

Algorithmen für 2D Bewegungen

mit Unity 5

Bachelorarbeit 1

zur Erlangung des akademischen Grades „Bachelor of Science in Engineering“

eingereicht am Studiengang Informationsmanagement

Verfasser:

Felix Rauchenwald

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)

[Name]

[Ort, Datum]

INHALTSVERZEICHNIS

KURZFASSUNG	6
ABSTRACT.....	7
1 EINLEITUNG	8
1.1 Motivation.....	8
1.2 Ziel.....	8
1.3 Vorgehensweise	8
2 NATURWISSENSCHAFTLICHE GRUNDLAGEN	9
2.1 Gleichförmige Bewegung	9
2.2 Konstante Beschleunigung.....	9
2.3 Gravitation	10
2.4 Euler - Cromer Methode	11
2.5 Punkte	11
2.6 Vektoren.....	11
2.7 Koordinatensysteme.....	16
2.8 Parameterdarstellung eines Kreises	16
2.9 Matrizen	17
3 GRUNDLAGEN UNITY5 UND C#.....	22
3.1 Programmierung mit C#	23
3.2 Darstellung im 2D Koordinatensystem	24
3.3 Physics Engine.....	31
4 ALGORITHMEN UND VISUALISIERUNG IN 2D	33
4.1 Gleichförmige Bewegung	33
4.2 Gleichmäßig beschleunigte Bewegung	33
4.3 Zeno's Paradoxon	34
4.4 Winkelbewegung	35
4.5 Robotik Gelenk	42
4.6 Elastische Bewegung.....	43
4.7 PingPong Spiel	44
4.8 Billard Spiel.....	45
5 ANWENDUNGSBEREICHE	46
5.1 Animation.....	46
5.2 Spiele.....	47
5.3 Gestenerkennung	48
6 RESÜMEE	49
LITERATURVERZEICHNIS	50

ABBILDUNGSVERZEICHNIS

Abbildung 1 – Parallelvektoren, x in Blau, y in Rot.....	12
Abbildung 2 - Summe zweier Vektoren.....	13
Abbildung 3 - Differenz zweier Vektoren	13
Abbildung 4 - Kreuzprodukt zweier Vektoren	14
Abbildung 5 - Kartesisches Koordinatensystem	16
Abbildung 6 – Polarkoordinaten	16
Abbildung 7 - Verschieben eines Punktes	19
Abbildung 8 - Drehung eines Punktes	20
Abbildung 9 - Spiegelung von Punkten über eine Gerade.....	21
Abbildung 10 - Linkshändiges Koordinatensystem.....	24
Abbildung 11 - Unity: 2D Koordinatensystem.....	24
Abbildung 12 - Unity: Punkt in 2D Darstellung.....	25
Abbildung 13 – Unity: Punkt im 3D Raum	26
Abbildung 14 - Unity: LineRenderer 2D.....	28
Abbildung 15 - Unity: Ray (Strahl)	29
Abbildung 16 - Unity: Ausgabe der Geschwindigkeit per Intervall.....	34
Abbildung 17 - Unity: Darstellung Zeno's Paradoxon.....	35
Abbildung 18 - Unity: Darstellung der Sinusfunktion	36
Abbildung 19 - Unity: Vergleich von Update und FixedUpdate.....	36
Abbildung 20 - Unity: Wurfparabel	37
Abbildung 21 - Unity: Pendel.....	39
Abbildung 22 - Unity: Darstellung Kreis	41
Abbildung 23 - Consolen Output Kreis	41
Abbildung 24 - Unity: Robo Arm	42
Abbildung 25 - Unity: Pingpong Spiel	44
Abbildung 26 - Unity: Billiard Spiel.....	45
Abbildung 27 - Bestandteile eines Robo Arms [http://wiki.ifs-tud.de/] []	46
Abbildung 28 - Motion Capture: Reales Skelett zu digitalem Skelett. [http://graphics.berkeley.edu/] [1].....	47
Abbildung 29 - 2D Jump and Run: Super Mario ([http://nintendo-online.de/] [1]	48

CODEVERZEICHNIS

Code 1 - Default C# Skript aus Unity	23
Code 2 - Initialisieren eines Vector2 Typs	25
Code 3 - Berechnen der Länge eines Vektors mit magnitude	26
Code 4 - Berechnung der Distanz zweier Punkte	27
Code 5 - Berechnen des Richtungsvektors zweier Punkte	27
Code 6 - Distanz zwischen zwei Punkten mittels magnitude	27
Code 7 - Normalisieren eines Vector2.....	27
Code 8 - Erstellen einer Gerade mit LineRenderer.....	28
Code 9 - Darstellung eines Ray im Szenen Modus	28
Code 10 - Festlegen der Position eines Objektes	29
Code 11 - Rotation eines Objektes mit eulerAngles.....	30
Code 12 - Rotation eines Objektes mit Rotate	30
Code 13 - Rotation relativ zur Weltachsen.....	30
Code 14 - Rotieren um eine Position mit RotateAround.....	30
Code 15 - Ausrichten zu einem Zielobjekt mit LookAt	30
Code 16 - Verschieben eines Objektes mit Translate.....	31
Code 17 - Erstellen der Rigidbody Komponente	31
Code 18 - Bestimmen des Schwerpunktes mit centerOfMass	32
Code 19 - Gleichförmige Bewegung eines Objektes	33
Code 20 - Gleichmäßige Beschleunigung eines Objektes.....	33
Code 21 - Algorithmus für Zeno's Paradoxon.....	34
Code 22 - Algorithmus für eine Sinusfunktion	35
Code 23 – Wurfparabel	37
Code 24 - Implementation Pendel	38
Code 25 - Algorithmus für Kreisbewegung.....	40
Code 26 - Robo Arm	42
Code 27 - Springender Ball.....	43
Code 28 - Pingpong Spiel.....	44
Code 29 - Wichtigste Berechnungen für Billiard Beispiel	45

KURZFASSUNG

Diese Bachelorarbeit befasst sich mit Theorie, Umsetzung und Visualisierung von Algorithmen für 2D Bewegungen. Die Visualisierung wird mit der Unity5 Engine dargestellt.

Es werden vertiefend 2D Algorithmen analysiert. Zu den visuell dargestellten Ergebnissen zählen Bewegungen wie gleichförmige Bewegung, beschleunigte Bewegung, Sinus- und Cosinusfunktion, Pendelbewegung, Wurfparabel, Kreisfunktion, Gravitationssimulation und Vektorreflexionen. Im letzten Beispiel, einer Billard Simulation, werden viele Berechnungen aus vorhergehenden Beispielen vereint. Das Ziel der Arbeit ist die visuelle Darstellung dieser Funktion, um die Komplexität der dazugehörigen Algorithmen in C# begreifbar zu machen und vor allem Anfängern, die beginnen, sich mit der Materie auseinandersetzen, näher zu bringen. Für die eigentliche Programmierung werden C# Scripts in der Unity5 Engine implementiert, dargestellt und dokumentiert.

Um allen Inhalten dieser Arbeit folgen zu können, sind die Grundlagen der Programmierung (speziell objektorientierte Programmierung mit C#) sowie grundlegende Fertigkeiten im Umgang mit Unity5 erforderlich. Die naturwissenschaftlichen Grundlagen, die für die benutzten Algorithmen zur Anwendung kommen, werden im Kapitel zwei aufbereitet und werden großteils der mathematischen Formelsammlung von Lothar Papula entnommen. Die wichtigsten Funktionen, die notwendig sind, um die Algorithmik in Unity zu integrieren werden in Kapitel drei behandelt und stammen ausschließlich von der Unity Scripting Documentation.

Auf die Vergleiche von selbständig berechneten Bewegungen und der Verwendung von vorintegrierten Funktionen der Unity Physics Engine wird in Kapitel vier in mehreren Beispielen eingegangen. Zusätzlich werden elementare Grundfunktionen in Unity, die unterschiedlichen Einfluss auf die implementierte Algorithmik haben, näher erläutert und verglichen. Neben Screenshots wird für die Erklärung der Algorithmen auch der Consolen Output der Unity GUI (Graphical User Interface) verwendet. Die behandelten Unity Projekte beinhalten teilweise interaktive Funktionen, mit denen Parameter für den Bewegungsablauf während der Laufzeit verändert werden können. Diese Beispiele sind auf der beigelegten CD zu finden und wurden mit der Unity Personal Edition (Version: 2017.1.1f1) erstellt.

ABSTRACT

This thesis deals with the theory, implementation and visualization of algorithms for 2D movements. The visualization is displayed with the Unity5 Engine.

This thesis focuses the analysis of 2D algorithms. The visually presented results include movements such as uniform motion, acceleration, sine and cosine function, pendulum motion, throw parabola, circular function, gravitational stimulation and vector reflections. The last example, a billiard simulation, combines many movement calculations of previous examples. The goal of the thesis is the visual representation of these functions in order to make the complexity of the related algorithms understandable in C# and to help beginners who start with the subject matter. For the actual programming, C# scripts are implemented, displayed and documented in the Unity5 Engine.

In order to be able to follow all the content of this work, the basic principles of programming (especially object-oriented programming with C#) as well as basic skills in dealing with Unity5 are required. The natural sciences used for the algorithms are presented in chapter two, and are taken from Lothar Papula's mathematical formulary. The most important functions that are necessary to integrate the algorithms in Unity are covered in chapter three and come exclusively from Unity Scripting Documentation.

The comparisons of independently calculated movements and the use of pre-integrated functions of the Unity Physics Engine are discussed in chapter four in several examples. In addition, elementary basic functions in Unity, which have different influences on the implemented algorithms, are explained and compared. In addition to screenshots, the console output of the Unity GUI (Graphical User Interface) is also used for the explanation of the algorithms. The Unity projects are partially interactive, so parameters can be adapted to change the movement process during runtime. These examples can be found on the enclosed CD and have been created with the Unity Personal Edition (version: 2017.1.1f1).

1 EINLEITUNG

1.1 MOTIVATION

Bewegungsalgorithmen im zweidimensionalem Raum finden heutzutage in vielen Bereichen der Technik Anwendung. Angefangen von Computergrafiken und einfachen Animationen, bis hin zu komplexen Computerspielen. Durch die fortschreitende Technik und die Hilfe von fertigen Frameworks, wird es immer einfacher, diese Algorithmen zu implementieren. Trotzdem braucht es ein Grundverständnis an mathematischen Wissen und Programmierkenntnissen, um komplexere Bewegungen zu implementieren. Diese Arbeit schafft ein Grundverständnis, das notwendig ist, um 2D Bewegungsalgorithmen zu verstehen und implementieren zu können. Durch die Visualisierung dieser Algorithmen mit der Unity5 Engine wird deren Komplexität anschaulich dargestellt und dadurch leichter verständlich präsentiert.

