Denne guide er oprindeligt
udgivet på Eksperten.dk

COMPUTERWORLD

Gennemsigtig, roterende terning med DirextX9 og C#

Lær at lave en roterende terning med transperante textures i C#, ved hjælp af DirectX9. Jeg kunne ikke finde en ordentlig tutorial på nettet, derfor denne artikel ;)

Skrevet den **03. Feb 2009** af **innercitydk** I kategorien **Programmering / C#** | ★★★☆☆

I denne artikel skal vi snuse lidt til mulighederne for at bruge DirectX sammen med C#. Vi vil lave et simpelt program som roterer en terning i 3d. Vi smider semi transperante png billeder på siderne af den for samtidig at vise muligheden for at lave Alpha blending. Vi laver også en Fps counter så vi kan følge med i antal frames i sekundet. Projectet laves i Visual Studio 2005.

Her er hvad vi prøver at opnå:

Resultat

Først og fremmest skal vi have hentet DirectX9 SDK (Software Development Kit). Du kan downloade den nyeste version her:

http://msdn.microsoft.com/directx/

eller her:

Fileupdates

Når du har downloadet pakken skal den pakkes ud. Husk hvor du udpakker filerne til, vi skal bruge dem om et øjeblik.

Først åbner vi Visual Studio og laver en ny C# windows applikation.

Vælg: File-->New-->Project, og vælg Windows Application. Omdøb applikationen til Managed3d og klik på ok.

Nu skal vi have tilføjet referencen til DirextX biblioteket. Højreklik på references i solution exploreren og vælg Add Reference. Tryk på fanen browse og find stien hvortil du udpakkede DirectX. Herefter skal du vælge mappen Developer Runtime-->x86-->DirectX for Managed Code, marker filerne Microsoft.DirectX.Direct3D.DLL og Microsoft.DirectX.DLL (marker begge ved at holde Ctrl knappen nede mens du klikker på dem), og tryk på ok.

Nu skal referencerne blot tilføjes i toppen af Form1.cs lige efter "using System;" eks:

```
using System;
using Microsoft.DirectX;
using Microsoft.DirectX.Direct3D;
```

Nu er vi klar til at kode.

Vi skal have overridet OnPaint metoden. OnPaint udføres normalt når der sker ændringer i skærmbilledet, f.eks ved resizing af billedet eller brugerinput. Ved 3d applikationer er det anderledes. Billedet skal opdateres mange gange i sekundet. For at kunne styre opdateringsprocessen overrider vi OnPaint metoden.

Indsæt følgende kode lige efter Form1 Constructoren:

```
protected override void OnPaint(PaintEventArgs e)
{
}
```

OnPaint metoden kaldes hver gang der sker ændringer i skærmbilledet, men den tager ikke højde for at skærmbilledet f.eks bliver minimeret. Vi kan heldigvis tvinge OnPaint eventen til at køre ved at kalde formens Invalidate metode.

I OnPaint metoden skal du tilføje følgende kode:

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

Windows kalder en event i baggrunden hver gang skærmbilledet ændres. For at omgå dette skal vi sikre os at den eneste event der må ændre i skærmbilledet er vores OnPaint metode. Samtidig sætter vi default størrelsen på vinduet til 640x480.

Indsæt fælgende kode lige efter InitializeComponent(); i form1 constructoren:

```
this.ClientSize = new Size( 640, 480 );
this.SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.Opaque, true);
```

Metoden InitializeComponent skal vi ikke bruge. Slet InitializeComponent(); fra Form1 constructoren. Vi laver vores egen initiering af formen senere.

Nu er det grundlæggende framework til vores terning færdigt.

Nu skal det dreje sig om timing. Som alle ved er der forskellige hastigheder på udførelsen af et program alt afhængig af hvilken maskine man arbejder med. F.eks vil en toptunet Pentium4 3Ghz afvikle koden mange gange hurtigere end en Pentium3 500Mhz. For at styre denne proces, og derved opnå en flydende afvikling af vores grafik, skal vi bruge en form for timer. Jeg vil ikke gå i dybden omkring de forskellige timere, mere kan læses ved at besøge linket i bunden af denne artikel(msdn). Det er ret simpel grafik vi skal lave så vi behøver ikke en særlig præcis timer. DateTime er ok til dette projekt.

