Open Gl - Delphi – Kurs Teil 2: Wie Initialisiert man Open Gl in Delphi

Einleitung

Ach Leute, ist es schon wieder soweit?
Schon wieder 2 Wochen rum? Na ja... dann werde ich wohl müssen...
In diesem Kursteil wollen wir also Open Gl in Delphi initialisieren, d.h., dass wir Open Gl klar machen wollen, dass es in unserem Fenster etwas zeichnen kann... mehr nicht!
Ich weiß, das dieses für extrem wissbegierige sehr wenig ist, aber ich möchte auch ausreichend genug auf Kleinigkeiten eingehen können, damit nicht wieder so was Unvollständiges wie in meinem Kurztut dabei rauskommt.
Leider muss ich auch sagen, das dieser Teil des Kurses der ziemlich Trockenste und Langweiligste wird. An sich ist die Initialisierung eh jedes mal das selbe... für alle andren Teile des Kurses würde es auch reichen, wenn ihr euch einfach das Sample runterladen würdet und es als Grundgerüst für alles andre verwenden würdet. Wer wirklich lust hat und sich nicht entmutigen lässt, der darf ja ruhig auch lesen (aber hinterher nicht entnervt aufgeben!!!)

Wie initialisiert man Open Gl

Dann wollen wir also...
Zunächst haben wir nur ein sehr leeres Projekt vor uns, ganz ohne eigenen Code...

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

end.
```

Soweit, so gut... das sollte nun zunächst jeder vor sich sehen.
Der erste Schritt besteht nun darin, an den Uses – Teil hinter „Dialogs“ noch die Unit „Opengl“ einzutragen, damit Delphi die nötigen Dinge erst mal parat hat.
Nun benötigen wir ein paar Variablen, welche wir in den Privat – Bereich schreiben. Es handelt sich bei den ersten Zwei um Variablen, welche bei der Erstellung des OGL Kontextes benötig werden. Die dritte ist die Variable für die Farbpalette. Zudem müssen wir schon mal die Prozedur mit Namen „SetupPixelFormat“ hier einragen. Sie wird später gebraucht, da in ihr die meisten Open Gl – Einstellungen vorgenommen werden. Das Ganze sollte nun so aussehen (Veränderungen sind rot markiert):

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Opengl;

type
  TForm1 = class(TForm)
  private
    { Private-Deklarationen }
   myDC : HDC;
   myRC : HGLRC;
   myPalette : HPALETTE;
   procedure SetupPixelFormat;
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.SetupPixelFormat;
begin

end;
end.
```

So Leute: weiter im Programm! Nun benötigen wir noch ein Paar kleine Dinge:

1) wir legen eine kleine Prozedur Namens „Render“ an (die kann auch anders heißen, aber ich möchte das ganze Normieren, d.h. ich möchte, dass ich nicht immer sagen muss „ruft nun eure Rendering – Methode auf“... Ihr wisst schon, was ich meine

2) Wir brauchen die 2 Prozeduren „Form.create“ und „Form.destroy“. In die erste soll später die eigentliche Initialisierung und die andere Brauchen wir, um den Rendering – Kontext (das Ding, was Open Gl sagt, wo es zeichnen soll) wieder freizugeben.
3) Und zuletzt: die OnRezise – Prozedur, damit wir auch ne Anständige Sichtweise bekommen (haben wir dich nicht schon???)

So... wenn wir das haben, dann könnten wir uns ja an die ganze Aktion wagen. Ich schreibe euch hier nun mal den ganzen Code rein (mit „Oncreate“,usw) und erkläre euch dann, was das alles zu Bedeuten hat:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Opengl;

type
  TForm1 = class(TForm)
  private
    { Private-Deklarationen }
   myDC : HDC;
   myRC : HGLRC;
   myPalette : HPALETTE;
   procedure SetupPixelFormat;
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.SetupPixelFormat;
begin

end;

procedure Render;
begin

end;

procedure TForm1.FormCreate(Sender: TObject);
begin
form1.myDC:= GetDC(Handle);
SetupPixelFormat;
```

