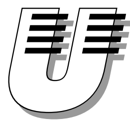


Fachbereich 4: Informatik

Visualisierung relativistischer Effekte

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von

Jan Robert Menzel

Betreuer: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Mai 2006

Erklärung

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

....

(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation 1
1.2 Ziel 1

2 Die spezielle Relativitätstheorie **2**
2.1 Inertialsysteme 2
2.2 Konstanz der Lichtgeschwindigkeit 3
2.3 Addition von Geschwindigkeiten 3
2.4 Längenkontraktion 4
2.5 Dopplereffekt 5
2.6 Suchscheinwerfereffekt 5
2.7 Lichtlaufzeit 6

3 Implementierung **8**
3.1 Die wichtigsten Klassen 8
3.1.1 MainWindow 8
3.1.2 MainLayout 8
3.1.3 Renderer 8
3.1.4 StaticScene 9
3.1.5 Mesh 9
3.1.6 Obj 9
3.1.7 Settings 10
3.2 Dateiformate 10
3.3 Implementierung der Effekte 10
3.3.1 Effekt Pipeline 10
3.3.2 Die Lichtlaufzeit bei konstanter Geschwindigkeit 11
3.3.3 Die Lichtlaufzeit bei variabler Geschwindigkeit 20
3.3.4 Der Suchscheinwerfereffekt 20
3.3.5 Der Software-basierte Renderer 20
3.3.6 Der Vertexshader-basierte Renderer 20
3.4 Die Benutzeroberfläche 22
3.5 Anforderungen an das Modell 23

4 Ergebnisse **26**
4.1 Bilder 27
4.2 Performance 30

5 Ausblick **33**

6 Modellverzeichnis **33**

Abbildungsverzeichnis

1 Zwei Minkowski-Diagramme. 2
2 Visualisierung nach George Gamov 4
3 Das bewegte Objekt wirkt weiter entfernt und länger. 6
4 Das bewegte Objekt wirkt zur Kamera hin verbogen. 7
5 Die Effekt Pipeline 11
6 Das grüne Objekt nähert sich der Kamera (blauer Punkt). 12
7 Zur Lichtlaufzeit 12
8 Der Schnitt mit dem Lichtkegel 13
9 Schnitt eines Vertex mit dem Lichtkegel. 14
10 Die GUI 22
11 Viermal das gleiche Modell bei $0,86c$. 24
12 Fehler durch ungleichmäßig unterteilte Meshes 24
13 Ungünstige Position der Vertizes. 25
14 Löcher durch ungleiche Verzerrungen 25
15 Das Innere eines Würfels bei $0,9c$ 27
16 Der Suchscheinwerfereffekt bei $0,65c$ 27
17 Vorbeiflug am Deutschen Eck mit $0,95c$ 28
18 Links: Start senkrecht nach oben. Rechts: Sturz nach unten 29
19 Würfel mit 24.576 Polygonen 30
20 Markt mit 76.000 Polygonen 31
21 Deutsches Eck mit 276.816 Polygonen 31
22 Spacedock mit 632.802 Polygonen 32
23 Die Spacedock Szene 32

1 Einleitung

1.1 Motivation

Im Jahr 1971 flogen die amerikanischen Wissenschaftler Richard Keating und Joseph Hafele einmal um die ganze Welt, allerdings nicht, um irgendwo anzukommen, es ging ihnen einzig um die Reise selbst. Der Relativitätstheorie zufolge, müsste die Zeit im Flugzeug geringfügig langsamer vergehen als auf der Erde. Um dies zu überprüfen, nahmen die Physiker zwei sechzig Kilo schwere Atomuhren mit und verglichen die verstrichene Zeit mit der auf einer Referenzuhr in Washington. Tatsächlich wichen die Uhren nach dem Flug um wenige Milliardstel Sekunden voneinander ab.

Das Ergebnis des Experimentes war wenig überraschend, die Relativitätstheorie, Anfang des vorigen Jahrhunderts von Albert Einstein formuliert, wurde bereits 1938 experimentell bestätigt [Sch04]. Allerdings zeigt dieser Versuch das wesentliche Problem beim Verständnis von Einsteins Theorie: Die Auswirkungen der Relativitätstheorie können zwar demonstriert werden, doch leider sind ein paar Zahlen auf den Displays einer physikalischen Versuchsanordnung wenig anschaulich.

Deutliche visuelle Effekte treten erst bei sehr hohen Geschwindigkeiten nahe der des Lichtes auf. Da man diese nicht erreichen kann, bleibt nur der Weg der Simulation, um sich ein Bild der Relativität zu machen.

1.2 Ziel

Das Ziel dieser Studienarbeit soll eine Anwendung sein, die es dem Benutzer ermöglicht, sich interaktiv ein Bild der optischen Effekte zu machen, die auftreten, wenn man sich mit Fast-Lichtgeschwindigkeit bewegen würde. Um dem Benutzer eine möglichst einfache Bedienung bereitzustellen, soll die Anwendung über eine GUI verfügen. Die unterschiedlichen Modelle sollen aus Dateien nachgeladen werden, welche möglichst leicht erstellt werden können.

Obwohl das Programm unter Linux entstehen sollte war die Platformunabhängigkeit von Anfang an eines der Ziele dieser Arbeit. Eine Portierung auf Windows XP sollte ohne großen Aufwand möglich sein.

Das Hauptziel der Arbeit sollte die Visualisierung der optischen Verzerrungen sein. Hierbei handelt es sich zum einen um die Längenkontraktion, also eine messbare Verkürzung sich bewegender Objekte, zum anderen um den Einfluss der Lichtlaufzeit. Das heißt, dadurch, dass sich das Licht kaum schneller, als der Betrachter bewegt, spielt auch die die Strecke, die das Lichtes zum Betrachter zurücklegen muss eine Rolle. Dies bewirkt eine unintuitive Verzerrung der Welt.

Obwohl ursprünglich nicht eingeplant, konnte der so genannte Suchscheinwerfereffekt noch implementiert werden. Dieser bewirkt, dass die Welt dem bewegten Betrachter erscheint, als leuchte er sie mit einem Scheinwerfer an. Eine genauere Beschreibung der hier genannten Effekte findet sich in den Kapiteln 2.6 und 2.7.

Die Performance der Simulation steht nicht im Vordergrund dieser Arbeit, dennoch soll die Anwendung interaktiv benutzbar sein.

2 Die spezielle Relativitätstheorie

In diesem Kapitel soll die spezielle Relativitätstheorie in den für die Studienarbeit relevanten Bereichen eingeführt werden. Die Auswirkungen sehr hoher Geschwindigkeiten widersprechen den Alltagserfahrungen, weswegen die Relativitätstheorie nicht intuitiv erfasst werden kann. So kann man Geschwindigkeiten nicht einfach addieren, zudem verkürzen sich Objekte und Uhren gehen langsamer.

Die spezielle Relativitätstheorie basiert auf den Annahmen, dass die Gesetze der Physik in jedem Bezugssystem gleich wirken, und dass die Geschwindigkeit des Lichtes immer konstant ist. Experimentell konnten diese Gesetzmäßigkeiten inzwischen bestätigt werden.

2.1 Inertialsysteme

Unter einem Inertialsystem versteht man ein Bezugssystem, das sich gleichmäßig gradlinig bewegt, also weder Geschwindigkeit, noch Richtung ändert. Es wirken somit keine Kräfte auf das System, wie z.B. eine Beschleunigung. In einem solchen Bezugssystem gelten die physikalischen Gesetzte unabhängig von der Bewegung die das System ausführt.

Als Beispiel soll ein Fahrstuhl dienen: Ein Beobachter im Fahrstuhl kann nicht unterscheiden, ob sich dieser nach oben oder nach unten bewegt. Vorausgesetzt es gibt keine Geräusche und Vibrationen bei der Fahrt, kann nicht einmal unterschieden werden, ob sich der Fahrstuhl überhaupt bewegt oder nicht. Der Unterschied kann weder "bemerkt", noch (mit irgendwelchen Experimenten) gemessen werden, da im Inertialsystem eines ruhenden Fahrstuhls die gleichen Gesetze gelten, wie in einem bewegten. Lediglich bei der Beschleunigungs- und Abbremsphase kann ein Unterschied ermittelt werden, da nun eine Kraft von außen auf den Fahrstuhl wirkt und dieser somit nicht mehr als Inertialsystem angesehen werden kann.

PSfrag replacements

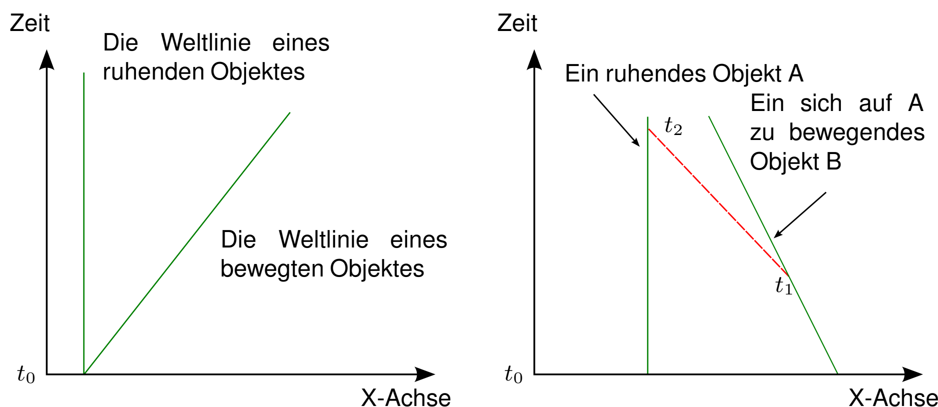


Abbildung 1: Links: Ein einfaches Minkowski-Diagramm mit zwei Objekten. Rechts: Ein Lichtimpuls (rot) wird vom Objekt B zu A geschickt.

Inertialsysteme kann man graphisch anhand von Koordinatensystemen darstellen. Da die Zeit ebenfalls als Raumachse eingetragen wird, handelt es sich

somit genau genommen um vierdimensionale Koordinatensysteme. Zur Erläuterung genügt allerdings ein zweidimensionales, d.h. es wird ein eindimensionaler Raum mit eindimensionaler Zeit betrachtet.

Gewöhnlicherweise stellt man diese Bezugssystem in Form von Minkowski-Diagrammen dar, wobei die X-Achse den Raum und die Y-Achse die Zeit darstellt. Abbildung1 zeigt ein Beispiel. Bewegt sich ein Punkt nicht, so stellt er eine Linie parallel zur Y-Achse im Diagramm dar, bewegt er sich, so wird er als geneigte Gerade wiedergegeben. Diese Linie bezeichnet man als Weltlinie des Punktes. Im Beispiel rechts sieht man ein Objekt B, das sich auf ein ruhendes Objekt zu bewegt. Zum Zeitpunkt t_1 sendet es einen Lichtimpuls los, der zum Zeitpunkt t_2 ankommt.

2.2 Konstanz der Lichtgeschwindigkeit

Wird in einem Zug, welcher sich mit der Geschwindigkeit $v = 0,6c$ bewegt[1], in Fahrtrichtung eine Kugel mit $v = 0,6c$ abgeschossen, so erwartet man, dass ein Beobachter, der sich außerhalb des Zuges befindet, für diese Kugel eine Geschwindigkeit von $1,2c$ misst. Ersetzt man die Kugel durch einen Lichtimpuls, so müsste ein Beobachter auf dem Bahnsteig diesen Lichtimpuls mit $v = 1,6c$ messen. Dies widerspricht aber der Voraussetzung, dass sich der Lichtimpuls immer mit genau Lichtgeschwindigkeit ausbreitet. Das bedeutet, sowohl ein Beobachter A im Inertialsystem des Zuges, wie auch ein Beobachter B im Inertialsystem außerhalb des Zuges müssen den Lichtimpuls mit derselben Geschwindigkeit messen, obwohl sich der Zug bewegt.

