Diplomarbeit

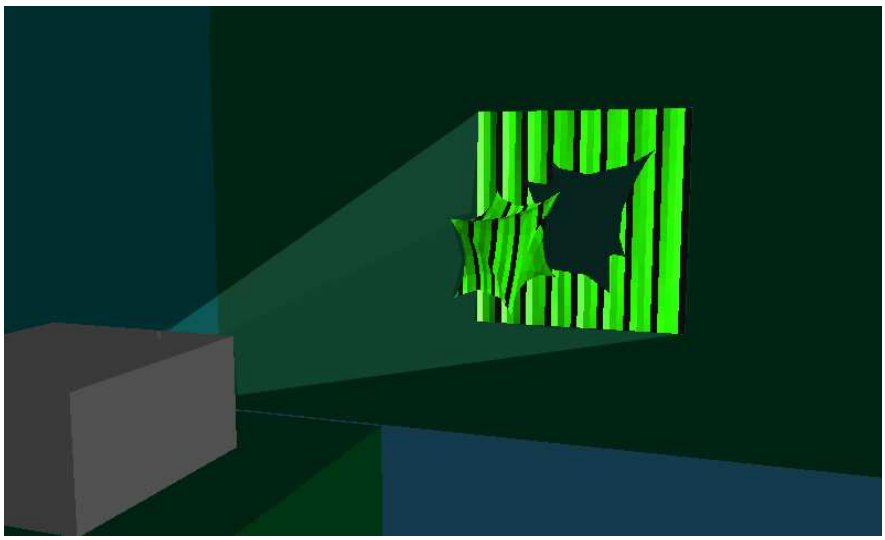
Über Projektionen und Spiegelungen von ebenen Texturen

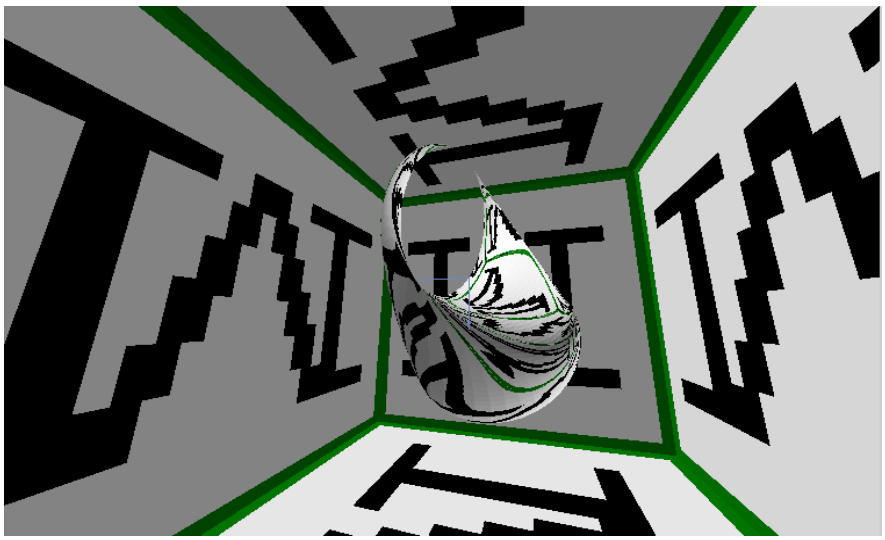
ausgeführt am Ordinariat für
Geometrie der Universität für
angewandte Kunst Wien

unter der Anleitung von
Univ.Prof.Dr. Georg Glaeser und
Univ.Ass.Dr. Hans-Peter Schröcker
als verantwortlich mitwirkenden Universitätsassistenten

von
Gruber Franz
Schweglerstr. 31/12
1150 Wien

Wien, September 2003





Über Projektionen und Spiegelungen von ebenen Texturen

Franz Gruber

1 Einleitung

Man stelle sich einen Diaprojektor vor, der das Bild eines beliebigen Fotos auf ein weißes, im Wind flatterndes Leintuch wirft. Man würde doch mit einer gewissen Neugierde dem Ergebnis entgegensehen, vor allem, wenn Wind, Foto, Projektorposition und Betrachterposition stetig veränderliche Größen sind.
Dies via Computeranimation zu simulieren war die Grundidee meiner Diplomarbeit, welche aus praktischen Experimenten mit Diaprojektor und rotierenden Projektionsflächen entstanden ist.
Die Beschäftigung mit diesem Thema führte mich fast automatisch zur Frage von Spiegelungen von Texturen an geometrischen Flächen. Dieser zweite Teil meiner Arbeit, sowie eine Übersicht der heutzutage verwendeten Methoden für Spiegelungen wird ab Seite 39 diskutiert.
Bei der Programmierung von Lichtprojektionen muss man mehrere Klassen von Projektionsflächen unterscheiden, weil geschlossene Körper (Würfel, Tetraeder, Kugel,...) anders zu behandeln sind als Funktionsgraphen. Funktionsgraphen sind wiederum in explizite und implizite dargestellte Funktionsgraphen zu unterteilen. Algebraische Funktionsgraphen sind weiters ihrem Grade nach zu unterscheiden, da ihre numerische Behandlung unterschiedlich ist.
Vor allem aber besteht ein großer programmiertechnischer Unterschied zwischen zwei Grundideen im Bildaufbau:

Methode A: Die Partitionierung des Dias mit anschließender Berechnung der Durchstoßpunkte der Lichtstrahlen durch die Projektionsfläche.

Methode B: Die Partitionierung der Projektionsfläche mit anschließender Berechnung der Durchstoßpunkte der Lichtstrahlen durch das Dia.

Außerdem treten die Fragen nach der Sichtbarkeit, Schattierung und perspektivischer Darstellung des Abgebildeten auf. Ein zentrales Problem war, bei allen Varianten des Problems die Geschwindigkeit des Algorithmus zu maximieren, um eine Echtzeit-Animation (in Bewegung) zu ermöglichen. Die einzelnen Programme wurde mit der Geometrie Software *Open Geometry* [1] erstellt, welche auf *C++* und *OpenGL* basiert. Den Autoren dieser Software, Georg Glaeser und Hans-Peter Schröcker, möchte ich an dieser Stelle herzlich für ihre hilfsbereite Betreuung und Unterstützung meiner Diplomarbeit danken.
Die Software samt einer ausführlichen Programmbeschreibung in Buchform ist im Handel erhältlich. Im folgenden Aufbau der verschiedenen Konzepte werde ich z.T. die zugehörigen Auszüge (Definitionen, Befehlszeilen, Beispiele . . .) des `Originalcodes` in den den Text einbinden, sodass man das Skriptum parallel zum Code lesen kann. Auf der beigelegten CD - ROM sind Demoversionen (exe files) von animierten Projektionen und Spiegelungen, sowie deren Source Codes zu finden.

2 OPEN GEOMETRY - Programmstruktur

Open Geometry Programme haben folgende Struktur:

```
#include opengeom.h"
      //andere Includefiles und
      //Deklaration der eigenen globalen Variablen
void Scene::Init()
{
      //Inititialisierung der globalen Variablen
      //und Vorbereitungen fürs Programm
}
void Scene::Draw()
{
      //Hauptteil des Programms:
      //Gibt die Bildszene aus
}
void Scene::Animate()
{
      //Animationsteil:
      //Bewegte Szenen können hier
      //Rahmen für Rahmen gesteuert werden
}
void Scene::CleanUp()
```

```
{
      //Freigabe von allokiertem,
      //globalem Speicher, falls notwendig (meistens ein Dummyfunktion)
}
void Projektion::Def()
{
      //Initialisierung des Ausgabefensters
      //z.b. Kameraposition
}
```

3 Projektionen

In diesem Kapitel werden verschiedene Möglichkeiten der Projektion von ebenen Texturen auf gekrümmte Flächen bzw. geschlossene Körper diskutiert, und mit der sogenannten *Methode der kollinearen Dreiecke* (siehe S. 17) wird eine dafür eigens entwickelte Abbildungsmethode vorgestellt.

3.1 Input Parameter

1. Lichtquelle : $\vec{L}$

   ```
   V3d  L(Lx,Ly,Lz);
   ```

2. Diamittelpunkt : $\vec{M}$

   ```
   V3d  M(Mx,My,Mz);
   ```

3. Dia (bitmap) : $Map1$

   ```
   TextureMap  Map1("PATH.bitmapname.bmp");
   ```

4. Projektionsfläche: ε

   ```
   Real Function1(Real x,Real y);
   ```

5. Parameter der Bewegung: Amplitude, Frequenz

   ```
   Real  amplitude,f;
   ```

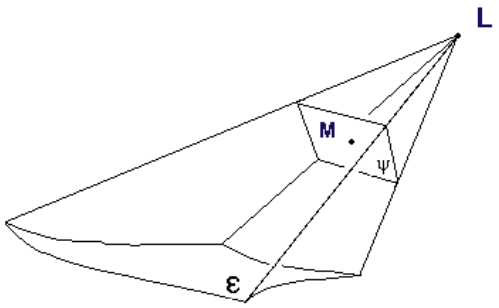


Abbildung 1: Licht,Dia und Projektionsfläche

3.2 Das Dia

Das sogenannte Dia ist in meinen Programmcodes stets ein Bitmapfile der Pixelgröße $n \times m$, wobei n und m Zweierpotenzen sein müssen, da sonst Probleme beim Einlesen des Files entstehen. Diese Bitmap braucht noch einen Rahmen

```
Poly3d TheSlide;
```

der Größe $a \times b$ in dem es sitzt, damit man es positionieren (verschieben und rotieren) kann. Die Ausmaße des Rahmens sind also grundsätzlich beliebig. Zumeist möchte man aber, dass das Verhältnis von Länge zu Breite dem der Bitmap gleicht, da sonst das Bild gestaucht oder gestreckt erscheint.
Wie bei einem gewöhnlichen Diaprojektor soll auch hier der Richtungsvektor der Achse $\vec{n} = \overrightarrow{LM}$ orthogonal zur Diaebene ψ,

```
Plane psi;
```

in der das Dia liegt, verlaufen.

3.3 Positionierung des Dias

`TheSlide` wurde im Programmteil `Scene::Init()` so initialisiert, dass es in der xz-Ebene liegt und seinen Mittelpunkt im Koordinatenursprung hat. Diese Initialposition wird in Abb. 2 mit ψ_0 bezeichnet. Um es in die richtige Lage zu bringen (Mittelpunkt= $\vec{M}$, $\vec{n} \perp \psi$), müssen zuerst zwei Rotationen um den Ursprung (Eulerwinkel) und erst dann eine Translation des Dias in den eigentlichen Mittelpunkt $\vec{M}$ vorgenommen werden. Der erste Drehwinkel (um die $x-$Achse) `angle_x` errechnet sich aus dem Skalarprodukt von $\vec{n}$ und dessen xy-Projektion `n_proj`= $(n_x, n_y, 0)$, und der zweite (um die $y-$Achse) `angle_y` aus dem Skalarprodukt von `n_proj` und dem Einheitsvektor in y-Richtung $\vec{e_y} = (0, 1, 0)$.

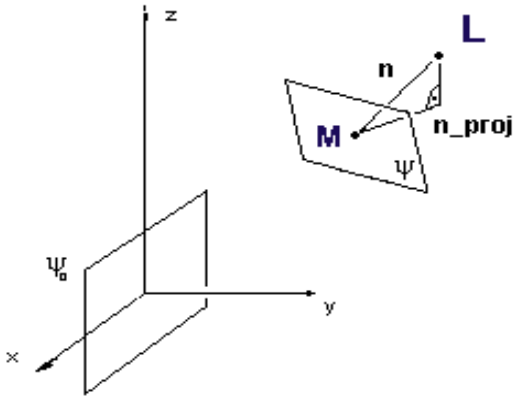


Abbildung 2: Das Dia in Initial.- und Endposition

```
// Scene::Init()

TheSlide[1]( a/2, 0, -b/2); TheSlide[2](-a/2,0,-b/2);
TheSlide[3](-a/2, 0,  b/2); TheSlide[4]( a/2,0, b/2);

Real   l= (M-L).Length();

V3d    n= (1/l*(M-L)),
       n_proj(n.x,n.y,0);

Real
 angle_x= acos (n*n_proj/n_proj.Length()),
 angle_z= acos (n_proj.y/n_proj.Length());

if (n.z < 0)  angle_x= -angle_x;
if (n.x < 0)  angle_z= -angle_z;
 TheSlide.Rotate(Xaxis,Deg(angle_x));
 TheSlide.Rotate(Zaxis,Deg(-angle_z));
 TheSlide.Translate(M.x,M.y,M.z);
```

3.4 Triangulierung

Es stellte sich nun grundsätzlich die Frage, wie man die Bitmap auf die Projektionsfläche „projiziert“. Naheliegenderweise wird man entweder die Fläche, in

der das Dia liegt, triangulieren (Methode A), oder auf der Projektionsfläche ein Triangulierung erzeugen (Methode B). Die Verbindungsgeraden dieser Triangulierung mit der Lichtquelle $\vec{L}$ erzeugen dann automatisch die „korrespondierende“ Triangulierung der anderen Fläche. Zu jedem Dreieck der Diafläche gehört dann im Idealfall genau ein Dreieck der Projektionsfläche. Man hat also zunächst die mathematische Aufgabe, zugehörige Dreiecke zu berechnen, genauer gesagt zugehörige Eckpunktpaare ($\vec{P_\psi}$, $\vec{P_\epsilon}$).

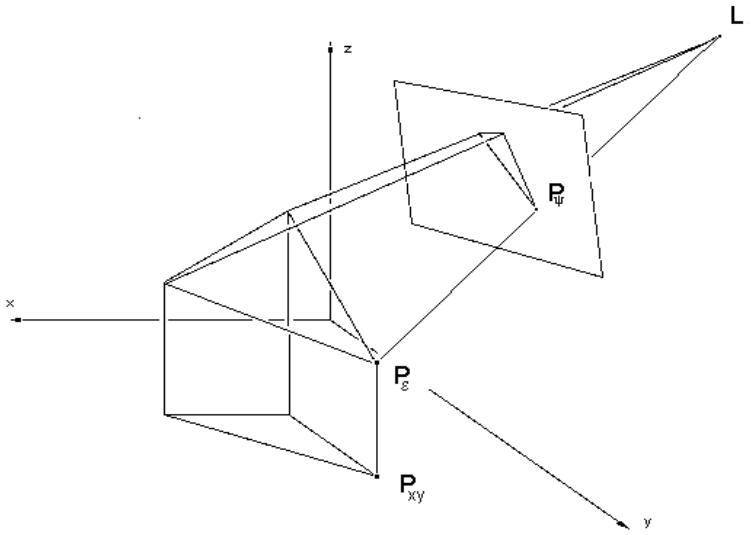


Abbildung 3: Zwei entsprechende Dreiecke

3.5 Methode A: Partitionierung des Dias

Das Dia sitzt also in einem rechteckigem Polygon (Rahmen), welches nun beliebig fein trianguliert wird (Methode A). In meinen Programmen wurde diese Triangulierung immer gleichmäßig vorgenommen. Es wäre jedoch denkbar, Flächen je nach vorliegender Situation „günstig“ zu parametrisieren, wozu allerdings relativ komplexe Algorithmen notwendig wären.