Tilføj følgende variabler i toppen af Form1 klassen:

```
static private float elapsedTime;
```

```
static private DateTime currentTime;
static private DateTime lastTime;
```

Elapsed time bruges bla. til at styre hvordan vores terning skal bevæge sig. Det kommer vi tilbage til senere. Felterne skal initieres hver gang OnPaint metoden kaldes, altså hver gang skærmbilledet opdaterer.

Tilføj følgende kode i OnPaint metoden lige før this.invalidate():

```
currentTime = DateTime.Now;
TimeSpan elapsedTimeSpan = currentTime.Subtract( lastTime );
elapsedTime = (float)elapsedTimeSpan.Milliseconds * 0.001f;
lastTime = currentTime;
```

Nu har vi hvad vi skal bruge for at kunne styre opdateringen af skærmbilledet.
Nu skal vi have tilføjet en fps tæller. Frames per second kan fortælle os hvor mange gange skærmbilledet opdateres i sekundet.

Tilføj følgende klasse lige efter Form1 Klassen:

```
class FrameRate
   {
      private static int lastTick;
      private static int lastFrameRate;
      private static int frameRate;

      public static int CalculateFrameRate()
      {
         if (System.Environment.TickCount - lastTick >= 1000)
         {
            lastFrameRate = frameRate;
            frameRate = 0;
            lastTick = System.Environment.TickCount;
         }

         frameRate++;

         return lastFrameRate;
      }
   }
```

Vi skal have udskrevet fps. For ikke at forvirre for meget vælger vi at udskrive fps i formens titel felt øverst.

Tilføj følgende kode i starten af OnPaint metoden:

```
this.Text = "DirectX9 med C#" + string.Format(", FPS: {0}", FrameRate.CalculateFrameRate());
```

Næste skridt er at lave vores egen InitializeComponent metode. Tilføj følgende kode lige efter OnPaint metoden:

```
public void Init()
{
}
```

Først skal vi tilføje kode som tjekker hvorvidt grafikkortet i computeren uderstøtter hardware acceleration. Hvis ikke det understøttes bruger vi software acceleration.

Tilføj følgende kode i Init() metoden:

```
Caps caps = Manager.GetDeviceCaps( Manager.Adapters.Default.Adapter, DeviceType.Hardware );
CreateFlags flags;

if( caps.DeviceCaps.SupportsHardwareTransformAndLight )
    flags = CreateFlags.HardwareVertexProcessing;
else
flags = CreateFlags.SoftwareVertexProcessing;
```

Nu er vi klar til at initiere 3d grafik enheden. Først laver vi en variabel af typen Device.

Tilføj følgende variabel øverst i Form1 klassen:

```
private Device d3d_enhed = null;
```

Nu kan vi ændre I grafik enheden (indstille grafikken) fra alle klassens metoder.

Vi skal tilføje kaldet til Init() metoden i vores Main klasse. Find Program.cs i solution exploreren og indsæt følgende kode i Main metoden, slet det der står der i forvejen:

```
Form1 frm = new Form1();
frm.Show();
frm.Init();
Application.Run(frm);
```

Næste kode vi vil tilføje indstiller og starter grafik enheden d3d_enhed. Indstillinger for det vindue vi vil starte gemmes i et objekt af typen PresentParameters fra direct3d biblioteket. Herefter instantieres d3d_enhed med 5 parametre. Adapter nr(gf kort),Type enhed, ejer(Form1),flags(hardware eller software accelerering) og parametre som bruges ved presentationen af grafikken.

Tilføj følgende kode i bunden af Init metoden:

```
PresentParameters d3dpp = new PresentParameters();

d3dpp.BackBufferFormat = Format.Unknown;
d3dpp.SwapEffect = SwapEffect.Discard;
d3dpp.Windowed = true;
d3dpp.EnableAutoDepthStencil = true;
d3dpp.AutoDepthStencilFormat = DepthFormat.D16;
d3dpp.PresentationInterval = PresentInterval.Immediate;

d3d_enhed = new Device(0, DeviceType.Hardware, this, flags, d3dpp);
```

Nu skal vi have tilføjet en render metode som er den metode hvor alt grafikken bliver genereret.

Indsæt følgende metode lige efter Init() metoden:

```
private void Render()
{
d3d_enhed.BeginScene();

d3d_enhed.EndScene();
d3d_enhed.Present();
}
```

Hele vores firkant bliver tegnet mellem BeginScene og EndScene. Vi kalder Render fra OnPaint metoden.

