise(CCW) culling*. Wenn wir unsere Vertices im Uhrzeigersinn anordnen, brauchen wir CCW

culling, damit die "unsichtbaren" Vertices wegfallen, wenn unsere Vertices gegen den Uhrzeigersinn angeordnet sind, müssen CW culling verwenden.
Der Standard Wert dieses RS ist CCW. Natürlich können wir das Culling auch komplett abschalten:

```
m_Device.RenderState.CullMode = Cull.CounterClockwise; // Standard. Unser
Dreieck aus dem vorigen Beispiel würde gezeichnet werden

m_Device.RenderState.CullMode = Cull.Clockwise; // CW Culling. Um unser
Dreieck jetzt zu zeichnen müssten wir die Elemente 0 und 2 austauschen

m_Device.RenderState.CullMode = Cull.None; // Alles wird gezeichnet. Es ist
egal wie wir unsere Vertices anordnen
```

Gut, das waren mal 3 Renderstate, wir werden jedoch im Laufe des Tutorials noch viele mehr kennen lernen...

Transformationen

Bisher waren unsere Programme eher langweilig. Jetzt wird sich das aber ändern, da wir jetzt Bewegung in die ganze Sache bringen. Und dazu brauchen wir jedoch ein wenig Mathematik. Wie wir bereits gesehen haben, haben wir 3 Werte um einen Punkt in unserer Szene zu beschreiben. Diese sind die x-Koordinate, die y-Koordinate und die z-Koordinate. Währe es nun nicht gut, wenn wir diese Werte zusammenfassen könnten und mit ihnen rechnen? Nun, das geht natürlich und das ganze nennt sich einen *Vektor*. Wir verwenden einen 3D Vektor, da wir ja 3 Werte verwenden. Wie man damit rechnet will ich hier mal ausklammern, da es bereits auf ↑ Wikipedia einen ausgezeichneten Artikel zu diesem Thema gibt.
Das nächste, was wir wissen müssen, ist der Begriff der Matrix. Eine Matrix ist eine rechteckige Anordnung von Zahlen. Wir werden gleich sehen, wozu wir eine Matrix brauchen (für die Rechenregeln verweise ich wieder auf ↑ Wikipedia). Ihr braucht eigentlich erstmal nur einen groben Überblick besitzen, da uns Direct3D viel Arbeit beim Rechnen abnimmt.
Nun kommen wir erstmal zu einer wichtigen Frage: Wie können wir unsere Szene nun animieren bzw. in Bewegung versetzen.
Um das zu programmieren, müssen wir erstmal verstehen, wie unsere Vertices aus unseren VertexBuffern auf der Grafikkarte behandelt werden. Und zwar wird jeder Vertex duch eine sogenannte *Fixed Function Geometry Pipeline* geschleust. Dort werden die Vertices(welche als Vektoren vorliegen) mit 3 verschiedenen Matrizen transformiert. Anschließend werden die Vertices geclippt(= Culling) und ihre Koordinaten werden auf den Monitor skaliert d.h. die Koordinaten werden so angepasst, das sie auf bzw. für den Monitor passen.
Für uns sind nun die ersten 3 Schritte dieser Pipeline wichtig. Diese 3 Transformationen sind: *World Transformation, View Transformation und Projection Transformation*.
Sehen wir uns diese 3 einmal genauer an:
World Transformation:
Normalerweise sind alle Vertices relativ zum Ursprung des Objekts(also in einem

lokalen Koordinatensystem). Jedoch müssen wir, um das Objekt auch anzeigen zu können, dieses lokale Koordinatensystem zu einem so genannten *world space* transformieren. Der world space ist nichts anderes, als ein Koordinatensystem, in dem alle Koordinaten relativ zu einem gemeinsamen Ursprung sind. Welche genauen mathematischen Vorgänge das nun sind, dies ist für uns eher uninteressant. Für uns ist es jetzt wichtig zu wissen, dass wir mit dieser World Transformation unser Dreieck aus dem vorigen Beispiel bewegen können. Und diese World Transformation wird durch eine Matrix dargestellt, die World Matrix. Wir können nämlich auf die World Matrix 3 verschiedenen Operationen anwenden: Translieren, Rotieren und Skalieren. Translieren bedeutet einfach ein Objekt zu verschieben. Man kann es nach oben, unten und links, rechts verschieben. Rotieren ist wohl klar. Man rotiert das Objekt entweder um die x-Achse oder um die y-Achse. Und beim Skalieren kann man ein Objekt um einen bestimmten Faktor vergrößern oder verkleinern.
Ich habe jetzt von Objekt gesprochen, nun warum rede ich hier von Objekt und nicht von Matrix? Nun, alle diese Operationen werden auf die World Matrix angewendet und diese World Matrix wird ja wiederrum auf alle Vertices eines Objektes angewendet (wir können die World Matrix während der Laufzeit jederzeit verändern). DirectX bietet uns natürlich mehrere Funktionen, damit wir uns nicht mit den genauen Rechnungen herumschlagen müssen. Wichtig ist außerdem, dass wir skalieren, translieren und rotieren nicht einfach in beliebiger Reiehenfolge durchführen dürfen(bzw. können schon, jedoch kommen dann unlogische Verschiebungen zum Vorschein...) Die richtige Reihenfolge lautet: Skalieren und danach rotieren und zum Schluss translieren. Man kann auch zuerst translieren und dann rotieren(skalieren kommt immer als erstes.). Der unterschied ist einfach der, das ja um den Koordinatenursprung rotiert wird. Wenn man nun zuerst transliert und dann rotiert, dann wird das Objekt zuerst irgendwohin in der Szene gesetzt und dann nochmal um die Rotierung verschoben. Um diesen Effekt zu vermeiden, rotieren wir zuerst und translieren erst später.

View Transformation

Mit der View Transformation können wir eine Kamera oder auch Viewport erzeugen. Damit ist einfach gemeint, wohin der Betrachter einer Szene schaut. Eine View Matrix wird aus 3 Vektoren gebildet. Diese sind der eye point Vektor, der look-at Vektor und der world's up Vektor. Der eye point Vektor beschreibt, wo der Augenpunkt(Also der Ausgangspunkt) des Betrachters ist. Der look-at Vektor gibt an, wohin(also zu welchem Punkt) der Betrachter schaut. Und der world's up Vektor beschreibt die Ausrichtig auf der y Achse (dieser Vektor ist normalerweise [0, 1, 0] außer man will die Kamera um ihre eigene z Achse drehen).
Um eine View Matrix zu bilden, bietet uns DirectX wie immer die Möglichkeit dies mit mehreren vorgefertigten Funktionen zu tun.

Projection Transformation

Die Projection Transformation(beschrieben durch die Projection Matrix) ist dafür verantwortlich, das wir auf dem 2D Monitor eine 3D Szene darstellen und auch sehen können. Und zwar sorgt sie dafür, dass Objekte, die weiter weg sind, kleiner sind und Objekte, die näher da sind, größer sind(also wie, als wenn man aus dem Fenster schaut).
Es gibt noch etwas, dass wir beachten müssen: Ab sofort ist unser Koordinatenursprung die Bildschirmmitte. Ein Vektor (0, 0,0) würde als den Koordinatenursprung (= Bildschirmmitte) meinen. Negative Werte für x-Koordinaten

bedeuten, dass wir uns entlang der Horizontalen nach links bewegen, positive nach rechts. Für die y-Achse bedeuten positive Werte, dass wir uns nach oben bewegen und negative nach unten. Und für die z-Achse bedeuten positive Werte so viel wie „in den Bildschirm hinein" und negative „aus dem Bildschirm heraus".
Nun, jetzt wissen wir die Theorie, jedoch noch nicht, wie man das in der Praxis anwendet. In der Praxis ist das aber äußerst einfach: Man berechnet einfach die jeweiligen Matrizen und gibt sie dann dem Device bekannt. Alle Objekte, die wir danach rendern, werden mit den derzeit gesetzten Matrizen transformiert. Jedes Objekt wird immer nur einmal durch die Pipeline geschleust, außer man rendert es nochmal im selben Frame.

Aber wir müssen noch etwas bedenken. Unsere Initalisierung wird von nun an länger sein. Deshalb werden wir ab jetzt auch eine neue Struktur des Quellcodes haben(der dem aus dem SDK Tutorial sehr ähnlich ist ^^).

Kommen wir nun endgültig zur Praxis. Jedoch vorher müssen wir noch den Typ unserer Vertices ändern. Bisher hatten wir sie vom Typ TransformedColored, dies müssen wir jetzt ändern. Das Transformed bedeutet nämlich, das wir uns nicht um die einzelnen Transformationen kümmern wollen und diese DirectX überlassen wollen(was bei statischen Objekten ja auch durchaus sinnvoll sein kann). Wir benötigen jetzt jedoch Vertices vom Typ PositionColored.