2.3 Addition von Geschwindigkeiten

Es wird deutlich, dass man Geschwindigkeiten nicht einfach addieren kann, da das Licht sonst "zu schnell" werden würde. Für die Addition großer Geschwindigkeiten gilt:

$$v_g = \frac{u+v}{1+\frac{u \cdot v}{c^2}} \tag{1}$$

Hierbei sind u und v die zu addierenden Geschwindigkeiten, v_g die resultierende Gesamtgeschwindigkeit und c die Lichtgeschwindigkeit. Setzt man in die Gleichung 1 die Werte für das Beispiel mit der Kugel ein, so erhalten wir:

$$v_g = \frac{0,6c + 0,6c}{1 + 0,6 \cdot 0,6} = 0,88c$$

Für das Beispiel mit dem Lichtimpuls bedeutet dies:

$$v_g = \frac{1c + 0,6c}{1 + 1 \cdot 0,6} = 1c$$

Hier wird auch ersichtlich, dass durch die Addition von Geschwindigkeiten nie eine Geschwindigkeit größer c erreicht werden kann. Diese Formel gilt im Übrigen auch bei geringen Geschwindigkeiten, allerdings kennen wir aus dem Alltag

[1] Die Lichtgeschwindigkeit bezeichnet man in der Regel als c, d.h. 0,6c entsprechen 60% der Lichtgeschwindigkeit.

nur solche, die wesentlich kleiner als 1% der Lichtgeschwindigkeit sind (1% Lichtgeschwindigkeit sind bereits fast $3000\frac{km}{h}$), hier ist der Unterschied zur gewohnten Addition zu klein um ihn wahrnehmen zu können.

2.4 Längenkontraktion

Die Längenkontraktion ist neben der Zeitdilatation einer der bekanntesten Effekte der speziellen Relativitätstheorie. Sie besagt, dass sich Objekte in Bewegungsrichtung zusammenziehen.

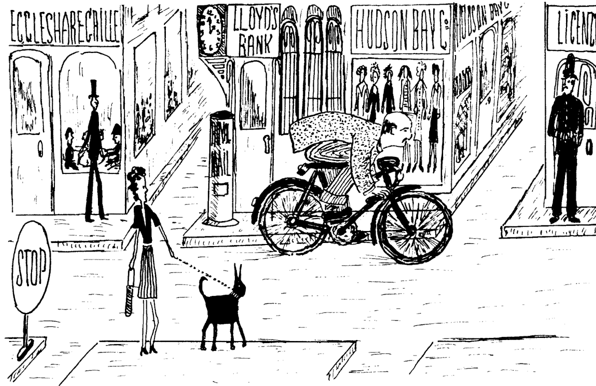


Abbildung 2: Visualisierung nach George Gamov

Bewegt sich ein Raumschiff der Länge $l = 100m$ an einem ruhenden Beobachter A mit einer Geschwindigkeit $v = 0,6c$ vorbei, so misst A, dass das Raumschiff nur 60 Meter lang ist. Der mit-bewegte Beobachter B misst weiterhin eine Eigenlänge von 100 Metern, für ihn ist aber die Welt verkürzt. Dies lässt sich mit der Voraussetzung der speziellen Relativitätstheorie erklären, dass in jedem Inertialsystem die gleichen Gesetzte herrschen: Aus der Sicht von B bewegt sich die Welt mit $0,6c$ an ihm vorbei, daher muss sie verkürzt werden.

Die zu einem Objekt der Länge l kontrahierte Länge l_k, die in einem relativ zum Objekt bewegten Inertialsystem gemessen wird lautet:

$$l_k = l \cdot \sqrt{1 - \beta^2} \tag{2}$$

Wobei β die Relativgeschwindigkeit $\frac{v}{c}$ der Bezugssysteme darstellt. Die Verkürzung tritt dabei nur in Bewegungsrichtung auf.

Obwohl auch dieses Phänomen wenig intuitiv ist, lässt es sich am leichtesten visualisieren. Oftmals ist dies sogar der einzige Effekt, der berücksichtigt wird. So zum Beispiel im Buch "Mr. Tompkins' seltsame Reisen durch Kosmos und Mikrokosmos" von George Gamov [Gam80]. Abbildung 2 zeigt, wie sich Generationen die Auswirkungen der Relativitätstheorie vorgestellt haben, in Wirklichkeit ist die hier gezeigte Längenkontraktion aber nur ein kleiner Teil der beobachtbaren Effekte.

2.5 Dopplereffekt

Der Dopplereffekt wurde zuerst vom österreichischen Physiker Christian Doppler 1842 beschrieben. Dieser Effekt bewirkt eine Frequenzänderung bei Wellen, wenn sich der Sender relativ zum Empfänger bewegt. Als akustischer Dopplereffekt ist er aus dem Alltag bekannt, so ändert sich der Ton einer Sirene beim schnellen Vorbeifahren eines Einsatzfahrzeuges. Nähert sich das Fahrzeug, so ist der Ton höher, entfernt es sich, so sinkt die Tonhöhe ab.

Analog hierzu gibt es einen optischen Dopplereffekt, der die Farbe eines Objektes verändert. Da Licht als Welle aufgefasst werden kann[2], hat es auch analog zum Schall eine Wellenlänge, die als Farbe wahrgenommen wird. Das Verhältnis von empfangener Wellenlänge λ_E zur gesendeten Wellenlänge λ_S berechnet sich in Abhängigkeit zur Geschwindigkeit v wie folgt:

$$\begin{aligned} \frac{\lambda_E}{\lambda_S} &= \sqrt{\frac{1+\beta}{1-\beta}} \\ \beta &= \frac{v}{c} \end{aligned} \tag{3}$$

2.6 Suchscheinwerfereffekt

Der so genannte Suchscheinwerfereffekt beschreibt das Phänomen, dass dem bewegten Beobachter die Objekte direkt vor ihm heller erscheinen. Es sieht so aus, als leuchte er mit einem Suchscheinwerfer vor sich. Stark vereinfacht kann man sich den Effekt an folgendem Vergleich klar machen: Ein Beobachter A steht bei Regen auf der Erde und wird nass. Ein zweiter Beobachter B bewegt sich mit konstanter Geschwindigkeit vom Boden in Richtung Wolken. Beobachter B benötigt nun die Zeitspanne Δt um die Wolken zu erreichen und sammelt so die Regentropfen ein, die in der Zeit Δt die Wolken verlassen und zusätzlich jene, die sich im freien Fall auf die Erde befanden. Beobachter A wird nur von der Anzahl Tropfen getroffen, die während Δt die Wolken verließen. Diese Überlegung lässt sich nun auch auf die Photonen übertragen, die von einer Lichtquelle ausgesendet werden. Je mehr Photonen pro Zeit auf die Kamera treffen, desto heller wird das Bild.

Beim Suchscheinwerfereffekt spielen zudem auch noch der optische Dopplereffekt und die Zeitdilatation eine Rolle. In dieser Arbeit wird eine etwas vereinfachte Form verwendet, die den Dopplereffekt nicht berücksichtigt. Eine detaillierte Beschreibung und Herleitung ist bei [WKR99] zu finden.

$$L_E = D^4 L_S \tag{4}$$

$$D = \frac{\sqrt{1-\beta^2}}{1-\beta cos\phi} \tag{5}$$

Wobei β für $\frac{v}{c}$, L_E für die empfangene und L_S für die gesendete Beleuchtungsstärke steht.

[2] Licht hat sowohl die Eigenschaften einer Welle, als auch die eines Teilchens.

2.7 Lichtlaufzeit

Das Bild, das wir sehen, setzt sich aus einzelnen Lichtstrahlen zusammen, welche von den betrachteten Objekten ausgesandt bzw. reflektiert werden. Da das Licht eine endliche Geschwindigkeit hat, braucht es von unterschiedlich weit entfernten Objekten auch unterschiedlich lange um zu uns zu gelangen. Objekte, die weit vom Betrachter entfernt sind, "sehen" wir also in einem bereits vergangenen Zustand. Die Sonne sehen wir zum Beispiel dort, wo sie vor rund $8\frac{1}{3}$ Minuten war, da das Licht von der Sonne eben diese $8\frac{1}{3}$ Minuten zur Erde benötigt.

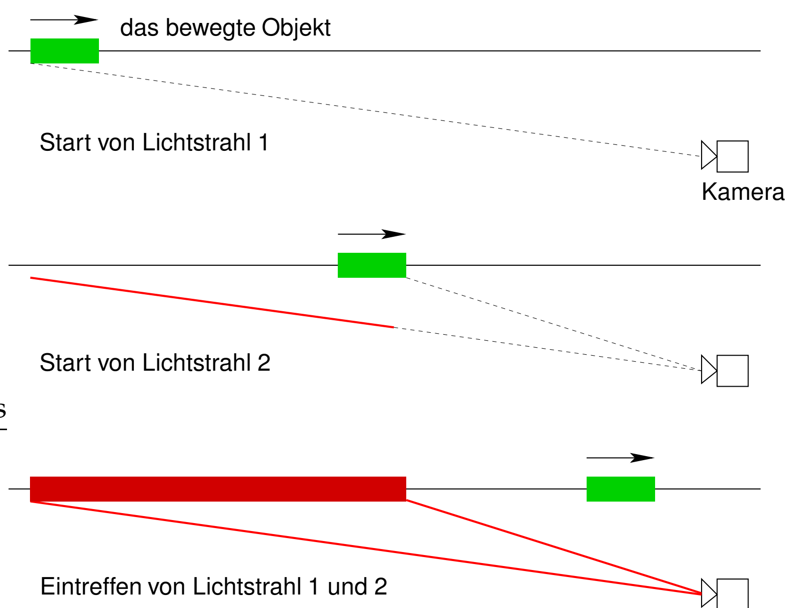


Abbildung 3: Das bewegte Objekt wirkt weiter entfernt und länger.

Die Laufzeit des Lichtes ist aber nicht nur bei großen Entfernungen, sondern auch bei hohen Geschwindigkeiten ein zu berücksichtigender Faktor. Ein schnell bewegter Beobachter wird folgende Unterschiede zur erwarteten Wahrnehmung feststellen: Zum einen wirken Objekte weiter entfernt, als sie eigentlich sind, des weiteren wirken sie lang gezogen und zu guter Letzt wölben sich Flächen zum Beobachter hin, obwohl sie flach sein sollten.

Folgende Abbildungen sollen diese drei Phänomene erklären. In Abbildung 3 bewegt sich ein Objekt nahe der Lichtgeschwindigkeit auf den Beobachter zu. Die Position des Objektes ist grün markiert, der Beobachter sieht es allerdings an einer anderen Position, da das Licht kaum schneller ist als das Objekt selbst (hier rot dargestellt).

Abbildung 4 zeigt, warum ein gerader Stab gekrümmt aussieht: Das Licht von den äußeren Enden des Objektes benötigt länger zum Betrachter, als das Licht, das direkt von vorne kommt. Durch die schnelle Bewegung des Objektes auf den Betrachter zu sieht er den Stab gleichzeitig in unterschiedlichen Stadien seiner Be-

wegung[3]. Auch hier ist die reale Position des Objektes grün dargestellt, die scheinbare Position, an der der Betrachter das Objekt sieht, ist rot eingezeichnet.

PSfrag replacements

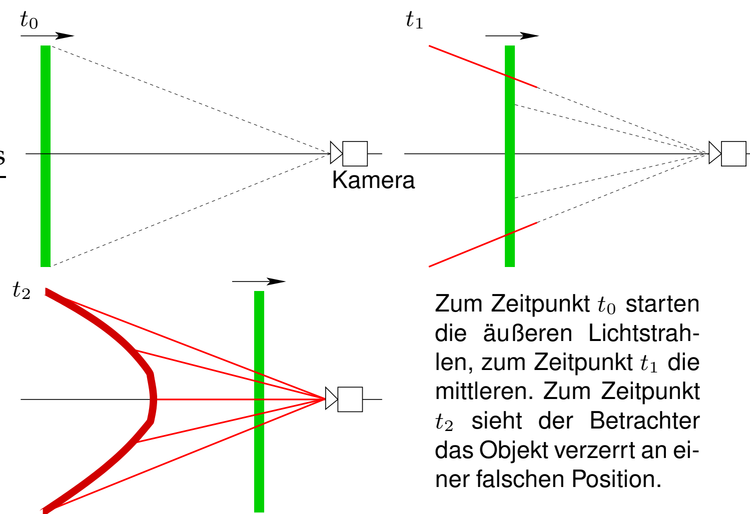


Abbildung 4: Das bewegte Objekt wirkt zur Kamera hin verbogen.

