

FH JOANNEUM - UNIVERSITY OF APPLIED SCIENCES

Dreidimensionale Repräsentation von Algorithmen in Unity

Bachelorarbeit 1

zur Erlangung des akademischen Grades „Bachelor of Science in Engineering“
eingereicht am Studiengang Informationsmanagement

Verfasser:

Lukas Schneider

Betreuer:

FH-Prof. Dipl.-Ing. Dr. Alexander Nischelwitzer

[Graz, 2017]

EHRENWÖRTLICHE ERKLÄRUNG

Ich erkläre ehrenwörtlich, dass ich die vorliegende Bachelorarbeit selbstständig angefertigt und die mit ihr verbundenen Tätigkeiten selbst erbracht habe. Ich erkläre weiters, dass ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle aus gedruckten, ungedruckten oder dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte sind gemäß den Regeln für gutes wissenschaftliches Arbeiten zitiert und durch Fußnoten bzw. durch andere genaue Quellenangaben gekennzeichnet.

Die vorliegende Originalarbeit ist in dieser Form zur Erreichung eines akademischen Grades noch keiner anderen Hochschule vorgelegt worden. Diese Arbeit wurde in gedruckter und elektronischer Form abgegeben. Ich bestätige, dass der Inhalt der digitalen Version vollständig mit dem der gedruckten Version übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

(Unterschrift)

Lukas Schneider

[Ort, Datum]

INHALTSVERZEICHNIS

1	<u>EINLEITUNG</u>	1
1.1	PROBLEMSTELLUNG	1
1.2	ZIELSETZUNG	2
2	<u>3D-GRUNDLAGEN IN UNITY UND C#</u>	3
2.1	3D-ACHSEN UND KOORDINATENSYSTEM	3
2.2	3D-PUNKTE	3
2.2.1	ORTSVEKTOREN	3
2.2.2	BERECHNUNGEN MIT EINEM PUNKT	4
2.2.3	BERECHNUNGEN MIT ZWEI PUNKTEN	4
2.2.3.1	Vektor zwischen zwei Punkten	4
2.2.3.2	Distanz zwischen zwei Punkten	4
2.2.4	DARSTELLUNG VON PUNKTEN	5
2.3	GERADEN	6
2.3.1	RICHTUNGSVEKTOREN	6
2.3.2	GERADE MITTELS RICHTUNGSVEKTOR	7
2.3.3	GERADE DURCH ZWEI PUNKTE	7
2.3.4	LIEGT EIN PUNKT AUF EINER GERADEN?	8
2.3.5	ABSTAND EINES PUNKTES ZU EINER GERADEN	8
2.3.6	DARSTELLUNG VON GERADEN	9
2.4	STRAHLEN, RAYS	9
2.4.1	DARSTELLUNG VON RAYS	10
2.5	STRECKEN	10
2.5.1	LIEGT EIN PUNKT AUF EINER STRECKE?	11
2.6	EBENEN	11
2.6.1	LIEGT EIN PUNKT AUF EINER EBENE?	12
2.6.2	ABSTAND EINES PUNKTES ZU EINER EBENE	12
2.6.3	SCHNITTPUNKT LINIE-EBENE	13
2.6.4	SCHNITT EBENE-EBENE	13
2.6.5	DARSTELLUNG VON EBENEN	14
3	<u>MATHEMATISCHE & PHYSIKALISCHE GRUNDLAGEN</u>	15
3.1	FORMELN	15
3.1.1	KONSTANTE BEWEGUNG (GLEICHFÖRMIGE BEWEGUNG, KONSTANTE TRANSLATION)	15
3.1.1.1	Konstante Bewegung als Funktion der Zeit	15
3.1.1.2	Konstante Bewegung als inkrementelle Änderung	16
3.1.2	BESCHLEUNIGUNG & VERZÖGERUNG	16
3.1.2.1	Beschleunigung als Funktion der Zeit	16
3.1.2.2	Beschleunigung als inkrementelle Änderung	16
3.2	GRAVITATIONS-KRAFT	16
3.3	FREIER FALL	17
3.3.1	FREIER FALL OHNE LUFTWIEDERSTAND	17

3.3.1.1	Streckenformel als Funktion der Zeit	17
3.3.1.2	Geschwindigkeitsformel als Funktion der Zeit	17
3.3.2	FREIER FALL MIT LUFTWIDERSTAND	18
3.3.2.1	Widerstandskraft	18
3.3.2.2	Gewichtskraft	18
3.3.2.3	Fallbeschleunigung	18
3.3.2.4	Geschwindigkeitsfunktion als Funktion der Zeit	18
3.3.2.5	Ortsfunktion als Funktion der Zeit	19
3.4	WURFPARABEL	19
3.4.1	WURFWEITE	19
3.4.2	WURFHÖHE	19
3.4.3	BESCHLEUNIGUNG	19
3.4.4	GESCHWINDIGKEIT	20
3.4.5	ZURÜCKGELEGTE STRECKE	20

4 3D-ALGORITHMEN, BEWEGUNG UND SIMULATION **21**

4.1	MONOBEHAVIOUR	21
4.1.1	DIE UPDATE-FUNKTION	21
4.1.2	DIE FIXEDUPDATE-FUNKTION	22
4.2	TIME UND METHODEN VON TIME	22
4.2.1	TIME.DELTATIME	22
4.2.2	TIME.FIXEDDELATIME	23
4.2.3	TIME.TIME	23
4.2.4	TIME.FIXEDTIME	23
4.3	TRANSFORM-VARIABLEN UND FUNKTIONEN	23
4.4	ALGORITHMEN	24
4.4.1	KONSTANTE BEWEGUNG	24
4.4.1.1	Konstante Bewegung als Funktion der Zeit	24
4.4.1.2	Konstante Bewegung als inkrementelle Änderung	24
4.4.2	BESCHLEUNIGUNG & VERZÖGERUNG	25
4.4.2.1	Beschleunigung als Funktion der Zeit	25
4.4.2.2	Beschleunigung als inkrementelle Änderung	25
4.4.2.3	Verzögerung als Funktion der Zeit	25
4.4.2.4	Verzögerung als inkrementelle Änderung	25
4.4.3	ROTATION	26
4.4.3.1	Rotation als Funktion der Zeit	26
4.4.3.2	Rotation als inkrementelle Änderung	26
4.4.4	WURFPARABEL	26
4.4.5	REIBUNGSSIMULATION	26
4.4.6	GRAVITATION	27
4.4.7	ELASTISCHE BEWEGUNG	27
4.4.8	ZENOS PARADOXON	29

5 VISUALISIERUNG UND DEMONSTRATIONEN **30**

5.1	ANWENDUNG DES GELERTEN/INTERAKTIVE DEMONSTRATIONEN	30
5.1.1	DEMO – SCHNITTPUNKT RAY/EBENE	30

5.1.2	DEMO – SCHNITTGERADE EBENE/EBENE	33
5.1.3	DEMO – WURFPARABEL	34
5.1.4	DEMO – ZENOS PARADOXON/ELASTISCHE BEWEGUNG	36
5.1.5	DEMO – ZENO 3D	38
5.1.6	DEMO – BEWEGUNGSÜBERLAGERUNG	39
5.2	PROBLEME & HERAUSFORDERUNGEN	40
6	<u>SCHLUSSFOLGERUNG</u>	<u>43</u>

ABBILDUNGSVERZEICHNIS

ABBILDUNG 1: LINKSSYSTEM IN UNITY	3
ABBILDUNG 2: DER ABSTAND ZWISCHEN P1 UND P2	4
ABBILDUNG 3: PUNKT P1 (SCHWARZ) IN UNITY.....	6
ABBILDUNG 4: DER NORMALISIERTE ORTSVEKTOR DES PUNKTES P1.....	7
ABBILDUNG 5: DIE STRECKE KOORDINATENURSPRUNG – P1, GEZEICHNET MITTELS LINERENDERER.....	9
ABBILDUNG 6: EIN STRAHL GEZEICHNET MITTELS DEBUG.DRAWRAY.....	10
ABBILDUNG 7: DARSTELLUNG DES SCHNITTPUNKTES ALS GRÜNER WÜRFEL	13
ABBILDUNG 8: OUTPUT VON UPDATE	21
ABBILDUNG 9: OUTPUT VON FIXEDUPDATE	22
ABBILDUNG 10: VERGLEICH DER TIME- UND DER FIXEDTIME-VARIABLE	23
ABBILDUNG 11: AUSWAHLMENÜ DER DEMONSTRATIONEN	30
ABBILDUNG 12: DEMONSTRATION SCHNITTPUNKT RAY/EBENE I	31
ABBILDUNG 13: DEMONSTRATION SCHNITTPUNKT RAY/EBENE II	31
ABBILDUNG 14: DEMONSTRATION SCHNITTGERADE EBENE/EBENE	33
ABBILDUNG 15: DEMONSTRATION WURFPARABEL.....	34
ABBILDUNG 16: DEMONSTRATION ZENOS PARADOXON/ELASTISCHE BEWEGUNG	36
ABBILDUNG 17: DEMONSTRATION ZENO3D	39
ABBILDUNG 18: DEMONSTRATION BEWEGUNGSÜBERLAGERUNG	39
ABBILDUNG 19: PUNKT OHNE VISUELLE HILFSMITTEL.....	40
ABBILDUNG 20: PUNKT MIT ZUSÄTZLICHEN EBENEN	41
ABBILDUNG 21: PUNKT MIT HILFSLINIEN	41
ABBILDUNG 22: PUNKT MIT HILFSLINIEN IN FARBEN DER AXSEN	42
ABBILDUNG 23: PUNKT MIT ANDERER PERSPEKTIVE	42

CODEVERZEICHNIS

CODE 1: ERSTELLUNG DES VECTOR3 P1	3
CODE 2: BERECHNUNG DER MAGNITUDE	4
CODE 3: ERSTELLUNG DES VECTOR3 P2	4
CODE 4: BERECHNUNG DES RICHTUNGSVEKTORS P1P2	4
CODE 5: BERECHNUNG DER DISTANZ ZWISCHEN P1 UND P2	5
CODE 6: BERECHNUNG DER DISTANZ ZWISCHEN P1 UND P2 MITTELS DISTANCE	5
CODE 7: NORMALISIERUNG DES VEKTORS P1	6
CODE 8: GERADENGLEICHUNG MIT RICHTUNGSVEKTOR	7
CODE 9: GERADENGLEICHUNG MIT ZWEI PUNKTEN	8
CODE 10: ÜBERPRÜFUNG OB DER PUNKT A AUF DER GERADEN P1P2 LIEGT	8
CODE 11: BERECHNUNG DES ABSTANDES ZWISCHEN PUNKT B UND DER GERADEN P1P2	8
CODE 12: ZEICHNUNG EINER GERADEN MITTELS LINERENDERER	9
CODE 13: ERSTELLUNG EINES RAYS	9
CODE 14: ZEICHNUNG EINER STRECKE MITTELS DRAWRAY	10
CODE 15: ÜBERPRÜFUNG OB DER PUNKT A AUF DER STRECKE P1P2 LIEGT	11
CODE 16: EBENENGLEICHUNG IN C#	12
CODE 17: ÜBERPRÜFUNG, OB PUNKT X AUF DER EBENE LIEGT	12
CODE 18: BERECHNUNG DER DISTANZ ZWISCHEN PUNKT D UND DER EBENE	12
CODE 19: BERECHNUNG DES SCHNITTPUNKTES ZWISCHEN EINER LINIE UND EINER EBENE	13
CODE 20: BERECHNUNG DER SCHNITTGERADEN ZWISCHEN ZWEI EBENEN	14
CODE 21: DIE ERSTELLTE MYUPDATE-KLASSE LEITET SICH VON MONOBEHAVIOUR AB	21
CODE 22: DEMONSTRATION DER UPDATE-FUNKTION	21
CODE 23: DEMONSTRATION DER FIXEDUPDATE-FUNKTION	22
CODE 24: DEMONSTRATION VON TIME.DELTATIME	22
CODE 25: VERGLEICH DER TIME- UND DER FIXEDTIME-VARIABLE	23
CODE 26: KONSTANTE BEWEGUNG ALS FUNKTION DER ZEIT	24
CODE 27: KONSTANTE BEWEGUNG ALS INKREMENTELLE ÄNDERUNG	24
CODE 28: BESCHLEUNIGUNG ALS FUNKTION DER ZEIT	25
CODE 29: BESCHLEUNIGUNG ALS INKREMENTELLE ÄNDERUNG	25
CODE 30: VERZÖGERUNG ALS FUNKTION DER ZEIT	25
CODE 31: VERZÖGERUNG ALS INKREMENTELLE ÄNDERUNG	25
CODE 32: ROTATION ALS FUNKTION DER ZEIT	26
CODE 33: ROTATION ALS INKREMENTELLE ÄNDERUNG	26
CODE 34: WURFPARABEL ALS FUNKTION DER ZEIT	26
CODE 35: REIBUNG ALS FUNKTION DER ZEIT	27
CODE 36: REIBUNG ALS INKREMENTELLE ÄNDERUNG	27
CODE 37: GRAVITATION	27
CODE 38: AUFSPRINGENDER BALL SIMULATION	28
CODE 39: ZENOS PARADOXON	29
CODE 40: MOUSE-SKRIPT	32
CODE 41: RAYCAST-SKRIPT	32
CODE 42: MOVEMENT-SKRIPT	34
CODE 43: THROW-SKRIPT	36
CODE 44: BOUNCE SKRIPT	38

ZUSAMMENFASSUNG

„Dreidimensionale Repräsentation von Algorithmen in Unity“ – diese Arbeit beschäftigt sich mit Algorithmen, welche im dreidimensionalen Raum berechnet werden. Diese Algorithmen basieren auf Formeln der Mathematik und Physik und werden für diese Arbeit in C# Computercode umgeformt.

Die dreidimensionale Repräsentation erfolgt mittels der Unity Engine.

Um Algorithmen im dreidimensionalen Raum verständlich zu machen, werden in dieser Bachelorarbeit die dreidimensionalen Grundlagen in Unity und C# vermittelt. Es werden dreidimensionale Punkte, Geraden, Strahlen, Strecken und Ebenen definiert und erläutert.

Weiters werden die mathematischen und physikalischen Grundlagen von Algorithmen und deren Formeln erklärt. Dies inkludiert Bewegungsvorgänge, welche sowohl als Funktionen der Zeit als auch als inkrementelle Änderungen beschrieben werden.

Es werden elementare Eigenheiten von Unity ausgeführt, die für das Verständnis und die Anwendung der dreidimensionalen Algorithmen in Unity unverzichtbar sind. Es werden die theoretischen mathematischen und physikalischen Formeln in der Praxis angewendet. Dies geschieht, indem die vermittelten Formeln als C#-Code umgesetzt werden.

Zusätzlich zur geschriebenen Arbeit wurden interaktive Demonstrationen angefertigt, welche die dreidimensionalen Algorithmen visualisieren.

Diese Demonstrationen umfassen Berechnungen der Schnittpunkte zwischen einem Strahl und einer Ebene sowie der Schnittgerade zwischen zwei Ebenen. Es wird Zenos Paradoxon veranschaulicht, der freie Fall, elastische Bewegungen in Form eines aufspringenden Balles, sowie eine Wurfparabel. Weiters werden die Überlagerungen von eindimensionalen Bewegungen zu einer dreidimensionalen Bewegung demonstriert.

Diese Arbeit setzt explizit eine gewisse Grundkenntnis in den Bereichen Programmieren, C# im Speziellen, Unity sowie Mathematik voraus. Als Zielgruppe wurden Personen mit dem Kenntnisstand von Informationsmanagement-Studierenden im vierten Semester festgelegt.

