

Open Gl - Delphi – Kurs Teil 8: Bewegung und die erste große Szene

Einleitung

Hi Leute! Schon wieder ist viel zu viel Zeit vergangen, seitdem der 7. Teil wortwörtlich das Licht der Welt erblickt hat (besser: er hat versucht etwas Licht in diese Dunkle Welt zu bringen *g*). Diesmal wollen wir uns mit etwas Bewegung beschäftigen, wenn dieses auch nicht viel mit Open Gl sondern eher was mir Mathe zu tun hat (ich weiß, die Ferien sind gerade vorbei, aber da müsst ihr nun durch). Aber ihr werdet ausreichend verschont: es wartet das bislang größte Sample auf euch, welches endlich komplett Schluss macht mit den einfachen, starren Szenen. Naja, ich sollte nicht so viel labern, denn nun kommt das GRAUEN!

Ahh! Mathe!!!

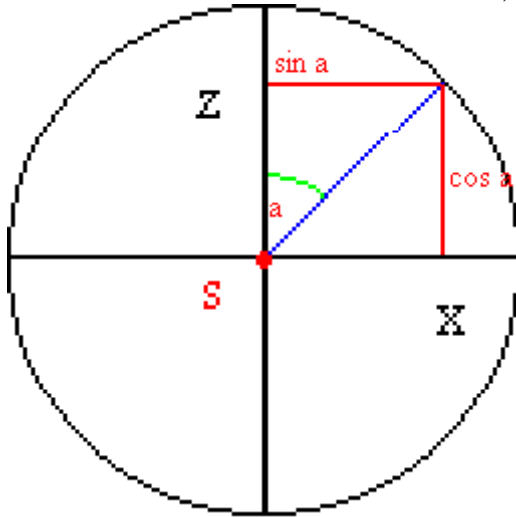
Ok ich weiß, dass einige diesen Abschnitt sicher nicht ganz nötig haben (was weiß ich: nachher liebt dieses Tut noch ein Mathe-Prof. *lol*). Für einige (jüngere) unter uns wird das hier aber evtl. was ganz neues sein. Nicht beunruhigt sein: ich habe ein paar Erfahrungen vom Mathe-Nachhilfegeben (evtl. lernt ihr hier ja wirklich was und wenn dann eure Lehrer mit der Miniabi bzw. Abschlussarbeit ende der 10. ankommen sind die ganz platt, das ihr das könnt *g*).

Also: wir wollen uns mit der Bewegung im Sinne aktueller Shooter beschäftigen. Dort ist man es ja gewöhnt, dass man mit der Maus seine (bald tote) Spielfigur drehen und mit den Pfeiltasten bewegen kann. Drehen würdet ihr sicher auch hinbekommen, da wir das ja bereits in Teil 3 gelernt haben, wie man Objekte dreht. Bewegen ist an sich auch nicht so schwer... wozu gibt es den „gltranslate“ – Befehl... Aber nun stellt euch mal vor: was macht ihr, wenn man z.B. um 45° gedreht steht? Geradeaus ist ja noch einfach (die z-koord um die schrittlänge verringern) und um 90° gedreht auch (x-koord erhöhen), aber was machen wir mit den ganzen Zwischenpositionen. (Wohl kaum sowohl um eins nach recht und eins nach vorne bewegen... jedes auch noch so schwache Augenmaß sieht, das die zurückgelegte Strecke viel zu groß ist) Was wir aber wohl wissen: Alle Punkte, zu denen wir uns hinbewegen könnten, egal in welcher Richtung sie liegen haben eines gemeinsam: die Distanz zu unserem Standpunkt. Nun strengen wir unsere (hoffentlich) noch nicht ganz weggesoffenen Gehirn-Zellen an und überlegen, an was uns das erinnern könnte: Richtig, ein Kreis!

