Open Gl - Delphi – Kurs Teil 10: Wie selektiert man Objekte

Einleitung

Hi Leute!
Eigentlich dachte ich schon fast, wir könnten uns diesen Jubiläums-Part fast schenken, da ihr Mitlehrerweile schon verdammt viel wisst:
Seht euch doch mal um: alles nur n ganzer Haufen an Quadraten, Würfeln und abgerundeten Gegenständen, noch dazu optimal beleuchtet und texturiert. Theoretisch könnten wir das alles verdammt gut nachbauen und wenn wir uns richtig viel Mühe geben und etwas zu viel Zeit über haben, dann sieht das ganze hinterher auch noch recht gut aus, was wollen wir mehr?
Na ja, wenn ich hier gerade so in die Tasten haue, da fällt mich etwas ein, was wir bislang leider noch nicht können: ich interagiere mit meiner Welt bzw. ich bediene die Tastatur. Was wäre das den für eine Welt, wenn wir nichts anheben, Türen öffnen oder einen Lichtschalter betätigen könnten? Na ja: eine ziemlich üble würde ich mal so vermuten...
Und genau darum geht es in meinem Tut heute!
Ich denke mal, ihr alle werdet erkannt haben, dass es wirklich nicht gerade einfach ist, herauszubekommen, auf welches Objekt eurer Szene der User gerade geklickt hat, obwohl diese Infos für viele Programme ziemlich unerlässlich sind. Stellt euch mal vor, ihr habt einen super genialen Map-Editor gebastelt und ihr könnt nicht herausfinden, auf welches Objekt der User gerade zwecks Editieren geklickt hat! Oder ihr habt ein super geniales Hacker-Agenten-Spiel geschrieben und ihr wisst nicht, welche Kombination der gute Herr Spieler gerade in den Computer einer Sicherheitstür getippt hat! Wäre doch ne echt schwache Leistung, oder?
Und um dieses Defizit auszugleichen habe ich hier niedergeschrieben, wie es geht.

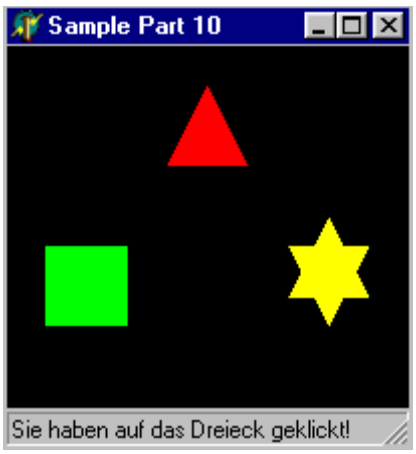
Open Gl und der Name-Stack

Etwas (aber nur ganz wenig) Theorie vorweg.
Und zwar will ich euch zunächst sagen, wie Open Gl Namen überhaupt ansieht: und zwar als reine Integer-Werte! Es gibt keine echten Namen, sondern leider nur Nummern, die man seiner Szene zuweisen kann...
Dass soll uns aber nicht wirklich davon abhalten, auch echte Namen zu verwenden. So gibt man normalerweise die Namen selber als Konstanten an, aber wir können ja zum Beispiel ein Array erschaffen, welches die dazugehörigen Namen enthält... Das ganze könnte man sich dann so vorstellen:

```
const
  dreieck : 1;
  viereck : 2;
  stern : 3;
  namen : array[-1..3] of string = ('nichts', '', 'das Dreieck', 'das Viereck', 'den Stern');
```

Erklären muss ich das nicht wirklich, oder?
Na ja evtl. wäre es für euch noch ganz interessant, weshalb das Namens-Array den Wert „-1“ hat:

Open Gl kann nämlich auch den Wert „-1“ ausgeben, wenn man auf rein gar nichts geklickt hat! Daher brauchen wir auch ein Element, welches „-1“ heißt.
Damit euch das ganze etwas klarer wird, stelle ich euch nun erst mal vor, was ich heute mit euch machen will:



Dieses nun wirklich einfache Programm hat folgende Funktion:
Es werden wie man sieht ein Dreieck, ein Viereck und ein Stern gezeichnet und je nachdem, auf was man geklickt hat, soll die Message unten in der Statusbar angezeigt werden.
Wie man eine derartige (für unsere Verhältnisse einfache) Szene rendert, dürfte recht einfach sein, nicht aber, wie man den Objekten nun die Namen zuweißt... aber das zeige ich euch hier:

```
procedure render;
begin
glMatrixMode(GL_MODELVIEW);
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT); //Farb und
Tiefenpuffer löschen
glLoadIdentity;
gltranslate(0,0,-6);

glinitnames;
glpushname(0);

glcolor3f(1,0,0);
glloadname(dreieck);
glbegin(gl_triangles);
glvertex3f(-0.5,1,0);
glvertex3f(0.5,1,0);
glvertex3f(0,2,0);
glend;

glcolor3f(0,1,0);
glloadname(viereck);
glbegin(gl_quads);
glvertex3f(-1,0,0);
glvertex3f(-2,0,0);
glvertex3f(-2,-1,0);
glvertex3f(-1,-1,0);
glend;
```

```
glcolor3f(1,1,0);
glloadname(stern);
glbegin(gl_triangles);
glvertex3f(1,0,0);
glvertex3f(2,0,0);
glvertex3f(1.5,-1,0);

glvertex3f(1,-0.65,0);
glvertex3f(2,-0.65,0);
glvertex3f(1.5,0.35,0);
glend;

SwapBuffers(form1.myDC);                    //scene ausgeben
end;
```

Also, das Erste ist, dass wir auf jeden Fall in die Model View Matrix schalten müssen. In dieser Matrix sind wir zwar auch normalerweise, aber in unserer späteren Selection – Funktion müssen wir zeitweilig in eine andere. Da es zum Rendern aber unabdingbar ist, dass wir gerade diese Matrix gesetzt haben, wechsle ich auf jeden fall vor dem Rendern in diese... sicher ist sicher :-)

Die nächsten markierten Zeilen sind „glinitnames;“ und „glpushname(0);“
Diese beiden Zeilen Quellcode bewirken, dass zunächst mal der Namen – Stack initialisiert wird und danach der „Name“ „0“ auf diesen Stack geschoben wird. Die Null hat eine ganz einfache Funktion: Wird nicht vor dem Laden des ersten Namens mindestens eine Zahl auf den Stack gepusht, dann endet der Versuch, einen Namen zu laden mit einem netten Error... wer’s nicht glaubt, kann es ja gerne mal ausprobieren.

Der Befehl „glloadname“, der den restlichen interessanten Teil dieser Render – Prozedur stellt, hat die einfache Wirkung, dass Open GL nun einen Namen lädt (halt aus unserer Konstanten) und dass alle Objekte, die nun folgen, mit diesem „Namen“ belegt werden, bis ein weiteres mal „glloadname“ aufgerufen wird... ist doch easy, oder?

Leider wird es nicht so einfach bleiben...
Na ja, nützt ja nichts, nun müssen wir nämlich die Funktion anlegen, welche die eigentliche Selektion durchführt. Ein Tipp: Die Funktion muss im Quelltext unter der Render-Pozedur angelegt werden, weil dieselbige nämlich in dieser Funktion aufgerufen wird und Delphi ansonsten die Prozedur nicht findet.

```
function Selection : integer;
var
Puffer      : array[0..256] of GLUInt;
Viewport    : array[0..3] of Integer;
Treffer,i   : Integer;
Z_Wert      : GLUInt;
Getroffen   : GLUInt;
begin
glGetIntegerv(GL_VIEWPORT, @viewport);      //Die Sicht speichern
glSelectBuffer(256, @Puffer);                 //Den Puffer zuordnen
glRenderMode(GL_SELECT);                    //In den Selectionsmodus schalten
```

```
glmatrixmode(gl_projection);          //In den Projektionsmodus
glPushMatrix;                         //Um unsere Matrix zu sichern
glLoadIdentity;                       //Und dieselbige wieder zurückzusetzen

gluPickMatrix(xs, viewport[3]-ys, 1.0, 1.0, @viewport);
gluPerspective(45.0, form1.ClientWidth/form1.ClientHeight, 1, 100);

render;                               //Die Szene zeichnen
glmatrixmode(gl_projection);          //Wieder in den Projektionsmodus
glPopMatrix;                          //um unsere alte Matrix wiederherzustellen

treffer := glRenderMode(GL_RENDER);   //Anzahl der Treffer auslesen

Getroffen := High(GLUInt);            //Höchsten möglichen Wert annehmen
Z_Wert := High(GLUInt);               //Höchsten Z - Wert
for i := 0 to Treffer-1 do
if Puffer[(i*4)+1] < Z_Wert then
 begin
 getroffen      := Puffer[(i*4)+3];
 Z_Wert := Puffer[(i*4)+1];
 end;

 Result := getroffen;
end;
```

Alle Stellen, zu denen ich bereits Kommentare geschrieben habe, brauche ich (glaube ich) nicht mehr wirklich erklären... aber zu den Variablen will ich durchaus was sagen:
Der „Puffer“ ist der eigentliche Selektion-Puffer. In ihm werden die Ergebnisse der Selektionsprüfung gespeichert. Das Array „Viewport“ speichert quasi unsere View, da die ja auch irgendwie wichtig ist ;-)
Die Variable „Treffer“ ist ein reiner Integerwert, der die Anzahl der Treffer aufnimmt und die Variable „Z_Wert“ nimmt die Tiefendaten auf.