Die einzelnen Matrizen setzen wir über uns device. D.h.
device.Transform.View = View Matrix
device.Transform.World = World Matrix
device.Transform.Projection = Projection Matrix

Jetzt wird es aber ernst: Die Render Funktion. Hier setzen wir zuerst die Matrizen und rendern dann aus unseren Vertex Buffer ein Dreieck. Die Besonderheit: Es dreht sich um die y-Achse. Sehen wir uns zuerst die World Matrix an:

```
int iTime = Environment.TickCount % 1000;
float fAngle = iTime * (2.0f * (float)Math.PI) / 1000.0f;
device.Transform.World = Matrix.RotationY(fAngle);
```

Zunächst berechnen wir den Winkel, in um den sich das Dreieck rotieren soll. Danach setzen wir die World Matrix einfach über die Funktion Matrix.RotationY, welche uns eine Roation um die y-Achse mit dem angegebenen Winkel(fAngle) berechnet.

Nun die View Matrix:

```
device.Transform.View = Matrix.LookAtLH( new Vector3( 0.0f, 3.0f,-5.0f ),
new Vector3( 0.0f, 0.0f, 0.0f ), new Vector3( 0.0f, 1.0f, 0.0f ) );
```

Die View Matrix ist nichts Besonderes. Zuerst wird der eye Point definiert, dann der look-at Vektor und dann noch world's up.

```
device.Transform.Projection = Matrix.PerspectiveFovLH( (float)Math.PI / 4,
1.0f, 1.0f, 100.0f );
```

Hier die die Projection berechnet. Ebenfalls nichts Besonderes.

In den nächsten Programmen werden wie die Projections Matrix und die View Matrix eher in den Schatten stellen und uns dafür ausgiebig mit der World Matrix beschäftigen.
Etwas, das wir noch klären müssen, ist warum wir das Culling ausschalten. Wir rotieren ja das Dreieck. Und beim rotieren sind die Vertices ja einmal gegen den Uhrzeigersinn angeordnet und einmal im Uhrzeigersinn. Wenn wir jetzt das Culling einschalten würden, würden wir je nach gesetzen Wert eine Seite des Dreiecks nicht sehen(einfach mal ausprobieren).
Ansonsten gibt es nicht mehr viel zu sagen. Der Code ist halt vom Microsoft DirectX Tutorial übernommen, aber ich hoffe mal, dass das kein Problem darstellt.

Die World Matrix

Sehen wir uns nun die World Matrix einmal genauer an. Im letzten Beispiel haben wir ja das Dreieck rotiert, was ist aber, wenn wir es still stehen lassen wollen. Dann müssen wir die World Matrix auf die Identitätsmatrix setzen(was mit 1 oder 0 verglichen wird. Also eine Matrix + der Identitätsmatrix ergibt wieder die Matrix selbst). Das machen wir so:

```
device.Transform.World = Matrix.Identity;
```

Damit steht unser Dreieck jetzt still. Kommen wir jetzt zum Rotieren. Wir können das Objekt um die x-Achse, die y-Achse und um die z-Achse rotieren. Die Parameter beschreiben immer den Winkel, um den rotiert werden soll:

```
int  iTime  = Environment.TickCount % 1000;
float fAngle = iTime * (2.0f * (float)Math.PI) / 1000.0f;
device.Transform.World = Matrix.RotationX(fAngle);
```

Hier wird unser Objekt jetzt eine Rotation um die X Achse vollziehen. Es gibt noch die Funktionen Matrix.RotationY(für die Y Achse) und Matrix.RotationZ(für die Z Achse)

Nun zum Translieren oder verschieben. Erstmal ein Beispiel:

```
device.Transform.World = Matrix.Translation(1.0f,0.0f,0.0f);
```

Wie unschwer zu erkennen, haben wir hier unser Dreieck auf der x-Achse verschoben(Vektorangaben sind immer x,y,z).
Und zum Schluss das skalieren. Das skalieren bedeutet ja vergrößern oder verkleinern.

```
device.Transform.World = Matrix.Scaling(1.5f,1.0f,1.0f);
```

Auch hier können wir auf den verschiedenen Achsen skalieren. Wichtig ist zu wissen, dass der Wert 1 bedeutet, dass das Objekt seine ursprüngliche Größe behalten soll. Jeder Wert kleiner 1 bedeutet eine Verkleinerung, jeder Wert größer 1(so wie hier), bedeutet eine Vergrößerung.

In einer echten Anwendung wollen wir jedoch nicht einfach nur mal rotieren und mal translieren, sondern diese Operationen auch kombinieren. Das ist eigentlich ganz einfach: Wir müssen einfach eine Matrix bilden und diese dann als World Matrix setzen. Wie wir bereits wissen, müssen wir zuerst skalieren, dann rotieren und dann translieren. Wichtig ist, das man die einzelnen Ergebnisse(Skalieren, rotieren, translieren) miteinander multiplizieren muss (eine Matrix Multiplikation ist jedoch nicht assoziativ d.h. Matrix M * Matrix B ist ungleich Matrix B * Matrix M).

```
Matrix world;

// skalieren
world = Matrix.Scaling(1.0f,1.0f,1.0f);
// rotieren
world *= Matrix.RotationY(1.0f);
// translieren
world *= Matrix.Translation(1.0f,0.0f,0.0f);

device.Transform.World = world;
```

3D Objekte

Bisher waren unsere Programme eher langweilig. Klar gibt es heute noch 2D Spiele und 2D war mal das Maß aller Dinge, aber die richtige Action gibt's heute mit 3D. Und deshalb werden wir uns jetzt mal ansehen, wie man einfache 3D Objekte mit Direct3D erstellt und diese anzeigt.
Zunächst jedoch ein wenig Theorie, um überhaupt zu verstehen, wie 3D Objekte gespeichert werden: Wie unschwer zu erkennen ist der Monitor vor euch eine Fläche. D.h. das er 2 Koordinaten hat. Die x und die y Koordinate(wie man es beim kartesischen Koordinatensystem in der Schule lernt ^^). Durch ein Koordinatenpaar x|y können wir nun jeden beliebigen Punkt auf diesem Monitor angeben. Aber wie kann man dann 3D Objekte, wie man sie in jedem Spiel hat, abbilden? Wie ist es möglich, dass man eine 3D Szene auf ein 2D Koordinatensystem abbildet? Die Antwort ist simpel: Man führt einfach noch eine Koordinate ein(die z Koordinate), die einfach mit den beiden vorhanden Koordinaten verknüpft wird. Nun, das klingt ein wenig schwer verständlich, jedoch ist es im Endeffekt für uns Programmierer sehr einfach: Wir geben einfach bei Koordinaten eine z Koordinate zusätzlich an, die Tiefeninformationen speichert. Eine positive z Koordinate zeigt dabei an, dass wir "in den Bildschirm hinein" wollen, eine negative "aus dem Bildschirm heraus". Mathematisch ausgedrückt müssen wir nun also eine Formel finden, die uns nun aus einem Koordinaten Tripel x|y|z ein Koordinatenpaar x|y macht. Dazu kommen wir nun auf etwas, das wir jeden Tag erleben: Wenn wir uns von einem Haus entfernen, dann erscheint es mit zunehmender Entfernung kleiner. Nähern wir uns dem Haus, wird es größer. Dazu dividieren wir x und y einfach durch z. Die Koordinaten, die wir

dadurch erhalten nennt man projizierte Koordinaten. Die Formel lautet nun:
x' (x projiziert) = x / z
y' (y projiziert) = y / z

Wir müssen uns diese Formel nicht merken und uns um diese Umrechnung auch keine Sorgen machen: Direct3D bzw. die Grafikhardware macht dies automatisch(dazu stecken wir ja mehrere hundert € in Grafikkarten(zumindest ich ^^)).

Jetzt können wir uns wieder Direct3D zuwenden. Normalerweise würde hier nun ein Beispiel kommen, in dem man einen Vertex Buffer und einen Würfel oder eine Pyramide erzeugt (und tausende Koordinaten von Hand eingibt), jedoch will ich jetzt mal nicht diesen Weg gehen. Und zwar nutzen wir einfach eine vorgefertigte Funktion, die uns bereits eine "Box" aus 3 Parametern berechnet und die wir dann leicht anzeigen können. Außerdem lernen wir dann gleich mal mit der Klasse Mesh umzugehen.
Zunächst deklarieren wir eine neue Variable box vom Typ Mesh. Mesh ist eine Klasse, in der man genau solche Vertex Informationen leicht speichern und anzeigen kann(bisher haben wir uns ja um die Erstellung eines Vertex Buffers gekümmert und um dessen anzeigen ...).

```
Mesh box;
```

Nun verwenden wir die statische Funktion Box der Klasse Mesh, die uns aus 3 Parametern eine Box berechnet und uns eine Variable vom Typ Mesh zurückgibt.

```
box = Mesh.Box(device,2.0f,1.0f,2.0f);
```

Die Parameter sind simpel: Der Erste ist einfach das device. Der zweite beschreibt die Ausdehnung auf der x Achse. Der dritte auf der y Achse und letzte schließlich, *trommelwirbel*, auf der z Achse.
Diese Funktion berechnet uns nun aus diesen Parametern eine Box die wir in der der Variable box speichern. Einfach, nicht wahr?