Und hier setzt die Mathematik ein. In der Mathematik gibt es eine so genannte Sinus-Funktion. Diese Funktion ist dazu in der Lage, die von uns benötigten Koordinaten herauszubekommen (fragt mich nicht wie...) Dazu muss die Funktion an sich nur mit einem Winkel (in Grad gefüttert werden). Das ganze sieht dann so aus: Man gibt z.B. als Winkel „0“ an. Dann liefert der Sinus ebenfalls den Wert „0“. Damit hätten wir ja an sich schon unsere x-Verschiebung. Geben wir dem Ding den Wert „90“, dann bekommen wir den Wert „1“ wieder raus, was soviel heißt wie: volle Granate in x-Richtung. Das lustige (und Praktische): nehmen wir „270“, dann bekommen wir „-1“... das Teil hilft uns also auch dann, wenn wir in die andere Richtung rennen wollen. Bei Zwischenwerten kommt dann halt ne Kommazahl raus. Analog zum Sinus gibt es auch noch den Kosinus. Dieser funzt fast genauso. Der Unterschied: füttert man ihn mit dem Wert „0“ bekommt man „1“ raus und bei dem Wert

„90“ gibt er „0“ von sich. Wenn man nun ein wenig drüber nachdenkt, ist dieser wie geschaffen für unseren Z-Wert! (So wie sich das hier anhört muss man ja fast denken, die Sinus-Funktion wäre nur zur Steuerung von Shooter-Games erschaffen worden ;-)

Um es etwas anschaulicher zu machen, habe ich auch noch ein Bild für euch:



Der große Außenkreis soll die Positionen angeben, wo wir uns hinbewegen könnten. S soll der Punkt sein, auf dem wir stehen und X bzw. Z. sind die X bzw. Z-Achse. „a“ soll der Winkel in Grad sein, um welchem wir gedreht zur Z-Achse stehen und was dann „sin a“ bzw. „cos a“ da zu suchen haben dürfte auch klar sein: über sie berechnet man unseren Neuen Standpunkt (am Ende der Blauen Linie).

Wie man daraus nen Code macht

So... nun wollen wir so langsam mit unserem Beispielprogramm anfangen. Entweder nehmt ihr dazu das Beispiel aus Teil 2 oder ihr entkernt das Texturen-Sample aus Teil 5...

Bevor wir richtig anfangen, möchte ich euch aber zunächst etwas über „Timebased Movement“ erzählen. Wie ihr sicherlich wisst, stellt jeder Rechner eure Projekte mit einer anderen FPS dar. Wenn man nun in der „Render-Prozedur“ das Objekt bewegt, würde es sich auf unterschiedlichen Rechnern unterschiedlich schnell bewegen, was nicht so wirklich Sinn der Sache ist.

Um diesem entgegenzuwirken wollen wir auf unser Formular einen Zweiten Timer setzen, welcher auf eine Sekunde (1000) eingestellt ist. Dieser zählt nun, wie viele Frames in der letzten Sekunde gerendert worden sind und erstellt daraus einen „Zeitfaktor“, welcher umgekehrt zur FPS reagiert: hat man eine Große FPS – Zahl, wird der Zeitfaktor sehr klein, da die Bewegungsschritte pro Frame ja kleiner werden sollen... haben wir eine niedrige FPS, ist er recht groß, da ja pro Frame eine größere Distanz zurückgelegt werden muss, damit das Programm genau so abläuft, wie auf anderen Rechnern.

Wir brauchen in unserem Projekt also insgesamt 5 Variablen:

```
var
  Form1: TForm1;
  xpos : double;
  zpos : double;
  rotation : double;
```

```
fps : integer = 20;  
timefactor : double = 1;  
mauspos : integer = 0;
```

Wozu genau diese Variablen gut sind, könnt ihr euch meistens doch sicher denken... die einzige, die Unklar sein könnte, ist „Mauspos“... diese dient mir an sich nur dazu, herauszufinden, ob der Mauscursor nahe dem Rand unseres Formulars ist bzw. ob ich die Kamera drehen möchte...

Nun müssen wir das Ereignis für unseren 2. Timer bearbeiten:

```
procedure TForm1.Timer2Timer(Sender: TObject);  
begin  
timefactor := 20 / fps;  
fps := 0;  
end;
```

Diese Code ist schnell erklärt:

Der Timer wartet jeweils eine Sekunde und läßt dann aus, wie viele Frames man hatte. (steht in der FPS-Variablen) Daraus wird dann der „timefactor“ berechnet, wobei wir den Wert „20“ als „Normalwert“ nehmen, was bedeutet, dass ein Framecount von 20 den Timefaktor „1“ ergibt... werte kleiner 20 ergeben Zeitfaktoren die deutlich größer als 1 sind und Werte größer 20 ergeben kleine Brüche als Zeitfaktoren.

