Rim van Wersch is senior application developer bij softwareontwikkelaar NetForge

Managed DirectX

GRAFISCH PROGRAMMEREN MET DIRECTX IN C#

Het programmeren van games en het werken met DirectX in het algemeen heeft altijd een beetje op een eiland gezeten. De leercurve is vrij steil, er komt aardig wat wiskunde bij kijken en de C++-syntax heeft menigeen al tot wanhoop gedreven. Met Managed DirectX wordt het een stuk toegankelijker, dus reden te meer om eens te kijken hoe je hier nu gebruik van kunt maken.

DirectX is de verzamelnaam voor een aantal Microsoft API's, zoals Direct3D, DirectSound en DirectInput, die vooral gebruikt worden voor multimediasoftware zoals games, maar ook voor algemenere visualisaties van data. In dit artikel richten we ons op het gebruik van Direct3D, verreweg het grootste onderdeel uit de DirectX-familie dat ontwikkelaars in staat stelt om zowel performante als indrukwekkende grafische applicaties te schrijven. Om applicaties te ontwikkelen in (Managed) DirectX heb je de DirectX SDK nodig. Die is gratis te downloaden vanaf deze pagina: http://msdn.microsoft.com/directx/. De SDK kan zowel in Visual Studio 2003 als 2005 gebruikt worden, al kan het zijn dat er voor Visual Studio 2005 eerst een MDA uitgeschakeld moet worden zoals beschreven aan het einde van dit artikel. Verwar de DirectX SDK niet met de DirectX Runtime. Het is goed mogelijk dat je voor recente games de laatste versie van de Runtime geïnstalleerd hebt, maar dan heb je toch nog de SDK nodig om applicaties te ontwikkelen. Gebruikers van Visual Studio Express hebben naast de DirectX SDK ook de Platform SDK nodig.

Direct3D in Winforms

Voor een Direct3D-applicatie kun je gewoon gebruikmaken van een standaard Windows Application-project. Nadat je het project hebt aangemaakt, moeten hier eerst nog enkele references aan worden toegevoegd voordat je gebruik kunt maken van Direct3D. Deze references zijn:

- Microsoft.DirectX (alleen versie 1.0.x, zie het einde van dit artikel)
- Microsoft.DirectX.Direct3D
- Microsoft.DirectX.Direct3DX

Nadat deze references zijn toegevoegd kunnen we beginnen met onze eerste Direct3D-applicatie. Om Direct3D iets te laten tekenen ('renderen') hebben we een Device-object nodig om van de videokaart gebruik te kunnen maken. De Device-klasse en bijbehorende klassen bevinden zich in de Microsoft.DirectX.Direct3D namespace, dus hiervoor voegen we eerst een using clause toe. Vervolgens voegen we in de constructor van de Form-klasse een eventhandler toe voor het Load-event. Dit is nodig omdat we voor het aanmaken van het Device-object een geldige handle naar een Form nodig hebben en die is gegarandeerd zodra het Load-event wordt aangeroepen. In de eventhandler kunnen we nu de code uit codevoorbeeld 1 invoegen om het Device-object aan te maken. De code is vrij eenvoudig en in de documentatie vind je een uitgebreide beschrijving van wat de verschillende parameters precies inhouden. In een notendop maken we hier een Device-object om de default videokaart (0 als eerste parameter) aan te sturen. Dat we op het Form-object willen renderen, geven we aan door this als de derde parameter te gebruiken en in het PresentParameters-object aan te geven dat we naar een Windowed control renderen. Op de eventhandlers komen we in de volgende paragraaf terug. Nu we een Device-object ter beschikking hebben, kunnen we het gebruiken om de videokaart onze graphics te laten renderen. Dit renderen is een actief proces, omdat we een bewegend beeld willen weergeven. We hebben dus een manier nodig om het device minimaal zo'n dertig keer per seconde de opdracht te geven iets te renderen. Hiervoor wordt een zogenaamde 'render loop' gebruikt, een oneindige lus waarin zowel het renderen als het verwerken van de input van de gebruiker wordt afgehandeld. Om dit te bereiken voegen we een eventhandler toe aan het Application.Idle-event, zodat we onze render loop kunnen starten zodra de applicatie niks te doen heeft. Vervolgens maken we gebruik van een Win32-functie om te blijven renderen totdat onze applicatie weer een Window Message moet verwerken. Zodra dit afgehandeld is, wordt het Application.Idle-event weer aangeroepen en gaan we verder met renderen. Afbeelding 1 geeft een eenvoudig stroomdiagram weer voor dit type render-loop en de code hiervoor vind je terug in codevoorbeeld 2.