1.2 ZIEL

Das Ziel der Arbeit ist die visuelle Darstellung von Algorithmen, um deren Komplexität greifbar zu machen und vor allem Anfängern, die sich mit der Materie auseinandersetzen, näher zu bringen. Allgemeine Programmierkenntnisse und die grundlegende Bedienung der Unity Oberfläche sind Voraussetzungen für das Verständnis dieser Arbeit und werden nicht extra behandelt. Für die Implementierung und Visualisierung der Algorithmen werden zum Vergleich fertige Physics Funktionen der Unity Engine verwendet. Das Schwerpunkt liegt jedoch auf der selbstständigen Berechnung der Bewegungen. Zur Darstellung der Funktionsgraphen werden Screenshots direkt aus dem jeweiligen Unity Projekt verwendet und zusammen mit Code Elementen der Algorithmik erklärt.

1.3 VORGEHENSWEISE

Diese Arbeit ist eine wissenschaftliche Arbeit, die auf Literatur Recherchen aus Büchern und Internetquellen bzw. Code Dokumentationen und Formelsammlungen basiert. Das zusammengetragene Wissen ist mit der selbsterarbeiteten Darstellung in Unity5 aufbereitet und zeigt den Zusammenhang zwischen naturwissenschaftlichen Grundlagen, Algorithmik und Darstellung für Bewegungen im zweidimensionalen Raum. Zu Beginn wird auf die naturwissenschaftlichen Grundlagen eingegangen, die notwendig sind, um die Implementierung der Algorithmen zu verstehen. Das nächste Kapitel beschäftigt sich mit der visuellen Darstellung in Unity5 und grundlegenden Funktionen und Klassen in C#, die das Grundgerüst zur Implementierung der verwendeten Algorithmen bilden. Im Hauptteil werden die Implementierung der Algorithmen und deren Darstellung aufgezeigt. Das letzte Kapitel befasst sich mit den Anwendungsgebieten für Bewegungsalgorithmen im zweidimensionalen Raum.

2 NATURWISSENSCHAFTLICHE GRUNDLAGEN

Dieses Kapitel beschäftigt sich mit den naturwissenschaftlichen Grundlagen und Formeln, die für das Themengebiet der Algorithmik in 2D und dem Verständnis der Unity Physics-Engine notwendig sind.

2.1 GLEICHFÖRMIGE BEWEGUNG

Die einfachste Variante einer gleichförmigen Bewegung ist die geradlinige Bewegung. Für diese Bewegung genügt eine einzelne Raumachse im Koordinatensystem. Bewegt sich ein Körper gleichförmig, ändert sich seine Position p abhängig vom zeitlichen Verlauf t mit $p(t)$. Befindet sich ein Körper im Ruhezustand, stellt das einen Spezialfall einer Bewegung mit konstanter Geschwindigkeit dar.

$$\nu = \frac{\Delta s}{\Delta t}$$

Um die Position p des Körpers angeben zu können, fehlt neben der Abhängigkeit der Zeit noch die Startposition p_0 und führt zu der Gleichung:

$$p(t) = p_0 + \nu_0 * \Delta t$$

([Grundwissen Physik], S. 14)

2.2 KONSTANTE BESCHLEUNIGUNG

Bewegt sich ein Objekt mit konstanter Geschwindigkeit, legt es in gleichen Zeitabschnitten die jeweils gleiche Wegstrecke zurück.

Somit ergibt sich die Geschwindigkeit ν als Verhältnis der zurückgelegten Wegstrecke s und der dazu benötigten Zeit t :

$$\nu = \frac{s}{t}$$

Bewegt sich ein Objekt mit konstanter Beschleunigung, nimmt dessen Geschwindigkeit konstant, in gleichen Zeitabschnitten um den gleichen Betrag zu.

Die Formel für die Beschleunigung a ist das Verhältnis aus der Geschwindigkeitsänderung $\Delta\nu$ und dem dazu benötigten Zeitabstand Δt :

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

([Grundwissen Physik], S. 21)

2.3 GRAVITATION

Gravitation beschreibt die Anziehungskraft zweier Massen zueinander, die über zunehmende Entfernung abnimmt. In der Physik wird diese Kraft mit dem newtonschen Gravitationsgesetz beschrieben. Aus der allgemeinen Relativitätstheorie nach Albert Einstein ergibt sich ein anderes Verständnis, auf das in dieser Arbeit nicht weiter eingegangen wird.

Die klassische Mechanik beschreibt die Gravitation als eine Eigenschaft aller Materie, die alleine abhängig von deren Masse ist und nicht von deren Art oder Bewegung. Jede Masse erzeugt eine Gravitationskraft bzw. ein Gravitationsfeld, das auf jede andere Masse in Form einer Anziehungskraft wirkt.

Isaac Newton beschreibt Gravitation in dem von ihm formulierten Gravitationsgesetz als eine Kraft, die zwischen zwei Körpern, welche zu deren gemeinsamen Schwerpunkt hin beschleunigt, die proportional zum Quadrat des Abstandes der Körper abnimmt. Dieses Gesetz liefert ein grundlegendes Verständnis der Dynamik des Sonnensystems und der Bewegungen von Planeten.

Das newtonsche Gravitationsgesetz gibt die momentane Kraft F_G an, mit der zwei punktförmig gedachte Körper mit den Massen m_1 und m_2 im Abstand r mit Berücksichtigung der Gravitationskonstante G einander anziehen.

$$F_G = G * \frac{m_1 * m_2}{r} .$$

Diese Kraft wirkt für beide Massen gleich stark, ist jedoch für jede Masse in Richtung der A anderen gerichtet. Die Beschleunigung lässt sich nach dem zweiten newtonschen Gesetz $F = m * A$ für m_1 mit

$$a_1 = \frac{F_1}{m_1} = G * \frac{m_2}{r^2}$$

und für m_2 mit

$$a_2 = \frac{F_2}{m_2} = G * \frac{m_1}{r^2}$$

beschreiben. Die Beschleunigung eines Körpers hängt demnach also nicht von seiner Masse, sondern der Masse des anderen Körpers ab.

(vgl. [<https://de.wikipedia.org/wiki/Gravitation>], 12.09.2017)

2.4 EULER - CROMER METHODE

In Kapitel 4 dieser Arbeit kommt ein spezieller Algorithmus vor, der in einer Pendel-Bewegung simuliert wird. Aus der Taylor-Entwicklung von Geschwindigkeit und Ort ergibt sich eine Anwendung des Euler-Verfahrens, bei dem in jedem Zeitintervall folgende Berechnungen durchgeführt werden:

$$v_{n+1} = v_n + a_n \Delta t$$

$$r_{n+1} = r_n + v_n \Delta t$$

Als Alternative ergibt sich die Euler-Cromer Methode, um die Geschwindigkeit für jeden neuen Zeitpunkt zuerst zu berechnen. Bei dieser Methode wird zur Berechnung der neuen Position nicht die Geschwindigkeit zum alten Zeitpunkt t_n sondern t_{n+1} verwendet und ergibt:

$$v_{n+1} = v_n + a_n \Delta t$$

$$r_{n+1} = r_n + v_{n+1} \Delta t$$

([Computational Physics 2013], S. 13)

2.5 PUNKTE

Ein Punkt beschreibt einen Ort im Koordinatensystem. Ein 2-dimensionaler Punkt hat 2 Koordinatenwerte, die die Position auf der jeweiligen Achse beschreiben. So gibt es im 2D Raum für einen Punkt einen X-Wert und einen Y-Wert, die wie folgt beschrieben werden können:

$$P = \begin{pmatrix} x \\ y \end{pmatrix}$$

Im 3D Raum kommt dem ein zusätzlicher Z-Wert hinzu:

$$P = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

([Papula], S. 41)

2.6 VEKTOREN

Vektoren haben in der Mathematik eine ähnliche Schreibweise wie Punkte, mit der Ausnahme, dass sie mit einem kleinen Buchstaben mit Pfeil darüber gekennzeichnet werden:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

Ein Vektor dieser Form gibt eine Richtung bzw. Verschiebung im 2D Raum an. Anders als der Punkt hat der Vektor keinen Ausgangspunkt, sondern kann sich im Raum als Strahl vorgestellt werden. Ein Vektor kann aus zwei Punkten aufgestellt werden in dem man eine Linie zeichnet, auf der beide Punkte liegen. So wird der Vektor für die Punkte $A = \begin{pmatrix} A_x \\ A_y \end{pmatrix}$ und $B = \begin{pmatrix} B_x \\ B_y \end{pmatrix}$ wie folgt berechnet:

$$\overrightarrow{AB} = B - A = \begin{pmatrix} B_x - A_x \\ B_y - A_y \end{pmatrix}$$

([Papula], S. 50)

Da ein Vektor lediglich eine Richtung angibt, gibt es im Raum mehrere Möglichkeiten, diese anzuschreiben. So hat zum Beispiel der Vektor

$$\vec{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

die gleiche Richtung wie der Vektor

$$\vec{y} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

und ist somit ein Parallelvektor:

$$\vec{x} \parallel \vec{y}$$

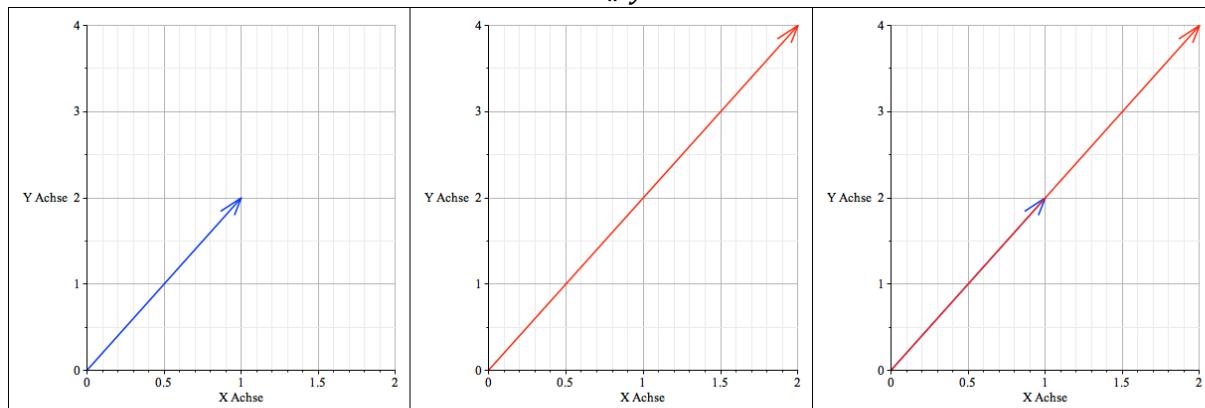


Abbildung 1 – Parallelvektoren, x in Blau, y in Rot

[Screenshot: Erarbeitet mit Maple]

2.6.1 Rechenoperationen

Betrag

Ein Vektor hat neben der Richtung die Eigenschaft des Betrages. Der Betrag ist die tatsächliche Länge des Vektors und kann für den Vektor $\vec{x} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ wie folgt berechnet werden:

$$|\vec{x}| = \sqrt{1^2 + 2^2} = \sqrt{5}$$

Die Berechnung beruht auf dem Satz des Pythagoras da jeder Vektor in 2D mit einem Dreieck dargestellt werden kann.

