Open Gl - Delphi – Kurs Teil 12: Vermischtes

Einleitung

Hi Leute!
Willkommen zurück zu meinem Delphi – Open Gl – Kurs! Dieser Teil wird ein wenig anders als alle bisherigen, da es sich hier mal nicht um ein einziges großes Thema, sondern um viele Kleine handelt. Eigentlich wollte ich grundsätzlich so was nicht machen, da dadurch der Eindruck entstehen könnte, dass bald alles vorbei sei, aber in diesem Falle ist das Gegenteil der Fall. Die kommenden Themen vordern leider an manchen Stellen kleine Techniken, welche man nicht so nebenher erklären kann und deshalb bekommt ihr hier eine sinnvolle Vorbereitung auf die Dinge, die da kommen. (Und das werden viele sein *gg*)
Aber keine Sorge, dass ihr an Langweile Sterben werdet, denn auch diesmal kommt viel auf euch zu: Echtes Multitexturing, eine neue Open Gl – Unit, Displaylisten, Schriftdarstellung uvm. Warten auf euch, also los!

Open Gl 1.5

Wie ihr alle vielleicht wisst ist das, womit wir momentan noch programmieren (Open Gl 1.0) nun wirklich nicht mehr ganz der neuste Stand der Technik. Aus diesem Grund möchte ich mit diesem Kursteil unsere Open Gl – Unit wechseln und auf die OpenGl15.pas der DGL (Delphi Open Glide Community: www.delphigl.com) umsteigen, welche den neuen Open Gl 1.5 – Standart (welche keine 2 Monate alt ist und welcher noch gar nicht richtig durch alle Treiber umgesetzt ist) umsteigen, womit wir wieder ganz vorne am Ball sind.
Saugen könnt ihr euch diese Unit unter www.delphigl.com à Files à Bibliotheken à Open Gl 1.5 Header... solltet ihr dazu keine Lust haben, saugt euch einfach mal das 1. Sample zu diesem Kursteil (wird gleich besprochen)... da ist eine Unit mit drinne...
Um diese Unit verwenden zu können, muss man am Anfang eures Form.create-Ereignisses lediglich einmal „loadopengl“ und am Ende des ondestroy-Ereignisses einmal „unloadopengl“.
Des weiteren ändert sich eine Kleinigkeit: Momentan verwenden wir immer noch solche Dinge wie „gltranslate“ oder „glrotate“... dieses müssen wir nun aufgeben und stattdessen „gltranslatef“ bzw. „glrotatef“ verwenden... außer diesen Kleinigkeiten ändert sich aber nichts... (alles wird noch genau so verwendet wie zuvor...)

Cu, Timer

Ja ja, lange haben wir den Timer (oder den DX-Timer) als Taktgeber benutzt, aber ist euch dabei schon mal aufgefallen, dass wir selten mehr als 20 FPS erreicht haben? (Ok, mit dem DX-Timer war es etwas mehr)
Das soll sich nun ändern: (wozu habe ich euch sonst das Timebased-Movement in Teil 8 beigebracht?)
Dieses wollen wir über das „onidle“-Event machen. Dieses ist ein Event, welches in allen Delphi-Applikationen vorhanden ist, aber nicht öffentlich. (heißt im Objektsinspector)

Aber was bitte bewirkt dieses Event? Tja, das ist ganz simpel: es wird immer dann ausgeführt, wenn aus irgendeinem Grunde die Anwendung lehr läuft, meint, wenn Rechenpower verfügbar ist... Und jedes bisschen Leistung können wir zum Rendern verwenden ... ist doch logo ;-)
Wir wollen also gerne dieses „versteckte“ Event nutzen... wie stellen wir das aber nun an?
Nun ja: zu nächst schmeißen wir erstmal unseren alten Timer raus. (Sicherlich keine schwere Aufgabe *ggg*)
Nun müssen wir unserer Form1 eine nette „Public“-Prozedur verpassen:

```
public
   { Public-Deklarationen }
  procedure myIdle(Sender: TObject; var done: boolean);
```