Was nun in dieser Funktion passiert, ist einfach zu beschreiben. Zunächst wird unsere View zwischengespeichert und dann Open Gl klar gemacht, was für einen Puffer mit welcher Größe es verwenden soll. Eine Größe von 64 ist quasi der Normalwert, da jede Open Gl Variante (auf den verschiednen Grafikkarten) diese Größe mindestens liefern muss... größere werte sind aber durchaus möglich.
Der Puffer hat ja nun an sich aber die 4-Fache Größe dieses Wertes. (Bzw. wir übergeben als Parametergröße ja auch den Wert 256) Das hat folgenden Grund:
Für jedes getroffene Objekt werden immer 4 werte gespeichert:
1) Anzahl der Namen auf dem Stack
2) Kleinester Z-Wert des getroffenen Objektes
3) Größer Z-Wert des getroffenen Objektes
4) Name des Objektes

Danach wird in den Selection – Modus geschaltet, die Matrix konfiguriert und Open Gl übergeben, welche x bzw. y- Koordinate auf dem Bildschirm angeklickt wurde. Dieses geschieht in der Prozedur „gluPickMatrix(xs, viewport[3]-ys, 1.0, 1.0, @viewport);“.
In dieser Stehen „xs“ bzw. „ys“ für unsere X und Y – Koordinate, auf die geklickt wurde. Die nächsten zwei Parameter bilden eine Art „Klick-Rechteck“. Wenn wir da einen Wert >1

angeben, dann wird praktisch eine Art Rechteck um den Punkt, auf den wir geklickt haben, gebastelt, welches Komplett ausgewertet wird. Dadurch würde nicht nur das Objekt, welches direkt unter dem Mauscursor lag ausgewertet, sondern auch die Objekte, welche evtl. direkt daneben lagen.
Mit der Zeile „treffer := glRenderMode(GL_RENDER);“ fragt man nun noch ab, wie viele Treffer es insgesamt gegeben hat. Hierbei gilt es zu beachten: angenommen, man hat eine Große Szene und klickt auf ein Objekt. Dann ist schon fast davon auszugehen, dass dahinter noch irgendetwas liegt (kann auch ganze 100 Einheiten entfernt liegen... das spielt keine echte Rolle), dann werden auch diese Objekte als Treffer zurückgegeben. Um nun nur das Objekt herauszubekommen, welches der Kamera am nächsten lag, ließt man nun in einer Schleife alle Z-Werte „Puffer[(i*4)+1]“ der getroffenen Objekte aus und vergleicht sie mit dem bislang niedrigsten Wert. Ist der ausgelesene Wert niedriger, als der bislang kleinste, dann wird der neue Z-Wert gespeichert und der Name des bislang nächsten Objektes wird gespeichert.
Ist die Schleife durchgelaufen, dann steht am Ende also in der Variable „getroffen“ der Integer – Name des nächsten, getroffenen Objektes, welches wir dann auch als Rückgabewert der Funktion verwenden.

Das war nun alles zwar recht viel komplizierter Kram, aber ich kann euch versichern, das mehr nicht kommt. Denn mehr, als das was wir nun eben geschrieben haben, werden wir nie für einen Selektionsvorgang brauchen... egal, ob wir den genialsten Geheimdiensttriller der Welt oder nur einen Moorhuhn – Klon schreiben. *g*

Zu guter Letzt müssen wir nun noch diese Prozedur aufrufen, wozu ich das „onmousedown“ – Ereignis des Formulars verwendet habe:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
xs := x;
ys := y;