([Papula], S. 49)

Multiplikation mit einer Zahl

Man kann einen Vektor mit einem Skalar multiplizieren und so den Betrag des Vektors vervielfachen. Diese Operation hat jedoch keinerlei Einfluss auf die Richtung des Vektors. So ergibt

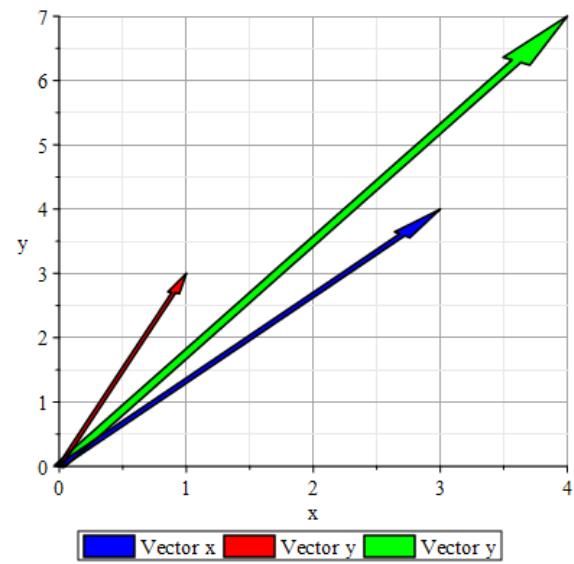
$$2 * \vec{x} = 2 * \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix} = \vec{y}$$

Damit wird bewiesen, dass die Multiplikation zu keiner Richtungsänderung führt, da $\vec{x} \parallel \vec{y}$.

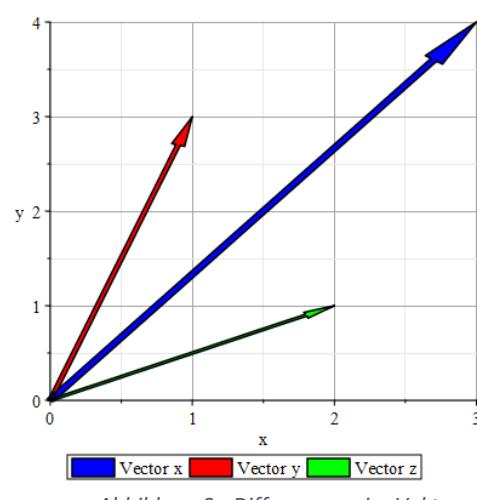
([Papula], S. 51)

Summe und Differenz

Die Summe zweier Vektoren ergibt die Strecke, die in einem neuen Vektor resultiert, wenn sie aufeinandergelegt werden. In Abbildung 2 wird der Vektor $\vec{x} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$ (blau) mit $\vec{y} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$ (rot) addiert und resultiert in $\vec{z} = \vec{x} + \vec{y} = \begin{pmatrix} 4 \\ 7 \end{pmatrix}$ (grün). Wird \vec{y} visuell auf die Spitze von \vec{x} gesetzt endet \vec{y} genau an der Spitze von \vec{z} und umgekehrt.



Die Differenz stellt den Verbindungsvektor der beiden Vektorspitzen dar und ist je nachdem ausgerichtet, welcher Vektor dem anderen abgezogen wird. Laut Abbildung 3 entsteht der Differenzvektor mit $\vec{z} = \vec{x} - \vec{y} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$ (grün).



([Papula], S. 50)

Kreuzprodukt

Üblicherweise wird das Kreuzprodukt zweier Vektoren im 3D Raum berechnet. Das Ergebnis stellt einen neuen Vektor dar, der senkrecht auf die Ebene der beiden Stammvektoren steht und stellt damit optisch eine Erweiterung in den 3D Raum her. Im 2D Raum kann das Kreuzprodukt über die Determinante berechnet werden, deren Ergebnis die dritte Komponente des entstehenden 3D Vektors darstellt. Die anderen beiden Komponenten sind logischerweise 0.

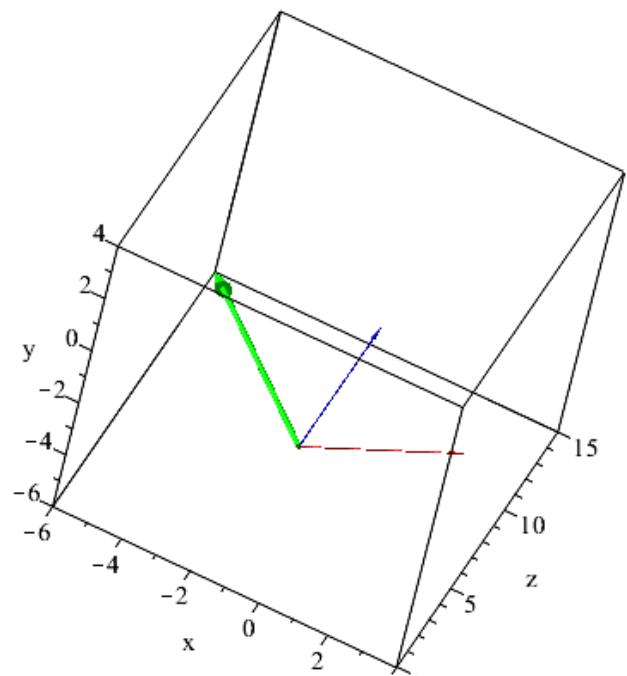


Abbildung 4 - Kreuzprodukt zweier Vektoren

$$\vec{x} \times \vec{y} = \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} = x_1 * y_2 - x_2 * y_1$$

([Papula], S. 53)

Skalarprodukt

Eine weitere wichtige Rechenoperation von Vektoren ist das Skalarprodukt. Dabei werden jeweils die ersten Komponenten der beiden Vektoren multipliziert und danach mit dem Produkt der zweiten Komponenten addiert. Das Ergebnis ist eine Zahl, die Auskunft darüber gibt, wie die beiden Vektoren im 2D Raum zueinanderstehen. Ist die Zahl > 0 , stehen sie in einem spitzen Winkel zueinander, ist sie < 0 in einem stumpfen Winkel, ist sie gleich 0, stehen sie rechtwinklig (orthogonal) zueinander. Berechnet man das Skalarprodukt für $\vec{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ und $\vec{y} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, die jeweils die X - und Y - Achse darstellen und somit orthogonal zueinanderstehen, erhält man den Wert $v = \vec{x} \cdot \vec{y} = 1 * 0 + 0 * 1 = 0$.

([Papula], S. 51)

Winkel

Der Winkel zwischen zwei Vektoren kann mit einer Umformung des Kosinussatzes berechnet werden:

$$\cos(\varphi) = \frac{\vec{x} * \vec{y}}{|\vec{x}| * |\vec{y}|}$$

([Papula], S. 52)

2.6.2 Geraden

Geradengleichung

Eine Gerade ist eine Linie, die keinen fixen Anfangspunkt bzw. Endpunkt besitzt. Man kann die Geradengleichung über zwei Punkte, oder über einen Punkt und einen Richtungsvektor aufstellen. Wird eine Gerade über zwei Punkte im Raum aufgestellt, ergibt sich die Geradengleichung mit

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

Für den Parameter t kann ein beliebiger Wert eingesetzt werden, um einen Punkt auf der Geraden zu ermitteln. Wird die Geradengleichung mit einem bekannten Punkt P auf der Geraden und einem bekannten Richtungsvektor \vec{x} aufgestellt, ergibt sich analog zu der Formel mit zwei Punkten

$$g : X = P + t * \vec{x}$$

([Papula], S. 57)

Abstand eines Punktes zu einer Geraden

Um den Abstand d eines Punktes Q zur Geraden

$$g : X = P + t * \vec{x}$$

im Raum zu ermitteln wird das Kreuzprodukt zweier Vektoren verwendet

$$d = \frac{|(Q - P) \times \vec{x}|}{|\vec{x}|}$$

Die Differenz der Beträge des Kreuzproduktes und des Richtungsvektors der Geraden ergeben den Wert des Abstandes d .

([Papula], S. 58)

Reflexion

Der Reflexionsvektor, der sich beim Aufprall eines Objektes an einem anderen ergibt, kann mit

$$r = d - 2 * (d \cdot n) * n$$

berechnet werden.

(<https://math.stackexchange.com/questions/13261/how-to-get-a-reflection-vector;>
25.09.2017)

2.7 KOORDINATENSYSTEME

2.7.1 Kartesisches Koordinatensystem

Das kartesische Koordinatensystem im zweidimensionalen Raum beschreibt Punkte und Vektoren mit zwei senkrecht aufeinander stehenden Achsen. Dabei wird der Punkt, an dem sich die Achsen schneiden als Ursprung oder Nullpunkt bezeichnet. So setzt sich ein Punkt

$P = \begin{pmatrix} x \\ y \end{pmatrix}$ aus einem Wert für die Verschiebung auf der X-Achse und einem auf der Y-Achse (Abbildung 5).

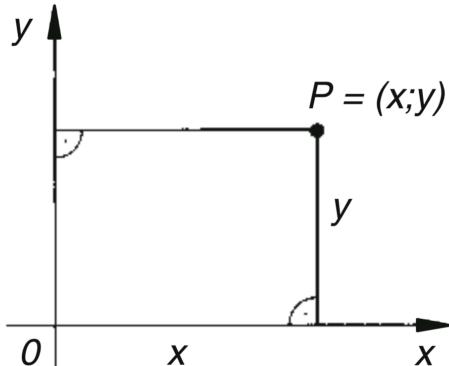


Abbildung 5 - Kartesisches Koordinatensystem

2.7.2 Polarkoordinatensystem

Ein Punkt als Polarkoordinate wird durch eine Abstandskoordinate r und eine Winkelkoordinate φ beschrieben (Abbildung 6). Der Ursprung O wird als Pol bezeichnet und der Winkel bei einer Drehung im Uhrzeigersinn negativ, bei einer Drehung gegen den Uhrzeigersinn positiv gemessen.

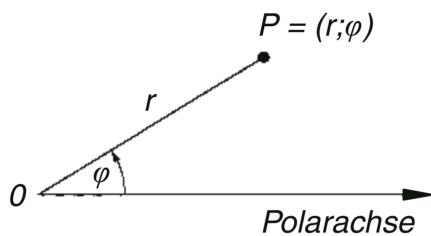


Abbildung 6 – Polarkoordinaten

([Papula], S. 41-42)

2.8 PARAMETERDARSTELLUNG EINES KREISES

Die X- und Y-Werte eines Kreises können durch die Parameterdarstellung eines Kreises mit dem Radius r und dem Winkel φ mit

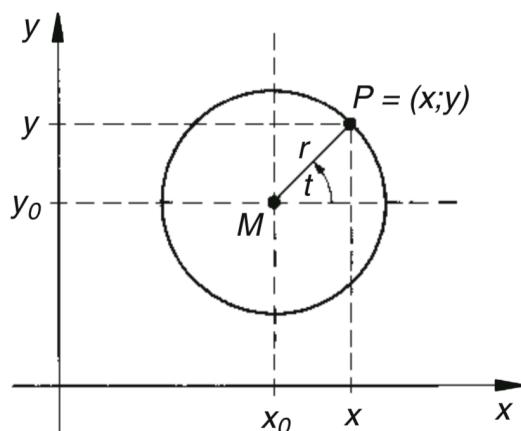
$$x = x_0 + r * \cos(\varphi)$$

und

$$y = y_0 + r * \sin(\varphi)$$

berechnet werden.