Nun müssen wir uns noch um unsere Render-Prozedur kümmern:

```
procedure render;  
begin  
glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT); //Farb und  
Tiefenpuffer löschen  
glLoadIdentity;  
  
fps := fps+1; //FPS erhöhen  
if Mauspos = 1 then  
begin  
rotation := rotation + 1 * timefactor; //Rechtsdrehen  
end;  
  
if Mauspos = 2 then  
begin  
rotation := rotation - 1 * timefactor; //Links drehen  
end;  
  
if.GetAsyncKeyState(vk_up) <> 0 then //Nach oben Taste gedrückt?  
begin //Ja: Bewegen  
xpos := xpos + sin(degtorad(rotation)) * timefactor * 0.2;  
zpos := zpos - cos(degtorad(rotation)) * timefactor * 0.2;  
end;  
  
glRotate(rotation,0,1,0); //Die entgültige Drehung  
glTranslate(xpos, 0, zpos); //Die entgültige Bewegung
```

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

Wirklich viel neues Bietet auch diese Prozedur nicht... evtl. bis auf das „degtorad“, welches an sich nur eine Funktion hat: Die Sinus-Funktion in Delphi nimmt leider keine echten Winkel an, sondern leider nur das Bogenmaß. Da wir ja aber leidern nur den Winkel dahaben, muss dieser via „degtorad“ ins Bogenmaß umgerechnet werden.

Ach ja: wir haben noch etwas ganz wichtiges vergessen:
Diese ganzen Mathe-Funktionen, welche wir bislang eingebastelt haben, sind in Delphi nicht standardmäßig eingebunden... um sie nutzen zu können müsst ihr die Unit „math“ in eurer Uses-Liste hinzufügen.

Als nächstes ist nun die Maus-Position gefragt, wozu wir auf das Onmousemove-Ereigniss des Formulars zurückgreifen:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
Y: Integer);
begin
if x < 50 then           //Mauscursor am linken Fensterrand?
begin
mauspos := 1;
end
else
if x > form1.clientwidth-50 then //Mauscursor am rechten Fensterrand?
begin
mauspos := 2;
end
else           //Keins von beidem?
begin
mauspos := 0;
end;
end;
```

So... damit wäre an sich die Kamerabewegung perfekt... leider kann man bei diesem Sample sich nur vorwärts bewegen und nicht rückwärts oder gar seitlich, aber an sich ist dieses auch nicht weiter schwer einzubauen... an sich braucht man ja nur den Wert „0,2“ in „-0,2“ zu editieren und als Butten natürlich vk_down zu nemen und schon haben wir den rückwärtsgang drinne. Zum sliden bräuchte man ja nur so tun, als wenn man um 90° mehr oder weniger gedreht sei und dass man vorwärts laufen würde... ihr wisst schon, was ich mein.
Bloß um was soll man sich denn nun drehen, bzw. rumbewegen ...
Also rein Testweise habe ich das ganze an den 3 Figuren aus Teil 4 getestet, aber ich finde, dass es nicht so ganz das wahre ist. Wir wollen nun also ein größeres Sample coden.

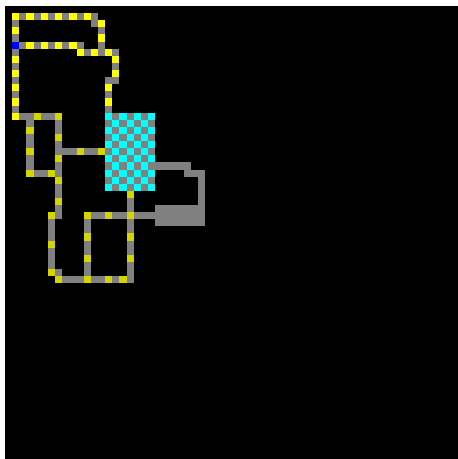
Das Sample

Dann las mal loslegen, gelle?

Nun bitte nicht erschrecken, wenn ihr das Bild seht... so schwer, wie es aussieht, ist es gar nicht.



Aber woraus sollen wir denn nun so einen Gang erstellen? Na ja: als Vorlage verwenden wir diese Bitmap:



Nanu? Wie soll das denn gehen?

Na ja: unser kleines Projekt ist eine kleine Mini - Engine, welche nach diesem Prinzip funzen soll:

Unser Programm lädt die kleine BMP(sie ist in Wirklichkeit nur 64x64 Pixel groß) bzw. lädt die Farbwerte aus der Datei. Nun bekommt unser Programm beim Rendern eine Schleife, welche alle Pixel durchgeht. Ist nun ein Pixel nicht gerade Schwarz, so zeichnet unser Programm an die entsprechende Stelle ein Stück unseres Ganges (unter Berücksichtigung der Nachbarpixel auf grund der Wände). Sollte das Pixel der Map farbig sein, bzw. sein Farbwert ein anderer als 128, 128, 128 (RGB) sein, so wird der Teil des Ganges dazu noch in der Farbe des Pixels beleuchtet.

Das hört sich nun zwar ziemlich schwer zu realisieren an, aber schließlich habe ich es auch geschafft... demnach müsstet ihr es doch zehn mal besser machen ;-)

Fangen wir als mit dem Einlesen des Bildes an.

Dazu nehmen wir das, was wir oben gecodet haben und hängen an die Initialisierungs-Prozedur ein wenig an:

```
procedure TForm1.FormCreate(Sender: TObject);  
var  
bit : Tbitmap;  
x,y : integer;  
begin  
form1.myDC:= GetDC(Handle);  
SetupPixelFormat;  
myRC:= wglCreateContext(myDC);  
wglMakeCurrent(myDC, myRC);  
glEnable(gl_texture_2d);  
glEnable(GL_DEPTH_TEST);  
InitTextures;  
glLoadIdentity;  
  
bit := Tbitmap.create;  
bit.LoadFromFile('Map.bmp');  
for x := 1 to 64 do  
begin  
  for y := 1 to 64 do  
    begin  
      pixelfarben[x,y] := bit.canvas.pixels[x,y];  
    end;  
  end;  
bit.free;  
end;
```

Was ich da gemacht habe ist an sich recht schnell erklärt: Zunächst habe ich die Bitmap geladen und habe dann Pixel für Pixel die Farbe des Pixels in ein Array kopiert, welches ich zuvor so definiert habe:

```
var  
Form1: TForm1;  
xpos : double;  
zpos : double;  
rotation : double;  
fps : integer;  
timefactor : double;  
mauspos : integer;  
pixelfarben : array[0..64, 0..64] of integer;
```

Am Ende habe ich dann noch die Bitmap freigegeben, um nicht noch mehr Speicher zu verbraten (das Tut diese Anwendung eh schon...)

PS: Falls sich noch wer wundern sollte, weshalb das Array von 0 bis 64 geht:

Erst hatte ich ne Definition von 1 bis 64 angegeben, musste dieses aber abändern, das es ansonsten später mal Speichererrors gegeben hat...

Nun müssen wir aus diesen Infos aber auch noch was darstellen, was wir Schritt für Schritt angehen werden.

Zunächst einmal möchte ich ne neue Prozedur anlegen, welche das Zeichnen beinhalten soll:

```
procedure drawmap;  
var  
x,y : integer;  
begin  
  
end;
```

Diese Prozedur muss nun ja auch noch aufgerufen werden. Dazu bastelt wir in unserer Render-Prozedur dieses ein:

```
...  
glrotate(rotation,0,1,0);  
gltranslate(xpos, 0, zpos);  
  
drawmap;  
  
SwapBuffers(form1.myDC);           //scene ausgeben  
end;
```

Ein gut gemeinter Rat: Schreibt bitte die neue Prozedur über die Render-Prozedur. Ansonsten erkennt Delphi nämlich nicht, dass es eine solche Prozedur gibt und gibt ne nette kleine Fehlermeldung aus ;-)

Nun muss die neue Prozedur nur noch mit Code gefüttert werden. Ich habe das nun teilweise schon gemacht:

```
procedure drawmap;  
var  
x,y : integer;  
begin  
for x := 1 to 64 do    //Alle Felder in X und Y-Richtung durchlaufen  
begin  
  for y := 1 to 64 do  
  begin  
    if not (pixelfarben[x,y] = clblack) then  
    begin           //Ist das Feld NICHT Schwarz? Ok, dann zeichnen  
      glColor3f(1,1,1);  
      glDepthMask(GL_FALSE);           //Tiefeninfos abschalten  
      glBindTexture(GL_TEXTURE_2D, bodentextur); //Der Fußboden  
      glBegin(gl_quads);           //wir zeichnen den Fußboden  
      gltexcoord2f(0,0);  
      glVertex3f(x*4 , -1, y*4);  
      gltexcoord2f(1,0);  
      glVertex3f(x*4+4, -1, y*4);  
      gltexcoord2f(1,1);  
      glVertex3f(x*4+4, -1, y*4+4);  
      gltexcoord2f(0,1);  
      glVertex3f(x*4 , -1, y*4+4);  
    end  
  end  
end  
end
```

```

glend;
glDepthMask(GL_True);    //Tiefeninfos anschalten
end;
end;
end;
end;


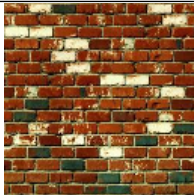

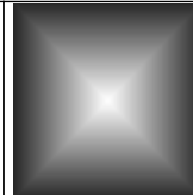
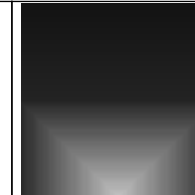

```

Ich denke mal, dass sich das meiste von selber erklärt. Und auch wenn nicht, ist es nicht allzu schwer: wir gehen hier jedes Feld unseres Arrays durch (die beiden Schleifen) und prüfen, ob die eingetragene Farbe schwarz ist. Ist sie es nicht, dann wird an die Stelle ein Quad von einer 4x4 Größe gezeichnet. Eine Sache muss hier nun aber wahrscheinlich doch noch erklären: den „glDepthmask(gl_false)“ – Befehl.

Also: ihr habt ja bereits gelernt, dass alle Objekte sich einem Tiefentest stellen müssen. In dem sogenannten Tiefenpuffer stehen alle Tiefenwerte anderer Objekte, mit denen verglichen wird, ob das zu zeichnende Objekt evtl. verdeckt werden könnte. Der „glDepthmask(gl_false)“ – Befehl bewirkt nun, dass sie ein Objekt zwar an den Tiefenpuffer hält, wenn es verdeckt wird, so wie es auch normal ist, aber selber keine Tiefeninfos in den Puffer schreibt, was sehr wichtig ist, da wir gleich noch Blenden müssen und da die Tiefeninfos nicht brauchen können. Einige werden sich Fragen: „Ja, aber woher wissen denn dann die Objekte weit hinten, dass sie verdeckt werden, denn die Vorderen keine Tiefeninfos schreiben.“ An sich haben die recht, und es würde ein echtes Durcheinander geben, wenn es keine Infos gebe... aber da gibt es Abhilfe: sobald das Licht, oder was auch immer zu blendende da ist, gammelt es nicht mehr wirklich, ob Tiefeninfos da sind, oder nicht. Deshalb lassen wir nachher einfach die Tiefeninfos durch unsere Lichtsimulation setzen... ganz einfach also.

Aber zurück zum Code:

An sich ist der schon genug erklärt. Was ich jedoch bislang verschwiegen habe, sind die benötigten Texturen. Wie im 7. Teil des Kurses brauchen wir 6 Texturen, bloß das die etwas anders aussehen (minimal):

					
Bodentextur	Wandtextur	Deckentextur	Bodenlicht	Wandlicht	Deckenlicht

Wie man die Texturen lädt, brauche ich euch doch nicht wirklich nicht mehr erzählen. Um es euch einfacher zu machen habe ich hier schon mal die Namen der Texturen beigeschrieben. (Ich werde sie im Verlaufe dieses Tuts nur noch so betiteln, wie hier angegeben)

@ alle Nasen, die immer noch nicht wissen, wie man Texturen lädt bzw. was Blending ist: schaut mal in die Teile 5 & 6... da findet ihr Antworten.

So, wo waren wir? Ach ja: bei der kleinen Szene oben. Na ja: wenn ihr nun mal das Programm startet, dann werdet ihr erst mal überhaupt nichts sehen! Nanu? Es wird doch was gerendert! Ja, aber ich Dussel habe etwas vergessen: die Quadrate werden ja in Richtung der Positiven Z-Ache gezeichnet, was bedeutet, dass das ganze exakt hinter unserer Kamera ist. Zugleich ist, wenn ich mal etwas dreht, das ganze etwas Versetzt zu uns, da wir am Punkt 0,0,0 stehen, aber die Szene erst bei 4,0,4 anfängt. Was tun wir dagegen? Na ja: wir verändern unsere Variablen-Angaben am Anfang ein wenig:

```

var
  Form1: TForm1;
  xpos : double = -6;

```



```
zpos : double = -6;  
rotation : double = 180;
```

So, nun solltet ihr mitten auf einer Ecke starten und alles überblicken können.
Nun wollen wir mal sehen, wie wir den Boden beleuchten könnten.

Wie oben irgendwann einmal gesagt (ich schreibe seit über 6 Wochen an diesem Tut), wollen wir die Pixelfarben als Licht-Farben verwenden bzw. bei der Farbe (128, 188, 128) nicht beleuchten.

Dazu brauchen wir zunächst mal eine Prozedur, die uns die normalen Farbwerte in OGL – Farbwerte umrechnet:

```
procedure getoglColor(color : TColor; var r,g,b:double);  
begin  
  R := getrvalue(colorToRgb(color)) / 255;  
  G := getgvalue(colorToRgb(color)) / 255;  
  B := getbvalue(colorToRgb(color)) / 255;  
end;
```

Ich denke, dass ich da nicht wirklich drauf eingehen muss. Schreibt aber bitte diese Prozedur über die, wo wir die Szene zeichnen, es seihe denn, ihr mögt die Compiler – Fehlermeldungen *g*

Wie ihr seht, braucht die Funk genau 4 Parameter. Das eine ist die Input – Color und die anderen drei sind die fertigen OpenGL – Farbwerte.

So... das wollen wir nun auch nutzen und basteln damit an der Drawmap - Prozedur ein wenig weiter:

```
procedure drawmap;  
var  
x,y : integer;  
r,g,b : double;  
begin  
  for x := 1 to 64 do    //Alle Felder in X und Y-Richtung durchlaufen  
  begin  
    for y := 1 to 64 do  
    begin  
      if not (pixelfarben[x,y] = clblack) then  
      begin    //Ist das Feld NICHT Schwarz? Ok, dann zeichnen  
        glColor3f(1,1,1);  
        glDepthMask(GL_FALSE);    //Tiefeninfos abschalten  
        glBindTexture(GL_TEXTURE_2D, bodentextur); //Der Fußboden  
        glBegin(gl_quads);    //wir zeichnen den Fußboden  
        glTexCoord2f(0,0);  
        glVertex3f(x*4, -1,y*4);  
        glTexCoord2f(1,0);  
        glVertex3f(x*4+4,-1,y*4);  
        glTexCoord2f(1,1);  
        glVertex3f(x*4+4,-1,y*4+4);  
        glTexCoord2f(0,1);  
        glVertex3f(x*4, -1,y*4+4);  
      glend;
```

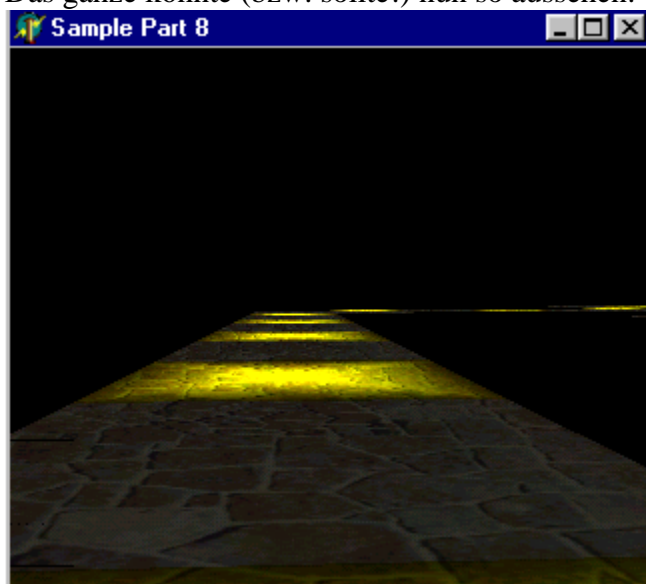
```

