Gymnasium Dorfen

Abiturjahrgang
2021

S E M I N A R A R B E I T

Rahmenthema des Wissenschaftspropädeutischen Seminars:
Simulation und Visualisierung der Relativitätstheorie

Leitfach: Physik

Titel der Arbeit:

Visualisierung der Relativitätstheorie

Verfasser/in:
Christiansen Lars Pascal

Kursleiter/in:
Christian Hoene

Abgabetermin:
(2. Unterrichtstag im November)

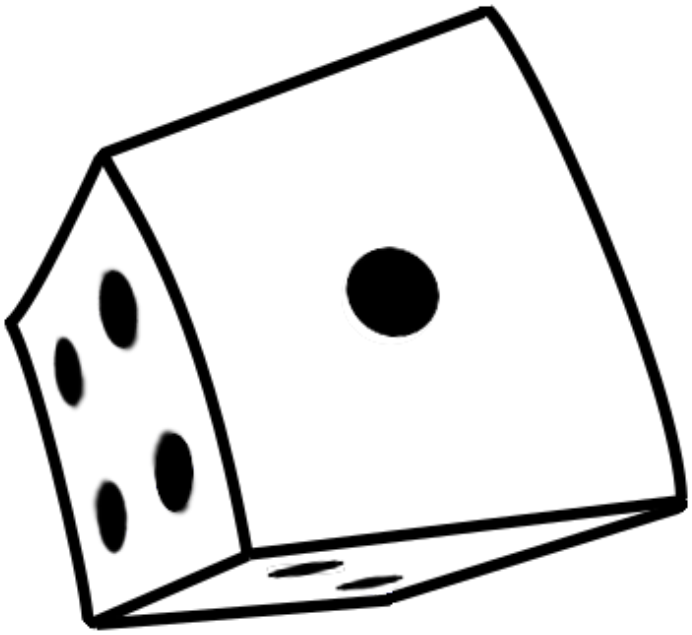
10. November

Bewertung	Note	Notenstufe in Worten	Punkte		Punkte
schriftliche Arbeit				x 3	
Abschlusspräsentation				x 1	
				Summe:	
				Gesamtleistung nach § 61 (7) GSO = Summe:2 (gerundet)	

Datum und Unterschrift der Kursleiterin bzw. des Kursleiters

Visualisierung der Relativitätstheorie

Teil 1: Ausarbeitung



Christiansen Lars

Lengdorf, November 2020

Inhaltsverzeichnis

1 Einleitung

Seitdem der Rechner erfunden wurde, wird er häufig eingesetzt, um physikalische Systeme zu simulieren und so neue Erkenntnisse zu erlangen. So gibt es zum Beispiel in der Mechanik Fragestellungen wie das 3 Körperproblem, welche nur numerisch gelöst werden können. Vor allem in der modernen Physik können Simulationen nicht nur zur Lösung von Problemen beitragen, sondern auch die immer komplexer werdenden Gedankenkonstrukte visualisieren, um uns ein intuitives Verständnis dafür zu ermöglichen. So ist es möglich einer breiten Schicht der Gesellschaft die meist abstrakten Prinzipien der modernen Forschung auf anschauliche Art und Weise zu vermitteln. Diese Arbeit soll sich mit der Visualisierung einer der bekanntesten Theorien der modernen Physik beschäftigen: Mit der Relativitätstheorie von Albert Einstein.

2 Visualisierung der Relativitätstheorie

Das Ziel dieser Arbeit besteht darin, ein Programm vorzustellen, welches einige der relativistischen Effekte der speziellen Relativitätstheorie darstellen kann. Bevor deshalb auf die verwendeten Algorithmen eingegangen wird, sollen zuerst die nötigen physikalischen Formeln und Grundlagen besprochen werden, damit anschließend die Funktionsweise des Programms besser verdeutlicht werden kann.

2.1 Physikalische Grundlagen der Relativitätstheorie

Die Relativitätstheorie ist ein gutes Beispiel, um zu zeigen, auf welch einfachen Annahmen komplexe Theorien beruhen können. So müssen für diese Theorie nur 2 Axiome gelten:

- In jedem Inertialsystem gelten die gleichen physikalischen Gesetze, wenn diese sich mit konstanter Geschwindigkeit zueinander bewegen.
- Die Lichtgeschwindigkeit c ist unabhängig von einem Bewegungszustand oder einem Relativsystem eines Beobachters immer konstant.

Allein durch diese Grundsätze war Einstein in der Lage eine für die moderne Wissenschaft so bedeutende Theorie zu folgern.

2.1.1 Das Gedankenexperiment

Um die nachfolgenden relativistischen Effekte besser verständlich zu machen soll, folgendes gängiges Gedankenexperiment dienen. Man nehme an, dass sich ein Zug durch einen Bahnhof mit der Geschwindigkeit v bewegt. Auf dem Bahnsteig sowie in dem Zug sollen sich nun verschiedene Beobachter befinden, deren Beobachtungen und Messergebnisse verglichen werden sollen. Ihre Messgeräte beschränken sich auf Maßstäbe und Uhren sowie Blitzlichter, welche zum Austausch von Lichtsignalen dienen sollen. Beide Beobachter können mit ihren Messungen nur Aussagen über ihr eigenes Bezugssystem treffen, einzig durch Lichtsignale können Rückschlüsse auf Ereignisse an anderen Orten gezogen werden (Abb. 1).

Abbildung 1: Gedankenexperiment

2.1.2 Addition der Geschwindigkeiten

Ein klassisches Beispiel für relativistisches Denken ist die Addition von Geschwindigkeiten. Nehmen wir an, dass ein Beobachter in einem bewegten Zug mit der Geschwindigkeit v eine Taschenlampe in Fahrtrichtung hält. Nach der klassischen Physik müssten sich die Geschwindigkeit des Zuges v_{zug} und die des Lichts c addieren, um auf die Gesamtgeschwindigkeit zu kommen. Somit käme man auf die Formel:

$$v_{ges} = v_{zug} + c \tag{1}$$

Da aber v_{zug} positiv ist, wäre die Gesamtgeschwindigkeit größer als die Lichtgeschwindigkeit, wodurch deren Konstanz verletzt wäre. Daran wird klar, dass hier ein anderes Prinzip gelten muss, und zwar lässt sich die relativistische Geschwindigkeitsaddition

wie folgt berechnen:

$$u_x = \frac{u'_x + v_x}{1 + \frac{u'_x \cdot v_x}{c^2}} \tag{2}$$

Wobei u'_x der Geschwindigkeit des Zuges für den außenstehenden Beobachter entspricht und v_x der Geschwindigkeit des sich im Zug bewegten Objektes aus Sicht des im Zug fahrenden Beobachters entlang der X-Achse. Auffällig ist hier, dass bei geringen Geschwindigkeiten der Bruch im Nenner gegen null geht und somit die Näherung:

$$u_x = u'_x + v_x \tag{3}$$

im Alltag durchaus ihre Relevanz behält.