Tilføj følgende kode I OnOaint metoden lige før this.Invalidate():

```
this.Render();
```

Næste skridt er at lave en Metode der bliver kaldt hver gang grafik enheden resettes. Metoden OnResetDevice indeholder Direct3D relaterede objekter, hvis værdier kan blive ugyldige ved reset, og derfor skal initieres igen.

Tilføj følgende metode under Init() metoden:

```
public void OnResetDevice(object sender, EventArgs e)
```

```
{
}
```

Vi skal nu have lavet en eventhandler som fanger reset eventen. Tilføj følgende kode i bunden af Init metoden:

```
d3d_enhed.DeviceReset += new System.EventHandler( this.OnResetDevice );
```

Vi kalder eventen med det samme for at gå videre med grafik initieringen. Tilføj følgende kode i slutningen af Init metoden:

```
OnResetDevice(d3d_enhed, null );
```

Nu skal vi have tilføjet den kode der skal udføres når eventen OnResetDevice opstår. Tilføj følgende kode lige før Render() metoden:

```
public void OnResetDevice(object sender, EventArgs e)
{
}
```

Første kode vi vil tilføje til metoden gør at perspektivet ved vores transformation (terningen skal dreje rundt) ser naturligt ud.

Tilføj følgende kode i starten af OnResetDevice metoden:

```
d3d_enhed.Transform.Projection = Matrix.PerspectiveFovLH( Geometry.DegreeToRadian( 45.0f
),(float)this.ClientSize.Width / this.ClientSize.Height,0.1f, 100.0f );
```

Du kan ændre værdien 45.0f, hvorefter du så kan se ændringerne I perspektivet.

Næste kode enabler Zbuffer og disabler lightning. Zbuffer er dybde bufferen, den gør at vi kan lave 3d grafik (x,y,z=bredde,højde,dybde). Vi skal ikke bruge en lyskilde på terningen derfor disabler vi lighting:

```
d3d_enhed.RenderState.ZBufferEnable = true;
d3d_enhed.RenderState.Lighting = false;
```

Nu mangler vi kun en ting for at fuldføre vores "render loop", og det er at nulstille d3d_enheden hver gang render metoden kaldes. Grunden til at vi gør dette er at ellers ville vi kun få displayet første frame ved afvikling. Ved at nulstille enheden gøres den klar til at displaye nye data ud på skærmen.

Tilføj følgende kode øverst i Render metoden:

```
d3d_device.Clear( ClearFlags.Target | ClearFlags.ZBuffer, Color.FromArgb(255, 0, 0, 0), 1.0f, 0 );
```

Når koden afvikles nu vises der blot en sort skærm. Det er fordi vores loop kører som det skal. Hvis du ændrer i værdien Color.FromArgb(255.0.0.0) kan du skifte baggrundsfarven på vinduet. 255.255.255.255 er f.eks hvid.

Nu er vi næsten klar til at lave en terning. Men først lidt om terningen selv. En terning er opbygget af 6 sider. Hver side er en firkant, og en firkant består af 4 koordinater. Før at vi kan tegne en firkant skal vi på en eller anden måde gemme de koordinater der udgør firkanten. Vi gemmer disse koordinater i et array. En koordinat består af 3 værdier, placering på X-akse, placering på Y-akse og placering på Z-akse. Disse 3 punkter gemmes i en Vertex. En vertex består af 2 ting, koordinaterne og placeringen af texturen (overfladen).
Vores terning består som sagt af 6 sider som hver især er opbygget af 4 vertexer (koordinater).

Tilføj følgende kode i starten af Form1 klassen, lige efter static private DateTime lastTime:

```
struct Vertex
   {
         public static readonly VertexFormats FVF_Flags = VertexFormats.Position |
VertexFormats.Texture1;
   float x, y, z;
   float tu, tv;

   public Vertex( float _x, float _y, float _z, float _tu, float _tv )
   {
      x = _x; y = _y; z = _z;
      tu = _tu; tv = _tv;
   }
}
```

Vi gemmer de egentlige informationer om siderne i et array, som så loades ind I en vertexbufferen til videre bearbejdning af DirectX. I arrayet skal vi have indsat 4 vertex objekter for alle 6 sider af terningen. Jeg vil ikke beskrive det yderligere, men leg med punkterne senere. Ændre f.eks et 1 til et 0 se så ændringerne i terningen. Det giver en bedre forståelse for de individuelle punkter i terningen.