Nun kommen wir zum anzeigen. Das geht ebenfalls entsprechend einfach. Wir springen in unsere Render Funktion und rufen die Funktion DrawSubset auf. Fertig.

```
box.DrawSubset(0);
```

Um den Parameter brauchen wir uns erstmal nicht kümmern(erst später, wenn wir dann zB Vertex Daten von Dateien laden o.ä.).

Die Mesh Klasse bietet uns jedoch nicht nur eine Funktion, für die Erstellung einer Box, sondern wir können mir ihr auch noch Teekannen, Zylinder, Kugeln und Ringe erzeugen. Und natürlich auch beliebige 2D Polygone(ein Polygon ist einfach der Sammelbegriff für Dreiecke, Vierecke, Sechsecke usw.). Einfach mal ein wenig in der Klasse rumstöbern ^^

Einen Schönheitsfehler hat das ganze jedoch. Die Box erscheint weiß. Logisch, wir haben ja auch keine Farbe angegeben. Wie können wir der Box jetzt jedoch eine Farbe zuweisen? Nun, dazu müssen wir einfach die Vertex Daten ändern, doch dazu im nächsten Kapitel mehr.

Der Box eine Farbe zuweisen

Nun, wir haben jetzt zwar eine schöne Box, jedoch ist sie noch komplett weiß. Das bringt uns natürlich nicht viel, da wir ja mal aufregende Spiele schreiben wollen. Wie bringen wir jetzt jedoch Direct3D dazu, die Box mit einer
Farbe zu rendern? Das ist eigentlich ganz einfach: Wir verändern einfach die Vertices der Box direkt. Und dazu kommen wir wieder auf unsere Vertex Buffer zu sprechen, aber wir brauchen diesmal keinen selber erstellen, sondern wir holen uns die Vertex Daten einfach aus der Mesh Klasse(die diese ja für uns speichert).
Und das ist mit ein paar Handgriffen gemacht. Zuerst müssen wir unseren Mesh "klonen" und bei diesem Vorgang das
Vertex Format ändern. Dies ändern wir so ab, dass wir die Box ganz normal transformieren können, jedoch jedem Vertex noch eine Farbe zuweisen können. Nachdem das getan ist, holen wir uns die Vertex Daten des Mesh und schreiben einfach unsere gewünschte Farbe hinein und geben die Vertex Daten der Mesh Klasse wieder bekannt. Und hierbei verändern wir nur die Farbkomponente, nicht die Koordinaten der Vertices, da diese ja bereits vorberechnet wurden.
Sehen wir uns nuneinmal an, wie wir unseren Mesh klonen und ihm ein neues Vertex Format zuweisen:

```
VertexFormats format = VertexFormats.PositionNormal |
VertexFormats.Diffuse;
Mesh tempBox = box.Clone( box.Options.Value, format, dev );
box.Dispose();
box = tempBox;
```

Zuerst definieren wir eine Variable vom Typ VertexFormat das unser VertexFormat enthält. Das VertexFormat für uns ist PositionNormal(das standardmäßig verwendet wird) und dazu fügen wir Diffuse hinzu. Diffuse beschreibt die diffuse Farbe eines Vertex. Die diffuse Farbe können wir uns derzeit einfach als jene Farbe merken, die man direkt sieht. Wir werden später, wenn wir unsere Szene beleuchten sehen, dass es noch andere Arten von Farben bzw. Lichtfarben gibt.
Danach verwenden wir die Funktion Clone um einen vorläufigen Mesh zu erstellen. Danach löschen wir den eigentlichen Mesh wieder und weisem unserem Mesh den Vorläufigen zu.
Nun müssen wir an die Vertex Daten kommen. Das geht ebenfalls äußerst einfach: Wie im Kapitel über Vertex Buffer locken wir ihn einfach und kommen so an seine Daten (Lock sperrt die Daten eines Vertex Buffers für weitere Zugriffe und gibt uns eine Referenz aus diese zurück. Die Änderungen werden übernommen und der Vertex Buffer wird entsperrt, wenn wir Unlock aufrufen)
Gespeichert werden sie natürlich in einem Array vom Typ PositionNormalColored:

```
CustomVertex.PositionNormalColored[] verts =
(CustomVertex.PositionNormalColored[])box.VertexBuffer.Lock( 0,
        typeof( CustomVertex.PositionNormalColored ),
        LockFlags.None,
        box.NumberVertices );
```

Da uns Lock nur ein System.Array zurückgibt, müssen wir dies erst explizit auf CustomVertex.PositionNormalColored casten.
Der Erste Parameter bestimmt die Menge an Daten, die wir sperren wollen. Wenn wir 0 angeben, wird der gesamte Vertex
Buffer gesperrt. Beim zweiten Parameter müssen wir bestimmen, welchen Typ die zurückgegebenen Daten haben sollen.
Der Dritte bestimmt die Lock Flags, die wir hier jedoch vernachlässigen können. Der letzte Parameter bestimmt, wie viele
Vertices zurückgegeben werden sollen. Da wir ja jeden Vertex des Mesh verändern wollen, müssen wir auch alle angeben.
Der nächste Schritt ist, die Vertex Daten zu verändern. Dazu gehen wir jeden Vertex durch und ändern seine Farbe:

```
for(int i=0;i < verts.Length;++i)
{
        verts[i].Color = Color.Red.ToArgb();
}
```

Die Variable Color ist uns bereits im Kapitel über Vertex Buffer begegnet, sie bestimmt welche Farbe der Vertex haben soll.
Als Letztes müssen wir den Vertex Buffer wieder entsperren, um die Änderungen zu übernehmen.

```
box.VertexBuffer.Unlock();
```

Nun rotieren wir unseren Mesh noch:

```
Matrix world;
world = Matrix.Scaling(1.0f,2.0f,1.0f);

int iTime = Environment.TickCount % 1000;
float fAngle = iTime * (2.0f * (float)Math.PI) / 1000.0f;

world *= Matrix.RotationY(fAngle);
world *= Matrix.Translation(0.0f,0.0f,0.0f);

device.Transform.World = world;
box.DrawSubset(0);
```

Und das wars schon. Direct3D ist einfach, wenn man weis wie es geht ^^
Achja: Über den Index in der for Schleife kann man entscheiden auf welcher Seite des Würfels man nun ist. Die ersten vier Vertices sind die erste Seite, die nächsten Vier die zweite usw.
Ich habe im Source noch ein paar if eingefügt, die zeigen, wie man so was macht.

Mehr als 2 Objekte

Bisher hatten wir immer nur 1 Objekt auf unserem Bildschirm. Das wird sich nun ändern, da wir nun zwei Objekte rendern werden. Wie immer baut dieses Tutorial auf den anderen auf, sowohl im Wissen, als auch im Code. Doch nun zum vergnüglichen Teil ^^
Zuerst brauchen wir natürlich zwei Objekte, dazu nehmen wir einfach die rote Box aus dem vorigen Teil und eine Teekanne.
Lassen wir die Teekanne einfach mal grün sein.
Dazu machen wir eigentlich dasselbe, was wir mit der Box getan haben: Eine Variable vom Typ Mesh erzeugen, in ihr eine
Teekanne speichern und danach die Farbe verändern. Zuerst definieren wir die Membervariable teapot:

```
Mesh teapot;
```

Danach weisen lassen wir uns wieder durch eine statische Funktion eine Kugel vorberechnen und speichern diese in der
Variable teapot:

```
teapot = Mesh.Teapot(device);
```

Die Mesh Klasse ist wirklich toll ^^
Nun machen wir dasselbe, was wir für die Box getan haben. Wir verändern die Farbe der Teekanne. Da wir den Code gleich
nach dem erstellen der Box einfügen, müssen wir nicht wieder extra Variablen einfügen und können so das Vertex Format von
oben verwenden:

```
tempBox = teapot.Clone( teapot.Options.Value, format, dev );
                        teapot.Dispose();
                        teapot = tempBox;

verts = (CustomVertex.PositionNormalColored[])teapot.VertexBuffer.Lock( 0,
                        typeof( CustomVertex.PositionNormalColored ),
                        LockFlags.None,
                        teapot.NumberVertices );


for(int i=0;i < verts.Length;++i)
{
        verts[i].Color = Color.Green.ToArgb();
}
teapot.VertexBuffer.Unlock();
```

Jetzt kommen wir zum interessanten Teil. Und zwar zum Rendern. Zuerst begnügen wir uns mal, einfach beide Objekte zu
rendern, ohne sie zu rotieren o.ä.
Dazu müssen wir für jedes Objekt zuerst eine World Matrix setzen und es dann damit

zeichnen.
Anders ausgedrückt:
Begin Scene

Projection Matrix setzen
View Matrix setzen

World Matrix für Objekt 1 setzen
Objekt 1 rendern

World Matrix(und evtl. auch Projection und View Matrix) für Objekt 2 setzen
Objekt 2 rendern