Mit Methode A können Bitmaps ungeschnitten auf die Projektionsfläche abgebildetet werden, d.h. es entstehen vollständige Bilder, wie sie ein *realer Projektor* auch erzeugen würde. Dies ist der große Vorteil dieser numerisch etwas aufwändigeren Methode.
Es sei g: $\vec{X(s)} = (X_1(s), X_2(s), X_3(s)) = \vec{L} + s \cdot \overrightarrow{LP_\psi}$ die Gerade, welche durch

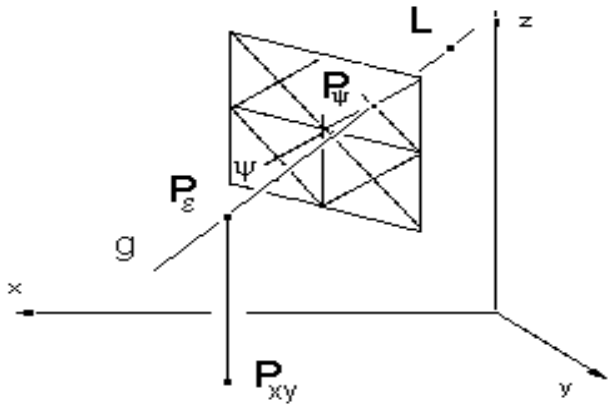


Abbildung 4: Die Triangulierung des Dias

$\vec{L}$ und $\vec{P_\psi}$ definiert ist. Nun gilt es den Schnittpunkt $\vec{P_\varepsilon}$ der Geraden g mit der Projektionsfläche ε zu finden.

Falls ε eine algebraische Fläche vom Grade kleiner gleich 4 ist, ist dies mit exakten Lösungsformeln möglich. Das bedeutet, dass hier ein Durchstoßpunkt besonders schnell und außerdem beliebig genau berechnet werden kann. Auch komplexe Lösungen können eindeutig erkannt werden, sodass die Frage ob überhaupt ein Schnittpunkt vorliegt oder nicht, eindeutig beantwortet werden kann.

Aufgrund der begrenzten Formen von Funktionsgraphen von höchstens 4. Ordnung greift man in den meisten Fällen auf allgemeine Funktionsgraphen zurück. In diesen Fällen habe ich zwei verschiedene Näherungsverfahren verwendet: das Newton´sche Näherungsverfahren und ein simpleres Intervallschrittverfahren, wobei ersteres zwar eine schnellere und elegantere Methode ist, aber bekannterweise nicht immer konvergiert. Dazu kommt die Frage, ob ein gefundener Schnittpunkt auch wirklich der gewüschte, näheste Durchstoßpunkt ist. Wegen diesem Problem habe ich in manchen Fällen die letztere Methode verwendet, bzw. mit der ersten kombiniert.

3.5.1 Durchstoßpunkte mittels Newtonverfahren

Es sei ε im allgemeinsten Fall ein stetiger Funktionsgraph, welcher explizit durch stückweise differenzierbare Funktionen definiert ist, z.B. $z(x,y) = sin(x) + cos(y)$ = `Function1(x,y)`.
Mit dem herkömmlichen Newtonverfahren findet man nun die Nullstellen der Funktion $\Delta z(s) = X_3(s) - f(X_1(s), X_2(s))$ (= `Function2`). Jede Nullstelle

(=s_m) von Δz ergibt dann wegen $\overrightarrow{X(s_m)} = \vec{P}_\varepsilon$ einen Schnittpunkt von g und ε.

```
Real Function1(Real x,Real y)
{
 return (sin(x)+cos(y));
}
Real Function2(Real s,P3d P)
{
  return (L.z+s*(P.z-L.z)-
          Function1(L.x+s*(P.x-L.x),L.y+s*(P.y-L.y)));
}

// Scene::Draw()

eps  = 0.001;
k_max= 5;
sn   = 1;

for (k=1; k<= k_max; k++ )            //NEWTON Approximation
{
   Fs = Function2(sn,P_psi);
   DFs= (Function2(sn+eps,P_psi)-Fs)/eps;
   sn=sn-Fs/DFs;
}

P_eps= L + sn*(P_psi - L);
```

Als Startwert der Newton-Iteration wurde $s_0 = 1$ gewählt, da dieser Parameter die Grenze zwischen Punkten vor und hinter dem Dia beschreibt. Ausgehend davon, dass kein Schnittpunkt des Lichtstrahls g mit der Projektionsfläche hinter dem Dia liegen soll (vernünftige Wahl der Input-Parameter), sucht nun das Newtonverfahren automatisch Parameter, die größer als 1 sind und nähert sich normalerweise relativ schnell dem nähestliegendem Durchstoßpunkt. Im Normfall genügen 5 Iterationen, um eine zufriedenstellend genaue Lösung zu bekommen.

Im Falle keiner Lösung (kein Schnittpunkt) divergiert das Verfahren. Gibt es mehrerer Lösungen, findet das Verfahren zumeist den „richtigen Punkt“, also den nähesten Schnittpunkt, jedoch nicht immer. Es wäre natürlich denkbar, mit verschiedenen Startwerten zu versuchen, nun alle Durchstoßpunkte zu berechnen und denjenigen, der die kleinste Distanz zu $\vec{P}_\psi$ hat, als richtige Lösung auszuwählen. Da diese Idee weder die schnellste noch die sicherste Methode wäre, habe

ich mich bei der Berechnung von kritischen Punkten für das folgende Verfahren entschieden.

3.5.2 Durchstoßpunkte mittels Intervallschrittverfahren

Diese Methode beruht auf einer einfache Idee: Startwert sei wiederum $s_0 = 1$. Um den ersten Schnittpunkt zu finden wird nun schrittweise der Iterationsparameter `sn` um `delta_sn` erhöht, wobei dieser Zuwachs vom Abstand Lichtquelle zu Diamittelpunkt abhängt, etwa ein 1/20 dessen Länge ausmacht. Die Iteration wird mit jenem Parameter s_m abgebrochen mit dem der Lichtstrahl die Graphenfläche durchdringt, also Δz sein Vorzeichen wechselt. Wie oben gilt wiederum $\overrightarrow{X(s_m)} = \vec{P}_\varepsilon$. Achtet man auf eine passende Abstimmung der Input-Parameter, so findet man auf diesem Wege fast immer den ersten Durchstoßpunkt des Lichtstrahls.

```
Real    length= (M-L).Length(),
        delta_sn= 1/(20*length),
        sn=1,
        counter=0,counter_max= 200;

//Intersectionmethod
while (counter<counter_max && (Function2(sn,P_eps)>0)

{
     counter++;
     sn= sn+delta_sn;
}

        P_eps= L+sn*(P_psi-L);
```

3.6 Methode B: Partitionierung der Projektionsfläche

Es sei nun ein bestimmter Bereich D der xy - Ebene unser Definitionsbereich, d.h. nur der über ihm liegende Funktionsgraph soll Projektionsfläche sein. Die Form von D ist grundsätzlich frei wählbar. So könnte der Bereich kreisförmig, unregelmäßig oder unzusammenhängend sein. Obwohl dies künstlerisch durchaus interessante Bilder erzeugen würde, genügt es für eine gewöhnliche Projektion, ein Rechteck zu betrachten.

Angenommen, D ist ein Rechteck, welches gleichmäßig trianguliert wurde (siehe

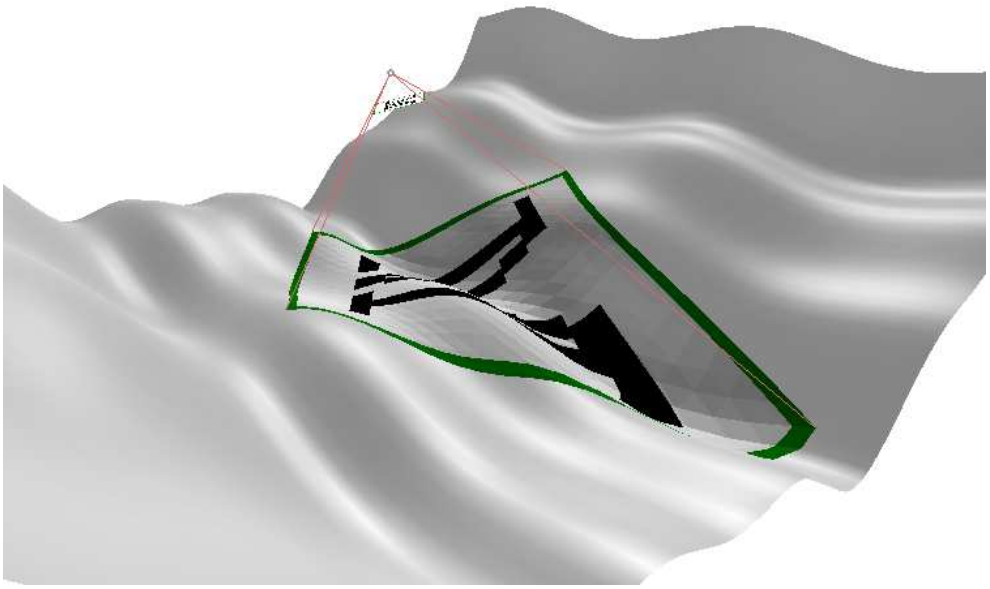


Abbildung 5: Projektion auf Funktionsgraphen

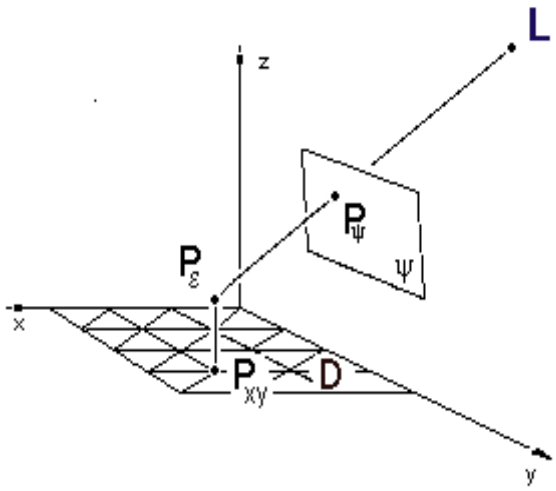


Abbildung 6: Die Triangulierung der xy Ebene

Skizze). Diese Triangulierung erzeugt die gewünschte Triangulierung der Projektionsfläche, indem zu jedem Eckpunkt P_{xy} der zugehörige Punkt $\vec{P}_\varepsilon$ auf dem Funktionsgraphen berechnet wird. Kurz gesagt:
(x,y) $\rightarrow$ *(x,y,Function1(x,y))*.

Die Verbindungsgeraden der Eckpunkte $\vec{P_\varepsilon}$ mit der Lichtquelle $\vec{L}$ ergeben dann durch einfache Vektorrechnung (Schnittpunkte von Geraden mit der Ebene ψ), die zugehörige Triangulierung der Diafläche.

```
StrL3d h0( P_eps,L - (V3d) P_eps);
P_psi= psi*h0;
```

Methode B benötigt also kein Näherungsverfahren, um die zugehörigen Dreiecke der zwei Flächen ε und ψ zu finden, d.h. sie ist eine exakte und schnelle Methode. Ihr großer Nachteil liegt darin, dass die willkürliche Wahl eines fixen Definitionsbereichs die Abbildung des gesamten Dias nicht gewährleisten kann. Die vorgegebene Triangulierung schneidet dann im Normalfall nicht das gewünschte Dia im Ganzen aus, sondern z.B. nur einen inneren Teil oder auch Teile außerhalb des Dias. Wird nämlich versucht, aus einer Bitmap ein Dreieck auszuschneiden, welches nicht zur Gänze im Rahmen liegt so entsteht aus softwaretechnischen Gründen ein fehlerhaftes Bild (Clipping, siehe Seite 27).

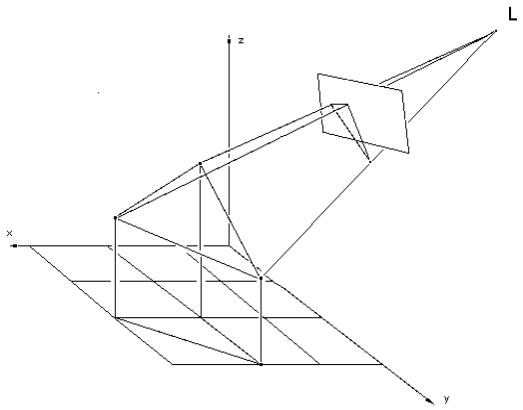


Abbildung 7: Unerwünschte Dreieckslage

Liegen also ein oder zwei Eckpunkte eines berechneten Dreiecks der Ebene ψ außerhalb des Rahmens, so kann dieses Dreieck nicht richtig abgebildet werden. Mit zusätzlichen Algorithmen kann man natürlich aus den Schnittpunkten des Dreiecks mit dem Rahmen den überstehenden Teil des Dreiecks wegschneiden, d.h den verbleibenden inneren Teil erneut triangulieren und abbilden.
Ein Trick, dieses Abbildungsproblem zu umgehen,war die Idee die Bitmap mit einem Passepartout in der Farbe der Projektionsfläche (z.B. weiß) zu umkleiden. Dazu muss man die ursprüngliche Bitmap zentrisch in eine größere Bitmap (z.B. doppelte Pixelgröße $2n \times 2m$) kopieren, die passende Passepartoutfarbe wählen,

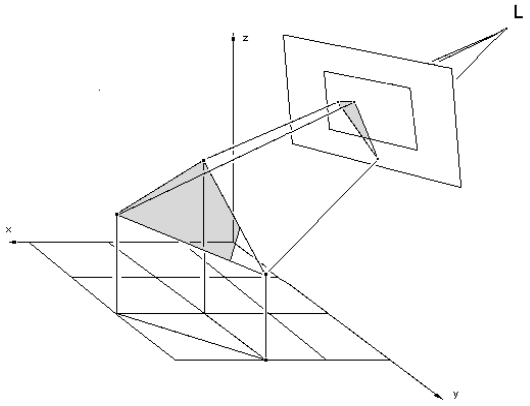


Abbildung 8: Das Dia mit Passepartout

und unter neuem Namen speichern. Im Programm wird dann auch nur mehr dieses File includiert (z.B. `sky_framed.bmp`). Wählt man auch die Rahmengröße etwa doppelt so groß wie gewohnt ($2a \times 2b$), so kann man bei vernünftiger Wahl des Definitionsbereichs D erwarten, dass man wie bei Methode A eine vollständige Abbildung des Dias erreicht. Dreiecke, die teilweise außerhalb des ursprünglichen Dias liegen, können nun ohne Clipping abgebildet werden, da sie jetzt zur Gänze im Rahmen liegen. Der im Passepartout liegende Teil ist auf der Projektionsfläche nicht zu sehen, da er die selbe Farbe hat wie diese (siehe Abb.8).
Wie man sieht, ist also immer darauf zu achten, dass der Definitionsbereich, Projektorposition und Rahmengröße aufeinander abgestimmt sind, was nach einigen Versuchen kein Problem mehr darstellt. Nachteil dieser Vorgangsweise ist klarerweise die Extraanfertigung einer Bitmapgrafik, deren größere Dimension sich außerdem auf die Rechenzeit schlägt.

3.7 Projektionsflächen

3.7.1 Punktweise definierte Objekte

Die Wahl der Projektionsfläche ist prinzipiell mit keinen Einschränkungen verbunden, man kann jedes Objekt frei definieren und dann als Leinwand verwenden. Man kann z.B. ein Objekt punktweise, in Form eines feinen Netzes festlegen, und eine sich daraus ergebende Triangulierung unter Verwendung von Methode B zur Abbildung benutzen. Möchte man die Triangulierung nach Methode A aufbauen, so hätte man das rechenintensive Problem, den richtigen Durchstoßpunkt eines Lichtstrahls, geschnitten mit all jenen Ebenen die das Projektionsnetz definieren, herauszufinden. Methode B eignet sich hingegen gut zur Berechnung der zuge-

hörigen Triangulierungen, dafür bringt sie die beschriebenen Probleme mit der Sichtbarkeit und einer vollständigen Abbildung des Dias mit sich.