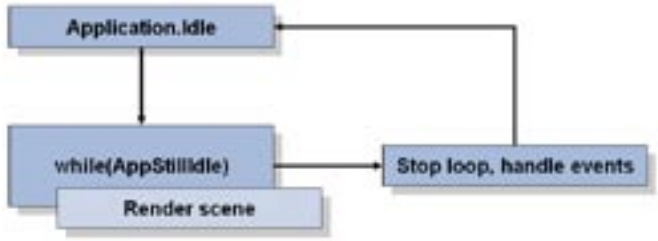
```
void Demo_Load(object sender, EventArgs e)
{
  // Algemene weergave opties
  PresentParameters pp = new PresentParameters();
  pp.SwapEffect = SwapEffect.Discard;
  pp.BackBufferFormat =  Manager.Adapters[0].CurrentDisplayMode.Format;
  pp.AutoDepthStencilFormat = DepthFormat.D24X8;
  pp.EnableAutoDepthStencil = true;
  pp.Windowed = true;
  pp.PresentationInterval = PresentInterval.Immediate;

  // Maak een nieuw Device aan met de ingestelde weergave
  // opties voor dit form
  device = new Device(0, DeviceType.Hardware, this,
  CreateFlags.HardwareVertexProcessing | CreateFlags.PureDevice, pp);

  // Installeer eventhandles om de resources op te ruimen en opnieuw
  // aante maken als we het device verliezen of een geresette
  // terug krijgen.
  device.DeviceReset += new EventHandler(device_DeviceReset);
  device.DeviceLost += new EventHandler(device_DeviceLost);

  // We roepen handmatig de DeviceReset handler aan
  // om de eerste keer onze resource aan te maken.
  device_DeviceReset(device, new EventArgs());
}
```

Codevoorbeeld 1. Demo_Load

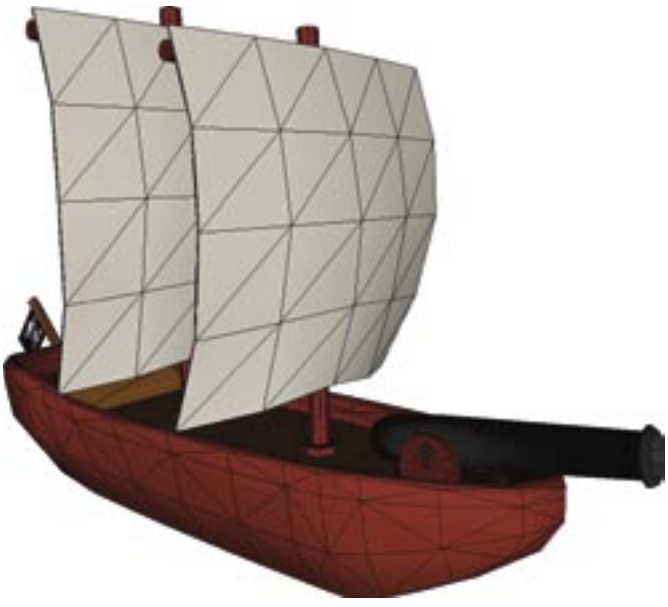


Afbeelding 1. AppStillIdle render-loop

Met de eerste instructie in de loop zorgen we ervoor dat de rendertarget schoongeveegd wordt, zodat deze klaar is om er het volgende frame op te renderen. Met de device.Present()-instructie laten we het device weten dat we klaar zijn met renderen en dat het resultaat op onze control getoond kan worden. Tussen deze twee instructies kunnen we onze rendercode plaatsen, omsloten door de device.BeginScene() en device.EndScene()-instructies.