Mom zeigt diese Prozedur aber noch ins Lehre, deshalb müssen wir die Eigentliche Prozedur auch noch hinzufügen. Dies machen wir am Ende des Quelltextes:

```
procedure tform1.MyIdle(Sender: TObject; var Done: Boolean);
begin
render;
done := false;
end;
```

Soweit, so gut... (Sorry, wenn meine Erklärungen etwas knapp ausfallen, aber des Kursteil beinhält ne ganze Menge)
Wenn wir nun unser Progie starten wollen sehen wir aber immer noch nichts! Das liegt ganz simpel daran, dass unsere Prozedur noch nicht mit dem eigentlichen Event verknüpft ist... um sie zu verknüpfen müssen wir nun noch am Ende der Oncreate-Prozedur diesen Quelltext aufrufen:

```
Application.OnIdle:= MyIdle;
```

Fertig! (Wem das alles zu schnell ging: einfach mal ins Sample gucken...)

Displaylisten

Oftmals wird euch schon aufgefallen sein, das man bestimmte Dinge (Open Gl – Objekte) immer wieder im Programm braucht bzw. statische Szenen, die sich nie verändern. (Ok, sie werden verschoben, aber in sich bleiben sie gleich)
Für solche Fälle bietet Open Gl etwas ganz besonderes: Die Displaylisten. Displaylisten sind an sich nichts anderes als Objekte, die einmal vorberechnet werden und denn nur noch an geeigneter Position dargestellt werden. Es liegt auf der Hand, dass auf einigen Grafikkarten dadurch ein netter Geschwindigkeitszuwachs erreicht werden kann... Fakt ist, das die Darstellung aber auf keinem Falle langsamer wird. (Allerdings sollte man bei all den Vorteilen auch bedenken, dass natürlich diese Objekte nicht mehr verändert werden können.)
Na ja, genug der einführenden Worte, lasst uns anfangen zu coden *ggg*
Um also eine neue Displayliste anzulegen benötigen wir zunächst eine globale Variable vom Typ gluint (bzw. bei unsere OpenGl15.pas Tglint).

```
Var
List : tglint;
...
```

Müssen wir den Code, welchen wir in eine Displaylist rendern wollen in eine extra Prozedur schreiben, halt so etwa:

```
procedure create_displaylist;
begin

Glbegin(gl_triangles);
glcolor3f(1,0,0);
glvertex3f(-1,0,0);
glcolor3f(0,1,0);
glvertex3f(1,0,0);
glcolor3f(0,0,1);
glvertex3f(0,2,0);
glend;

end;
```

So, nun wird ’s interessant...
Nun müssen wir also unsere eigentliche Displayliste erst mal „kompilieren“ und das geht so:

```
procedure create_displaylist;
begin
List := glGenLists(1);
glNewList(List,GL_COMPILE);

Glbegin(gl_triangles);
glcolor3f(1,0,0);
glvertex3f(-1,0,0);
glcolor3f(0,1,0);
glvertex3f(1,0,0);
glcolor3f(0,0,1);
glvertex3f(0,2,0);
glend;

glendlist;
end;
```

So, mit dem bischen Code haben wir auch schon eine Displayliste erzeugt...
Ach ja? Nein, bislang passiert gar nichts *ggg* Und wieso nicht? Weil ich Dummerchen euch noch nicht gesagt habe, wo ihr den Code aufrufen sollt... Ich persönlich würde hier mal wieder das beliebte OnCreate-Event vorschlagen, und zwar direkt vor dem „Application.OnIdle:= MyIdle;“ (siehe oben). Damit wäre denn eigentlich unsere Displayliste erzeugt... nun müssen wir nur noch das Ding auf den Schirm bringen ... An sich auch ganz simpel:

```
procedure render;
begin
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
glLoadIdentity;
gltranslatef(0,0,-5);
glCallList(List);
SwapBuffers(form1.myDC);                    //scene ausgeben
```

```
end;
```