3.7.2 Parametrisierte Objekte

Eine andere Möglichkeit, die Projektionsfläche zu definieren, sind mathematisch definierte Objekte (Funktionsgraphen, parametrisierte Flächen und Körper, stückweise definierte Objekte wie z.B. Pyramiden), von denen hier ein paar einfache Beispiele angeführt werden. Es ist naheliegend zunächst harmonische Flächen zu verwenden, also explizite Funktionsgraphen (Abb.9) der Form

$$f(x,y,t) = \sum_{n=1}^{N} A_n sin(nx + a_n t) + B_n cos(y + b_n t)$$

Die Abhängigkeit von der Zeit (t entspricht im Programm `mytime`) ermöglicht eine Bewegung des wellenförmigen Funktionsgraphen, wobei A_n bzw. B_n die Amplituden und a_n bzw. b_n die Fortpflanzungsgeschwindigkeiten der einzelnen Teilschwingungen festlegen.

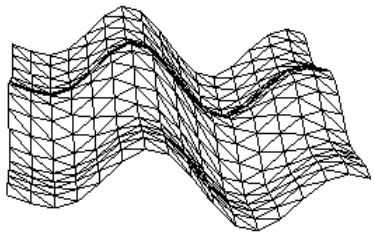


Abbildung 9: Harmonischer Funktionsgraph

Ein anderes Beispiel für die Wahl der Projektionsfläche ist ein parametrisierter geschlossener Körper (Abb.10):

$$(x,y,z)_{(u,v)} = (r * cos(u) + sin(v + \pi/4), r * cos(v) * sin(u), r * sin(v))$$

mit $u \in [0, 2\pi], v \in [-\pi/2, \pi/2]$.

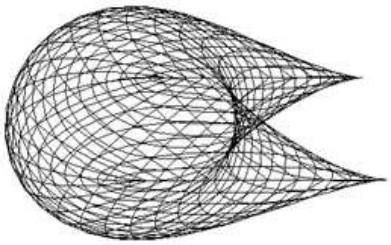


Abbildung 10: Parameterisierter Projektionskörper

Eine mathematische Definition der Projektionsfläche ermöglicht eine mathematische Berechnung der Durchstoßpunkte, auch bei Verwendung von Methode A (siehe S. 9 und 11). Zur Modellierung von beliebigen Flächen, durch selbstgewählte Stützpunkte, kann eine Interpolation von 3D-Flächen (B-Splines, NURBS ...) verwendet werden.

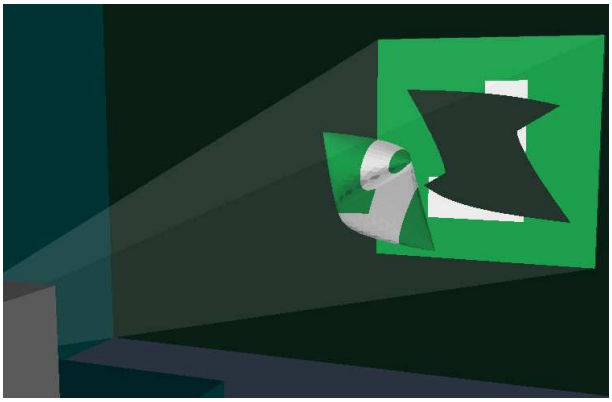


Abbildung 11: Projektion auf geschlossenen Körper

3.8 Dreiecke im Schatten

Fällt das Projektorlicht auf stark hügeliges Gelände, so kann es passieren, dass Teile der Projektion im Schatten liegen. Möchte man eine realitätssgetreue Abbildung erreichen, so müssen diese Teile, besser gesagt die betreffenden Dreiecke unbeleuchtet bleiben.

Methode A teilt das Dia in Dreiecke und wirft ihre Bilder auf ε. Das Intervallschrittverfahren sucht automatisch jene Punkte, die von den Lichtstrahlen tatsächlich beleuchtet werden und Bereiche, die dem Licht abgewandt sind, bleiben unbeleuchtet. Um die Grenze zwischen Licht und Schatten zu finden, braucht man allerdings eine sehr feine Triangulierung (etwa 100×100), da ansonsten eine sichtlich „gezackte" Trennlinie entsteht. Eine derart hohe Auflösung wirkt sich beträchtlich auf die Rechenzeit aus.

Methode B unterteilt die Projektionsfläche ε in Dreiecke und bildet die zugehörigen Dreiecke des Dias auf diese ab. Es werden also auch den Dreiecken im Schatten Diaausschnitte zugeordnet. Damit man diese Teile unbeleuchtet darstellen kann, muss man zuerst feststellen, ob ein Dreieck im Eigenschatten liegt oder nicht. Dazu muss man seine Lage zu den einfallenden Lichtstrahlen prüfen, genauer gesagt den Winkel zwischen dem äußerem Normalvektor der Dreiecksebene und einer der drei Verbindungsgeraden vom Lichtpunkt zu einem der Dreieckspunkte. Ist dieser Winkel stumpf, so ist die Außenseite des Dreiecks dem Licht abgewendet, also unbeleuchtet, darzustellen. Für die Auflösung gilt gleiches wie oben.

Das Problem der Schattenbildung ist tatsächlich ein Problem für sich. Exakte Algorithmen müssten auf der gesamten Projektionsfläche nach Eigenschattengrenzen suchen, also alle Punkte von ε, deren Tangentialebene die Lichtquelle $\vec{L}$ enthält. Da diese Suche ebenfalls einen sehr großen numerischen, wie auch programmiertechnischen Aufwand bedeutet, habe ich es vorgezogen, in solchen Fällen eine feine Triangulierung zu verwenden.

3.9 Die Methode der kollinearen Dreiecke

3.10 Grundidee

Die korrespondierenden Triangulierungen von ε und ψ werden nun verwendet, um den Bildinhalt eines jeden Dreiecks $\triangle ABC$ (`P3d A,B,C;`) der Diaebene, in das zugehörige Dreieck der Projektionsfläche (`Poly3d SurfTriangle`) abzubilden. Anstatt eine solche Abbildung mittels *Texture Mapping* (siehe S. 27) zu realisieren, entstand die ungewöhnliche Idee zur sogenannten *Methode der kollinearen Dreiecke*.
Gegeben seien $\triangle ABC$, $\triangle A_1B_1C_1$ und $\vec{E}$, der Augpunkt des Betrachters (`P3d Eye`). Um den Bildinhalt von $\triangle ABC$ auf $\triangle A_1B_1C_1$ zu „projizieren", besser gesagt um ihn darin kollinear vergrößert zu sehen, kann man nun einen optischen Trick anwenden. Man betrachte die *Sehpyramide* mit der Basis $\triangle A_1B_1C_1$ und der Spitze

$\vec{E}$. Es seien a, b, c die Trägerhalbgeraden der Seitenkanten dieser Pyramide, also

$$a : a(\vec{x}) = \vec{E} + x \cdot \vec{\mathbf{a}}$$
$$b : b(\vec{y}) = \vec{E} + y \cdot \vec{\mathbf{b}}$$
$$c : c(\vec{z}) = \vec{E} + z \cdot \vec{\mathbf{c}}$$

mit $\vec{\mathbf{a}} = E\vec{A}_1^0$, $\vec{\mathbf{b}} = E\vec{B}_1^0$ und $\vec{\mathbf{c}} = E\vec{C}_1^0$, den drei normierten Richtungssvektoren und $x, y, z > 0$. Wir suchen nun Punkte $A_2 \in a$, $B_2 \in b$ und $C_2 \in c$, sodass $\triangle ABC$ und $\triangle A_2B_2C_2$ kongruent sind. Falls dies möglich ist, so kann das $\triangle ABC$ samt seinem Bildinhalt in die Position A_2, B_2, C_2 gebracht, bzw. in die Sehpyramide eingepaßt werden. Da das Monoauge (im Modus der Zentralperspektive) zwischen kollinearen Dreiecken innerhalb einer Sehpyramide nicht unterscheiden kann, täuscht ihm das vorgesetzte Dreieck das gewünschte Bild vor, nämlich die Projektion von $\triangle ABC$ auf $\triangle A_1B_1C_1$.

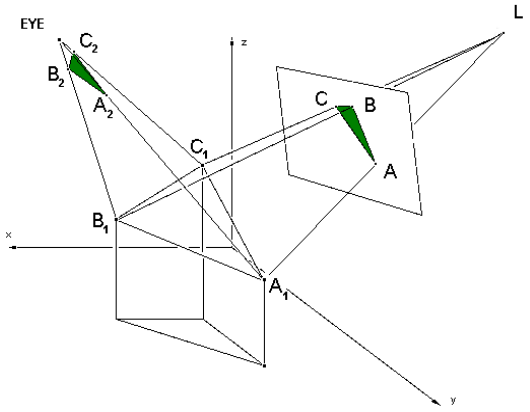


Abbildung 12: $\triangle ABC$ in der Position A_2, B_2, C_2

3.11 Berechnung des Dreiecks $A_2B_2C_2$

Seien d_a,d_b,d_c die Seitenlängen von $\triangle ABC$ und x,y,z die zu A_2,B_2,C_2 gehörenden gesuchten Parameterwerte. Aufgrund der gewünschten Kongruenz von

$\triangle ABC$ und $\triangle A_2B_2C_2$ müssen folgende Beziehungen erfüllt sein:

$$\begin{aligned} I: &\quad \mid x\vec{\mathbf{a}} - y\vec{\mathbf{b}} \mid &= d_c \\ II: &\quad \mid x\vec{\mathbf{a}} - z\vec{\mathbf{c}} \mid &= d_b \\ III: &\quad \mid y\vec{\mathbf{b}} - z\vec{\mathbf{c}} \mid &= d_a \end{aligned}$$

Dieses Gleichungssystem hat folgende betragsfreie Form:

$$\begin{aligned} I: &\quad x^2 - 2\vec{\mathbf{a}}\vec{\mathbf{b}}xy + y^2 &= d_c^2 \\ II: &\quad x^2 - 2\vec{\mathbf{a}}\vec{\mathbf{c}}xz + z^2 &= d_b^2 \\ III: &\quad y^2 - 2\vec{\mathbf{b}}\vec{\mathbf{c}}yz + z^2 &= d_a^2 \end{aligned}$$

Deutet man x, y und z als Koordinaten in einem euklidischen Dreiraum, so stellen die drei Gleichungen in dieser Reihenfolge Zylinder ζ_c, ζ_b bzw. ζ_a von 2.Ordnung mit sich paarweise orthogonal schneidenden Achsen (nämlich den Achsen des Koordinatensystems) dar. Der Typus der Zylinders wird vom Vorzeichen der Zahlen $(\vec{\mathbf{a}}\vec{\mathbf{b}})^2$-4, $(\vec{\mathbf{a}}\vec{\mathbf{c}})^2$-4 und $(\vec{\mathbf{b}}\vec{\mathbf{c}})^2$-4 entschieden. Sie sind aufgrund der Voraussetzungen alle negativ, die Zylinder daher elliptisch. Im Sinne der algebraischen Geometrie gibt es acht Lösungstripel (x, y, z). Mit (x, y, z) ist allerdings auch $(-x, -y, -z)$ eine Lösung des Problems. Insgesamt kann das Problem daher durch Auffinden der Wurzeln eines Polynoms vierten Grades gelöst werden.
Anschaulich ist dabei klar, dass alle Lösungen imaginär sein können: Man kann nämlich, z.B. durch Vergrößern von d_a und durch Verkleinern von d_c (Vergrößern und Verkleinern der Zylinderquerschnitte) erreichen, dass die Durchdringungskurve $\zeta_b \cap \zeta_a$ keinen reellen Punkt mit ζ_c gemeinsam hat. Praktisch entspricht das folgender Situation: Man wähle für $\vec{\mathbf{a}}, \vec{\mathbf{b}}, \vec{\mathbf{c}}$ drei orthogonalen Einheitsvektoren und $\triangle ABC$ stumpfwinkelig. Dann ist es offensichtlich unmöglich, das Dreieck so in die Sehpyramide einzupassen, dass die Parameterwerte $x, y, z > 0$ sind. Das Problem ist also genau dann schlecht lösbar, wenn Winkel zwischen den Seitenkanten der Sehpyramide groß sind, bzw. der Augpunkt dem beobachteten Dreieck nahe liegt.

Um nun aus unserem gekoppelten Gleichungssystem in drei Variablen eine Gleichung vierten Grades in einer Variable zu bekommen, sind einige Umformungen nötig. Subtrahiert man Gleichung II von Gleichung I, so erhält man

$$x^2 - 2\vec{\mathbf{a}}\vec{\mathbf{b}}xy + 2\vec{\mathbf{b}}\vec{\mathbf{c}}yz - z^2 = d_c^2 - d_a^2$$

Addiert man diese Gleichung zu Gleichung III, so ergibt sich

$$A: 2x^2 - 2\vec{\mathbf{ab}}xy - 2\vec{\mathbf{ac}}xz + 2\vec{\mathbf{bc}}yz = -d_a^2 + d_b^2 + d_c^2$$

Aus Gleichung I wurde somit eine Gleichung, die nur mehr in einer Variablen quadratisch ist. Dasselbe kann man mit Gleichung II und Gleichung III machen und erhält:

$$B: 2y^2 - 2\vec{\mathbf{ab}}xy + 2\vec{\mathbf{ac}}xz - 2\vec{\mathbf{bc}}yz = d_a^2 - d_b^2 + d_c^2$$

$$C: 2z^2 + 2\vec{\mathbf{ab}}xy - 2\vec{\mathbf{ac}}xz - 2\vec{\mathbf{bc}}yz = d_a^2 + d_b^2 - d_c^2$$

Mit den Abkürzungen $L = \vec{\mathbf{ab}}$, $M = \vec{\mathbf{bc}}$, $N = \vec{\mathbf{ac}}$ und $F = (-d_a^2 + d_b^2 + d_c^2)/2$, $G = (d_a^2 - d_b^2 + d_c^2)/2$, $H = (d_a^2 + d_b^2 - d_c^2)/2$ ergibt sich letztlich

$$\begin{aligned} A: &\quad x^2 - Lxy + Myz - Nzx &&= F \\ B: &\quad y^2 - Lxy - Myz + Nzx &&= G \\ C: &\quad z^2 + Lxy - Myz - Nzx &&= H \end{aligned}$$

Aus dem ursprünglichem Gleichungssystem wurde also ein System dreier Gleichungen mit jeweils nur einer quadratischen Variablen. Aus Gleichung A ergibt sich:

$$z = \frac{x^2 - Lxy - F}{Nx - My} \tag{1}$$

Setzt man dieses in Gleichung B ein erhält man mit

$$y^2 - 2Lxy - F = G$$

eine quadratische Gleichung in y, mit der Lösung:

$$y_{1/2} = Lx \pm \sqrt{L^2x^2 - x^2 + F + G} \tag{2}$$

(2) in (1) eingesetzt macht z von y unabhängig:

$$z = \frac{x^2(1 - L^2) \mp \sqrt{L^2x^2 - x^2 + F + G} - F}{Nx - My} \tag{3}$$

Setzt man schlussendlich (2) und (3) in Gleichung C ein, so ergibt sich ein gerade