World Matrix(und evtl. auch Projection und View Matrix) für Objekt n setzen
Objekt n rendern
End Scene

Das ist natürlich stark vereinfacht ausgedrückt, da in echten Spielen natürlich noch haufenweise andere Sachen dazukommen (wie z.B. Kollisionsabfrage, Benutzereingabe etc.). Wir werden später noch sehen, wie man eine einfache Benutzereingabe verwirklichen kann.
Sehen wir uns nun einmal an, wie wir das rendern verwirklichen. Zuerst setzen wir die Projection und View Matrix, für unser Beispiel reicht eine Projection und eine View Matrix.

```
device.Transform.View = Matrix.LookAtLH( new Vector3( 0.0f, 3.0f,-5.0f ),
new Vector3( 0.0f, 0.0f, 0.0f ),
new Vector3( 0.0f, 1.0f, 0.0f ) );

device.Transform.Projection = Matrix.PerspectiveFovLH( (float)Math.PI / 4,
1.0f, 1.0f, 100.0f );
```

Danach setzen wir die World Matrix für das erste Objekt und zeichnen es:

```
Matrix world;
world = Matrix.Scaling(0.5f,0.5f,1.5f);

world *= Matrix.Translation(-1.0f,0.0f,0.0f);
device.Transform.World = world;

box.DrawSubset(0);
```

Zuerst verkleinern wir das Objekt und strecken es auf der z Achse "nach hinten". Danach verschieben wir das Objekt um
eine Einheit nach links(auf der x-Achse, negative Koordinaten sind hierbei nach links gerichtet). Schließlich rendern wir es.
Nun zum zweiten Objekt. Wir setzen unsere Variable world zuerst auf die Identitätsmatrix:

```
world = Matrix.Identity;
```

Nun verkleinern wir das Objekt ein wenig und rotieren seitlich. Danach verschieben wir es um eine Einheit nach rechts.

```
world = Matrix.Scaling(0.5f,0.5f,1.0f);
world *= Matrix.Translation(1.0f,0.0f,0.0f);
world *= Matrix.RotationX(380);
```

Nun setzen wir die neue World Matrix und rendern die Teekanne:

```
device.Transform.World = world;
teapot.DrawSubset(0);
```

Und das wars schon. Was ganz wichtig ist, man darf vor dem rendern nicht vergessen, eine neue World Matrix zu setzen, denn
ansonsten wir weiterhin die Alte verwendet. Das könnt ihr ja ausprobieren. Kommentiert einfach mal den device.Transform.World =
world; vor dem teapit.DrawSubset(0); aus. Ihr werdet sehen, dass die Teekanne dann auf der Box liegt. Kein schöner Anblick(ich weis, die derzeitige Szene ist auch nicht gerade "schön", aber das muss man einfach können. Wenn wir dann Spiele programmieren wollen, müssen wir uns auf diese Grundlagen zurückerinnern ^^)

Input -> Output

Nun, in unseren Anwendungen war jetzt zwar bereits etwas Bewegung. In Spielen kommt jedoch noch ein weiterer Faktor hinzu: Der Userinput. Ohne diesen gäbe es eigentlich keine Spiele. Deshalb sehen wir uns einmal eine vereinfachte Form an, die wir in unseren Anwendungen nutzen können. Unser Ziel ist einfach: Wir wollen anhand der Leertaste steuern ob sich unser Objekt dreht oder nicht. Dazu überschreiben wir einfach die Funktion ProcessCmdKeys der Basisklasse Form. Diese liefert uns im Parameter die gerade gedrückte Taste.

```
protected override bool ProcessCmdKey(ref Message msg, Keys keyData)
{
        return base.ProcessCmdKey( ref msg, keyData );
}
```

Nun können wir unseren bisherigen Eventhandler für Input löschen und fügen hier folgendes ein:

```
if(keyData == Keys.Escape)
        Application.Exit();
```

Nun kommen wir zum eigentlichen Teil: Zunächst reduzieren wir den Rendercode auf eine Box(und löschen nebenbei natürlich überflüssigen Code für das zweite Objekt aus dem letzten Tutorial)

```
private void Render()
{
        if (device == null)
                return;

        device.Clear(ClearFlags.Target, System.Drawing.Color.Blue, 1.0f,
0);
        device.BeginScene();

        device.Transform.View = Matrix.LookAtLH( new Vector3( 0.0f, 3.0f,-
5.0f ), new Vector3( 0.0f, 0.0f, 0.0f ), new Vector3( 0.0f, 1.0f, 0.0f ) );
                        device.Transform.Projection =
Matrix.PerspectiveFovLH( (float)Math.PI / 4, 1.0f, 1.0f, 100.0f );

        Matrix world;
        world = Matrix.Scaling(0.5f,0.5f,1.5f);

        device.Transform.World = world;

        box.DrawSubset(0);

        device.EndScene();
        device.Present();
}
```

Nun müssen wir eine Variable vom Typ bool in unsere Klasse aufnehmen, die anzeigt ob das Objekt derzeit rotieren soll oder nicht:

```
bool rotate = false;
```

Und schon kommen wir wieder zur ProcessCmdKey Funktion:

```
if(keyData == Keys.Space)
        rotate = !rotate;
```

Wir fragen ab ob die Leertaste(Space) gedrückt wurde, wenn ja dann wird rotate invertiert (aus false wird true gemacht, aus true false).

Nun müssen wir jedoch noch in der Render Funktion etwas ändern.
Vor dem device.Transform.World = world müssen wir noch abfragen ob rotate auf true ist, wenn ja dann müssen wir unser Objekt drehen lassen.

```
Matrix world;
world = Matrix.Scaling(0.5f,0.5f,1.5f);

if(rotate)
        world *= Matrix.RotationX(Environment.TickCount * (float)0.0025);

device.Transform.World = world;

box.DrawSubset(0);
```

Und damit ist es schon fertig. So einfach kann Benutzereingabe sein ^^

Texture Mapping

Alle unsere Objekte hatten bisher immer nur eine einzige Farbe und die Oberfläche der Objekte war glatt und durchgehend.
Jedoch hat eine solche Oberfläche so gut wie kein Objekt der realen Welt. Wenn man sich umschaut sind solche regelmäßigen Anordnungen sehr selten und Oberflächen sind auch nicht sehr oft so glatt und "perfekt" wie wir sie bisher hatten.
Deshalb greifen wir auf das Texture Mapping zurück: Wir weisen unserem Objekt nicht einfach eine Farbe zu, sondern eine Textur. Was ist jetzt eine Textur? Eine Textur ist ganz einfach ein Bild das wir über das Objekt legen. Dadurch können wir jetzt zB einen Ziegelstein nicht einfach nur rot färben, sondern ihn auch ein realistisches Aussehen verpassen.
Sehen wir uns nun einmal an wie wir unsere Box texturieren können. Zunächst brauchen wir eine neue Variable vom Typ Texture. In ihr speichern wir unsere Textur:

```
Texture textur;
```

Nun müssen wir unsere Textur laden. Das geht ziemlich einfach, da uns Direct3D die Klasse TextureLoader zur Verfügung
stellt, in der bereits vorgefertigte Funktionen zum Laden von Texturen enthalten sind:

```
textur = TextureLoader.FromFile(device,"iron04.jpg");
```

Wir verwenden die Funktion FromFile, die als ersten Parameter das Device verlangt und als zweiten den Pfad zum Bild Bild, das wir in der Textur speichern wollen. Die FromFile Funktion ist ein wahres Multitalent, denn laut SDK unterstützt
es die folgenden Bildformate: .bmp, .dds, .dib, .hdr, .jpg, .pfm, .png, .ppm, und .tga
In echten Spielen sollte man wenn möglich nicht jpg verwenden, so wie ich das hier tue, jedoch ist eine jpg schön klein und als Beispiel sollte es reichen. Der Nachteil ist halt das ein jpg als Textur nicht wirklich toll aussieht.
Der Rückgabewert der Funktion ist ein Texture Objekt das die geladene Textur enthält.
Als nächstes müssen wir unsere Box anpassen. Bisher haben wir ja die Vertices der Box mit einer Farbkomponente erweitert, jetzt müssen wir sie mit Texturkoordinaten erweitern.
Texturkoordinaten? Was sind Texturkoordinaten?
Wenn wir eine Textur auf ein Dreieck legen, woher soll Direct3D jetzt wissen wie es die Textur auf das Dreieck legen soll?
Deshalb gibt es die Texturkoordinaten, mit ihnen können wir bestimmen wie eine Textur auf ein Dreieck gelegt werden soll.
Als Anlehnung an das Wort Vertex werden diese Koordinaten auch Texel genannt. Ein Texel ist immer zweidimensional (Bilder sind ja auch nur zwei dimensional) und