[3] Dies ist ein rein optischer Effekt auf Grund der endlichen Geschwindigkeit des Lichtes und hat nichts mit dem relativistischen Effekt der Zeitdilatation zu tun.

3 Implementierung

Das Programm entstand in der Programmiersprache C++ unter Linux. Für die GUI wurde QT 4 von Trolltech verwendet. Diese GUI Bibliothek kann kostenlos verwendet werden, solange keine kommerziellen Applikation damit entwickelt wird. Sie steht unter anderem für Linux, MacOS X und Windows zur Verfügung. Das Rendering erfolgt in OpenGL und kann als Widget in einem QT Fenster realisiert werden.

Die Entscheidung fiel auf diese Kombination, da mit OpenGL und C++ ein ausreichend performantes Programm umgesetzt werden kann. Die Bibliothek QT genügt darüber hinaus den Ansprüchen der Portabilität.

Im Verlauf der Studienarbeit kam ein Vertexshader hinzu, dieser war anfangs nicht eingeplant, ließ sich aber problemlos integrieren.

3.1 Die wichtigsten Klassen

Die folgende Auflistung beschreibt nicht alle in dieser Arbeit implementierten Klassen, sondern nur die, die für das Verständnis wichtig sind.

3.1.1 MainWindow

Die Klasse *MainWindow* ist die eigentliche Anwendung und zugleich das Fenster in welchem diese angezeigt wird. *MainWindow* wird von der Klasse *QMainWindow*, die von QT bereitgestellt wird, abgeleitet und enthält die Menüs und Toolbars. Im Hauptbereich des Fensters wird nur ein Widget erstellt, dieses ist vom Typ *MainLayout* (s.u.).

3.1.2 MainLayout

Die Klasse *MainLayout* wurde von der QT Klasse *QWidget* abgeleitet und kümmert sich um die Anordnung und Anbindung der weiteren GUI-Elemente. Das heißt, diese Klasse ist dafür verantwortlich, dass ein Klick auf einen Button auch die richtige Aktion im Programm auslöst.

Links wird die Anzeige der Szene positioniert, rechts die Checkboxes und Slider, mit denen das Programm gesteuert werden kann. Die Benutzereingaben werden zunächst an die Klasse *Settings* weitergereicht.

3.1.3 Renderer

Die von der Klasse *QGLWidget* abgeleitete Klasse *Renderer* kümmert sich um die Initialisierung von OpenGL und der Interpretation von Benutzereingaben. Diese können von der Tastatur kommen (optionale Steuerung mit den Pfeiltasten) oder von der Maus (Blickrichtung ändern durch gedrückt halten der linken Taste und bewegen der Maus bzw. bewegen nach vorn mit Hilfe der rechten Maustaste).

Der *Renderer* enthält eine Szene, an die diese Informationen weitergereicht werden. Die Berechnung der Frames pro Sekunde, sowie das regelmäßige Aufrufen einer Renderingfunktion der Szene übernimmt ebenfalls diese Klasse. Hier wird entschieden, welche Renderingfunktion aufgerufen werden soll.

3.1.4 StaticScene

Die Klasse *StaticScene* enthält das Modell einer statischen Szene als Objekt der Klasse *Mesh*. Statisch bezeichnet hier eine Szene, in der sich selbst nichts bewegt. Diese Klasse kümmert sich im wesentlichen darum, dass das Modell korrekt gerendert wird.

Zusätzlich zum Modell werden die Kameraposition und -ausrichtung bereitgestellt. Die Kamera wurde als Klasse modelliert und ist innerhalb der Klasse *StaticScene* ein `public` Attribut, da es durchaus erwünscht ist, dass die Kameraposition etc. von außen (in diesem Fall von der Klasse *Renderer*) manipuliert wird.

Soll nun die Szene dargestellt werden, wird eine von zwei Renderingfunktionen in *StaticScene* aufgerufen. Es kann gewählt werden, zwischen einer reinen Softwareimplementierung der Verzerrungen (`renderGL_software()`) [4] und einer Version als Vertexshader (`renderGL_hardware()`). Letztere verfügt über mehr Features und ist in der Regel schneller.

Soll die Anwendung um weitere Features wie zum Beispiel dynamischen Szenen oder anderen Renderingansätzen erweitert werden, so muss diese Klasse erweitert, bzw. ersetzt werden. Abhängig vom zu ladenden Dateiformat kann die Instanz vom *Renderer* dann ein Objekt dieser oder der neuen Klasse erstellen und gegebenenfalls die Instanz der Klasse *MainLayout* anweisen weitere Bedienelemente bereitzustellen.

3.1.5 Mesh

Das Modell der Szene wird in einem Objekt der Klasse *Mesh* gespeichert. Dieses Modell kann aus einer Obj-Datei geladen werden, alternativ aber auch aus einem selbst definierten Dateiformat (siehe Kapitel 3.2).

Die Szene wird intern in einem `std::vector` von Polygonen gespeichert, wobei hierzu eine *MyPolygon* Klasse definiert wurde, die schlicht die drei Vertizes (als Klasse *Point* modelliert) enthält. Jedes Objekt der *Point* Klasse speichert den jeweiligen Farbwert. Auf dieser Datenstruktur wird gearbeitet, wenn das Mesh geladen wird, oder zur Laufzeit verfeinert werden soll.

Alternativ stellt die Klasse *Mesh* die Geometrie des Modells als `float[]` Array zur Verfügung. Die Methode `renderGL_software(void)` der *StaticScene* Klasse arbeitet aus Performancegründen mit dieser Datenstruktur. Des weiteren kann das Modell auch als Displayliste zurückgegeben werden, was von der Methode `renderGL_hardware(void)` genutzt wird.

Hauptaufgabe der *Mesh* Klasse ist somit, die unterschiedlichen Versionen des Modells auf dem gleichen Stand zu halten.

3.1.6 Obj

Die *Obj* Klasse liest eine Datei im Obj-Format aus und stellt die Geometrie und Farbwerte als `std::vector` mit Elementen der Klasse *MyPolygon* zur Verfügung. Es können bisher noch nicht alle Informationen dieses Formates interpretiert werden. So werden zum Beispiel nicht alle Materialeigenschaften geladen. Unbekannte Informationen führen allerdings nicht zum Programmabbruch, sondern werden einfach ignoriert.

[4] Das Rendering erfolgt allerdings über OpenGL.

3.1.7 Settings

Um die Einstellungen des Programms zwischen den einzelnen Elementen auszutauschen, gibt es die Klasse *Settings*. Diese speichert die allgemeinen Optionen (ob der Vertexshader genutzt werden soll, welche Farbe das Wireframe hat etc.) und kümmert sich darum, dass bei Änderungen alle relevanten Objekte informiert werden. Damit alle anderen Klassen bequem auf diese Einstellungen zugreifen können, wurde sie als Singleton implementiert.

Wird zum Beispiel eine Checkbox der GUI aktiviert, so wird diese Änderung dem Settingsobjekt mitgeteilt. Dieses ruft dann die Objekte auf, die an dieser Änderung interessiert sind. Hierfür registrieren sich alle Objekte beim Settingsobjekt. Dies geschieht über die von QT bereitgestellten `Slots` und `Signals`.

3.2 Dateiformate

Am Anfang der Entwicklung wurde ein möglichst einfaches Dateiformat für das Modell definiert (`.mesh` Dateien). Es sollte leicht zu erstellen und zu laden sein, daher enthält es zunächst die Anzahl der Polygone als unsigned integer, darauf folgend für jedes Polygon die X- ,Y- und Z-Koordinate als float und den Farbwert im RGB Farbraum als unsigned char.

Da die Konvertierung von Modellen, die in anderen Formaten vorlagen sich als zu aufwändig erwiesen hatte, kam eine Unterstützung des Obj-Formates hinzu. Dieses wird von einigen 3D Modelling Programmen direkt unterstützt.

Beide Formate können gelesen werden, zudem kann ein Modell als `.mesh` Datei abgespeichert werden. Dies kann nützlich sein, da Dateien in diesem Format wesentlich kleiner sind als im Obj-Format.

3.3 Implementierung der Effekte

Die in dieser Studienarbeit beschriebene Implementierung berücksichtigt die Längenkontraktion, die Lichtlaufzeit und den Suchscheinwerfereffekt. Letzterer ist standardmäßig deaktiviert, da er sonst die anderen Effekte dominiert.

3.3.1 Effekt Pipeline

Da die verwendeten Algorithmen in beiden Versionen gleich sind, sollen diese hier zunächst behandelt werden, bevor auf die Details der Implementierung auf der CPU bzw. der GPU eingegangen wird. Jedes Vertex der Szene durchläuft folgende Pipeline:

- Beleuchtung[5]
- Transformation in das Kamerakoordinatensystem
- Anwendung der Längenkontraktion
- Anwendung der Lichtlaufzeit
- Berechnung des Suchscheinwerfereffektes

[5] In der Software-basierten Version ist dies bisher nicht implementiert.

Abbildung 5 zeigt die gleiche Szene in unterschiedlichen Stadien der Pipeline. Der Suchscheinwerfereffekt wirkt hierbei implementierungsbedingt nicht auf den blauen Hintergrund.

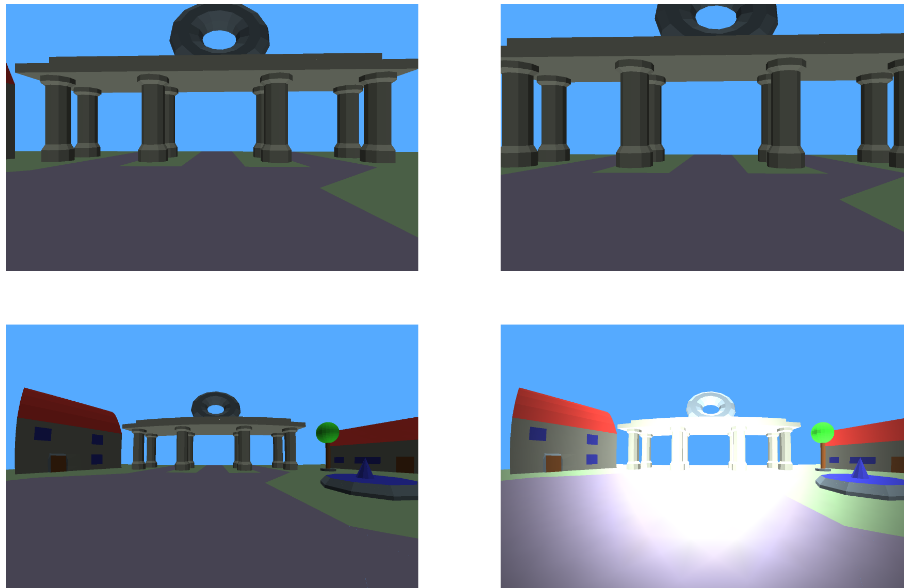


Abbildung 5: Die Effekt Pipeline: Von oben links nach unten rechts: Das beleuchtete Modell, Längenkontraktion, Lichtlaufzeit und Suchscheinwerfereffekt

Dadurch, dass die relativistischen Effekte nur auf die einzelnen Vertizes angewendet werden, ergeben sich besondere Anforderungen an die Geometrie der Szene. Diese werden in Kapitel 3.5 genauer behandelt.

Die Beleuchtung dient nur einem realistischeren Eindruck der Szene und soll hier nicht näher behandelt werden.