Gleichung 8.Grades in x:

$$a + bx^2 + cx^4 + dx^6 + ex^8 = 0 \tag{4}$$

Setzt man $u := x^2$, wird aus (4) eine Gleichung 4.Grades:

$$a + bu + cu^2 + du^3 + eu^4 = 0 \tag{5}$$

Die Koeffizienten dieser Gleichung wurden mithilfe des Programms *Mathematica* vereinfacht und werden hier in Form eines Auszugs aus dem Originalcode angegeben (a= `coeff[0]`, b= `coeff[1]`, c= `coeff[2]`, d= `coeff[3]`, e= `coeff[4]`).

```
Real L=a*b,                          // Routine Solve_Formula1
     M=c*b,
     N=a*c,
     F= (Sqr(dc)-Sqr(da)+Sqr(db))/2,
     G= (Sqr(da)-Sqr(db)+Sqr(dc))/2,
     H= (Sqr(db)-Sqr(dc)+Sqr(da))/2,
     coeff[5];

  coeff[0]= -Sqr(F^2*(-1+M^2)+M^2*(F*G+F*H+G*H));

  coeff[1]= 2*(F^3*(2-L^2+M^2*(-4+L^2)+2*M^4+2*L*M*N*(1-M^2)
            +N^2*(-1+M^2))+F^2*(G*L^2+M^2*(-3*G-3*H+H*L^2)
            +(3*M^4+2*L*M*N-4*L*M^3*N)*(G+H)+N^2*(H+G*M^2))
            +H*L*M^3*N*(-F*(6*G+2*H)-2*G*(G+H))
            +H^2*M^2*N^2*(F+G)
            +M^4*(G^2*(F+H)+4*F*G*H+H^2*(F+G))
            +M^2*(-2*+F*G*H+L^2*G^2*(F+H))
            +2*F*G*L*M*N*(2*H-G*M^2));

  coeff[2]= F^2*(-6+L^2*(6-L^2-6*M^2+4*L*M*N-2*N^2-4*Sqr(M*N))
            +M^2*(12-6*M^2+12*L*M*N-6*N^2)+N^2*(6-N^2)
            +L*M*N*(-12+4*N^2))+L^4*(G*(2*F-G))
            +M^4*(G*(-6*F-G)+H*(-6*F-4*G-H))
            +4*L^3*M*N*(F*H+G*(G+H))
            +M^3*4*L*(G*N*(4*F+G+3*H)+H*N*(4*F+H))
```

```
            +L^2*(-4*F*G +M^2*(-2*G^2-4*F*H
            +4*N^2*( -3*F*G-G^2-3*F*H-3*G*H-H^2))
            +2*N^2*(3*F*G+3*F*H+G*H))
            +2*M^2*(3*F*G+3*F*H+G*H+N^2*(-2*F*G-H^2))
            +8*L*M*N*(-F*G-F*H-G*H)+4*L*M*N^3*(F*G+G*H+H^2)
            +H*N^2*(-4*F+N^2*(2*F-H));

coeff[3]= 2*(-1+L^2+M^2-2*L*M*N+N^2)
            *(-2*F+L^2*(F-G)+M^2*(2*F+G+H)+2*L*M*N*(-F-G-H)
            +N^2*(F-H+L^2*(2*G+2*H)));

coeff[4]= -Sqr(-1+L^2+M^2-2*L*M*N+N^2);
```

3.12 Lösbarkeit des Problems

Gleichungen 4.Grades besitzen exakte Lösungsformeln, welche auch in *Open Geometry* implementiert sind (`ZerosOfPoly4(coeff,zero)`).
Dabei sind `coeff[5]` und `zero[4]` die Koeffizienten- bzw. die Nullstellenliste. Der Rückgabewert der Funktion `ZerosOfPoly4` ist eine Integerzahl, welche die Anzahl der reellen Lösungen angibt (`number_of_RealRoots`).
u_1, u_2, u_3, u_4 (= `zero[0],zero[1],zero[2],zero[3]`) seien also die Lösungen der Gleichung (5). Zieht man aus diesen die Wurzeln, so bekommt man die acht Lösungen von Gleichung (4).

Für positive, reelle x-Lösungen, ist y und z gemäß (2) bzw. (1) zu berechnen. Dabei muss über die richtige Wahl des Vorzeichens in Gleichung (2) entschieden werden, um letztendlich ein Zahlentripel (x, y, z) zu finden mit $x, y, z > 0$, das Lösung aller drei Gleichungen ist. Im häufigsten Fall gibt es zwei verschiedene Lösungen, also zwei verwendbare Lagen für das $\triangle ABC$, um vor dem Auge in die Sehpyramide eingepasst zu werden.
Gibt es keine solche Lösung, muss man sich mit einer Notlösung zufriedengeben ($x = y = z = 1$). Damit treten zumindest keine Lücken im Netz der Dreiecke auf, die Notlösung verursacht aber eine Verzerrung vom Bildinhalt des betroffenen Dreiecks. Solche Situationen treten jedoch sehr selten auf, und sind im Gesamtbild kaum augenfällig.

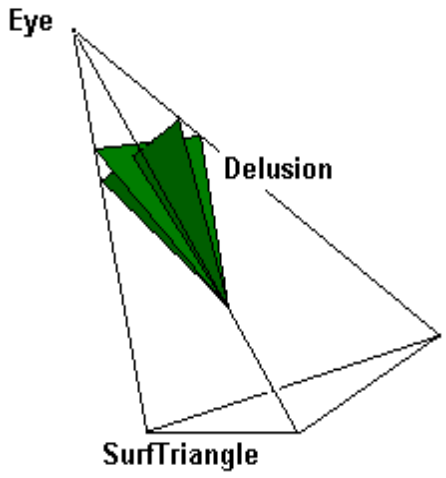


Abbildung 13: Zwei brauchbare Lösungslagen

```
// Routine Solve_Formula1
   int number_of_RealRoots = ZerosOfPoly4(coeff,zero);

   Real eps1 = 0.001,
        eps2 = 10000000;

   if (number_of_RealRoots > 0)
   {
      for (i=0;i <=3 ;i++)
      {
         x= zero[i];

         if ((x > eps1) & (x < eps2))
         {
             x = Sqrt(x);
             y = L*x+Sqrt(fabs(Sqr(L*x)-Sqr(x)+F+G));
             z = (x*(x-L*y)-F)/(N*x-M*y);

               if(fabs(Sqr(z)-2*c*a*z*x+Sqr(x)-Sqr(db)) > eps1)
               {
                  y = L*x-Sqrt(fabs(Sqr(L*x)-Sqr(x)+F+G));
                  z = (x*(x-L*y)-F)/(N*x-M*y);
               }
```

```
            break;
        }
    }
}

if (number_of_RealRoots == 0 || x<0 || y<0 || z<0)
{ // Emergency situation
     x=1;
     y=1;
     z=1;
}
```

3.13 Ein Dreiecksschwarm

Der Aufbau von Programmen, die die Methode der kollinearen Dreiecke verwenden, ist der folgende: Im Programmteil `Scene::Draw()` wird in einer `for`-Schleife jeweils ein Dreieck berechnet und in der zugehörigen Sehpyramide abgebildet. Für jedes Dreieck müssen die Punkte `A,B,C` und die dazu passenden Dreieckspunkte A_1, B_1, C_1, also die Eckpunkte von `SurfTriangle` (Methode A oder Methode B) berechnet werden.
Jedes dieser entsprechenden Dreieckspaare $(\triangle ABC, \triangle A_1B_1C_1)$ hat ein dazu passendes $\triangle A_2B_2C_2$ (`Poy3d Delusion`), welches durch den Aufruf der Subroutine `Solve_Formula1(A,B,C,SurfTriangle,Delusion)` berechnet wird.
In dieser Subroutine wird zunächst (x, y, z), eine geeignete Lösung von Gleichung (4), und anschließend $\triangle A_2B_2C_2$ berechnet:

$$\begin{aligned}\vec{A_2} &= \vec{E} + x \cdot \vec{a} \\ \vec{B_2} &= \vec{E} + y \cdot \vec{b} \\ \vec{C_2} &= \vec{E} + z \cdot \vec{c}\end{aligned}$$

bzw:

```
Delusion[1](Eye.x+x*a.x,Eye.y+x*a.y,Eye.z+x*a.z);
Delusion[2](Eye.x+y*b.x,Eye.y+y*b.y,Eye.z+y*b.z);
Delusion[3](Eye.x+z*c.x,Eye.y+z*c.y,Eye.z+z*c.z);
```

Vor dem Auge liegen nun viele kleine Dreiecke in genauso vielen kleinen Sehpyramiden. Diese partitionieren das gesamte Sehfeld und sind zueinander disjunkt. Von der richtigen Position gesehen, nämlich vom Augpunkt aus, ergibt diese Wolke von Dreiecken ein geschlossenes Bild - das auf die Projektionsfläche projizierte Dia, besser gesagt die Illusion der Projektion.

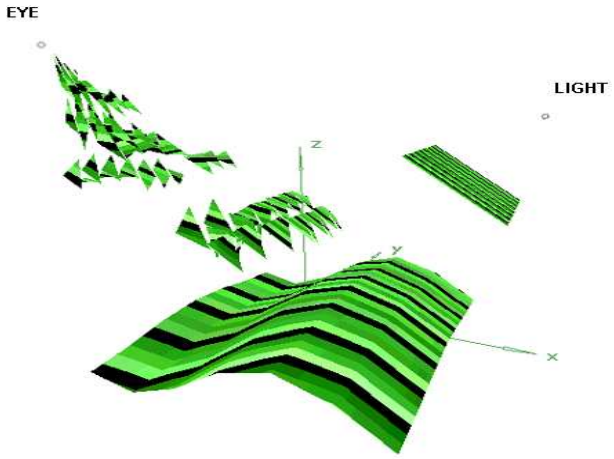


Abbildung 14: Illusion und wahres Bild

Abb. 14 ist leicht verwirrend: ein mit *Open Geometry* erzeugtes Bild, in dem ich zum Zwecke der Illustration sowohl das zerschnipselte Dia (Dreiecksschwarm) als auch das dadurch erblickte Bild in eine Szene gesetzt habe. Außerdem ist die Betrachterposition der Szene nicht wie gewöhnlich das Auge, sondern ein willkürlich gewählter Raumpunkt (`DefaultCamera(15,-15,7)`), um die Dreieckswolke überhaupt zu sehen. Dieser kann z.B. im Programmteil Projection::Def( ) initialisiert werden.

```
void Projection::Def( )
{
   if ( FrameNum( ) == 1 )   // Ausgangsposition
   {
     DefaultCamera( 15, -15, 7 );
   }
}
```

Im Programm ist also normalerweise der für die Kalkulation benötigte Augpunkt stets der Kameraposition gleichzusetzen.

Insbesondere gilt dies, wenn sich die Betrachterposition stetig ändert, was prinzipiell für jedes Objekt in *Open Geometry* möglich ist(Menüsteuerung bzw. Tastenkombinationen). Dann muss nämlich der Augpunkt in jedem „Augenblick“ (d.h. in jeder Szene) nachjustiert werden:

```
// in Routine Solve_Formula1

   P3d Eye= TheCamera.GetPosition();
```

3.14 Schattierung

Um realistische Bilder gekrümmter Flächen zu erhalten, muss man die einzelnen Teildreiecke der Flächen schattieren. Die im Zusammenhang mit der Schattierung verwendete Bezeichnung *Lichtquelle* sollte nicht mit der Lichtquelle des Diaprojektors verwechselt werden, sie meint natürlich das globale Beleuchtungszentrum der Szene. Angenommen wir haben nur eine einzige Lichtquelle, die durch Menusteuerung bzw. Tastenkombination beliebig einstellbar ist. Im Programm erhält man ihre aktuellen Koordinaten durch den Aufruf des Befehls `GetLightDir()`. Jedem abzubildenden Dreieck wird nun, abhängig von seiner Lage zur Lichtquelle, ein Schattierungsparameter g zugewiesen. Dieser errechnet sich aus dem skalaren Produkt der einfallenden Lichtstrahlen und dem Normalvektor des Dreiecks. Das Maß der Schattierung ist im Wesentlichen zum Kosinus des Einfallswinkels proportional.

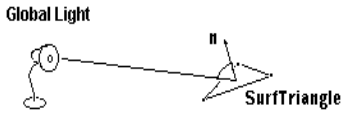


Abbildung 15: Winkel zwischen Lichtstrahl und Normalvektor

Andere Kriterien für die Helligkeit eines Dreiecks sind der durchschnittliche Abstand aller Dreiecke des dargestellten Objekts zur Lichtquelle, sowie deren Distanz zum Auge. Diese sind hier im allgemeinenen Parameter `brightness`, als *globale Helligkeit* inbegriffen zu sehen. Der Wert von `brightness` gibt die unterste Grenze des Schattierungsparameter g an, der sich zwischen diesem Wert und 1 bewegt.

```
Real
```

```
brightness = 0.6,
         g = brightness+fabs(GetLightDir()
         *SurfTriangle.GetNormalizedNormal())*(1-brightness);
```

3.15 Texture Mapping

Unter Texture Mapping versteht man die Abbildung eines Bildausschnittes auf eine ebene Fläche. Die oben beschriebene Methode der kollinearen Dreiecke ist also eine alternative Methode zum üblichen Texture Mapping. Bei den Projektionen soll der Bildinhalt des $\triangle ABC$ in das $\triangle A_2B_2C_2$ (`Delusion`) gesetzt werden (vgl. Abb. 12). Zu diesem Zweck wurde die *Open Geometry* - Funktion `ShadeWithTexture` aus der Klasse `Poly3d` leicht modifiziert, und `MyShadeWithTexture` genannt. Sie verwendet ausschließlich *OpenGL* - Befehle, und kann in optimaler Rechenzeit ein 3 - dimensionales Polygon mit dem Polygonausschnitt einer Bitmapgraphik ausfüllen. Für das betreffende Dreieck `Delusion` wird nun diese Funktion aufgerufen. Als Argument benötigt sie neben dem Schattierungsparameter g die Bitmap `Map1` und die betreffenden Texturkoordinaten `Tex_coord`.
Die Texturkoordinaten geben die Eckpunkte des Polygons an, das den Bildausschnitt der Bitmapgrafik festlegt. Es handelt sich dabei um zweidimensionale „Relativkoordinaten“ :
Liegt ein Punkt P innerhalb der Bitmap, so entspricht seine x - Texturkoordinate genau dem Bruchteil, der sich aus seinem Abstand zum linken Bildrand und der Gesamtbreite des Rahmens ergibt. Analoges gilt für die y - Richtung, wobei der Abstand zum unteren Bildrand und die Gesamthöhe des Rahmens ins Verhältnis zu setzen sind. Die Texturkoordinaten (x, y) eines Punktes P stellen also die „prozentuellen Abstände“ zu den Bildrändern dar ($x, y \in [0, 1]$). Liegt ein Punkt P außerhalb der Bitmap, tritt das bereits auf Seite 13 besprochene Clipping Problem auf.