Fertig!
Zugegeben, ein Dreieck ist nichts und der Geschwindigkeitsvorteil nicht vorhanden, aber so konnte ich zumindest leicht zeigen, wie es geht… Im Beispielprogramm zu diesem Kapitel findet ihr eine einfache Anwendung, welche über Displaylisten n ganzen Haufen Dreiecke darstellt… auch hier macht es Geschwindigkeitstechnisch nicht viel, aber naja :-)

Culling

Nachdem wir nun also die Displaylisten hinter uns gebracht haben, wollen wir uns einer anderen Technik widmen, welche ebenfalls die Darstellung unserer Open Gl Szenen beschleunigen kann. Ich sage extra „kann“, weil wie leider auch bei den Displaylisten eine Anwendung dabei erheblich schneller werden kann (bis zu 50%) aber es auch passieren kann, dass die Anwendung nicht schneller läuft als zuvor. (Abhängig von der Grafikkarte und dem Treiber)
Das Culling ist eine simple Technik welche sich folgenden Umstand zunutze mache: Angenommen wir zeichnen eine einfache Wand mit Open Gl (könnte auch Direkt 3D sein ... ist in diesem Falle wurscht), denn werden doch immer zwei Seiten gezeichnet: eine für den User sichtbare und eine Rückseite, welche - wenn man glück hat - im Spiel bzw. der Anwendung nie gezeigt wird, weil meistens ja eine Wand eine gewisse Dicke aufweist und diese Deshalb aus mehreren flachen Quadraten besteht.
Nun gibt es einige Grafikkarten, welche beide Seiten einzeln berechnen (es gibt auch welche, die Berechnen die eine und spiegeln denn die eine Seite), was natürlich n recht netten Rechenaufwand ergibt, den man sich sparen könnte, wenn die Grafikkarte doch nur wüsste, welche Seite wirklich sichtbar ist bzw. sein sollte.
Und hier setzt das Culling an.
Sicherlich ist euch schon mal aufgefallen, dass es in Open Gl zwei verschiedene Methoden gibt, um Vierecke, welche von den Punkten her gleich sind, zu zeichnen. So kann man die Punkte gegen und mit dem Urzeigersinn (aus Sicht des Betrachters) angeben und so darstellen. Mithilfe des Cullings wird nun also OpenGl gesagt, dass nur eine Seite gezeichnet werden soll bzw. man kann sagen welche. (Im oder gegen den Urzeigersinn)
Normal ist es, dass die gegen den Urzeigersinn gezeichnet werden, keine Ahnung, warum. (Gehört zum guten Ton *g*)
Na gut, denn wollen wir mal Anfangen.
Hier seht ihr noch mal das Dreieck aus dem letzen Kapitel

```
Glbegin(gl_triangles);
glcolor3f(1,0,0);
glvertex3f(-1,0,0);
glcolor3f(0,1,0);
glvertex3f(1,0,0);
glcolor3f(0,0,1);
glvertex3f(0,2,0);
glend;
```

Wie ihr alle sehen könnt (bzw. erraten *g*), wird es, wenn es uns direkt anguckt gegen den Urzeigersinn gezeichnet...

Mein Ziel ist nun folgendes: Von diesem kleinen Dreieck wollen wir nun nur die eine Seite painten und es dabei rotieren lassen... das Resultat ist klar „sichtbar, weg, sichtbar, weg, sichtbar...“
Also ans Werk. Zunächst einmal müssen wir in unserer beliebten Oncreate-Prozedure das culling aktivieren:

```
...
glEnable(GL_CULL_FACE);
…
```

Der nächste Schritt ist nun, dass wir Open Gl beibringen müssen, welche Seite es als „Vorne“ anzusehen hat (sonst wäre Open Gl auch ziemlich aufgeschmissen):

```
...
glEnable(GL_CULL_FACE);
glFrontFace(GL_CCW);
…
```

