# Vorstellung und Vergleich zweier Methoden zur Berechnung statischer Beleuchtung in virtuellen Welten am Beispiel von Außenlandschaften

An der Katholischen Theresienschule Berlin

als

**5. Prüfungskomponente des Abiturs**

**in Form der**

**besonderen Lernleistung**

eingereicht

von Benjamin Bisping

Berlin, den 14.12.2007

# Inhalt

Inhalt....2

Einleitung....3

1 Einführung....4
1.1 Das Problem....4
1.2 Die Vorgehensweise....4

2 Die Umgebung....5
2.1 Das Programm....5
2.2 Die Welt....5
2.3. Die Lightmap....6

3 Methode A: Raytracing....7
3.1 Die Grundidee....7
3.2. Die Vorgehensweise....7
3.3 Die Implementierung....8
3.3.1 Vorbereitungen....8
3.3.2 Die Lumel-Schleife....8
3.3.3 Die SendRay()-Funktion....9
3.3.4 SendRay() und Meshes....11
3.3.4.1 Der Quadtree....12
3.3.4.2 Die SendRay-Erweiterung....12
3.3.5 Himmelslicht....13
3.4 Die Vor- und Nachteile....13

4 Methode B: Texelmapping....14
4.1 Die Grundidee....14
4.2 Die Vorgehensweise....15
4.3 Die Implementierung....15
4.3.1 Vorbereitungen....15
4.3.2 Die Beleuchtung....16
4.3.2.1 Das Licht einrichten....16
4.3.2.2 Die Szene rastern....16
4.3.2.3 Auswertung....17
4.3.2.4 Anwendung....18
4.3.3 Aufräumen....18
4.4 Die Vor- und Nachteile....19

5 Vergleich und Fazit....20
5.1 Vergleich....20
5.2 Fazit....21
5.3 Weitere Entwicklung....21

Literaturverzeichnis....22

Selbstständigkeitserklärung....23

Glossar....24

Anhang A: Das Szenen-Format....25

Anhang B: Der Quelltext....26

# Einleitung

Schon immer haben mich Licht und Schatten beeindruckt. Bilder leben von ihnen und ich lebe von Bildern. Früh machte ich sie zu einem Element meiner Zeichnungen. Seit ich Spiele programmiere[1], bin ich auch dort bemüht, Schatten sowohl ästhetisch ansprechend als auch physikalisch korrekt umzusetzen. Jenseits der Spieleprogrammierung habe ich mich auch häufig mit *Raytracing* beschäftigt, einer Methode zur realistischen, aber nur begrenzt Echtzeit-tauglichen Lichtsimulation[2].

Obwohl seit Jahren viele das Ende der *Lightmap*-gestützten statischen Beleuchtung prophezeien[3], bleibt *Lightmaps* besonders bei Hobbyprogrammierern wie mir, die Anwendungen für Low-End-Systeme entwickeln, eine wichtige Technik zur performanten und realistischen Darstellung von Beleuchtung in Echtzeitanwendungen.

Im Folgenden möchte ich zunächst auf die grundlegende Idee bei Lichtberechnung eingehen, dann am Beispiel von *Terrainlightmaps* zwei unterschiedliche Verfahren der *Lightmap-G*enerierung vorstellen und abschließend vergleichen. Die beschriebenen Algorithmen lassen sich mit einigen Anpassungen auch jenseits von *Terrains* und *Lightmaps* auf anderen Anwendungsgebieten nutzen.

Am Ende der Arbeit findet sich ein Glossar. Dort erklärte Wörter habe ich im Text durch Kursivdruck kenntlich gemacht.

Fragen zur Arbeit beantworte ich gerne unter bll@mrkeks.net.

1 Liste meiner Projekte: http://www.mrkeks.net/?show=produzieren

2 Absatz „Einsatzgebiete“ im Artikel „Raytracing“. In: Wikipedia, Die freie Enzyklopädie. Bearbeitungsstand: 14. November 2007, 21:37 UTC. URL: http://de.wikipedia.org/w/index.php?title=Raytracing&oldid=38983303 [2007-12-08]

3 Absatz „Zukunft“ im Artikel „Lightmaps“. In: DGL-Wiki. Bearbeitungstand: 22. April 2006, 18:12. URL: http://wiki.delphigl.com/index.php/Lightmaps#Zukunft [2007-12-12]

# 1 Einführung

## 1.1 Das Problem

Das Hauptaugenmerk dieser Arbeit soll sein, zu ermitteln, wo sich auf einem *Terrain* Schatten finden, um mit diesen Informationen eine so genannte *Lightmap* zu generieren. Schatten sind bekanntlich an den Stellen, zu denen kein Licht gelangt. Und Licht erreicht einen Ort nicht, wenn sich zwischen der Lichtquelle und dem betrachteten Punkt ein Objekt befindet, das den Lichtweg unterbricht. Ziel jeder *Lightmapper*-Entwicklung muss es also sein, schnelle Algorithmen zu schreiben, die angesichts der Szenengeometrie verschiedene Lichtausbreitungswege finden oder verwerfen. Diese Arbeit soll an zwei Beispielen aufzeigen, welche Möglichkeiten es gibt, wie sie zu implementieren sind und für welche Fälle sie sich eignen. Das erste Beispiel verwendet *Raytracing* und folgt den gängigen Verfahren. Das zweite ist etwas unkonventioneller, indem es die Sichtbarkeitsprüfung größtenteils der Grafikkarte überlässt.

## 1.2 Die Vorgehensweise

Für das Testen der zwei vorzustellenden Algorithmen benutze ich eine kleine, rudimentär umgesetzte Umgebung, die das Laden und Speichern von in Dateien definierten virtuellen Welten ermöglicht (siehe Anhang). Diese Umgebung wird zunächst kurz beschrieben. Danach werden die zwei Lichtberechnungstechniken vorgestellt, indem wir zuerst die Grundidee betrachten, dann die Vorgehensweise erarbeiten und ausführlich die Implementierung erläutern. Diese erfolgt in BlitzMax[4]; für den Zugriff auf 3D-Beschleuniger-Hardware kommt *OpenGL*[5] zum Einsatz. Nach der Implementierung werden wir die Vor- und Nachteile sowie die Erweiterungsmöglichkeiten der jeweiligen Technik betrachten. Den Abschluss macht ein Vergleich der beiden Methoden und eine Einordnung, für welche Anwendungsfälle welche Umsetzung besser geeignet ist.

4 Ein schön zu lesender OOP-fähiger BASIC-Dialekt, offizielle Website: http://www.blitzbasic.com/Products/blitzmax.php [2007-12-09]; eine Demoversion der IDE inklusive Dokumentation und Compiler befindet sich auf der dieser Arbeit beigelegten CD (siehe Anhang B).

5 Offizielle Website: http://www.opengl.org [2007-12-09] Siehe auch Glossar und Literaturverzeichnis.

# 2 Die Umgebung

## 2.1 Das Programm

Da Licht und Schatten nicht im leeren Raum entstehen, sondern Informationen über die Lichtquellen und Objekte benötigt werden, brauchen wir zunächst eine Umgebung. Diese stellt das kleine, für diese Arbeit geschriebene „Terrainlightmapping-Testprogramm" zur Verfügung (siehe Anhang B). Die Fähigkeiten dieses Programms beschränken sich auf das Laden und Speichern (zum Format siehe Anhang A), Generieren, *Lightmap*pen und Anzeigen von *Terrains*. Das meiste ist nur grundlegend, aber gut erweiterbar umgesetzt.
Das Hauptprogramm hat drei große Unterobjekte (siehe auch Abbildung 2 auf Folgeseite):

- `interface:TInterface` erzeugt und verwaltet die graphische Benutzeroberfläche.
- `display:TDisplay` zeigt die Welt aus der Sicht von `camera:TCamera` in einem Fenster an.
- `world:TWorld` handhabt die Informationen der Welt, besonders die *Terrain*informationen in `terrain:TTerrain` und die Liste aller Entities (Lichter und *Meshes*).