([Papula], S. 115)



2.9 MATRIZEN

Eine Matrix ist ein mehrdimensionaler Vektor. Durch Matrizen können lineare Abhängigkeiten mehrerer Variablen ausgedrückt werden. Sie können Spiegelungen, Drehungen und Projektionen beschreiben. Eine 2×2 Matrix A kann folgendermaßen aufgestellt werden:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Eine der wichtigsten Formen der Matrizen ist die Einheitsmatrix:

$$E = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Man kann diese mit der 1 bei einer Multiplikation mit einfachen Zahlen vergleichen. Wird also eine Matrix A mit der Einheitsmatrix E multipliziert ergibt sich die gleiche Matrix A .

([Papula], S. 194)

Multiplikation

Bei einer Multiplikation mit einer Zahl werden alle Komponenten der Matrix mit der Zahl multipliziert. Durch die Multiplikation zweier Matrizen derselben Dimension entsteht eine gleichdimensionale Matrix:

$$A * B = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} * B_{11} + A_{12} * B_{21} & A_{11} * B_{12} + A_{12} * B_{22} \\ A_{21} * B_{11} + A_{22} * B_{21} & A_{21} * B_{12} + A_{22} * B_{22} \end{bmatrix}$$

Bei der Multiplikation mit einem Vektor entsteht ein neuer Vektor:

$$A * \vec{x} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} A_{11} * x_1 + A_{12} * x_2 \\ A_{21} * x_1 + A_{22} * x_2 \end{pmatrix}$$

([Papula], S. 199)

Inverse Matrix

Die Inverse einer Matrix $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ wird mit $A^{-1} = \begin{bmatrix} A_{22} & -A_{12} \\ -A_{21} & A_{11} \end{bmatrix}$ gebildet und es gilt:
 $A * A^{-1} = 1$

([Papula], S. 201)

Determinante

Die Determinante einer Matrix wird mit

$$D_A = \begin{vmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{vmatrix} = D_{11} * D_{22} - D_{12} * D_{21}$$

gebildet und gibt an, ob es möglich ist sie zu invertieren. Solange $D_A \neq 0$ kann die Matrix A invertiert werden.

([Papula], S. 205)

Transponierte Matrix

Für manche Berechnungen kann es notwendig sein, eine Matrix zu transponieren. Dabei werden die Spalten mit den Reihen der Matrix $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ vertauscht:

$$A^T = \begin{bmatrix} A_{11} & A_{21} \\ A_{12} & A_{22} \end{bmatrix}$$

Die Determinante D_{A^T} einer transponierten Matrix ist gleich der ursprünglichen Determinante D_A .

([Papula], S. 197)

Verschiebung

Mit Matrizen können beispielsweise Punkte verschoben werden. Dazu ist aber eine Erweiterung des Vektors und der Verschiebungsmatrix in den 3D Raum notwendig.

Der Punkt $P = \begin{pmatrix} 5 \\ 4 \end{pmatrix}$ wird dabei um eine Zeile erweitert. Diese ist für diesen Fall immer 1.

$$\vec{y} = \begin{pmatrix} 5 \\ 4 \\ 1 \end{pmatrix}$$

Möchte man laut Abbildung 5 P um den Wert $k = 4$ parallel zur X-Achse verschieben, muss die Verschiebungsmatrix

$$A_V = \begin{bmatrix} 1 & 0 & k \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

aufgestellt werden. Diese wird dann mit \vec{y} multipliziert und ergibt mit den ersten beiden Zeilen den verschobenen Punkt.

(Rudolf Rüdiger; Skript Mathe2, IMA 2015)

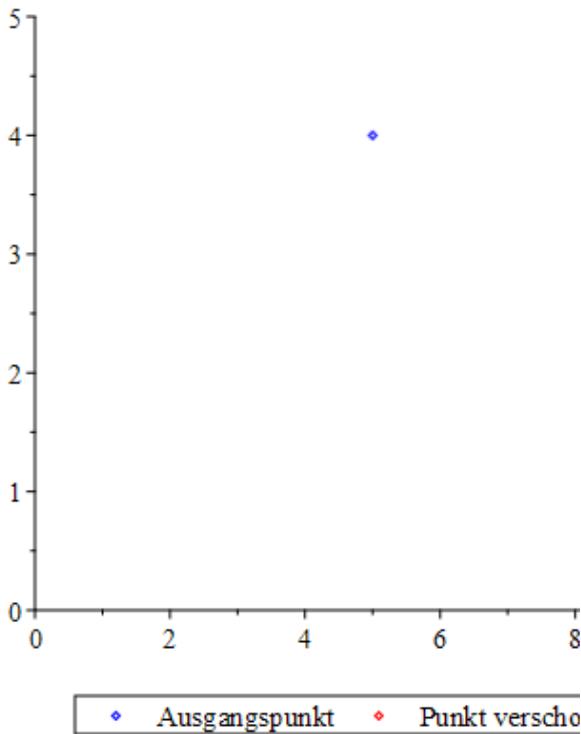


Abbildung 7 - Verschieben eines Punktes

Eine Verschiebung um einen Vektor $\vec{z} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix}$ wird mit einer Multiplikation der Matrix

$$B_V = \begin{bmatrix} 1 & 0 & z_1 \\ 0 & 1 & z_2 \\ 0 & 0 & 1 \end{bmatrix}$$

erreicht.

Drehung

Jeder Punkt kann mit der Rotationsmatrix

$$A_R = \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) & 0 \\ \sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

um den Winkel φ um den Ursprung gedreht werden. Dazu wird der Punkt $P = \begin{pmatrix} 3 \\ 4 \\ 1 \end{pmatrix}$, der rotiert werden soll wieder um eine Zeile erweitert und mit der Rotationsmatrix multipliziert.

$$P_R = \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) & 0 \\ \sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} 3 \\ 4 \\ 1 \end{pmatrix}$$

In Abbildung 8 wird der Punkt $P = \begin{pmatrix} 3 \\ 4 \\ 1 \end{pmatrix}$ (blau) um 30° gegen den Uhrzeigersinn um den Ursprung gedreht und ergibt den Punkt $Q = \begin{pmatrix} \sim 0.5 \\ \sim 5 \\ 1 \end{pmatrix}$.

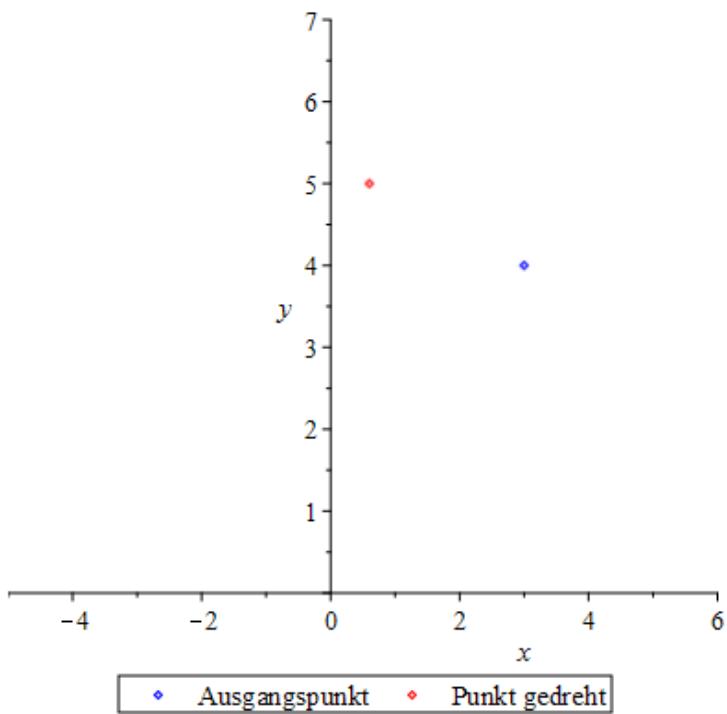


Abbildung 8 - Drehung eines Punktes

Soll ein Punkt $P = \begin{pmatrix} 6 \\ 4 \end{pmatrix}$ um einen anderen Punkt $S = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$ gedreht werden, muss der P zuerst mit einer Verschiebungsmatrix - mit der Negativverschiebung von S - zum Ursprung gesetzt werden, dann mit der Rotationsmatrix A_R gedreht und anschließend mit einer Positivverschiebung von S wieder zurückverschoben werden.

$$\begin{bmatrix} 1 & 0 & -3 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} 6 \\ 4 \\ 1 \end{pmatrix} \cdot \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) & 0 \\ \sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix}$$

Dabei ist die Reihenfolge der Berechnungen sehr wichtig, da eine falsche Abfolge der Rechenoperationen zu einem falschen Ergebnis führt.

Spiegelung

Um beispielsweise einen Punkt $P = \begin{pmatrix} 6 \\ 4 \end{pmatrix}$ am Ursprung zu spiegeln ist folgende Spiegelmatrix notwendig:

$$M_S = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Die Spiegelung erfolgt durch das Skalarprodukt der Spiegelmatrix und des Punktes.

$$M_S \cdot P = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} 6 \\ 4 \\ 1 \end{pmatrix} = \begin{pmatrix} -6 \\ -4 \\ 1 \end{pmatrix}$$

(Rudolf Rüdiger; Skript Mathe2, IMA 2015)

Möchte man den Punkt P an einer Geraden $g: x = 4$ im 2D Raum spiegeln muss man eine Matrix erstellen, die den Punkt zuerst abhängig von der Geradengleichung in den Ursprung verschiebt, ihn dann am Ursprung spiegeln und letztendlich wieder zurückverschieben. Die Verschiebungsmatrix wird mit

$$M_V = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -4 \\ 0 & 0 & 1 \end{bmatrix}$$

aufgestellt. Danach kann die Matrix zur Spiegelung an der X-Achse aufgestellt werden:

$$M_{S_x} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Die Transformationsmatrix ergibt sich also mit:

$$M_T = \frac{1}{M_V} \cdot M_S \cdot M_V$$

Der fertig gespiegelte Punkt P_S mit:

$$P_S = M_T \cdot P$$

In Abbildung 9 werden die Punkte $P_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$, $P_2 = \begin{pmatrix} 5 \\ 1 \end{pmatrix}$ und $P_3 = \begin{pmatrix} 3 \\ 8 \end{pmatrix}$, die die Eckpunkte des roten Dreiecks ergeben auf der Geraden $g : x = 8$ (blau) gespiegelt. Als Ergebnis entstehen die drei gespiegelten Eckpunkte, die das grüne Dreieck darstellen.