Damit hätten wir Open GL gesagt, dass es „gegen den Urzeigersinn“ (CCW = Counter Clock Wise) und nicht mit dem Urzeigersinn (CW = Clock Wise) als Vorderseite annehmen soll.
Nun sind wir mit dem Culling schon fast durch... das einzige, was noch fehlt ist, dass wir Open Gl noch sagen müssen, welche Seite (Vorder- oder Hinterseite) vernachlässigt werden soll... dieses Stellt man so ein „glCullface(GL_FRONT);“ bzw. „glCullface(GL_Back);“...
In unserem Falle wollen wir also die Rückseite „loswerden“, also schreiben wir:

```
glCullface(GL_Back);
Glbegin(gl_triangles);
glcolor3f(1,0,0);
glvertex3f(-1,0,0);
glcolor3f(0,1,0);
glvertex3f(1,0,0);
glcolor3f(0,0,1);
glvertex3f(0,2,0);
glend;
```

Fertig!

Die Macht der Kamera

Leider musste ich schon mehrfach hören, dass einige Leute deutliche Probleme mit der Kameraführung haben, was sicherlich daran liegt, dass die meisten Schwierigkeiten haben umzudenken. (Das nicht sie, sondern alles andere sich bewegt)
Für alle diese will ich hier einen glu-Hilfsbefehl noch kurz vorstellen. Ich werde zu diesem Mini-Abschnitt kein extra Sample bauen, aber mit nur einem Befehl solltet ihr wohl noch gerade klarkommen.
Es handelt sich hierbei um den glulookat – Befehl, welcher quasi eine Kamera simuliert. Dieser Befehl verlangt ganze 9! Parameter, weil er wirklich die ganze Kameraführung beinhält:

Die ersten drei Parameter sind Punktangaben, welche genau Aussagen, wo die Kamera eigentlich steht... (weshalb 3 Zahlen ist klar: x,y und z Koordinaten)
Das nächste Tripel an Zahlen ist ein Vektor und stellt dar, wohin die Kamera guckt... wenn also das stände 0,-1,-1 , dann würde die Kam nach vorne-unten gucken (im 45°Winkel)...
Fehlt noch das letzte Zahlentripel... dieses hat im Grunde nur einen Job: sagen, wie rum die Kamera steht! (bzw. welche Seite oben ist) ... normal ist der Wert 0,1,0 --> Die Positive y – Achse ist oben ... Es kann aber auch mal eine solche Kamera zustande kommen:
glulookat(0,4,0,0,-1,-1,0,-1,0)
Das Resultat wäre denn eine Kamera, welche im Ursprung 4 einheiten hoch stände, nach Vorne und Unten guckt und dabei auf dem Kopf steht... na ja, wer's braucht *ggg*

Die Kraft der zwei Texturen

So Leute, nun sind wir schon soweit gekommen, nun muss ich euch nochmals mit was nerven... Wobei: an sich ist es ganz einfach, was ich will: Überlegt euch mal, wie wir bislang immer Lightmaps und andere Tricks mit mehreren Texturen realisiert haben... Richtig, immer nach dem Muster:

- Grundtextur setzen
- An die richtige Stelle Transformieren
- Das Objekt zeichnen
- Lichttextur setzen
- An die richtige Stelle Transformieren
- Das Objekt zeichnen

Irgendwie etwas umständlich, oder? Besonders nicht gerade das beste für unsere Geschwindigkeit... Was also tun? Na ja, ab den alten TNT (1) Grakas gibt's da so eine nette Möglichkeit, welche sich Multitexturring nennt. Dieses Multitexturring hat den Vorteil, dass das Objekt gleich mit mehreren Texturen gerendert wird und wir damit wertvolle Geometrieleistung sparen.
Das erste, was wir tun sollten ist, dass wir die Grafikkarte überhaupt mal Fragen sollten, ob überhaupt mindestens zwei Texturen-Units zur Verfügung stehen (eine Grafikkarte kann durchaus mehr haben! Alle neuen Karten, wie die neueren Geforce haben 8 davon... aber wir arbeiten erstmal nur mit 2 :-))
Dazu gibt es eine ganz einfache Methode (wie so oft in Oncreate ausgeführt):

```
glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB, @TexUnits);
 if TexUnits<2 then begin
  showmessage('Grafikkarte unterstützt kein Multitexturing!');
  halt(1);
 end;
```

Ach ja, die Variable „TexUnits" ist ein lokaler Integer ;-)
Was wird hier also gemacht?
Naja, zunächst lesen wir mit „glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB, @TexUnits);" die Anzahl der Texturunits aus und denn prüfen wir, ob es mindestens zwei sind... wenn ja, denn läuft alles normal Weiter, wenn nein, denn wird das Programm beendet.