```
myRC:= wglCreateContext(myDC);
wglMakeCurrent(myDC, myRC);
glEnable(GL_DEPTH_TEST);
glLoadIdentity;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
wglmakecurrent(0,0);
wgldeletecontext(mydc);
releasedc(handle,mydc);
end;

procedure TForm1.FormResize(Sender: TObject);
begin
glViewport(0, 0, Width, Height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
 gluPerspective(45.0, Width/Height, 1.0, 100.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
end;

end.
```

So und nun die Erklärung:

Ich denke, wir fangen mal mit „OnCreate“ an:

Also: „DC“ bedeutet soviel wie Device Context... womit dann auch klar wäre, das in der ersten Zeile von „oncreate“ zunächt erst mal der Device – Context unseres Hautfensters ausgelesen wird (wer nicht weiß, was das bedeutet: es ist nicht wirklich notwendig das zu wissen... wer noch nicht ewig mit PCs arbeitet muss es auch nicht wissen... die, die schon länger damit arbeiten wissen, was gemeint ist – ich kann es nämlich nicht wirklich gut erklären)
In der nächsten Zeile wird dann unsere Prozedur „SetupPixelFormat“ auf... was die bedeutet, wisst ihr ja bereits: Dort werden wir einige kleine Settings vornehmen.
So... weiter im Kontext (im wahrsten sinne des Wortes):
Die nächsten beiden Zeilen bewirken zunächst, das aus dem Device Context unseres Fensters ein Open Gl Rendering Context (rc) erzeugt wird und daraufhin aktiviert wird.
Die nächste Zeile dürfte dagegen an sich klar sein: Hier aktivieren wir via „glenable“ den Open Gl Tiefentest. Dies bewirkt, das Open Gl immer überprüft, wo in der Tiefe das zu Zeichnende Objekt liegt... ist es zu weit weg, oder von etwas anderem verdeckt, wird es gar nicht erst berechnet. Insbesondere bei großen Szenen ist diese Tiefenüberprüfung also mehr oder weniger maßgeblich für die Performance der Szene... Open Gl erzwingt diesen Modus aber nicht (wäre auch schlecht... es gibt auch Situationen, wo der Tiefentest deaktiviert werden muss... z.B. wenn man mit viel Transparenz arbeitet)
Der letzte Befehl ist nicht ganz einfach zu erklären. Dazu muss man über die >Funktionsweise von Open Gl näher bescheid wissen. Und zwar: alle Objekte in Open Gl würden, wenn sie denken könnten und wüssten, wie sie entstehen praktisch wahnsinnig werden: Alle Objekte

werden Praktisch über die sogenannte „Welt-Matrix“ (die dem aus dem Film Matrix nicht unähnlich ist) zunächst erstellt, dann gedreht, skaliert und natürlich auch verschoben. Dazu gibt es in Open Gl praktisch einen imaginären Punkt, welcher sagt, an welcher Stelle nun gezeichnet werden soll (besser: um welchen Punkt rum zu gezeichnet werden soll) „glloadidentity“ lädt nun den Ursprung (0,0,0) als diesen Punkt, oder auch gesagt: Es setz die ganze zukünftige Szene zurück. Dies ist zwingend notwendig, denn wir wollen unsere Szene ja nicht irgendwo ins Nirwana zeichnen, sondern wir wollen wissen, wo die Szene ist, damit wir uns diese ansehen können.
Im übrigen: es gibt nicht nur diese eine World – Matrix, sondern es gibt noch viele mehr: eine für Texturen, eine für die Projektionseigenschaften, usw.

So... nachdem wir diesen Textbrocken von onCreate hinter uns gelassen haben, machen wir mal was kleineres: wir beschrieben onDestroy:

Denn an sich gibt es da nichts zu erklären: alle drei Prozeduren dienen nur dazu, den Rendering – Kontext wieder frei zu geben, um Open Gl klar zu machen, dass es nun nicht mehr auf unserem Formular zeichnen soll.

Hmm... nun fehlt natürlich nur noch onresize...
Wie ihr vielleicht wisst, wird onresize aufgerufen, wenn die Größe des Formular geändert wird. An diesem Punkt muss man Open Gl quasi neu konfigurieren, damit nicht alle Objekte unserer Szene total verzerrt gezeichnet werden (ihr könnt ja mal Testweise diesen Code weglassen... ihr dürftet aber kaum glücklich damit werden)
An sich sind die Prozeduren hier sehr einfach zu verstehen:
Zunächst wird Open Gl klar gemacht, wie die neuen Maße unseres Fensters aussehen... besser: Wir legen den rechteckigen Rahmen fest, in dem gezeichnet werden soll. Dann schalten wir in die Projektionsmatrix damit wir die Darstellungsweise unserer Szene direkt verändern können. Nun setzen wir per glloadidenity diese Matrix komplett zurück (man erinnert sich ;-)) Als nächstes setzen wir die neuen Settings für die Darstellung vor... das übernimmt für uns „gluperspective“, welches 4 Parameter erwartet:
Der erste (bei uns 45) steht für die Gradzahl des Blickwinkels. Hier werden meistens werte zwischen 30 und 45 gewählt, die dem natürlichen Blickwinkel entsprechen. Der nächste steht für das Breiten – Längen – Verhältnis, welches dadurch angegeben wird, dass man die Breite durch die Höhe teilt. Die letzen zwei Parameter stehen für den nächsten und den entferntesten sichtbaren Punkt für unsere Kamera. Das heißt: wenn wir auf dem Ursprung steht, und in die z-Richtung sieht, dann ist der nächste Punkt, den man sehen kann einer mit der Z-Koordinate 1 und der entfernteste (der Horizont) wir durch Objekte mit der Z-Koordinate 100 gebildet.
Die letzen beiden Funktionen schalten an sich nur noch in den Normalmodus zurück, damit auch wieder normal gezeichnet werden kann.

Im übrigen: In dieser Initialisierung verwenden wir die perspektivische Darstellungsweise verwendet, welche die Realitätsnahe ist. Als Alternative gibt es noch die Orthographische, welche hauptsächlich in CAD – Programmen verwendet wird. Allerdings werden hier alle Objekte mit ihrer Originalgröße dargestellt... unabhängig vor ihrer Entfernung zum Betrachter.

So... das war nun aber derbe viel. Ich hoffe, dass alle anderen Teile nicht ganz so theorieastig werden. Aber eine Sache fehlt noch: „SetupPixelFormat“ ist noch sehr mager mit Code bestückt. Wie bereits gesagt: in dieser Prozedur nehmen wir die Einstellungen für Open Gl vor. Ich poste hier einfach mal eine gültige Variante. Ich werde dieses nun nicht ganz so

genau erklären, aber ich habe das Wichtigste mit Kommentaren versehen, damit ihr bei bedarf Änderungen problemlos vornehmen könnt (es wäre etwas arg viel, ins Detail einzugehen)