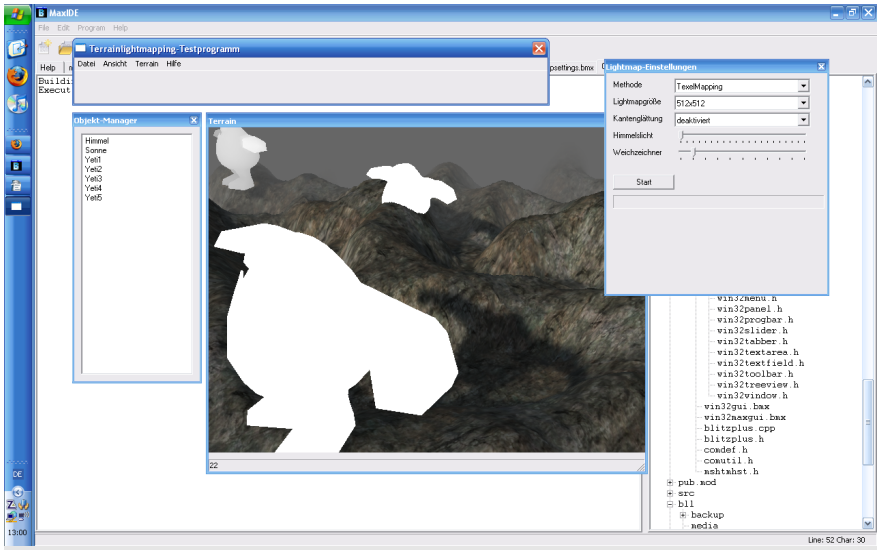


*Abbildung 1: Screenshot aus "Terrainlightmapping-Testprogramm"*

## 2.2 Die Welt

Ein `TTerrain` ist eine Zusammenstellung aus einer *Heightmap* mit Höheninformationen, einer Material*textur* und einer *Lightmaptextur* zur statischen Beleuchtung. Ferner enthält es eine *OpenGL*-Displaylist zur Anzeige mittels `TDisplay`.

Ein `TEntity` ist ein in der Welt positioniertes Objekt. Das kann eine Lichtquelle, eine Kamera oder auch ein *Mesh* (das heißt ein aus Polygonen zusammengesetztes Objekt) sein.

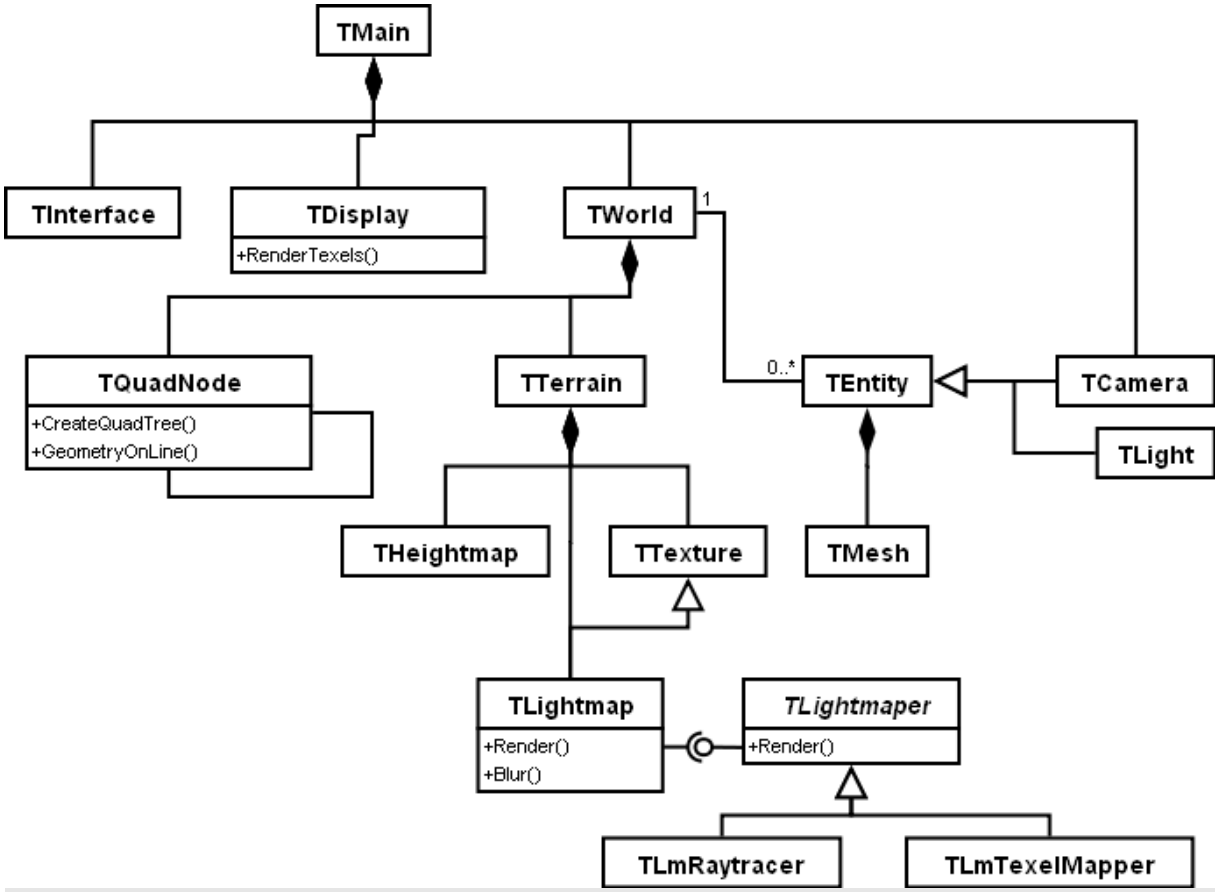


*Abbildung 2: UML-Diagramm der relevanten Programmelemente*

Wie sich aus dem Diagramm (Abbildung 2) erkennen lässt, befindet sich alles für diese Arbeit relevante in `TWorld` und in davon abhängigen Klassen. Alle Quelltextausschnitte auf den nächsten Seiten stammen - bis auf eine Ausnahme - aus `TLmRaytracer` und `TLmTexelMapper`.

# 2.3. Die Lightmap

Die auf dem Terrain liegende *Lightmap* ist eine zweidimensionale Map von *Lumeln.* Sie enthält die Beleuchtung des *Terrains* an verschiedenen Stellen, gespeichert in einer BlitzMax-*Pixmap*. Die *Lightmap* ist eine erweiterte Textur, sodass die *Lumel* zugleich *Texel* sind, die bei der Darstellung mittels *Texture*mapping über das Terrain gezogen und mit der Materialfarbe multipliziert werden. *Lightmaps* verfügen über eine *Render*methode. Diese skaliert die Textur, ruft einen der beiden auf den nächsten Seiten beschriebenen Algorithmen zur Beleuchtung auf und wendet zum Schluss einen Weichzeichner[6] an, um weiche Schatten zu erzeugen.

6 Fischer, Robert; Perkins, Simon; Walker, Ashley; Wolfahrt, Erik (2003): Hyper Media Image Processing Reference, Gaussian Smoothing. URL: http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm [2007-12-10]

# 3 Methode A: Raytracing

## 3.1 Die Grundidee

Ein verbreitetes Konzept zur Beleuchtungsberechnung ist das *Raytracing*, sprich das Verfolgen von Strahlen im 3D-Raum. Das soll auch gleich unser erster Ansatz sein: Um zu ermitteln, ob eine Lichtquelle einen *Lumel* beleuchtet oder nicht, senden wir vom Punkt, an dem sich der *Lumel* im 3D-Raum befindet, einen Strahl in Richtung der Lichtquelle. Erreicht er diese, so wird der Punkt beleuchtet. Erreicht er sie nicht, so liegt der Punkt im Dunkeln. In der Realität ist es natürlich so, dass sich Strahlen von der Lichtquelle aus ausbreiten; doch bietet es sich beim *Raytracing* im Allgemeinen an, die Umkehrbarkeit des Lichtweges auszunutzen: Wir interessieren uns nicht für jeden möglichen Weg, sondern nur für die Wege zwischen *Lumel* und Lichtquelle.
Mathematikern wird auffallen, dass aus den Lichtstrahlen somit Lichtstrecken werden. Daran wollen wir uns aber nicht stören, denn endlicher Berechnungsaufwand kommt uns entgegen. Wir werden die Verbindungen trotzdem weiterhin *Rays* nennen, um in der Terminologie zu bleiben.

## 3.2. Die Vorgehensweise

Der *Lightmapper* muss prinzipiell nur für jeden *Lumel* der *Lightmap* alle Lichtquellen durchlaufen und diese auf Sichtbarkeit prüfen. Dazu werden wir zunächst alle Lichtquellen in einem Array listen und die Umgebungslichtfarbe speichern. Danach arbeiten wir die *Lumel* ab. Jeder *Lumel* wird zuerst auf die Umgebungshelligkeit gestellt. Darauf wird ein *Ray* zu jeder Lichtquelle gesandt (in unserem Beispiel wollen wir nur Richtungslicht verwenden) und bei Erfolg der *Lumel* um die Lichtfarbe aufgehellt. Zuletzt wird die *Lumel*farbe in die *Pixmap* der *Lightmap* gespeichert.