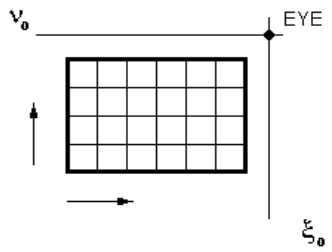
```
StrL3d g1 (TheSlide[1],TheSlide[2]),
       g2 (TheSlide[1],TheSlide[4]);

         Tex_coord[1](g2.Distance(A)/a,g1.Distance(A)/b);
         Tex_coord[2](g2.Distance(B)/a,g1.Distance(B)/b);
         Tex_coord[3](g2.Distance(C)/a,g1.Distance(C)/b);

       Delusion.MyShadeWithTexture(Map1,Tex_coord,(float) g);
```

```
void Poly3d::MyShadeWithTexture(TextureMap &Map,
                                Poly2d &c,float g)
{
  Texture = new TextureInfo;
  AttachMap( Map );
  Texture->map->Activate( false );
  glColor4f( g, g, g, 1 );

  P3d *p = &pPoints[1];
  P3d *q = &c[1];
  glBegin( GL_POLYGON );

   for (i = 1; i <= nPoints; i++, p++, q++ )
   {
    glTexCoord2dv( &q->x );
    glVertex3dv( &p->x );
   }

  glEnd( );
  Texture->map->DeActivate( );
  delete Texture;
  Texture = NULL;
}
```

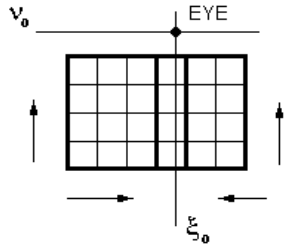
3.16 Sichtbarkeit bei Funktionsgraphen

Der Bildaufbau, also die Reihenfolge in der die einzelnen Dreiecke ausgegeben werden, bestimmt die Sichtbarkeitsverhältnisse einer jeden Szene. Sichtbar sollen klarerweise jene Bereiche sein, die im freien Sichtfeld des Auges liegen.
Ein Dreieck, das vom Betrachter größere Distanz hat als ein anderes, kann niemals dessen Sichtbarkeit beeinflussen, es überdecken. Auf diesem Grundgedanken baut der folgende *Fast Hidden Surface Algorithmus* [2] auf:
Angenommen der Definitionsbereich unseres Funktionsgraphen sei, wie in Methode B (S. 11) ein Rechteck, sodass man den Graphen Γ mit Ebenen ν und ξ, welche normal zur xy - Ebene und parallel zu den Seiten des Definitionsbereichs sind, partitionieren kann. Die Aufgabenstellung lautet nun, den Bildaufbau dieses Graphen so zu gestalten, dass korrekte Sichtbarkeitverhältnisse vorliegen.
Es seien ν_0 und ξ_0 jene zwei Ebenen (Hauptebenen), deren Schnittgerade den Augpunkt enthält. Je nach Lage des Augpunktes zum Definitionsbereich müssen nun 3 verschiedene Fälle unterschieden werden:

1. ν_0 und ξ_0 schneiden den Graphen Γ nicht:

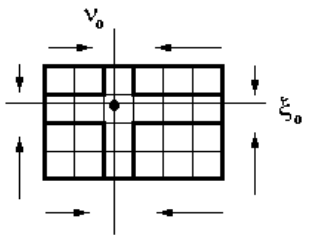


Hier wird der Graph zum Beispiel zeilenweise (die selbe Idee funktioniert analog auch spaltenweise) aufgebaut, d.h. beginnend mit der Zeile, die den größten Abstand von ν_0 hat. In jeder einzelnen Zeile werden dann jene Rechtecke mit größerem Abstand zu ξ_0 zuerst ausgegeben. Jedes Rechteck wird durch eine seiner Diagonalen (egal welche) in zwei Dreiecke geteilt, von denen jenes, dass nicht in der selbe Halbebenen wie der Augpunkt liegt, zuerst ausgegeben werden muss. Da alle Prioritätskriterien zweidimensionale Probleme sind, können sie in schneller Rechenzeit überprüft werden.
2. Eine der beiden Ebenen (ν_0 oder ξ_0) schneidet Γ: Dann wird Γ in die zwei Teile Γ_1 und Γ_2, und in einen zusätzlichen *kritischen Streifen* geteilt. Der kritische Streifen ist jene Zeile oder Spalte, die von ν_0 oder ξ_0 geschnitten wird:

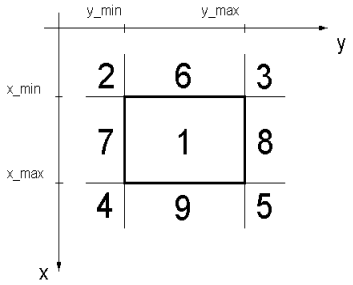


Γ_1 und Γ_2 sind Teilgraphen, die sich nicht überdecken und können getrennt betrachtet, wie im ersten Fall behandelt werden. Erst dannach wird der kritische Streifen abgebildet, wobei auch hier die vom Auge weiter entfernten Rechtecke zuerst gezeichnet werden.
3. Beide Hauptebenen ν_0 und ξ_0 schneiden den Graphen Γ, d.h. das Auge befindet sich also direkt über dem Definitionsbereich. Hierbei wird Γ in vier Teilgraphen Γ_i ($i = 1 \ldots 4$) aufgeteilt. Außerdem entstehen vier kritische Streifen

und ein zusätzliches *kritisches Rechteck*: Die vier Teilgraphen Γ_i sind wiederum



nicht überdeckend und können damit gemäß Fall 1 dargestellt werden. Anschließend werden die vier kritischen Streifen in beliebiger Reihenfolge abgebildet. Dem Auge ferner liegende Rechtecke werden wieder zuerst abgebildet. Das kritische Rechteck wird als letztes ausgegeben.
Um diese drei theoretischen Fälle im Programm zu unterscheiden, muss zunächst in jeder Szene die aktuelle x und y Koordinate der Kameraposition abgefragt werden. Dies geschieht im Programmteil `Draw`, wo mit `if` - Abfragen die Lage des Auges zum Definitionsbereich festgestellt wird. Praktisch gesehen gibt 9 mögliche Lagen:



Nachdem die richtige Lage gefunden ist, wird die Subroutine `plotfield` aufgerufen. Die Argumente dieser Routine sind die x und y Grenzen des auszugebenden Rechteckbereichs, wobei die Reihenfolge der Argumente der Reihenfolge im Bildaufbau entspricht. Auf eine vollständige Programmierung des oben beschriebenen *Fast Hidden-Surface Algorithmus* wurde insofern verzichtet, dass hier kritische Streifen nicht getrennt ausgegeben werden, sondern als Teil eines Teilgraphen verstanden werden. Außerdem wird der Fall 1, wo nur sehr selten Sichtbarkeitsprobleme auftreten, vereinfacht behandelt.

```
void Scene::Draw( )
{
  Real  Eye_x= TheCamera.GetPosition().x,
```

```
Eye_y= TheCamera.GetPosition().y;

if (Eye_x <=x_min && Eye_y <=y_min)       // FALL 2
   plot_field(x_max,x_min,y_max,y_min);

else if (Eye_x <=x_min && Eye_y >=y_max)  // FALL 3
   plot_field(x_max,x_min,y_min,y_max);

else if (Eye_x >=x_max && Eye_y <=y_min)  // FALL 4
   plot_field(x_min,x_max,y_max,y_min);

else if (Eye_x >=x_max && Eye_y >=y_max)  // FALL 5
   plot_field(x_min,x_max,y_min,y_max);

else if (Eye_x <= x_min && Eye_y > y_min && Eye_y< y_max)
   {
   plot_field(x_max,x_min,y_min,Eye_y);   // FALL 6
      plot_field(x_max,x_min,y_max,Eye_y);
}

else if (Eye_x > x_min && Eye_x < x_max && Eye_y <= y_min)
{
   plot_field(x_max,Eye_x,y_max,y_min);    // FALL 7
   plot_field(x_min,Eye_x,y_max,y_min);
}

else if (Eye_x > x_min && Eye_x < x_max && Eye_y >= y_max)
{
   plot_field(x_max,Eye_x,y_min,y_max);   // FALL 8
   plot_field(x_min,Eye_x,y_min,y_max);
}

else if (Eye_x >= x_max && Eye_y > y_min && Eye_y< y_max)
{
   plot_field(x_min,x_max,y_max,Eye_y);   // FALL 9
   plot_field(x_min,x_max,y_min,Eye_y);
}

else
{
   plot_field(x_min,Eye_x,y_min,Eye_y);    // FALL 1
```

```
            plot_field(x_min,Eye_x,y_max,Eye_y);
            plot_field(x_max,Eye_x,y_min,Eye_y);
            plot_field(x_max,Eye_x,y_max,Eye_y);
         }

}
```

Für Projektionen auf Funktionsgraphen, welche die Methode der kollinearen Dreiecke verwenden, wird also die Subroutine `plotfield` verwendet, welche das Bild eines Teilgraphen Γ_i in der richtigen Reihenfolge ausgibt. In der hier angeführten Version von `plotfield` wird Methode B (Seite 11) verwendet:

```
void plot_field(Real x1,Real x2,Real y1,Real y2)
{
  TheCamera.Zbuffer(false);      // Z - buffering deaktivieren

  int sy= Signum(y2-y1),
      sx= Signum(x2-x1);

  Real  dx= 0.2,         // Gitterabstand der Partitonierung
        dy= 0.2;

 for (Real y = y1; sy*y<=sy*y2; y += sy*dy)
 {
  for (Real x = x1; sx*x<=sx*x2; x += sx*dx)
  {
   for (k=0; k<=1; k++)
   {
     Real x_1= x+k*dx,
          y_1= y+k*dx,
          x_2= x+dx,
          y_2= y,
          x_3= x,
          y_3= y+dx;

     SurfTriangle[1]( x_1, y_1, Function1(x_1,y_1));
     SurfTriangle[2]( x_2, y_2, Function1(x_2,y_2));
     SurfTriangle[3]( x_3, y_3, Function1(x_3,y_3));

     StrL3d h0( SurfTriangle[1],L - (V3d) SurfTriangle[1] );
```

```
      A= psi*h0;

      StrL3d i0(SurfTriangle[2],L - (V3d) SurfTriangle[2]);
      B= psi*i0;

      StrL3d j0(SurfTriangle[3],L - (V3d) SurfTriangle[3] );
      C= psi*j0;

      Solve_Formula1(A, B,C, SurfTriangle,Delusion);

      StrL3d g1 (TheSlide[1],TheSlide[2]),
             g2 (TheSlide[1],TheSlide[4]);


       Tex_coord[1](g2.Distance(A)/a,g1.Distance(A)/b);
       Tex_coord[2](g2.Distance(B)/a,g1.Distance(B)/b);
       Tex_coord[3](g2.Distance(C)/a,g1.Distance(C)/b);

        Real
         brightness=0.6,
         g= brightness+fabs(GetLightDir()*
           SurfTriangle.GetNormalizedNormal())*(1-brightness);

       Delusion.MyShadeWithTexture(Map1,Tex_coord,(float) g);
   }
  }
 }
}
```

3.17 Animation und Rechengeschwindigkeit

Eine noch offenstehende Frage ist die nach der Bildanimation. Wie animiert man eine Szenen, also wie bringt man Bewegung in das Bild? Dies geschieht im Programmteil `void Scene::Animate()`, der sozusagen die Steuerzentrale der Animation ist. Im laufenden Programm werden die Routinen `void Scene::Draw()` und `void Scene::Animate()` zyklisch aufgerufen, und erzeugen so Szene für Szene (bzw. Rahmen für Rahmen). Am Ausgabeschirm wird die dabei erreichte Geschwindigkeit im Bildwechsel mit *frames per second (fps)* angezeigt. Um eine Echtzeitanimation zu erreichen sind dazu mindestens 5 *fps* nötig, andernfalls kommt es zu merkbar unstetigen Bildabläufen. Der maßgeblichste Einflussfaktor für die Geschwindigkeit der Bildabfolge ist die Größe der verwendeten Bitmap-

grafiken. Man kann in etwa beobachten, dass eine Verdoppelung der Pixelgröße ($2n \times 2m$ anstatt $n \times m$), beinahe eine Halbierung der Rechengeschwindigkeit zur Folge hat.
Die zur Animation verwendeten Parameter (z.B. Amplitude und Frequenz der Projektionsflächenbewegung, Drehzahl des rotierenden Dias, Lichtquelle des Projektors, Mittelpunkt des Dias, Größen von Projektionskörpern wie Kugelradius, Länge der Würfelkante,...) müssen als globale Variablen definiert (und initialisiert) werden, und können dann in `Scene::Animate()` verändert werden. Das sieht dann z.B. so aus:

```
void Scene::Animate( )
{
  Real f= 0.02,          // Frequenz bzw. Zeitgeschwindikeit
       amplitude_0= 1.5, // Anfangsauslenkung der Projektionsfläche
       n_side     = 2;

  mytime   = mytime+f;   // absolute Zeit

  amplitude= amplitude_0*sin(mytime);
  // zeitliche Auslenkung der Projektionsfläche
  TheSlide.Rotate(StrL3d(L,M-L),n_slide);
  // Drehung des Dias um die Achse ML im Winkel von n_slide Grad
}
```

Dieses Beispiel sollte zeigen wie die Projektion eines um die Achse LM rotierenden Dias auf eine harmonisch schwingende Projektionsfläche animiert wird. Die globalen Variablen `mytime` und `amplitude` werden dann in `Scene::Draw()`, oder in `void Projection::Def()` zur Berechnung der aktuellen Bildszene verwendet.

3.17.1 Bewegter Projektor

Etwas komplizierter ist eine Bewegung des Projektors, also eine Animation der Parameter L und M. Eine Veränderung dieser Größen hat nämlich i.A. eine Nachjustierung des Dias zur Folge, da es stets orthogonal zur Achse $\overrightarrow{LM}$ liegen muss. Die auf Seite 6 beschriebene Positionierung des Dias muss also hier nicht nur anfänglich, sondern in jeder Szene von Neuem vorgenommen werden. Die Positionierung des Dias muss daher vom Programmteil `Scene::Init()` in den Programmteil `Scene::Draw()` verlagert werde. Für jede Szene wird dann das

Dia aus der Initialposition in der xz - Ebene, mittels der jeweils neu zu berechnenden Eulerwinkel in die zu $\overrightarrow{LM}$ orthogonale Lage gedreht.

3.17.2 Bewegter Betrachter

Die Beobachterposition kann in *Open Geometry* jederzeit manuell per Menüsteuerung bzw. Tastenkombinationen problemlos verändert werden. Für eine programmierte Bewegung des Augpunktes, also eine automatische Flugreise durch die Szene, kann im Programmteil `void Projection::Def()` eine beliebige, mathematisch definierte Kurve implementiert werden. Als laufenden Parameter kann man z.B. die in `Scene::Animate()` definierte absolute Zeit `mytime` verwenden:

```
void Projection::Def( )
{
  Real Radius  =30,
  height_z=12;

  DefaultCamera(Radius*sin(mytime),Radius*cos(mytime),height_z);
}
```

Dieses Beispiel beschreibt eine kreisende Bewegung des Beobachters um die z-Achse, wobei die z-Koordinate konstant bleibt. Im Ausgabefenster kann die Animation mit der Tastenkombination *Strg + F* gestartet werden.