Läuft euer Programm weiter und beendet sich nun nicht? Denn ist alles in Ordnung und ihr könnt fortfahren. Leider kann es in diesem Falle Besitzer älterer Karten treffen, für die nun leider Entstation ist. Da kann man leider nicht viel machen :-(

Aber keine Sorge: noch soll für euch nicht Exitus sein im Kurs, nur dieses Kapitel läuft halt nicht... Sorry.

Für alle, die nun weitermachen können geht's nun um die Wurst... wir müssen beim setzen („binden") der Textur klar machen, welche Texturunit wir brauchen, also ran ans Werk:

```
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
grundtextur.bind;
```

Was geht da denn ab ?
Keine Panik, Onkel T kommt ja schon und hilft euch *g*
Also das erste, was wir machen ist, die Texturunit unserer Wahl zu wählen. Hierbei werden - wie so oft in Delphi - die Textur – Units von „0" ab gezählt... wir Sprechen hier also die 1. Texturunit an.
Die nächste Zeile ist easy, oder? Na ja: damit aktivieren wir an sich nur die Texturunit ... wir wollen die Textur ja schließlich sehen...
Die nächste Zeile dürfte euch nun ein Wenig befremdlich vorkommen, es ist aber halb so wild...
An sich sagt diese Zeile nämlich nur, wie Open GL mit der Textur umgehen soll... Das hat man sich so vorzustellen, als das eine Textur ja theoretisch mehrere Möglichkeiten hat, wie z.b. alle anderen Texturen zu überdecke (in diesem Falle ziemlich schwachsinneig) oder wie in diesem Falle zu „modulieren" (Gl_Modulate halt). Wenn nun alle Texturunits (in unserem Falle zwei) so eingestellt werden, denn werden für das Gesamtergebnis einfach alle Texturwerte multipliziert ... ist im übrigen für fast alle Anwendungen die richtige (Standart-) Einstellung.
Dasgleich machen wir nun auch noch mal mit unserer Lightmap:

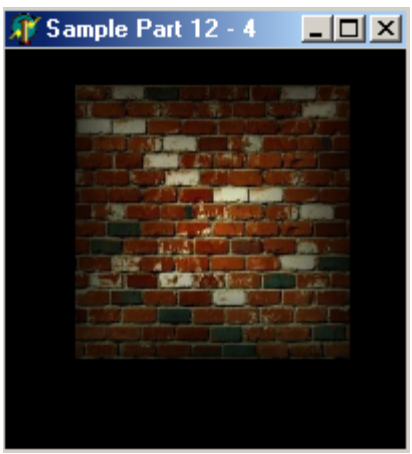
```
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
lightmap.bind;
```

Und schon sind wir soweit, dass wir in den ersten zwei Texturunits eine Textur haben (eine Grund und eine Licht-Textur) und diese nun zeichnen wollen...
Nun haben wir an sich nur noch eine Hürde zu nehmen: die Texturkoordinaten!
Wie ihr wisst, werden wir in diesem Falle unser Objekt (Im Sample in einfaches Quadrat) nur ein einziges Mal zeichnen. Das heißt zugleich, dass man (leider) beide Texturkoordinaten vor dem Vertex setzen muss. Dafür gibt es in Open Gl einen extra Befehl, welcher sich „glMultiTexCoord2fARB" nennt. Dieser Befehl erwartet 3 Parameter. Der erste ist die gewünschte Texturunit für die das gelten soll und die beiden anderen sind unsere zwei Texturkoordinaten, die wir ja bereits aus Teil 5 kennen ;-)
Meine ganze Render-Prozedur sieht nun (mit allen Schikanen) so aus:

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

glActiveTextureARB(GL_TEXTURE0_ARB);
```

```
glEnable(GL_TEXTURE_2D);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
grundtextur.bind;

glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
lightmap.bind;

glbegin(gl_quads);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB,0,0);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB,0,0);
glvertex3f(-1,-1,-4);

glMultiTexCoord2fARB(GL_TEXTURE0_ARB,1,0);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB,1,0);
glvertex3f( 1,-1,-4);

glMultiTexCoord2fARB(GL_TEXTURE0_ARB,1,1);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB,1,1);
glvertex3f( 1, 1,-4);

glMultiTexCoord2fARB(GL_TEXTURE0_ARB,0,1);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB,0,1);
glvertex3f(-1, 1,-4);
glend;
SwapBuffers(form1.myDC);
end;
```

Und das Ergebniss? Naja:



Achtung!
Wundert euch bitte nicht bei der Tatsache, dass ich für das Beispiel eine OpenGL12.pas verwendet habe und das da zwei neue Befehle in der Initialisierung auftauchen:

```
myRC := CreateRenderingContext(myDC, [opDoubleBuffered], 32, 0, 0, 0, 0, 0, myPalette);
ActivateRenderingContext(myDC, myRC);
```

Das hängt schlicht und einfach damit zusammen, dass ich mit der OpenGl15.pas massive Probleme mit dem Multitexturing hatte. Leider ließen sich diese Fehler vor Beendigung des Tuts nicht mehr beheben... ich werde aber schnellstmöglich ein neues Beispielprogramm nachliefern!

Modelle über Modelle

Wir alle kennen sie, wir alle mögen sie, aber meistens erschießen wir sie: 3d Modelle *g*
Heute soll also der große Tag da sein, wo wir endlich die ersten von ihnen in unsere Open Gl Anwendungen laden werden.
Leider bin ich selber noch nicht ganz dazu in der Lage einen eigenen Loader zu schrieben (das wär auch was *ggg*), aber ich will euch hier zumindest einen kleinen Loader vorstellen, mit dem ihr sehr einfach *.3ds – Dateien öffnen könnt.
Der Loader stammt vom Noeska, einem Mitglied des DGL-Forums, welcher es geschafft hat einen sehr simplen Loader zu kreieren, welcher wirklich für jedermann brauchbar ist.
(Eines Version des Loaders liegt dem Beispiel bei)
Um nun also eine 3DS-File zu laden brauchen wir zunächst nichts weiter als eine globale Variable vom Typ „TAll3dsMesh“ (natürlich erst, nachdem ihr die Unit „gl3ds“ per „uses“ eingebunden habt... um die anderen Units braucht ihr euch noch nicht zu kümmern, das sind nur Hilfsunits für den Loader)... der einfachheit halber habe ich die Variable einfach mal „model“ genannt.
So, nun müssen wir ein Model laden. Dies machen wir (wie immer) in unserer OnCreate-Prozedur:

```
…
model:=TAll3DSMesh.Create(nil);
model.TexturePath:='textur\';
model.LoadFromFile('model\Messer.3ds');