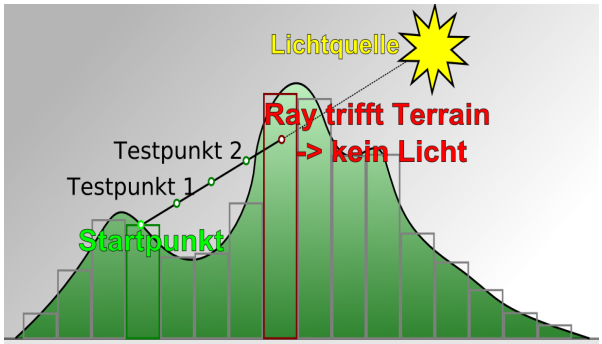

*Abbildung 3: Skizze vom Prinzip der Beleuchtungsprüfung*

Die Überprüfung, ob der Weg zwischen *Lumel* und Licht frei ist (siehe Abbildung 3), übernimmt die Funktion `SendRay()`. Sie verfolgt den Strahlenverlauf zwischen den beiden und prüft, ob sich *Terrain*hügel auf dem Weg befinden. Später werden wir auch noch eine Routine zur Überprüfung auf behindernde *Mesh*-Dreiecke einbauen und das Beleuchtungsmodell des Umgebungslichts weiterentwickeln.

# 3.3 Die Implementierung

Der vollständige Quelltext findet sich in „lmraytracer.bmx“ (siehe Anhang B).

### 3.3.1 Vorbereitungen

Zuerst sammeln wir alle Lichter der betrachteten virtuellen Welt in einem Array und ermitteln die Farbe des Umgebungslichts.

```
For ent = EachIn world.entities      ' Gehe jedes Entitiy durch,
     If ent.typ = TEntity.LIGHT      ' wenn es ein Licht ist,
          Select TLight(ent).ltyp    ' von welchem Typ ist es?
          Case TLight.AMBIENT        ' Falls es Umgebungslicht ist,
               ambr = TLight(ent).r  ' dann merke seine Farbe fürs Umgebungslicht.
               ambg = TLight(ent).g
               ambb = TLight(ent).b
          Default                    ' Im Normalfall:
               lights = lights + [TLight(ent)] 'Füge es einfach zur Lichtliste hinzu
          End Select
     EndIf
Next
```

THeightmap stellt zwei besonders aufgelöste Versionen der *Heightmap* zur Verfügung: Eine auf 1024x1024 Elemente hochgerechnete und eine auf 256x256 heruntergerechnete Version. Die erste ist sinnvoll, da das *Terrain* auch zwischen den einzelnen Höhenpunkten eine bestimmte Höhe hat und bei jedem Zugriff von Neuem linear zu interpolieren massive Rechenzeitverschwendung wäre. Die zweite enthält die Höchstwerte für jeweils 4x4 Werte der ersten. Welche Optimierung das ermöglicht, wird sich später bei der *Ray*funktion herausstellen.

Unabhängig von *Heightmap*- und *Lightmap*auflösung werden wir zur Einfachheit innerhalb der Funktion immer so tun, als wäre das Terrain 256x256 Längeneinheiten im Raum groß. Da wir den Umrechnungsfaktor der *Lightmap*koordinaten in diesen 256er-Zahlenraum nur einmal berechnen wollen, speichern wir ihn in einer Variable. pm ist die *Pixmap* der *Lightmap.*

```
fact = Float(256)/pm.width
```

### 3.3.2 Die Lumel-Schleife

Nun sind wir bereit, jeden *Lumel* der *Lightmap* abzuarbeiten und seine Beleuchtung zu berechnen. Als erstes muss dabei die Koordinate des betrachteten *Terrain*-Punktes im Raum herausgefunden werden, wobei das *Terrain* wie gesagt auf 256x256 Einheiten skaliert wird.

```
For lx = 0 To pm.width-1 ' Gehe jeden einzelnen Lumel durch:
     x = lx*fact            ' Ermittle seine X-Koordinate im 256x256er-Raum
     For lz = 0 To pm.height-1

          z = lz*fact       ' sowie Z-Koordinate im 256x256er-Raum.
          y = ter.heightmap.datahr[Int(x*4),Int(z*4)] ' y-Koordinate des betrachteten
                                ' Terrainpunktes aus der Heightmap holen.
```

Die Umgebungslichtfarbe, die selbst im Schatten liegende Punkte haben, können wir dem *Lumel* schon einmal getrost zusprechen. Im Abschnitt „3.3.5 Himmelslicht“ werden wir das verfeinern. `r`,`g` und `b` sind übrigens die Rot-, Grün- und Blau-Komponente der *Lumel*farbe.

```
        r = ambr
        g = ambg
        b = ambb
```

Jetzt gehen wir für den betrachteten Punkt alle Lichtquellen durch, überprüfen mittels `SendRay()`, ob diese ihn „sehen können“, also beleuchten. Gegebenenfalls addieren wir die Lichtfarbe zur *Lumel*farbe. Was bei `SendRay()` genau passiert, ist Thema des nächsten Abschnitts. Für den Moment genügt uns, dass es `True` zurückgibt, wenn freie Sicht auf die Lichtquelle besteht.

```
        For i = 0 To lights.length-1      ' Gehe jede mögliche Lichtquelle durch,
            Select lights[i].ltyp         ' und sende je nach Lichttyp...
            Case TLight.DIRECTIONAL
                If SendRay(x,y,z, -lights[i].nx, -lights[i].ny, -lights[i].nz, 50) Then
                    r :+ lights[i].r ' ...einen Ray in entsprechender Richtung
                    g :+ lights[i].g ' und addiere die Beleuchtung bei Erfolg.
                    b :+ lights[i].b
                EndIf
            Case TLight.POINT ' Mit Punktlichtern wollen wir uns hier nicht weiter
                          ' beschäftigen. Sie funktionieren aber ähnlich.
            End Select
        Next
```

Übersteuerte Farbkomponenten müssen auf 255 begrenzt werden, da sonst die Bytes, in denen sie gespeichert werden, überlaufen und psychedelische Muster entstehen.

```
        If r>255 Then r=255
        If g>255 Then g=255
        If b>255 Then b=255
```

Nach getaner Arbeit können wir dann endlich auch den *Lumel* in die *Lightmap* schreiben.

```
        pm.WritePixel(lx,lz,(r Shl 16) | (g Shl 8) | b)
    Next
Next
```

### 3.3.3 Die SendRay()-Funktion

Der komplizierte, rechenintensive und schwierig zu optimierende Teil steht jedoch erst noch an: nämlich die Funktion, die überprüft, ob etwas den Lichtweg blockiert. Mathematisch betrachtet, haben wir die Funktionen *g(i)* für die Lichtstrecke und *h(x;z)* für die *Heightmap.*

$$I.\begin{pmatrix} x \\ y_1 \\ z \end{pmatrix} = g(i) \qquad II.\ y_2 = h(x;z)$$

Wir wollen wissen, ob es einen Punkt gibt, an dem der Lichtstrahl abgehalten wird, also unter das Terrain geht. Wir suchen demnach ein *i*, für das das Gleichungssystem ergibt:

$$y_1 \leq y_2$$

Anders formuliert ist das dort der Fall, wo der Ausdruck zutrifft: $g_y(i) < h(g_x(i); g_z(i))$
Da *h* ein Array ist, können wir es nicht weiter aufschlüsseln. *g* hingegen ist eine Gerade, die durch einen Stütz- und einen normierten Richtungsvektor bestimmt wird.

$$g(i) = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + i * \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}$$

Die Vektorgleichung des Strahls lautet in Komponentengleichungen zerlegt:

$$g_x(i) = x_0 + x_1 * i \qquad g_y(i) = y_0 + y_1 * i \qquad g_z(i) = z_0 + z_1 * i$$

Mit $D = \{0 \ldots l\}$ wird daraus eine Strecke der Länge *l*. Da wir nur eine endliche Zahl von Überprüfungen durchführen können, darf die Funktion auch nur für eine endliche Anzahl von Zahlen definiert sein: $i \in \mathbb{R} | 0 < i = n * s \leq l$ , wobei *n* eine ganze positive Zahl und *s* die Schrittweite ist.

Wir suchen nun ein `i`, an dem gilt: $y_0 + y_1 * i < h(x_0 + x_1 * i, z_0 + z_1 * i)$ Trifft dies für ein `i` zu, so schneidet der Lichtstrahl das *Terrain* und wir können `False` zurückgeben.