3.18 Exakt lösbare Probleme

Wählt man als Projektionsfläche ε eine algebraische Fläche vom Grade kleiner gleich 4, so lassen sich die numerischen Berechnungen exakt, also ohne Näherungsverfahren lösen. Die Berechnung eines Durchstoßpunktes erfolgt dann durch Einsetzen in eine geschlossene Formel, was die Rechenzeit minimiert. Außerdem liefert eine exakte Lösungsformel nicht nur einen, sondern alle möglichen (reelle und komplexe) Durchstoßpunkte eines Lichtstrahls durch die Projektionsfläche, was für die Bestimmung der Sichtbarkeit von großem Wert ist. In der expliziten Form lassen sich solche Fläche als

$$f(x,y) = \sum_{i=0}^{3} c_{0i}x^{4-i} + c_{1i}y^{4-i} + d$$

darstellen.

```
 C[0][0]= 0.01;           // coefficient of x^4
 C[0][1]= 0.1;            // coefficient of x^3
 C[0][2]= 0.2;            // coefficient of x^2
 C[0][3]= 0.5;            // coefficient of x

 C[1][0]= 0.02;           // coefficient of y^4
 C[1][1]= 0.1;            // coefficient of y^3
 C[1][2]= 0.3;            // coefficient of y^2
 C[1][3]= 0;              // coefficient of y

 Real d=2;

Real Function1(Real x,Real y)
{
  sum=0;
  for(l=0;l<=3;l++)
  {
   sum= sum + C[0][l]*pow(x,4-l)+ C[1][l]*pow(y,4-l);
  }
  sum= sum+d;
  return (sum);
}
```

Um die Durchstoßpunkte zu berechnen, wird das Dia gleichmäßig partitioniert (Methode A,Seite 8). Es sei

$$g : \vec{Y(s)} = (Y_1(s), Y_2(s), Y_3(s)) = \vec{L} + s \cdot L\vec{P}_\psi \tag{6}$$

die Gerade, welche durch die Lichtquelle $\vec{L}$ und einen beliebigen, aber fixen Punkt der Diafläche $\vec{P_\psi}$ definiert ist. Gesucht ist wiederum der Schnittpunkt $\vec{P_\varepsilon}$ der Geraden g mit der Projektionsfläche ε. Jede Nullstelle von $\Delta z(s) = Y_3(s) - f(Y_1(s), Y_2(s))$ entspricht dann einer Lösung, d.h. eingesetzt in Gl. 6 einem Schnittpunkt von g und ε.
Die Funktion $z(s)$ ist i.A. eine Funktion 4.Grades :

$$z(s) = \sum_{n=0}^{4} coeff[n]s^n$$

Unter Verwendung der Abkürzung $\vec{X} := \overrightarrow{LP_\psi}$ lauten die Koeffizienten:

$$coeff[0] = \sum_{k=0}^{1}\sum_{l=0}^{3} c_{kl} L_k^{4-l} - L_2 + d$$

$$coeff[1] = \sum_{k=0}^{1}\sum_{l=0}^{3} (4-l) c_{kl} L_k^{3-l} X_k - X_2$$

$$coeff[2] = \sum_{k=0}^{1}\sum_{l=0}^{2} 6/(l+1) c_{kl} L_k^{2-l} X_k^2 - c_{02} X_0^2 - c_{12} X_1^2$$

$$coeff[3] = \sum_{k=0}^{1} (4 c_{k0} L_k + c_{k1}) X_k^3$$

$$coeff[4] = \sum_{k=0}^{1} c_{k0} X_k^4$$

```
X[0]=Pij.x-L.x;         // Pij = P_psi
X[1]=Pij.y-L.y;
X[2]=Pij.z-L.z;

Light[0]=L.x;
Light[1]=L.y;
Light[2]=L.z;

sum=0;

for (k=0;k<=1;k++)
{
 for(l=0;l<=3;l++)
 {
  sum= sum + C[k][l]*pow(Light[k],4-l);
 }
```

```
 }

coeff[0]= sum + d - Light[2];

sum=0;

 for (k=0;k<=1;k++)
 {
   for(l=0;l<=3;l++)
   {
    sum= sum + C[k][l]*(4-l)*pow(Light[k],3-l)*X[k];
   }
 }

 coeff[1]= sum - X[2];

 sum=0;

 for (k=0;k<=1;k++)
 {
   for(l=0;l<=2;l++)
   {
    sum= sum + C[k][l]*6/(l+1)*pow(Light[k],2-l)*Sqr(X[k]);
   }
 }

 coeff[2]= sum - Sqr(X[0])*C[0][2] - Sqr(X[1])*C[1][2];
 coeff[3]= (4*C[0][0]*Light[0]+C[0][1])*pow(X[0],3)+
           (4*C[1][0]*Light[1]+C[1][1])*pow(X[1],3);
 coeff[4]= C[0][0]*pow(X[0],4)+C[1][0]*pow(X[1],4);

 int number_of_RealRoots = ZerosOfPoly4(coeff,zero);
```

Falls alle 4 Nullstellen `zero[0],..,zero[3]` komplex sind, existiert kein Schnittpunkt des Lichtstrahls g mit der Projektionsfläche ε, der Lichtstrahl läuft an der Projektionsfläche vorbei und geht ins Unendliche. In solchen Fällen trifft also nur ein Teil des gesamten Dias auf die Projektionsfläche, und es treten Abbildungsfehler auf (vgl. Dreiecke im Schatten,Seite 16).
Geht man davon aus, dass das ganze Dia abgebildet wird, so existiert immer eine oder mehrere reelle Nullstellen. Man verwendet die kleinste positive Nullstelle, da

diese dem in Projektionsrichtung nächstgelegenen Schnittpunkt entspricht.

4 Spiegelungen

4.1 Übersicht

In Weiterführung des Themas *Projektionen auf gekrümmte Flächen* war es interessant und naheliegend, Spiegelungen von Fotos an gekrümmten Flächen computergraphisch zu untersuchen. Wie die allgemeine, ausführlich behandelte geometrische Theorie (z.B [5] oder [7]) zeigt, existieren nur für sehr wenige, spezielle Flächen exakte Formeln zur Berechnung der Spiegelpunkte. Der reflektierte Punkt **R** (Reflex) auf der Spiegelfläche φ ist schwer zu finden und erfordert bereits in einfachsten Spezialfällen die Lösung einer Gleichung höheren Grades. Z.B. führt die Berechnung des Reflexes eines Punktes für Kugeln und Drehzylinder auf eine algebraische Gleichung 4. Ordnung. Die Formeln und Parameterdarstellungen dafür sind in [3] angegeben.

4.1.1 Ebene Spiegelung

Die ebene Spiegelung kann als Spezialfall von Spiegelungen an beliebigen Flächen betrachtet werden. Ihre Berechnung erfolgt allerdings viel einfacher und schneller als im allgemeinen Fall. Das Spiegelbild eines 3D - Objektes erscheint aufgrund der Gesetze der Optik ($\alpha_{ein} = \alpha_{aus}$) wie das Bild des, an der Reflexionsebene gespiegelten, virtuellen Objektes.

Objekte, die gänzlich hinter dem Spiegel liegen, haben klarerweise kein Spiegelbild. Objekte, die nur teilweise im Halbraum des Augpunktes liegen, werden auch nur teilweise gespiegelt. *OpenGL* bietet die Möglichkeit, eine sogenannte *clipping plane* zu definieren [8]. Diese beliebig definierte Ebene erzeugt zwei Halbräume, von denen nur einer gerendert wird. Definiert man die Spiegelebene als *clipping plane*, so wird nur der gewünschte Teil des Objektes dargestellt.
Ferner ermöglicht die Verwendung von *clipping planes* eine weitere, noch einfachere und schnellere Methode der ebenen Spiegelung: anstatt das ganze Objekt zu spiegeln, kann man auch nur den Augpunkt spiegeln und die Szene von diesem Punkt **E'** aus rendern. Die eventuell störenden Objekte hinter dem Spiegel können durch die Verwendung von *clipping planes* abgeschirmt werden. Anschließend muss das entstandene Spiegelbild nochmals mit dem ursprünglichen Augpunkt **E** gerendert werden.
Eine ausführliche Behandlung ebener Spiegelungen findet sich in [9, 10, 11].

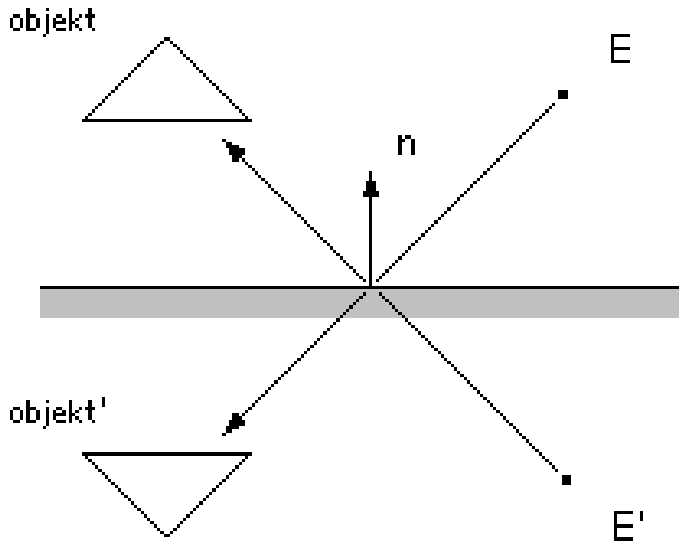


Abbildung 16: Ebener Spiegel

4.1.2 Ray Tracing

Ray Tracing ist eine weitgehend etablierte Methode, um realistische Visualisierungen von 3D - Szenen zu erstellen. Dabei wird durch jedes Pixel$[i, j]$ der Bildebene ein Strahl (*trace ray)* gelegt, der vom Augpunkt **E** ausgehend, den Raum „abtastet“ [12, 13]. Der Schnittpunkt des Strahls mit seinem ersten „Hindernis“ hat bestimmte Oberflächen., Material. und Farbeigenschaften. Der Strahl wird gemäß diesen Parametern absorbiert, oder als reflektierter bzw. gebrochener Strahl im Raum weiterverfolgt. Unter Berücksichtigung der umgebenden Lichtquellen wird letzlich aus allen Informationen ein Farbwert für das Pixel$[i, j]$ ermittelt (vgl. Abb. 17). Diese Methode ist in der Lage photorealistische Bilder bzw. Spiegelungen zu erzeugen, allerdings sind dazu große Rechenzeiten, fern von Echtzeitanimationen, notwendig.

4.1.3 Environment Mapping

Environment Mapping ist eine einfache und schnelle Methode, um Raumspiegelungen an gekrümmten Flächen näherungsweise darzustellen. Die Reflexionsobjekte sollen dabei weit entfernt sein, und der Spiegel sollte sich nicht selbst widerspiegeln. Es gibt mehrere Variationen dieser Technik, von denen die gebräuchlichsten im folgenden kurz erläutert werden. *OpenGL* unterstützt Spherical, Dual-Paraboloid und Cubic Environment Mapping.

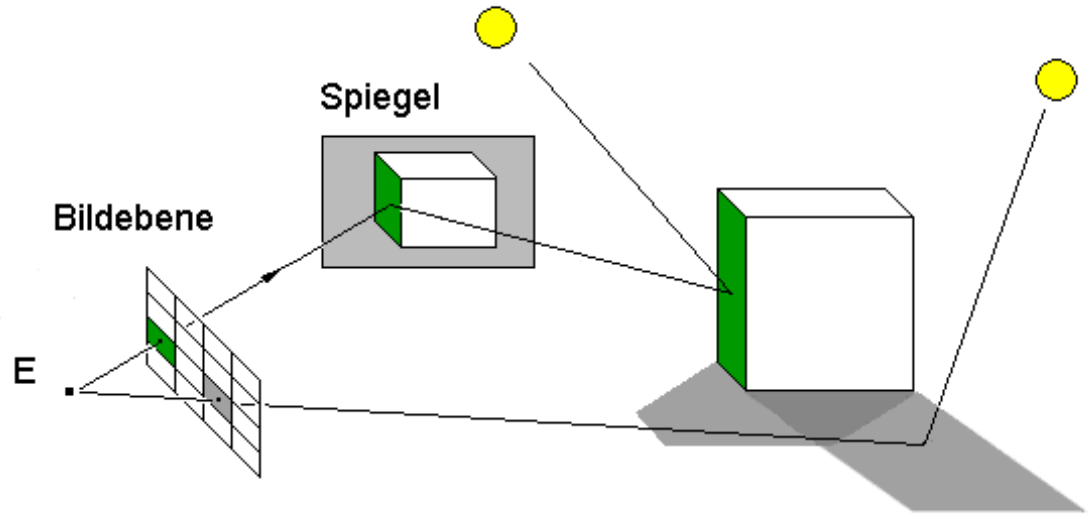


Abbildung 17: Ray Tracing

Bei dieser Methode wird ein 3-D Hilfsobjekt(Würfel, Kugel oder Paraboloid) um den Spiegel gelegt, dessen Oberfläche ein Abbild der Spiegelumgebung darstellt. Der Spiegel reflektiert also nicht die Umgebung selbst, sondern eine animierte Textur auf dem Hilfsobjekt, welche die Umgebung repräsentieren soll.

Beim *Cubic Environment Mapping* [14] befindet sich der Spiegel im Zentrum eines Würfels. Von diesem Zentrum aus erstellt eine virtuelle, 90° geöffnete Kamera Texturen („Fotos“) aller sechs Raumrichtungen. Diese Texturen bilden das Netz des Würfels (*environment map*, Abb. 18 links) und sind unabhängig von der Position des Betrachters, so dass sie nur aktualisiert werden müssen, wenn die Szene sich ändert. Die Aktualisierung ist in Echtzeit möglich, der hohe Speicherbedarf der Texturen belastet jedoch die Rechenzeit. Cubic Environment Mapping ist das flexibelste und gleichzeitig komplexeste Environment Mapping-Verfahren.

Das *Spherical Environment Mapping* [15] verwendet nur eine einzige Textur, um die Umgebungsreflektion zu erzeugen. Um zu verstehen, wie diese Textur entsteht, kann man sich vorstellen, die gesamte Umgebung würde auf die Oberfläche einer Kugel projiziert. Abhängig von der Richtung, in der man in der Kugel schaut, sieht man nun die jeweilige projizierte Umgebung. Wenn man diese Kugel nun flach ausbreitet, bekommt man eine Textur (*environment map*, Abb. 19 links), die die gewünschte Umgebung repräsentiert. Diese Environment-Map kann um dreidimensionale Objekte gelegt werden, um realistische Reflexionen zu erzeugen.

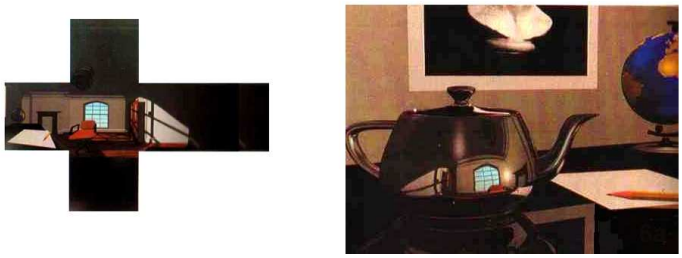


Abbildung 18: Environment Mapping

Abbildung 19: Spherical Mapping

Falls sich die Szene ändert, oder der Spiegel sich bewegt, muss die Umgebungstextur neu erstellt werden. Auch wenn sich der Ort des Betrachters ändert, wird die Umgebung teilweise nicht mehr korrekt reflektiert. Und wie bei jeder anderen Kugelprojektion können sich deutliche Störungen in der Reflexion zeigen, da die Oberfläche einer Kugel sich nur schwierig auf eine ebene Fläche abbilden lässt. Spherical Environment Mapping braucht im Gegensatz zu anderen Verfahren wenig Speicher und funktioniert gut, um Glanzlichter auf unbeweglichen, reflektierenden Objekten darzustellen (Chrome Effekt).

Das *Dual Paraboloid Environment Mapping* [16, 17] ist komplexer als das Spherical-Verfahren und verwendet zwei Texturen als *environment maps* (Abb. 20 links), von denen eine die Umgebung vor dem Objekt und die andere die Umgebung hinter dem Objekt darstellt. Die Texturen sind quadratisch, werden aber mathematisch in die Form einer Halbkugel umgewandelt (Abb. 20 rechts). Der Vorteil des Dual Paraboloid Environment Mapping ist seine Unabhängigkeit vom Standort des Betrachters.
Daher müssen die Spiegeltexturen nicht neu erstellt werden, wenn der Betrachter sich bewegt, wie dies beim Spherical Environment Mapping der Fall ist. Der Nachteil besteht darin, dass die Texturen aufgrund der aufwändigen Umwandlung nicht so schnell erzeugt und aktualisiert werden können, wie dies beim Cubic Environment Mapping der Fall ist. Das Dual Paraboloid Verfahren benötigt weniger

Speicherplatz als das Cubic-Verfahren und eignet sich daher gut für Reflexionen, die nicht dynamisch aktualisiert werden müssen.

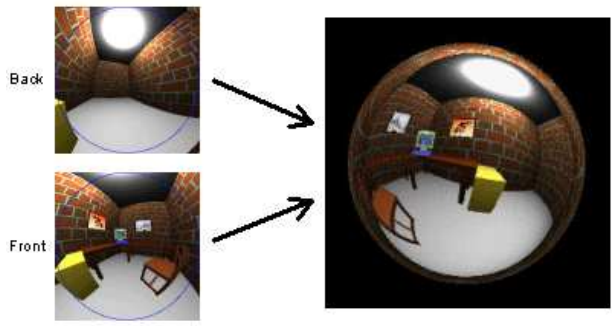


Abbildung 20: Dual Parabolic Environment Mapping

4.1.4 Virtual Object Methode

Eyal Ofek und Ari Rappoport stellten 1998 eine interessante Methode [18] zur Spiegelung von 3D-Objekten an gekrümmten Flächen vor. Im Unterschied zum Environment Mapping, wo lediglich eine animierte Textur widergespiegelt wird, können bei dieser Methode Objekte in der Spiegelnähe realistisch reflektiert werden, auch in stark gekrümmten Spiegelflächen.
Das Spiegelbild eines Objektes entsteht hier, indem es in ein virtuelles Objekt transformiert und gerendert wird. Die Transformation besteht darin, jedem Gitterpunkt Q eines Szenenobjektes einen virtuellen Punkt Q' zuzuordnen. Q' erhält man durch Spiegelung von Q an der Tangentialebene im Reflexpunkt R (Abb. 21). Die eigentliche Aufgabe besteht also darin, eine gute Approximation für den Reflexpunkt R zu finden.