ABSTRACT

“Dreidimensionale Repräsentation von Algorithmen in Unity” – this means “Three-dimensional Representation of Algorithms in Unity”. This bachelor thesis is concerned with algorithms that are computed in the three-dimensional space. Such algorithms are based on mathematical and physical formulas and are translated to C# code.

The three-dimensional representation is implemented using the Unity engine.

To elucidate algorithms in the three-dimensional space, this bachelor thesis imparts the three-dimensional basics of Unity and C#. Three-dimensional points, lines, rays, line segments and planes are defined and explained.

Furthermore, the mathematical and physical foundations of algorithms and their formulas are elucidated. This includes movement processes that are described both as function of time and incremental changes.

The fundamental behaviour of Unity is explained, which is necessary for the understanding and application of algorithms in Unity. The theoretical mathematical and physical formulas are put into praxis when written as C# code.

Additionally to the written thesis, interactive demonstrations were created visualising the explained three-dimensional algorithms.

Those demonstrations aim to visualise the intersection point of a ray and a plane and the intersection line of two planes. Zeno’s Paradox will be demonstrated as well as the free fall, a throw-trajectory, and elastic movement in form of a bouncing ball. The last demonstration visualises how several overlapping one-dimensional movements create a single three-dimensional movement.

This bachelor thesis is written in German. Additionally, basic knowledge of programming (especially C#), Unity, and mathematics are required. The target audience are students of Information Management in the 4th semester.

1 EINLEITUNG

In dieser Arbeit sollen Algorithmen untersucht werden, die sich im dreidimensionalen Raum bewegen. Algorithmen sind nichts anderes als eine festgelegte Reihenfolge von vordefinierten Einzelanweisungen. Algorithmen, welche Bewegungen im Raum abbilden, basieren auf Formeln der klassischen Mathematik und Physik. Diese müssen in einen für Computer verständlichen Programmcode umgeformt werden.

Dreidimensionale Repräsentationen dieser Algorithmen bedeuten, dass sowohl die verwendeten Algorithmen sich im dreidimensionalen Raum bewegen, also drei Achsen eines Koordinatensystems berücksichtigen, als auch dass die visuelle Darstellung in einem dreidimensionalen Raum stattfindet.

Diese Visualisierung der dreidimensionalen Berechnung findet unter Verwendung der Unity Engine statt. Diese für nicht-kommerzielle Zwecke freie Engine ist der Rahmen dieser Arbeit. Für diese Arbeit wird nicht auf Unitys bestehende Physikengine zurückgegriffen, stattdessen wird Unity nur für die dreidimensionale Wiedergabe der Berechnungen verwendet.

Sämtliche mathematischen und physikalischen Berechnungen finden stattdessen über in C# geschriebene Skripte statt.

Diese Arbeit setzt explizit eine gewisse Grundkenntnis in den Bereichen Programmieren, C# im Speziellen, Unity sowie Mathematik voraus. Als Zielgruppe wurden Personen mit Kenntnisstand von Informationsmanagement-Studierenden im vierten Semester festgelegt.

1.1 PROBLEMSTELLUNG

Algorithmen können insbesondere dann, wenn sie sich auf die dritte Dimension erstrecken, komplex ausfallen und somit schwer verständlich sein. Darüber hinaus kann es sich als schwierig erweisen, Codepassagen den entsprechenden Resultaten zuzuordnen. Vereinfacht formuliert: Welche Variable beeinflusst welche Änderung?

Es kann sich als problematisch erweisen bekannte mathematische und physikalische Formeln in Computercode umzusetzen. Dies ist insbesondere dann der Fall, wenn Gleichungen oder Gleichungssysteme gelöst werden müssen.

Ohne Erfahrung mit Unity kann es daher unklar sein, welche Datentypen oder Objekte für eine Problemstellung benötigt werden. Weiters ist Unitys Umgang mit Zeit und Berechnungen mit einer zeitlichen Komponente problematisch, wenn der Zusammenhang der unterschiedlichen Funktionen und Methoden nicht gänzlich verstanden wurde.

Schlussendlich kann auch die Visualisierung und Demonstration des zu Vermittelnden heikel sein, insbesondere dann, wenn ein dreidimensionaler Raum auf einem zweidimensionalen Screenshot abgebildet wird.

1.2 ZIELSETZUNG

Ziel der Arbeit ist eine Vermittlung grundlegender Algorithmen sowie die visualisierte, dreidimensionale Darstellung dieser Algorithmen unter Verwendung der Unity Engine. So soll den Algorithmen die wahrgenommene Komplexität genommen werden, indem die mathematischen und physikalischen Grundlagen erläutert und erklärt werden und auf die einzelnen Komponenten der Algorithmen eingegangen wird. Dies soll für ein tiefgehendes theoretisches Verständnis sorgen.

Zusätzlich zu den mathematischen und physikalischen Grundlagen der Algorithmen wird gezeigt, wie diese in C#-Code geschrieben aussehen und dieser Code sich verhält. Dieser Code wird kommentiert und ausgewertet, um ein optimales Verständnis zu vermitteln.

Es soll auf die Eigen- und Besonderheiten von Unity eingegangen werden, wie beispielsweise das Verhalten der Update-Funktion. Dieses Wissen soll bei späteren Implementierungsentscheidungen von Nutzen sein, denn in Unity lassen sich Aufgabenstellungen meist über verschiedene Wege bearbeiten, welche jeweils Eigenheiten aufweisen können.

Die Arbeit soll entsprechend die gebotenen Freiheiten und Möglichkeiten mit Unity demonstrieren und als Inspiration sowie Nachschlagewerk für darauf aufbauende Arbeiten dienen.

Weiters sind der Arbeit interaktive Demonstrationen beigelegt, die sich mit den erläuterten Themen beschäftigen. Diese sollen aufzeigen, wie das theoretisch vermittelte Wissen sich in der Praxis anwenden lässt.

2 3D-GRUNDLAGEN IN UNITY UND C#

In diesem Kapitel wird auf jene Objekte eingegangen, welche die essenziellen Bausteine für Berechnungen im dreidimensionalen Raum bilden.

2.1 3D-ACHSEN UND KOORDINATENSYSTEM

Unity verwendet ein linkshändiges Koordinatensystem, kurz Linkssystem genannt (vgl. [Seifert 2015], S. 107).

Koordinaten werden in der Form (X, Y, Z) angegeben.

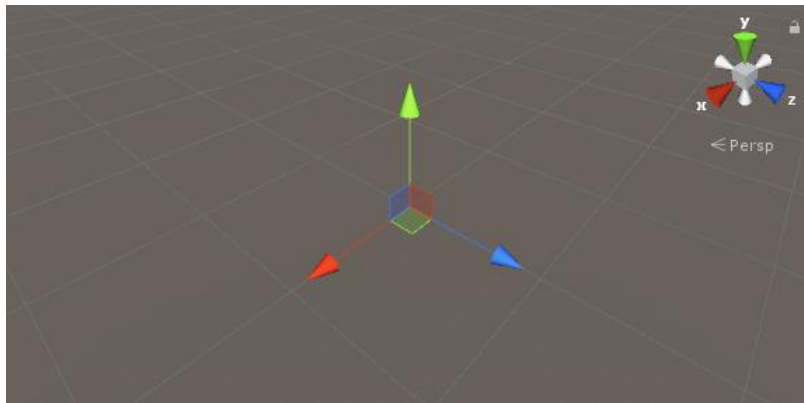


ABBILDUNG 1: LINKSSYSTEM IN UNITY

Es gibt ein globales Koordinatensystem, welches für alle Objekte gleich ist und über `Space.World` referenziert werden kann. Außerdem gibt es ein lokales Koordinatensystem, bei dem ein Objekt selbst den Koordinatenursprung darstellt, welches über `Space.Self` referenziert werden kann (vgl. [<https://docs.unity3d.com>][1]).

Das Koordinatensystem in Unity verwendet keine gesonderten Maßeinheiten. Somit ist der Punkt (1,0,0) eine Längeneinheit (Englisch: Length Unit) vom Koordinatenursprung entfernt. In der weiteren Arbeit werden, falls nicht anders beschrieben, Längeneinheiten als Maßeinheiten der angegebenen Werte angenommen.

2.2 3D-PUNKTE

Ein Punkt bestimmt eine exakte Stelle im Koordinatensystem, ohne selbst ein Volumen oder eine Dimension zu besitzen. Punkte werden deshalb auch als 0-Dimension bezeichnet und stellen das kleinste Element der Geometrie dar. In Unity werden Punkte über den Datentyp `Vector3` beschrieben.

01	<code>Vector3 p1 = new Vector3(1, 2, 3);</code>
----	---

CODE 1: ERSTELLUNG DES VECTOR3 P1

2.2.1 ORTSVEKTOREN

Beschreibt ein Vektor einen Ort, nennt man ihn Ortsvektor. Weiters kann ein `Vector3` eine Richtung in Form eines Richtungsvektors angeben, siehe Absatz „Geraden, Richtungsvektoren“.

2.2.2 BERECHNUNGEN MIT EINEM PUNKT

Will man von diesem Punkt die Entfernung zum Ursprung berechnen, kann dies durch die Methode „magnitude“ erfolgen, welche sich den Satz des Pythagoras zu Nutze macht.

```
01 var magnitude = Vector3.Magnitude(p1);
02 print("Distance to origin of ordinates: " + magnitude);
```

CODE 2: BERECHNUNG DER MAGNITUDE

Der Output in der Konsole lautet:

Distance to origin of ordinates: 3.741657

2.2.3 BERECHNUNGEN MIT ZWEI PUNKTEN

Um weitere Berechnungen durchzuführen, bedarf es eines zweiten Punktes.

```
01 Vector3 p2 = new Vector3(3, 4, 5);
```

CODE 3: ERSTELLUNG DES VECTOR3 P2

2.2.3.1 VEKTOR ZWISCHEN ZWEI PUNKTEN

Nun kann der Vektor zwischen den beiden Punkten ermittelt werden. Dafür muss einfach der Ausgangspunkt vom Zielpunkt subtrahiert werden:

```
01 Vector3 p1p2 = p2 - p1;
02 print("Vector from P1 to P2: " + p1p2);
```

CODE 4: BERECHNUNG DES RICHTUNGSVEKTORS P1P2

Vector from P1 to P2: (2.0, 2.0, 2.0)

2.2.3.2 DISTANZ ZWISCHEN ZWEI PUNKTEN

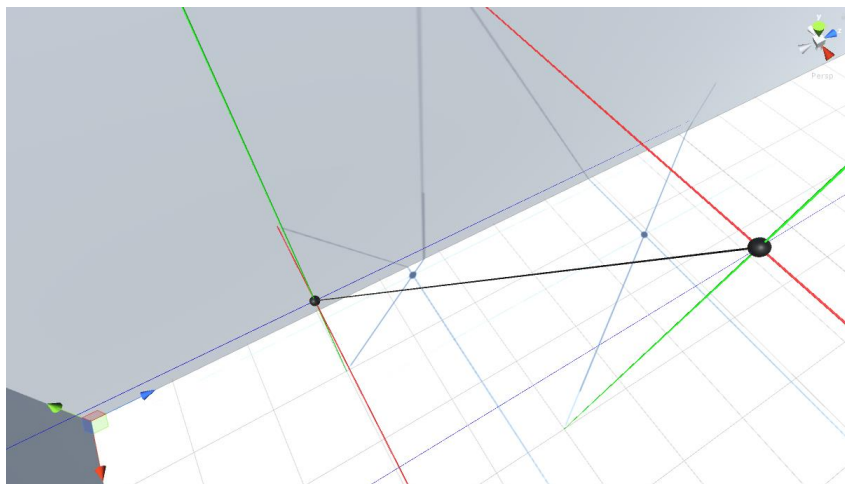


ABBILDUNG 2: DER ABSTAND ZWISCHEN P1 UND P2

Weiters kann nun die Distanz der beiden Punkte unter Verwendung der folgenden Formel ermittelt werden:

$$|\overrightarrow{PQ}| = \sqrt{(Q_x - P_x)^2 + (Q_y - P_y)^2 + (Q_z - P_z)^2}$$

([Papula 2009], S. 49)

Als Code geschrieben ergibt das:

01	<code>var distance1 = Math.Sqrt(Math.Pow(p1.x - p2.x, 2) + Math.Pow(p1.y - p2.y, 2) + Math.Pow(p1.z - p2.z, 2));</code>
----	---

CODE 5: BERECHNUNG DER DISTANZ ZWISCHEN P1 UND P2

Es kann auch die vorgefertigte Methode von Vector3 verwendet werden:

01	<code>var distance2 = Vector3.Distance(p1, p2);</code>
02	<code>print("Distance 1: " + distance1 + ", Distance 2: " + distance2);</code>

CODE 6: BERECHNUNG DER DISTANZ ZWISCHEN P1 UND P2 MITTELS DISTANCE

Dies ergibt folgenden Output:

Distance 1: 3.46410161513775, Distance 2: 3.464102

Die Abweichung der Nachkommastellen im Output entsteht dadurch, dass Distance 1 vom Datentyp „Double“ und Distance 2 vom Datentyp „Single“ ist.

Daraus ergibt sich die wichtige Schlussfolgerung, dass Unity viele Funktionen bereits von Haus aus mitbringt, diese aber bei Bedarf – etwa bei einer gewünschten höheren Genauigkeit – auch selbst implementiert werden können.

2.2.4 DARSTELLUNG VON PUNKTEN

Wie beschrieben haben Punkte keine Dimension und sind somit unendlich klein. Demnach sind Punkte de facto unsichtbar. Unity stellt auch keine Funktion bereit, um einen Punkt zu zeichnen.

In dieser Arbeit werden 3D-Objekte vom Typ „Sphere“ verwendet, um Punkte zu visualisieren. Diese „Punkt-Sphären“ werden, falls nicht anders beschrieben, mit einem Durchmesser von 0.1 (Maßstab: (0.1, 0.1, 0.1)) gezeichnet.

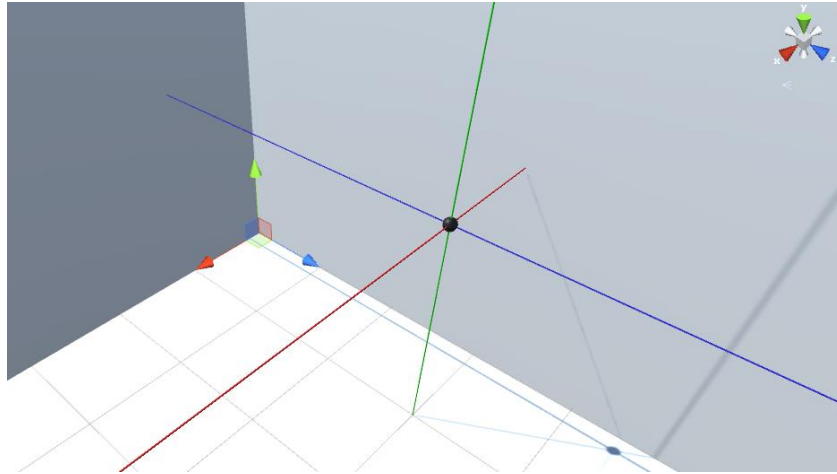


ABBILDUNG 3: PUNKT P1 (SCHWARZ) IN UNITY

Anmerkung zur Abbildung: Um die Lage des Punktes im Raum besser zu veranschaulichen, wurden außer dem Punkt p1 auch drei Ebenen hinzugefügt, welche die XY-, XZ-, und YZ-Ebenen darstellen. Weiters wurden drei Sphären mit dem Maßstab (100, 0.1, 0.1) gezeichnet, um die Lage des Punktes proportional zu genannten Ebenen zu demonstrieren.