Tilføj koden herunde lige efter efter den sidste kode vi indsatte:

```
Vertex[] terning =
{
   new Vertex(-1.0f, 1.0f,-1.0f,  0.0f,0.0f ),
   new Vertex( 1.0f, 1.0f,-1.0f,  1.0f,0.0f ),
```

```
    new Vertex(-1.0f,-1.0f,-1.0f,  0.0f,1.0f ),
    new Vertex( 1.0f,-1.0f,-1.0f,  1.0f,1.0f ),

    new Vertex(-1.0f, 1.0f, 1.0f,  1.0f,0.0f ),
    new Vertex(-1.0f,-1.0f, 1.0f,  1.0f,1.0f ),
    new Vertex( 1.0f, 1.0f, 1.0f,  0.0f,0.0f ),
    new Vertex( 1.0f,-1.0f, 1.0f,  0.0f,1.0f ),

    new Vertex(-1.0f, 1.0f, 1.0f,  0.0f,0.0f ),
    new Vertex( 1.0f, 1.0f, 1.0f,  1.0f,0.0f ),
    new Vertex(-1.0f, 1.0f,-1.0f,  0.0f,1.0f ),
    new Vertex( 1.0f, 1.0f,-1.0f,  1.0f,1.0f ),

    new Vertex(-1.0f,-1.0f, 1.0f,  0.0f,0.0f ),
    new Vertex(-1.0f,-1.0f,-1.0f,  1.0f,0.0f ),
    new Vertex( 1.0f,-1.0f, 1.0f,  0.0f,1.0f ),
    new Vertex( 1.0f,-1.0f,-1.0f,  1.0f,1.0f ),

    new Vertex( 1.0f, 1.0f,-1.0f,  0.0f,0.0f ),
    new Vertex( 1.0f, 1.0f, 1.0f,  1.0f,0.0f ),
    new Vertex( 1.0f,-1.0f,-1.0f,  0.0f,1.0f ),
    new Vertex( 1.0f,-1.0f, 1.0f,  1.0f,1.0f ),

    new Vertex(-1.0f, 1.0f,-1.0f,  1.0f,0.0f ),
    new Vertex(-1.0f,-1.0f,-1.0f,  1.0f,1.0f ),
    new Vertex(-1.0f, 1.0f, 1.0f,  0.0f,0.0f ),
    new Vertex(-1.0f,-1.0f, 1.0f,  0.0f,1.0f )
};
```

Nu er vi klar til at gemme koordinaterne til terningen I en VertexBuffer. En VertexBuffer holder på de informationer DirectX skal bruge for at generere terningen. Bla holder den på en instans af d3d_enheden(grafik enheden),og Arrayet med koordinaterne til terningen.
Indsæt følgende kode lige efter private Device d3d_enhed = null, i starten Form1 klassen:

```
private VertexBuffer vertexBuffer = null;
```

I OnReset metoden indsættes følgende kode (laver en instans af vertexBuffer objektet), lige efter d3d_enhed.RenderState.Lighting=false:

```
vertexBuffer = new VertexBuffer( typeof(Vertex),terning.Length, d3d_enhed,Usage.Dynamic |
Usage.WriteOnly,Vertex.FVF_Flags,Pool.Default );
```

Jeg vil ikke beskrive yderligere hvad de forskellige parametre gør, det er op til læseren selv at undersøge dette. Du kan læse funktionerne ved at holde musen hen over hver enkel parameter i vs2005.
Nu skal vi have kopieret vores vertex data ind i vertex bufferen. Dette gøres ved hjælp af GraphicStream

(en del af DirextX).
Tilføj følgende kode lige efter den sidste kode vil tilføjede:

```
GraphicsStream gStream = vertexBuffer.Lock( 0, 0, LockFlags.None );
gStream.Write( terning );
vertexBuffer.Unlock();
```

Nu har vi alt på plads, og vi er klar til at lave noget output i vores Render metode.
Først skal vi deklarere 3 variabler som er tilgængelige for alle medlemmer i Form1 klassen.
Tilføj følgende kode øverst i Form1 klassen:

```
private float xAkse = 0.0f;
private float yAkse = 0.0f;
private float zAkse = 0.0f;
private float zoom = 5.0f;
```

Variablerne skal bruges når vi skal laver bevægelse af vores terning. Først indstiller vi hvordan terningen skal rotere. Zoom afgør afstanden til terningen. Hvis vi ikke indstillede dette ville vi kun se en side af terningen, som så ville fylde hele billedet.