Um dieses Problem zu lösen, verwendet der Algorithmus von Ofek und Rappoport eine Triangulierung auf der Spiegelfläche, d.h. eine Auswahl von Stützpunkten (keine Facettierung des Spiegels).
Legt man Sehstrahlen vom Augpunkt durch die Ecken eines beliebigen, sichtbaren Dreiecks $\triangle V_1V_2V_3$ der Triangulierung, so definieren die drei entstehenden, im allgemeinen windschiefen Reflexstrahlen einen gewissen Raumbereich, eine *reflektierte Zelle* (Abb. 23). Für Ebene Spiegel ist dieser Raumbereich ein Pyramidenstumpf, im allgemeinen Fall wird die reflektierte Zelle von den drei, durch die Reflexionsstrahlen R_1, R_2, R_3 gebildeten bilinearen Regelflächen

$$s(V_i + tR_i) + (1 - s)(V_j + tR_j) \tag{7}$$

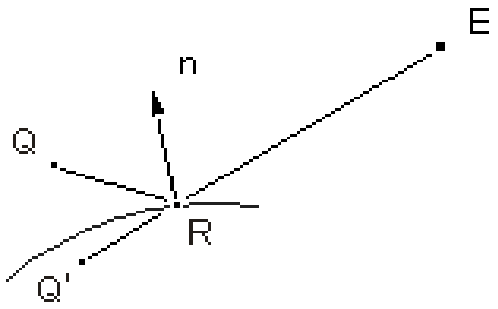


Abbildung 21: Virtueller Punkt Q'

$s \in [0,1], t \geq 0$ und dem Dreieck $\triangle V_1V_2V_3$ begrenzt. Die so definierten Zellen zerteilen also den gesamten Reflexionsbereich (*reflection subdivision*), und jeder Szenenpunkt Q im Reflektionsbereich befindet sich in einer, oder auch in mehreren dieser Zellen (Mehrfachreflex).

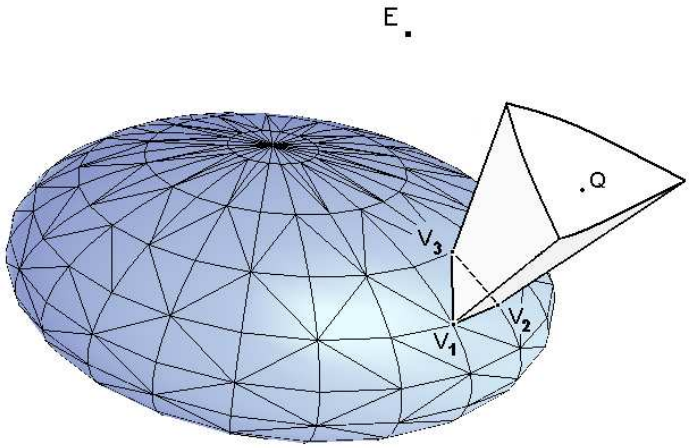


Abbildung 22: Reflektierte Zelle

Mithilfe der sogenannten *Explosion Map Methode* kann man die zu Q gehörige Zelle (bzw. Zellen) sehr effektiv bestimmen und kennt somit das zu Q gehörige Dreieck (Dreiecke) der Spiegeloberfläche. Aus der *relativen Lage* von Q zu

einer ihn umgebenden Zelle, lässt sich ein Tripel von baryzentrischen Koordinaten bestimmen. Diese Koordinaten werden dann zu einer Interpolation der drei Tangentialebenen der Eckpunkte V_1, V_2 und V_3 verwendet, um daraus eine Spiegelebene (bzw. einen Reflexpunkt R) für den Punkt zu erhalten. Eine ausführliche Beschreibung und Diskussion des Algorithmus wird in [18] gegeben.

4.2 Texture Reflection

Mit *Texture Reflection* wird eine Methode vorgestellt, die Spiegelungen von Texturen schnell und beliebig genau approximiert. Die Spiegelung wird, wie bei der *Virtual Object Methode*, durch eine Triangulierung auf der Spiegeloberfläche berechnet. Dadurch ist die Rechenzeit, bei gleicher Bildqualität, deutlich geringer als beim pixelweisen Bildaufbau (siehe Zeitvergleich von Ray Tracing und Textur Reflection).

4.2.1 Szenenaufbau

Nach dem Reflexionsgesetz berechnet sich der einfallende und ausfallende Lichtstrahl folgendermaßen: Gegeben sei die Spiegelfläche ϕ (`mirror`), der Augpunkt **E**, ein Punkt (Reflex) der triangulierten Spiegelfläche **R**, sowie die Lage der Bildebene ψ, in welcher die zu spiegelnde Textur liegt.

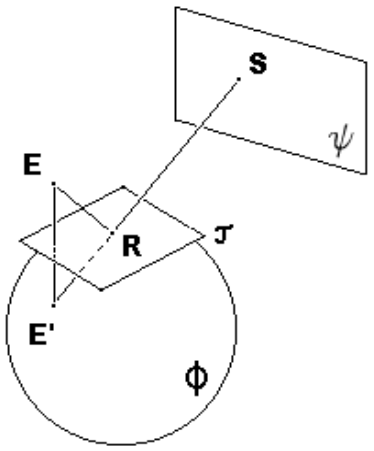


Abbildung 23: Spiegelung des Beobachters

Spiegelt man den Punkt **E** an der im Punkt **R** aufgespannten Tangentialebene τ (`tangential_plane`), erhält man den Punkt **E'**. Der zum Reflex **R** gehörige

Punkt **S** ist der Schnittpunkt der Geraden $X\vec{(}s) = \vec{E'} + s \cdot \overrightarrow{E'R}$ mit der Bildebene ψ.

```
for (i = 0; i< number_x; i++)
{
 for (j = 0; j< number_y;  j++ )
 {
      P3d E=EYE;

      Real  x= x_min+(Real)i*delta_x,  // Parameter vom Reflex
            y= y_min+(Real)j*delta_y;

      Reflex[i][j]= mirror.SurfacePoint(x,y);
      tangent_plane= mirror.TangentPlane(x,y);
      E.Reflect(tangent_plane);                  // E -> E'

      StrL3d reflect(E,Reflex[i][j]);
      S[i][j]= psi_slide*reflect;

      Orient_Eye[i][j]=
           Signum(tangent_plane.OrientedDistance(EYE));
      Orient_slide[i][j]=
           Signum(tangent_plane.OrientedDistance(S[i][j]));
 }
}
```

Mithilfe der beiden Integervariablen `Orient_Eye` und `Orient_slide` lässt sich feststellen, ob der Punkt `S[i][j]` und der Augpunkt `EYE` im selben Halbraum bezgl. der jeweiligen Tangentialebene liegen oder nicht. Diese Abfrage wird später prüfen, ob der ausfallende Strahl die Bildebene ψ *vom Spiegel weglaufend* geschnitten hat, also wirklichkeitegetreu berechnet wurde. Es ist nämlich auch möglich, dass sich der Schnittpunkt **S** hinter der Spiegeloberfläche ergibt, also die Punkte **E** und **S** in verschiedenen Halbräumen (bzgl. τ) liegen.
Auf diese Weise kann für jedes Dreieck der triangulierten Spiegelfläche das zugehörige Dreieck der Bildebene berechnet werden. Das Spiegelbild der Textur ergibt sich, indem man den jeweiligen Bildinhalt eines Dreiecks der Bildebene in das zugehörige Dreieck der Spiegeloberfläche abbildet. Wie in Methode B (Seite 11) kann auch hier das Dreieck der Bildebene teilweise oder gänzlich außerhalb der rechteckigen Texturfläche liegen, und damit die dort beschriebenen Probleme der

Bildabbildung bereiten (Clipping). Um diese Probleme zu vermeiden, könnte man sich wiederum der in Methode B entwickelten Technik des umrahmten Dias bedienen. Im Unterschied zu Projektionen sind die korrespondieren Triangulierungen von Bild- und Spiegelfläche viel stärker gestreut, d.h. zugehörige Dreiecke können schwer abschätzbare Größeunterschiede haben, wodurch Methode B ungeeignet wird.
Im Folgenden wird nun genau diese Problematik (*Reflexionsalgorithmus*) vom Ausschneiden und Spiegeln eines, teilweise außerhalb der Textur liegenden Dreiecks behandelt. Diese Methode sollte nur den tatsächlich innerhalb des Rechtecks liegenden Bildinhalt erfassen (und dessen Texturkoordinaten bestimmen), und auf den entsprechenden Teil des korrespondierenden Dreiecks abbilden. Dies ist zwar ein programmiertechnisch aufwändigerer Weg als Methode B, ist aber letztlich wesentlich praktischer und effizienter. Die Aufgabenstellung lautet folgendermaßen:
Gegeben sei ein Rechteck $S_1S_2S_3S_4$, ein Dreieck ΔRGB (bzw. $\Delta R_mG_mB_m$), und gesucht ist deren Schnittmenge (grüne Fläche), bzw. alle Eckpunkte des, die Schnittmenge umgrenzenden Polygons Γ, und das dazu zugehörige Reflexpolygon Γ_m der Spiegelfläche ϕ.

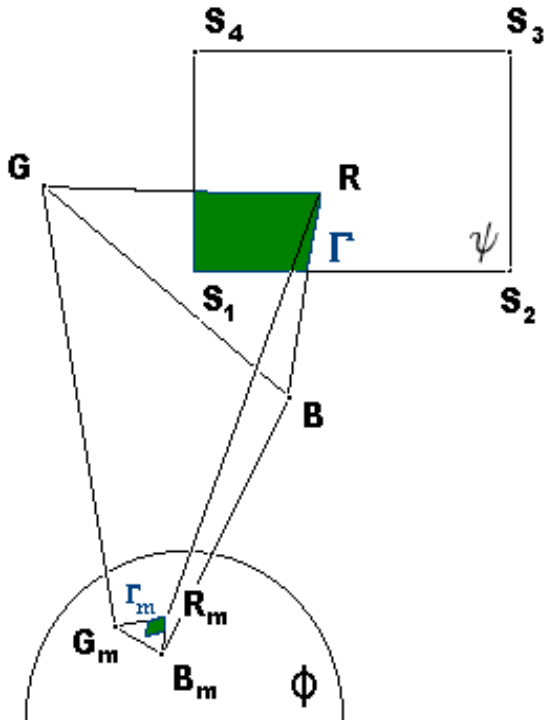


Abbildung 24: Polygon Γ und Reflexpolygon Γ_m

Um die Eckpunkte dieser Polygone systematisch zu finden, muss man sich zunächst Gedanken über die möglichen Lagen des Dreiecks zum Rechteck machen:

4.2.2 Ungeteiltes Dreieck

Liegen alle drei Eckpunkte R, G, B innerhalb des Rechtecks, so wird wie gewöhnlich der ganze Bildinhalt von auf $\Delta R_m G_m B_m$ abgebildet. Der erste Schritt des Reflexionsalgorithmus ist damit die Bestimmung derjenigen Eckpunkte des ΔRGB, die innerhalb des Rechtecks liegen. Dazu werden die drei Eckpunkte der Reihen nach auf ihre Distanz zu den vier begrenzenden Geraden der Textur (`g[0],g[1],g[2],g[3]`) abgefragt. Liegt ein Punkt innerhalb des Rechteckbereichs $a \times b$, so wird er in eine Liste der gesuchten Polygoneckpunkte (`Liste_slide`) aufgenommen, und gleichzeitig die Anzahl der, in dieser Liste befindlichen Punkte (`points`) um 1 erhöht. Der zugehörige Punkt der Spiegelfläche ist bereits bekannt und wird in eine Liste der korrespondierenden Polygoneckpunkte (`Liste_mirror`) aufgenommen.

Falls alle drei Eckpunkte innerhalb des Rechtecks liegen (`points==3`), so werden die entsprechenden Texturkoordinaten (`T[1],T[2],T[2]`) und der Schattierungsparameter `g` berechnet, und das Dreieck `SurfTriangle` mit dem berechneten Bildausschnitt der Bitmapgrafik `Map1` gefüllt.

```
if (g[0].Distance(R)<b && g[1].Distance(R)<a &&
    g[2].Distance(R)<b && g[3].Distance(R)<a)
{
  Liste_slide[points]=R;
  Liste_mirror[points]= R_mirror;
  points++;
}
if (g[0].Distance(G)<b && g[1].Distance(G)<a &&
    g[2].Distance(G)<b && g[3].Distance(G)<a)
{
       Liste_slide[points]=G;
       Liste_mirror[points]= G_mirror;
       points++;
}
if (g[0].Distance(B)<b && g[1].Distance(B)<a &&
    g[2].Distance(B)<b && g[3].Distance(B)<a)
{
```

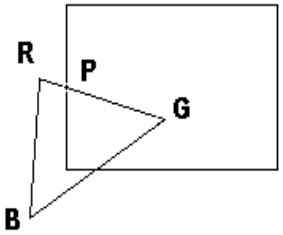
```
        Liste_slide[points]=B;
        Liste_mirror[points]= B_mirror;
        points++;
}


  if (points == 3)
  {
        T[1](g[3].Distance(R)/a,
             g[0].Distance(R)/b);
        T[2](g[3].Distance(G)/a,
             g[0].Distance(G)/b);
        T[3](g[3].Distance(B)/a,
             g[0].Distance(B)/b);

        SurfTriangle[1] = R_mirror;
        SurfTriangle[2] = G_mirror;
        SurfTriangle[3] = B_mirror;

        Real g= bright+fabs(GetLightDir()
             *SurfTriangle.GetNormalizedNormal())*(1-bright);

        SurfTriangle.MyShadeWithTexture(Map1,T,(float) g);

  }
```

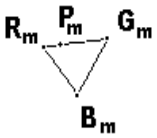
4.2.3 Geteiltes Dreieck

Ist dies nicht der Fall, also `points` < 3, so liegt das ΔRGB entweder gänzlich außerhalb des Rechtecks oder Γ hat noch weiter Eckpunkte. Diese können dann auch Eckpunkte des Rechtecks sein, oder Schnittpunkte des Dreiecks mit dem Rechteck. Zunächst berechnet der Algorithmus alle Schnittpunkte der Trägergeraden der Dreiecksseiten `red,green,blue` mit den Trägergeraden der Rechteckseiten `g[i]` i= 0...3. Das ergibt je vier Schnittpunkte, die in den Listen `Rot[l], Gruen[l], Blau[l]`, mit l = 0 ...3 gespeichert werden. Jeder dieser zwölf Kandidaten ist genau dann ein Eckpunkt von Γ, wenn er zwischen zwei Eckpunkten von ΔRGB und zwischen zwei Eckpunkten des Rechtecks $S_1S_2S_3S_4$ liegt, also tatsächlich auf dem Rechteck und auf dem Dreieck liegt. Die Routine `IsInBetween` prüft ob ein Punkt `P` der Geraden g im Parameterbereich $]t_1, t_2[$ liegt:

```
Boolean IsInBetween(StrL3d g,P3d P,Real t1,Real t2)
{
 Real t_P= g.GetParameter(P);
 Boolean IsIn= false;
 if( t_P > t1 && t_P < t2)
 {
  IsIn= true;
 }
 return(IsIn);
}
```

Es sei also P ein solcher Schnittpunkt, der zwischen zwei Eckpunkten des Dreiecks ΔRGB liegt. Der zugehörige Punkt P_m der Spiegelfläche, liegt dann zwischen den korrespondierenden Eckpunkten des $\Delta R_m G_m B_m$, und soll die Seite auf der er liegt im selben Verhältnis teilen wie in der Texturebene. Z.B. teilt der Punkt P in der unteren Abbildung die Seite $\overline{RG}$ im selben Verhältnis wie P_m die Seite $\overline{R_m G_m}$:

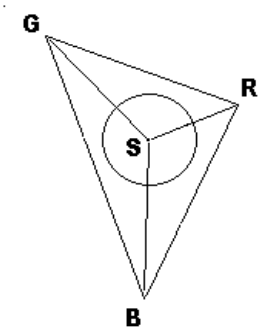




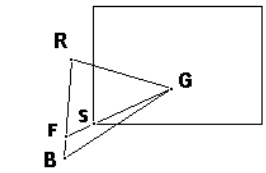
Es gilt also: $P_m = R + \frac{|RP|}{|RG|} * \overrightarrow{R_m G_m}$

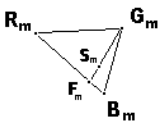
Obiges Teilungsverhältnis ist im allgemeinen nur bei kollinearen Abbildungen von zueinander kollinearen Dreiecken konstant, kann aber ohne Probleme auch für Spiegelungen als solches verwendet werden, da bei einigermaßen feiner Triangulierung kein optischer Unterschied zur Wirklichkeit erkennbar ist.