2.1.3 Zeitdilatation

Auch die Zeit verhält sich bei hohen Geschwindigkeiten anderes als wir es gewohnt sind. Würde man auf dem Zug sowie auf dem Bahnsteig eine Uhr montieren, so würde man nach einer Fahrt des Zuges feststellen, dass die beiden Uhren nicht mehr dieselbe Zeit anzeigen. Dieses Phänomen lässt sich mit der Zeitdilatation erklären, welche im Kern sagt, dass bewegte Uhren aus Sicht eines ruhenden Inertialsystems langsamer gehen. Die genaue Berechnung der Zeitdilatation sieht wie folgt aus:

$$\Delta\tau = \Delta t \cdot \sqrt{1 - \frac{v^2}{c^2}} \tag{4}$$

$\Delta\tau$ = Zeitintervall der auf dem Zug montierten Uhr.
Δt = Zeitintervall der Bahnhofsuhr.

Durch diese Annahme konnte auch erstmals das Paradoxon des Myonenzerfalls erklärt werden. Myonen entstehen in der Atmosphäre und dürften nach der klassischen Physik gar nicht an der Erdoberfläche ankommen, da sie davor schon zerfallen hätten müssen. Da sie sich aber nahe an der Lichtgeschwindigkeit bewegen, vergeht aus Sicht der Myonen viel weniger Zeit bis zur Ankunft auf der Erdoberfläche, weshalb es für den menschlichen Beobachter so aussieht, als ob ihre Zerfallszeit um ein vielfaches länger sein müsste.

2.1.4 Lorentzkontraktion

In Hinsicht auf die Visualisierung ist der Effekt der Längenkontraktion wohl einer der eindrucksvollsten Konsequenzen der speziellen Relativitätstheorie. Um die Länge bewegter Objekte messen zu können, ergibt sich das Problem, dass man nicht zur selben Zeit an unterschiedlichen Orten Messen kann, ohne in unterschiedlichen Bezugssystemen verschiedene Ergebnisse zu erhalten. Deshalb müssen die Auswirkungen der relativen Gleichzeitigkeit mitberücksichtigt werden. Aus diesem Grund ist es einfacher die Längen zu berechnen statt zu messen. Um nun zu verstehen, warum schnell bewegte Objekte kürzer erscheinen, nehmen wir an, dass der Zug mit einer gewissen Geschwindigkeit v in den Bahnhof einfährt und zwei Beobachter mit Hilfe von Uhren versuchen die Länge des Bahnsteiges zu errechnen. Beide Beobachter messen jeweils die Zeit, bei der die Mitte des Zuges und die Enden des Bahnsteiges sich treffen. Der Beobachter im Zug kann nun die Länge berechnen, indem er seine eigene gemessene Zeit mit der Relativgeschwindigkeit multipliziert. Auch der außenstehende Beobachter kann aus seiner gemessenen Zeit nach dieser Formel die Länge bestimmen:

$$L_{bahnsteig} = v \cdot \Delta t_{bahnsteig} \tag{5}$$

$$L_{zug} = v \cdot \Delta t_{zug} \tag{6}$$

Da aber nach der Zeitdilatation beide verschiedene Zeiten bei der gleichen Geschwindigkeit messen, müssen die Strecken folglich unterschiedlich lang sein. Somit lässt sich die Formel für die Längenkontraktion direkt aus der Zeitdilatation herleiten:

$$L' = L \cdot \sqrt{1 - \frac{v^2}{c^2}} \tag{7}$$

So ergibt sich, dass bewegte Objekte verkürzt erscheinen: Angenommen, ein Zug fährt mit einer Geschwindigkeit von $0.5c$ und der Bahnsteig hat eine Länge von 100 Metern, so hätte der Bahnsteig für den Beobachter im Zug nur eine Länge von 87 Metern.

2.1.5 Lorentztransformation

Alle oben beschriebenen Effekte gehen immer aus der Betrachtung von einem Koordinatensystem in das andere hervor. Deshalb lassen sich diese Phänomene alle auf die Lor-

entztransformation zurückführen[1]. Sie beschreibt die Koordinatentransformation von einem Inertialsystem in das andere und erweitert die Galilei-Transformation[2] auf die Relativitätstheorie. Da die Raumzeit in der Relativitätstheorie als Einheit betrachtet werden muss, ist in der Lorentztransformation jeder Koordinatenpunkt X vierdimensional:

$$X = \begin{pmatrix} x \\ y \\ z \\ c \cdot t \end{pmatrix} \tag{8}$$

Wenn sich nun zwei Inertialsysteme in X-Richtung mit der Geschwindigkeit v relativ zueinander bewegen, so berechnet man:

$$t' = \gamma(t - \frac{v_x}{c^2}) \quad \text{und} \quad x' = \gamma(x - v_x t) \tag{9}$$

Die Koordinaten y und z erfahren dabei keine Veränderung. Weiterhin sollen in dieser Arbeit die Abkürzungen gelten:

$$\gamma = \sqrt{1 - \beta^2}^{-1} \quad \text{und} \quad \beta = \frac{v}{c} \tag{10}$$

Damit lässt sich nun jegliche relativistische Koordinatentransformation in 1-dimensionaler Bewegungsrichtung beschreiben. Um diese Transformation in allen drei Raumrichtungen berechnen zu können, wird die Formel meist in Matrixform geschrieben:

$$B(v) = \begin{pmatrix} (\gamma-1)n_x^2 + 1 & (\gamma-1)n_x n_y & (\gamma-1)n_x n_z & -\beta\gamma n_x \\ (\gamma-1)n_y n_x & (\gamma-1)n_y^2 + 1 & (\gamma-1)n_y n_z & -\beta\gamma n_y \\ (\gamma-1)n_z n_x & (\gamma-1)n_z n_y & (\gamma-1)n_z^2 + 1 & -\beta\gamma n_z \\ -\beta\gamma n_x & -\beta\gamma n_y & -\beta\gamma n_z & \gamma \end{pmatrix} \tag{11}$$

$$X' = B(v)X \tag{12}$$

[1]Dies vereinfacht die Implementierung der Effekte deutlich, da nur die Lorentztransformation im Programm umgesetzt werden muss.

[2]Diese wird zB. in der klassischen Mechanik verwendet.

Dabei ist $\vec{n}$ der Einheitsvektor von $\vec{v}$ und v die Relativgeschwindigkeit gesehen von dem System X.