Tilføj følgende kode lige efter d3d_enhed.Clear; i Render metoden:

```
xAkse += 10.1f * elapsedTime;
yAkse += 20.2f * elapsedTime;
zAkse += 20.3f * elapsedTime;
```

Her bestemmer vi hastigheden på vores rotation. Hastigheden afgøres af vores timer. Hvis vi ikke gangede hastigheden op ville den køre meget langsomt.

Tilføj følgende kode lige efter d3d_device.BeginScene():

```
d3d_enhed.Transform.World = Matrix.RotationYawPitchRoll(
Geometry.DegreeToRadian(xAkse),Geometry.DegreeToRadian(yAkse),Geometry.DegreeToRadian(zAkse
)) * Matrix.Translation(0.0f, 0.0f, zoom);
```

Koden roterer vores objekt ud fra de givne værdier af xAkse, yAkse og zAkse.

Det næste vi skal gøre er at fortælle d3d_enhed hvor den skal streame grafikken fra, altså vores vertexBuffer. Samt hvilken type Vertex vi bruger. Tilføj følgende kode lige efter den sidste kode vi tilføjede:

```
d3d_device.SetStreamSource(0, vertexBuffer, 0);
d3d_device.VertexFormat = Vertex.FVF_Flags;
```

Nu er vi klar til at tegne vores terning. Terningen skal tegnes af to omgange da vi jo vil have at den skal være transparant.

Tilføj følgende kode lige efter den sidste kode vi tilføjede:

```
// Forsiden af terningen
d3d_enhed.RenderState.CullMode = Cull.Clockwise;

d3d_enhed.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 2);
d3d_enhed.DrawPrimitives(PrimitiveType.TriangleStrip, 4, 2);
d3d_enhed.DrawPrimitives(PrimitiveType.TriangleStrip, 8, 2);
d3d_enhed.DrawPrimitives(PrimitiveType.TriangleStrip, 12, 2);
d3d_enhed.DrawPrimitives(PrimitiveType.TriangleStrip, 16, 2);
d3d_enhed.DrawPrimitives(PrimitiveType.TriangleStrip, 20, 2);

// Bagsiden af terningen
d3d_enhed.RenderState.CullMode = Cull.CounterClockwise;

d3d_enhed.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 2);
d3d_enhed.DrawPrimitives(PrimitiveType.TriangleStrip, 4, 2);
d3d_enhed.DrawPrimitives(PrimitiveType.TriangleStrip, 8, 2);
d3d_enhed.DrawPrimitives(PrimitiveType.TriangleStrip, 12, 2);
d3d_enhed.DrawPrimitives(PrimitiveType.TriangleStrip, 16, 2);
d3d_enhed.DrawPrimitives(PrimitiveType.TriangleStrip, 20, 2);
```

DrawPrimitives sørger for at tegne den egentlige terning. Det vil sige den forbinder punkterne hvilket gør at vi opfatter det som en terning. PrimitiveType.TriangleStrip fortæller på hvilket måde punkterne skal forbinde. Værdierne 0,4,8,12,16 og 20 fortæller hvilket punkt(vertex), fra vores vertex array, der er startpunktet.

Nu er vi næsten færdige. Hvis vi afvikler projektet på nuværende tidspunkt ser vi en helt hvid terning der roterer i midten af skærmbilledet.

Det er ikke sjovt med en helt hvid terning så lad os få smidt en texture på siderne. En texture er en bitmap fil der dækker overfladen på hver enkelt firkant i vores terning. Der skal tilføjes kode 4 steder for at vi kan få en texture overflade. Først tilføjes følgende variabler i toppen af Form1 klassen:

```
Texture texture = null;
Bitmap testImage = null;
```

Indsæt derefter følgende metode i bunden af Form1 klassen:

```
private void LoadTexture()
{
 try
 {
  testImage = (Bitmap)Bitmap.FromFile( "eksperten.png" );
  texture = Texture.FromBitmap( d3d_device, testImage, 0, Pool.Managed );
 }
 catch
 {

  System.IO.Directory.SetCurrentDirectory(    System.Windows.Forms.Application.StartupPath +
@"\..\..\");

       testImage = (Bitmap)Bitmap.FromFile("eksperten.png");
  texture = Texture.FromBitmap(d3d_enhed, testImage, 0, Pool.Managed );
 }

  d3d_enhed.SamplerState[0].MinFilter = TextureFilter.Linear;
  d3d_enhed.SamplerState[0].MagFilter = TextureFilter.Linear;
}
```

Den try/catch der er I koden tjekker hvor programmet afvikles fra (vs.net eller app folder). Texture sættes lig det billede vi vil importere, og grafikenhedens forstørrelses tilstand(magnification, minification) sættes til lineær.