Eine Transformation ins Kamerakoordinatensystem erfolgt bereits am Anfang der Pipeline, da sich hierdurch die folgenden Schritte vereinfachen.

Die Längenkontraktion erfolgt in Bewegungsrichtung des Beobachters durch Anwendung der Formel (2) auf Seite 4. Da sich der Betrachter nach der Transformation der Welt ins Kamerakoordinatensystem entlang der Z-Achse bewegt, muss lediglich die Z-Koordinate transformiert werden.

3.3.2 Die Lichtlaufzeit bei konstanter Geschwindigkeit

Der Einfluss der Lichtlaufzeit ist etwas aufwändiger zu berechnen. Zum besseren Verständnis soll hier zunächst der besser vorstellbare Fall eines zweidimensionalen Raumes betrachtet werden. Der Beobachter soll hierbei als ruhend und die Szene als bewegt angesehen werden[6]. Die Abbildung 6 zeigt die Szene zu den Zeitpunkten t_0 bis t_5. Legt man diese nun übereinander, so gelangt man zur Dar-

[6]Wenn sich der Beobachter durch eine unbewegte Szene bewegt, gelangt man zum gleichen Ergebnis, allerdings ist der hier vorgestellte Fall leichter nachzuvollziehen.

stellung 7. Die dritte Dimension ist für uns normalerweise eine räumliche Dimension, hier stellt sie die Zeit dar.

PSfrag replacements

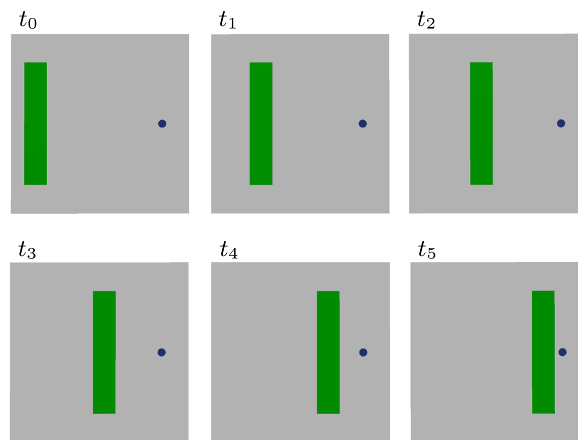


Abbildung 6: Das grüne Objekt nähert sich der Kamera (blauer Punkt).

Man kann somit die zwei Raumdimensionen zusammen mit der Zeit als drei Raumdimensionen darstellen (Abbildung 7 rechts). Nun muss das Licht, das von dem Objekt zur Kamera gelangt, durch die Zeit zurück von der Kamera zum Objekt verfolgt werden. Ganz ähnlich verfolgt ein Ray-Tracer das Licht von der Kamera durch die Szene zu den Objekten. Die rechte Darstellung in Abbildung 7 zeigt den Kegel, der alle möglichen Lichtstrahlen darstellt, die von der Szene zur Kamera gelangen können.

PSfrag replacements

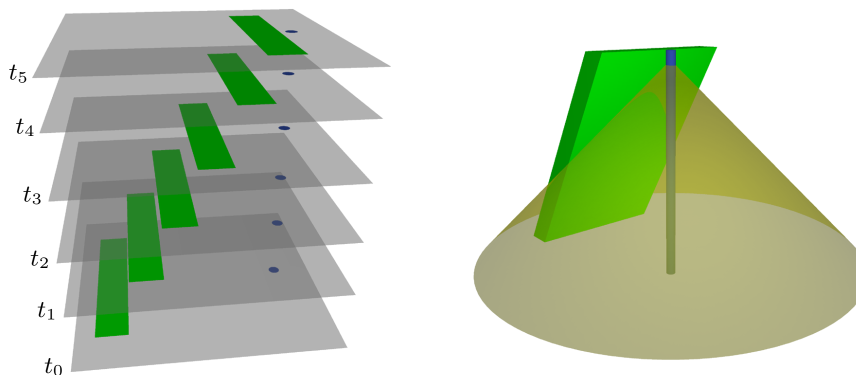


Abbildung 7: Links: Die Bilder aus Abb. 6 übereinander. Rechts: Der Vorgang als 3D Raum mit Lichtkegel.

Schneidet man nun die dreidimensionale Raumzeit der Szene mit der Kegeloberfläche, so bekommt man einen zweidimensionalen Raum. Dieser Raum enthält die Objekte zu den Zeitpunkten, an denen sie Licht ausgesendet hätten müssen,

damit die Kamera sie jetzt sehen kann. Somit kann dieser Raum direkt dargestellt werden.

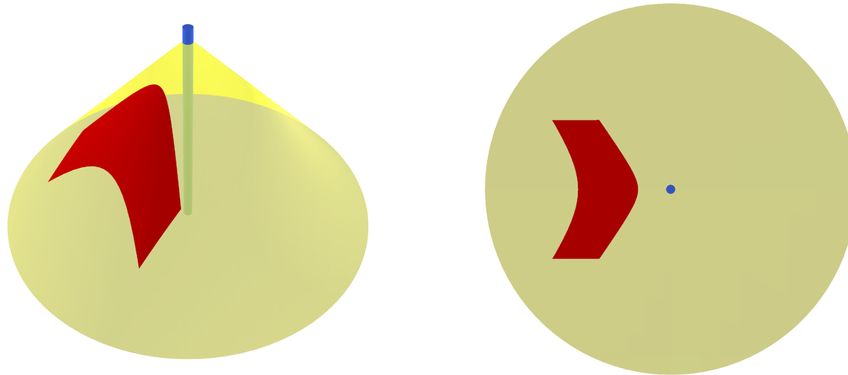


Abbildung 8: Links: Der Schnitt mit dem Lichtkegel. Rechts: Der Schnitt als 2D Raum, wie er dem Betrachter zum Zeitpunkt t_5 erscheint. Die Geometrie des Objektes ist aufgrund der Lichtlaufzeit deutlich verzerrt.

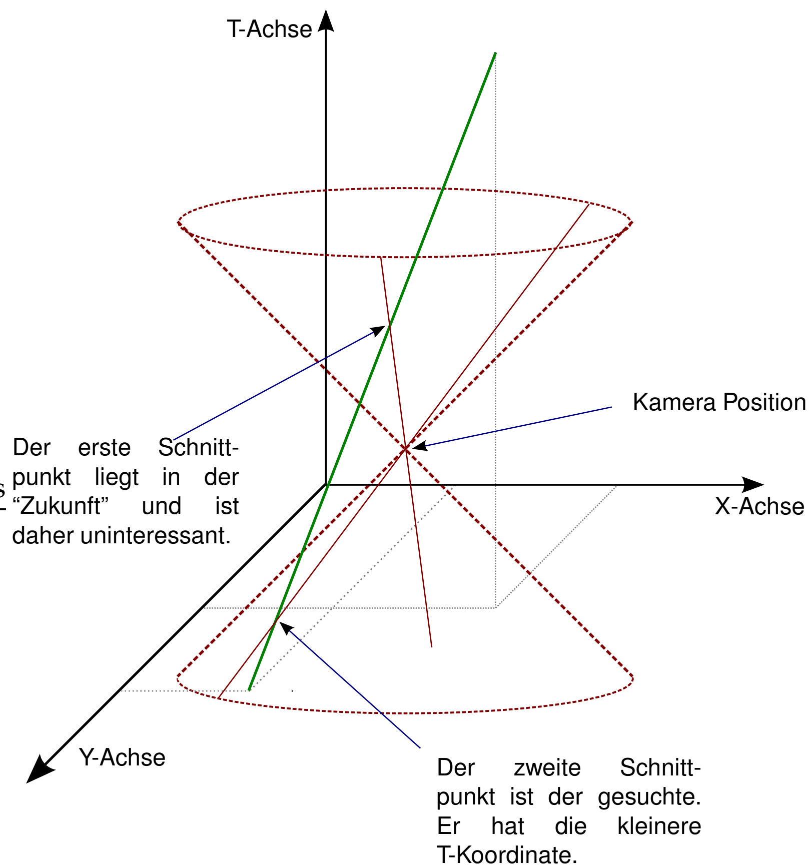
Es ließe sich nach diesen Überlegungen leicht ein vierdimensionaler Ray-Tracer entwickeln, indem man nur einen Teil des Kegels betrachtet, diesen mit einzelnen Strahlen annähert und jeweils nur den vordersten Schnittpunkt mit diesen Strahlen berücksichtigt. Aus Performance Gründen wurde jedoch ein anderer Weg gewählt.

Hier wird mit dem gleichen Ansatz die Position von jedem Vertex auf der Kegeloberfläche berechnet. Dazu wird zu jedem 2D Vertex eine 3D Gerade errechnet. Diese Gerade ist die Weltlinie des Vertex und hat mit dem Kegel genau einen Schnittpunkt. Dieser kann wiederum als 2D Vertex auf der Kegeloberfläche interpretiert werden, wenn die Z-Koordinate des Schnittpunktes vernachlässigt wird.

Diese Überlegung lässt sich nun auf den dreidimensionalen Raum erweitern: Durch Hinzufügen der Zeitachse T entsteht ein vierdimensionaler Raum, in dem für jeden Vertex ein Schnitttest mit dem 4D-Äquivalent eines Kegels gemacht wird. Wenn nun von diesem Schnittpunkt die T-Koordinate verworfen wird, so erhält man einen Vertex im 3D an der Stelle, wo der Punkt gesehen wird. Somit kann man für jeden Vertex der Szene diese Schnittpunktberechnung durchführen und das resultierende 3D Modell z.B. mit OpenGL darstellen.

Der Kegel lässt sich durch Geraden beschreiben, die durch die Position der Kamera gehen und einen bestimmten Winkel α zur T-Achse haben. Dieser Winkel gibt die Geschwindigkeit des Lichtes wieder und wurde auf $45°$ festgelegt. Durch die Modellierung des Kegels als Menge von Geraden, erhält man nun zwei Schnittpunkte für jeden Vertex. Einen in der "Vergangenheit" und einen in der "Zukunft". Der gesuchte Schnittpunkt befindet sich in der Vergangenheit und hat daher die kleinere Koordinate auf der T-Achse.

PSfrag replacements



t_5

Abbildung 9: Der Schnitt mit dem Kegel kann als Schnitt mit einer Menge von Geraden angesehen werden.