Das schreit nach einer `For`-Schleife. Allerdings sind zwei Kniffe ratsam:
- Addition statt Multiplikation: Anstatt jedes Mal `x0 + x1 * i` zu berechnen, setzen wir die Schrittweite `s` einfach auf 1 und erhöhen `x0` schlicht bei jedem Durchlauf um `x1`. So sparen wir uns die Multiplikation. Man könnte alternativ auch auf Bresenham[7] zurückgreifen.
- Von Grob nach Genau: Es bringt einen massiven Geschwindigkeitsschub, erstmal mit weniger Schritten auf einer gröber gerasterten Version der *Heightmap* herauszufinden, ob sich eine genauere Überprüfung überhaupt lohnt. Hierfür wird die oben bereits erwähnte 256x256er Version der *Heightmap* benötigt, die für jeden Quadranten des Raums den Maximalwert der *Heightmap* beinhaltet. Nur wenn ein *Ray* diese niedrig aufgelöste *Heightmap* trifft, ist die Prüfung nötig, ob er auch die hochauflösende Version schneidet.

Die Eingaben für unsere Funktion sind das *Terrain* und die Welt, durch die der *Ray* sich bewegt, der Anfangspunkt (`x0,y0,z0`) und der Vektor eines Schrittes (`x1,y1,z1`), sowie die Anzahl der für die Gesamtstrecke zu machenden Schritte (`steps`). Des Weiteren werden für die Funktion zwei Schleifenvariablen `i` und `i2` benötigt.

```
Function SendRay:Int ( ter:TTerrain, world:TWorld, x0:Float,y0:Float,z0:Float,..
                                                   x1:Float,y1:Float,z1:Float, steps:Int)
    Local i:Int, i2:Int ' Schleifenvariablen
```

7 Bresenham, Jack (1965): „Algorithm for computer control of a digital plotter“. URL: http://www.research.ibm.com/journal/sj/041/ibmsjIVRIC.pdf [2007-12-12]

Wir gehen die Lichtstrecke Punkt für Punkt durch.

```
    For i = 1 To steps ' Lichtstrecke durchgehen,
        ' Betrachtungspunkt linear fortbewegen:
        x0 :+ x1
        y0 :+ y1
        z0 :+ z1
```

Wenn der *Terrain*rand erreicht wird, brechen wir ab:

```
        If x0<1 Or z0<1 Or x0>254 Or z0>254 Then Return True
```

Es wird zuerst in der 256er-Auflösung überprüft, ob etwas getroffen wird.

```
        If ter.heightmap.datamr[Int(x0),Int(z0)] > y0 Then
```

Ist das der Fall, so werden fünf kleinere Schritte durch den Quadranten durchgeführt, um detaillierter zu erfahren, ob wirklich Treffer vorliegen. Wenn ja, dann wird gleich `False` zurückgeliefert, andernfalls am Ende `True`.

```
            For i2 = -2 To 2
                If ter.heightmap.datahr[Int(x0*4-x1*i2),Int(z0*4-z1*i2)] > y0-y1*i2/4 Then
                    Return False ' Der Ray trifft das Terrain.
                EndIf
            Next
        EndIf
    Next
    Return True ' Wenn nichts getroffen wurde, liefere True zurück.
End Function
```

### 3.3.4 SendRay() und Meshes

Bis jetzt haben wir eine recht schlanke, simple und akzeptabel schnelle Funktion realisiert, die in Sekunden ganze *Terrains* beleuchten kann. In Landschaften, wie sie in der Realität und in Spielen vorkommen, gibt es jedoch zahlreiche Objekte, die ebenfalls Schatten werfen. Diese komplexen Objekte werden in der 3D-Grafik zumeist mit Dreiecken beschrieben. Solche Objekte werden *Meshes* genannt. Darum benötigen wir weiterhin eine Funktion, die überprüft, ob der *Ray* Dreiecke von *Meshes* trifft.

Da mit hunderten *Meshes* zu rechnen ist, die allesamt aus tausenden von Dreiecken bestehen, und es sehr rechenintensiv wäre, bei jedem Strahl auf Schnitt mit jedem einzelnen Dreieck der Welt zu prüfen, brauchen wir eine Struktur, um die Menge der zu überprüfenden Dreiecke einzuschränken. Dazu eignet es sich, die Geometrieinformationen der Welt in Sektoren einzuteilen. Manche *Raytracer* verwenden für solche Probleme *Voxel*. Angesichts der Größe eines Außenlevels ist das für uns keine Option. Für große, in sich sehr unterschiedlich detaillierte Welten verwendet man meist *Octtrees*. Diese selbst werden dort detailreicher in ihrer Unterteilung, wo sie viele Details enthalten. In unserer sich größtenteils zweidimensional erstreckenden Landschaft, reicht der Gebrauch eines *Quadtrees*.

### *3.3.4.1 Der Quadtree*

Ein *Quadtree* ist eine Baumstruktur, bei der jeder Ast einen Sektor der 3D-Welt darstellt und, wenn er kein Endast ist, vier neue Äste hervorbringt. Diese vier Äste unterteilen den vorigen Sektor in vier kleinere gleichgroße Sektoren auf einer Ebene. Endäste enthalten Geometrie-Informationen.

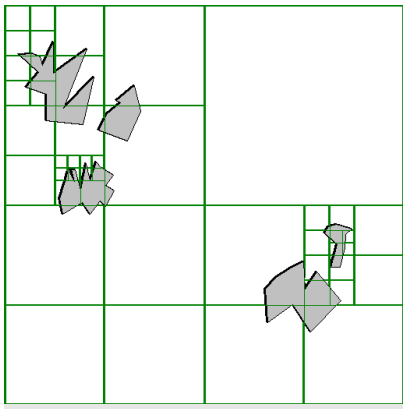

*Abbildung 4: Ein Quadtree unterteilt den Raum rekursiv nach der Informationsdichte*

```
Type TQuadNode
     ' Erstellt einen Baum rekursiv
     Function CreateQuadTree:TQuadNode(tris:Float[],..
                              space:Float, x:Float, z:Float)

     ' Liefert eine Liste von TQuadNodes zurück, die von einer
       Strecke geschnitten werden
     Method GeometryOnLine:TQuadNode[](x0:Float,y0:Float,..
                         z0:Float, x1:Float,y1:Float,z1:Float)
End Type
```

Wie genau ich die Funktionen implementiert habe, lässt sich in „`quadtree.bmx`" nachlesen. Entscheidend ist hier bloß, dass `CreateQuadTree()` die Geometrie vor dem *Lightmappen* strukturiert und `GeometryOnLine()` eine Liste der geschnittenen Endast-Sektoren zurückgibt.

### *3.3.4.2 Die SendRay-Erweiterung*

Wir erweitern also `SendRay` vor dem `Return True` um eine Abfrage auf Kollision mit *Meshes*: Da die vorangegangenen Operationen `xyz0` verändert haben und wir in `xyz1` den Vektor vom Anfangs- zum Endpunkt der Lichtstrecke benötigen, müssen wir die Zahlen etwas anpassen:

```
     x1 = x1*i ; y1 = y1*i; z1 = z1*i
     x0 :- x1; y0 :- y1; z0 :- z1
```

Nun lassen wir uns von `GeometryOnLine` die `QuadTreeNodes` auf der Linie ausgeben und überprüfen mit `RayHitsGeometry`, ob der *Ray* Dreiecke dieser Nodes trifft.

```
     geo = TMain.world.quadtree.GeometryOnLine(x0,y0,z0,x0+x1*2,y0+y1*2,z0+z1*2)
     If RayHitsGeometry(x0,y0,z0,x1,y1,z1,geo) Then Return False
```

Die `RayHitsGeometry`-Funktion ist an sich recht einfach:

```
Function RayHitsGeometry:Int(x0:Float,y0:Float,z0:Float, x1:Float,y1:Float,z1:Float,..
                                                          nodes:TQuadNode[])
     Local le:Int, i:Int
     Local node:TQuadNode

     For node = EachIn nodes ' Alle Nodes durchlaufen,
          le = node.tris.length-1
          For i = 0 To le Step 9 ' darin alle Dreiecke durchlaufen
               ' und falls Schnitt vorliegt, True zurückgeben!
          Next
     Next
End Function
```

`node.tris[i..i+8]` ergeben hierbei jeweils ein Dreieck, das dann auf einen Schnitt geprüft wird. Sollte einer vorliegen, wird `True` zurückgegeben. Der Strahl-Dreiecks-Schnitt soll hier nicht weiter betrachtet werden. Im Internet finden sich viele Algorithmen für dieses Problem[8].