Weiters müssen die vier Eckpunkte des Rechtecks S_1, S_2, S_3, S_4 auf ihre Lage bzgl. ΔRGB überprüft werden. Liegen sie innerhalb des Dreiecks, so sind sie Eckpunkte von Γ. Um festzustellen ob ein Punkt S innerhalb des Dreiecks ΔRGB liegt, kann man folgendes Kriterium verwenden:
Ist die Summe der drei Winkel, den die Vektoren $\overrightarrow{SR}, \overrightarrow{SG}, \overrightarrow{SB}$ jeweils miteinander einschließen gleich 2π, so liegt S innerhalb, ansonsten außerhalb des ΔRGB:



Etwas schwieriger ist es, für einen innerhalb des Dreiecks liegenden Rechteckseckpunkt S, den zugeordneten Punkt der Spiegelfläche S_m zu finden. Dazu definiert man eine Gittergerade (`StrL3d h`) durch S und beispielsweise durch den Eckpunkt G. Diese Gerade schneidet die Seite $\overline{RB}$ im Punkt F, dessen zugeordneter Punkt F_m sich wie oben beschrieben berechnen läßt:





$$F_m = R_m + \frac{|RF|}{|RB|} * \overrightarrow{R_m B_m}$$

und damit

$S_m = G_m + \frac{|GS|}{|GF|} * \overrightarrow{G_m F_m}$

Insgesamt stehen also neben den drei Eckpunkten des ΔRGB 16 mögliche Kandidaten für die Eckpunkte des Polygons Γ (bzw. Γ_m) zur Auswahl: 12 Schnittpunkte der Trägergeraden des Dreiecks mit dem Rechteck und 4 Eckpunkte des Rechtecks. In einer `for` - Schleife werden der Reihe nach die vier Rechtecksseiten samt ihren Rechteckseckpunkten überprüft: Die Rechteckseiten auf mögliche Schnittpunkte mit den Dreieckseiten, und die Eckpunkte auf ihre Lage zum Dreieck. Erfüllt ein Punkt die angeführten Kriterien, wird er in in die `Liste_slide` aufgenommen, und `points` um 1 erhöht. Höchstens sieben der insgesamt 19 möglichen Kanditaten können Eckpunkte von Γ sein, d.h. Γ kann ein Dreieck, Viereck, Fünfeck, Sechseck oder ein Siebeneck sein. Falls es keinen einzigen passenden Punkt gibt (`points` $= 0$), so liegt das Dreieck gänzlich außerhalb des Rechtecks und braucht nicht weiter beachtet werden.

```
int l;
for(l=0;l<=3;l++)
{
  Rot[l]  = red*g[l];
  Gruen[l]= green*g[l];
  Blau[l] = blue*g[l];

  V3d S_R(TheSlide[l+1],R),
      S_G(TheSlide[l+1],G),
      S_B(TheSlide[l+1],B);

  Real alpha= S_R.Angle(S_G,false)+S_G.Angle(S_B,false)+
              S_B.Angle(S_R,false);

   if(fabs(alpha-2*PI)<1e-10)
   {
    Liste_slide[points]= TheSlide[l+1];
    StrL3d h(G,TheSlide[l+1]);
    P3d Fuss= h*green,
        Fuss_mirror=  R_mirror + (Fuss-R).Length()/
                     (R-B).Length()*(B_mirror-R_mirror);

    Liste_mirror[points]=
     G_mirror+(TheSlide[l+1]-G).Length()/
    (Fuss-G).Length()*(Fuss_mirror-G_mirror);
```

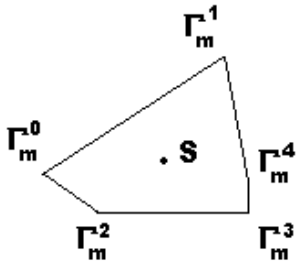
```
 points++;
}

 if (IsInBetween(g[l],Rot[l],0,dimen[l]) &&
     IsInBetween(red, Rot[l],0,(B-G).Length()))
 {
  Liste_slide[points]=  Rot[l];
  Liste_mirror[points]= G_mirror+(Rot[l]-G).Length()/
                        (B-G).Length()*(B_mirror-G_mirror);
  points++;
 }
 if (IsInBetween(g[l],Gruen[l],0,dimen[l]) &&
     IsInBetween(green, Gruen[l],0,(B-R).Length()))
 {
  Liste_slide[points]=  Gruen[l];
  Liste_mirror[points]= R_mirror+(Gruen[l]-R).Length()/
                        (B-R).Length()*(B_mirror-R_mirror);
  points++;
 }
 if (IsInBetween(g[l],Blau[l],0,dimen[l]) &&
     IsInBetween(blue, Blau[l],0,(G-R).Length()))
 {
  Liste_slide[points]=  Blau[l];
  Liste_mirror[points]= G_mirror+(Blau[l]-G).Length()/
                        (R-G).Length()*(R_mirror-G_mirror);
  points++;
 }
}
```

4.2.4 Sortieren von Γ

Bisher wurde keine Rücksicht darauf genommen, in welcher Reihenfolge die Eckpunkte von Γ (bzw. Γ_m) gespeichert wurden. Nachdem also alle Eckpunkte bestimmt worden sind, müssen sie in eine richtige Reihenfolge gebracht werden, z.B. im Gegenuhrzeigesinn geordnet werden. Ansonsten wäre es möglich, dass sich der Polygonzug überkreuzen würde und Fehler bei der Abbildung des Bildinhaltes von Γ auf Γ_m auftreten.
Als Schnittmenge zweier konvexer Figuren (Dreieck und Rechteck) ist auch das Polygon Γ (bzw. Γ_m) konvex, und somit liegt der Schwerpunkt $\mathtt{S}$ des Polygons Γ_m,

$$S = \frac{1}{points} \sum_{n=0}^{points-1} \Gamma_m[n]$$

im Inneren von Γ_m und kann als Zentrum der Polygoneckpunkte betrachtet werden, weil diese quasi um ihn angeordnet sind:



Die Liste `angle[i]` speichert die Winkel (mit gleichgerichtetem Drehsinn) der Vektoren $\overrightarrow{S\Gamma_m^i}$ ($i = 0 \ldots$ `points` $- 1$) zum Vektor $\overrightarrow{S\Gamma_m^0}$ und bildet die Grundlage zum Sortieren der Eckpunkte von Γ_m, die in steigender Reihenfolge ihrer entsprechenden Winkel in eine neue Liste (`sorted_poly`) eingeordnet werden. Außerdem werden die zur Abbildung benötigten Texturkoordinaten des Polygons Γ im 2D - Polygon `TEXT` in gleicher Reihenfolge abgespeichert. Nachdem all dies geschehen ist wird der Schattierungsparameter g berechnet, und das Polygon `sorted_poly` mit dem zugehörigen Bildausschnitt gefüllt.

```
sorted_poly.Def(PureWhite,points);
TEXT.Def(PureWhite,points);

P3d S(0,0,0);

for (l=0;l<points;l++)
{
 S= S+Liste_mirror[l];
}
```

```
S= (Real) 1 / Real (points) * S;

V3d SL0(S,Liste_mirror[0]);
Real angle_min,last_sorted=0,angle[7];

  for(t=0;t<points;t++)
  {
    V3d SLn(Center,Liste_mirror[t]);
    angle[t]= SL0.Angle(SLn,false,true);
    if (angle[t]<0)
    {
     angle[t]= angle[t]+2*PI;
    }
  }

  int index=0;
  sorted_poly[1]= Liste_mirror[0];
  TEXT[1](g[3].Distance(Liste_slide[0])/a ,
          g[0].Distance(Liste_slide[0])/b);

 for(m=2;m<=points;m++)
 {
   angle_min= 3*PI;
   for(n=1;n<points;n++)
   {
      if(angle[n]<angle_min && angle[n]>last_sorted)
      {
        angle_min= angle[n];
        index= n;
      }
   }

      sorted_poly[m]= Liste_mirror[index];
      TEXT[m](g[3].Distance(Liste_slide[index])/a ,
              g[0].Distance(Liste_slide[index])/b);

      last_sorted= angle_min;
 }

Real g= bright+fabs(GetLightDir()*
        sorted_poly.GetNormalizedNormal())*(1-bright);
```

```
sorted_poly.MyShadeWithTexture(Map1,TEXT,(float) g);
```

4.2.5 Vergleich: Ray Tracing / Textur Reflection

Im Folgenden werden die Methoden Ray Tracing und Textur Reflection, hinsichtlich ihrer Bildqualität und Rechenzeit verglichen. Zu diesem Zweck wurden in *POV Ray 3.5 für Windows* mit der Ray Tracing Methode, und in *Open Geometry* mit der Texture Reflection Methode Tests durchgeführt. In beiden Programmen wurden identische Szenen, im gleichen Spiegel, vom gleichen Augpunkt aus betrachtet, und Testbilder erstellt. Um die Qualität der Ergebnisse zu vergleichen ist es notwendig, dass die Objekte auf den Testbildern die gleiche Pixelgröße haben. Die Rechenzeit kann in beiden Programmen abgelesen werden. Da *POV Ray 3.5* nicht speziell für Spiegelungen programmiert wurde, kann man davon ausgehen, dass es schnellere Ray Tracer für diesen Zweck gibt. Der Vergleich sollte daher im Wesentlichen die Größenordnungen der Rechenzeiten gegenüberstellen, und die Stärken und Schwächen der beiden Methoden zeigen.

```
#include "colors.inc"                  // POV Ray - Source

background{ color White }

    camera{orthographic
           location <1, 0.5, -1.0>
           look_at <0, 0, 0>
           angle 60
          }

    light_source {<5, 4, -13> color White shadowless }

    sphere{<0, 0, 0>, 0.25
           texture{ pigment{color Black}
                    finish{specular 0.9 reflection 0.8}
                   }
          }
    box{<0, 0, 0> <1, 1, 1>
        texture { pigment{image_map{png "m.png"}}
       }
```

```
    translate <0, -0.5, 0.3>
}
```

Die Testergebnisse zeigen, dass die Größe der zu spiegelnden Bitmapgraphik darüber entscheidet, welche der beiden Methode leistungsstärker ist. Der Grund dafür ist die Routine `MyShadeWithTexture` (S. 27) in der Texture Reflection Methode. Dieses Phänomen war schon bei den Bildprojektionen (vgl. S. 33) zu beobachten.

Bei der Spiegelung einer Textur mit den Ausmaßen 512×512, sind die Rechenzeiten der beiden Methoden etwa gleich, gemessen an Ergebnissen von gleicher Bildqualität. Die Bildqualität beim Ray Tracing ist durch die Wahl der Auflösung des Ausgabebildes grunsätzlich festgelegt, während sie bei der Texture Reflection Methode davon abhängt, wie fein man die Spiegeltriangulierung wählt.
Die Spiegelung von Texturen kleiner als 512×512 Pixel ist die Texture Reflection Methode dem Ray Tracing überlegen, Texturen bis zu 128×128 Pixel werden in Echtzeit gespiegelt. Da beim Ray Tracing immer die ganze Bildebene gerastert wird, ist die Rechenzeit völlig unabhängig von der verwendeten Textur. Diese Tatsache wird bei der Verwendung großer Texturen zum Vorteil gegenüber der Textur Reflection Methode.

Abbildung 25: Spiegelungen mittels Texture Reflection

Inhaltsverzeichnis

Literatur

[1] G. Glaeser & H.P. Schröcker, *Open Geometry : OpenGL + Advanced Geometry,* Springer, New York, 2002.

[2] G. Glaeser, *Fast Algorithms for 3D - Graphics,* Springer, New York, 1994.

[3] G. Glaeser, *Über die Spiegelung an Kugel und Drehzylinder,* Journal for Geometry and Graphics 1999.

[4] W. Wunderlich, *Über den gefährlichen Rückwärtseinschnitt - Jahresbericht der Deutschen Mathematikervereinigung, Bd.53/Heft 2,* B.G.Teubner, Leipzig, 1943.

[5] W. Wunderlich, *Darbouxsche Verwandtschaft und Spiegelung an Flächen zweiten Grades,* Deutsche Math.7, 1943.

[6] W. Fuhs, *Ebene Schnitte von Pyramiden,* IBDG, 1996.

[7] H. Horninger, *Zur geometrischen Theorie der Spiegelung an krummen Oberflächen,* Sb.Öst.Akad.Wiss., Wien 145, 1996.

[8] T. Hall, *A How to for using OpenGL to Render Mirrors,* comp.graphics.api.opengl newsgroup, August 1996 .

[9] J.P. Diefenbach & N.I. Badler, *Multi-Pass Pipeline Rendering: Realism for Dynamic Environments,* Proceedings 1997 Symposium on Interactive 3D Graphics, pp. 59-70, April 1997, http://www.openworlds.com/employees/paul/index.html.

[10] T. Möller, *Real Time Algorithms and Intersection Test Methods for Computer Graphics,* Ph.D. Thesis, technology, Technical ReportNo.341, Department of Computer Engeneering, Chalmers Universtity of October 1998.

[11] T. McReynolds, D. Blythe, B. Grantham & S. Nelson, *Programming with OpenGL: Advanced Techniques,* Course 17 notes at SIGGRAPH 1998, http://www.sgi.com/software/opengl/advanced98/notes.

[12] T. Whitted, *An improved illumination model for shaded display,* Communications of the ACM 23(6): 343-349, 1980.

[13] A. Glassner, *An Introduction to RayTracing,* Academic Press Inc., London 1989.

[14] N. Greene, *Environment Mapping and Other Applications of World Projection,* IEEE Computer Graphics and Applications vol.6, no.11, pp.21-29, November 1986.

[15] L. Williams, *Pyramidial Parametrics,* Computer Graphics, vol. 7, no. 3, pp. 1-11, July 1983.

[16] W. Heidrich & H.P. Seidel, *View-independent Environment Maps,* Proceedings of the 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware, Lisbon, Portugal, pp. 39-45, August 1998.

[17] W. Heidrich & H.P. Seidel, *Realistic, Hardware- accelerated Shading and Lighting,* Computer Graphics, SIGGRAPH 1999 Proceedings, August 1999, http://www.mpi-sb.mpg.de .

[18] Eyal Ofek & Ari Rappoport, *Interactive Reflections on Curved Objects,* Proceedings of Siggraph: Annual Conference Series 1998 ACM SIGGRAPH pp. 333-342.