Die Weltlinie eines Vertex wird beschrieben als Gerade

$$v = o + m \cdot r \tag{6}$$

Die Position der Kamera zum Zeitpunkt t_0 sei $k = (k_x, k_y, k_z, t_0)^T$. Die den Kegel beschreibenden Lichtstrahlen seien die Geraden $k + n \cdot l_i$, wobei $|l_i| := 1$ gelten soll. Den gesuchten Punkt erhält man durch gleichsetzen der Geraden mit der Gleichung (6):

$$k + n \cdot l_i = o + m \cdot r \tag{7}$$

Der Winkel zwischen den Lichtstrahlen und der T-Achse $t = (0, 0, 0, 1)^T$ wurde als $45°$ definiert, somit gilt weiterhin:

$$\begin{aligned} \frac{|t \cdot l_i|}{|t| \cdot |l_i|} &= \cos 45° \\ \frac{|l_{it}|}{1 \cdot 1} &= \sqrt{0,5} \\ |l_{it}| &= \sqrt{0,5} \end{aligned} \tag{8}$$

Zusammen mit der Bedingung $|l_i| := 1$ ergibt sich:

$$\begin{aligned} \sqrt{l_{ix}^2 + l_{iy}^2 + l_{iz}^2 + l_{it}^2} &= 1 \\ l_{ix}^2 + l_{iy}^2 + l_{iz}^2 &= 0,5 \end{aligned} \tag{9}$$

Setzt man dies in die Gleichung (7) ein, so erhält man ein lineares Gleichungssystem:

$$\begin{aligned} k_x + n\sqrt{0,5 - l_{iy}^2 - l_{iz}^2} - o_x - m \cdot r_x &= 0 \\ k_y + n \cdot l_{iy} - o_y - m \cdot r_y &= 0 \\ k_z + n \cdot l_{iz} - o_z - m \cdot r_z &= 0 \\ t_0 + n\sqrt{0,5} - o_t - m \cdot r_t &= 0 \end{aligned} \tag{10}$$

Dieses Gleichungssystem hat vier Unbekannte l_{iy}, l_{iz}, n und m. Interessant ist lediglich der Parameter m. Wird dieser in die Gleichung (6) eingesetzt, erhält man den gesuchten Punkt. Löst man nun das lineare Gleichungssystem (10) nach m auf, so entsteht folgende quadratische Gleichung:

$$\begin{aligned} & m^2 \cdot (r_t^2 - r_x^2 - r_y^2 - r_z^2) \\ + \quad & m \cdot 2(r_t(o_t - t_0) - r_x(o_x - k_x) - r_y(o_y - k_y) - r_z(o_z - k_z)) \\ + \quad & (o_t - t_0)^2 - (o_x - k_x)^2 - (o_y - k_y)^2 - (o_z - k_z)^2 = 0 \end{aligned} \tag{11}$$

Somit kann man die zwei Lösungen für m errechnen:

$$
\begin{aligned}
m_{\frac{1}{2}} &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\
a &= r_t^2 - r_x^2 - r_y^2 - r_z^2 \\
b &= 2(r_t(o_t - t_0) - r_x(o_x - k_x) - r_y(o_y - k_y) - r_z(o_z - k_z)) \\
c &= (o_t - t_0)^2 - (o_x - k_x)^2 - (o_y - k_y)^2 - (o_z - k_z)^2
\end{aligned} \tag{12}
$$

Gesucht ist nun das m, das beim Einsetzen in (6) den Punkt zurück liefert, der den kleineren Wert v_t hat. Der Schnittpunkt kann nun mit Hilfe der Funktion `calculateIntersection` errechnet werden. `k[]` gibt dabei die Kameraposition an, `o[]` den Stützvektor der Weltlinie und `r[]` den Richtungsvektor der Weltlinie. Der gefundene Schnittpunkt wird über die Parameter `x`, `y` und `z` zurückgegeben.

Listing 1: Schnittpunkttest für beliebige Weltlinien

```
1  void calculateIntersection(float k[4],
2                             float o[4],
3                             float r[4],
4                             float &x, float &y, float &z)
5  {
6      float a,b,c;
7      float m1,m2;
8
9      a = r[3]*r[3] - r[0]*r[0] - r[1]*r[1] - r[2]*r[2];
10
11     b = 2*(  r[3]*(o[3]-k[3]) - r[0]*(o[0]-k[0])
12            - r[1]*(o[1]-k[1]) - r[2]*(o[2]-k[2]) );
13
14     c = (o[3]-k[3])*(o[3]-k[3]) - (o[0]-k[0])*(o[0]-k[0])
15       - (o[1]-k[1])*(o[1]-k[1]) - (o[2]-k[2])*(o[2]-k[2]);
16
17     float d = (-1.0)*b / (2.0*a);
18     float e = sqrt(b*b - 4.0*a*c) / (2.0*a);
19
20     m1 = d-e;
21     m2 = d+e;
22
23     float t1 = o[3] + m1*r[3];
24     float t2 = o[3] + m2*r[3];
25
26     float m = (t1<t2)?m1:m2;
27
28     x = o[0] + m*r[0];
29     y = o[1] + m*r[1];
30     z = o[2] + m*r[2];
31 }
```

Die hier verwendete Implementierung geht allerdings von einem vereinfachten Modell aus. So bewegen sich alle Objekte mit der gleichen Geschwindigkeit entlang der Z-Achse auf den Betrachter zu. Die Kamera blickt dabei entlang der positiven Z-Achse. Um dennoch in beliebige Richtungen fliegen zu können wird zunächst die Welt in das Kamerakoordinatensystem transformiert. Durch diese Einschränkung vereinfacht sich der Algorithmus wie folgt: Als Eingabe werden nur noch der Vertex in 3D Koordinaten $(v_x, v_y, v_z)^T$ und die Geschwindigkeit v benötigt. Die Weltlinie (und somit die 4D Gerade des Vertex) muss nicht mehr vorweg berechnet werden.

Übergibt man also der Funktion `calculateIntersection` aus Listing 1 nur noch den 3D Vertex und die Geschwindigkeit v unter den genannten Annahmen, so kann man `o[]` und `r[]` innerhalb der Funktion errechnen. Listing 2 zeigt den Anfang dieser neuen Funktion. Der 3D Vertex wird als `vertex[]` übergeben, die Geschwindigkeit als `float v`. Die Kameraposition muss nicht mehr explizit übergeben werden, da die Kamera im Koordinatenursprung ruht.

Listing 2: Schnittpunkttest für Objekte die sich auf die Kamera zu bewegen

```
1  void calculateIntersection2(float vertex[3],
2                             float v,
3                             float &x, float &y, float &z)
4  {
5      const float T = 100.0f;
6      float o[4] = {vertex[0], vertex[1], vertex[2], 0.0f};
7      float r[4] = {0.0f, 0.0f, v*T, -T};
8      float k[4] = {0.0f, 0.0f, 0.0f, 0.0f};
9
10     float a,b,c;
11
12     (...)
13 }
```

Die T-Koordinate der Kamera, also die aktuelle Zeit wird auf 0 gesetzt. Der Stützvektor der Weltlinie des übergebenen Vertex soll seine aktuelle Position sein, somit wird auch hier die T-Komponente auf 0 gesetzt. Der Richtungsvektor zeigt quasi in die Vergangenheit zu der Position, wo sich der Punkt zum Zeitpunkt $t = -T$ befand. Daher ergibt sich als T-Koordinate $-T$.

Das Verhältnis der Z- zur T-Koordinate, also die Steigung des Vektors, gibt die Geschwindigkeit v des Objektes wieder.

Dadurch, dass ein Teil der früheren Parameter nun bekannt ist, lässt sich die Funktion stark vereinfachen.

Listing 3: Vereinfachter Schnittpunkttest

```
1  void calculateIntersection3(float vertex[3],
2                             float v,
3                             float &x, float &y, float &z)
4  {
```

```
5       const float T = 100.0f;
6       float o[4] = {vertex[0], vertex[1], vertex[2], 0};
7       float r[4] = {0.0f, 0.0f, v*T, -T};
8       float k[4] = {0.0f, 0.0f, 0.0f, 0.0f};
9
10      float a,b,c;
11      float m1,m2;
12
13      a = T*T - 0 - 0 - r[2]*r[2];
14
15      b = 2*(  0 - 0
16              - 0 - r[2]*(vertex[2]) );
17
18      c = 0 - (o[0])*(o[0])
19        - (o[1])*(o[1]) - (o[2])*(o[2]);
20
21      (...)
22  }
```

Nun setzt man die Eingabeparameter direkt in die entsprechenden Stellen ein. Die Variable `a` wird immer mit `2.0` multipliziert, somit kann man diesen Faktor vorziehen.

Listing 4: Vereinfachter Schnittpunkttest

```
1   void calculateIntersection3(float vertex[3],
2                               float v,
3                               float &x, float &y, float &z)
4   {
5       const float T = 100.0f;
6
7       float a,b,c;
8       float m1,m2;
9
10      a = 2.0*T*T - 2.0*(v*T)*(v*T);
11
12      b = 2*( v*T )*vertex[2];
13
14      c = -(vertex[0]*vertex[0])
15          -(vertex[1]*vertex[1])
16          -(vertex[2]*vertex[2]);
17
18      float d = (-1.0)*b / a;
19      float e = sqrt(b*b - 2.0*a*c) / a;
20
21      m1 = d-e;
22      m2 = d+e;
23
24      float t1 = m1*(-T);
```

```
25      float t2 = m2*(-T);
26
27      float m = (t1<t2)?m1:m2;
28
29      x = vertex[0];
30      y = vertex[1];
31      z = vertex[2] + m*(-v*T);
32  }
```

Stellt man `a` zu `(1-v*v)*2.0*T*T` um, so wird ersichtlich, dass `a` immer positiv ist. Dies liegt daran, dass `v` kleiner `1` sein muss, da `v` die Geschwindigkeit in $\frac{v}{c}$ angibt. Somit ist auch klar, dass `e` immer positiv sein wird.

Der Punkt bewegt sich entlang der Z-Achse vor der Kamera zu dieser hin und hinter der Kamera von dieser weg. Da die Kamera entlang der positiven Z-Achse schaut, muss der gesuchte Wert größer als `vertex[2]` sein. Daher muss auch `m` positiv sein.

Somit ist `m` der größere der Werte `m1` und `m2` und zudem positiv. Daher muss `m = m2 = d+e` sein, da `e` positiv ist.

Nach diesen Überlegungen hat sich der Code wesentlich vereinfacht:

Listing 5: Die verwendete Schnittpunktberechnung

```
1   void calculateIntersection3(float vertex[3],
2                               float v,
3                               float &x, float &y, float &z)
4   {
5       const float T = 100.0f;
6
7       float a,b,c;
8       a = (1-v*v)*2.0*T*T;
9       b = 2.0*( v*T )*vertex[2];
10      c = -(vertex[0]*vertex[0])
11          -(vertex[1]*vertex[1])
12          -(vertex[2]*vertex[2]);
13
14      x = vertex[0];
15      y = vertex[1];
16      z = vertex[2] - (v*T)*
17                  ((-1.0*b + sqrt(b*b - 2.0*a*c))/a);
18  }
```

Da `a` nur einmal pro Frame für alle Vertizes berechnet werden muss, und die Zuweisungen `x = vertex[0]` bzw. `y = vertex[1]` nicht mehr nötig sind, lässt sich dieser Ansatz leicht in einer Schleife über alle Vertizes verwenden.

3.3.3 Die Lichtlaufzeit bei variabler Geschwindigkeit

Der hier vorgestellte Algorithmus geht davon aus, dass sich Objekte mit einer konstanten Geschwindigkeit fortbewegen. Er lässt sich aber auch leicht für Objekte anpassen, die im Laufe der Simulation beschleunigen oder abbremsen. Hierfür wird ein Vertex des Objektes nicht mehr als eine Gerade dargestellt, sondern als Linienzug, d.h. als Menge von Geraden $g_1, ..., g_n$, die jeweils nur auf einem bestimmten Intervall $[g_{i1}, g_{i2}]$ definiert sind.

Der vorgestellte Schnittpunkttest muss dann für alle Geraden $g_1, ..., g_n$ durchgeführt werden, bis ein Schnittpunkt gefunden wurde, der innerhalb des jeweiligen Intervalls liegt. Auf diese Weise kann man allerdings nur abrupte Geschwindigkeitsänderungen simulieren, da Beschleunigungen keine geraden Weltlinien darstellen, sondern Kurven.

3.3.4 Der Suchscheinwerfereffekt

In dieser Arbeit wird der Suchscheinwerfereffekt nach [WKR99] implementiert, allerdings ohne die Berücksichtigung der Farbe. Da der Effekt die Helligkeit erhöht, muss diese zunächst verringert werden, da sonst nicht viel zu erkennen ist.

Ein Problem bei diesem Ansatz sind Farben, bei denen der Rot-, Grün- oder Blauwert 0 ist: Die anderen Kanäle werden schnell verstärkt und erreichen ihren maximal darstellbaren Wert von 255 bzw. 1.0, der Kanal mit 0 bleibt 0. Dies ist zwar korrekt, aber in der Realität gibt es selten reine Farbtöne. Daher wird zu jedem Farbkanal ein kleiner Wert hinzu addiert (bei `float` Werten 0.01), wodurch jede Farbe bei hinreichender Geschwindigkeit zu Weiß wird.