2.3 GERADEN

Eine Gerade ist nicht gekrümmt und erstreckt sich unendlich in eine Dimension. Sie kann über zwei Punkte oder einen Punkt und einen Richtungsvektor definiert werden.

2.3.1 RICHTUNGSVEKTOREN

Vektoren können nicht nur Punkte, sondern auch Richtungen abbilden. Diese Richtungsvektoren haben auch einen Betrag, also ihren Abstand zum Nullpunkt (siehe „magnitude“) (vgl. [Seifert 2015], S. 109).

Ist nur die Richtung, aber nicht der Betrag von Interesse, kann ein Vektor normalisiert werden. Normalisierte Vektoren haben immer eine Länge von 1 und werden beispielsweise für „Rays“ benötigt (vgl. [Seifert 2015], S. 110).

```
01 Vector3 normP1 = Vector3.Normalize(p1);  
02 print("Normalized vector p1: " + normP1);
```

CODE 7: NORMALISIERUNG DES VEKTORS P1

Dies ergibt folgenden Output:

```
Normalized vector p1: (0.3, 0.5, 0.8)
```

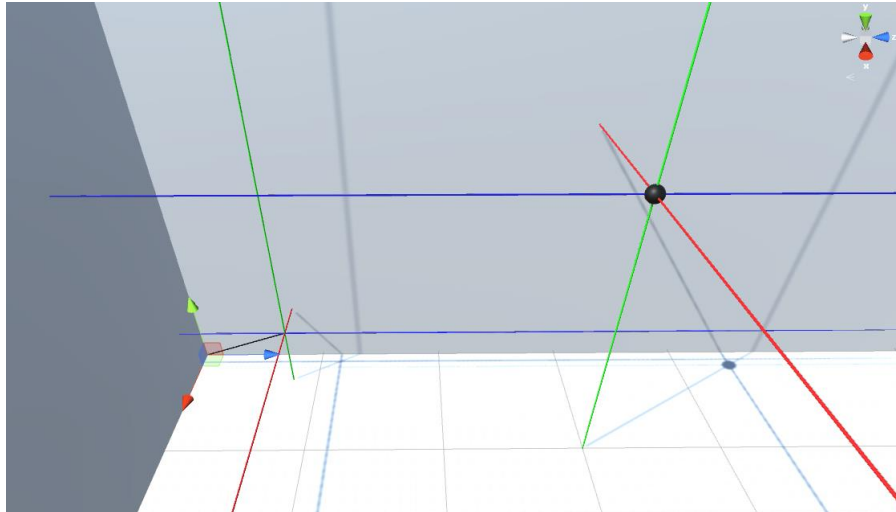


ABBILDUNG 4: DER NORMALISIERTE ORTSVEKTOR DES PUNKTES P1

2.3.2 GERADE MITTELS RICHTUNGSVEKTOR

Kennt man einen Punkt P der Geraden und deren Richtungsvektor \vec{a} , so lautet die Geradengleichung:

$$g: X = P + t \cdot \vec{a}$$

([Papula 2009], S. 75)

Dies stellt die sogenannte „Parameterform der Gleichung“ dar, da der Parameter t mit einem beliebigen Wert ersetzt werden kann, um einen Punkt X auf der Geraden zu bekommen.

In C# würde man die Formel folgendermaßen anwenden können:

01	<code>Xx = p1.x + t * (p1p2.x);</code>
02	<code>Xy = p1.y + t * (p1p2.y);</code>
03	<code>Xz = p1.z + t * (p1p2.z);</code>
04	<code>Vector3 X = new Vector3(Xx, Xy, Xz);</code>

CODE 8: GERADENGLEICHUNG MIT RICHTUNGSVEKTOR

Generiert man t nun mit einem for-loop oder der Random-Funktion, erhält man beliebig viele Punkte auf der Geraden.

2.3.3 GERADE DURCH ZWEI PUNKTE

Soll eine Gerade durch zwei Punkte bestimmt werden, so kann die Formel für die Berechnung des Vektors zwischen zwei Punkten in die Geradenformel eingesetzt werden:

$$g: X = P_1 + t \cdot (P_2 - P_1)$$

01	<code>Xx = p1.x + t * (p2.x - p1.x);</code>
02	<code>Xy = p1.y + t * (p2.y - p1.y);</code>
03	<code>Xz = p1.z + t * (p2.z - p1.z);</code>


```
04 Vector3 X = new Vector3(Xx, Xy, Xz);
```

CODE 9: GERADENGLEICHUNG MIT ZWEI PUNKTEN

2.3.4 LIEGT EIN PUNKT AUF EINER GERADEN?

Um zu überprüfen, ob ein Punkt X auf einer bekannten Geraden liegt, muss die Geradengleichung umgeformt werden.

$$t = \frac{X-P}{\vec{a}} \quad \text{beziehungsweise} \quad t = \frac{X-P_1}{P_2-P_1}$$

```
01 Vector3 A = new Vector3(2, 3, 4);
02 var tx = (A.x - p1.x) / p1p2.x;
03 var ty = (A.y - p1.y) / p1p2.y;
04 var tz = (A.z - p1.z) / p1p2.z;
05 print("tx: " + tx + ", ty: " + ty + ", tz: " + tz);
06 if(tx == ty && tx == tz) print("Point A is part of the
    line defined by p1 and p2");
07 else print("Point A is NOT part of the line defined by
    p1 and p2");
```

CODE 10: ÜBERPRÜFUNG OB DER PUNKT A AUF DER GERADEN P1P2 LIEGT

tx: 0.5, ty: 0.5, tz: 0.5

Point A is part of the line defined by p1 and p2

In Zeile 6 wird verglichen, ob die Parameter tx, ty und tz übereinstimmen. Ist dies der Fall, liegt der überprüfte Punkt auf der Geraden.

2.3.5 ABSTAND EINES PUNKTES ZU EINER GERADEN

Liegt ein Punkt B nicht auf einer Geraden $g: X = P + t \cdot \vec{a}$, kann man seinen Abstand zur Geraden mit Hilfe folgender Formel berechnen.

$$d = \frac{|(B - P) \times \vec{a}|}{|\vec{a}|}$$

(<https://de.serlo.org/>[1])

Diese Formel erfordert das Kreuzprodukt von Vektoren. Dieses muss jedoch nicht selbst geschrieben werden, da Vector3 diese Funktion bereits mitbringt.

Der Betrag eines Vektors ist dessen Abstand zum Koordinatenursprung. Dieser lässt sich, wie bereits beschrieben, mit der „magnitude“-Methode bestimmen.

Die Formel zur Berechnung des Abstandes einer Geraden und eines Punktes in C# wird demnach wie folgt geschrieben:

```
01 Vector3 B = new Vector3(3, 3, 4);
02 var distB = (Vector3.Cross((B - p1), p1p2)).magnitude /
    p1p2.magnitude;
03 print("Distance between line and point B: " + distB);
```

CODE 11: BERECHNUNG DES ABSTANDES ZWISCHEN PUNKT B UND DER GERADEN P1P2

Distance between line and point B: 0.8164966

2.3.6 DARSTELLUNG VON GERADEN

Geraden werden in Unity mit Hilfe der „LineRenderer“-Klasse dargestellt.

```

01 | LineRenderer lineRenderer =
    |     gameObject.AddComponent<LineRenderer>();
02 | lineRenderer.material = lineMaterial;
03 | lineRenderer.widthMultiplier = 0.01f;
04 | lineRenderer.SetPosition(0, new Vector3(0.3F, 0.5F,
    |     0.8F));
05 | lineRenderer.SetPosition(1, new Vector3(0, 0, 0));
  
```

CODE 12: ZEICHNUNG EINER GERADEN MITTELS LINERENDERER

Da LineRenderer mehr als zwei Punkte aufnehmen können, muss in Zeile 4 und 5 zusätzlich zu den Punkten deren Index angegeben werden (0 und 1).

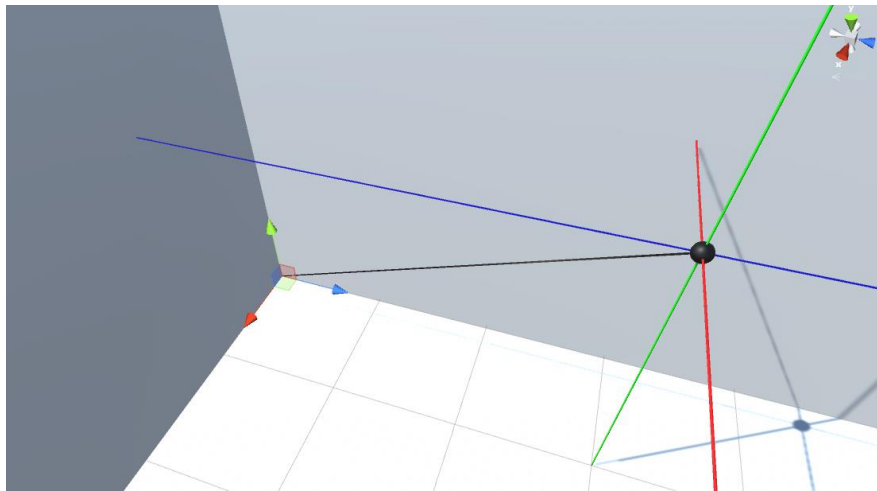


ABBILDUNG 5: DIE STRECKE KOORDINATENURSPRUNG – P1, GEZEICHNET MITTELS LINERENDERER

2.4 STRAHLEN, RAYS

Ein Strahl hat einen Ausgangs- aber keinen Endpunkt. Er ist nicht gekrümmt und erstreckt sich in eine Richtung unendlich.

In Unity werden Strahlen durch den Datentyp „Ray“ abgebildet. Ein Ray nimmt einen normalisierten Richtungsvektor und einen Punkt und erstellt daraus einen Strahl.

```

01 | Vector3 direction = new Vector3(0, 2, 4);
02 | Vector3 point = new Vector3(5, 5, 5);
03 | Ray ray = new Ray(direction.normalized, point);
  
```

CODE 13: ERSTELLUNG EINES RAYS

2.4.1 DARSTELLUNG VON RAYS

Rays können in Unity per Standardeinstellung nur im Szenen-Modus und nicht im Game-Modus gezeichnet werden. Will man Strahlen im Game-Modus visualisieren, muss man im Reiter über der 3D-Darstellung „Gizmos“ aktivieren.

Strahlen können über die „DrawRay“-Methode der „Debug“-Klasse gezeichnet werden.

Soll eine Szene kompiliert werden, müssen Strahlen über die „LineRenderer“-Klasse gezeichnet werden..

01	<code>Debug.DrawRay(point, direction, Color.magenta, float.MaxValue);</code>
----	--

CODE 14: ZEICHNUNG EINER STRECKE MITTELS DRAWRAY

Der letzte Parameter gibt dabei die Anzeigedauer des Rays in Sekunden an.

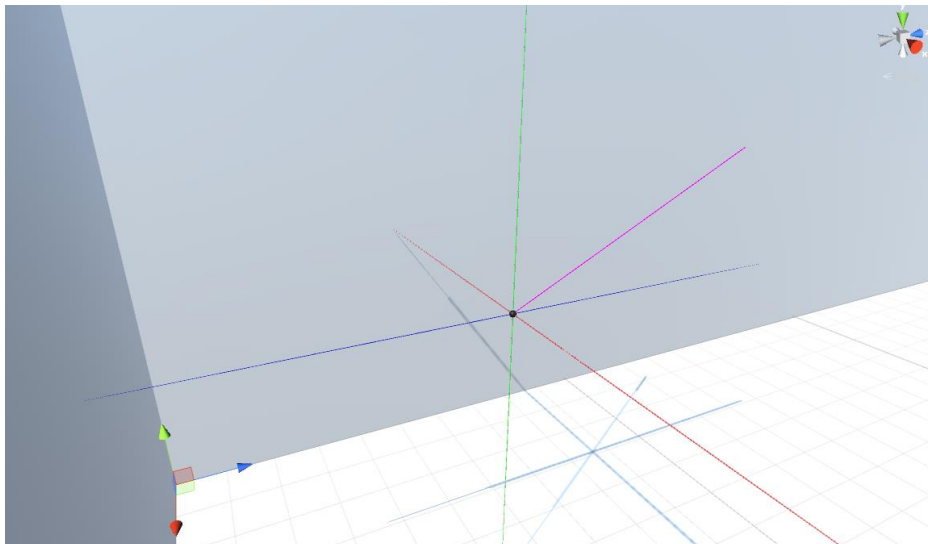


ABBILDUNG 6: EIN STRAHL GEZEICHNET MITTELS DEBUG.DRAWRAY

In dieser Abbildung sieht man die Eigenheit von „DrawRay“, dass eigentlich kein Strahl, sondern eine Strecke, gerendert wird. Diese Strecke hat als Endpunkt den Ausgangspunkt zuzüglich des Richtungsvektors, in diesem Fall also (5, 7, 9).

2.5 STRECKEN

Eine Strecke ist ein Abschnitt einer Geraden. Dieser Abschnitt wird durch zwei Punkte, welche auf der Geraden liegen, eingegrenzt.

Strecken haben in Unity keinen eigenen Datentypen.

Da Strecken über zwei Punkte definiert werden, können Strecken ebenfalls über die „LineRenderer“-Klasse gezeichnet werden.

2.5.1 LIEGT EIN PUNKT AUF EINER STRECKE?

Um zu überprüfen, ob ein Punkt auf einer Strecke liegt, kann analog zur Überprüfung, ob ein Punkt auf einer Geraden liegt, vorgegangen werden.

Stimmen die Parameter überein, liegt ein Punkt auf einer Geraden. Haben die Parameter zusätzlich einen Wert größer gleich 0 oder kleiner gleich 1, liegt der Punkt auf der Strecke.

```

01 Vector3 A = new Vector3(5, 6, 7);
02 var tx = (A.x - p1.x) / p1p2.x;
03 var ty = (A.y - p1.y) / p1p2.y;
04 var tz = (A.z - p1.z) / p1p2.z;
05 print("tx: " + tx + ", ty: " + ty + ", tz: " + tz);
06 if (tx == ty && tx == tz) {
07     print("Point A is part of the line defined by p1
08         and p2");
09     if(tx >= 0 && tx <= 1) print("Point A is part of
10         the line segment between p1 and p2");
11     else print("Point A is NOT part of the line
12         segment between p1 and p2");
13 }
14 else print("Point A is NOT part of the line defined by
15     p1 and p2");

```

CODE 15: ÜBERPRÜFUNG OB DER PUNKT A AUF DER STRECKE P1P2 LIEGT

tx: 2, ty: 2, tz: 2

Point A is part of the line defined by p1 and p2

Point A is NOT part of the line segment between p1 and p2

2.6 EBENEN

Eine Ebene wird durch drei Punkte, welche nicht auf einer Geraden liegen, bestimmt. Ebenen erstrecken sich unendlich zweidimensional, haben also keine Begrenzung.

Da die definierenden Punkte nicht auf einer Geraden liegen dürfen, kann man eine Ebene ε auch durch einen Punkt P und zwei von diesem Punkt ausgehenden Richtungsvektoren (\vec{a} und \vec{b}) darstellen.

$$\varepsilon: X = P + u \cdot \vec{a} + v \cdot \vec{b}$$

([Papula 2009], S. 60)

Dies stellt die sogenannte Parameterform der Ebenengleichung dar, da u und v Parameter darstellen.