8 Verschiedene Autoren (1989-2007): „3D Object Intersection". URL: http://www.realtimerendering.com/int/ [2007-12-09]

### 3.3.5 Himmelslicht

In einer Landschaft ist das Richtungslicht der Sonne nicht die einzige natürliche Lichtart. Das Sonnenlicht wird in unserer Atmosphäre gestreut und fällt aus allen Richtungen auf die Szene. Das simulieren wir bisher mit dem Umgebungslicht, das auch im Schatten liegende *Lumel* beleuchtet. In der Realität ist diese ambiente Beleuchtung allerdings davon abhängig, wie viel Himmel vom betrachteten Punkt aus sichtbar ist. Dieser Umstand lässt sich simulieren, indem wir pro *Lumel* eine Hand voll (Anzahl wird durch `skylight` bestimmt) *Rays* in zufällige Richtungen senden. Das Umgebungslicht wird daraufhin für diesen *Lumel* mit dem Verhältnis aus den den Himmel erreichenden zu den insgesamt gesandten *Rays* multipliziert.

```
c = 0
For i = 1 To skylight ' Sende skylight zufällige Rays und zähle, wie viele etwas treffen.
     RandomVector(sx,sy,sz, -1,1, 0,2, -1,1) ' Zufällige Richtung sxyz.
     If Not SendRay(x,y,z, sx,sy,sz, 7) Then
          c :+ 1 ' Mitzählen, wenn etwas die Sicht blockierte.
     EndIf
Next
c = (skylight-c)/skylight   ' Errechne den Anteil der erfolgreichen an allen gesandten Rays.
' Setze die Umgebungsfarbe je nach Himmelsichtbarkeit als Farbe für den Lumel.
r = ambr*c
g = ambg*c
b = ambb*c
```

# 3.4 Die Vor- und Nachteile

Das *Lightmapping*-Problem unter Verwendung von *Rays* zu lösen, hat vielerlei Vorteile. Das beginnt mit der Flexibilität und Erweiterbarkeit: Bereits das Himmelslicht ließ sich sehr leicht integrieren. Auch Punktlichter wären mittels *Raytracing* in wenigen Zeilen umsetzbar. Weiterhin ließen sich weiche Schatten durch das Aussenden mehrerer *Rays* pro *Lumel* leicht erzeugen. Sogar ein wenig Lichtstreuung für Unterwasserszenen wäre denkbar. De facto lässt sich mit *Raytracing* jedes bekannte Lichtphänomen von Reflexion bis Brechung, von Streuung bis zu indirekter Beleuchtung berechnen, denn man simuliert tatsächliche Lichtausbreitung. Je mehr Genauigkeit man hinzufügt, desto mehr Performance-Einbußen muss man hinnehmen. Besonders komplexe Dreiecksobjekte können konventionellem *Raytracing* Geschwindigkeitsprobleme bereiten, wenn man nicht auf fortgeschrittene Optimierungen wie *BSP* und *Octtrees* zurückgreift. Und selbst dann bleibt das Verfahren nicht ideal für Dreiecke geeignet.

# 4 Methode B: Texelmapping

## 4.1 Die Grundidee

Im Folgenden möchte ich noch ein zweites Verfahren vorstellen, das ich in keiner anderen Publikation ausführlich beschrieben fand.

Dieses Mal soll die Beleuchtung von der Lichtquelle aus betrachtet werden und somit der tatsächlichen Ausbreitungsrichtung des Lichts folgen. Das wird meistens unter Verwendung von Shadowmapping[9] realisiert. Diese Methode setzt man sowohl im Echtzeit-*Rendering* als auch in verschiedenen *Raytracern* ein. Aus Sicht der Lichtquelle wird hierbei eine Map berechnet, die für einen Raum von radialen oder auch parallelen Richtungen (je nach Lichtquellenart) die Tiefe berechnet, wie weit man vom Licht aus sozusagen in die Szene hineinschauen kann. *Raytracer* verwenden hierfür Forward-*Raytracing.*
Echtzeitanwendungen nutzen die normale *Rendering*-Pipeline der Grafikkarte, indem sie die Szene aus Lichtquellensicht *rastern* und danach den Tiefenbuffer (Depth-Buffer oder Z-Buffer genannt) in die Shadowmap kopieren. Beim Zeichnen des finalen Bildes wird dann jeder Pixel ausgewertet: Liegt er nach Transformation ins Lichtkoordinatensystem im durch die Shadowmap definierten, von der Lichtquelle aus sichtbaren Bereich, wird der Pixel beleuchtet gezeichnet, andernfalls abgedunkelt.

Ich möchte dieses Verfahren dahingehend abwandeln, dass nicht nur die Tiefe für eine Richtung ermittelt wird, sondern die Grafikkarte tatsächlich ausgibt, welche *Lumel* sichtbar sind und wie viel von jedem *Lumel* zu sehen ist.

Dazu wird jedem einzelnen *Lumel* schon von vornherein eine individuelle Farbe, eine Zeigerfarbe[10], zugewiesen, sodass man nur noch alle Pixel des *Render*bildes aus Lichtsicht auswerten muss, um zu ermitteln, welche *Lumel* sichtbar und demnach beleuchtet sind. Die *Lumel*, die nicht gerendert wurden, bzw. nach einem Depth-Test überschrieben wurden, bleiben in der Umgebungslichtfarbe.

Da das Verfahren darauf basiert, eine Map davon zu erstellen, wo sich welcher *Lumel* als *Texel* befindet, nenne ich es hier *Texel*mapping. Mir ist keine andere Bezeichnung bekannt.

9 Everit; Rege; Cebenoyan (2001): Hardware Shadow Mapping. URL: http://developer.nvidia.com/attach/8456 [2007-12-08]
10 Elias, Hugo (2000): „Radiosity“, Absatz: Optimising with 3D Rendering Hardware. URL: http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm [2007-12-12]

# 4.2 Die Vorgehensweise

Zunächst werden wir wieder alle Lichter in einer Liste sammeln und die Umgebungslichtfarbe ermitteln. Danach werden wir eine Kamera erstellen und jedem *Lumel* eine individuelle Indexfarbe geben. Nachdem die *Lightmap* mit der Umgebungslichtfarbe gefüllt wurde, gehen wir dann alle Lichter durch, um Licht und Schatten zu berechnen. Dabei wird die Kamera jeweils entsprechend positioniert und die Szene je nach *Raster*- und *Lightmap*auflösung mehrmals gerendert, wobei wir zählen, wie oft die einzelnen *Lumel* auf den Rasterbildern zu sehen sind. In der *Lightmaptextur* werden daraufhin die sichtbaren *Lumel* aufgehellt.

# 4.3 Die Implementierung

Der vollständige Quelltext findet sich in `„lmtexelmapper.bmx“` (siehe Anhang B).

### 4.3.1 Vorbereitungen

Zunächst stellen wir wie gehabt eine Liste der Lichter zusammen.

```
For ent = EachIn world.entities      ' Gehe jedes Entitiy durch,
    If ent.typ = TEntity.LIGHT       ' wenn es ein Licht ist,
        Select TLight(ent).ltyp      ' von welchem Typ ist es?
        Case TLight.AMBIENT          ' Falls es Umgebungslicht ist,
            amb = (TLight(ent).r Shl 16) | (TLight(ent).g Shl 8) | TLight(ent).b
                                     ' dann merke seine Farbe fürs Umgebungslicht,
        Default                      ' Om Normalfall:
            lights = lights + [TLight(ent)] 'Füge das Licht einfach zur Lichtliste hinzu!
        End Select
    EndIf
Next
```

Da wir die Szene *rendern* wollen, benötigen wir natürlich auch eine Kamera dazu. Die für diese Tests geschaffene Umgebung stellt dafür die Klasse `TCamera` bereit.

```
camera = TCamera.Create()
```

Für einen späteren Pixeltransfer reservieren wir einen Speicherbereich `pix` von den Maßen, in denen wir *rendern,* `TM_RESOLUTION * TM_RESOLUTION *` Farbtiefe in Bytes.

```
GCSuspend ' BlitzMax-Speichermanagement durch GarbageCollector ausschalten
pix = MemAlloc(TM_RESOLUTION*TM_RESOLUTION*3)
```

Nun kommen wir zu einem der entscheidenden Gedanken hinter dieser Methode: Jeder *Lumel* soll auf der *Textur* eine Farbe haben, die man ihm eindeutig zuordnen kann. Hierzu bedienen wir uns des Umstandes, dass Farben aus drei Bytes zusammengesetzt sind, die wir für eine Art 24-Bit-Integer nutzen können. Wir können somit einfach die Map-Position des *Lumels* speichern als `ly*pm.width+lx` mit `ly` als vertikaler Koordinate, `lx` als horizontaler und `pm.width` als Breite einer horizontalen Reihe. Die *Texel*-Farbe wird dadurch ein Index für den *Lumel*. Zuletzt sendet `TTexture.Update` die *Textur* an *OpenGL*.

```
For lx = 0 To pm.width-1 ' Alle Lumel durchgehen und ihnen
     For ly = 0 To pm.height-1
          pm.WritePixel(lx,ly,ly*pm.width+lx) ' eine Indexfarbe zuweisen.
     Next
Next
range = pm.width*pm.height ' Merken, in welchem Zahlenbereich sich die Indices bewegen.

tex.Update ' Sende die geänderte Textur an die Grafikkarte
```

Wir färben die *Lightmappixmap* `pm` mit der Umgebungsfarbe `amb` ein.

```
pm.ClearPixels(amb)
```

### 4.3.2 Die Beleuchtung

Jetzt kann das eigentliche *Rendern* der *Lightmap* erfolgen. Wie schon beim *Raytracing* arbeiten wir dafür Licht für Licht ab.