statusbar1.Simpletext := 'Sie haben auf ' + namen[selection] + ' geklickt!';
end;
```

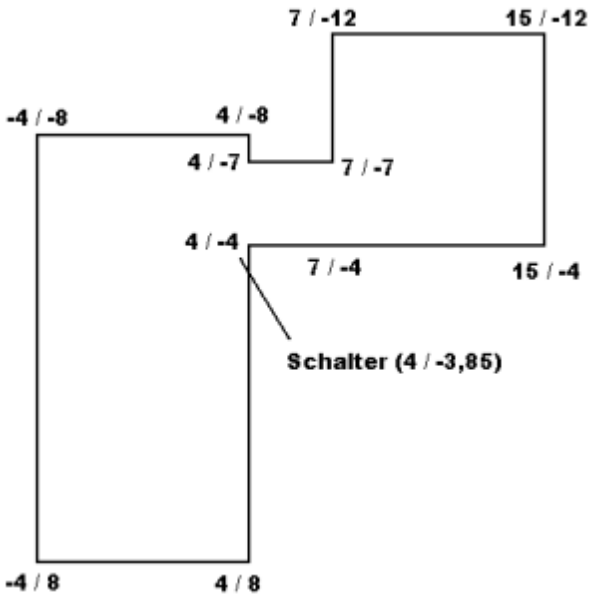
Ich denke, es besteht kaum Erklärungsbedarf...

Geht’s auch größer?

Jupp, natürlich kann man diese ganze Technik auch zu was anderem verwenden, als nur Formen zu identifizieren. (Das Sample oben ist ja nun wirklich unterhalb unseres Niveaus, auch wenn es einen schönen Einblick in die Materie gebracht hat)
Aber – größenwahnsinnig, wie ich nun mal bin – ich strebe zu größerem *muhaha*:
Der Weltherrschaft ;-)
Nein, Spaß beiseite, ich will euch noch eben ein kleines Briefing geben, wie ihr auch etwas größeres bauen könnt.
So habe ich ein kleines Sample gebastelt, in welchem man zwei Räume hat (mit einem kleinen Zwischengang), von denen der zweite Raum unbeleuchtet ist, wobei man das Licht

mithilfe eines Schalters im ersten (beleuchtetem) Raum einschalten kann. Das ganze sieht dann etwas so aus: (links: Licht aus, rechts: Licht an)

Das macht doch durchaus was her, oder? Ok, ich gebe zu, die Texturen sind nicht alle von mir (die Lightmaps aber schon), aber ansonsten finde ich die Szene recht gelungen. Um euch eine kleine Bauanleitung zu geben, habe ich hier eben den Grundriss der Szene für euch:



An sich sollte so was für euch kein echtes Problem mehr sein... die x und z – Koords stehen alle auf dem Plan, den ihr hier seht. Dazu sei gesagt, dass alles in der Szene genau 2 Einheiten hoch ist.
Des weiteren könnte es für euch noch nützlich sein, zu wissen, dass ich für die Lightmaps den Blendmodus „glblendfunc(gl_dst_color, gl_zero);“ verwende und dass ansonsten alles etwa so ist, wie auch in den Teilen 7 und 8.
Das letzte, was ein kleiner Stolperstein sein könnte, ist die Distanz zum Schalter. Es ist zwar schon wunderherrlich, wenn man ihn überhaupt betätigen kann, aber irgendwie ist es auch schwachsinnig, wenn man am anderen Ende des Raumes stehen kann und ihn dennoch verwenden kann. Deshalb habe ich meine klick- Prozedur so verändert, dass die Distanz zum Schalter, wen er funzen soll höchstens 3 Einheiten betragen darf:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
xdis : double;
ydis : double;
begin
xdis := my_x+4;
ydis := my_y-3.85;

xs := x;
ys := y;

if selection=1 then
begin
 if sqrt(xdis*xdis + ydis*ydis) <= 3 then
 lichtan := not lichtan;
end;

end;
```

Und bitte nicht vergessen: bindet die Unit „Math“ mit ein... ansonsten dürftet ihr auf die Wurzelfunktion (sqrt) keinen Zugriff haben! Im übrigen: Der Schalter hat von mir den Namen „1“ bekommen und die ganze Umgebung hat die Nummer „2“.

Damit solltet ihr eigentlich kaum ein Problem haben, dieses Sample zu verstehen, bzw. nachzubasteln. Notfalls stehe ich per Mail, ICQ oder auch in unserem DCW- Board immer zur Hilfe zur Verfügung. Im übrigen: falls ihr mal hängt, könnt ihr ja auch ins Sample gucken und sehen, wie ich es gelöst habe.

Nachwort

Stellt euch mal vor, dass war auch schon der 10. Teil meines Kurses, aber es ist entgegen alter Planungen noch kein Ende! Die Roadmap ist quasi fertig und wenn nichts dazwischen kommt, dürften sogar noch Anfang 2004 einige Teile das Licht der Welt erblicken. (Im wahrsten Sinne des Wortes... irgendwo da wird wohl Teil 16 liegen, für den Licht geplant ist) Wenn alles gut geht und ich nicht zwischendurch auf andere dumme Gedanken komme, dann sehen wir uns also nächstes Mal wieder, wenn wir etwas mit dem Stencil Buffer rumspielen und versuchen, damit unseren Szenen eine völlig neue Perspektive zugeben. (Was genau ich vorhabe, verrate ich noch nicht ;-)) Ich hoffe, ihr seit auch dann wieder dabei.

Mr_T