Vertices, faces, buffers, meshes

Om onze demo wat interessanter te maken, werken we toe naar het weergeven van een 3D-model. Een model in Direct3D bestaat uit een aantal vertices (punten in een 3D-ruimte) waartussen driehoeken (zogenaamde 'faces') worden opgespannen. In afbeelding 2 zie je de randen van de faces in een 3D-model in het zwart weergegeven. De vertices liggen op de knooppunten van deze faces.
Om een model te renderen moeten de vertices naar de videokaart worden gestuurd, zodat deze het 3D-model kan vertalen naar een 2D-afbeelding op het scherm. De efficiëntste manier hiervoor is de vertices in een VertexBuffer-object te plaatsen en deze door de videokaart te laten gebruiken. Hoe dit precies werkt wordt in talloze boeken en tutorials op internet uitgelegd, meestal aan de hand van het renderen van een eenvoudige driehoek. Nu ben je dit artikel waarschijnlijk niet aan het lezen om te leren hoe je een driehoek tekent. Voor deze informatie verwijzen we je door naar onze website http://www.mdxinfo.com, zodat we nu verder kunnen met het renderen van een wat interessanter model. Complexere 3D-modellen worden meestal vormgegeven in een aparte tool zoals 3D Studio Max en vervolgens opgeslagen in een bestand. Daarna kan de Mesh-klasse uit de Direct3DX-assembly gebruikt worden om dit bestand in te laden. Hiervoor vind je de code in codevoorbeeld 3, die wordt aangeroepen via de device_DeviceReset-functie zodra het device is aangemaakt.



Afbeelding 2. Vertices en faces van een 3D-model

```
void Application_Idle(object sender, EventArgs e)
{
  // Zolang we niks te doen hebben, blijven we renderen
  while (AppStillIdle)
  {
    // Reset de ZBuffer (voor diepte) en veeg de target schoon
    device.Clear(ClearFlags.Target | ClearFlags.ZBuffer,
      Color.Black, 1.0f, 0);

    // Laat het device weten dat we met onze scene beginnen
    device.BeginScene();

    // Render ons 3D model
    RenderMesh();

    // Tot zo ver onze scene
    device.EndScene();

    // Laat het resultaat van de render zien op ons form
    device.Present();

    // Hou het aantal frames per seconde bij in de titelbalk
    MeasureFramerate();
  }
}
```

Codevoorbeeld 2. Application_Idle en AppStillIdle

```
private void LoadMesh()
{
  // Hierin plaatst de Mesh.FromFile functie informatie
  // over welke materialen gebruikt worden voor het model
  ExtendedMaterial[] mtrls;

  // Laad het model
  this.mesh = Mesh.FromFile(@"..\..\media\nexus.x", MeshFlags.Managed,
  device, out mtrls);

  // Maak arrays aan voor onze materialen en textures
  this.materials = new Material[mtrls.Length];
  this.textures = new Texture[mtrls.Length];

  // Neem de materialen over en laad eventuele textures indien nodig
  for (int i = 0; i < mtrls.Length; i++)
  {
    this.materials[i] = mtrls[i].Material3D;

    if (mtrls[i].TextureFilename != string.Empty)
    {
      try
      {
        textures[i] = TextureLoader.FromFile(device,
        @"..\..\media\" + mtrls[i].TextureFilename);
      }
      catch { }
    }
  }
}
```

Codevoorbeeld 3. LoadMesh

Een belangrijk detail hier is dat we het Mesh-object aanmaken nadat het Device-object is aangemaakt. Om de data van ons model naar de videokaart te sturen hebben we dus een geldig Device-object nodig. Het kan zijn dat het Device-object ongeldig wordt tijdens het renderen, bijvoorbeeld doordat de grootte van het Form wordt veranderd of doordat de screensaver de videokaart opeist. Om dit te ondervangen gebruiken we het device.DeviceLost-event om onze oude data op de videokaart

op te ruimen en het device.DeviceReset-event om de data opnieuw te laden zodra de videokaart weer beschikbaar is.

Renderen, matrices en transformaties

Wat ons nu nog rest is om onze Mesh daadwerkelijk te tekenen. Een Mesh bestaat uit één of meer subsets die elk hun eigen materiaal met visuele eigenschappen hebben. We tekenen ons model dus per subset, waarvoor we eerst de gewenste textures en de Material-properties op het device instellen die we met de LoadMesh-functie hebben ingeladen, zoals weergegeven in codevoorbeeld 4.

Aan het begin van deze method stellen we de World-matrix in op ons device. Deze matrix transformeert de vertices van het model naar hun uiteindelijke positie in onze virtuele 3D-wereld. Dit concept komt uit de lineaire algebra en we zullen hier verder niet op de wiskundige details in gaan, maar het is belangrijk om te weten dat nagenoeg alle transformaties op vertices worden uitgevoerd met behulp van matrices. Je kunt hier zelf mee experimenteren in de voorbeeldcode door de World-matrix aan te passen. De Matrix-struct bevat een aantal statische methods om Matrix-objecten aan te maken voor translaties (verplaatsingen), rotaties en veranderingen in de schaal van een model. Om meer transformaties op een model toe te passen kun je de matrices met elkaar vermenigvuldigen in de volgorde waarop je ze toe wilt passen, zoals weergegeven in de voorbeeldcode.