Næste skridt er at kalde vores Loadtexture metode.

Tilføj følgende kode nederst i OnResetDevice metoden:

```
LoadTexture();
```

Sidste skridt før vi har texture på vores terning, er at fortælle d3d_enheden at den skal bruge en texture. Tilføj følgende kode lige før d3d_enhed.SetStreamSource(0, vertexBuffer, 0) i Render metoden:

```
d3d_enhed.SetTexture(0, texture);
```

Når programmet afvikles nu kan man se at terningen har fået png billedet som texture på alle siderne. Eller det må vi gå ud fra da vi stadig kun ser fronten af terningen ;)

Næstsidste skridt er at gøre terningen gennemsigtig. Alphablending som det kaldes er hele grunden til at jeg lavede dette projekt. Det irriterede mig at jeg ikke kunne finde nogle tutorials i c#, men kun i c++.

Det er faktisk utrolig så lidt der skal til for at gøre vores terning gennemsigtig. 3 linier kode! Tilføj følgende

3 linier kode lige efter d3d_enhed.RenderState.Lighting = false, i OnResetDevice metoden:

```
d3d_enhed.RenderState.AlphaBlendEnable = true;
d3d_enhed.RenderState.SourceBlend = Blend.SourceAlpha;
d3d_enhed.RenderState.DestinationBlend = Blend.InvSourceAlpha;
```

Nu er programmet sådan sett færdig, men vi vil lige tilføje en enkelt ting mere, nemlig at give mulighed for at zoome ind og ud af terningen. Vi starter med at override OnKeyDown metoden så vi kan styre hvad der skal ske ved knaptryk.

Indsæt følgende metode i bunden af Form1 klassen:

```
protected override void OnKeyDown(System.Windows.Forms.KeyEventArgs e)
      {
         switch (e.KeyCode)
         {
            case System.Windows.Forms.Keys.Up:
               zoom += -0.1f;
               break;
            case System.Windows.Forms.Keys.Down:
               zoom += +0.1f;
               break;
            case System.Windows.Forms.Keys.NumPad0:
               {
                  if (d3d_enhed.RenderState.AlphaBlendEnable == true)
                     d3d_enhed.RenderState.AlphaBlendEnable = false;
                  else
                     d3d_enhed.RenderState.AlphaBlendEnable = true;
               }
               break;
         }
      }
```

Koden er meget simpel. Hvis der trykkes på pil op zoomes der ind på firkanten, pil ned zoomer ud. På Keypad 0 kan du toggle alphablending on og off.

Thats it!

Download sourcekoden til denne artikel her:

Download Sourcekode

Jeg lavede artiklen da det jeg ikke kunne finde nogle ordentlige tutorials omkring alpha blending i c# Sourcekoden til denne artikel er sammenstykket af 2 forskellige kode eksempler samt egne modifikationer.

<a
href="http://msdn.microsoft.com/coding4fun/gamedevelopment/">http://msdn.microsoft.com/coding4fun/

gamedevelopment/
http://www.drunkenhyena.com

Kommentar af mysitesolution d. 27. Jun 2007 | 1

Fin :) og til dem der ikke fatter så meget af den men som synes spil og dx3d er interessant, så har MS udgivet programmer til at programmere spil, og som gør det lettere.
http://msdn.microsoft.com/vstudio/express/game/ tilmed til Xbox360 også..

Kommentar af tuxic d. 23. May 2006 | 2

Kommentar af daxiez d. 24. May 2006 | 3

Kommentar af jck_true d. 29. Jan 2006 | 4

Imponerende artikel... også selvom jeg ikke forstår det... hehe

Kommentar af mr-kill d. 25. Jan 2006 | 5

Super artikel...

Kommentar af gentoo2005 d. 01. Feb 2006 | 6

Flot artikel, dog var det ikke smartere at bruge f.eks. loops til d3d_enhed.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 2); ? og genneralt designe det som en n-kantet figur ?

Kommentar af dexo-fan d. 11. Apr 2007 | 7

Den siger:

Programmet har oprettet en undtagelse, som ikke kunne afvikles.

Proces-id=0xdd4 (3540), Tråd-id=0xdb4 (3508).

Klik på OK for at afslutte programmet.
Klik på Annuller for at udføre fejlfinding i programmet.

Og kunne du ikke putte nogle billeder ind i artiklen? :)