2.1.6 Relativistischer Dopplereffekt

Viele der oben genannten Effekte treten im alltäglichen Leben kaum auf, da große Geschwindigkeiten nötig sind, um diese sichtbar werden zu lassen. Anders sieht es mit dem Dopplereffekt aus, welcher vor allem im Zusammenhang mit Schall alltäglich auftritt. Fährt zum Beispiel ein Krankenwagen mit eingeschalteter Sirene an einem Beobachter vorbei, wird solange er sich auf den Beobachter zubewegt der Ton höher, da die Schallwellen durch die Geschwindigkeit des Krankenwagens für den Beobachter gestaucht erscheinen und somit die Frequenz steigt. Gleiches geschieht, sobald der Krankenwagen sich vom Beobachter entfernt mit dem Unterschied, dass die Wellen für den Beobachter gedehnt erscheinen und deshalb der Ton tiefer wird.

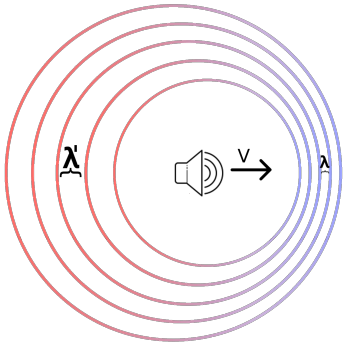


Abbildung 2: Dopplereffekt

Den gleichen Effekt kann man nahezu mit jeder Form von Welle beobachten. Da es sich bei Licht um eine Mischform aus Teilchen und Welle handelt, tritt auch hier der Effekt in Form von Farbverschiebungen auf. Um von der alten Wellenlänge nun unter Rücksichtnahme der Relativgeschwindigkeit die neue Wellenlänge berechnen zu können wird diese Formel verwendet:

$$f_r = \gamma(1 - \beta \cos \theta_s) \cdot f_s \tag{13}$$

θ_s = Skalarprodukt aus dem Lichtstrahl und der Geschwindigkeit
f_s = Die Frequenz im System des Senders
f_r = Die Frequenz im System des Empfängers

2.2 Numerische Umsetzung

Nachdem nun die physikalischen Prinzipien erläutert wurden, sollen nun die verwendeten Algorithmen und die explizite Umsetzung im Programm besprochen werden.

2.2.1 Idee des Programms

Die Idee hinter dem Programm ist relativ simpel. Das Ganze soll die relativistischen Effekte darstellen und dabei wie eine Sandbox funktionieren. So soll man verschiedene Objekte erzeugen und dabei Parameter wie zum Beispiel deren Geschwindigkeit verändern können. Die Darstellung der Effekte soll dabei nicht möglichst fotorealistisch sein, sondern es soll eher das Konzept und die optischen Auswirkungen der Formeln verdeutlicht werden.

2.2.2 Die richtige Entwicklungsumgebung

Um die richtige Entwicklungsumgebung zu finden, ist es vor allem wichtig zu wissen, welche Anforderungen das jeweilige Programm erfüllen soll. Da diese Anwendung allem voran für schnelles Rendern von dreidimensionalen Objekten ausgelegt sein soll, steht hauptsächlich die Performance im Vordergrund. Da diese Problematik meist auch bei Computerspielen auftritt, schien es sinnvoll, ebenso eine für Spiele ausgelegte Entwicklungsumgebung wie UNITY[3] zu nutzen. Hier ist vor allem die Kommunikation zwischen CPU und GPU leicht zu bewerkstelligen, was bei herkömmlichen Programmiersprachen oft nur mit großen Umwegen möglich ist.

2.2.3 Die Szene

Bevor überhaupt etwas angezeigt werden kann, muss zuerst eine für den Computer geeignete Beschreibung der zu visualisierenden Objekte vorliegen. In diesem Fall gibt es dafür eine Klasse `Scene`, welche eine Liste aller in der Simulation beteiligten Objekte beinhaltet.

2.2.4 Objekte

Da es sich hier um eine dreidimensionale Szene handelt, wird für die Beschreibung von Objekten der Ansatz mit Hilfe von Polygonen benutzt. Dabei besteht jedes Objekt, hier durch die Klasse `SceneObject` repräsentiert, aus einer beliebigen Anzahl von diesen. Um

[3] UNITY ist eine für Spiele ausgelegte Entwicklungsumgebung basierend auf `C#`.

die spätere Kollisionsprüfung zu vereinfachen, werden hier nur Dreiecke[4] (Im Programm die Klasse `Triangle`) verwendet.

2.2.5 Der Raytracing Algorithmus

Der erste Schritt, um nun ein Bild auf dem Computer aus einer dreidimensionalen Szene zu erzeugen, besteht darin, einen geeigneten Algorithmus für die Visualisierung zu finden. Für realistische Ergebnisse wird zum Beispiel häufig der Raytracing-Algorithmus verwendet, da dieser den Weg jedes einzelnen Lichtstrahls berechnet und somit physikalisch am nächsten an der Realität liegt. Aus diesem Grund eignet er sich auch besonders für den Zweck dieses Programms, da der bestehende Algorithmus nur um relativistische Annahmen erweitert werden muss. Das grundlegende Konzept besteht darin, dass vom Sensor einer virtuellen Kamera Lichtstrahlen ausgesendet werden, welche dann auf Schnittstellen mit Objekten untersucht werden. Trifft ein Strahl auf ein Objekt, so nimmt der dazugehörige Bildpunkt auf dem Sensor die Farbe des Objektes an beziehungsweise die des Hintergrundes, falls keine Kollision stattgefunden hat.

2.2.5.1 Die virtuelle Kamera

Die virtuelle Kamera ahmt in diesem Fall eine Lochkamera nach (Abb. 3). Um nun zu wissen, an welcher Stelle und in welche Richtung die Strahlen erzeugt werden sollen, sind die Position, Ausrichtung und Brennweite der Kamera sowie die Höhe und Breite des Sensors entscheidend. Der Strahl wird als Struktur definiert und setzt sich aus dem Richtungsvektor, welcher von der Pixelkoordinate zur Lochblende festgelegt ist, sowie dem Ortsvektor zusammen, welcher durch die Position dieser festgelegt ist. Damit der Strahl in die aktuelle Richtung der Kamera zeigt, muss man im Programm den Richtungsvektor mit der Transformationsmatrix `CameraToWorldMatrix` multiplizieren, welche praktischerweise schon durch UNITY gegeben ist.

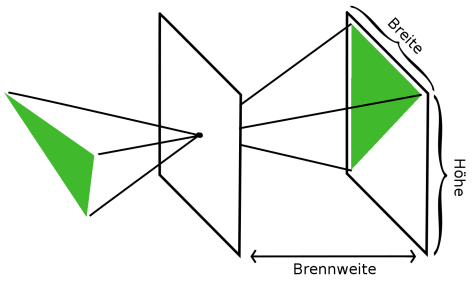


Abbildung 3: Ein einfaches Modell einer Lochkamera

[4]Dreiecke sind die Polygonen mit der geringsten Anzahl an Eckpunkten, welche aber immer noch eine Fläche abbilden.