```
procedure TForm1.SetupPixelFormat;
var    hHeap: THandle;
  nColors, i: Integer; //Anzahl der Farben
  lpPalette : PLogPalette; //Die Farbpalette
  byRedMask, byGreenMask, byBlueMask: Byte; //Blau, Grün und rot
  nPixelFormat: Integer;
  pfd: TPixelFormatDescriptor; //Dies ist der „Descriptor“... in ihm werden die Infos
                                        // zwischengespeiichert
begin
  FillChar(pfd, SizeOf(pfd), 0);
  with pfd do begin                      //wir woollen das format bearbeiten
    nSize     := sizeof(pfd);            // Länge der pfd-Struktur
    nVersion  := 1;                      // Version
    dwFlags   := PFD_DRAW_TO_WINDOW or PFD_SUPPORT_OPENGL or
                 PFD_DOUBLEBUFFER;         // Flags
    iPixelType:= PFD_TYPE_RGBA;            // RGBA Pixel Type
    cColorBits:= 24;                       // 24-bit color (Anzahl der Farben)
    cDepthBits:= 32;                       // 32-bit depth buffer
    iLayerType:= PFD_MAIN_PLANE;           // Layer Type
  end;
  nPixelFormat:= ChoosePixelFormat(myDC, @pfd); //Das Pixel-Format
  SetPixelFormat(myDC, nPixelFormat, @pfd);         // wird übertragen
// Farbpalettenoptimierung wenn erforderlich
  DescribePixelFormat(myDC, nPixelFormat,  //alles ab hier ist zwecks einstellungen nicht
//mehr wichtig, sondern schlicht und einfach nur noch notwendig... ;-)
                      sizeof(TPixelFormatDescriptor),pfd);
  if ((pfd.dwFlags and PFD_NEED_PALETTE) <> 0) then begin
    nColors  := 1 shl pfd.cColorBits;
    hHeap    := GetProcessHeap;
    lpPalette:= HeapAlloc
      (hHeap,0,sizeof(TLogPalette)+(nColors*sizeof(TPaletteEntry)));
    lpPalette^.palVersion := $300;
    lpPalette^.palNumEntries := nColors;
    byRedMask  := (1 shl pfd.cRedBits) - 1;
    byGreenMask:= (1 shl pfd.cGreenBits) - 1;
    byBlueMask := (1 shl pfd.cBlueBits) - 1;
    for i := 0 to nColors - 1 do begin
      lpPalette^.palPalEntry[i].peRed  :=
        (((i shr pfd.cRedShift)  and byRedMask)  *255)DIV byRedMask;
      lpPalette^.palPalEntry[i].peGreen:=
        (((i shr pfd.cGreenShift)and byGreenMask)*255)DIV byGreenMask;
      lpPalette^.palPalEntry[i].peBlue :=
        (((i shr pfd.cBlueShift) and byBlueMask) *255)DIV byBlueMask;
      lpPalette^.palPalEntry[i].peFlags:= 0;
    end;
    myPalette:= CreatePalette(lpPalette^);
    HeapFree(hHeap, 0, lpPalette);
    if (myPalette <> 0) then begin
```