Um folgende Rechnung möglichst anschaulich zu gestalten, wurde als Ebene jene Ebene gewählt, welche von der X- und Y-Achse aufgespannt wird. Dazu werden die Punkte A:(0, 0, 0), B:(1, 0, 0) und C:(0, 1, 0) verwendet.

```

01 var E = A + u * AB + v * AC;

```

CODE 16: EBENENGLEICHUNG IN C#

Wie auch bei Geraden können u und v nun per Zufallsgenerator oder for-loop bestimmt werden, um beliebig viele Punkte auf der Ebene zu erhalten.

2.6.1 LIEGT EIN PUNKT AUF EINER EBENE?

Eine analoge Vorgehensweise zur Überprüfung, ob ein Punkt auf einer Geraden liegt, ist nicht möglich, da wir es nun mit zwei Unbekannten – u und v – zu tun haben.

Stattdessen muss die Ebenengleichung auf die sogenannte Normalform gebracht werden.

$$\varepsilon: \vec{n} \cdot (X - P) = 0$$

(<https://de.serlo.org/>[2])

\vec{n} stellt dabei den Normalvektor der Ebene dar, welcher das Kreuzprodukt der Richtungsvektoren \vec{a} und \vec{b} darstellt.

Setzt man nun in diese Gleichung den zu überprüfenden Punkt X ein, so ergibt die Gleichung eine wahre Aussage ($0 = 0$), falls der Punkt auf der Ebene liegt.

```
01 Vector3 X = new Vector3(2, 3, 0);
02 Vector3 n = Vector3.Cross(AB, AC);
03 Vector3 XA = (X - A);
04 if ((XA.x * n.x + XA.y * n.y + XA.z * n.z) == 0)
    print("Point X is part of the plane");
05 else print("Point X is NOT part of the plane");
```

CODE 17: ÜBERPRÜFUNG, OB PUNKT X AUF DER EBENE LIEGT

Point X is part of the plane

2.6.2 ABSTAND EINES PUNKTES ZU EINER EBENE

Die Formel für den Abstand eines Punktes zu einer Ebene lautet:

$$d = \frac{|(X - P) \cdot \vec{n}|}{|\vec{n}|}$$

([Papula 2009], S. 62)

Hierbei ist X der gegebene Punkt und P jener Punkt auf der Ebene, von welchem jene Vektoren ausgehen, die den Normalvektor \vec{n} bilden.

```
01 Vector3 D = new Vector3(4, 5, 10);
02 Vector3 DA = (D - A);
03 var numerator = Math.Abs(DA.x * n.x + DA.y * n.y + DA.z
    * n.z);
04 var distance = numerator / n.magnitude;
05 print("Distance: " + distance);
```

CODE 18: BERECHNUNG DER DISTANZ ZWISCHEN PUNKT D UND DER EBENE

Distance: 10

2.6.3 SCHNITTPUNKT LINIE-EBENE

Um den Schnittpunkt einer Linie und einer Ebene zu bestimmen, kann die Funktion „Raycast“ des „Plane“-Datentyps verwendet werden.

```

01 Plane plane = new Plane();
02 plane.Set3Points(new Vector3(0, 0.1F, 0),
    new Vector3(1, 0.1F, 0), new Vector3(0, 0.1F, 1));
03 Vector3 direction = new Vector3(0, 2, 0);
04 Vector3 point = new Vector3(0, -1, 0);
05 Ray ray = new Ray(direction.normalized, point);
06 float rayDistance;
07 plane.Raycast(ray, out rayDistance);
08 print("Intersection: " + ray.GetPoint(rayDistance));

```

CODE 19: BERECHNUNG DES SCHNITTPUNKTES ZWISCHEN EINER LINIE UND EINER EBENE

Die „Raycast“-Funktion gibt einen float-Wert zurück, welcher angibt, nach welcher Distanz der Strahl auf die Ebene trifft, zu beobachten in Zeile 7 über das Keyword „out“.

Intersection: (0.0, 0.1, 0.0)

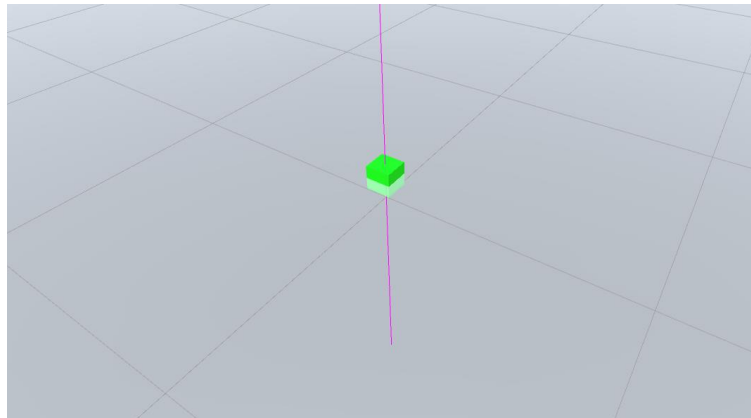


ABBILDUNG 7: DARSTELLUNG DES SCHNITTPUNKTES ALS GRÜNER WÜRFEL

2.6.4 SCHNITT EBENE-EBENE

Definiert man zwei Ebenen, welche nicht ident sind, können diese parallel im Raum liegen und sich nicht schneiden oder nicht parallel sein und sich schneiden. Schneiden sich die beiden Ebenen, so erhält man eine Gerade, welche auf beiden Ebenen liegt.

Um zu überprüfen, ob sich zwei nicht idente Ebenen schneiden, müssen beide Ebenen in der Normalform vorliegen.

```

01 lineVec = Vector3.Cross(plane1Normal, plane2Normal);
02 Vector3 ldir = Vector3.Cross(plane2Normal, lineVec);
03 float denominator = Vector3.Dot(plane1Normal, ldir);

```

```
04 if (Mathf.Abs(denominator) > 0) {  
05     Vector3 plane1ToPlane2 = plane1Position  
        - plane2Position;  
06     float t = Vector3.Dot(plane1Normal,  
        plane1ToPlane2) / denominator;  
07     linePoint = plane2Position + t * ldir;  
08     print("Planes are not parallel");  
09 }  
10 print("Planes are parallel, there is no intersection");
```

CODE 20: BERECHNUNG DER SCHNITTGERADEN ZWISCHEN ZWEI EBENEN

(vgl. [<http://wiki.unity3d.com>][1])

Die Schnittgerade wird durch „linePoint“ und „lineVec“ definiert.

2.6.5 DARSTELLUNG VON EBENEN

Ebenen haben in Unity ein eigenes 3D-Objekt, „Plane“. Planes können über die Funktion „Set3Points“ über drei Punkte, wie im Beispiel „Schnittpunkt Linie-Ebene“, oder über die Funktion „SetNormalAndPosition“ mittels eines Punkts und den Normalvektor der Ebene definiert werden.

3 MATHEMATISCHE & PHYSIKALISCHE GRUNDLAGEN

In diesem Kapitel soll auf jene Formeln eingegangen werden, welche die Grundlage für Bewegungsalgorithmen bilden.

3.1 FORMELN

Bewegung kann sowohl als Funktion der Zeit („Wo befindet sich ein Körper zum Zeitpunkt X?“) als auch als inkrementelle Änderung („Wo befindet sich der Körper als nächstes?“) gesehen werden.

Ein wichtiger Unterschied ist, dass für Berechnungen als Funktionen der Zeit die aktuelle Position des Körpers nicht berücksichtigt werden muss. Dies ist für Simulationen interessant, bei denen Userinput das Ergebnis beeinflussen kann und somit keine Vorhersage für den Zeitpunkt X getroffen werden kann.

3.1.1 KONSTANTE BEWEGUNG (GLEICHFÖRMIGE BEWEGUNG, KONSTANTE TRANSLATION)

Wie der Name bereits beschreibt, ändert sich bei einer konstanten Bewegung der aktuelle Bewegungszustand nicht. Ruht der betrachtete Körper, wird er auch im nächsten Zeitabschnitt ruhen; führt der Körper eine Bewegung aus, wird diese Bewegung fortgeführt. Im Gegenschluss bedeutet dies, dass eine konstante Bewegung stattfindet, wenn keine Kraft auf einen Körper ausgeübt wird.

Im physischen Raum ist eine konstante Bewegung eine Idealisierung, da auf einen Körper immer eine Kraft wirkt, wie Gravitation oder Reibungswiderstand. Für eine annähernd konstante Bewegung, beispielsweise ein Auto mit Tempomat, welches konstant 130 km/h fährt, muss ein Auto genau so viel Kraft aufbringen wie Luftwiderstand, Rollwiderstand etc. ausmachen.

Daraus lässt sich schließen, dass konstante Bewegung nicht nur dann stattfindet, wenn keine Kräfte wirken, sondern auch wenn Kräfte wirken, welche sich gegenseitig aufheben.

Im digitalen Raum ist konstante Bewegung viel geläufiger, da Faktoren wie Gravitation oder Reibung nicht zwangsläufig simuliert werden müssen.

3.1.1.1 KONSTANTE BEWEGUNG ALS FUNKTION DER ZEIT

Die Formel für konstante Bewegung setzt sich folgendermaßen zusammen:

$$\vec{s}(t) = \vec{s}_0 + \vec{v}_0 * t$$

([Kurzweil, Frenzel, Gebhard 2008], S. 11)

Daher benötigt ein Algorithmus folgende Informationen:

- \vec{s}_0 : Die Start- oder Ausgangskordinaten des Körpers, genannt Ortsvektor
- \vec{v}_0 : Einen konstanten Richtungsvektor
- t : Die vergangene Zeit

Somit erhält man die Koordinaten des Körpers nach der Zeit t , genannt $\vec{s}(t)$.

3.1.1.2 KONSTANTE BEWEGUNG ALS INKREMENTELLE ÄNDERUNG

Bei der konstanten Bewegung als inkrementelle Änderung fällt der Zeitfaktor aus der Gleichung, es sind nur noch die aktuelle Position und der Richtungsvektor relevant.

$$\vec{s}_{x+1} = \vec{s}_x + \vec{v}$$

- \vec{s}_x : Die aktuelle Position des Körpers
- \vec{v} : Der konstante Richtungsvektor

3.1.2 BESCHLEUNIGUNG & VERZÖGERUNG

Beschleunigung bedeutet immer, dass sich der Bewegungszustand eines Körpers ändert. Dies umfasst nicht nur eine positive Beschleunigung, welche die Geschwindigkeit eines Körpers erhöht, sondern auch eine negative Beschleunigung (auch Verzögerung genannt), welche die Geschwindigkeit eines Körpers verringert.

Darüber hinaus ist jegliche Richtungsänderung eine Beschleunigung, auch dann, wenn sich die absolute Geschwindigkeit nicht ändert. Auf Richtungsänderungen wird im nächsten Absatz eingegangen, dieser Absatz beschränkt sich auf Beschleunigung ohne Richtungsänderung, also die Bewegung entlang eines Vektors.

3.1.2.1 BESCHLEUNIGUNG ALS FUNKTION DER ZEIT

Die Formel für Beschleunigung in einem Zeitintervall setzt sich folgendermaßen zusammen:

$$a = \frac{\Delta v}{\Delta t}$$

([Kurzweil, Frenzel, Gebhard 2008], S. 12)

Daher benötigt ein Algorithmus folgende Informationen:

- Δv : Die Veränderung der Geschwindigkeit
- Δt : Die Veränderung der Zeit

3.1.2.2 BESCHLEUNIGUNG ALS INKREMENTELLE ÄNDERUNG

Bei der Beschleunigung als inkrementelle Änderung besteht die Beschleunigung aus der bestehenden Beschleunigung zuzüglich einer Änderung.

$$a_{x+1} = a_x + \Delta v$$

- a_x : Die aktuelle Beschleunigung
- Δv : Die Veränderung der Beschleunigung

3.2 GRAVITATIONSKRAFT

Gravitationskraft beschreibt die Kraft, welche zwei Körper bei einem gewissen Abstand aufeinander ausüben.

Die Formel der Gravitationskraft setzt sich folgendermaßen zusammen:

$$F = \frac{G \cdot m \cdot M}{r^2}$$

([Kurzweil, Frenzel, Gebhard 2008], S. 17)

Diese Formel wird auch Newtonsches Gravitationsgesetz, nach seinem Entdecker Isaac Newton, genannt.

Daher benötigt ein Algorithmus folgende Informationen:

- G : Die sogenannte Gravitationskonstante $G = \frac{6,672 \cdot 10^{-11} \text{m}^3}{\text{kg} \cdot \text{s}^2}$
- m : Die Masse des ersten Körpers in Kilogramm
- M : Die Masse des zweiten Körpers in Kilogramm
- r : Der Abstand zwischen den Massepunkten in Metern

3.3 FREIER FALL

Die folgenden Formeln beziehen sich auf den freien Fall im Gravitationsfeld der Erde.

3.3.1 FREIER FALL OHNE LUFTWIEDERSTAND

Der freie Fall ohne Berücksichtigung des Luftwiderstandes ist ein Sonderfall der gleichmäßig beschleunigten Bewegung.

3.3.1.1 STRECKENFORMEL ALS FUNKTION DER ZEIT

$$h = \frac{g \cdot t^2}{2}$$

([Kurzweil, Frenzel, Gebhard 2008], S. 11)

Daher benötigt ein Algorithmus folgende Informationen:

- g : Die Erdbeschleunigung. Dies ist eine Konstante und beträgt $g = \frac{9,81 \text{m}}{\text{s}^2}$
- t : Die Fallzeit in Sekunden

3.3.1.2 GESCHWINDIGKEITSFORMEL ALS FUNKTION DER ZEIT

$$v = g \cdot t$$

([Kurzweil, Frenzel, Gebhard 2008], S. 11)

Dabei ist g wiederum die Erdbeschleunigung und t die Fallzeit in Sekunden.

Eine wichtige Beobachtung ist, dass die Fallgeschwindigkeit im luftleeren Raum nicht von der Masse eines Objektes beeinflusst wird.

3.3.2 FREIER FALL MIT LUFTWIDERSTAND

Der freie Fall mit Luftwiderstand ist weitaus weniger trivial als die Berechnungen für ein Vakuum.

3.3.2.1 WIDERSTANDSKRAFT

Zur Berechnung des freien Falls mit Luftwiderstand benötigt man eine Möglichkeit, um jenen Widerstand auszudrücken. Diese Widerstandskraft F_w hat folgende Formel:

$$F_w = c_w \frac{1}{2} v^2 \rho A$$

(<http://www.virtual-maxim.de>)[1])

Daher benötigt ein Algorithmus folgende Informationen:

- c_w : Der c_w -Wert kann exakt nur experimentell bestimmt werden und ist abhängig von der Form des Objektes. Ein fallender Mensch hat ungefähr einen c_w -Wert von 0,3 (vgl. <https://rechneronline.de>)[1]).
- v : Die Geschwindigkeit des fallenden Objektes in m/s
- ρ : Die Luftdichte in kg/m³. Für Luft beträgt diese ungefähr 1,2 kg/m³ (vgl. <http://www.chemie.de>)[1]).
- A : Die Projektionsfläche des fallenden Objektes

3.3.2.2 GEWICHTSKRAFT

Außer der Widerstandskraft wirkt auf ein fallendes Objekt die sogenannte Gewichtskraft G .

$$G = mg$$

(<http://www.spektrum.de>)[1])

Hierbei ist m die Masse des Körpers in kg und g die Erdbeschleunigung (9,81) in m/s².

3.3.2.3 FALLBESCHLEUNIGUNG

Kennt man Gewichtskraft und Widerstandskraft, kann man die Fallbeschleunigung errechnen.

$$ma = G - F_w$$

(<http://www.virtual-maxim.de>)[1])

Die Endgeschwindigkeit, also die maximale Fallgeschwindigkeit, kann berechnet werden, wenn man für ma 0 einsetzt.