hat also 2 Achsen: Die u-Achse und die v-Achse. Die u-Achse ist eigentlich die x-Achse, während die v-Achse die y-Achse ist. Der Koordinatenursprung einer Textur liegt jedoch in der linken oberen Ecke.
Eine weitere Besonderheit von Texel sind das sie in einem Intervall von [0,1] liegen. Dh das es bei allen Texturen nur Texel zwischen 0 und 1 gibt unabhängig von der eigentlichen Größe der Textur. Der Texel 0,5 liegt daher immer in der Mitte einer Achse.
Wir müssen nun also jedem Vertex einen Texel zuweisen, der anzeigt welcher Teil einer Textur auf diesen Vertex gelegt werden soll.
Ein Beispiel: Nehmen wir an wir haben ein Rechteck mit 4 Vertices und eine Textur. Die Textur ist ebenfalls rechteckig und wir wollen nun die Textur auf dieses Rechteck legen. Dazu müssen wir jetzt jedem Vertex des Rechtecks einen Texel der Textur zuweisen.
Sehen wir uns es jetzt den 1. Vertex an. Nehmen wir mal an das dieser in der linken oberen Ecke liegt. Wir können diesem
Vertex jetzt den Texel (0.0/0.0) zuweisen. Oder (1.0/1.0). Uns sind hierbei keine Grenzen gesetzt. Wir wollen aber die Textur normal auf das Rechteck legen, also so wie ein Bildprogramm uns die Textur anzeigt, so wollen wir diese auf dem Rechteck haben. Also weisem wir dem 1. Vertex den Texel (0/0) zu. Der 2. Vertex liegt nun in der rechten oberen Ecke.
Also weisen wir diesem Vertex den Texel (1/0) zu(wir müssen auf der u-Achse ganz nach rechts). Der 3. Vertex ist nun
die linke untere Ecke. Also brauchen wir hier den Texel (0/1). Und für den letzten Vertex brauchen wir schließlich den Texel (1/1).
Ich hoffe mit diesem Beispiel versteht man die Texturkoordinaten recht gut. Am Besten ist es hier eine Zeichnung zu machen.
Kommen wir nun zum praktischen:

```
box = Mesh.Box(device,2.0f,1.0f,2.0f);
VertexFormats format = VertexFormats.PositionNormal |
VertexFormats.Texture1;
Mesh tempBox = box.Clone( box.Options.Value, format, device );
box.Dispose();
box = tempBox;

CustomVertex.PositionNormalTextured[] verts =
(CustomVertex.PositionNormalTextured[])box.VertexBuffer.Lock( 0,
typeof( CustomVertex.PositionNormalTextured ),
               LockFlags.None,
       box.NumberVertices );
```

Wir wir sehen ist der Code nicht allzu schwer. Das VertexFormat der Box müssen wir auf PositionNormal und Texture1 setzen.
Texture1 beschreibt, das wir pro Vertex 1 Texel haben.
Danach müssen wir noch die neuen Vertices die wir schreiben wollen erstellen. Hierbei müssen wir PositionNormalTextured verwenden, anstatt PositionNormalColored.
Nun kommen wir zum zuweisen der Texel, was nicht sonderlich schwer ist:

```
for(int i=0;i < verts.Length;++i)
```

```
{
        verts[i].Tu = verts[i].X * 0.8f;
        verts[i].Tv = verts[i].Y * 0.8f;
}
box.VertexBuffer.Unlock();
}
```

Sehen wir uns jetzt das Rendern an. Bevor unsere Box normal rendern können, müssen wir noch die Textur setzen. Dh wir müssen Direct3D sagen das es jetzt für die nächsten Operationen wo es Texturen braucht die Textur nehmen soll, die wir gesetzt haben.

```
device.SetTexture(0,textur);
```

Was hat die 0 hier zu bedeuten? Wie wir später sehen werden gibt es verschiedene Texture Stages. Die unterste dieser Schichten ist die Stufe 0. Mit diesen Texturstages werden wir später zwei oder mehr Texturen miteinander verbinden können und auf unser Objekt legen. Dieses Verfahren nennt man dann Multi-Texturing.
Nun können wir unser Objekt mit der Textur rendern:

```
box.DrawSubset(0);
```

Und das wars schon. Mit Texturen kann man natürlich noch viel mehr anfangen, doch dazu mehr in späteren Teilen 

Der Indexbuffer - Indizieren von Vertices

Diesmal sehen wir uns den sogenannten Indexbuffer an. Indexbuffer sind dazu da um Vertices zu reduzieren.
Bei einer Indizierung fallen nämlich doppelte Vertices weg und man kann so einen Vertexbuffer deutlich optimieren.
Nehmen wir an wir wollen zwei Dreiecke zeichnen. Grafisch würde das so aussehen:
1 2|4

0|3 5

Die Nummern geben dabei die einzelnen Vertices an. Für dieses Beispiel bräuchten wir also 6 Vertices in einem Vertexbuffer. Jedoch fällt uns auf das zwei Vertices mit denselben Koordinaten zweimal im Vertexbuffer stehen.
Ist das nicht sinnloser Verbrauch von Speicherplatz? Deshalb erstellt man nun einen Indexbuffer in dem man speichert, an welcher Position welcher Vertex ist.

Index	Wert	Vertex
0	0	(x0, y0)
1	1	(x1, y1)
2	2	(x2, y2)
3	0	(x0, y0)
4	2	(x2, y2)

5 5 (x5, y5)

Wie wir in der Tabelle sehen müssen wir nur mehr 4 verschiedene Vertices speichern, nicht mehr 6 verschiedene.
In diesem Beispiel mag das nicht als viel erscheinen, aber wenn man große Objekte hat, kann man mit Indexbuffer viel einsparen und sehr gut optimieren. Zu beachten ist aber, dass man jedoch 6 Indices braucht. In DirectX SDK Hilfe wird übrigens empfohlen nur mit indizierten Vertices zu arbeiten. Und nun können wir bereits zur Praxis kommen. Direct3D stellt uns die Klasse IndexBuffer zur Verfügung, um einen IndexBuffer zu erstellen. Wir werden jedoch nicht direkt mit dem Indexbuffer arbeiten sondern verwenden ein Mesh Objekt. Ein Mesh Objekt ist einfach eine Klasse die das Arbeiten mit Vertexdaten vereinfacht.

```
Mesh Object;
```

Nun erstellen wir zuerst 4 Vertices, danach 6 Indices und speichern diese im Mesh Objekt:

```
CustomVertex.PositionOnly[] verts = {
                new CustomVertex.PositionOnly(-1f,  0f, -1f),
                new CustomVertex.PositionOnly( 1f,  0f, -1f),
                new CustomVertex.PositionOnly( 1f,  0f,  1f),
                new CustomVertex.PositionOnly(-1f,  0f,  1f),
};

short[] indices = { 0,1,2,            // erstes Dreieck
                  0,2,3 };     // zweites Dreieck
```

Dieser Code bietet uns keine Probleme. Wir verwenden short Indices aus Speicherplatzgründen, da wird ja nur kleine Werte brauchen(von 0 bis 3) und so müssen wir auf dem RAM der Grafikkarte pro Index nur 2 Bytes verwenden anstatt 4 Byte wie bei int. Die Zahlen im indices Array weisen auf die Vertices hin, die beim Zeichnen verwendet werden. Für das erste Dreieck verwenden wir die Vertices 0, 1 und 2. Fürs zweite 0, 2 und 3.

```
Object = new Mesh(indices.Length /
3,verts.Length,MeshFlags.WriteOnly,CustomVertex.PositionOnly.Format,
                                  device);

Object.VertexBuffer.SetData(verts,0,LockFlags.None);
Object.IndexBuffer.SetData(indices,0,LockFlags.None);

}
```

Hier erstellen wir ein neues Mesh Objekt und setzen die VertexBuffer und IndexBuffer Werte dafür. Die Parameter des Mesh Konstruktors sind einfach:
Der erste Parameter gibt die Anzahl der Flächen eines Objektes an. Wir brauchen logischerweise 2 Flächen. Einmal fürs erste Dreieck und einmal fürs Zweite.