Application.OnIdle:= MyIdle;
```

Also wem ich das nun erklären muss, der ist in Teil 12 fehl am Platz *g*
An sich ist es ganz einfach: zuerst initialisieren wir die Variable, denn geben wir den Pfad für die Textur(en) an und zu guter Letzt laden wir das ganze...
An sich sind wir nun schon fast fertig! Wir brauchen es an sich ja nur noch anzeigen:

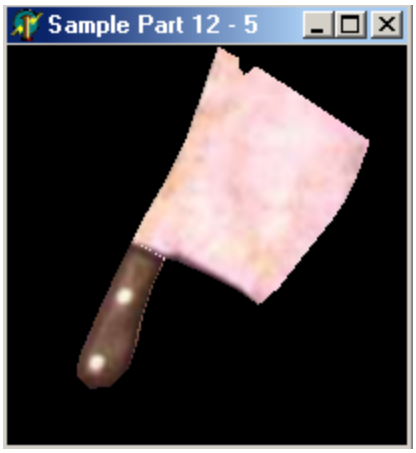
```
procedure render;
begin
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
glLoadIdentity;
gltranslatef(0,0,-6);
glrotatef(90,0,1,0);
glscalef(0.1,0.1,0.1);
model.render;
SwapBuffers(form1.myDC);
end;
```

An sich gibt’s da nicht viel zu sagen, außer, dass ich euch eine Erklärung schuldig bin, weshalb wir das Model auf 1/10 seiner Größe schrumpfen... das hängt damit zusammen, dass 3ds-Modelle aus irgendeinem Grunde immer viel zu groß gespeichert werden ... aber macht ja nüscht ;-)

Nun müssen wir am ende des Programms nur noch den lieben Speicher wieder freigeben:

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

Und schon dürfte diesem Anblick nichts mehr im Wege stehen:



Im übrigen ist das 3ds-Modell nicht von mir, sondern von Logaan - einem eifrigem Leser dieses Kurses, welcher mir Model und Textur freundlicherweise zur Verfügung gestellt hat... Auf seinem Mist ist es im übrigen auch gewachsen, dass dieses Kapitel überhaupt im Tut ist, also bedankt euch bei ihm *g*
Ich hoffe, dies ist ein kleiner Anreiz für euch, auch mal eure Wünsche zu äußern ;-)

PS: begeisterte Milkshape – User müssen nun nicht traurig sein. Auf dem „quasi - Mekka“ für deutschsprachige Delphi und Open Gl – Anhänger www.delphigl.com gibt es einen sehr guten Loader mit Beispiel. Ich bin in diesem Tut nicht mehr drauf eingegangen, weil das mein Zeitkontingent zum Ende hin doch etwas arg strapaziert hätte... Aber einen Vorteil könnt ihr Milkshape – User euch dennoch anrechnen: mit dem Loader kann man auch animierte Objekte laden ;-)

3d Fonts

Eigentlich sollte an dieser Stelle schon Schluss sein, weil ich Schussel schlicht und einfach ein Thema vergessen habe. Aber so habt ihr noch mal glück gehabt und kommt hier in den Genuss eines Kapitels, welches erstens erst am Tage der Veröffentlichung des Kurses entstanden ist und zugleich wie auch das Kapitel über 3d Modelle aufgrund von Leseranfragen hier reingerutscht ist.
Wir wollen also versuchen 3d-fonts zu erstellen und anzuzeigen. Dass ich hier keinen Mega-Teil draus machen kann, sondern nur ein wenig Code poste dürfte hoffentlich klar sein :-)
Um also 3d-Fonts auf den Schirm zu bringen müssen wir zunächst eine globale Variable anlegen, welche ein Displayliste kapseln soll (Typ: tglint)
Diese Displayliste ist dazu da, damit wir irgendwo die kompletten Buchstaben des Alphabetes speichern können. Aber wie legen wir diese Liste(n) nun an? Nun ja, wir werden sicherlich nicht jeden Polygon einzeln berechnen *g*. Eher greife ich da zu folgender Prozedur:

```
procedure BuildFont;                                    // Build Our  Font
var font: HFONT;                                        // Windows Font ID
   gmf : array [0..255] of GLYPHMETRICSFLOAT;     // Address Buffer For Font Storage
begin
  displayliste := glGenLists(256);                      // Storage For Characters
  font := CreateFont(12,                                // Height Of Font
         0,                                             // Width Of Font
         0,                                             // Angle Of Escapement
         0,                                             // Orientation Angle
         0,                                             // Schrift Fett? Nein!
         0,                                             // Schrift Kursiv? Nein!
         0,                                             // Auch nicht unterstrichen….
         0,                                             // … oder gar durchgestrichen
         ANSI_CHARSET,                                  // Character Set Identifier
         OUT_TT_PRECIS,                                 // Output Precision
         CLIP_DEFAULT_PRECIS,                           // Clipping Precision
         ANTIALIASED_QUALITY,                           // Output Quality
         FF_DONTCARE or DEFAULT_PITCH,                  // Family And Pitch
         'Arial Black');                                // Font Name