Camera's en belichting

Nu we bekend zijn met matrixtransformaties kunnen we het laatste stuk code uit de demo doornemen. Om de 3D-wereld te vertalen naar een 2D-scherm hebben we een virtuele camera nodig, net zoals we een camera nodig hebben om opnames van de echte wereld te maken. Een dergelijke virtuele camera wordt op een locatie in de 3D-wereld geplaats en gericht op een bepaald punt in deze wereld. Net als een echte camera heeft onze virtuele camera ook een bepaalde zichthoek die bepaalt wat de camera vanaf zijn huidige positie kan zien. Omdat onze spelwereld echter niet oneindig is, moeten we dit bereik verder begrenzen door aan te geven vanaf en tot welke afstand van de camera we objecten willen 'opnemen'. Dit stellen we in met behulp van respectievelijk de 'near clipping plane' en de 'far clipping plane'. Het deel van de 3D-wereld dat de camera met deze beperkingen kan zien, heeft de vorm van een piramide waarvan de top is afgesneden ter hoogte van de near clipping plane en waarvan de far clipping plane de basis vormt. Dit gebied wordt het 'view frustum' genoemd. In afbeelding 3 zie je een voorbeeld van een dergelijk view frustum.
Objecten die buiten dit gebied vallen worden niet weergegeven op het scherm, maar ze worden wel gewoon gerenderd. Om dit te voorkomen en zo de prestaties van je 3D-applicatie te verbeteren kun je gebruikmaken van de zogenaamde 'view frustum culling'-techniek. Op internet is hierover veel informatie te vinden, dus beperken we ons in deze introductie tot de code die nodig is om onze camera in te stellen. Deze code vind je in het demoproject terug in de DeviceReset-eventhandler en in codevoorbeeld 5 in dit artikel.

```
private void RenderMesh()
{
   // Transformatie van ons model, eerst verkleinen we het tot
   // de deel van de oorspronkelijke groote, daarna roteren we
   // het om de y-as en daarna nog eens om de x-as
   double time = timer.ElapsedMilliseconds;
   device.Transform.World = Matrix.Scaling(0.85f, 0.85f, 0.85f)
      * Matrix.RotationY(0.5f * (float)Math.Sin(time / 2000f))
      * Matrix.RotationX(0.25f * (float)Math.Sin(time / 1000f));

   // Render alle delen van ons model met de
   // bijbehorende materialen en textures
   for (int i = 0; i < materials.Length; i++)
   {
      device.Material = materials[i];
      device.SetTexture(0, textures[i]);

      mesh.DrawSubset(i);
   }
}
```

Codevoorbeeld 4. RenderMesh

```
// Stel de opties in voor onze camera
device.Transform.Projection = Matrix.PerspectiveFovLH(
   (float)Math.PI / 4,       // helft van de zichthoek
   (float)Width / (float)Height, // aspect ratio
   10,  // near clipping plane
   600); // far clipping plane

device.Transform.View = Matrix.LookAtLH(
   new Vector3(0, 0, -300), // positie van de camera
   new Vector3(0, -25, 0),    // punt waar de camera op gericht is
   new Vector3(0, 1, 0));  // 'omhoog' vanuit de camera
```

Codevoorbeeld 5. Het instellen van de camera

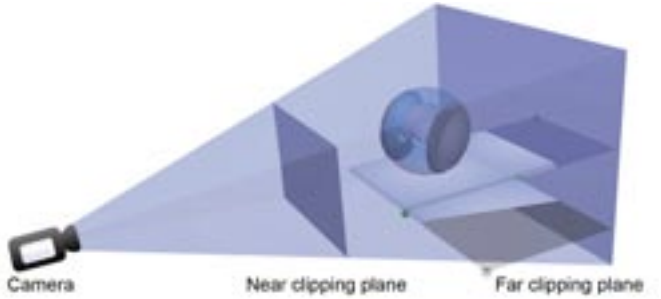
Zoals je ziet gebruiken we voor het instellen van de camera ook twee matrices. De eerste is de zogenaamde Projection-matrix. Deze definieert de vorm van ons view frustum en emuleert het effect van perspectief voor onze 'opname', door ervoor te zorgen dat objecten dichtbij de camera groter worden weergegeven dan objecten verder weg. De tweede matrix, de View-matrix, definieert waar de camera zich bevindt in onze 3D-wereld en waar hij op gericht is. Met het instellen van de camera hebben we alle stappen om een eenvoudige Managed DirectX-applicatie te programmeren doorgenomen. Wat nog over blijft is het instellen van de belichting. Voor deze demo gebruiken we een eenvoudige belichting met een zogenaamd 'directional' licht. Hiervoor hoeven we alleen de kleur en de richting in te stellen, dus de code hiervoor spreekt redelijk voor zich.