```
    SelectPalette(myDC, myPalette, False);
    RealizePalette(myDC);
  end;
end;
```

Nun ist es an sich fast vollbracht. Uns fehlen nur noch 2 Dinge: Einen Platz, wo wir ein Objekt rendern können und ein Objekt.
Um ersteres werden wir uns nun noch kümmer:
Legt (direkt unter SetupPixelFormat)ne Prozedur „Render“ an, welche nur drei Befehle beinhalten sollte:

```
Prozedure render;
begin
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
glLoadIdentity;

SwapBuffers(form1.myDC);
end;
```

Der erste Befehl bewirkt, dass unsere ganze Szene, welcher sich noch im Puffer befindet, gelöscht wird, damit es nicht zu Überschneidungen und Verschmierungen kommt. Wer mal sehen will, was es heißt, diesen Befehl nicht auszuführen, der sollte mal ne Runde CS zocken und dann als Spectator durch eine Wand laufen... alles verwischt und verschmiert total.
Der zweite Befehl sollte schon bekannt sein: er setzt die World – Matrix wieder zurück, damit die Szene auch wieder am richtigen Ort mit richtiger Rotation und Skalierungen gezeichnet wird. Der letzte Befehl bewirkt nun noch, das Open Gl die Szene auch wirklich darstellt.

Nun müssen wir nur noch die Render – Prozedur irgendwie aufrufen. Unter der VCL bietet sich dafür der Timer an. Wir erstellen nun also einen normalen Timer, dessen Intervall wir auf 1 einstellen und lasen diesen unsere Render – Prozedur aufrufen.

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
render;
end;
```

Wenn ihr die Delphi X – Komponenten habt, solltet ihr den DX – Timer verwenden. Dies hat folgende Gründe:
Wenn ihr den normalen Timer auf eins stellt, schafft er dennoch nichts schnelleres, als etwa 50. Das bedeutet, dass wir maximal 20 fps erreichen können. Der DX – Timer hingegen ist viel korrekter. Mit ihm erreicht man normalerweise ein Maximum von ca. 80 – 160 FPS. Diese schon recht dollen FPS – Zahlen wird man zwar insbesondere mit großen szenen fast nie erreichen, aber man schafft fast immer mehr, als mit dem normalen Timer.

So... nun haben wir Open Gl schon beinahe komplett lauffähig. Nun fehlt nur noch unseres Objekt... aber das behandeln wir im nächsten Kapitel. Ich hoffe, dieses Kapitel hat euch nicht entmutigt, denn die Programmierung von 3d – Szenen mit Open Gl ist sehr einfach... bloß die Initialisierung ist eine etwas umfangreiche und harkeilige Sache.
Wer in diesem Kapitel nicht mitgekommen ist, den kann ich beruhigen: das nächste wird viel einfacher. Wenn ihr wollt, könnt ihr euch auch einfach die ganze Initialisierung sauen (unten ist ein Download Link... nur in der Online Version)
Ich wünsche viel Spaß und bis in zwei Wochen:

Mr_T