#### *4.3.2.1 Das Licht einrichten*

Die Kamera wird der Lichtquelle entsprechend gedreht und so positioniert, dass die ganze Szene im Sichtfeld liegt.

```
For light = EachIn lights ' Alle Lichter durchgehen und
     ' die Lichtkamera entsprechend der Lichtposition und -richtung ausrichten.
     Select light.ltyp
     Case TLight.DIRECTIONAL
          camera.Position(ter.heightmap.size/2-light.nx*ter.heightmap.size,-
light.ny*ter.heightmap.size,ter.heightmap.size/2-light.nz*ter.heightmap.size)
          camera.Rotate(-light.pitch, 180+light.yaw, 0)
     End Select
```

Wie viele Pixel in der Kameraansicht ein *Lumel* füllt, speichern wir in einem Array `values[]` von den Maßen der *Lightmap*. Natürlich müssen wir bei jedem Licht als erstes alle `values` auf `Null` initialisieren.

```
     For x = 0 To pm.width-1
          For y = 0 To pm.height-1
               values[x,y] = 0
          Next
     Next
```

#### *4.3.2.2 Die Szene rastern*

Da wir stets in ein begrenzt großes Feld *rendern*, aber auch größere *Lightmaps* möglich sind und selbst bei kleinen nur die wenigsten *Lumelflicken* direkt auf die Kamera zeigen, müssen wir die Szene mehrmals *rastern* lassen. Je höher die Auflösung der *Lightmap* im Verhältnis zur *Render*auflösung ist, desto häufiger bzw. in desto kleineren Schritten muss gerendert werden. Dies ist notwendig, da sonst einige *Flicken* durchs Raster fielen, was dazu führen würde, dass diese immer dunkel blieben - auch ohne Schatten werfende Objekte.
Die Schrittweite `stepsize` ergibt sich wie folgt:

```
     stepsize = TM_RESOLUTION / 1.5 / Float(pm.width)
```

```
    gx = -1; While gx <= 1; gx :+ stepsize ' Der Schrittweite entsprechend mehrmals rendern.
         gy = -1; While gy <= 1; gy :+ stepsize
              Select light.ltyp ' Lichtkamera leicht verschieben für den Antialiaseffekt.
              Case TLight.DIRECTIONAL
                   camera.OffSet(gx*.64,gy*.64)
              End Select
```

Nun ist die Szene zu *rendern*. Dafür stellt `TDisplay` eine spezielle Funktion `RenderTexels()` bereit. Wir lassen diese die Szene orthogonal aus Sicht der `camera` in der Auflösung `TM_RESOLUTION²` in einen Raum von 1000 Einheiten in der Tiefe und mit einer Höhe und Breite von 1,5-facher *Terrain*größe *rastern*. Woher kommen die letzten drei Parameter? Die Tiefe sollte im Allgemeinen möglichst gering sein, damit der Tiefenbuffer genau arbeitet, darf die Terraingröße aber natürlich nicht unterschreiten. Für die *Terrains*, mit denen wir arbeiten, reicht eine Tiefe von 1000 Einheiten. Weiterhin muss natürlich ein größerer Bereich in Höhe und Breite berechnet werden, wenn die Seitenlänge des *Terrains* groß ist. Da das *Terrain* seine größte Ausdehnung in der Diagonale hat und diese im schlimmsten Fall parallel zur einer der Seiten des *Render*bereichs liegt, muss dieser Bereich $\sqrt{2} * ter.heightmap.size$ Einheiten, also das rund 1,5-Fache einer *Terrain*seite, breit und hoch sein. Wichtig an `TDisplay.RenderTexels` ist vor allem: Es die Szene orthogonal darstellt, da wir Richtungslicht verwenden; Es keinen Nebeleffekt gibt, der die *Pixel*farben verändert; Bei der Darstellung von *Texturen* wird die *Pixel*farbe nicht zwischen den *Texeln* interpoliert, sondern wird jeder *Texel* als *Flicken* von einheitlicher Farbe dargestellt - sonst wäre eine eindeutige Indexierung nicht mehr möglich.

```
              TMain.display.RenderTexels(camera,TM_RESOLUTION,1000,1.5*ter.heightmap.size)
```

Dieses in den Framebuffer ge*rasterte* Bild (siehe Abbildung 5) wollen wir jetzt auswerten. Dazu müssen wir auf sehr niedriger Ebene einen PixelTransfer der RGB-Werte durchführen:

```
              ' Grafikausgabe in den Arbeitsspeicher übertragen:
              glReadPixels 0,0,TM_RESOLUTION,TM_RESOLUTION, GL_RGB,GL_UNSIGNED_BYTE, pix
```

#### *4.3.2.3 Auswertung*

Nun werden alle *Pixel* des Bildes durchlaufen und ihre Farben ausgelesen. Falls die Farbe (die Index zu einem *Lumel* sein soll) sich im Raum der möglichen Indices `0...range` bewegt, wird diese danach in die Koordinate des *Lumels* auf der *Lightmap* umgerechnet. Hierzu muss `rgb=y*pm.width+x` umgekehrt werden. `y` ergibt sich als der ganzzahlige Anteil der Division von `rgb/pm.width`, `x` als der Rest dieser Division.

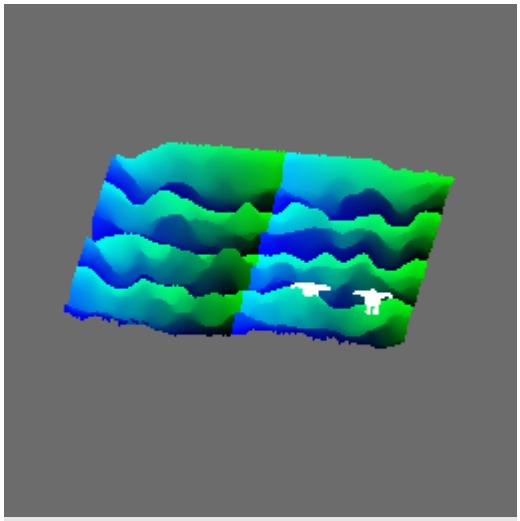

*Abbildung 5: Ergebnis des Rasterns*

Für den bestimmten *Lumel* wird daraufhin in `values` mitgezählt, dass er einen Pixel füllte.

```
            For lx = 0 To TM_RESOLUTION-1 ' Alle Pixel auswerten:
                For ly = 0 To TM_RESOLUTION-1
                    ' Pixelfarbe aus dem Speicherbereich holen
                    pixc = Byte Ptr (pix+ly*TM_RESOLUTION*4+lx*4)
                    ' und in einen Integer für den Index umwandeln.
                    rgb = (pixc[0] Shl 16) | (pixc[1] Shl 8) | pixc[2]

                    If rgb > 0 And rgb < range ' Wenn der Index im gültigen Bereich liegt,
                        ' die Pixelfarbe in Lumel-Koordinaten auf der Lightmap umwandeln.
                        y = rgb / pm.width
                        x = rgb Mod pm.width

                        ' ... und die Lumelbeleuchtung speichern:
                        values[x,y] :+ 1
                    EndIf
                Next
            Next
        Wend ' Ende der Mehrfach-Raster-Schleife
    Wend
```

#### *4.3.2.4 Anwendung*

Nachdem wir nun alle notwendigen Informationen zu jedem *Lumel* für diese Lichtquelle haben, muss nur noch die *Lumel*farbe in der *Lightmap* geändert werden, indem die durch diese Lichtquelle erfolgende Beleuchtung zur Farbe addiert wird.

```
    For x = 0 To pm.width-1
        For y = 0 To pm.height-1
            ' Farbinformationen aus der Lightmaptextur holen
            pixc = pm.PixelPtr(x,y)
            ' ... und die Beleuchtung durch diese Lichtquelle addieren:
            r = pixc[0]+values[x,y]*light.r/12
            g = pixc[1]+values[x,y]*light.g/12
            b = pixc[2]+values[x,y]*light.b/12
            ' Da die Farbkomponenten in Bytes gespeichert werden
            ' und es nichts röteres als Rot geben kann,
            ' wird bei Überbelichtung der Wert auf 255 heruntergeschraubt.
            If r>255 Then r=255
            If g>255 Then g=255
            If b>255 Then b=255
            ' Pixel setzen:
            pm.WritePixel(x,y,(r Shl 16) | (g Shl 8) | b)
        Next
    Next
Next
```