3.3.5 Der Software-basierte Renderer

Bei der Berechnung der Lichtlaufzeit und der Längenkontraktion wird in einer Schleife über alle Vertizes gegangen und zunächst der Vertex in das Kamerakoordinatensystem transformiert. Dann wird die Längenkontraktion angewendet und wie in Listing 5 auf Seite 19 die Lichtlaufzeit berechnet.

Um diese Berechnung zu beschleunigen wird ein Array angelegt, in dem nur die Geometrie des Modells abgelegt ist. In jedem Frame wird nun ein zweites Array mit den transformierten Vertizes gefüllt. Diese Arrays werden von einem Objekt der Klasse *Mesh* bereitgestellt. Das heißt, es werden zuerst alle Vertizes verzerrt, bevor diese an OpenGL weitergereicht werden.

Erst in diesem zweiten Schritt werden die Farbwerte benötigt.

3.3.6 Der Vertexshader-basierte Renderer

Das Rendering auf der Graphikhardware verläuft ähnlich. Hier erfolgt das Caching mit Hilfe von Displaylisten. Die Transformationen werden komplett auf der Hardware durchgeführt. Im Gegensatz zum Softwarerenderer wird hier zunächst eine simple Beleuchtung angewendet.

Listing 6: Der Vertexshader

```
void main(float4 inPosition   : POSITION,
          float4 inNormal     : NORMAL,
          float4 inColor      : COLOR,
      out float4 outPosition  : POSITION,
      out float4 outColor     : COLOR,
  uniform float  v,
  uniform float  con,
  uniform float  lightRun) {
    // Beleuchtung
    float3 normal = normalize(
                    mul(glstate.matrix.invtrans.modelview[0],
                        inNormal).xyz );
    float3 light  = {0.0, 1.0, 0.0};
    float3 look   = {0.0,0.0,-1.0};
    float  cos_gamma = abs( dot( light, inNormal.xyz ) );
    float  cos_delta = abs( dot( look, normal.xyz ) );
    outColor = ( inColor*cos_gamma*2.0f
                +inColor*cos_delta
                +inColor) /4.0f;

    // Vertex ins Kamerakoordinatensystem
    float4 vertex = mul(glstate.matrix.modelview[0],
                            inPosition);

    vertex.z = vertex.z * con;

    if (lightRun >= 1.0f) {
        float a2 = 200.0f *(1.0f-v*v);
        float b = -20.0f*v*vertex.z;
        float c = -(vertex.x*vertex.x)
                  -(vertex.y*vertex.y)
                  -(vertex.z*vertex.z);
        float d = (b + sqrt(b*b - 2.0f*a2*c))/a2;
        vertex.z = (vertex.z - d*10.0f*v);
    }

    float4 white = {1.0,1.0,1.0,1.0};
    outColor = outColor/3.0f + 0.01*white;
    float cos_alpha = -vertex.z / length(vertex.xyz);
    float sla = (1-v*v);
    float slb = (1-v*cos_alpha);
    outColor *= ((sla*sla) / (slb*slb*slb*slb));

    // Projektionsmatrix
    outPosition = mul(glstate.matrix.projection, vertex);
}
```

In Listing 6 ist der komplette Shader abgedruckt. Übergeben werden ihm die üblichen Parameter Vertexposition, Normale und Farbe (Zeilen 1 - 3). Mit Ausnahme der Normalen werden diese Werte nach der Transformation auch wieder ausgegeben (Zeilen 4 und 5). Die Parameter `v`, `con` und `lightRun` werden vom Programm jeden Frame neu gesetzt. `v` gibt hierbei die Geschwindigkeit an. `con` den Wert $1 - v^2$, falls die Längenkontraktion simuliert werden soll, ansonsten 1. Wenn die Lichtlaufzeit simuliert werden soll, wird `lightRun` auf 1 gesetzt, sonst auf 0. Der Suchscheinwerfereffekt wird deaktiviert, indem in einem zweiten Shaderprogramm die Zeilen 37 bis 42 fehlen. Er ließe sich auch durch eine weitere Bedingung abschalten, dies erwies sich allerdings als weniger performant.

Die Zeilen 11 bis 20 beleuchten den Vertex. Nachdem das Vertex in den Zeilen 22 und 23 ins Kamerakoordinatensystem transformiert wurde, wird in der Zeile 25 die Längenkontraktion angewendet. Wenn die Lichtlaufzeit aktiviert ist, wird diese in den Zeilen 27 bis 35 berechnet. Zu beachten ist, dass die Kamera hier entlang der negativen Z-Achse blickt und daher ein paar Vorzeichen von der Software Version in Listing 5 abweichen.

Wie im Kapitel 3.3.4 beschrieben, soll zum Farbwert ein kleiner Anteil Weiß hinzu addiert werden. Dies geschieht in den Zeilen 37 und 38. 1% reicht aus, der Unterschied zur Originalfarbe ist nicht erkennbar. Die folgenden Zeilen berechnen den Suchscheinwerfereffekt nach [WKR99].

3.4 Die Benutzeroberfläche

Abbildung 10 zeigt die Oberfläche des Programms. Die linke Seite zeigt die Szene, rechts können Einstellungen vorgenommen werden. Diese Einstellungen sind in zwei Gruppen eingeteilt, oben die allgemeinen Renderingoptionen und Anzeigen, unten die für die Simulation interessanten.

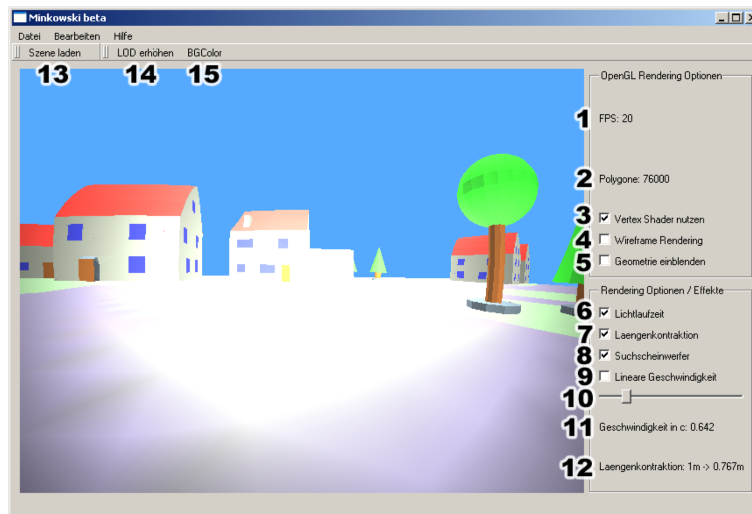


Abbildung 10: Die in QT erstellte GUI des Programmes.

Im oberen Block werden die Frames pro Sekunde (1) und die Anzahl der Polygone der Szene angezeigt (2). Mit der Checkbox *Vertex Shader nutzen* kann man zwischen der Software Version und der GPU beschleunigten Version umschalten (3). Zudem lässt sich die Szene als Wireframe rendern (4). Die Option *Geometrie einblenden* bewirkt, dass die Szene als Wireframe nicht-relativistisch über das Bild gelegt wird. Somit kann man in einem Bild sehen, wo der Betrachter die Objekte sehen würde und wo sie tatsächlich sind (5).

Die Effekte der Lichtlaufzeit, der Längenkontraktion und des Suchscheinwerfereffektes lassen sich einzeln de-/aktivieren (6 – 8).

Der Slider 10 steuert die Geschwindigkeit des Betrachters. Da die geometrischen Effekte erst bei Geschwindigkeiten größer als $0,8c$ deutlich werden, ist der Slider in diesen Bereichen empfindlicher. Das heißt, der Bereich von 0 bis 0.85 nimmt die erste Hälfte des Sliders in Anspruch, die zweite Hälfte steuert den Bereich von 0.85 bis $1c$. Da beim Suchscheinwerfereffekt die geringeren Geschwindigkeiten interessanter ist, lässt sich der Slider auf eine lineare Abbildung der Stellung zur Geschwindigkeit einstellen. Dies geschiegt mit der Checkbox *Lineare Geschwindigkeit* (9).

Die aktuelle Geschwindigkeit, sowie die Auswirkung der Längenkontraktion werden als Text eingeblendet (11 und 12).

Eine neue Szene laden kann man sowohl über das Datei Menü, als auch über die Toolbar (13). Will man die Polygone der Szene zur Laufzeit weiter unterteilen, kann man dies mit dem Button 14 erreichen. Hintergrundfarbe und Wireframefarbe können im Bearbeiten Menü eingestellt werden, erstere auch über die Toolbar (15).

3.5 Anforderungen an das Modell

Nicht jedes Modell ist für das relativistische Rendering geeignet, da diese Art der Visualisierung besondere Anforderungen an das Mesh stellt. Die Verzerrung durch die Lichtlaufzeit führt zu Krümmungen auf bisher geraden Flächen. Da aber nicht die ganze Fläche, sondern nur die Vertizes transformiert werden, können diese typischen Krümmungen bei hohen Geschwindigkeiten nur dann dargestellt werden, wenn die Fläche in genügend viele Polygone unterteilt wurde. Abbildung 11 zeigt die Auswirkungen unterschiedlicher Modellauflösungen.

Die Unterteilung des Modells muss zudem gleichmäßig sein, da es sonst nicht nur zu einem falschen Eindruck der speziellen Relativitätstheorie kommen kann, sondern auch falsche Geometrien entstehen können. Abbildung 12 zeigt ein Modell bestehend aus zwei parallelen Flächen. Die hintere Fläche ist aus 128 Polygonen zusammengesetzt und rot eingefärbt. Die vordere besteht lediglich aus zwei Polygone und ist grün gefärbt. Im linken Bild bewegt sich der Betrachter nicht, im rechten fliegt er mit $0,8$ facher Lichtgeschwindigkeit auf das Objekt zu. Aufgrund der Lichtlaufzeit müssten sich nun die Flächen zum Betrachter hin biegen, aufgrund der zu geringen Auflösung der vorderen Fläche, geschieht dies nur bei der hinteren Seite. Diese durchstößt nun die Vorderseite. In beiden Bildern wurde das Wireframe zusätzlich eingeblendet.

Das Modell darf nur aus Dreiecken aufgebaut sein, da die Transformation auf die einzelnen Vertizes eines Polygons abhängig von Geschwindigkeit und Position unterschiedlich wirken. Liegen zum Beispiel die Eckpunkte eines Vierecks vor der relativistischen Verzerrung durch die Lichtlaufzeit auf einer Ebene, so muss dies

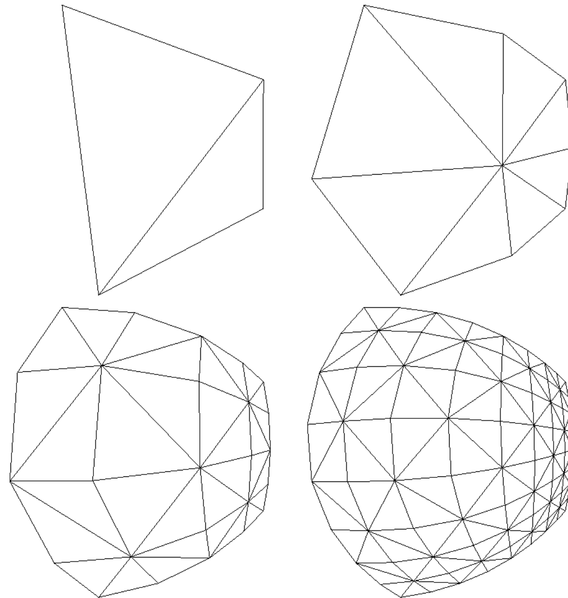


Abbildung 11: Viermal das gleiche Modell bei $0,86c$. Von oben links nach unten rechts: 2 Polygone, 8 Polygone, 32 Polygone und 128 Polygone.