2.2.5.2 Die Kollisionsprüfung

Für die Kollisionsprüfung wird nun über jedes Objekt der Szene iteriert und mit dem Möller-Trumbore Algorithmus[1] überprüft, ob der Strahl mit diesem kollidiert. Die Funktion `intersectsTriangle` gibt nun die Struktur `RayHit` zurück, welche Informationen über die Kollision beinhaltet. Falls keine Kollision stattgefunden hat, wird diese Struktur leer zurückgegeben. Da ein Strahl aber mit mehreren Objekten überschneiden kann, muss anschließend aus allen Treffern noch derjenige gesucht werden, welcher am nächsten zur Kamera liegt.

2.2.5.3 Schattierung

Nachdem der am nächsten liegende Kollisionspunkt gefunden wurde, gibt es nun mehrere Möglichkeiten zu verfahren. Um ein möglichst realistisches Bild zu erzeugen, könnten nun noch weitere Strahlen von diesem Punkt aus zurückverfolgt werden, um Effekte wie Reflexion anzuzeigen, welche dann anhand der Materialeigenschaften berechnet werden könnten. Um Rechenleistung zu sparen und die relativistischen Effekte in den Vordergrund zu stellen, wird hier einzig die Farbe des Objektes für die Entstehung des Bildes berücksichtigt.

2.2.6 Erweiterung des Raytracing-Algorithmus

Um den Raytracing Algorithmus in seiner derzeitigen Form um die relativistischen Annahmen zu erweitern, sind hauptsächlich zwei große Veränderungen nötig, die auf den Ideen von Howard, Dance und Kitchen[2] basieren:

2.2.6.1 Erweiterung um die Lichtlaufzeit

Ein Problem des Algorithmus in seiner jetzigen Form besteht darin, dass angenommen wird, dass sich Licht mit unendlich großer Geschwindigkeit bewegt. Diese Annahme funktioniert sehr gut in alltäglichen Situationen, da sich selbst bewegte Objekte zwischen dem Senden des Lichtstrahls und dessen Ankunft nur minimal bewegen. Bei extrem hohen Geschwindigkeiten kann dies allerdings zu großen Abweichungen führen, weshalb der Strahl um die Zeitdimension erweitert wird, damit bei der Kollision die Bewegung der Objekte berücksichtigt werden kann. Des Weiteren spielt auch das Modell der Kamera eine große Rolle. Bei dem verwendeten Lochkameramodell muss zusätzlich noch die Zeit berücksichtigt werden, die ein Strahl vom Sensor bis zur Lochblende benötigt, da zum Beispiel die Ränder des Sensors eine größere Distanz zur Blende haben als die Mitte. Hier wird klar, wie groß die Bedeutung des jeweiligen Kameramodells in der Visualisierung ist. Deshalb können die relativistischen Effekte, die in dem

Programm auftreten nicht die Wahrnehmung des Menschen oder einer DSLR Kamera darstellen.

2.2.6.2 Erweiterung mit der Lorentztransformation

Eine weitere Problematik ist, dass nach Einstein Systeme mit unterschiedlichen Inertialgeschwindigkeiten in unterschiedlichen Koordinatensystemen liegen. Damit man nun Kollisionen korrekt berechnen kann, muss man die Koordinaten des Lichtstrahls in das Koordinatensystem des bewegten Objektes umrechnen, was mit Hilfe der Lorentztransformation möglich ist. In der Methode `getTransformedRay` wird nun jeweils der Richtungs- sowie der Ortsvektor mit der Transformationsmatrix multipliziert, um dann einen neuen Strahl in dem entsprechenden Koordinatensystem zurückzugeben.

2.2.6.3 Umsetzung im Programm

Zuerst werden die beiden Vektoren des Strahls in der Methode `RenderKernel` um die Zeitdimension erweitert:

$$\vec{r}_{orig} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ ct_0 \end{pmatrix} \quad \text{und} \quad \vec{r}_{dir} = \begin{pmatrix} x \\ y \\ f \\ -ct_1 \end{pmatrix} \tag{14}$$

Dabei steht t_0 für die Zeit, wenn der Strahl an der Lochblende auftrifft und t_1 für die Zeit, die der Strahl von dem Sensor zu dieser benötigt. f beschreibt in diesem Fall die Brennweite. Aus der Natur des Lochkameramodells ergibt sich also für die beiden Zeiten:

$$ct_0 = 0 \tag{15}$$

$$ct_1 = \left| \begin{pmatrix} x \\ y \\ f \end{pmatrix} \right| \tag{16}$$

Damit nun der Strahl in der Funktion `getTransformedRay` in das Koordinatensystem des Objektes transformiert werden kann, wird der Ortsvektor des Strahls mit den Raumzeitkoordinaten von dem Objekt addiert und anschließend die Lorentzmatrix anhand der Geschwindigkeit des Objektes kreiert. Nun werden die beiden Vektoren

des Strahles jeweils noch mit dieser Matrix multipliziert, um daraus dann einen neuen transformierten Strahl in `createRay` zu erstellen und zurückzugeben.

2.2.7 Der Dopplereffekt

Auch für den Dopplereffekt müssen einige Änderungen und Ergänzungen an dem Raytracing-Algorithmus vorgenommen werden. Diese umfassen aber gänzlich neue Aspekte und sollen im Folgenden gesondert dargestellt werden.

2.2.7.1 Farben auf Monitoren

Da Monitore Farben nicht in einer Wellenlänge abgeben, sondern diese immer aus einem Spektrum von den drei Farben Rot, Grün und Blau bestehen, ergibt sich eine Schwierigkeit bei der Darstellung, wenn genaue Wellenlängen angezeigt werden sollen. Zusätzlich hängt die Farbdarstellung auch noch speziell von dem verwendeten Monitor ab, weshalb es auf unterschiedlichen Systemen fast unmöglich ist, immer ein physikalisch korrektes Bild anzuzeigen.

2.2.7.2 Der RGB-Farbraum

Um Farben anzuzeigen, haben sich mehrere Farbräume entwickelt, wobei auf Monitoren der additive RGB-Farbraum verwendet wird. Dabei besteht genau wie bei einem Pixel jede Farbe aus den drei Primärfarben Rot, Grün und Blau. Aufgrund der Ähnlichkeit zur Farbwahrnehmung menschlichen Sehens hat sich dieser Farbraum zur Darstellung von Farben auf Monitoren durchgesetzt. Dies ist auch der Grund weshalb immer in diesen Farbraum umgerechnet werden muss, damit Farben angezeigt werden können.

2.2.7.3 Farben von Objekten

Damit nun auch die Farben von Objekten durch den Dopplereffekt beeinflusst werden können, ist es wichtig, dass sie in ihrer Wellenlänge angegeben sind und nicht in spezifischen Farbräumen. Deshalb emittiert in diesem Programm jedes Objekt eine Wellenlänge, die der grünen Farbe entspricht, da diese Wellenlänge ungefähr in der Mitte des menschlich sichtbaren Spektrums liegt. Dadurch ist auf jeder Seite des Spektrums noch Platz für Farbverschiebungen durch den Dopplereffekt.