### 4.3.3 Aufräumen

Zuletzt muss der Speicherbereich wieder freigegeben sowie die normale Speicherverwaltung reaktiviert werden.

```
            MemFree pix ' Speicherbereich wieder freigeben
            GCResume ' Speicherverwaltung wieder an BlitzMax-GarbageCollector zurückgeben
```

# 4.4 Die Vor- und Nachteile

Der ohne Zweifel größte Vorteil dieser *Render*methode ist es, dass Dreiecksgeometrie dank der 3D-Hardware-Beschleunigung sehr schnell verarbeitet werden kann. Funktionalität, die die 3D-Umgebung zur Verfügung stellt, kann auch hier genutzt werden. So wären zum Beispiel teilweise transparente *Texturen*, wie sie in Bäumen und Zäunen sehr häufig vorkommen, ein sich von selbst lösendes Problem. Weiterhin ergibt sich eine oberflächenrichtungsabhängige Schattierung genau wie Antialiasing von selbst, da die Methode von Grund auf dazu ausgelegt ist, dass ein *Texel* mehrere Pixel des *Raster*bildes füllen soll.
Sehr praktisch ist ebenfalls die Unabhängigkeit von der tatsächlichen Form des Objektes: Der Prozessor muss nicht die Raumkoordinaten eines *Lumles* ermitteln, denn die Hardware erledigt das. Solange die *Lumel* halbwegs gleichmäßig große Bereiche auf den Oberflächen füllen, können wir jede Art von Objekt ohne weitere Gedanken *lightmappen* lassen.

Als Nachteil ist jedoch zu sehen, dass auch das *Terrain* mit Dreiecken dargestellt werden muss, was zumindest bei *Terrains* deutlich komplexer ist als ein paar Zugriffe auf die *Heightmap,* wie sie der *Raytracer* nur benötigt. Punktlichter müssen bei dieser Technik durch eine Cubemap, also sechsfaches *Rendern*, und das Überlagern eines Filters umgesetzt werden, da sonst nicht alle Richtungen der Lichtausbreitung behandelt und ordnungsgemäß gewichtet werden.
Der größte Flaschenhals ist und bleibt allerdings die Übertragung der *Raster*daten aus dem Framebuffer in den Arbeitsspeicher. Eine Messung der Rechenzeit der einzelnen Abschnitte auf meinem System für eine 256x256 *Lumel* große *Lightmap* brachte folgende Ergebnisse:

| | |
|---|---|
| Vorbereitung (Einfärben der Texel etc.): | 3 ms |
| Rendern (Rastern der Szene): | 42 ms |
| Datenübertragung (Grafik -> Hauptspeicher): | 172 ms |
| Datenauswertung (Analyse der sichtbaren Texel): | 16 ms |
| Zeichnen der Ergebnisse in die Lightmap: | 7 ms |

Mir fehlt leider das tiefere Wissen über 3D-Hardware sowie die langjährige Erfahrung mit *OpenGL*, welche nötig wären, um an dieser Stelle noch größere Optimierungen vorzunehmen, denn mit konventionellen Mitteln kommt man nicht mehr weit.

# 5 Vergleich und Fazit

## 5.1 Vergleich

Die Vor- und Nachteile der beiden Verfahren wurden bereits in den jeweiligen Kapiteln beschrieben. Im Folgenden soll verglichen und bewertet werden, für welche Anwendungen welcher Algorithmus besser geeignet ist.

*Raytracing* hat den großen Vorteil, dass es zu beliebig komplexen Lichtsimulationen verwendet werden kann, wohingegen das Texelmapping nur so gut simulieren kann, wie es die 3D-Hardware erlaubt. Während man beim *Raytracing* einfach eine Hand voll weiterer *Rays* emittieren muss, um globale Beleuchtung zu simulieren, müsste man beim Texelmapping das gesamte System umkrempeln und mit mehreren Passes arbeiten, um ein Radiosity-Ergebnis[11] zu erzielen. Wenn es also auf ein möglichst realistisches Beleuchtungsmodell ankommt, sollte man wohl zu einer *Raytracing*-Lösung greifen.

Des Weiteren erlaubt *Raytracing* das Arbeiten mit sehr hoch auflösenden *Terrains*, ohne dadurch deutlich langsamer zu werden. Die folgende Grafik zeigt, wie die *Terrain*größe sich auf die Berechnungszeit für die *Lightmap* auswirkt. Relativierend sollte hierbei jedoch bedacht werden, dass meine Implementierung des *Raytracings* das *Terrain* intern ohnehin auf eine einheitliche Größe umrechnet. Bei diesen Testläufen gab es keine Schatten werfenden *Meshes*. Man kann gut erkennen, wie viel schneller *Raytracing* bloße *Terrains* verarbeiten kann. Texelmapping hingegen hat Schwierigkeiten mit großen *Terrains*, da diese mehr *Raster*zeit benötigen. Der von sich aus hohe Wertebereich des Texelmappings kommt vom oben genannten Flaschenhals an der Pixelübertragung in den Arbeitsspeicher, welche unabhängig von der Szenenkomplexität viel Zeit kostet.

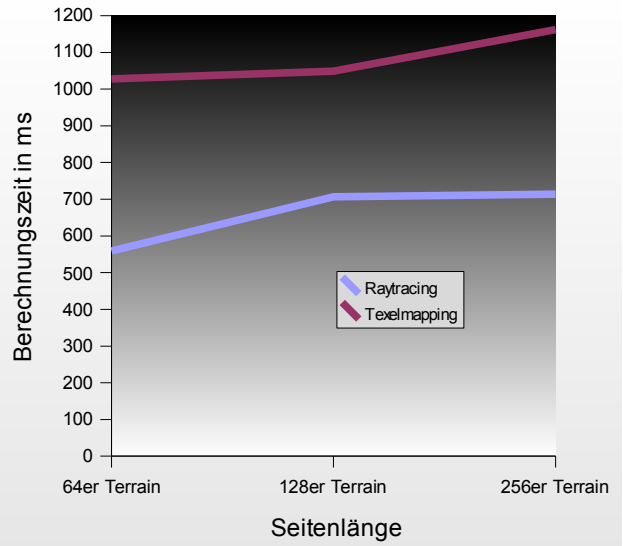


*Abbildung 6: Berechnungszeit in Abhängigkeit von der Terraingröße bei 512er Lightmap, keine Meshes*

11 Elias, Hugo: "Radiosity" (2000). URL: http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm [2007-12-12]

Wenn wir anstelle der *Teraingröße* die Menge an *Meshes* erhöhen, werden die Stärken des Texelmappings deutlich. Es zeigt sich von ein paar mehr oder weniger zu überprüfenden Dreiecken vollkommen unbeeindruckt, wohingegen die Performance des *Raytracings* einbricht. Auch hier muss zugestanden werden, dass man bei der Implementierung des *Raytracing*-Algorithmus einiges wiedergutmachen kann. Weder verwende ich die allerbeste Dreiecksschnittfunktion, noch die beste Szenenunterteilung (Siehe *QuadTree*), denn dies hätte den Rahmen gesprengt. Vielleicht könnte man sich das Leben leichter machen, wenn alles auf *Voxeln* basieren würde. Dennoch bleibt der Fakt bestehen: 3D-Hardware ist für Dreiecke geschaffen, mathematisches *Raytracing* nicht. Mit *Voxeln* und 3D-Hardware-Beschleunigung kann man nichtsdestotrotz recht schnell *raytracen*.[12]

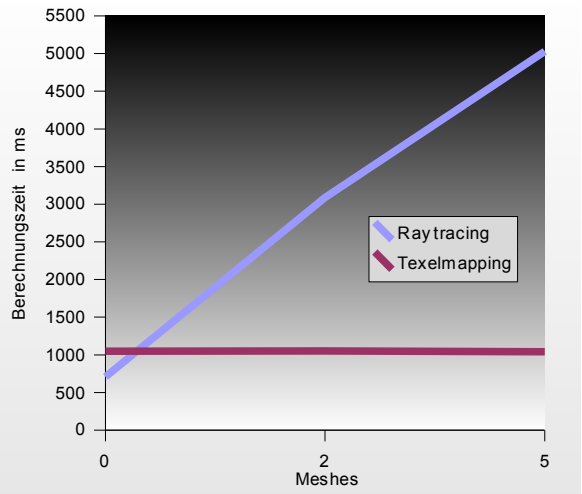


*Abbildung 7: Berechnungszeit in Abhängigkeit von der Anzahl der Meshes. 512er Lightmap, 128er Terrain*

# 5.2 Fazit

- *Raytracing* bietet sich bei simplen *Terrains* an, die nur auf sich selbst Schatten werfen sollen. Ansonsten ist es zur *Lightmap*generierung nur sinnvoll, falls man sehr komplexe Beleuchtungssituationen simulieren möchte, was allerdings nur wenig Qualitätsgewinn gemessen am Aufwand bringt.
- Texelmapping hat zwar eine gewisse Grundausführungszeit, lohnt sich jedoch ab einer größeren Anzahl von *Meshes* massiv. Man muss nicht wissen, wo die *Lumel* sich in der 3D-Szene befinden oder wo sie auf der *Lightmap* sind. Entsprechend lässt sich das Verfahren ohne Anpassung bei beliebigen zu *lightmappenden* Objekten verwenden.