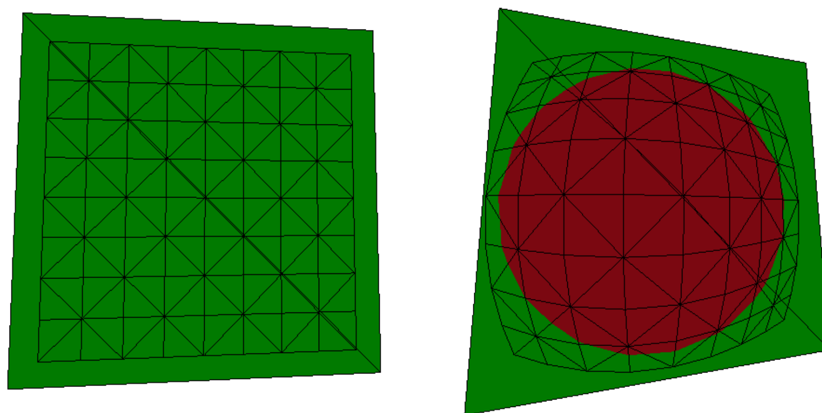


Abbildung 12: Ist die Unterteilung der Flächen ungleichmäßig, so kann die Rückseite eines Modells (rot) durch die Vorderseite stoßen (grün). Links: ruhend, Rechts: $0,8c$.

nach der Verzerrung nicht mehr der Fall sein. Die Darstellung dieses Viereckes ist somit nicht mehr eindeutig.

Ein weiteres Problem kann auftreten, wenn in der Mitte einer Kante eines Polygons ein Eckpunkt eines anderen Polygons liegt. Abbildung 13 zeigt ein Beispiel: Hier teilen sich die Polygone A und B eine Ecke e, durch welche eine Kante des Polygons C verläuft. Aufgrund der relativistischen Verzerrung kann nun folgendes passieren: Der Bereich um e wird nach unten gezogen, auf die Polygone A und B wirkt sich dies durch eine Änderung der Vertexposition e aus, Polygon C hingegen wird nicht verzerrt. Hierdurch entsteht ein kleiner Spalt im transformierten Modell, der besonders bei hohen Geschwindigkeiten auffällt (vgl. Abbildung 14).

PSfrag replacements

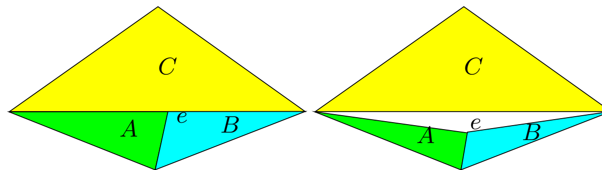


Abbildung 13: Ist die Position der Vertizes so ungünstig wie im linken Bild, können beim relativistischen Rendering Löcher im Modell entstehen (rechts).

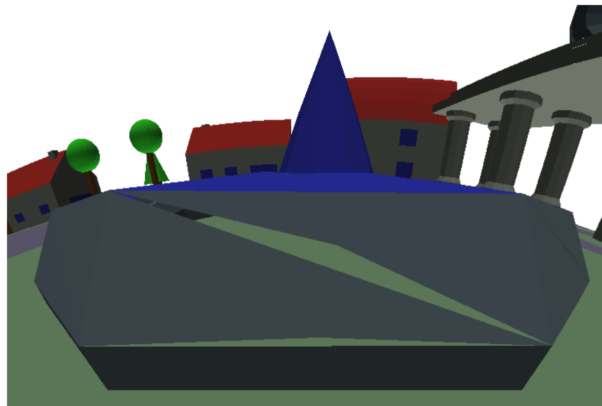


Abbildung 14: Durch ungünstige Positionen der Vertizes entstehen Löcher im Modell nach der Verzerrung.

4 Ergebnisse

Die Ziele dieser Studienarbeit wurden im Wesentlichen erreicht. So visualisiert die Anwendung nicht nur die geometrischen Verzerrungen, sondern zudem auch den Suchscheinwerfereffekt. Die Implementierung der Effekte als Vertexshader war zunächst nicht vorgesehen, erwies sich aber als unproblematisch und verbesserte die Performance wesentlich. Hierdurch wurde auch ein zusätzlicher Lerneffekt erzielt, da dies mein erstes Programm mit Shadern ist.

Das Laden der Modelle als Obj-Dateien erweist sich als bequeme Lösung, da diese direkt in einem Modelling Programm erstellt werden können. Dieses Feature ist erst in einer späten Entwicklungsphase hinzugekommen, vorher musste das Modell durch einen Freewarekonverter und zusätzlich durch ein selbst geschriebenes Tool in das simple `.mesh` Format konvertiert werden. Daher unterstützt das Programm bisher noch nicht alle Möglichkeiten des Obj-Formates, so können leider noch keine Texturen geladen werden, obwohl die relativistischen Effekte auch mit texturierten Modellen funktionieren würde.

Die Einarbeitung in die GUI Bibliothek QT erwies sich als zeitaufwändiger als zunächst angenommen, dies lag zum einen an unzureichender Literatur zur Version 4 von QT (die mitgelieferte Dokumentation war die Hauptinformationsquelle), zum anderen an der Tatsache, das dies meine ersten Versuche mit GUI Programmierung waren. Zwischendurch gab es auch immer wieder größere und kleinere Programmierprobleme, wodurch die Implementierung mehr Zeit in Anspruch genommen hatte, als erwartet.

Auch die Platformunabhängigkeit konnte erreicht werden, so ließ sich das Programm problemlos auf Windows XP portieren.

4.1 Bilder

Neben den technischen Aspekten war das Hauptziel dieser Arbeit die Erstellung einer Anwendung, mit der man sich ein Bild der Effekte bei hohen Geschwindigkeiten machen kann. Daher sollen hier ein paar Bilder folgen, die einen Einblick in Einsteins Relativitätstheorie bieten.

Die meisten Bilder entstanden bei deaktiviertem Suchscheinwerfereffekt, da dieser sonst das Bild zu stark dominiert hätte.

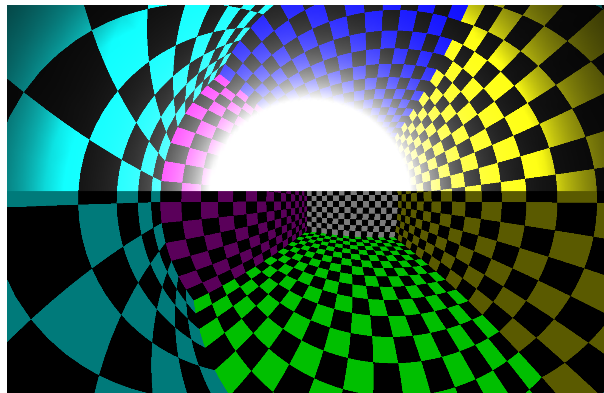


Abbildung 15: Das Innere eines Würfels bei $0,9c$, oben mit, unten ohne Suchscheinwerfereffekt.

Abbildung 15 zeigt das Innere eines Würfels, in dem sich der Betrachter mit 90% der Lichtgeschwindigkeit bewegt. Die geometrischen Verzerrungen sind so stark, dass alle Wände sichtbar werden. Der Suchscheinwerfereffekt dominiert das Bild (obere Hälfte des Bildes), obwohl die Helligkeit des Würfels bereits auf ein Drittel reduziert wurde.

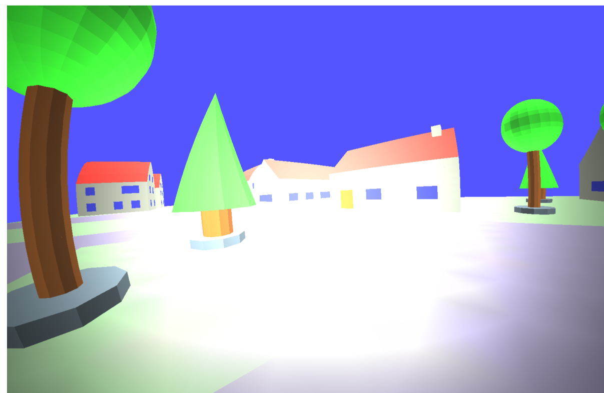


Abbildung 16: Der Suchscheinwerfereffekt bei $0,65c$

Der Suchscheinwerfereffekt ist allerdings schon bei geringeren Geschwindigkeiten deutlich zu erkennen. Abbildung 16 zeigt die "Markt" Szene bei $0,65c$. Die Verzerrungen hingegen sind noch gering.

Einen Flug vorbei am Deutschen Eck mit 95% der Lichtgeschwindigkeit zeigen die Bilder in Abbildung 17. Zunächst wirkt das Denkmal lang gezogen, dann erkennt man deutlich die Wölbung zum Betrachter hin. Im dritten Bild sieht man, wie die Rückwand des rechteckigen Sockels verbogen wird. Die halbkreisförmige Rückwand des Denkmals wirkt hier noch rund, im vierten Bild erkennt man sie schon nicht mehr wieder.

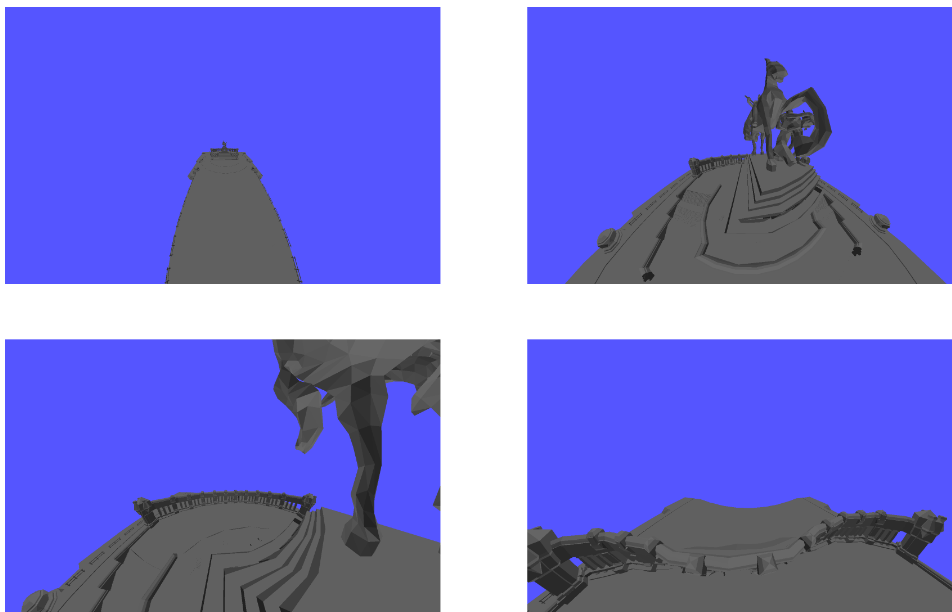


Abbildung 17: Vorbeiflug am Deutschen Eck mit $0,95c$

In Abbildung 18 sieht man, wie einem Beobachter eine Innenstadt erscheint, wenn dieser sich mit $0,95c$ senkrecht nach oben bewegt (links), bzw. mit der gleichen Geschwindigkeit auf den Boden stürzt. Beide Bilder wurden von der gleichen Position aus aufgenommen, lediglich die Flugrichtung wurde geändert.

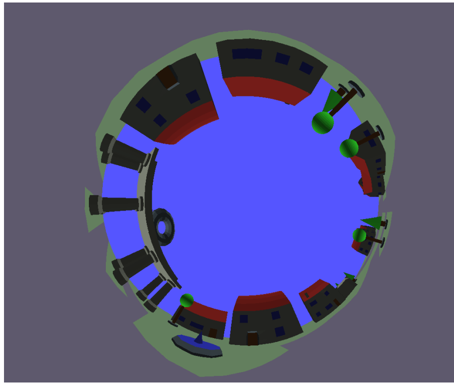
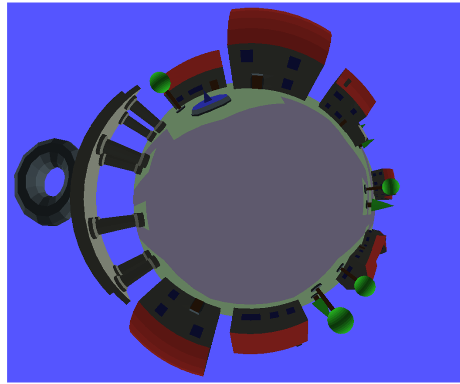



Abbildung 18: Links: Start senkrecht nach oben. Rechts: Sturz nach unten

4.2 Performance

Zur Beurteilung der Performance dieser Anwendung sollen hier ein paar Messwerte präsentiert werden. Da in jedem Frame immer alle Vertizes transformiert werden, sind die Messwerte unabhängig von der Kameraposition und Blickrichtung.