3.3.2.4 GESCHWINDIGKEITSFUNKTION ALS FUNKTION DER ZEIT

Um die Geschwindigkeit zu einem Zeitpunkt zu berechnen, kann man folgende Formel verwenden:

$$v(t) = v_E \tanh\left(\frac{g}{v_E} t\right)$$

(<http://www.virtual-maxim.de>)[2])

Dabei stellt v_E die Endgeschwindigkeit des Objektes dar.

3.3.2.5 ORTSFUNKTION ALS FUNKTION DER ZEIT

$$s(t) = \frac{v_E(t)^2}{g} \ln\left(\cosh\left(\frac{g}{v_E(t)} t\right)\right)$$

(<http://www.virtual-maxim.de>)[3])

3.4 WURFPARABEL

Die Wurfparabel, auch schiefer Wurf genannt, ist die Überlagerung einer schräg nach oben gerichteten, gleichförmigen Bewegung mit bestimmter Anfangsgeschwindigkeit und dem freien Fall (vgl. <http://www.maschinenbau-wissen.de>)[1]).

Der schiefe Wurf wird als zweidimensionale Bewegung mit x- und y-Koordinaten betrachtet.

Bei den nachfolgenden Formeln wird der Luftwiderstand nicht berücksichtigt.

3.4.1 WURFWEITE

$$x = v_x \cdot t$$

(<http://www.maschinenbau-wissen.de>)[1])

Hierbei stellt v_x die konstante Geschwindigkeit in x-Richtung in m/s und t die Zeit in Sekunden dar.

3.4.2 WURFHÖHE

$$y = v_{0y} \cdot t - \frac{g}{2} \cdot t^2$$

(<http://www.maschinenbau-wissen.de>)[1])

Daher benötigt ein Algorithmus folgende Informationen:

- v_{0y} : Die Abwurfgeschwindigkeit entlang der y-Achse in Metern pro Sekunde
- g : Die Erdbeschleunigung. Dies ist eine Konstante und beträgt $g = \frac{9,81m}{s^2}$
- t : Die Zeit seit Abwurf in Sekunden

3.4.3 BESCHLEUNIGUNG

Entlang der X-Achse wirkt keine Kraft auf das geworfene Objekt, entlang der y-Achse wirkt die Erdbeschleunigung.

$$a_x = 0 \quad \text{und} \quad a_y = -g$$

(<http://www.maschinenbau-wissen.de>)[1]

3.4.4 GESCHWINDIGKEIT

Die Geschwindigkeit in x-Richtung v_x und die Anfangsgeschwindigkeit in y-Richtung v_{0y} können mittels folgender Formeln bestimmt werden:

$$v_x = v_0 \cdot \cos(\alpha) \quad \text{beziehungsweise} \quad v_{0y} = v_0 \cdot \sin(\alpha) - g \cdot t$$

(<http://www.maschinenbau-wissen.de>)[1]

Dabei ist v_0 die Abwurfgeschwindigkeit und α der Abwurfwinkel in Winkelgrad relativ zur x-Achse.

3.4.5 ZURÜCKGELEGTE STRECKE

$$x(t) = v_0 \cdot \cos(\alpha) \cdot t \quad \text{beziehungsweise} \quad y(t) = \sin(\alpha) \cdot t - \frac{g}{2} \cdot t^2$$

(<http://www.maschinenbau-wissen.de>)[1]

4 3D-ALGORITHMEN, BEWEGUNG UND SIMULATION

In diesem Kapitel wird auf Funktionen und Methoden von Unity eingegangen und Algorithmen mit deren Hilfe abgebildet.

4.1 MONOBEHAVIOUR

MonoBehaviour ist die Basis-Klasse von Unity, von der alle Unity-Skripte standardmäßig erben (vgl. [<https://docs.unity3d.com>][2]).

```

01 public class MyUpdate : MonoBehaviour {
02     void Start () {...}
03     void Update () {...}
04     void FixedUpdate () {...}
05 }
```

CODE 21: DIE ERSTELLTE MYUPDATE-KLASSE LEITET SICH VON MONOBEHAVIOUR AB

In C# müssen Klassen von MonoBehaviour erben, in UnityScript nicht.

4.1.1 DIE UPDATE-FUNKTION

Die Update-Funktion der MonoBehaviour-Klasse wird für jeden gerenderten Frame aufgerufen. Dadurch können Probleme entstehen, da die Framerate von vielen Faktoren abhängt, wie zum Beispiel dem gerenderten Bildausschnitt oder der verwendeten Hardware.

```

01 void Update () {
02     timeStampNew = DateTime.Now;
03     print(timeStamp - timeStampNew);
04     timeStamp = timeStampNew;
05 }
```

CODE 22: DEMONSTRATION DER UPDATE-FUNKTION

Dieses simple Skript ergibt folgenden Output in der Konsole:



```

! -00:00:00.0069998
  UnityEngine.MonoBehaviour:print(Object)
! -00:00:00.0065822
  UnityEngine.MonoBehaviour:print(Object)
! -00:00:00.0069861
  UnityEngine.MonoBehaviour:print(Object)
! -00:00:00.0070141
  UnityEngine.MonoBehaviour:print(Object)
! -00:00:00.0065279
  UnityEngine.MonoBehaviour:print(Object)
! -00:00:00.0072522
  UnityEngine.MonoBehaviour:print(Object)
! -00:00:00.0064995
  UnityEngine.MonoBehaviour:print(Object)
```

ABBILDUNG 8: OUTPUT VON UPDATE

Obwohl nur eine leere Szene dargestellt wurde und keine Interaktion stattfand, unterscheiden sich die Zeiten der gerenderten Frames.

4.1.2 DIE FIXEDUPDATE-FUNKTION

Um Probleme, welche durch den unregelmäßigen Aufruf der Update-Funktion verursacht werden, zu vermeiden, gibt es in Unity die FixedUpdate-Funktion.

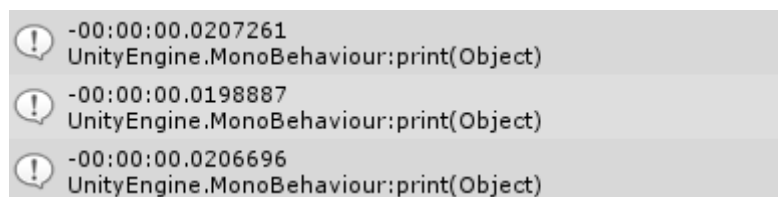
„This function is called every fixed framerate frame, if the MonoBehaviour is enabled“ (vgl. [<https://docs.unity3d.com>][3]).

Übersetzt würde dies ungefähr lauten: „Diese Funktion wird jeden fixierten Framerate Frame aufgerufen...“ was zunächst etwas kryptisch klingt. Sucht man weiter in der Unity-Dokumentation, wird man bei der Dokumentation der „Time Manager“-Klasse fündig. So bietet Unity unter Edit – Project Settings – Time ein Feld namens „Fixed Timestep“. Dieses hat den Standardwert 0.02, was bedeutet, dass die FixedUpdate-Funktion alle 0,02 Sekunden aufgerufen wird (vgl. [<https://docs.unity3d.com>][4]).

```
01 void FixedUpdate () {
02     timeStampNew = DateTime.Now;
03     print(timeStamp - timeStampNew);
04     timeStamp = timeStampNew;
05 }
```

CODE 23: DEMONSTRATION DER FIXEDUPDATE-FUNKTION

Der Output der Konsole sieht nun wie folgt aus:



```
! -00:00:00.0207261
UnityEngine.MonoBehaviour:print(Object)
! -00:00:00.0198887
UnityEngine.MonoBehaviour:print(Object)
! -00:00:00.0206696
UnityEngine.MonoBehaviour:print(Object)
```

ABBILDUNG 9: OUTPUT VON FIXEDUPDATE

Die FixedUpdate-Funktion wird demnach tatsächlich alle 0,02 Sekunden aufgerufen.

4.2 TIME UND METHODEN VON TIME

Mit der Time-Klasse werden in Unity zeitbezogene Methoden und Variablen zur Verfügung gestellt.

4.2.1 TIME.DELTATIME

Die deltaTime-Variable gibt in Sekunden an, wie lange die Berechnung des letzten Frames dauerte. [<https://docs.unity3d.com/ScriptReference/Time-deltaTime.html>]

```
01 void Update () {
02     float x = velocity * Time.deltaTime;
03     sphere.transform.Translate(x, 0.0F, 0.0F);
04 }
```

CODE 24: DEMONSTRATION VON TIME.DELTATIME

Da `deltaTime` verwendet wird, wird die Sphäre, obwohl nicht `FixedUpdate` benutzt wird, in einer Sekunde immer um den Wert der Variable „`velocity`“ bewegt.

4.2.2 TIME.FIXEDELTATIME

Anders als der Name es vermuten lässt, hat die `fixedDeltaTime`-Variable keinen direkten Bezug zur `deltaTime`-Variable. Stattdessen kann über `Time.fixedDeltaTime` das Zeitintervall, mit dem die `FixedUpdate`-Funktion aufgerufen wird, ausgelesen und geändert werden.

4.2.3 TIME.TIME

Die `time`-Variable gibt die Zeit in Sekunden seit dem Start der Unity-Laufzeitumgebung an.

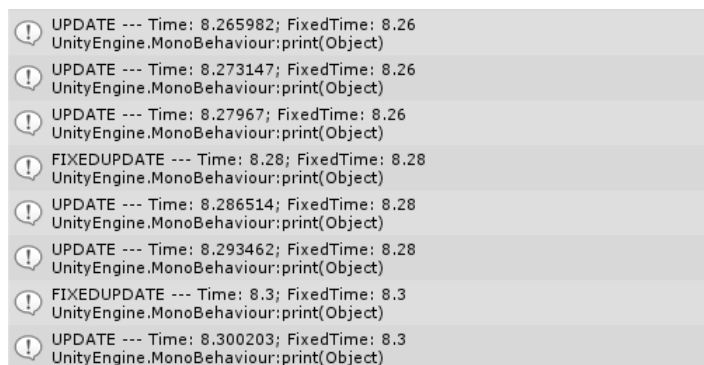
4.2.4 TIME.FIXEDTIME

Ähnlich der `time`-Variable gibt die `fixedTime`-Variable die vergangene Zeit seit dem Start der Unity-Laufzeitumgebung an. `FixedTime` wird jedoch nur beim Aufruf der `FixedUpdate`-Funktion aktualisiert.

```

01 void Update () {
02     print("UPDATE --- Time: " + Time.time + ";
           FixedTime: " + Time.fixedTime);
03 }
04
05 void FixedUpdate () {
06     print("FIXEDUPDATE --- Time: " + Time.time + ";
           FixedTime: " + Time.fixedTime);
07 }
    
```

CODE 25: VERGLEICH DER TIME- UND DER FIXEDTIME-VARIABLE



```

! UPDATE --- Time: 8.265982; FixedTime: 8.26
UnityEngine.MonoBehaviour:print(Object)
! UPDATE --- Time: 8.273147; FixedTime: 8.26
UnityEngine.MonoBehaviour:print(Object)
! UPDATE --- Time: 8.27967; FixedTime: 8.26
UnityEngine.MonoBehaviour:print(Object)
! FIXEDUPDATE --- Time: 8.28; FixedTime: 8.28
UnityEngine.MonoBehaviour:print(Object)
! UPDATE --- Time: 8.286514; FixedTime: 8.28
UnityEngine.MonoBehaviour:print(Object)
! UPDATE --- Time: 8.293462; FixedTime: 8.28
UnityEngine.MonoBehaviour:print(Object)
! FIXEDUPDATE --- Time: 8.3; FixedTime: 8.3
UnityEngine.MonoBehaviour:print(Object)
! UPDATE --- Time: 8.300203; FixedTime: 8.3
UnityEngine.MonoBehaviour:print(Object)
    
```

ABBILDUNG 10: VERGLEICH DER TIME- UND DER FIXEDTIME-VARIABLE

4.3 TRANSFORM-VARIABLEN UND FUNKTIONEN

Über die `Transform`-Klasse können Position, Rotation und Skalierung eines Objektes angepasst werden (vgl. [<https://docs.unity3d.com>][5]).

`transform.position:`

Über die position-Variable kann die Position eines Objektes ausgelesen und geändert werden.

Ändert man über die position-Variable die Position eines Objektes, wird keine Kraft auf dieses Objekt ausgeübt, es wird quasi teleportiert.

`transform.rotation:`

Über die position-Variable kann die Rotation eines Objektes ausgelesen und geändert werden.

`transform.Translate`

Im Gegensatz zu `transform.position` und `transform.rotation` ist `transform.Translate` eine Funktion und keine Variable.

Über die Translate-Funktion kann ein Objekt um einen Richtungsvektor verschoben werden. Als Bezugssystem kann das Koordinatensystem der Welt oder das lokale Koordinatensystem des Objektes gewählt werden.

`transform.Rotate`

Mittels der Rotate-Funktion kann man ein Objekt um einen Winkel im Gradmaß rotieren.

Eine Besonderheit ist, dass Winkel dabei als Vektoren abgebildet werden, deren jeweilige Koordinaten die Rotation in die jeweilige Achse darstellen. Der Vektor (180, 0, 0) würde also eine Rotation um 180° um die x-Achse darstellen.

Rotationen können relativ zum Objekt oder relativ zum globalen Koordinatensystem erfolgen.

4.4 ALGORITHMEN

In diesem Abschnitt werden Bewegungen als C# Code nachgebildet.

4.4.1 KONSTANTE BEWEGUNG

4.4.1.1 KONSTANTE BEWEGUNG ALS FUNKTION DER ZEIT

```
01 void Update () {  
02     float x = velocity * Time.deltaTime;  
03     sphere.transform.Translate(x, 0.0F, 0.0F);  
04 }
```

CODE 26: KONSTANTE BEWEGUNG ALS FUNKTION DER ZEIT

4.4.1.2 KONSTANTE BEWEGUNG ALS INKREMENTELLE ÄNDERUNG

```
01 void Update () {  
02     sphere.transform.Translate(velocity, 0.0F, 0.0F);  
03 }
```

CODE 27: KONSTANTE BEWEGUNG ALS INKREMENTELLE ÄNDERUNG

4.4.2 BESCHLEUNIGUNG & VERZÖGERUNG

4.4.2.1 BESCHLEUNIGUNG ALS FUNKTION DER ZEIT

```

01 void Update () {
02     velocity = velocity * 1.1F;
03     float x = velocity * Time.deltaTime;
04     sphere.transform.Translate(velocity, 0.0F, 0.0F);
05 }

```

CODE 28: BESCHLEUNIGUNG ALS FUNKTION DER ZEIT

Da „velocity“ bei jedem Aufruf von „Update“ erhöht wird, handelt es sich um Beschleunigung. Je größer „velocity“ wird, desto größer wird die Erhöhung. Es liegt eine exponentielle Beschleunigung vor.

4.4.2.2 BESCHLEUNIGUNG ALS INKREMENTELLE ÄNDERUNG

```

01 void Update () {
02     velocity = velocity * 1.1F;
03     sphere.transform.Translate(velocity, 0.0F, 0.0F);
04 }

```

CODE 29: BESCHLEUNIGUNG ALS INKREMENTELLE ÄNDERUNG

4.4.2.3 VERZÖGERUNG ALS FUNKTION DER ZEIT

```

01 void Update () {
02     if (velocity >= 0) {
03         velocity = velocity - 1;
04         float x = velocity * Time.deltaTime;
05         sphere.transform.position = new Vector3
06             (x, 0.0F, 0.0F);
07     }
08 }

```

CODE 30: VERZÖGERUNG ALS FUNKTION DER ZEIT

Dies stellt eine lineare Verzögerung dar. Da in Zeile 2 geprüft wird, ob die Geschwindigkeit größer oder gleich null ist, wird sichergestellt, dass es zu keiner Beschleunigung in die entgegengesetzte Richtung kommt.