2.2.7.4 Wellenlänge in RGB umrechnen

Für die Umrechnung von Wellenlänge in den RGB-Farbraum sind zwei Schritte notwendig. Zuerst muss die Wellenlänge in den nach der CIE[5] bestimmten

[5] International Commission on Illumination

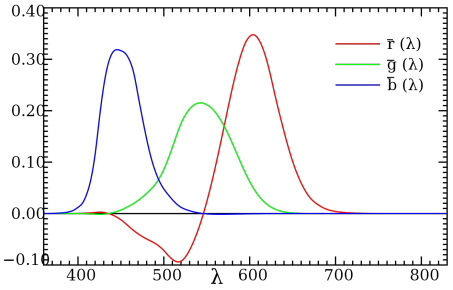


Abbildung 4:
RGB-Farbanpassungsfunktionen [3]

XYZ-Farbraum umgerechnet werden, welcher die Farbwahrnehmung eines durchschnittlichen Menschen imitiert. Der zweite Schritt besteht darin, dass der XYZ-Farbraum in einen RGB-Farbraum umgewandelt werden muss, damit der Bildschirm die Farbe anzeigen kann. Dies geschieht in der Methode `wave2rgb` (Abb. 4), welche zuvor die Wellenlänge noch auf den für das menschliche Auge sichtbaren Teil des Lichtspektrums[6] beschränkt und die Farbe Schwarz für den Fall von nicht sichtbarem Licht zurückgibt.

2.2.7.4.1 Wellenlänge in XYZ-Farbraum umrechnen

In der Funktion `wave2xyz` wird nach der Methode von Wyman, Sloan und Shirley[4] durch die Summe aus Gauß-Funktionen die Wellenlänge in den entsprechenden XYZ-Farbraum umgerechnet.

2.2.7.4.2 XYZ in RGB umrechnen

Die Umrechnung in RGB erfolgt in der Funktion `xyz2rgb` mithilfe einer linearen Transformation durch eine 3 mal 3 Matrix. Die Werte der Matrix werden durch den jeweiligen Farbraum festgelegt, in diesem Fall wird das Ganze in Beta RGB umgerechnet.

3 Schlussbetrachtung

3.1 Verbesserungen und Grenzen des Programms

Die Simulation umfasst zwar schon einige der relativistischen Effekte, bietet aber trotzdem noch einen großen Raum für Verbesserungen und auch Vervollständigungen.

3.1.1 Beschleunigung von Objekten

Bis jetzt ist es in dem Programm noch nicht möglich, die Geschwindigkeit eines Objektes dynamisch zu ändern. Um die Effekte, die bei beschleunigten Objekten auftreten zu visualisieren, müssten die bisherigen Vorgehensweisen komplett umgestaltet werden und um ein Vielfaches komplexer gemacht werden.

[6] In diesem Fall wird eine Wellenlänge von 380 bis 710 Nanometer für den Menschen als sichtbar angenommen.

3.1.2 Licht und Schatten

Des Weiteren müsste die Simulation noch um eine realistische Darstellung von Licht erweitert werden. Es können in diesem Programm zwar Objekte Licht emittieren, jedoch werden keine umliegenden Objekte bestrahlt, weshalb Effekte wie Schattierungen noch fehlen.

3.1.3 Die relativistische Aberration

Die relativistische Aberration stellt noch einen sehr wichtigen Punkt dar, der in dieser Arbeit noch nicht beachtet wurde. Diese sagt im Groben aus, dass gesendete Lichtstrahlen einer Lichtquelle in die Bewegung dieser geneigt werden. Das hat zur Folge, dass die Helligkeitsverteilung auf Objekten in die Bewegungsrichtung verschoben wird und somit ein Suchscheinwerfereffekt eintritt.

Abbildung 5: Ein Vorbeiflug an der Sonne mit 99.9% der Lichtgeschwindigkeit unter Berücksichtigung der relativistischen Aberration [5]

3.1.4 Verbesserung der Leistung

Das Programm in seiner derzeitigen Form fordert viel Leistung und benötigt dementsprechend gute technische Komponenten[7]. Aber auch mit entsprechendem Equipment kommt ein leistungsstarker PC bei vielen Objekten mit hoher Polygonenanzahl schnell an seine Grenzen.

3.2 Ausblick

Abschließend soll noch angemerkt werden, dass dieses Programm nicht das erste in seinem Gebiet ist. Viele der oben genannten Verbesserungen findet man deshalb schon in anderen Simulationen. Dennoch haben alle diese Programme gemeinsam, dass sie

[7] Bei diesem Programm wird vor allem eine gute Grafikkarte benötigt.

eine sehr gute Hardware voraussetzen. Bleibt abzuwarten, ab wann der durchschnittliche Computer über die erforderliche Leistung verfügt, um eine solche Simulation in erweiterter Komplexität und Echtzeit laufen zu lassen.

4 Literaturverzeichnis

Literatur

[1] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[2] Andrew Howard, Sandy Dance, and Les Kitchen. Relativistic ray-tracing: Simulating the visual appearance of rapidly moving objects. 10 1995.

[3] Wikipedia contributors. Cie 1931 color space — Wikipedia, the free encyclopedia, 2020. [Online; accessed 30-August-2020].

[4] Chris Wyman, Peter-Pike Sloan, and Peter Shirley. Simple analytic approximations to the CIE XYZ color matching functions. *Journal of Computer Graphics Techniques (JCGT)*, 2(2):1–11, July 2013.

[5] Ute Kraus, Hanns Ruder, Daniel Weiskopf und Corvin Zahn. Was Einstein noch nicht sehen konnte. *Physik Journal 1*, (7/8):77–82, 2002.

A Anhang

A.1 Anmerkung & Erklärung

In dieser Arbeit werden verschiedene Schriftstile verwendet, um die Bedeutung bestimmter Wörter hervorzuheben:

- `Monospaced Font:` Alle Variablen und Funktionsnamen, die im Programm vorkommen, werden in einer sog. "monospaced"(= einheitlicher Abstand) Schriftart dargestellt.
- KAPITÄLCHEN: Diese werden dazu benutzt, Namen ein eigenes Erscheinungsbild zu geben (zB. UNITY)

A.2 Abbildungsverzeichnis

A.3 Code

Der ganze Source-Code befindet sich auf dem beiliegenden USB-Stick. Um den Inhalt auf das Wesentliche zu beschränken, sind nur die wichtigsten Abschnitte des Programmes in gedruckter Form nach der Seminararbeit enthalten:

1. „RaytracingCompute.compute": Dieser Teil des Programms ist das Herzstück der Simulation. Darin befindet sich der Raytracer sowie die dazugehörigen relativistischen Erweiterungen. Er umfasst 9 Seiten mit 448 Zeilen.
2. „RaytracingMaster.cs": In diesem Abschnitt befindet sich die Schnittstelle zwischen Grafikkarte und Prozessor. Er umfasst 3 Seiten mit 129 Zeilen.