# 5.3 Weitere Entwicklung

Und genau das Letztgenannte werde ich wohl demnächst noch mit Texelmapping umsetzen. Auch andere Erweiterungen der Algorithmen und der Umgebung hin zu einem brauchbaren Terraineditor sind denkbar, falls meine Zeit dies zulässt. *Raytracing* werde ich vorerst nur in meinem Testraytracer weiter verfolgen:[13]

12 Christen, Martin: "Implementing Ray Tracing on GPU" (2005). URL: http://www.clockworkcoders.com/oglsl/rt/index.html [2007-12-12]
13 http://www.mrkeks.net/?show=artikel160

# Literaturverzeichnis

Blitz Research (2005): „Official Blitz Max Reference“.

Channa, Keshav (2003): „Light Mapping - Theory and Implementation“. (Flipcode).
URL: http://www.flipcode.com/articles/article_lightmapping.shtml [2007-12-12]

Elias, Hugo: "Radiosity" (2000).
URL: http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm [2007-12-12]

Fischer, Robert; Perkins, Simon; Walker, Ashley; Wolfahrt, Erik (2003): Hyper Media Image Processing Reference, Gaussian Smoothing.
URL: http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm [2007-12-10]

Marghidanu, Mircea (2002): „Fast Computation of Terrain Shadow Maps“. (Gamedev.net).
URL: http://www.gamedev.net/reference/articles/article1817.asp [2007-12-12]

NeHe Productions: „NeHe OpenGL Tutorials“.
URL: http://nehe.gamedev.net [2007-12-12]

OpenGL Architectural Review Board, The: „The OpenGL Reference Manual“.
URL: http://www.talisman.org/opengl-1.1/Reference.html [2007-12-12]

Soconne (2006): „Faster Ray Traced Shadow Maps“. (The Game Programming Wiki).
URL: http://gpwiki.org/index.php/Faster_Ray_Traced_Terrain_Shadow_Maps [2007-12-12]

TrippleX (2006): „Terrain Shadowmapping“. (GameStudio Wiki).
URL: http://www.coniserver.net/wiki/index.php/Terrain_Shadowmapping [2007-12-12]

# Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die Facharbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

Der gesamte Quelltext stammt von mir, insofern es sich nicht um gängige Abfolgen von Befehlen handelt oder die Funktionen eindeutig als von anderen übernommen gekennzeichnet wurden.

Wenn nicht anders gekennzeichnet, sind die verwendeten Grafiken und Illustrationen sowohl im Programm als auch in der Facharbeit mein Werk, beziehungsweise im Fall der Screenshots aus Programmen entnommen, an deren Entwicklung ich beteiligt war.

Benjamin Bisping
Berlin, den 14.12.2007

# Glossar

**BSP, Quadtree, Octtree**
Methoden der hierarchischen, rekursiven Unterteilung des Raums mit jeweils zwei / vier / acht Kinderästen zum optimierten prüfen auf Schnitte, Kollisionen etc.

**Heightmap**
zweidimensionale Karte von Höheninformationen in Form eines Graustufen-Rasterbildes, meist für Terrains

**Lightmap** auch *Shadowmap*
Textur zum Speichern der statischen Beleuchtung einer Szene, meist im Blendmodus Multiplikation/Modulate

**Lumel** auch *Patch/Flicken* oder schlicht *Pixel*
Element einer Lightmap; wie Pixel zu Picture

**Mesh** auch *Polygonnetz*
Struktur zur Beschreibung von aus Polygonen (meist nur Dreiecke) zusammengesetzten Objekten

**OpenGL**
API für grafikkartenbeschleunigte Grafikausgabe, meist 3D-Grafik

**Pixmap** auch *Bitmap* oder *Rastergrafik*
Struktur für aus Pixeln zusammengesetzte Bilder [in der Arbeit meist verwendet in Anlehnung an die benutzten TPixmap-Objekte aus dem Pixmap-Modul von BlitzMax]

**Ray** (engl. für *Strahl*)
auf Schnitte zu prüfender Strahl, manchmal auch nur Strecke, beim Raytracing

**Raytracing** auch *Raycasting*
Methode zur Lichtsimulation oder Bildgenerierung durch Rückverfolgen des Lichtwegs mittels Rays

**Terrain**
In der 3D-Computergrafik: Objekt zur Anzeige von Landschaften, die mit Höheninformationen aus Heightmaps beschrieben werden

**Texel**
Textur-Element; wie Pixel zu Picture

**Textur** (auch engl: *Texture*)
Rastergrafik, die auf 3D-Oberflächen gezogen wird, um Detail hinzuzufügen

**Rastern**
Prozess der Erzeugung einer Rastergrafik aus Geometrieinformationen (für gewöhnlich Vektorgrafiken oder 3D-Szenen)

**Rendern**
Erzeugung von Mediendaten aus Rohdaten (z.B. 3D-Szene -> Rasterer -> Bildschirmausgabe oder 3D-Szene -> Lightmapper -> Lightmap)

**Voxel**
Volumenelement, ein würfelförmiger Bereich des Raums; wie Pixel zu Picture

# Anhang A: Das Szenen-Format

Die Beschreibung der Szene erfolgt in Form der üblichen INI-Syntax:

```
[Sektionsname]
; Kommentar
Attributname = Wert
```

Die einzige Abweichung im Vergleich zu üblichen INI-Dateien ist die, dass Sektionen nicht unbedingt einzigartig sein müssen, sondern mehrfach vorkommen dürfen.

Aus jeder Sektion wird ein Objekt in der Welt erzeugt.

Bisher sind die folgenden Sektionen mit ihren Parametern integriert:

**[terrain]**: Beschreibt ein Terrain. Zur Zeit wird nur maximal ein Terrain unterstützt.

| | |
|---|---|
| size: | Seitenlänge des Terrains |
| heightmap: | Bilddatei der Höhenkarte; schwarz = niedrig, weiß = hoch |
| material: | Bilddatei für Materialtextur |
| materialscale: | Wie häufig soll die Materialtextur sich wiederholen? |
| lightmap: | Bilddatei für Lightmaptextur |

**[light]**: Beschreibt eine Lichtquelle.

| | |
|---|---|
| name: | Name des Objekts |
| x: | X-Koordinate (horizontale Verschiebung) |
| y: | Y-Koordinate (vertikale Verschiebung) |
| z: | Z-Koordinate (Verschiebung in der Tiefe) |
| pitch: | Rotation an X-Achse |
| yaw: | Rotation an Y-Achse |
| roll: | Rotation an Z-Achse |
| type: | Typ der Lichtquelle (1: Umgebungslicht, 2: Richtungslicht) |
| red: | Rotkomponente der Lichtfarbe |
| green: | Grünkomponente der Lichtfarbe |
| blue: | Blaukomponente der Lichtfarbe |

**[mesh]**: Beschreibt ein Mesh. Polygoninformationen werden aus externer Datei geladen.

| | |
|---|---|
| x, y, z, pitch, yaw, roll, name: | siehe [light] |
| file: | Pfad zu Meshdaten, bisher nur B3d-Format[14] teilweise unterstützt |

14 Sibly, Mark / Blitz Research Ltd (2002): Blitz3d file format V0.01. (Public Domain) URL: http://www.blitzbasic.com/sdkspecs/sdkspecs/b3dfile_specs.txt [2007-12-10]

# Anhang B: Der Quelltext

Auf der beiliegenden CD findet sich im Ordner „programm/“ eine EXE-Datei des „Terrainlightmapping-Testprogramms“ sowie der Quelltext. Den Quelltext sollte man in der Datei „main.bmx“ zu lesen beginnen.
Im Ordner „programm/media/“ befinden sich einige Testlandschaften.

Außerdem enthält der Ordner „text/“ eine PDF-Version dieser Arbeit.

Alternativ lässt sich der CD-Inhalt auch als Archiv von http://www.mrkeks.net/files/bll.zip herunterladen.