4.4.2.4 VERZÖGERUNG ALS INKREMENTELLE ÄNDERUNG

```

01 void Update () {
02     if (velocity >= 0) {
03         velocity = velocity - 1;
04         sphere.transform.position = new Vector3
05             (x, 0.0F, 0.0F);
06     }
07 }

```

CODE 31: VERZÖGERUNG ALS INKREMENTELLE ÄNDERUNG

4.4.3 ROTATION

4.4.3.1 ROTATION ALS FUNKTION DER ZEIT

```

01 void Update () {
02     Vector3 eulerAngles = new Vector3(1, 0, 0);
03     cube.transform.Rotate(eulerAngles*Time.deltaTime);
04 }

```

CODE 32: ROTATION ALS FUNKTION DER ZEIT

Bei diesem Beispiel ändert sich die Rotation des Würfels jede Sekunde um 1° um die x-Achse.

4.4.3.2 ROTATION ALS INKREMENTELLE ÄNDERUNG

```

01 void Update () {
02     Vector3 eulerAngles = new Vector3(1, 0, 0);
03     cube.transform.Rotate(eulerAngles*Time.deltaTime);
04 }

```

CODE 33: ROTATION ALS INKREMENTELLE ÄNDERUNG

Bei diesem Beispiel ändert sich die Rotation des Würfels bei jedem Update um 1° um die x-Achse.

4.4.4 WURFPARABEL

```

01 void Update () {
02     var v0 = 10;    // Anfangsgeschwindigkeit
03     var a = 45;     // Abwurfwinkel
04     var g = 9.81;   // Erdbeschleunigung
05     var x = v0 * Mathf.Cos(a);
06     var y = Mathf.Sin(a) * Time.time - (g/2)
07             * Time.time;
08     if(y > 0) sphere.transform.position = new
09         Vector3(x, (float)y, 0.0F);
10 }

```

CODE 34: WURFPARABEL ALS FUNKTION DER ZEIT

Durch die if-Bedingung in Zeile 7 wird sichergestellt, dass die Kugel keine negativen y-Koordinaten bekommt, sprich durch den Boden fällt.

4.4.5 REIBUNGSSIMULATION

Reibung als Funktion der Zeit

```

01 void Update () {
02     if (velocity >= 0.1) {
03         float friction = 0.95F;
04         velocity = velocity *
05             (friction * Time.deltaTime);
06     }
07 }

```

05	sphere.transform.Translate
06	(velocity, 0.0F, 0.0F);
	}
07	}

CODE 35: REIBUNG ALS FUNKTION DER ZEIT

Da in Zeile 2 überprüft wird, ob die Geschwindigkeit größer oder gleich 0,1 ist, wird verhindert, dass die Sphäre lange braucht, um zum vollständigen Stillstand zu kommen.

Reibung als inkrementelle Änderung

01	void Update () {
02	if (velocity >= 0.1) {
03	float friction = 0.95F;
04	velocity = velocity * friction * Time.deltaTime);
05	sphere.transform.Translate
	(velocity, 0.0F, 0.0F);
06	}
07	}

CODE 36: REIBUNG ALS INKREMENTELLE ÄNDERUNG

4.4.6 GRAVITATION

01	const float g = -9.81F;
02	void Update() {
03	if (sphere.transform.position.y > 0) {
04	sphere.transform.Translate(0, g * Time.time, 0);
05	}
06	}

CODE 37: GRAVITATION

Da die Erdbeschleunigung eine Konstante ist und sich die Variable „g“ nicht ändert, kann man dies demonstrieren, indem man das Schlüsselwort „const“ zur Variable hinzufügt. In Zeile 3 wird überprüft, ob die Sphäre nicht schon am Boden liegt.

4.4.7 ELASTISCHE BEWEGUNG

Als Beispiel für eine elastische Bewegung wurde ein aufspringender Ball gewählt. Dabei greifen einige der erläuterten physikalischen Formeln und Funktionen von Unity ineinander.

01	public GameObject bounceBall;
02	
03	const float g = -9.81F;
04	float t;
05	
06	float change;
07	int factor = 100;

```

08
09 float upward;
10
11 bool up;
12 bool lastUp;
13
14 float bounciness = 0.7F;
15
16 void Start () {
17     t = Time.time;
18 }
19
20 void FixedUpdate () {
21     if(bounceBall.transform.position.y > 0) {
22         change = g * (Time.time - t);
23         bounceBall.transform.Translate(0, upward +
24             change / factor, 0);
25
26         lastUp = up;
27
28         if (upward + change / factor > 0.0F) up =
29             true;
30         else up = false;
31
32         if (up == false && lastUp == true) {
33             t = Time.time;
34             upward = 0;
35         }
36     }
37     else {
38         t = Time.time;
39         upward = change / factor * (-1) *
40             bounciness;
41         if(upward > 0.01)
42             bounceBall.transform.Translate(0,
43                 upward, 0);
44     }
45 }

```

CODE 38: AUFSPRINGENDER BALL SIMULATION

Die Variable „change“ bildet die Änderung der y-Koordinate des Balls ab. „factor“ ist eine Variable zum Beeinflussen der Ablauf-Geschwindigkeit der Simulation. „upward“ stellt die Kraft dar, welche nach einem Abprall auf den Ball wirkt. Die bool-Werte „up“ und „lastUp“ dienen zum Vergleich, ob der Scheitelpunkt des Balls überschritten wurde. „bounciness“ gibt an, wieviel Energie beim Aufprall verloren ging (beziehungsweise in nicht-kinetische Energie übergang).

In Zeile 21 wird überprüft, ob der Ball nicht bereits am Boden liegt.

Ist dies nicht der Fall, befindet sich der Ball im freien Fall.

In Zeile 25 wird die Richtung der letzten Bewegung gespeichert, bevor die Richtung der aktuellen Bewegung geprüft wird.

War die letzte Bewegung nach oben, die aktuelle aber nach unten, so hat der Ball den Scheitelpunkt überschritten. Da am Scheitelpunkt die kinetische Energie gleich null ist und keine kinetische Energie von der Zeit zwischen Abprall und Erreichen des Scheitelpunktes erhalten bleibt, fängt die Beschleunigung des Balls bei null an. Daher muss die Zeit auf null gesetzt werden. Weiters wird die nach oben gerichtete Kraft des Abpralls auf null gesetzt.

Liegt der Ball aktuell am Boden (Zeile 35), wird wie beim Scheitelpunkt die Zeit zurückgesetzt. Außerdem wird berechnet, wieviel Kraft auf den Ball nach oben wirkt. Diese ergibt sich aus der Aufprallgeschwindigkeit multipliziert mit -1 , um sie nach oben umzukehren.

Zuletzt wird in Zeile 38 geprüft, ob der Ball überhaupt noch weit genug aufspringt, um ein Zittern auszuschließen, welches von immer kleiner werdenden Werten stammt.

4.4.8 ZENOS PARADOXON

Zenos Paradoxon beschreibt zwei Objekte, wobei das erste Objekt sich unabhängig vom anderen bewegt. Das zweite Objekt folgt dem ersten Objekt – je weiter die Distanz zwischen den Objekten ist, desto schneller. Befinden sich beide Objekte nahe beieinander, wird die Annäherung des zweiten Objektes immer langsamer, sodass es nahezu die Position des ersten Objektes erreicht, aber nie wirklich.

01	<code>void zenoTransform(GameObject destination,</code>
	<code>GameObject origin, int coeff)</code>
02	<code>{</code>
03	<code>var d = destination.transform.position;</code>
04	<code>var o = origin.transform.position;</code>
05	<code>var zeno = (d - o) / coeff;</code>
06	<code>origin.transform.Translate(zeno.x, zeno.y,</code>
	<code>zeno.z);</code>
07	<code>}</code>

CODE 39: ZENOS PARADOXON

Bei dieser Funktion, welche Zenos Paradoxon für zwei Objekte implementiert, stellt „destination“ das erste Objekt dar, welches verfolgt wird, und „origin“ das zweite Objekt, welches dem ersten folgt. Über den Koeffizienten kann ein Wert mitgegeben werden, welcher bestimmt, wie schnell das zweite Objekt dem ersten folgt.

5 VISUALISIERUNG UND DEMONSTRATIONEN

Zum Zweck der Visualisierung einzelner Abschnitte wurden sechs Demonstrationsapplikationen geschrieben. Diese sind auf der der Arbeit beigelegten DVD zu finden.

Weiters wird in diesem Kapitel auf aufgetretene Herausforderungen eingegangen.

5.1 ANWENDUNG DES GELERNTEN/INTERAKTIVE DEMONSTRATIONEN

Alle Demonstrationen können über ein Menü ausgewählt werden. Von allen Demonstrationen kann mittels Escape zum Hauptmenü zurück gelangt werden.

FH JOANNEUM - UNIVERSITY OF APPLIED SCIENCES
Interaktive Demonstrationen für die Bachelorarbeit 1:
Dreidimensionale Repräsentation
von Algorithmen in Unity



Verfasser: Lukas Schneider
Betreuer: FH-Prof. Dipl.-Ing. Dr. Alexander Nischelwitzer



ABBILDUNG 11: AUSWAHLMENÜ DER DEMONSTRATIONEN

5.1.1 DEMO – SCHNITTPUNKT RAY/EBENE

In dieser Demonstration wird mit der Maus ein blauer Würfel gesteuert. Ausgehend von einer roten Kugel wird ein Strahl erzeugt, welcher durch den blauen Würfel führt.

Zwischen Kugel und Würfel befindet sich eine Ebene. An der Position, an welcher der Strahl auf die Ebene trifft, wird ein grüner Würfel eingezeichnet.

Anmerkung: Da Rays, wie im Abschnitt „Darstellung von Rays“ beschrieben, nicht gezeichnet werden, wenn eine Szene kompiliert wird, wurde der Strahl als „LineRender“ gezeichnet.

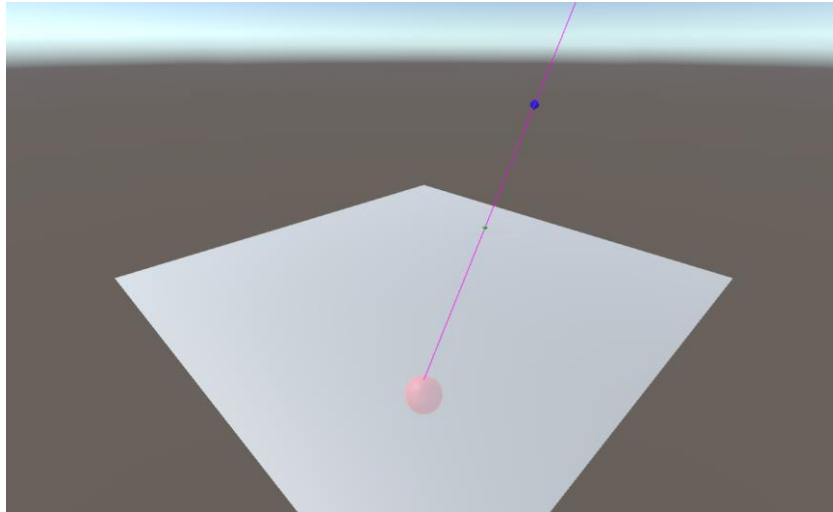


ABBILDUNG 12: DEMONSTRATION SCHNITTPUNKT RAY/EBENE I

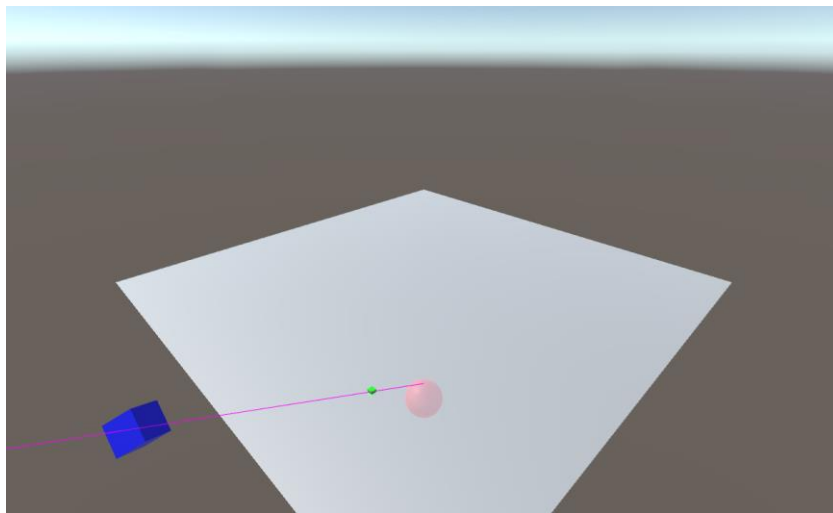


ABBILDUNG 13: DEMONSTRATION SCHNITTPUNKT RAY/EBENE II

Dafür werden zwei Skripte benutzt.

```
01 public class Mouse : MonoBehaviour {  
02  
03     int width;  
04     int height;  
05  
06     void Start () {  
07         width = Screen.width;  
08         height = Screen.height;  
09     }  
10  
11     void Update() {  
12         var x = Input.mousePosition.x / width;  
13         var y = Input.mousePosition.y / height;  
14     }
```


15	transform.position = new Vector3(-x * 10, 6,
16	y * 10);
17	}

CODE 40: MOUSE-SKRIPT

Dieses Skript sorgt dafür, dass der blaue Würfel sich immer entsprechend der Maus bewegt. Da sich der Würfel immer über der Ebene befindet, wurde seine y-Koordinate auf 6 festgelegt (Zeile 15).

01	public class Raycast : MonoBehaviour {
02	
03	public Transform markerObject;
04	public GameObject mouse;
05	
06	Plane plane = new Plane();
07	
08	void Start () {
09	plane.Set3Points(new Vector3(0, 3, 0), new
10	Vector3(1, 3, 0), new Vector3(0, 3, 1));
11	}
12	void Update () {
13	Vector3 direction = new Vector3(
	mouse.transform.position.x,
	mouse.transform.position.y,
	mouse.transform.position.z);
14	Vector3 point = new Vector3(0, 0, 0);
15	Ray ray = new Ray(point, direction - point);
16	float rayDistance;
17	plane.Raycast(ray, out rayDistance);
18	Debug.DrawRay(point, direction*10, Color.magenta,
	0.1f);
19	markerObject.position = ray.GetPoint(rayDistance);
20	}
21	}

CODE 41: RAYCAST-SKRIPT

Dieses Skript sorgt dafür, dass eine Ebene erstellt wird, welche normal zur x- und z-Achse liegt und y=3 erfüllt (Zeile 3).

Bei jedem Update wird ein Strahl zwischen der roten Kugel, welche im Koordinatenursprung liegt (Zeile 14), und dem blauen Würfel, welcher durch die Mausposition bestimmt wird (Zeile 13), gezeichnet (Zeile 18). In Zeile 19 wird der Schnittpunkt mit dem „markerObject“ – dem grünen Würfel – gekennzeichnet.

5.1.2 DEMO – SCHNITTGERADE EBENE/EBENE

In dieser Demonstration wird mit der Maus eine grüne Ebene gesteuert. Eine zweite, blaue Ebene liegt unbewegt unter der grünen Ebene.