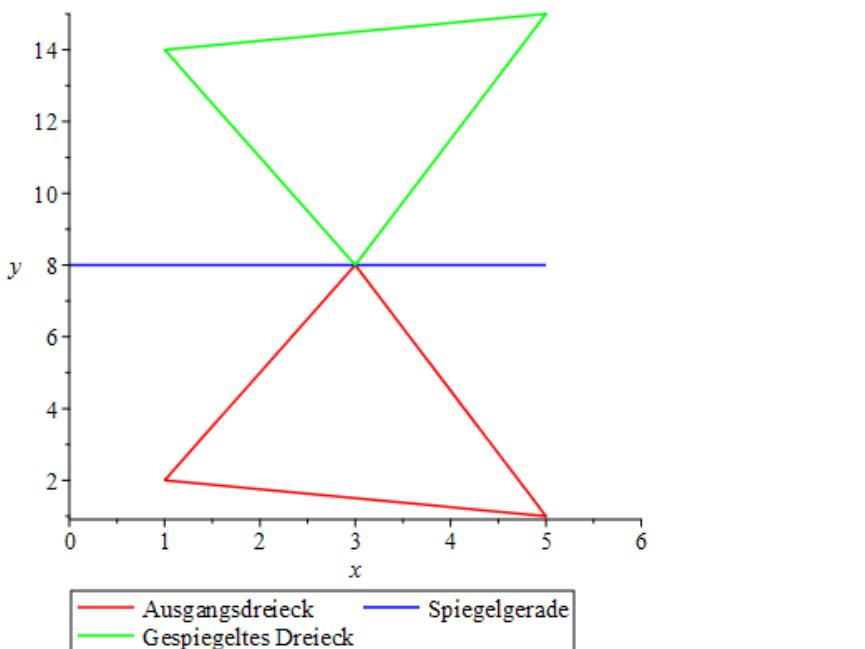


Abbildung 9 - Spiegelung von Punkten über eine Gerade

3 GRUNDLAGEN UNITY5 UND C#

Viele Inhalte dieser Arbeit basieren auf einem starken Bezug zur Unity Game Engine, die eine Vielfalt an Werkzeugen mitliefert, um verschiedenste 2D- und 3D-Anwendungen zu entwickeln. Neben einer eigenen Physik-Engine gibt es auch Build-in-Tools für Partikeleffekte, zur Terraingestaltung und einem eigenen Plug-In für Animationen. Im Vergleich zu anderen Game Engines wie der Unreal Engine, Frostbite, und CryEngine besitzt Unity weniger Features, ist aber leicht zu erlernen und in einer offiziellen Personal Edition kostenfrei verfügbar. [Paul E. Dickson, Jeremy E. Block, Gina N. Echevarria, and Kristina C. Keenan, (2017)]

Neben der Hauptanwendung wird mit Unity5, die speziell dafür angepasste Entwicklungsumgebung MonoDevelop mitgeliefert. Mit dieser Entwicklungsumgebung (Integrated Development Environment) kann die eigentliche Programmierung und das Debugging durchgeführt werden. Neben MonoDevelop gibt es viele andere Möglichkeiten die Programmierung umzusetzen, angefangen von einfachen Text Editoren, mit denen man jedoch kein Debugging durchführen kann, bis zu ausgefeilten IDEs wie Microsoft Visual Studio. [Lubomir Ivanov (2015)]

Obwohl in Unity5 einige Grundpakete für 3D-Objekte von Haus aus mitgeliefert werden, gibt es jedoch keine eigene 3D-Modellierungssoftware. Bei Bedarf muss dafür eine Zusatzsoftware verwendet werden, wie zum Beispiel die kostenlose Software Blender.

Alle erarbeiteten Inhalte dieser Arbeit wurden mit der kostenlosen Personal Edition von Unity implementiert. Neben dieser gibt es eine kostenpflichtige Version mit der Option eines einmaligen Kaufes oder einer monatlichen Miete. Beide Versionen unterstützen aktuell standardmäßig Windows Desktop Programme, OS X, Linux, Windows Store Apps, iOS, Android, Windows Phone 8, Blackberry 10, Google Native Client sowie einen eigenen Webplayer als Publishing Format. Die wichtigsten Unterschiede zur kostenpflichtigen Version Unity Professional beziehen sich auf zusätzliche Funktionen wie z. B. der Cloud-Unterstützung für die Softwareentwicklung und Kollaboration.

Die Entwickler von Unity arbeiten derzeit an verschiedenen Integrationen für Virtual Reality, die in der aktuellen Version 5 von Unity noch nicht integriert sind. Vermutlich werden diese Features zunächst nur in der kostenpflichtigen Pro Version ausgerollt und später in die Personal Edition übernommen. ([Seifert 2014], S. 35)

3.1 PROGRAMMIERUNG MIT C#

C# ist eine objektorientierte Programmiersprache, die von Aufbau und Syntax Java sehr ähnlich ist. Ursprünglich wurde sie entwickelt, um Anwendungen auf Basis des .NET Framework zu entwickeln, welches eine große Palette an Bibliotheken, Programmierinterfaces und Dienstprogrammen als Plattform zur Verfügung stellt. Unity jedoch nutzt C# in Kombination mit einer Open-Source Variante des .NET Frameworks. Dadurch können C#-Anwendungen auch auf anderen Systemen als Microsoft betrieben werden. ([Seifert 2014], S. 35)

Die wichtigsten Basisfunktionen, die für ein C# Skript in Unity notwendig sind, sind folgende vier:

Start

Jedes C# Skript, das über das UI von Unity erstellt wird, besteht aus einer Klasse, mit dem Namen des Skripts, die von der Unity Basis Klasse *MonoBehaviour* erbt und damit über alle Basisfunktionen der Unity Engine verfügt. In der erstellten Klasse befinden sich standardmäßig 2 leere void Funktion. Die *MonoBehaviour.Start()* Funktion und *MonoBehaviour.Update()*. Die *Start* Funktion wird beim ersten Frame, nach dem Start des Skripts, genau einmal aufgerufen, bevor eine der Update Methoden aufgerufen wird. Sie wird meistens zum Initialisieren verwendet.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class newCSharpScript : MonoBehaviour
6  {
7      // Use this for initialization
8      void Start () {
9
10     }
11
12     // Update is called once per frame
13     void Update () {
14
15     }
16 }
```

Code 1 - Default C# Skript aus Unity

Update

Die *Update* Funktion wird mit jedem neuen Frame aufgerufen und für die meisten Programmieranwendungen verwendet.

FixedUpdate

Anders als das normale *Update* ist die Wiederholungsrate in *FixedUpdate* nicht von der *Framerate* abhängig, sondern von der fixen *physics* Wiederholungsrate, die synchron und gleichmäßig aufgerufen wird.

Time

Die Time Klasse wird in Unity verwendet, um von der Zeit abhängige Berechnungen und Funktionsaufrufe durchzuführen. Mit `Time.deltaTime` kann die Zeit, die seit dem letzten Frame vergangen ist, gelesen werden. Dadurch kann sichergestellt werden, dass sich beispielsweise ein Objekt nicht mit 5 Einheiten pro Frame, sondern mit 5 Einheiten pro Sekunde fortbewegt. (vgl. [<https://docs.unity3d.com/ScriptReference/>] [1])

3.2 DARSTELLUNG IM 2D KOORDINATENSYSTEM

Unity ist grundsätzlich für Arbeiten im 2D und 3D Raum ausgelegt. Es kann zwischen 2D und 3D Ansicht gewechselt werden. Unabhängig davon, in welcher Ansicht man sich befindet können jeweils 2D oder 3D Objekte in die Szene eingebaut werden. Unity nutzt zur Darstellung von Objekten im 3D Raum das linkshändige Koordinatensystem. Dabei stellen drei Finger der linken Hand die X, Y und Z – Achse dar. Der Daumen wird waagrecht nach rechts weggestreckt und stellt so die X-Achse dar. Der Zeigefinger zeigt nach oben und symbolisiert die Y-Achse. Damit ist bereits der zweidimensionale Bereich realisiert. Wird nun der Mittelfinger gerade nach vorne abgespreizt zeigt dieser in den Raum und stellt so die Z-Achse dar.

([Seifert 2014], S. 107)

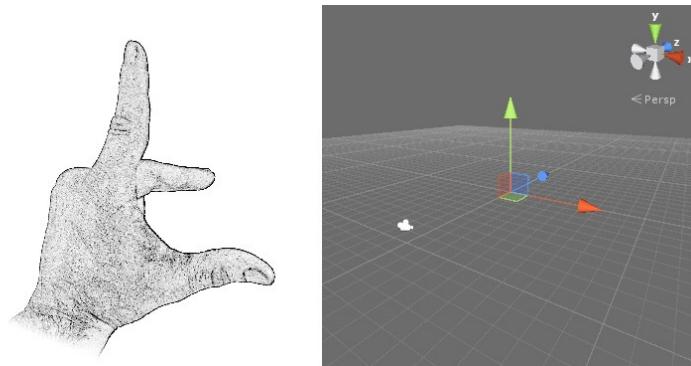


Abbildung 10 - Linkshändiges Koordinatensystem

([Seifert 2014], S. 107)

Für die Darstellung im zweidimensionalen Raum wird von Unity die Z-Achse „deaktiviert“, das heißt, es sind nur noch X- und Y-Achse als Achsen sichtbar. Die Z-Achse wird zu einem Punkt (Abbildung 11, blau).

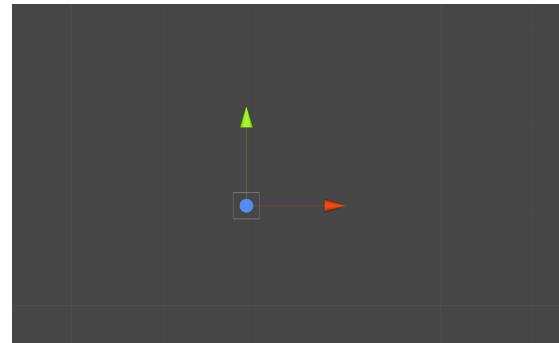


Abbildung 11 - Unity: 2D Koordinatensystem

3.2.1 Vektoren

Vektoren werden benutzt, um Punkte und Richtungen im 2D oder 3D Raum zu beschreiben. Ein Punkt beschreibt eine bestimmte Stelle im Koordinatensystem und ist selbst dimensionslos. In Unity gibt es dafür jeweils die Typen *Vector2* und *Vector3*. Ein *Vector2* besitzt Attribute für X und Y, ein *Vector3* für X, Y und Z. Ein *Vector2* wird wie folgt initialisiert:

```

1 //Initialisieren mit Position
2 Vector2 punkt1 = new Vector2(2, 3);
3 //Initialisieren ohne Position
4 Vector2 punkt2 = new Vector2();
5 //Zuweisen der Parameter für X und Y
6 point1.x = 2;
7 point1.y = 3;
```

Code 2 - Initialisieren eines Vector2 Typs

In Unity kann man über einen Button im UI zwischen 3D und 2D Ansicht wechseln, was für die 2D Darstellung optimal ist. Für 2D Darstellungen wurde eine Szene mit einer gekachelten Plane erstellt. Eine Kachel in Abbildung 12 entspricht dem Wert 1 im Koordinatensystem mit der Unity-Längeneinheit. Der Ursprung liegt am linken unteren Ende des Rasters. Für den Punkt $P = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$ wurde ein *Sprite* mit Kreisform gewählt, das den Punkt darstellt und zwei langgezogene Sphären für die X und Y – Achse verwendet. ([Seifert 2014], S. 108)

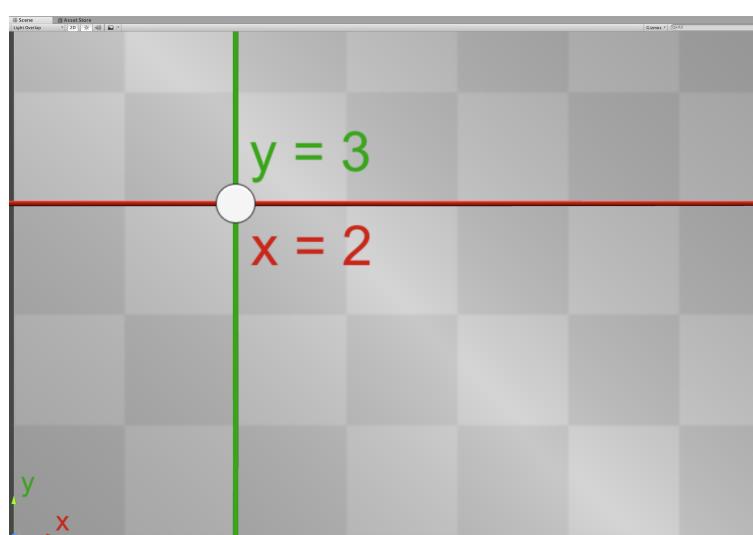


Abbildung 12 - Unity: Punkt in 2D Darstellung

Für die 3D Darstellung in Unity wird ein ähnliches Prefab verwendet, dass aus einer Sphäre besteht die den Punkt darstellt und drei langgezogenen, farbigen Sphären, die die jeweiligen Achsen darstellen um visuell einen besseren Eindruck für die Entfernung zu schaffen. Weiters wurde auf den Ebenen für die Begrenzung der XY, XZ und ZY ebenfalls ein Kachelraster eingeführt.

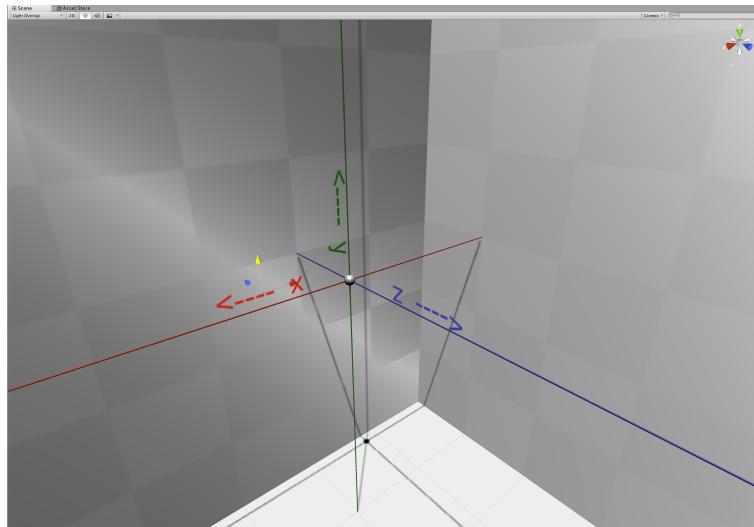


Abbildung 13 – Unity: Punkt im 3D Raum

Hierfür findet nun der `Vector3` Typ Anwendung. Mit der Angabe `(2, 3, 1)` wird die Position des Punktes im Raum beschrieben. Neben der Position kann mit einem Vektor in Unity auch eine Richtung mit einem Betrag dargestellt werden. Der Betrag ist die Länge des Vektors vom Ursprung `(0, 0)` bis zu den Koordinaten des Punktes und wird in `Vector2` und `Vector3` als Attribut *magnitude* bereitgestellt. Dieser wird über den Satz des Pythagoras berechnet. Diese Methode zur Berechnung der Länge ist aber durch das Wurzelziehen sehr rechenintensiv. Dafür gibt es das Attribut *sqrMagnitude*, das den quadrierten Betrag liefert.

Um beispielsweise den Betrag des Vektors vom Ursprung bis zum Punkt $P = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$ in Abbildung 12 zu berechnen gibt es mehrere Möglichkeiten. Berechnet man die Distanz mit *magnitude* ergibt sich:

```

1 void Start ()
2 {
3     float distance = new Vector2(2,3).magnitude;
4     print(distance);
5 }
```

Code 3 - Berechnen der Länge eines Vektors mit *magnitude*

(vgl. [<https://docs.unity3d.com/ScriptReference/>] [2])

Oder man berechnet sie mit der Funktion *Distance* die ebenfalls in *Vector2* vorimplementiert ist. Das Ergebnis beider Berechnungen ist *distance*~3.74.

```
1 void Start ()
2 {
3     float distance = Vector2.Distance(new Vector2(2,3), new Vector2(0,0));
4     print(distance);
5 }
```

Code 4 - Berechnung der Distanz zweier Punkte

(vgl [<https://docs.unity3d.com/ScriptReference/>] [3])

Richtung

Um einen Richtungsvektor von der Position *A* als *Vector2* zu einem Zielobjekt mit der Position *B* zu ermitteln, kann in Unity die Vektor Subtraktion verwendet werden mit

$$\overrightarrow{AB} = \vec{B} - \vec{A}.$$

Die Berechnung in C# ergibt sich beispielsweise mit

```
1 Vector3 directionToTarget = target.transform.position - this.transform.position;
```

Code 5 - Berechnen des Richtungsvektors zweier Punkte

Die Distanz zwischen den Objekten kann über den Betrag mit *magnitude* ermittelt werden:

```
1 Vector3 directionToTarget = target.transform.position - this.transform.position;
2 Vector3 distanceToTarget = directionToTarget.magnitude;
```

Code 6 - Distanz zwischen zwei Punkten mittels magnitude

Normalisieren

Durch das Normalisieren eines Vektors kann man die Vektorlänge auf die Länge 1 kürzen, was verwendet werden kann, wenn nur die Richtung des Vektors relevant ist, da diese erhalten bleibt.

Um einfache Vektorrichtungen zu verwenden, stellt Unity in der *Vector3* Klasse die normalisierten Attribute *forward*, *back*, *up*, *down*, *left*, *right*, *one*, und *zero* zur Verfügung. Das Attribut *up* beispielsweise stellt nichts anderes als den Vektor

$$\vec{V}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

dar. Das heißt der y-Wert ist 1, der in der Normalansicht die Achse darstellt, die nach oben zeigt. Um einen *Vector2* zu normalisieren, kann das Attribut *normalized* verwendet werden.

```
1 Vector2 directionToTarget = target.transform.position - this.transform.position;
2 Vector2 directionNormalized = directionToTarget.normalized;
```

Code 7 - Normalisieren eines Vector2

(vgl. [<https://docs.unity3d.com/ScriptReference/>] [4])

Geraden

Um Geraden in Unity darzustellen, kann die Klasse *LineRenderer* verwendet werden. Um eine Gerade (Abbildung 14, in Magenta) vom Ursprung zum Punkt $P = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$ mit LineRenderer darzustellen kann man folgendes Skript verwenden:

```

1 void Start ()
2 {
3     LineRenderer lr = gameObject.AddComponent<LineRenderer>();
4     lr.widthMultiplier = 0.1f;
5     lr.material = Resources.Load("Blue.mat", typeof(Material)) as Material;
6     lr.SetColors(Color.black, Color.blue);
7     lr.SetPosition(0, new Vector2(0,0));
8     lr.SetPosition(1, new Vector2(2,3));
9 }
```

Code 8 - Erstellen einer Gerade mit LineRenderer

Dabei wird zweimal die Funktion *SetPosition* von *LineRenderer* aufgerufen, um den Startpunkt und den Endpunkt der Geraden zu definieren.

(vgl. [<https://docs.unity3d.com/ScriptReference/>] [5])

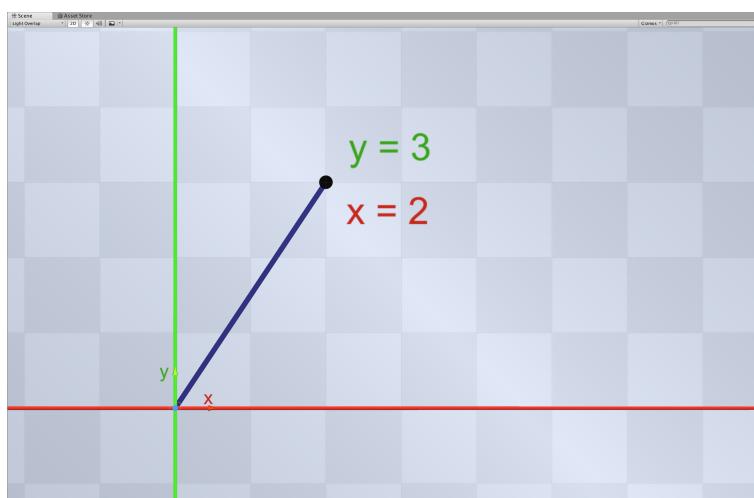


Abbildung 14 - Unity: LineRenderer 2D

Eine weitere Möglichkeit, um Geraden darzustellen ist das Verwenden von Rays. Ein Ray (Strahl) hat einen Ausgangspunkt, eine Richtung aber keinen Endpunkt. Weiters können Rays nur im Szenen-Modus verwendet werden. Für den Game-Modus muss *LineRenderer* verwendet werden. Um einen Ray (Abbildung 15) vom $P = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ in die Richtung $\vec{v} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$ aus zu implementieren, kann folgendes Skript verwendet werden:

```

1 void Start ()
2 {
3     Vector2 startPoint = new Vector2(0,0);
4     Vector2 direction = new Vector2(2,3);
5     Debug.DrawRay(startPoint, direction * 10, Color.blue, float.MaxValue);
6 }
```

Code 9 - Darstellung eines Ray im Szenen Modus

(vgl. [<https://docs.unity3d.com/ScriptReference/>] [6])

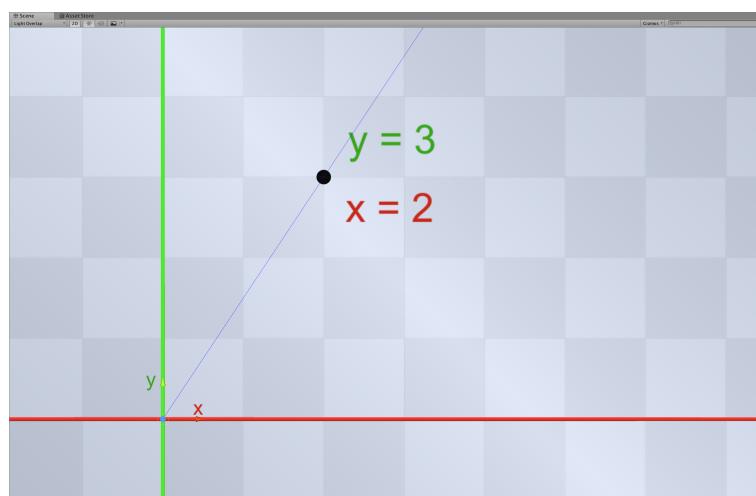


Abbildung 15 - Unity: Ray (Strahl)

Gizmo

Gizmos werden als visuelle Unterstützung für Debugging oder Setups im Szenen-Modus verwendet. Gizmos werden entweder in der Funktion *OnDrawGizmo* oder *OnDrawGizmosSelected* gezeichnet. *OnDrawGizmo* wird mit jedem Frame aufgerufen, *OnDrawGizmosSelected* nur wenn das Objekt, dem das Skript angehängt ist, ausgewählt wird. (vgl. [https://docs.unity3d.com/ScriptReference/] [7])

3.2.2 Transform

(vgl. [https://docs.unity3d.com/ScriptReference/] [8])

Jedes Objekt in einer Unity Szene hat ein Transform Objekt. Dieses wird verwendet, um die Position, Größe und Rotation des Objektes im Raum zu manipulieren und zu speichern. Dazu kann jedes Transform Objekt ein *parent* Objekt haben, mit dem hierarchisch die Position, Größe und Rotation angewendet werden kann.

Position

Mit dem Attribut *position* kann beispielsweise einem Objekt eine neue Position im Raum zugewiesen werden.