Der zweite Paramter gibt dann die Anzahl der Vertices an. Der Dritte gibt sogenannte MeshFlags an. Diese können dazu verwendet werden der Klasse gewisse Informationen zu geben, wie sie die Daten behandeln soll. Wir wollen nur Daten in das Mesh Objekt schreiben, jedoch keine herauslesen. Dazu geben wir MeshFalgs.WriteOnly an.
Der nächste ist das Format der Vertexdaten. Der letzte Parameter gibt dann schließlich das Device an.
Danach greifen wir direkt auf den VertexBuffer und den IndexBuffer zu und setzen die Daten mit SetData.
Nun können wir auch schon zur Renderfunktion kommen.
Diese ist diesmal auch ganz einfach:

```
device.Clear(ClearFlags.Target , System.Drawing.Color.Blue, 1.0f, 0);

device.BeginScene();

device.Transform.View = Matrix.LookAtLH( new Vector3( 0.0f, 3.0f,-5.0f ),
new Vector3( 0.0f, 0.0f, 0.0f ), new Vector3( 0.0f, 1.0f, 0.0f ) );
device.Transform.Projection = Matrix.PerspectiveFovLH( (float)Math.PI / 4,
1.0f, 1.0f, 100.0f );

device.Transform.World = Matrix.Identity;

device.RenderState.CullMode = Cull.Clockwise;
Object.DrawSubset(0);

device.EndScene();
device.Present();
```

Der Code bietet eigentlich nur vertraute Dinge -.-

Was ist jetzt jedoch wenn wir direkt aus einem VertexBuffer und einem IndexBuffer heraus rendern wollen? Dazu
können wir die Funktion DrawIndexedPrimitives verwenden (oder DrawIndexedUserPrimitives wenn wir ohne Vertexbuffer
arbeiten wollen). Doch um diese Funktion müssen wir uns erstmal nicht kümmern, da uns ja die Mesh Klasse die meiste Arbeit abnimmt.

Fonts und Textrendering

So ziemlich jedes Spiel heutzutage hat auch eine Textausgabe. Direct3D bietet uns einen sehr komfortablen Weg Text auszugeben. Dazu können wir die Klasse Font verwenden die uns die ganze Arbeit des Fontrendering abnimmt.
Zunächst brauchen wir eine Variable der Font Klasse:

```
Microsoft.DirectX.Direct3D.Font text;
```

Wir müssen hier den vollen Namespace angeben, da es sonst zur Kollision mit der

Font Klasse von System.Windows.Forms kommt. Nun erstellen wir eine Instanz davon:

```
text = new Microsoft.DirectX.Direct3D.Font(device,new
System.Drawing.Font("Tahoma",12));
```

Wir verwenden den einfachsten Konstruktor, bei dem wir nur ein Device angeben müssen und den Fontnamen. Als Fontnamen kann jeder Font im Verzeichnis C:\Windows\Fonts verwendet werden(<WINDOWSROOT>\Fonts). Wir verwenden erstmal Tahoma. Als Schriftgröße wählen wir 12. Die Schriftgröße wird in Geviert angegeben. Nun springen schon in die Render Funktion:

```
text.DrawText(null,"Hello World",100,100,Color.Red);
```

Der erste Parameter beschreibt ein Sprite Objekt. Es kann null sein wenn die Font Klasse ein eigenes verwenden soll (wir werden später noch sehen wie wir das selbst angegeben). Wenn DrawText öfters aufgerufen wird, sollte ein eigenes Spriteobjekt verwendet werden, um die Performance zu erhöhen(die Font Klasse muss ansonsten bei jedem DrawText Aufruf ein neues Spriteobjekt erstellen). Der zweite Parameter gibt den auszugebenden Text an. Der dritte und vierte Parameter die x- und y-Koordinaten des Texts. Der letzte Parameter schließlich die Farbe des Texts. Zu beachten ist, dass die DrawText Funktion unabhängig von unseren Transformationen arbeitet und nur 2D Text ausgeben kann. Wir sehen später noch eine Möglichkeit 3D Text auszugeben(und zwar über die Mesh Klasse).
Das wars auch schon zu der Font Klasse, wir sehen sie uns aber noch genauer an. Die Font Klasse bietet uns die Möglichkeit Ressourcen in den Video RAM vorzuladen(Preloading). Dies können wir zB für Menütexte ausnutzen die ja statisch sind. Das Preloading erhöht normalerweise die Performance des Rendering. Sehen wir uns nun einmal PreloadText an, womit wir Text bereits vor dem Aufruf von DrawText laden können(was natürlich nur Sinn macht bei großen Mengen an Text). Zunächst deklarieren wir eine string Variable:

```
string message = "Preloaded Text";
```

Den DrawText Aufruf müssen wir nur geringfügig umändern:

```
text.DrawText(null,message,100,100,Color.Red);
```

DrawText erkennt nun das die Variable message bereits im Video RAM geladen wurde und muss dadurch den string nicht mehr bei jedem Frame zum Video RAM schicken sondern nimmt einfach die bereits im Video RAM vorhandende Variable. Was wir uns noch ansehen können ist die Ausgabe der Framerate. Die Framerate ist das wichtigste Mittel zum messen der Performance einer Anwendung und wird überlicherweise in frames per second(fps) angegeben. fps geben an, wie viele Bilder pro Sekunde angezeigt werden. Ab rund 25 fps erkennt das menschliche

Auge eine Szene als flüssig. Wie können wir nun die Frames berechnen? Zunächst brauchen wir einmal die Funktion QueryPerformanceCounter aus der WinAPI. Diese liefert uns den aktuellen Wert des high-performance Counters.

```
[DllImport("user32.dll")]
private static extern bool QueryPerformanceCounter(ref System.Int64
counter);
```

Danach brauchen wir noch QueryPerformanceFrequency

```
[DllImport("kernel32.dll")]
        private static extern bool QueryPerformanceFrequency(out
System.Int64 freq);
```

Nun kommen wir zum Berechnen: Wir benötigen die Zeit die zwischen zwei Frames vergangen ist und müssen die Freuquenz des high-performance Counters durch den Delta Wert der zwischen den zwei Frames vorherrscht dividieren.
Ganz am Anfang der Render Funktion holen wir uns die Frequenz des Counters und den aktuellen Wert:

```
System.Int64 LastFrame;
System.Int64 Freq;

QueryPerformanceFrequency(out Freq);
QueryPerformanceCounter(out LastFrame);
```

Nun rendern wir alles. Zum Schluss berechnen wir noch die Frames

```
System.Int64 CurrentFrame;
QueryPerformanceCounter(out CurrentFrame);

fps = Freq / (CurrentFrame - LastFrame);
```

Und nun können wir die Framerate ausgeben

```
text.DrawText(null,"Framerate: " +
Convert.ToString(fps),150,150,Color.Red);
```

Vertikale Synchronisierung

Wenn wir uns das letzte Beispiel noch einmal ansehen, dann sehen wir das die Framerate bis zu einem bestimmten Punkt geht und nicht darüber (bei mir 85). Das Ganze hat den Grund das die Anwendung das wir Direct3D implizit sagen, das es für uns die Framerate mit der Bildwiederholfrequenz des Bildschirms synchronisieren soll. Das können wir abstellen indem wir unsere PresentParameters verändern.
Wir müssen den Wert PresentationInterval verändern

```
presentParams.PresentationInterval = PresentInterval.Immediate;
```

Das hebt diese Beschränkung auf und man erhält höhere Frameraten.

Alpha Blending

Alphablending ist ein wichtiger Effekt den man für viele Dinge gebrauchen kann. So kann man damit Transparenzeffekte erzielen.
Sehen wir uns erstmal an was man mit Transparenz überhaupt meint. Jede Farbe besteht zunächst aus 3 Komponenten. Rot - Grün - Blau. Nun führen wir einfach noch eine Komponente ein, den Alpha Wert. Dieser Wert beschreibt die
Opazität eines Objketes. Die Opazität beschreibt die Lichtundurchlässigkeit eines Objektes. Ein Alpha Wert von 255 bedeutet das das die Farbe vollkommen opak ist dh man sie vollkommen sieht. Ein Alpha Wert von 0 bedeutet das die Farbe vollkommen durchsichtig ist. Sobald wir Alphablending einschalten, berechnet Direct3D alle Farbwerte nach der Formel:

*final color = source color * source blend factor + destination color * destination blend factor*

source color ist jene Farbe, die in den Backbuffer geschrieben werden soll.
destination color ist jene Farbe, die derzeit im Backbuffer steht.
Die beiden blend factor können wir mit Direct3D selbst bestimmen und so verschiedenste Effekte erzielen.
final color ist dann die Farbe, die letztendlich in den Backbuffer geschrieben wird.