Dort, wo sich beide Ebenen schneiden, wird eine Gerade eingezeichnet.

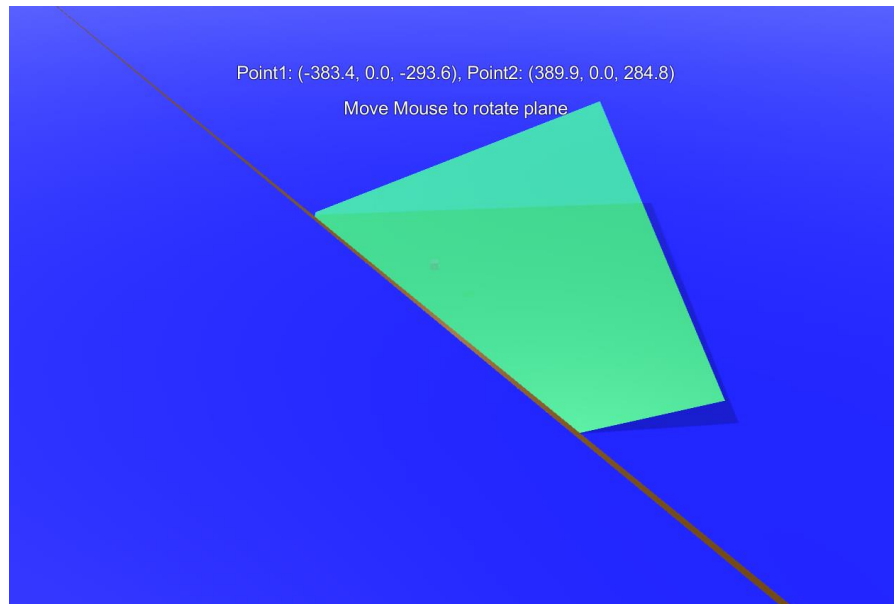


ABBILDUNG 14: DEMONSTRATION SCHNITTGERADE EBENE/EBENE

Weiters werden zwei Punkte, welche auf der Schnittgeraden liegen, berechnet und ausgegeben.

Die Bewegung der Ebene und Berechnung der Schnittgerade erfolgt über ein Skript.

```

01 void Update () {
02     nP2 = Vector3.Normalize(mouse.transform.position);
03     anchor.transform.up = nP2;
04
05     lineVec = Vector3.Cross(nP1, nP2);
06     ldir = Vector3.Cross(nP2, lineVec);
07
08     denominator = Vector3.Dot(nP1, ldir);
09
10     if (Mathf.Abs(denominator) > 0.0001F) {
11         plane1ToPlane2 = pP1 - pP2;
12         t = Vector3.Dot(nP1, plane1ToPlane2) /
            denominator;
13         linePoint1 = pP2 + t * ldir;
14
15         linePoint2 = linePoint1 + lineVec * 1000;
16         linePoint3 = linePoint1 + lineVec * -1000;
17
18         lineRenderer.SetPosition(0, linePoint2);
    
```

19	lineRenderer.SetPosition(1, linePoint3);
20	
21	info.text = "Point1: " + linePoint2 + ", Point2: " + linePoint3;
23	}
24	else info.text = "The planes are parallel.";
25	}

CODE 42: MOVEMENT-SKRIPT

Die Ausrichtung der grünen Ebene erfolgt über ein Hilfsobjekt, den „anchor“. Dieses wird in Zeile 3 so ausgerichtet, dass dessen „up“-Vektor der normalisierte Ortsvektor des „Mouse“-Objektes ist. Dieses „Mouse“-Objekt wird von der Maus gesteuert.

Anschließend erfolgt die Berechnung der Schnittgerade wie im gleichnamigen Abschnitt beschrieben.

Um mittels „LineRenderer“ eine lange Strecke, welche eine Gerade repräsentiert, zu zeichnen, wird der errechnete Schnittpunkt mit 1000 und -1000 multipliziert (Code-Zeile 15 und 16).

5.1.3 DEMO – WURFPARABEL

Die dritte Demonstration stellt eine Wurfparabel dar. Dazu wird ein roter Ball geworfen. Die Wurfkraft kann mittels Maus bestimmt werden. Zusätzlich zur Wurfparabel führt der Ball eine elastische Bewegung aus, er springt also auf.

Die Geschwindigkeit des Balles wird in LU/s angezeigt, also in „Length Units“ pro Sekunde.

Die Demonstration kann mittels Leertaste zurückgesetzt werden.

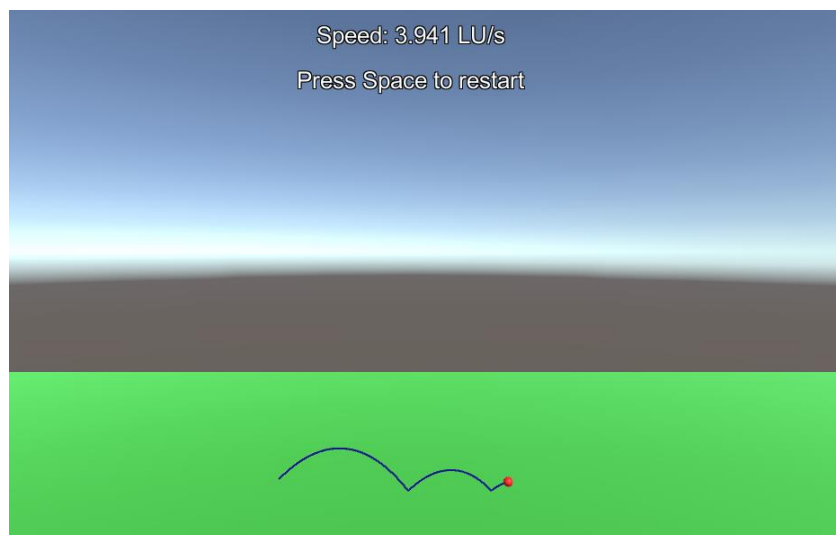


ABBILDUNG 15: DEMONSTRATION WURFPARABEL

01	void FixedUpdate () {
02	if (release) {

```

03     if (ball.transform.position.y > 0.4F) {
04         speed = speed * 0.999F;
05
06         change = g * (Time.time - t);
07         ball.transform.Translate(speed , upward +
08             change / factor, 0);
09
10         lastUp = up;
11
12         if (upward + change / factor > 0.0F) up =
13             true;
14         else up = false;
15
16         if (up == false && lastUp == true) {
17             t = Time.time;
18             upward = 0;
19         }
20     }
21     else {
22         speed = speed * 0.95F;
23
24         t = Time.time;
25         upward = change / factor * (-1) * bounciness;
26         if (upward > 0.01)
27             ball.transform.Translate
28                 (speed, upward, 0);
29         else ball.transform.Translate(speed, 0, 0);
30     }
31     info.text = "Speed: " + Math.Round(speed /
32         Time.fixedDeltaTime, 3) + " LU/s";
33 }
34
35 else {
36     if (Input.GetMouseButton(0)) {
37         if (click < 0 || click > 0.3F) x = -x;
38         click = click + Time.fixedDeltaTime *
39             x;
40         info.text = "Strength: " +
41             Math.Round(click, 2);
42     }
43 }
44
45 void Update() {
46     if (!release && Input.GetMouseButtonUp(0)) {
47         release = true;
48         t = Time.time;
49         upward = click;

```

```

44     }
45
46     if(Input.GetKeyDown(KeyCode.Space))
47         SceneManager.LoadScene("Demo_Throw");

```

CODE 43: THROW-SKRIPT

Das „Throw“-Skript baut auf dem Skript für elastische Bewegung auf. In Zeile 2 wird überprüft, ob die linke Maustaste bereits losgelassen wurde.

In Code-Zeile 4 und Zeile 20 wird der Luft- beziehungsweise der Rollwiderstand berechnet.

In Code-Zeile 28 wird die Geschwindigkeit des Balles berechnet. Dies erfolgt, indem die aktuelle Geschwindigkeit mit der Berechnungszeit der „FixedUpdate“-Funktion multipliziert wird.

In den Zeilen 30 bis 36 wird die Kraft des Abwurfes festgelegt.

In Zeile 40 wird die Abwurfbedingung festgelegt – dazu darf der Ball noch nicht geworfen worden sein und die linke Maustaste muss gerade losgelassen worden sein. Trifft dies zu, wird die Zeit auf null gesetzt und die Kraft festgelegt.

5.1.4 DEMO – ZENOS PARADOXON/ELASTISCHE BEWEGUNG

Die vierte Demonstration stellt eine Kombination aus den Abschnitten Zenos Paradoxon und Elastische Bewegung des vierten Kapitels dar. Dabei wird eine Basis mit der Maus bewegt, welche durch ein rotes Rechteck repräsentiert wird. Diese Basis wird als Zielobjekt für eine blaue Kugel benutzt, welche sowohl die Bewegung von Zenos Paradoxon als auch eine aufspringende elastische Bewegung durchführt.

Zur besseren Visualisierung der Bewegung des Balles durch den Raum wurde zusätzlich noch ein „Trail Renderer“ (Grün) hinzugefügt.

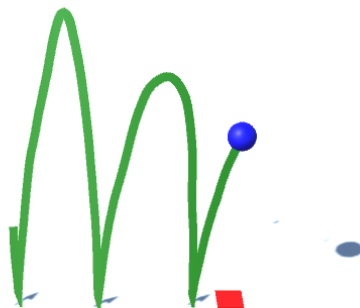


ABBILDUNG 16: DEMONSTRATION ZENOS PARADOXON/ELASTISCHE BEWEGUNG

Dafür wurden erneut zwei Skripte verwendet. Das erste bindet die Mausbewegungen auf die rote Basis und funktioniert analog zum Mouse-Skript in der ersten Demonstration.

Das zweite Skript bestimmt die Bewegung des blauen Balles.

```

01 public class Bounce : MonoBehaviour {
02
03     public GameObject bounceBall;
04     public GameObject zenoBase;
05
06     float g = -9.81F;
07     float t;
08
09     float change;
10     int factor = 100;
11
12     float upward;
13
14     bool up;
15     bool lastUp;
16
17     float bounciness = 0.9F;
18
19     void Start () {
20         t = Time.time;
21     }
22
23     void FixedUpdate () {
24
25         if (bounceBall.transform.position.y > 0.4F) {
26             change = g * (Time.time - t);
27             zenoTransform(zenoBase, bounceBall, 15);
28             lastUp = up;
29
30             if (upward + change / factor > 0.0F) {
31                 up = true;
32             }
33             else up = false;
34
35             if (up == false && lastUp == true) {
36                 t = Time.time;
37                 upward = 0;
38             }
39         }
40         else {
41             t = Time.time;
42

```

```

43         upward = change / factor * (-1) *
44             bounciness;
45         if (upward > 0.01) {
46             bounceBall.transform.Translate(0,
47                 upward, 0);
48         }
49     }
50
51 void zenoTransform (GameObject destination,
52     GameObject origin, int coeff) {
53     var d = destination.transform.position;
54     var o = origin.transform.position;
55     var zeno = (d - o) / coeff;
56     origin.transform.Translate(zeno.x, upward +
57         change / factor, zeno.z);
58 }

```

CODE 44: BOUNCE SKRIPT

Dieses Skript sorgt dafür, dass der Ball mit 90 % (Zeile 17) der Aufschlaggeschwindigkeit aufspringt. Das restliche Skript setzt sich aus den Skripten zusammen, welche für die Absätze „Elastische Bewegung“ und „Zenos Paradoxon“ geschrieben wurden. Die dortigen Ausführungen treffen auch hier zu.

5.1.5 DEMO – ZENO 3D

In dieser Demonstration wird Zenos Paradoxon im dreidimensionalen Raum dargestellt. Der rote Würfel wird per Maus gesteuert, wobei die Mausbewegung die x- und z-Koordinaten steuert und das Mausrad die y-Koordinaten.

Über die „A“- und „Y“-Tasten kann der Faktor angepasst werden, mit welchem bestimmt wird, wie schnell oder langsam die blaue Kugel dem roten Würfel folgt.

Weiters wird die Distanz zwischen den beiden Objekten in Längeneinheiten berechnet und angezeigt.

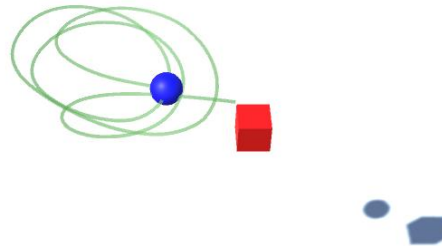
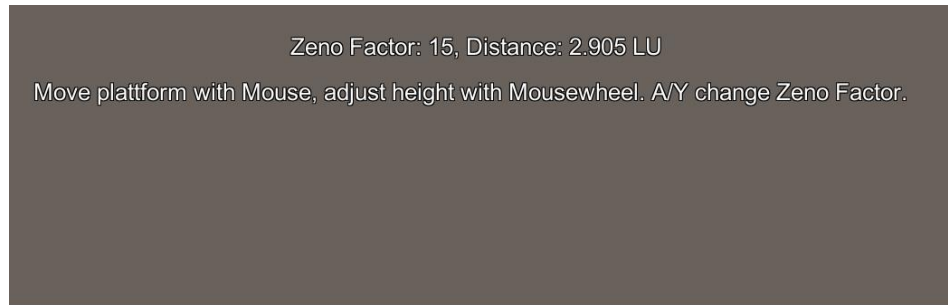


ABBILDUNG 17: DEMONSTRATION ZENO3D

5.1.6 DEMO – BEWEGUNGSÜBERLAGERUNG

In dieser Demonstration wird gezeigt, wie drei eindimensionale Sinus- und Cosinus-Bewegungen übereinander gelegt eine komplexe dreidimensionale Bewegung ergeben. Diese Bewegung beschreibt den Rand eines Hyperparaboloids. Ein bekanntes Beispiel dafür wäre ein Pringles-Chip.

Die Achsen können einzeln an- und abgewählt werden, um den Zusammenhang der Bewegungen deutlich zu machen.

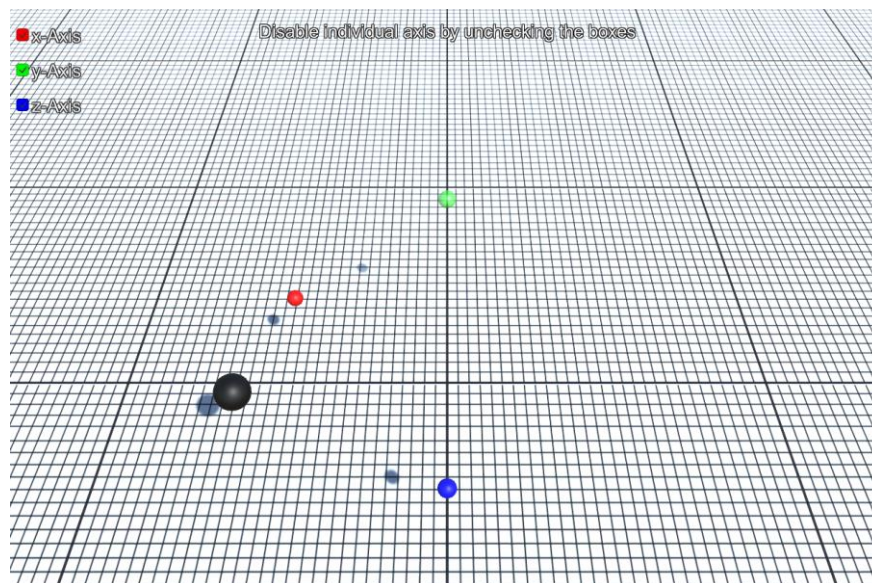


ABBILDUNG 18: DEMONSTRATION BEWEGUNGSÜBERLAGERUNG

5.2 PROBLEME & HERAUSFORDERUNGEN

Die meisten Probleme und Herausforderungen bezüglich Algorithmen könnten über eine Google-Suche gelöst werden, da sowohl C# als auch Unity im Internet sehr gut dokumentiert sind. Dieser Teil der Arbeit lief weitgehend reibungslos ab. Ist eine Herausforderung dennoch problematisch, so kann man meist eine andere Herangehensweise wählen.