```
1 transform.position = new Vector2(2,5);
```

Code 10 - Festlegen der Position eines Objektes

Rotation

Um einem Objekt eine Rotation zuzuteilen gibt es mehrere Möglichkeiten. Es können *transform.eulerAngles* verwendet werden, um die Rotation eines Objektes in der *Update* Funktion zu bestimmen. Die X- und Y- Winkel bestimmen die Rotation in die jeweilige Richtung der Achsen und können mit *Vector2* aufgestellt werden.

Grundlagen Unity5 und C#

```
1 void Update()
2 {
3     transform.eulerAngles = new Vector2(0, 5.0f);
4 }
```

Code 11 - Rotation eines Objektes mit eulerAngles

Man sollte diese aber keinesfalls in *Update* inkrementieren, da diese Funktion einen Fehler produzieren würde, wenn der Winkel 360 Grad überschreitet. Um das zu vermeiden, kann

```
1 void Update()
2 {
3     //Rotation um die eigene X-Achse mit 1 Grad pro Sekunde
4     transform.Rotate(Vector2.right * Time.deltaTime);
5 }
```

Code 12 - Rotation eines Objektes mit Rotate

transform.Rotate verwendet werden. Hierfür wird ein Vektor für die gewünschte Richtung mit einem Wert für die Geschwindigkeit der Rotation angegeben.

Rotate kann auch verwendet werden, um die Rotation relativ zu einem anderen Objekt und nicht um seine eigenen Achsen zu bestimmen. Dafür stellt Unity das *relativeTo* zur Verfügung das im Konstruktor von *Rotate* gesetzt werden kann.

```
1 void Update()
2 {
3     //Rotation in Relation zur Y-Achse der Welt
4     transform.Rotate(Vector2.up * Time.deltaTime, Space.World);
5 }
```

Code 13 - Rotation relativ zur Weltachsen

Es gibt die spezielle Funktion *RotateAround*, mit der ein Objekt um einen bestimmten Punkt rotiert werden kann. Dabei muss der gewünschte Punkt, die Achse als Vektor um die Richtung der Rotation zu bestimmen und der Winkel übergeben werden.

```
1 void Update()
2 {
3     //Rotation um den Nullpunkt in Richtung der Y-Achse
4     transform.RotateAround(Vector2.zero, Vector2.up, Time.deltaTime);
5 }
```

Code 14 - Rotieren um eine Position mit RotateAround

Mit der Funktion *LookAt* kann ein Objekt so rotiert werden, dass dieses immer in die Richtung des Zielobjektes zeigt.

```
1 public Transform target;
2
3 void Update()
4 {
5     //Rotation damit Vector3.forward immer in Richtung des Zielobjektes zeigt
6     transform.LookAt(target);
7 }
```

Code 15 - Ausrichten zu einem Zielobjekt mit LookAt

Verschiebung

Mit der Funktion `transform.Translate` kann ein Objekt im Raum verschoben werden. Dabei wird ein `Vektor2` als Parameter angegeben, in dessen Richtung und Distanz des Betrages die Verschiebung des Objektes stattfindet. Um die Verschiebung nicht abhängig der Eigenachsen des Objektes, sondern dem Koordinatensystem der Welt zu implementieren, kann das Attribut `relativeTo` neben dem `Vector2` verwendet werden.

```

1 void Update ()
2 {
3     transform.Translate(Vector2.up * Time.deltaTime);
4     transform.Translate(Vector2.right * Time.deltaTime, Space.World);
5 }
```

Code 16 - Verschieben eines Objektes mit Translate

3.3 PHYSICS ENGINE

Die Physics Engine in Unity 5 stellt Entwicklern Komponenten für die physikalische Simulation zur Verfügung, damit Game Objekte korrekt beschleunigen können und das richtige Verhalten bei Kollisionen bzw. für Gravitation etc. aufweisen. Die Physik Eigenschaften werden in Unity in regelmäßigen Zeitabständen berechnet (siehe `Update` und `FixedUpdate`). Damit zum Beispiel eine Kollision in Unity von der Physics Engine erkannt wird, muss diese Berechnung genau in dem Moment, in dem sich die beiden betroffenen Objekte berühren, durchgeführt werden. Je schneller sich die Objekte bewegen, desto öfter müssen diese Berechnungen durchgeführt werden, da die Physics Engine sonst die Kollision möglicherweise nicht erkennen kann. Natürlich steigt mit der Anzahl der Berechnungen auch der Performance Impact. Unity beinhaltet jeweils eine eigene Engine für 2D und 3D Physics, die vom Grundkonzept, bis auf die dritte Dimension ident sind. ([Seifert 2014], S. 209)

Diese Physics Eigenschaften für ein Game Objekt können in Unity 5 sehr leicht erreicht werden, indem man zum Beispiel für ein Objekt die Komponente `Rigidbody` hinzufügt, die den Kern der Unity-Physik darstellt. Dafür gibt es wiederum eine 3D und eine 2D Implementation. Dadurch wird das Objekt abhängig von der Physics Engine und reagiert auf Gravitation und Kollisionen mit anderen Objekten. Weiters können durch die Verwendung von Scripts, Objekten bestimmte physikalische Eigenschaften zugewiesen oder abgeändert werden.

In Unity kann man jedem Objekt eine `Rigidbody` Komponente hinzufügen, indem man sie direkt im `Inspector` des Objektes mit der `Add Component` Funktion oder durch ein Skript hinzufügt. Dafür muss das Game Objekt mit dem Namen bzw. mit der Identifikation initialisiert werden damit Unity weiß, welchem Objekt der `Rigidbody` hinzugefügt werden soll.

```

1 GameObject rigidbodyObject= new GameObject("Object_ID");
2 Rigidbody objectRigidBody = myGameObject.AddComponent<Rigidbody>();
```

Code 17 - Erstellen der `Rigidbody` Komponente

([Seifert 2014], S. 210)

Das Hinzufügen der *Rigidbody*-Komponente im *Inspector* lässt das Definieren eines Masseschwerpunktes aber nicht zu, was mit einem Skript jedoch einfach durch ein Statement zu realisieren ist. Dafür gibt es im *Rigidbody* Objekt das Attribut *centerOfMass*. Damit kann als Schwerpunkt eine Position relativ zum eigenen Nullpunkt mit der *Vector3* Klasse als Punkt gesetzt werden.

```
1 GameObject rigidbodyObject= new GameObject("Object_ID");
2 Rigidbody objectRigidBody = myGameObject.AddComponent<Rigidbody>();
3 objectRigidBody.centerOfMass = new Vector3(0, -1, 0);
```

Code 18 - Bestimmen des Schwerpunktes mit centerOfMass

4 ALGORITHMEN UND VISUALISIERUNG IN 2D

4.1 GLEICHFÖRMIGE BEWEGUNG

Um einem Game Objekt eine gleichförmige Bewegung zuzuweisen, kann in Unity die Funktion für Verschiebungen (*Translate*) verwendet werden. Die Bewegung aus dem folgendem Skript findet entlang der X-Achse statt und es wird deshalb nur der X-Wert des *Vector2* mit *deltaTime* verändert. Weiters wird *FixedUpdate* benutzt, um die gleichförmige Bewegung von der physics Wiederholungsrate abhängig zu machen. Um die Geschwindigkeit im Skript Code 19 zu überprüfen, wird mit der Formel für die Geschwindigkeit (Zeile 8) die variable *checkSpeed* in jedem *FixedUpdate* neu berechnet und in der Console ausgegeben. Das Ergebnis ist abgesehen von einem kleinen Rundungsfehler konstant auf 0.5 und beweist den fix gesetzten Wert von 0.5 für die Variable *v* (Zeile 5).

```

1  Vector2 lastPosition;
2
3  void FixedUpdate ()
4  {
5      float v = 0.5f;
6      transform.Translate(v * Time.deltaTime, 0, 0);
7
8      float checkSpeed = (transform.position - lastPosition).magnitude / Time.deltaTime;
9      lastPosition = transform.position;
10     print(checkSpeed);
11 }
```

Code 19 - Gleichförmige Bewegung eines Objektes

4.2 GLEICHMÄßIG BESCHLEUNIGTE BEWEGUNG

Für eine gleichmäßige Beschleunigung muss über eine Berechnung im Skript Code 20 die Geschwindigkeit *v* (Zeile 6) des Objektes gleichmäßig inkrementiert werden. Um sicherzustellen, dass die Berechnung gleichmäßig durchgeführt wird, wird ebenfalls *FixedUpdate* verwendet.