Wie aus der Formel ersichtlich, müssen Objekte mit Alphablending müssen immer nach vollkommen opaken Objekten
gerendert werden.
Für unsere Beispielanwendung benötigen wir zwei Texturen. Die erste Textur enthält eine Mauer und die zweite
ein Windowslogo. Der Code ist zunächst simpel. Wir erstellen erstmal ein Meshobjekt und setzen entsprechende
Vertices und Indices. Außerdem laden wir noch die Texturen:

```
public void OnCreateDevice(object sender, EventArgs e)
               {
                       Device dev = (Device)sender;

                       CustomVertex.PositionTextured[] verts = { new
CustomVertex.PositionTextured(-1f,  0f, -1f,0,1),
new CustomVertex.PositionTextured( 1f,  0f, -1f,1,1),
                                new CustomVertex.PositionTextured( 1f,  0f,
1f,1,0),
                       new CustomVertex.PositionTextured(-1f,  0f,
1f,0,0),
};

short[] indices = { 0,1,2,
                          0,2,3 };

Object = new
Mesh(2,verts.Length,MeshFlags.WriteOnly,CustomVertex.PositionTextured.Forma
t,
device);
```

```
Object.VertexBuffer.SetData(verts,0,LockFlags.None);
        Object.IndexBuffer.SetData(indices,0,LockFlags.None);

baseTexture = TextureLoader.FromFile(dev,"base.jpg");
alphaTexture = TextureLoader.FromFile(dev,"windowslogo.dds");

}
```

Und schon können wir in die Renderfunktion springen. Dort rendern wir zuerst das Objekt mit der Mauertextur:

```
private void Render()
                {
                        if (device == null)
                                return;

                        device.Clear(ClearFlags.Target ,
System.Drawing.Color.Blue, 1.0f, 0);

                        device.BeginScene();

                        device.Transform.View = Matrix.LookAtLH( new
Vector3( 0.0f, 3.0f,-5.0f ), new Vector3( 0.0f, 0.0f, 0.0f ), new Vector3(
0.0f, 1.0f, 0.0f ) );
                        device.Transform.Projection =
Matrix.PerspectiveFovLH( (float)Math.PI / 4, 1.0f, 1.0f, 100.0f );

                        Matrix world;
                        world = Matrix.Scaling(2.0f,2.0f,2.0f);
                        world *= Matrix.RotationX(-1.04f);
                        world *= Matrix.Translation(0.0f,0.0f,0.0f);

                        device.Transform.World = world;
                        device.SetTexture(0,baseTexture);
                        Object.DrawSubset(0);
```

Bis hierher gibt es nichts neues. Nun rendern wir das Objekt nochmal, jedoch mit dem Unterschied, das wir jetzt mit eingeschaltetem Alphablending rendern.

```
device.RenderState.AlphaBlendEnable = true;
device.RenderState.SourceBlend = Blend.InvSourceAlpha;
device.RenderState.DestinationBlend = Blend.DestinationAlpha;

world = Matrix.Identity;
world *= Matrix.Scaling(1.5f,1.5f,1.5f);
world *= Matrix.RotationX(-1.04f);
world *= Matrix.Translation(0.0f,0.0f,0.0f);
device.Transform.World = world;

device.SetTexture(0,alphaTexture);
Object.DrawSubset(0);
device.RenderState.AlphaBlendEnable = false;
```

Ganz am Anfang schalten wir Alpha Blending ein. Dadurch sagen wir Direct3D das es die oben genannte Formel zum Herausfinden der final color verwenden soll.

Die nächsten zwei Zeilen definieren die blend factoren.
Als source blend setzen wir den Faktor (1 - As, 1 - As, 1 - As, 1 - As). As ist der Alpha Wert der source Farbe.
Als destination blend setzen wir den Faktor (Ad, Ad, Ad, Ad). Ad ist hierbei der Alpha Wert der destination
Farbe.
Danach setzen wir noch unsere Textur, die überblendet werden soll und rendern das ganze. Zum Schluss schalten
wir Alpha Blending wieder aus, denn wir wollen ja nicht das die Mauer auch mit Alphablending gerendert werden
soll.

Natürlich gibt es noch mehr Blend Faktoren. Diese kann man unter
http://msdn.microsoft.com/archive/defaul...blend/blend.asp
nachschlagen.

Textur Transformationen

Nach etwas längerer Pause werde ich mich nun wieder diesem Tutorial hier zuwenden. Ich werde versuchen es jetzt regelmäßig zu erweitern. Ich will nicht großartige Programme hier vorstellen sondern einfach in kurzen und überschaubaren Programmen verschiedene Techniken vorstellen.
Das nächste Kapitel hier ist eher kurz aber es handelt von einem sehr interessanten und nützlichen Thema: Texturtransformationen.
Wie wir wissen durchlaufen ja unsere Vertices eine Transformationspipeline. In dieser Pipeline können wir Matritzen setzen um diese Vertices zu transformieren. Direct3D bietet uns jetzt jedoch die Möglichkeit auch Texturkoordinaten zu transformieren. Damit können wir jetzt alle Transformationen, die wir bei Vertices anwenden konnten auch bei Texturen anwenden. Eine Anwendung für eine solche Transformation ist zB ein sich bewegendes Hintergrundbild. Wenn man sich zB das Spiel Moorhuhn ansieht dann erkennt man das der Hintergrund sich beweget, er scrollt. Anstatt nun kompliziert mehrere Primitive zu zeichnen, diese entsprechend anzuzeigen und zu transformieren könnten wir ja einfach ein Rechteck zeichnen, auf dieses eine riesige Hintergrundtextur zeichnen und diese Texture dann entsprechend den Usereingaben nach links oder rechts zu verschieben. Eine weitere Anwendung wären Himmeltexturen. Wolken sind ja nicht statisch sondern bewegen sich. Mit Texturkoordinatentransformation können wir nun die Wolken bewegen. Aber dazu in einem späteren Tutorial mehr, da dies bereits zum Thema Skybox gehört.

Kommen wir jetzt zum Spaß der ganzen Sache und gehen direkt in den Code.
Das Beispielprogramm soll einfach ein texturiertes Rechteck anzeigen auf dem die Textur ständig scrollt. Ich zeige außerdem beide Varianten auf, wie man die Texturkoordinaten transformieren kann: Entweder man verwendet die Texturkoordinatentransformation von Direct3D oder man passt die Texturkoordinaten jeden Frame selber an.
Eine Anmerkung noch: Ich verwende hier User Primitives (dh die Vertices liegen im RAM und werden bei jedem Frame zur Grafikkarte geschickt). In Spielen oder Grafikdemos sollte man immer einen Vertexbuffer verwenden. Ich verwende hier

einfach User Primitives weil es einfacher ist als zuvor einen Vertexbuffer zu erstellen usw.
Zunächst brauchen wir 3 neue Variablen:

```
float   counter;
bool    useTransformation = true;
Texture texture;
```

counter speichert unseren aktuellen Translationswert.(dh um den Wert der in counter steht wird später unsere Textur verschoben). useTransformation zeigt an ob die Anwendungen den Hintergrund mit der Texturkoordinatentransformation rendern soll oder ob es die Texturkoordinaten selbst anpassen soll. Und texture schließlich speichert unsere Hintergrundtextur.

Als nächstes kommt der übliche Direct3D Code. Present Parameter, CreateDevice usw.
Das einzig noch interessante hier ist das Laden der Textur:

```
texture = TextureLoader.FromFile(device,"cloud3.bmp");
```

cloud3.bmp ist eine Hintergrundtextur aus dem DirectX 8 SDK und ist eine einfache Wolkentextur.

Kommen wir jetzt direkt zur Renderfunktion. Die ersten paar Zeilen beinhalten nichts Neues:

```
private void Render()
               {
                       if (device == null)
                              return;

                       device.Clear(ClearFlags.Target ,
System.Drawing.Color.Black, 1.0f, 0);

                       device.BeginScene();
                       device.Transform.World = Matrix.Identity;
                        device.Transform.View = Matrix.LookAtLH ( new
Vector3 ( 0, 0, 5f ), new Vector3 ( 0, 0, 0 ), new Vector3 ( 0, 1, 0 ) );
                       device.Transform.Projection =
Matrix.PerspectiveFovLH ( (float)Math.PI / 4, this.Width / this.Height,
1.0f, 100.0f );
```

Jetzt kommen wir zum eigentlich Rendercode. Zunächst setzen wir ein Vertexformat und eine Textur. Dann kommt als erstes etwas Neues:

```
device.VertexFormat = CustomVertex.PositionTextured.Format;
device.SetTexture(0,texture);
device.SamplerState[0].AddressU = TextureAddress.Wrap;
```

Und zwar Samplerstates. Samplerstates definieren wie eine Textur *gesampelt* wird also quasi abgetastet. Aber sehen wir uns erstmal an was diese Anweisung bedeutet.

Texturkoordinaten sind immer als ein uv-Koordinatenpaar angegeben. Mithilfe dieser Koordinaten können wir bestimmen wie eine Textur auf ein Primitiv gemappt wird. (-> Textur Mapping). Wie wir wissen ist das Verhalten der Texturkoordinaten nur im Intervall [0.0f; 1.0f] definiert. Wenn Texturkoordinaten außerhalb dieses Bereichs sind müssen wir Direct3D sagen was es mit den Koordinaten anstellen soll. Und das tun wir hier. Mit Wrap sagen wir das Direct3D die Textur einfach wiederholen soll. Hätten wir jetzt als u-Koordinaten die den Intervall [0.0f; 5.0f] einspannen wird die Textur 5x wiederholt. Es gibt noch TextureAddress.Mirror, die im Prinzip dasselbe wie Wrap ist, aber die Textur wird zusätzlich noch gespiegelt. Es gibt daneben noch Clamp, MirrorOnce und Border.