Gemessen wurde an zwei Rechnern, einem Pentium 4 2,4Ghz, 1GB RAM, GeForce4 TI 4200 (Sulley) und einem sehr schnellen Desktop-Rechner mit AMD Athlon 64 X2 4400+, 2 GB RAM, 2 GeForce 7800 SLI (Bob).

Getestet wurden die Szenen unter Windows XP mit dem Software Renderer und dem Vertexshader Renderer, jeweils einmal mit und einmal ohne aktiviertem Suchscheinwerfereffekt. Somit ergeben sich pro Szene 8 Messungen. Zum Teil wurden die Szenen vor der Messung weiter unterteilt, um eine höhere Polygonanzahl zu erhalten.

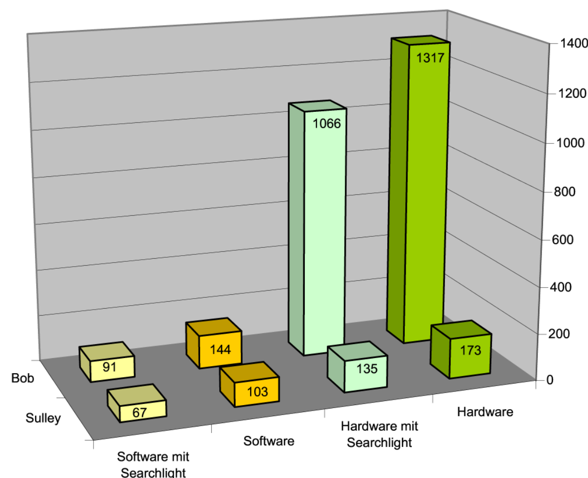


Abbildung 19: Würfel mit 24.576 Polygonen

Der Würfel wurde vor allem zum Debugging verwendet, da hier die Verzerrungen am besten beurteilt werden können. Er ist in Abbildung 15 zu sehen. Die Meshunterteilung wurde von 3072 auf 24.576 Polygone erhöht, Diagramm 19 zeigt die Ergebnisse.

Die Markt Szene wurde bereits in mehreren Bildern gezeigt, so zum Beispiel in Abbildung 18. Die Auflösung wurde auch hier stark erhöht.

Die Szene Spacedock hat mit 316.401 Polygonen die mit Abstand höchste Auflösung. Für den Benchmark in Diagramm 22 wurde die Polygonanzahl auf 632.802 verdoppelt. Abbildung 23 zeigt die Szene.

Mit der Performance der Anwendung bin ich sehr zufrieden, auch große Szenen können in Echtzeit dargestellt werden. Besonders der Einsatz des Vertexshaders hat die Geschwindigkeit der Simulation stark erhöht.

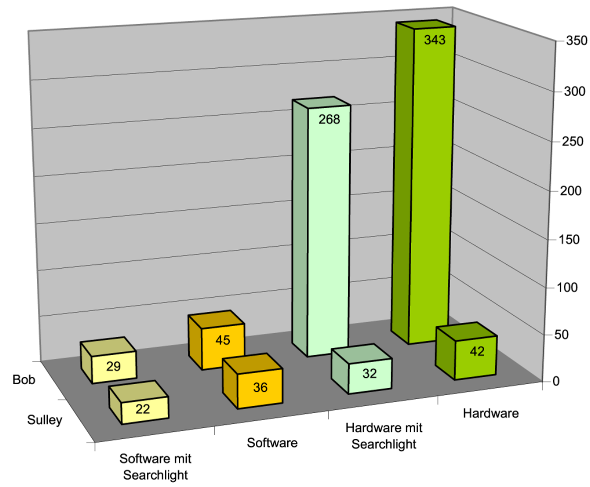


Abbildung 20: Markt mit 76.000 Polygonen

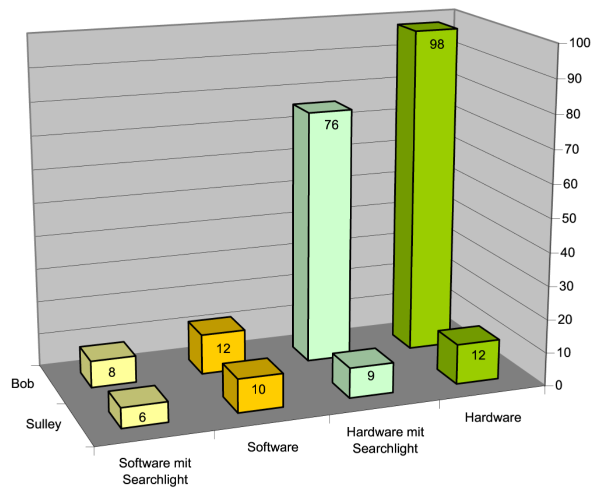


Abbildung 21: Deutsches Eck mit 276.816 Polygonen

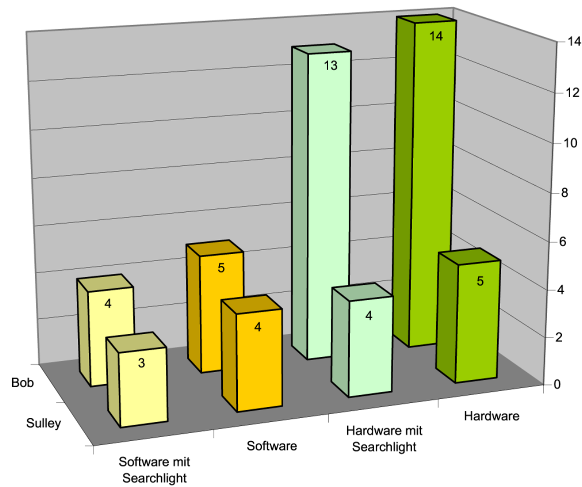


Abbildung 22: Spacedock mit 632.802 Polygonen

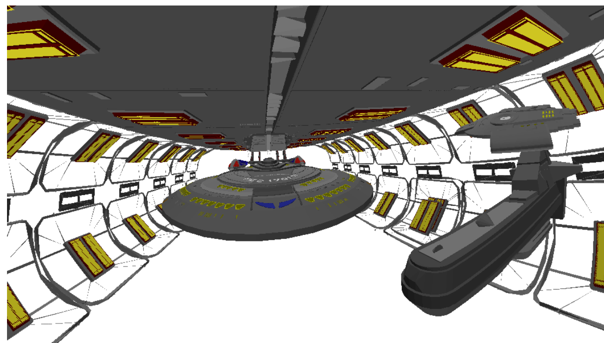


Abbildung 23: Die Spacedock Szene

5 Ausblick

Es gibt noch immer viele Möglichkeiten, das Programm zu erweitern. Zunächst einmal wäre eine bessere Unterstützung bestehender 3D Dateiformate wünschenswert, so dass auch Modelle mit Texturen geladen werden können. Dies würde den Szenen eine wesentlich realistischere Wirkung verleihen. Zudem sollte der Scheinwerfereffekt als Pixelshader implementiert werden, um eine bessere Bildqualität zu erreichen.

Eine denkbare Erweiterung stellt die Integration des Dopplereffektes dar. Hierbei ergibt sich das Problem, dass die Farbe eines Objektes in der Computergraphik nur als Rot-, Grün- und Blauanteil gespeichert wird, man für eine realistische Umsetzung des Dopplereffektes aber ein komplettes Farbspektrum benötigen würde.

Auch lässt sich bisher nur der Flug eines bewegten Beobachters durch eine statische Welt simulieren, wünschenswert ist aber eine dynamische Welt. Also eine Szene, in der sich Objekte mit unterschiedlichen Geschwindigkeiten bewegen, die ein bewegter oder unbewegter Beobachter betrachten kann. Des weiteren wäre eine Simulation denkbar, in der die bewegten Objekte zudem ihre Geschwindigkeit ändern können. Ein Ansatz hierfür wurde in Kapitel 3.3.3 vorgestellt.

Bei starken Verzerrungen ist es wichtig, dass das Modell hoch genug aufgelöst ist. Die Anwendung bietet bereits eine Funktion, um ein zu gering aufgelöstes Modell zur Laufzeit weiter zu unterteilen. Das Problem hierbei ist, dass nicht nur die Polygone unterteilt werden, die (nach der Verzerrung) nah an der Kamera sind, sondern auch die, die soweit entfernt sind, dass die Unterteilung keinen sichtbaren Effekt mehr hat. Wünschenswert wäre daher ein dynamischer Ansatz, der jeweils nur die Polygone (gegebenenfalls mehrfach) unterteilt, die nah genug an der Kamera sind.

Zu guter Letzt lässt sich die Geschwindigkeit der Simulation verbessern. Zwar wurde die Implementierung im Laufe der Arbeit immer effizienter (nicht zuletzt durch den Einsatz von Vertexshadern), trotzdem dürften hier noch einige Optimierungen möglich sein. Dann ließe sich dieser Ansatz auch mit anderen Implementierungen wie dem "Relativator" aus [Sud05] vergleichen, die einen stärkeren Fokus auf Geschwindigkeit legen.

6 Modellverzeichnis

Deutsches Eck Das Modell Deutsches Eck wurde von Hendrik Wiebel modelliert und freundlicherweise zur Verfügung gestellt. Es musste nachträglich modifiziert werden, um eine regelmäßigere Meshstruktur zu erhalten.

Spacedock Die Raumschiffe in diesem Modell, sowie das Raumdock stammen von der Webseite http://www.trekmeshes.ch. Farben und Meshunterteilung wurden nachträglich angepasst.

Würfel Das Modell Würfel wurde von einem von mir geschriebenen Programm erstellt.

sonstige Die restlichen Modelle wurden von mir in Maya erstellt.

Literatur

[Ein05] Albert Einstein. *Zur Elektrodynamik bewegter Körper*. *Annalen der Physik und Chemie*, 17:891–921, 1905.

[FK03] Randima Fernando und Mark J. Kilgard. *The Cg Tutorial*. Addison-Wesley, 2003.

[Gam80] George Gamov. *Mr. Tompkins' seltsame Reisen durch Kosmos und Mikrokosmos*. Vieweg, Braunschweig/Wiesbaden, 1980.

[GK98] J. Grehn und J. Krause. *Metzler Physik*. Schroedel Verlag GmbH, Hannover, dritte Auflage, 1998.

[NR05] Hans-Peter Nollert und Hanns Ruder. *Die relativistische Welt in Bildern*. *Spektrum der Wissenschaft Spezial*, 3, 2005.

[Sch04] Reto U. Schneider. *Das Buch der verrückten Experimente*. C. Bertelsmann Verlag, München, fünfte Auflage, 2004.

[Sud04] Oliver Sudmann. *Algorithmen zur Visualisierung der Relativitätstheorie*. Studienarbeit, Universität Paderborn, Heinz Nixdorf Institut, Theoretische Informatik, February 2004.

[Sud05] Oliver Sudmann. *Interaktive Visualisierung der Speziellen Relativitätstheorie auf programmierbarer Grafikhardware*. Diplomarbeit, Universität Paderborn, 2005.

[Wei01] Daniel Weiskopf. *Visualization of Four-Dimensional Spacetimes*. Dissertation, Eberhard-Karls-Universität zu Tübingen, 2001.

[WKR99] Daniel Weiskopf, Ute Kraus, und Hanns Ruder. *Searchlight and Doppler Effects in the Visualization of Special Relativity: A Corrected Derivation of the Transformation of Radiance*. *ACM Transactions on Graphics*, 18(3):278–292, 1999.