A.4 USB-Stick Inhalt

Auf dem beiliegenden USB-Stick befindet sich der Programm-Code (40 Seiten und 1671 Zeilen), die Seminararbeit in LaTeX Code, das Handbuch als PDF, alle Dateien

des UNITY Projektes und die Simulation (`setup.exe`, alternativ auch in „RawExe" `Special Relativity Project.exe`).

A.5 Materialverzeichnis

Folgende Materialien müssen der Seminararbeit beiliegen, um ein optimales Verständnis zu gewährleisten:

- Seminararbeit (20 Seiten)
- ausgedruckter Source-Code (12 Seiten - 564 Zeilen)
- Handbuch für die Simulation (8 Seiten)
- USB-Stick mit der Simulation und allen Texten

A.6 Erklärung

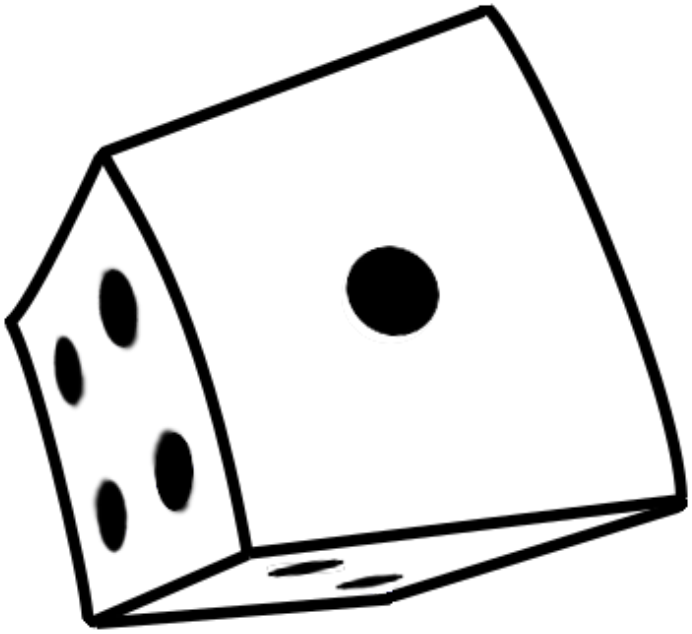
Hiermit erkläre ich, dass ich die vorliegende Arbeit und das dazugehörige Programm selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Lengdorf, 21.10.2020

Christiansen Lars

Visualisierung der Relativitätstheorie

Teil 2: Implementierung



Code:

„RaytracingCompute.compute”: 9 Seiten

„RaytracingMaster.cs”: 3 Seiten

```
// Each kernel tells which function to compile; you can have many kernels
#pragma kernel RenderKernel
#pragma enable_d3d11_debug_symbols

// resulting image
RWTexture2D<float4> Result;
float focalLengthFactor;

// 4 x 4 float Matrix for transformations
float4x4 _CameraToWorld;
float4x4 _CameraInverseProjection;

// environmental texture
Texture2D<float4> _SkyboxTexture;
SamplerState sampler_SkyboxTexture;

// constants
static const float c = 1.0f;
static const float PI = 3.14159265f;

// realism indicator is determined by the user
uint realism = 1;

float currentTime;

struct Triangle
{
    float3 vertex0;
    float3 vertex1;
    float3 vertex2;

    float3 edge0;
    float3 edge1;
    float3 edge2;

    float color;
    float3 velocity;
    float3 offset;

    float time;
};

// Scene
StructuredBuffer<Triangle> _triangles;

struct Ray
{
    float4 origin;
    float4 dir;
};

struct RayHit
{
    Triangle tri;
    float3 position;
    float time;
```

```
57      float distance;
58      float3 normal;
59      float color;
60  };
61
62  RayHit createRayHit()
63  {
64      RayHit hit;
65      hit.position = float3(0.0f, 0.0f, 0.0f);
66      hit.time = 0.0f;
67      hit.distance = 1.#INF;
68      hit.normal = float3(0.0f, 0.0f, 0.0f);
69      hit.color = 0;
70      return hit;
71  }
72
73  Ray createRay(float4 origin, float4 dir)
74  {
75      Ray r;
76      r.origin = origin;
77      r.dir = dir;
78      return r;
79  }
80
81  /*
82  This method creates and returns the lorentzmatrix for the associated speed v
83  */
84  float4x4 getLorentzMatrix(float3 v)
85  {
86      float B = length(v) / c;
87      float y = 1 / sqrt(1 - (B * B));
88
89      float3 vn = normalize(v);
90      float nx = vn.x;
91      float ny = vn.y;
92      float nz = vn.z;
93
94      float4 col1 = float4(y, -y * B * nx, -y * B * ny, -y * B * nz);
95      float4 col2 = float4(-y*B*nx, 1 + (y - 1)*nx*nx, (y - 1)*nx*ny, (y - 1) *
          nx * nz);
96      float4 col3 = float4(-y*B*ny, (y - 1)*nx*ny, 1 + (y - 1)*ny*ny, (y - 1) *
          ny * nz);
97      float4 col4 = float4(-y*B*nz, (y - 1)*nx*nz, (y - 1)*nz*ny, 1 + (y - 1) *
          nz * nz);
98
99      float4x4 m = float4x4(col1, col2, col3, col4);
100
101     return m;
102 }
103
104 /*
105 This method creates and returns the galileomatrix for the associated speed v
106 */
107 float4x4 getGalileoMatrix(float3 v)
108 {
109     float B = length(v) / c;
```

```
110
111     float3 vn = normalize(v);
112     float nx = vn.x;
113     float ny = vn.y;
114     float nz = vn.z;
115
116     float4 col1 = float4(1, -B * nx, -B * ny, -B * nz);
117     float4 col2 = float4(-B * nx, 1, 0, 0);
118     float4 col3 = float4(-B * ny, 0, 1, 0);
119     float4 col4 = float4(-B * nz, 0, 0, 1);
120
121     float4x4 m = float4x4(col1, col2, col3, col4);
122
123     return m;
124 }
125
126 /*
127 This method returns a new Ray which gets transformed relativistically into the reference frame of the Triangle
128 */
129 Ray getTransformedRay(Ray ray, Triangle t)
130 {
131     float3 v = t.velocity;
132     Ray r = createRay(ray.origin + float4(0, t.offset), ray.dir);
133     r.origin.x = t.time;
134
135     float4x4 m = getLorentzMatrix(v);
136     float4 dir = mul(m, float4(r.dir));
137     float4 orig = mul(m, float4(r.origin));
138
139     return createRay(orig, dir);
140 }
141
142 /*
143 This method returns a new Ray which gets transformed non relativistically into the reference frame of the Triangle
144 */
145 Ray getGalileoRay(Ray ray, Triangle t)
146 {
147     float3 v = t.velocity;
148     Ray r = createRay(ray.origin + float4(0, t.offset), ray.dir);
149     r.origin.x = t.time;
150
151     float4x4 m = getGalileoMatrix(v);
152     float4 dir = mul(m, float4(r.dir));
153     float4 orig = mul(m, float4(r.origin));
154
155     return createRay(orig, dir);
156 }
157
158 // Moller Trumbore algorithm for intersection detection:
159 static const float EPSILON = 0.0000001;
160
161 RayHit intersectsTriangle(Ray ray, uint triangleIndex)
162 {
163
```

```
164     float3 dir = ray.dir.yzw;
165     float3 origin = ray.origin.yzw;
166
167     Triangle tris = _triangles[triangleIndex];
168
169     float a, f, u, v;
170     float3 h, s, q;
171     float3 edge1 = tris.edge0; // 2 edges from vertex 0 to vertex 1 and 2
172     float3 edge2 = tris.edge1;
173
174     h = cross(dir, edge2);
175     a = dot(edge1, h);
176
177     if (a > -EPSILON && a < EPSILON)
178     {
179         return createRayHit(); // no intersection because ray is parallel to
              this triangle
180     }
181
182
183     f = 1.0 / a;
184     s = origin - tris.vertex0;
185
186     u = f * dot(s, h);
187
188     if (u < 0.0 || u > 1.0)
189     {
190         return createRayHit();
191     }
192
193     q = cross(s, edge1);
194     v = f * dot(dir, q);
195
196
197     if (v < 0.0 || u + v > 1.0)
198     {
199         return createRayHit();
200     }
201
202     float t = f * dot(edge2, q);
203
204     float3 relativePos = ray.dir * t;
205     float distance = length(relativePos);
206
207     if (t > EPSILON) // intersection occurs
208     {
209         RayHit hit = createRayHit();
210         hit.distance = distance;
211         hit.position = origin + relativePos;
212         hit.normal = normalize(cross(edge1, edge2));
213         hit.color = tris.color;
214         hit.tri = tris;
215
216         hit.time = ray.origin.x - (distance / c); // time at intersection
217         return hit;
218     }
```

```
219
220     return createRayHit();
221 }
222
223 // CIE color matching functions:
224
225 double gaussian(double x, double alpha, double mu, double sigma1, double
      sigma2)
226 {
227     double squareRoot = (x - mu) / (x < mu ? sigma1 : sigma2);
228     return alpha * exp(-(squareRoot * squareRoot) / 2);
229 }
230
231 double3 wave2xyz(double wavelength)
232 {
233     double3 xyz;
234     xyz[0] = gaussian(wavelength, 1.056, 5998, 379, 310)
235         + gaussian(wavelength, 0.362, 4420, 160, 267)
236         + gaussian(wavelength, -0.065, 5011, 204, 262);
237
238     xyz[1] = gaussian(wavelength, 0.821, 5688, 469, 405)
239         + gaussian(wavelength, 0.286, 5309, 163, 311);
240
241     xyz[2] = gaussian(wavelength, 1.217, 4370, 118, 360)
242         + gaussian(wavelength, 0.681, 4590, 260, 138);
243
244     return xyz;
245 }
246
247 double3 xyz2rgb(float3 xyz)
248 {
249     double3 rgb;
250     double3 r1 = double3(1.6832270, -0.4282363, -0.2360185);
251     double3 r2 = double3(-0.7710229, 1.7065571, 0.0446900);
252     double3 r3 = double3(0.0400013, -0.0885376, 1.2723640);
253     double3x3 m = float3x3(r1, r2, r3);
254
255     rgb = mul(m, xyz);
256     return normalize(rgb);
257 }
258
259 float3 wave2RGB(float w)
260 {
261     // clip val between 380 and 710 nm because other values dont make sense
262     if (w > 710 || w < 380)
263     {
264         return float3(0, 0, 0);
265     }
266
267     //convert nm to armstrong
268     w = w * 10;
269
270     double3 xyz = wave2xyz(w);
271     return xyz2rgb(xyz);
272 }
273
```

```
274 double dopplerShift(double wavelength, Ray r, float3 v)
275 {
276     float B = length(v) / c;
277     float y = 1 / sqrt(1 - (B * B));
278 
279     // formula : fr = y (1 - Bcos(theta)) * fs
280 
281     float fs = c / wavelength;
282     float fr = fs * y * (1 - B*dot(normalize(r.dir.yzw), normalize(v)));
283 
284     float newWavelength = c / fr;
285     return newWavelength;
286 }
287 
288 // basic vector operations
289 float getAngle(float3 a, float3 b)
290 {
291     return acos((dot(a, b)) / (length(a) * length(b)));
292 }
293 
294 float3 reflectRay(float3 d, float3 n)
295 {
296     d = normalize(d); //ray
297     n = normalize(n); // normal of surface
298     //formula: d - 2 * n * cos(theta);
299     return d - 2 * n * cos(getAngle(d, n));
300 }
301 
302 // relativistic abberation:
303 //(This method is not used in this programm because no formula for the
      intensity of the ligth has been found)
304 float abberation(Ray r, RayHit hit)
305 {