Jetzt beginnt das zeichnen: Zunächst überprüfen wir ob wir Texturkoordinatentransformation verwenden oder ob wir die Texturkoordinaten selbst berechnen. Ich bespreche erstmal die Koordinatentransformation.

```
if(useTransformation)
{
                                device.TextureState[0].TextureTransform =
TextureTransform.Count2;

                                Matrix m = Matrix.Identity;
                                m.M31 = counter;
                                device.Transform.Texture0 = m;


        device.DrawUserPrimitives(PrimitiveType.TriangleStrip,2,CreateQuad(
));
                                device.TextureState[0].TextureTransform =
TextureTransform.Disable;
}
```

Die erste Zeile schaltet erstmal die Texturkoordinatentransformation ein. Daneben bestimmen wir gleichzeitig wie viele Texturkoordinaten wir Direct3D zur Verfügung stellen. In unserem Beispiel sind das 2D-Koordinaten. (Es gibt 1D,2D,3D und 4D und noch Projected, bei dem wir Texturkoordinaten ähnlich dem Pixelprojektion projizieren können)
Nun kommen wir mathematischen Teil dieses Tutorials. Aber keine Angst, es nicht schwer und selbst wenn man es nicht verstehe, man muss nur wissen wie man es richtig anwendet. *m* ist unsere Translationsmatrix für die Textur. Mit Hilfe dieser werden wir verschieben wir die Texturkoordinaten in u- und v-Richtung. Die Matrix die wir Direct3D schicken muss diese Form haben:

1 0 0 0
0 1 0 0
(delta)u (delta)v 1 0
0 0 0 1

Wie man mehr oder weniger leicht erkennen kann ist diese Matrix einfach eine Identitätsmatrix wobei aber die 31. und 32. Postion durch 2 Werte ersetzt werden. Diese beiden Werte, (delta)u und (delta)v bestimmen die Translation entlang der

entsprechenden Koordinatenachse. In unserem Beispiel wollen wir die Textur nur entlang der u-Richtung verschieben (dh nach links/rechts). Würden wir *m.M31* durch *m.M32* ersetzen würden wir nicht mehr nach links/rechts verschieben sondern in der v-Richtung dh oben/unten. Wer aber jetzt aufgepasst hat der wundert sich warum die delta werte in der 3. und nicht in der 4. Zeile sind. Normalerweise sind ja bei einer Translationsmatrix diese Werte in der 4. Zeile. Aber: Wir haben ja keine 3D-Raum. Unsere Textur liegt im 2D-Raum daher müssen wir auch eine 2D-Translationsmatrix anfertigen. (Was zunächst mal eine 3x3 Matrix ist), aber Direct3D eine 4x4 Matrix für alle seine Operationen verlangt müssen wir noch die 4. Zeile hinzufügen. Dh wir haben hier einfach eine 3x3 2D-Translationsmatrix die einfach nicht als 3x3 angeschrieben wird sondern als 4x4. Zu beachten ist das wir auch in der 4. Zeile noch eine 1 als letztes Element hinschreiben müssen um die Identitätsmatrix nicht zu beschädigen. Das ganze muss übrigens auch ein Zeilenvektor sein, da Direct3D mit diesen arbeitet. Im Gegensatz dazu arbeitet OpenGL etwa nur mit Spaltenvektoren. Aber das nur nebenbei und tut hier eigentlich nichts zu Sache.
Soviel dazu. Eigentlich nicht schwer. Als nächstes setzten wir noch die Matrix, ähnlich wie zB die World Matrix. Man kann für jede Texturstufe eine Matrix setzen und somit diese Transformation auch verwenden wenn man Multitexturing verwendet.

Nun zum Zeichnen. Ebenfalls nichts Aufregendes bis auf die Funktion CreateQuad. Diese Funktion liefert uns einfach 4 Vertices mit Texturkoordinaten zurück.

```
private CustomVertex.PositionTextured[] CreateQuad()
{
CustomVertex.PositionTextured[] verts = new
CustomVertex.PositionTextured[4];

verts[0].Position = new Vector3 ( -1.0f,-1.0f, 3.0f ); verts[0].Tu =1.0f;
verts[0].Tv = 1.0f;
verts[1].Position = new Vector3 ( -1.0f, 1.0f, 3.0f ); verts[1].Tu =1.0f;
verts[1].Tv = 0.0f;
verts[2].Position = new Vector3 ( 1.0f, -1.0f, 3.0f ); verts[2].Tu =0.0f;
verts[2].Tv = 1.0f;
verts[3].Position = new Vector3 ( 1.0f,  1.0f, 3.0f ); verts[3].Tu =0.0f;
verts[3].Tv = 0.0f;

return verts;
}
```

Danach schalten wir noch die Texturkoordinatentransformation aus. Dies ist wichtig falls wir noch weitere texturierte Objekte zeichnen würden, da ja die Transformationsmatrix auf jede Textur angewendet werden würde.

Jetzt kommen wir noch zu dem Teil der Anwendung die ohne die von Direct3D bereitgestellte Funktionalität arbeitet. Wir berechnen uns unsere Texturkoordinaten einfach selbst. Und das ist einfacher als es sich anhört. Zurück in der Renderfunktion sieht das erstmal so aus:

```
else
       device.DrawUserPrimitives(PrimitiveType.TriangleStrip,2,CreateTrans
formedQuad());
```

Wie man vermuten kann spielt sich wohl alles in CreateTransformedQuad ab. Diese Funktion ist dasselbe wie CreateQuad, nur mit dem Unterschied das diesmal die Texturkoordinaten nicht statisch sind, sondern wir diese in der Funktion berechnen (daher auch Transformed).

```
private CustomVertex.PositionTextured[] CreateTransformedQuad()
{
        CustomVertex.PositionTextured[] verts = new
CustomVertex.PositionTextured[4];

                        verts[0].Position = new Vector3 ( -1.0f,-1.0f, 3.0f
); verts[0].Tu =0-counter; verts[0].Tv = 1.0f;
                        verts[1].Position = new Vector3 ( -1.0f, 1.0f, 3.0f
); verts[1].Tu =0-counter; verts[1].Tv = 0.0f;
                        verts[2].Position = new Vector3 ( 1.0f, -1.0f, 3.0f
); verts[2].Tu =1-counter; verts[2].Tv = 1.0f;
                        verts[3].Position = new Vector3 ( 1.0f,  1.0f, 3.0f
); verts[3].Tu =1-counter; verts[3].Tv = 0.0f;

                        return verts;
}
```

Besondere Aufmerksamkeit lenken wir auf die Zeilen mit den u-Koordinaten. Hier berechnen wir einfach die Koordinaten und verschieben so die Textur entlang der u-Achse.
Nun aber wieder zurück in die Renderfunktion:

```
counter += 0.0015f;

                        this.Text = "MDXTutoria12:
Texturkoordinatentransformation"  + "- useTransformation = " +
useTransformation.ToString();

                        device.EndScene();
                        device.Present();
}
```

Zunächst erhöhen wir hier counter damit es auch eine Bewegung gibt. Hier ist es aber zu beachten, dass man das in Spielen oder Demos nicht so machen sollte. Das Problem ist einfach das counter auf langsamen Rechnern langsamer erhöht wird und auf schnelleren Rechnern eben schneller. Wir müssen die Erhöhung von der Framerate abhängig machen, aber das zeige ich in einem anderen Tutorial.
Danach zeigen wir in der Fensterleiste noch an wie die Szene derzeit gerendert wird und schließlich beenden wir die Szene und rendern sie endgültig.

Nun haben wir alles, bis auf die ProcessCmdKey. Hier verarbeiten wir Benutzereingaben:

```
protected override bool ProcessCmdKey(ref Message msg, Keys keyData)
{
        if(keyData == Keys.Escape)
                Application.Exit();
```

```
        if(keyData == Keys.T)
                useTransformation = !useTransformation;

        return base.ProcessCmdKey( ref msg, keyData );
}
```

Nichts Schweres an dem Code. Wenn der User auf die Taste t drückt dann schaltet die Anwendung zwischen den Transformationen hin und her.

Und das war es auch schon. Wie ich versprochen habe, nicht allzu schwer dieses Tutorial, aber doch lassen sich damit coole Effekte machen. (Ungefähr wie beim Alphablending). Noch etwas: Man sollte in einer produktiven Anwendung immer Texturkoordinatentransformation verwenden außer es geht irgendwie nicht. Die Texturkoordinatentransformation spielt sich nämlich normalerweise direkt auf der Hardware ab und ist dadurch viel schneller. Ein weiterer Vorteil ist außerdem das wir damit die Koordinaten nicht ständig verändern müssen. Hier kommt das nicht so gut raus da ich User Primitives verwende, aber stellen wir uns vor, wir würden einen Vertexbuffer verwenden. Wir müssen dann in jedem Frame den Vertexbuffer sperren, u-Koordinaten auslesen und verändern und wieder Zurückschreiben. Das verlangsamt natürlich für die Performanz der Anwendung sehr.