Die mathematischen und physikalischen Formeln stellten ebenfalls kein Problem dar, da sich auch hierfür genügend Ressourcen im Internet finden lassen.

Dafür erwies sich ein anderer, für diese Arbeit essenzieller, Aspekt als Herausforderung: die Visualisierung von dreidimensionalen Punkten auf zweidimensionalen Abbildungen.

Dies kann durch einen Punkt $(1,2,3)$, welcher auch in Kapitel 2 verwendet wurde, dargestellt werden.

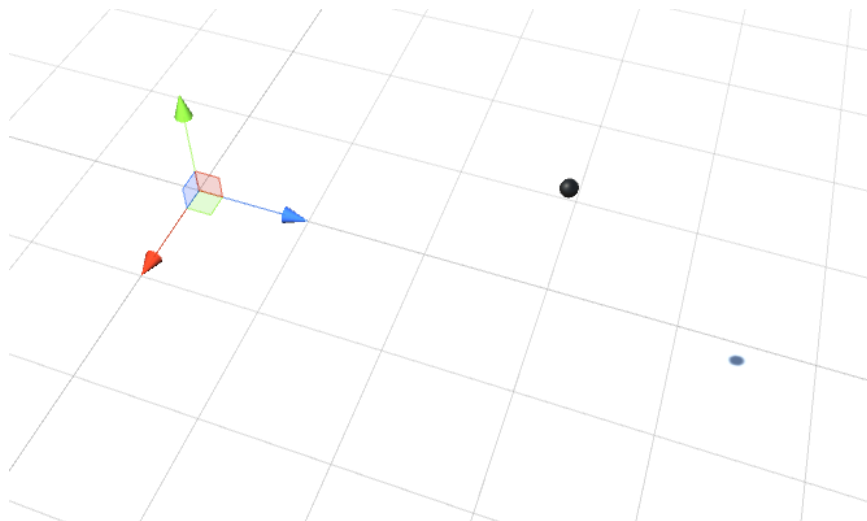


ABBILDUNG 19: PUNKT OHNE VISUELLE HILFSMITTEL

Durch den Schatten erkennt man zwar, dass der Punkt nicht auf der Ebene ist. Wo im Raum er sich genau befindet, kann man jedoch nicht sagen.

Im nächsten Schritt wurden zusätzlich zur xz-Ebene auch eine xy- und zy-Ebene eingeblendet.

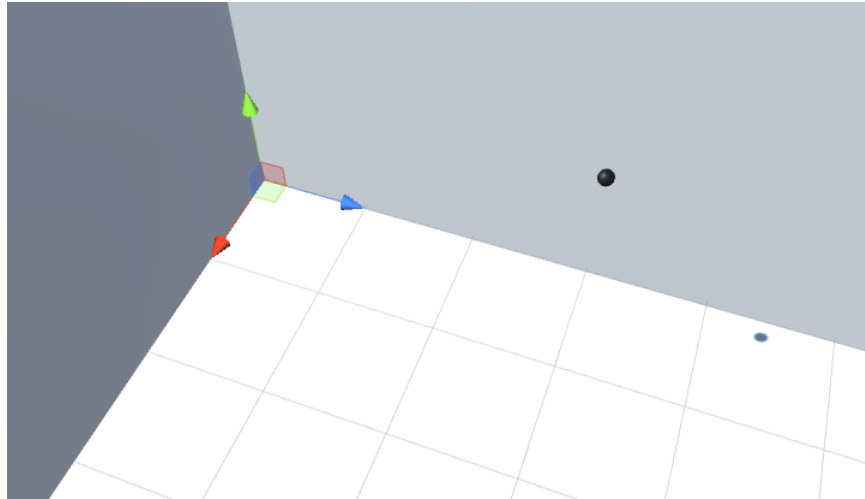


ABBILDUNG 20: PUNKT MIT ZUSÄTZLICHEN EBENEN

Durch die zusätzlichen Ebenen lässt sich nun die Position des Punktes besser ermitteln, eine exakte Position ist aber noch immer nicht erkennbar.

Zu diesem Zweck wurden Hilfslinien eingezeichnet, welche parallel zu den Achsen verlaufen.

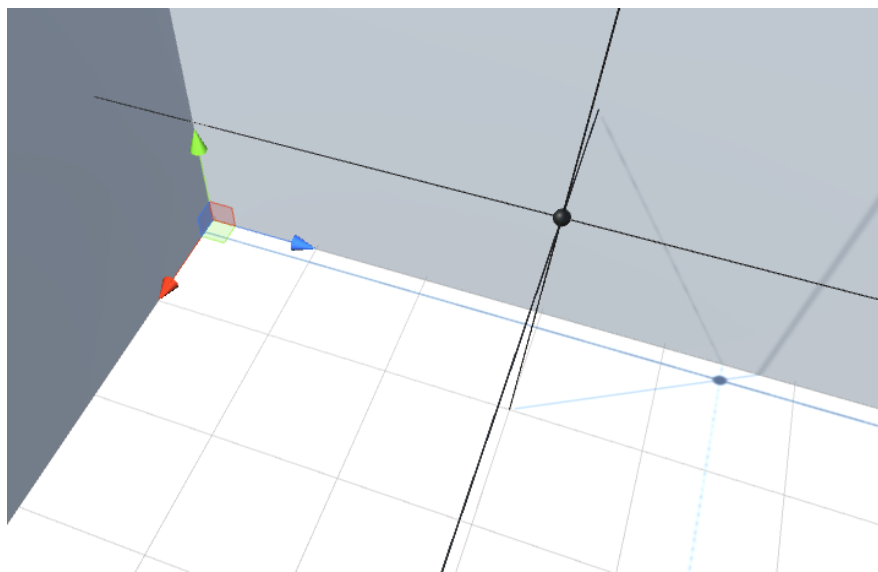


ABBILDUNG 21: PUNKT MIT HILFSLINIEN

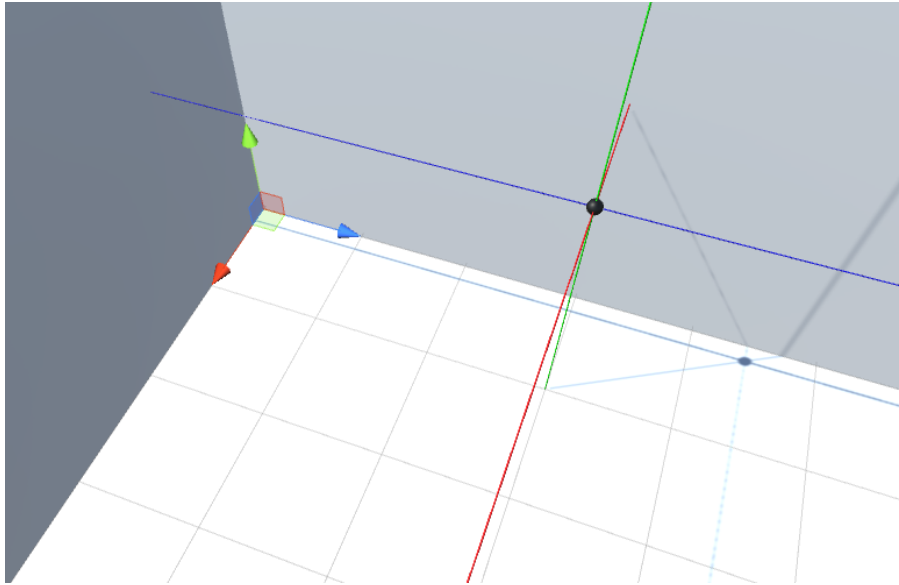


ABBILDUNG 22: PUNKT MIT HILFSLINIEN IN FARBEN DER AXSEN

Im letzten Schritt wurden die Hilfslinien farblich auf die Achsen in Unity abgestimmt, wodurch die wohl beste visuelle Repräsentation eines dreidimensionalen Punkts mit einer zweidimensionalen Abbildung erreicht wurde.

Ein weiterer wichtiger Aspekt bei der Darstellung einer räumlichen Abbildung ist die Perspektive.

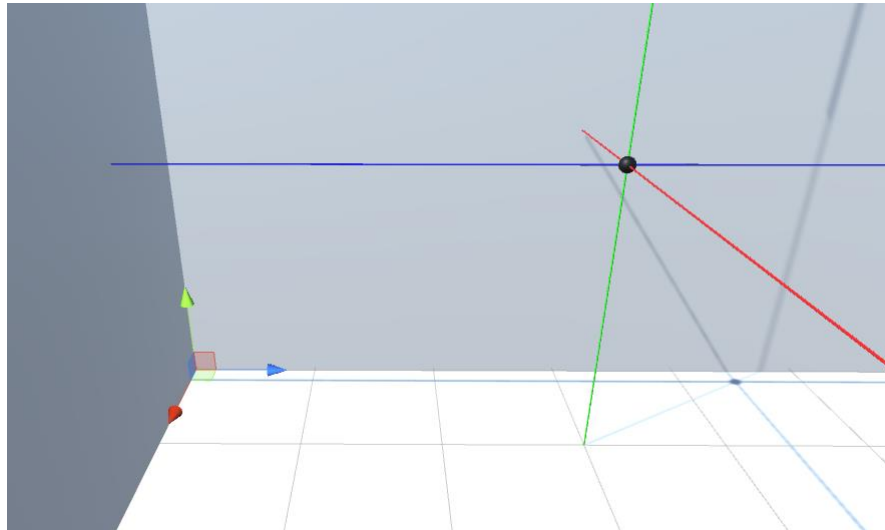


ABBILDUNG 23: PUNKT MIT ANDERER PERSPEKTIVE

Diese Abbildung unterscheidet sich von der letzten Abbildung nur durch die Platzierung der Kamera. Aufgrund der Lage der Hilfsachsen ist jedoch eine genauere Bestimmung der Position des Punktes im Raum möglich.

6 SCHLUSSFOLGERUNG

Eignet sich Unity nun also dazu, dreidimensionale Algorithmen darzustellen?

Ja, Unity eignet sich ideal, um Berechnungen und Simulationen im dreidimensionalen Raum durchzuführen und darzustellen. So bringt Unity standardmäßig viele Klassen, Methoden und Funktionen mit, welche für eben jene Berechnungen essenziell sind und einen schnellen Einstieg ermöglichen.

Soll jedoch nicht auf diese vorgefertigten Codestellen zurückgegriffen werden, lassen sich diese auch mit entsprechendem Aufwand selbst implementieren. Dies bietet den Vorteil des genauen Verständnisses, der Anpassbarkeit nach eigenen Vorstellungen sowie eine mögliche höhere Genauigkeit (siehe Abschnitt „Distanz zwischen zwei Punkten“).

Weiters konnte gezeigt werden, inwiefern sich Algorithmen in Unity und C# umsetzen lassen und welche Faktoren die Ergebnisse auf welche Weise beeinflussen.

Mathematische und physikalische Formeln lassen sich ebenfalls auf verschiedenen Wegen umsetzen, so können Formeln von der Zeit abhängig sein oder inkrementell berechnet werden. Folglich gibt es nicht eine einzige, sondern nahezu unbegrenzt viele Möglichkeiten, eine Problemstellung zu bearbeiten.

Es wurden die Funktionen und Methoden erklärt, welche in Unity verwendet werden können, um zeitbezogene Berechnungen durchzuführen.

Die Visualisierung von Algorithmen kann mittels Unity ebenfalls sehr umfangreich vorgenommen werden, wie in Kapitel 5 anhand der interaktiven Demonstrationen belegt werden konnte. Unity erweist sich auch in dieser Hinsicht als vielfältig einsetzbar.

Dementsprechend erhebt diese Arbeit auch keinen Anspruch darauf, die einzig richtigen Lösungen auf die jeweiligen Problemstellungen zu bieten, sondern vielmehr für jedes Problem eine Lösungsmöglichkeit zu beschreiben und die gebotenen Freiheiten aufzuzeigen.

LITERATURVERZEICHNIS

[Seifert 2015]

Seifert C.: Spiele entwickeln mit Unity 5 - 2D- und 3D-Games mit Unity und C# für Desktop, Web & Mobile. Hanser Verlag 2015

[Papula 2009]

Papula L.: Mathematische Formelsammlung - für Ingenieure und Naturwissenschaftler. Vieweg+Teubner Verlag 2009

[Kurzweil, Frenzel, Gebhard 2008]

Kurzweil P., Frenzel B., Gebhard F.: Physikformelsammlung – für Ingenieure und Naturwissenschaftler. Vieweg & Sohn Verlag 2008

[<https://docs.unity3d.com/>]

- [1] Script Reference – Space; <https://docs.unity3d.com/ScriptReference/Space.html>; 29.9.2017
- [2] Script Reference – MonoBehaviour; <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>; 29.9.2017;
- [3] Script Reference – FixedUpdate; <https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html>; 29.9.2017
- [4] Manual – TimeManager; <https://docs.unity3d.com/Manual/class-TimeManager.html>; 29.9.2017;
- [5] Script Reference – Transform; <https://docs.unity3d.com/ScriptReference/Transform.html>; 29.9.2017;

[<https://de.serlo.org/>]

- [1] Abstand Punktes einer Geraden berechnen analytische Geometrie; <https://de.serlo.org/mathe/geometrie/analytische-geometrie/abstaende-winkel/abstaende/abstand-punktes-einer-geraden-berechnen-analytische-geometrie>; 29.9.2017
- [2] Ebene von Parameterform in Normalform umwandeln; <https://de.serlo.org/entity/view/1893>; 29.9.2017

[<http://wiki.unity3d.com>]

[1] 3d Math functions; [http://wiki.unity3d.com/index.php/3d Math functions](http://wiki.unity3d.com/index.php/3d_Math_functions); 29.9.2017

[<http://www.virtual-maxim.de>]

[1] Freier Fall mit und ohne Luftwiderstand; <http://www.virtual-maxim.de/downloads/freier%20fall%20mit%20und%20ohne%20luftwiderstand.pdf>, Seite 4; 29.9.2017

[2] Freier Fall mit und ohne Luftwiderstand; <http://www.virtual-maxim.de/downloads/freier%20fall%20mit%20und%20ohne%20luftwiderstand.pdf>, Seite 7; 29.9.2017

[3] Freier Fall mit und ohne Luftwiderstand; <http://www.virtual-maxim.de/downloads/freier%20fall%20mit%20und%20ohne%20luftwiderstand.pdf>, Seite 8; 29.9.2017

[<https://rechneronline.de>]

[1] Fallgeschwindigkeit; <https://rechneronline.de/g-beschleunigung/fallgeschwindigkeit.php>; 29.9.2017

[<http://www.chemie.de>]

[1] Luftdichte; <http://www.chemie.de/lexikon/Luftdichte.html>; 29.9.2017

[<http://www.spektrum.de>]

[1] Gewichtskraft; <http://www.spektrum.de/lexikon/physik/gewichtskraft/5824>; 29.9.2017

[<http://www.maschinenbau-wissen.de>]

[1] Schiefer Wurf; <http://www.maschinenbau-wissen.de/skript3/mechanik/kinematik/334-schiefer-wurf>; 29.9.2017