306     float B = length(hit.tri.velocity) / c;
307 
308     // angle between light source (envoronment texture) and the velocity of
        the triangle
309     float theta_s = getAngle(reflectRay(r.dir.yzw, hit.normal),
        hit.tri.velocity);
310 
311     float newTheta = acos((cos(theta_s) - B) / (1 - B * cos(theta_s)));
312 
313     // get new Intensity:
314     float newIntensity; // = ?
315 
316     return newIntensity;
317 }
318 
319 // get the associated color of the world to a ray
320 float3 shadeSkyBox(float4 _dir)
321 {
322     float3 dir = _dir.yzw;
323     dir = normalize(dir);
324     float theta = acos(-dir.y) / -PI;
325     float phi = atan2(dir.x, -dir.z) / -PI * 0.5f;
326     float3 result = _SkyboxTexture.SampleLevel(sampler_SkyboxTexture, float2
```

```
          (phi, theta), 0).xyz * 1.3f;
327     return result;
328 }
329
330 // in this method the Ray gets traced
331 RayHit Trace(Ray ray)
332 {
333     RayHit hit = createRayHit();
334
335     uint numTris, stride;
336     _triangles.GetDimensions(numTris, stride);
337
338     uint index = 0;
339
340     // iterate trough all triangles in the scene an check for colission
341     for (uint i = 0; i < numTris; i++)
342     {
343         RayHit currentHit;
344         Ray r;
345
346         if (realism >= 1) // lorentztransformation
347         {
348             r = getTransformedRay(ray, _triangles[i]);
349             currentHit = intersectsTriangle(r, i);
350         }
351         else // galileotransformation
352         {
353             r = getGalileoRay(ray, _triangles[i]);
354             currentHit = intersectsTriangle(r, i);
355         }
356
357         // check which colission is closer to the camera --> this object can
            be seen
358         if (currentHit.distance < hit.distance)
359         {
360             hit = currentHit;
361         }
362
363     }
364
365     return hit;
366 }
367
368 float3 shade(Ray ray, RayHit hit)
369 {
370     // resulting color
371     float3 result;
372
373     if (hit.distance < 1.#INF)
374     {
375         // wavelength
376         double w;
377         if (realism >= 3)
378         {
379             w = dopplerShift(hit.color, ray, hit.tri.velocity);
380         }
```

```
381         else
382         {
383             w = hit.color;
384         }
385 
386         float3 color = wave2RGB(w);
387 
388         // reflect the ray at the surface to give some 3D-illusion
389         result = (color + 0.15 * shadeSkyBox(float4(0, reflectRay(ray.dir.yzw, 
                hit.normal)))) / 1.15;
390 
391         return result;
392     }
393 
394     result = shadeSkyBox(ray.dir);
395     return result;
396 
397 }
398 
399 [numthreads(8, 8, 1)]
400 void RenderKernel(uint3 id : SV_DispatchThreadID)
401 {
402     // width and height of the displayed image
403     uint width;
404     uint height;
405     Result.GetDimensions(width, height);
406 
407     // id is a Vector with the current Pixel coordinate
408     int x = id.x - width / 2;
409     int y = -(id.y - height / 2);
410 
411     // focalLengthfactor is there to preserve the focal length even when the 
          window size gets changed
412     uint focalLength = 350 * focalLengthFactor;
413 
414     float4 imageCoords;
415 
416     if (realism >= 2)
417     {
418         float originToImageTime = length(float3(x, y, focalLength));
419         imageCoords = float4(-originToImageTime, x, y, focalLength);
420     }
421     else
422     {
423         imageCoords = float4(0, x, y, focalLength);
424     }
425 
426 
427     float3 cam = mul(_CameraToWorld, float4(0.0f, 0.0f, 0.0f, 1.0f)).xyz;
428 
429     float4 ORIGIN = float4(0, 0, 0, 0);
430     Ray ray = createRay(ORIGIN - float4(0, cam.x, cam.y, cam.z), imageCoords);
431     ray.dir.yzw = mul(_CameraToWorld, float4(ray.dir.yzw, 0)).xyz;
432 
433 
434     RayHit besthit = Trace(ray);
```

```
435     float3 color = shade(ray, besthit);
436
437     // set the pixel of the canvas to the resulting color
438     Result[id.xy] = float4(color, 1);
439 }
440
```

```
1  using UnityEngine;
2  using UnityEngine.UI;
3
4  public class RaytracingMaster : MonoBehaviour
5  {
6      private Vector2 resolution;
7
8      public Slider realismSlider;
9      public RawImage rawImage;
10
11     public ComputeShader RayTracingShader;
12     public Texture skyboxTexture;
13
14     private Camera camera;
15     private Scene scene;
16
17     private ComputeBuffer triangles;
18     private RenderTexture renderTexture;
19
20     private static float c = 1f;
21
22     private void Awake()
23     {
24         resolution = new Vector2(Screen.width, Screen.height);
25         InitRenderTexture();
26         onWindowResize();
27         camera = GetComponent<Camera>();
28         scene = new Scene();
29         triangles = scene.getAllTrisBuffered();
30     }
31
32     // gets called on every frame
33     private void Update()
34     {
35         scene.Update(Time.deltaTime);
36         if (resolution.x != Screen.width || resolution.y != Screen.height)
37         {
38             onWindowResize();
39
40             resolution.x = Screen.width;
41             resolution.y = Screen.height;
42         }
43     }
44
45     private void SetShaderParameters()
46     {
47         InitRenderTexture();
48         triangles.Release();
49         triangles = scene.getAllTrisBuffered();
50
51         RayTracingShader.SetFloat("c", c);
52         RayTracingShader.SetMatrix("_CameraToWorld",
             camera.cameraToWorldMatrix);
53         RayTracingShader.SetMatrix("_CameraInverseProjection",
             camera.projectionMatrix.inverse);
54         RayTracingShader.SetBuffer(0, "_triangles", triangles);
```

```
55          RayTracingShader.SetTexture(0, "_SkyboxTexture", skyboxTexture);
56          RayTracingShader.SetFloat("currentTime", Time.time);
57          RayTracingShader.SetInt("width", renderTexture.width);
58          RayTracingShader.SetInt("height", renderTexture.height);
59          RayTracingShader.SetInt("realism", (int)realismSlider.value);
60          RayTracingShader.SetFloat("focalLengthFactor", (float)(Screen.width) /
               1920f);
61      }
62
63
64      private void OnRenderImage(RenderTexture source, RenderTexture
          destination)
65      {
66          SetShaderParameters();
67          Render(destination);
68      }
69
70      private void Render(RenderTexture dest)
71      {
72          RayTracingShader.SetTexture(0, "Result", renderTexture);
73
74          int threadGroupsX = Mathf.CeilToInt(camera.scaledPixelWidth / 8.0f);
75          int threadGroupsY = Mathf.CeilToInt(camera.scaledPixelHeight / 8.0f);
76
77
78          RayTracingShader.Dispatch(0, threadGroupsX, threadGroupsY, 1);
79
80          // Blit texture to our Screen
81          Graphics.Blit(renderTexture, dest);
82      }
83
84      public void setScene(Scene s)
85      {
86          this.scene = s;
87      }
88
89      private void InitRenderTexture()
90      {
91          if (renderTexture == null)
92          {
93              renderTexture = new RenderTexture((int)Screen.width, (int)
                  Screen.height,
94                  0, RenderTextureFormat.ARGBFloat,
                      RenderTextureReadWrite.Linear);
95
96              renderTexture.enableRandomWrite = true; // for GPU access
97              renderTexture.Create();
98          }
99
100         rawImage.texture = renderTexture;
101     }
102
103     private void OnApplicationQuit()
104     {
105         triangles.Dispose();
106     }
```

```
107
108     public Scene GetScene()
109     {
110         return this.scene;
111     }
112
113     private void onWindowResize()
114     {
115         int width = (int)(Screen.width * 0.8);
116         renderTexture = new RenderTexture((int)Screen.width, (int)
              Screen.height,
117                 0, RenderTextureFormat.ARGBFloat,
                      RenderTextureReadWrite.Linear);
118
119         renderTexture.enableRandomWrite = true; // for GPU access
120         renderTexture.Create();
121         rawImage.texture = renderTexture;
122     }
123 }
124
```