glDepthMask(GL_True);      //Tiefeninfos anschalten

if pixelfarben[x,y] = strtoint('$808080') then //Beleuchten oder Verdunkeln?
begin //verdunkeln
glcolor3f(0.5,0.5,0.5);
glBindTexture(GL_TEXTURE_2D, deckenlicht); //Wir missbrauchen das Deckenlicht
end
else
begin //beleuchten
getoglcolor(pixelfarben[x,y],r,g,b);
glcolor3f(r,g,b);
glBindTexture(GL_TEXTURE_2D, bodenlicht);
end;
glEnable(gl_BLEND);          //Blending an
glBlendFunc(gl_src_color,gl_src_color); //Blending-Modus
glbegin(gl_quads);           //wir beleuchten den Fußboden
gltexcoord2f(0,0);
glvertex3f(x*4 ,-1,y*4);
gltexcoord2f(1,0);
glvertex3f(x*4+4,-1,y*4);
gltexcoord2f(1,1);
glvertex3f(x*4+4,-1,y*4+4);
gltexcoord2f(0,1);
glvertex3f(x*4 ,-1,y*4+4);
glend;
glDisable(gl_BLEND);        //Blending aus
end;
end;
end;
end;

```

So... an sich sollte es für alle, die bis hierher gelesen haben einigermaßen verständlich sein. An sich prüfe ich ja nur, ob beleuchtet oder verdunkelt werden muss, setze die nötige Farbe und Textur und zu guter Letzt wird einmal fröhlich geblendet. Das ganze könnte (bzw. sollte!) nun so aussehen:



(habe mich vom Startpunkt aus genau um 90° nach links gedreht)

So... nun müssen wir das ganze Prozedere auch noch für die Seitenwände und die Decke machen. Da es aber viel zuviel Platz brauchen würde, alles einzeln zu erklären, poste ich hier einfach ne kurze Anleitung, wie es gemacht werden sollte:

- 1) Nur für die Seitenwände! Test, ob eine Seitenwand überhaupt nötig ist!
- 2) Zeichnen der Rohwand bzw. Decke
- 3) Test, ob beleuchtet werden muss (entfällt bei der Decke!) und ggf. Farbe und Textur setzen
- 4) Zeichnen bzw. Blenden der Beleuchtung

Für alle, die näheres wissen wollen oder nicht ganz damit klarkommen: guckt bitte in das Sample zu diesem Kurs. Zunächst wollte ich den ganzen Code hier reinschreiben, aber dann wären wir hier nun schon auf Seite 16...

Nachwort

So, das war's denn leider auch schon wieder. Sorry an alle, die so lange auf dieses Tut warten mussten, aber ich hatte einfach sehr wenig Zeit die letzten Tage. Auf jeden Fall: schmeißt das hier beschriebene Programm nicht weg! Ich überlege nämlich momentan, ob ich aus der Reihe ein kleines Tut darüber bringen sollte, wie man dieses Sample noch so erweitern könnte, bzw. die Technik mit den BMPs als Mapgrundlage noch mehr ausnützen könnte. Als nächstes im Open Gl - Kurs habt ihr entweder mit Partikeln und evtl. Spiegelung derer zu tun, oder ich schreibe ein wenig über das Thema „selection“.

Ach ja, was ich noch sagen wollte: während der Zeit zwischen Teil 7 und 8 kamen bei mir diverse Mail betreffend dieses Kurses an. Einer dieser Mails war auch ein sehr schicker Anhang mit einem durch die Gegend fliegenden, durch „Blending“ halb transparent gemachten Würfel angehängt. Dieses brachte mich auf eine Idee: was würdet ihr von einer kleinen Datenbank mit Projekten von Lesern halten, zu der alle Zugriff haben. Ich denke besonders für Anfänger ist es immer wieder hilfreich einige Sources von anderen sich anzusehen und für erfahreneren ist sicher auch was dabei, da diese gucken könnten, wie z.b. ein anderer einen bestimmten Effekt realisiert hat. Schreibt mir ne Mail oder postet ins Forum und teil mir mit, was ihr von der Idee halten würdet.

Bis Teil 9!

Mr_T