Hoe nu verder?

Als je verder wilt gaan met Managed DirectX leer je het meeste door er iets mee te doen. Je kunt het beste een spel of visualisatieproject uitzoeken en dit programmeren in Managed DirectX. Dit gaat misschien met vallen en opstaan, maar alleen zo leer je de valkuilen van 3D-programmeren echt kennen. Voor meer informatie over Managed DirectX kun je terecht op de websites die aan het einde van dit artikel staan genoemd. Spelontwikkeling is een populair onderwerp onder programmeurs, dus er zijn veel mensen die hun kennis erover graag delen. De *For Beginners, DirectX*- en *.NET*-fora van www.gamedev.net zijn een aanrader als je vast zit met een vraag. De onlangs opgerichte website www.mdxforum.com is een ander initiatief waar je met vragen terecht kunt.

De verschillende versies van Managed DirectX

Er bestaan op het moment twee versies van Managed DirectX, MDX 1 en MDX 2. De versienummers komen overeen met de versie van het .NET-platform waarvoor ze oorspronkelijk geschreven



Afbeelding 3. Het view frustum

zijn, maar met de recente ontwikkelingen rond de API leiden ze momenteel vooral tot verwarring. MDX 1 is nog steeds de officieel aangeraden versie van Managed DirectX, ook voor gebruik in combinatie met het .NET Framework 2.0. De MDX 2-versie wordt nog als bètaversie meegeleverd met de DirectX SDK, maar bij de presentatie van het XNA Framework in maart 2006 heeft Microsoft aangekondigd dat de MDX 2-versie in zijn huidige vorm niet gereleased zal worden. MDX 2 zal overgaan in het XNA Framework, een aangepaste versie van de DirectX runtime, die zowel op het Windows-platform als op de Xbox 360 zal kunnen draaien.

Managed Debug Assistants in Visual Studio 2005

Om applicaties die gebruikmaken van MDX 1 in Visual Studio 2005 te kunnen debuggen, dient mogelijk eerst de 'Loader Lock' Managed Debug Assistant uitgeschakeld te worden. Dit kun je doen door in Visual Studio 'CTRL-D' en vervolgens 'E' in te typen en in het scherm dat dan verschijnt 'Loader Lock' onder Managed Debug Assistants uit te schakelen. Meer informatie hierover is beschikbaar op deze pagina: http://www.thezbuffer.com/articles/304.aspx.

Goed nieuws voor ontwikkelaars van games kwam tijdens GameFest 2006. In de keynote werd XNA Game Studio Express aangekondigd. Deze gratis developmenttool stelt iedereen in staat om games te bouwen. Ook voor de Xbox 360! Een abonnement van $99 per jaar op de 'creators club' is wel nodig om toegang te build en test tools voor de Xbox 360. http://www.microsoft.com/presspass/press/2006/aug06/08-13XNAGameStudioPR.mspx

Rim van Wersch werkt als senior application developer bij de Limburgse softwareontwikkelaar NetForge (http://www.netforge.nl) en is medeoprichter van http://www.mdxinfo.com, een website met informatie over het werken met Managed DirectX. Daarnaast is Rim actief binnen verschillende communities over spelontwikkeling en grafisch programmeren, zoals GameDev en GraphicsDev. Voor vragen over dit artikel of Managed DirectX is Rim te bereiken via rim@mdxinfo.com.

Aan te raden boeken

Managed DirectX 9 Kickstart door Tom Miller, ISBN 0-672-32596-9

Beginning 3D Game Programming door Tom Miller, ISBN 0-672-32661-2

Referenties

http://www.mdxinfo.com
http://www.gamedev.net
http://www.thezbuffer.com
http://www.mdxforum.com
http://www.graphicsdev.net
http://msdn.microsoft.com/directx
http://msdn.microsoft.com/coding4fun/
http://www.microsoft.com/xna/
http://go.microsoft.com/fwlink/?linkid=62092
http://www.c-unit.com/tutorials/mdirectx/