  SelectObject(form1.myDC, font);                       // Die Schriftart Wählen

  wglUseFontOutlines(   form1.myDC,             // Der Aktuelle DC
                  0,                            // Der erste “Buchstabe”
                  255,                          // Die gesamte Anzahl der Buchstaben
                  displayliste,                 // Die Displayliste
                  0.0,                          // Deviation From The True Outlines
                  0.3,                          // Die “Dicke” der Schrift in Z-Richtung
                  WGL_FONT_POLYGONS,            // Wir wollen ganze Polygone, keine Linien
                  @gmf);                        // Unserer Puffer

end;
```

Der Einfachheit halber habe hier mal alle Kommentare, welche mir wichtig erschienen in ’s Deutsche übersetzt. (Der Originalcode hatte nur Englische)
Natürlich muss dieses ganzes nun auch noch aufgerufen werden … am besten relativ am Anfang des Programms (wo, dürft ihr raten *ggg* ein Tipp: guckt mal nach oben in dieses Dokument und seht hin, wo die meisten Initialisierungen vorgenommen wurden *g* notfalls habt ihr ja noch das Sample :-))

So, das nächste, was wir machen müssen ist eine Prozedur schreiben, welche uns dazu befähigt, den Text nun auch darzustellen:

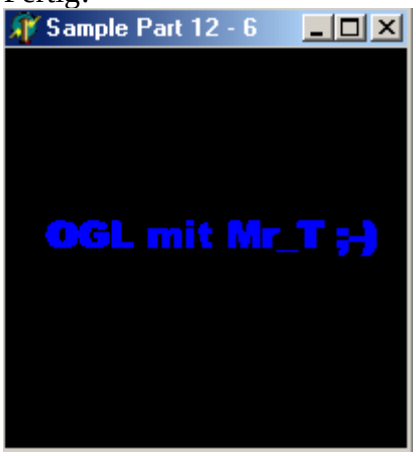
```
procedure glPrint(text : pchar);                // Custom GL "Print" Routine
begin
  if (text = '') then                           // If There's No Text
        Exit;                                   // Do Nothing

  glPushAttrib(GL_LIST_BIT);                    // Pushes The Display List Bits
  glListBase(displayliste);                     // Sets The Base Character
  glCallLists(length(text), GL_UNSIGNED_BYTE, text);   // Draws The Display List Text
```

```
  glPopAttrib();                    // Pops The Display List Bits
end;
```

Ich hoffe, das ist nicht zu schwer zu verstehen (habe leider keine Zeit, dass detailliert zu erklären :-( )
Na ja, damit haben wir es aber fast schon! Nun können wir beim Rendern nämlich ganz Bequem ein wenig Text anzeigen lassen:

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

gltranslatef(-3,0,-9);
glcolor3f(0,0,1);
glprint('OGL mit Mr_T ;-)');

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

Fertig!



Nachwort

Nu ist er also vorbei, der 12. Teil des Kurses :-(
Ich muss ganz ehrlich sagen, dass mir das schreiben dieses Teils echt nicht leicht gefallen ist, da es zu viele Themen gab, welche ich abhandeln wollte und ich leider denn zwischendurch auch noch eine Null Bock – Phase auf Open Gl hatte ... aber ich wollte unbedingt pünktlich zum Jahrestag des Kurses diesen 12. Teil bringen und hier ist er *g*
Was euch im nächsten Teil genau erwartet, wird hier noch nicht verraten, aber ich gebe mal n kleinen Tip: ihr solltet euch schon mal mit dem allseits beliebten Q3Radiant oder auch alternativen wie Quark anfreunden :-) Schon ne Ahnung, was es werden wird? Ja, ganz nahe dran *ggg*
Bis dann denn

Mr_T