```

1  private float v = 0;
2  private int cycle = 0;
3
4  void FixedUpdate ()
5  {
6      v += 0.001f;
7      transform.Translate(v + Time.deltaTime, 0, 0);
8
9      float speed = (transform.position - lastPosition).magnitude / Time.deltaTime;
10     lastPosition = transform.position;
11
12     if (speed > cycle)
13     {
14         print( "Speed at : " + Math.Round(Time.fixedTime) +
15               " seconds is " + Math.Round(speed));
16         cycle += 5;
17     }
18 }
```

Code 20 - Gleichmäßige Beschleunigung eines Objektes

Um die Beschleunigung zu überprüfen, wird mit einer If-Abfrage (Zeile 12) gleichmäßig überprüft, ob die Geschwindigkeit größer als ein Hilfsintervall *cycle* (Zeile 2) ist, das nach Erfolg auch inkrementiert wird und die Geschwindigkeit bei der aktuell vergangenen Zeit (*fixedTime*: Zeile 14) auf die Console in Abbildung 16 ausgibt.

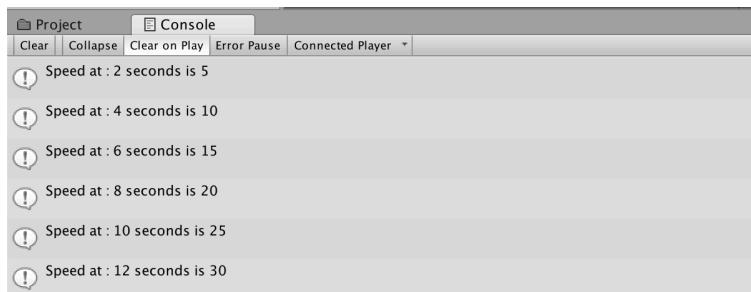


Abbildung 16 - Unity: Ausgabe der Geschwindigkeit per Intervall

4.3 ZENO'S PARADOXON

Dieser Algorithmus berechnet eine Bewegung, bei der ein Objekt o_1 der Position eines anderen Objektes o_2 folgt. Die Berechnung der Position im Skript Code 21 (Zeile 25) lässt sich mit folgender Formel erreichen:

$$Position_{neu} = \frac{Position_{o_2} - Position_{o_1}}{coeff}$$

Die Differenz der Position wird durch einen Koeffizienten *coeff* dividiert, mit dem bestimmt wird, wie schnell o_1 dem Objekt o_2 folgt.

```

1  public class zenos : MonoBehaviour
2  {
3      private GameObject sphere, ghostSprite;
4      private float divident, distance;
5      Text distanceText;
6
7      void Start ()
8      {
9          transform.position = new Vector2(0,0);
10         sphere = GameObject.Find ("Sphere");
11         ghostSprite = GameObject.Find ("ghostSprite");
12         distanceText = GameObject.FindGameObjectWithTag("distanceText")
13             .GetComponent<Text>();
14         divident = 20.0f;
15     }
16
17     void FixedUpdate ()
18     {
19         if (Input.GetKey("up")) transform.position += Vector3.up;
20         if (Input.GetKey("down")) transform.position += Vector3.down;
21         if (Input.GetKey("right")) transform.position += Vector3.right;
22         if (Input.GetKey("left")) transform.position += Vector3.left;
23
24         Vector3 spherePosition = sphere.transform.position;
25         sphere.transform.position += (transform.position - spherePosition) / divident;
26         ghostSprite.transform.position = sphere.transform.position;
27
28         distance = (transform.position - sphere.transform.position).magnitude;
29         distanceText.text = "Distance is: " + System.Math.Round(distance, 2);
30     }
31 }
```

Code 21 - Algorithmus für Zeno's Paradoxon

Im Skript wurde zur Darstellung (Abbildung 17) des Algorithmus ein Rechteck (blau) erzeugt, dass mit den Pfeiltasten auf der X- und Y-Achse bewegt werden kann. Dazu wurde ein Kreis (Baseball) erzeugt, auf die in Zeile 10 referenziert wird, die der Position des Rechtecks folgt. Die Distanz zwischen dem Rechteck und dem Baseball wird ebenfalls im Bild angezeigt. Die Einheit der Länge ist eine Unity Einheit.

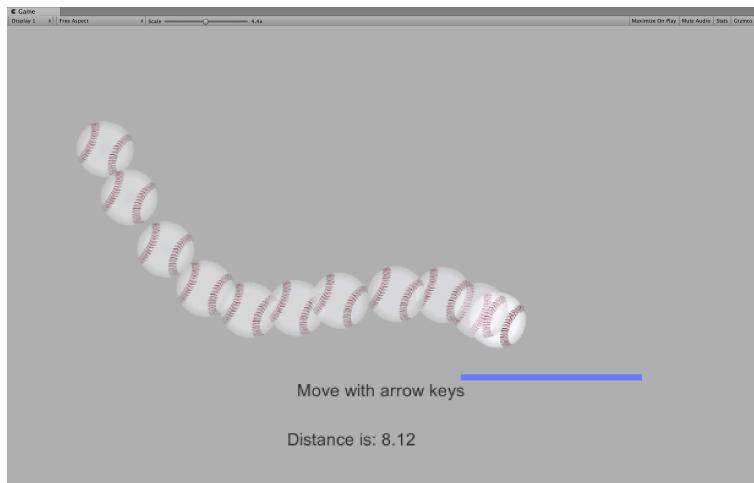


Abbildung 17 - Unity: Darstellung Zeno's Paradoxon

4.4 WINKELBEWEGUNG

4.4.1 Sinus

Für das Implementieren einer Sinusbewegung in Unity benötigt der Algorithmus Code 22 drei Variablen (Zeile 3 bis 5) zur Bestimmung der neuen Position auf der Y-Achse.

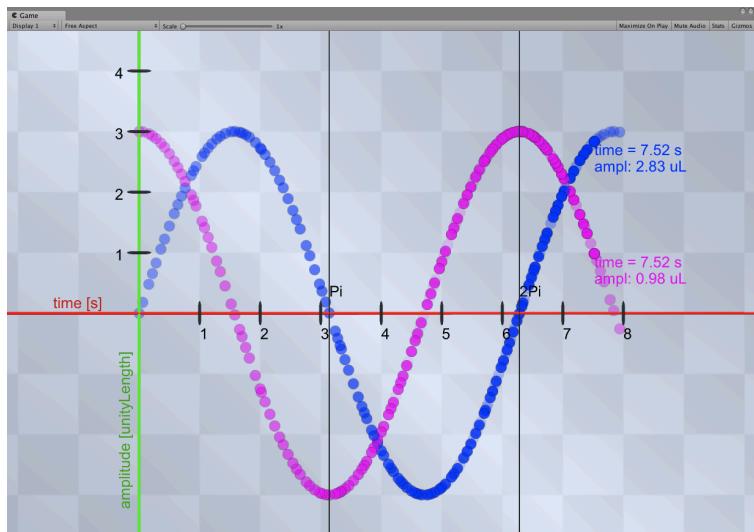
```

1  public class sinMove : MonoBehaviour
2  {
3      private float frequency = 1;
4      private float amplitude = 3;
5      private float timeFrame = 0;
6      private Vector3 xMovement, yMovement;
7      private TextMesh xValue, yValue;
8
9      void Start ()
10     {
11         transform.position = Vector3.zero;
12         xValue = GameObject.Find ("sinXValue").GetComponent<TextMesh>();
13         yValue = GameObject.Find ("sinYValue").GetComponent<TextMesh>();
14     }
15
16     void FixedUpdate ()
17     {
18         timeFrame += Time.deltaTime;
19         if (timeFrame > 8) { timeFrame = 0; }
20         xMovement = transform.right * timeFrame;
21         yMovement = transform.up * (float)System.Math.Sin (frequency * timeFrame);
22         transform.position = xMovement + yMovement * amplitude;
23         xValue.text = "time = "
24         System.Math.Round(timeFrame, 2).ToString() + " s";
25         yValue.text = "ampl: "
26         System.Math.Round(transform.position.y, 2).ToString() + " uL";
27     }
28 }
```

Code 22 - Algorithmus für eine Sinusfunktion

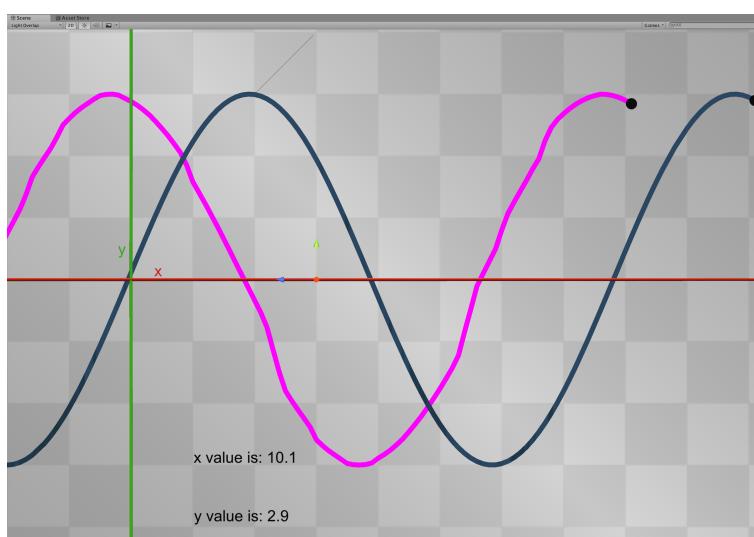
Algorithmen und Visualisierung in 2D

Die Variable *timeFrame* bestimmt die Bewegung entlang der X-Achse und wird in Zeile 18 mit *Time.deltaTime* inkrementiert. *TimeFrame* wird für die Visualisierung alle acht Sekunden nullgesetzt. Die Variable *frequency* bestimmt zusammen mit der Sinus Funktion aus Unity und dem *Vector3.up* die Position auf der Y-Achse (Zeile 21). Die Variable *amplitude* regelt in Zeile 22, wie weit die Bewegung in die Y-Achse ausschlägt. Zur Darstellung der Sinusfunktion in Abbildung 18 (blau) wurde, dem sich bewegenden Punkt ein *GhostSprites2D* hinzugefügt, der ihm eine blaue Spur hinterherzieht. Die exakt gleiche Berechnung für eine Kosinus Funktion (pink) durchgeführt.



Durch die eingezeichneten Werte π und 2π kann die Periodenlänge mit den Werten der Variablen überprüft werden. Die Einheit der X-Achse ist Sekunden und die Einheit der Y-Achse die Unity-Längeneinheit.

Um den Unterschied zwischen *Update* und *FixedUpdate* hervorzuheben, sind in Abbildung 19 zwei Sinusfunktionen dargestellt, wobei die schwarze Funktion mit *FixedUpdate* berechnet wurde und die pinke Funktion mit *Update*.



4.4.2 Wurfparabel

In Unity kann der Wurf eines Objektes leicht mit einer *Rigidbody* Komponente erreicht werden. Für dieses Beispiel wird der Wurf (Code 23) jedoch mit einer Variablen (Zeile 10) für die Geschwindigkeit, die für die Verschiebung auf der X-Achse und einer Gravitationskraft (Zeile 9) für die Verschiebung auf der Y-Achse simuliert.

```

1  public class throwBall : MonoBehaviour
2  {
3      private float gravityForce, velocity;
4      private TextMesh xValue, yValue;
5
6      void Start ()
7      {
8          transform.position = new Vector3(-4,4,0);
9          gravityForce = 0.1f;
10         velocity = 7;
11         xValue = GameObject.Find ("xValue").GetComponent<TextMesh>();
12         yValue = GameObject.Find ("yValue").GetComponent<TextMesh>();
13     }
14
15     void FixedUpdate ()
16     {
17         gravityForce += 0.1f;
18         float y = gravityForce * Time.deltaTime;
19         float x = Time.deltaTime * velocity;
20         transform.position += new Vector3(x, -y, 0);
21         xValue.text = "time: " +
22             System.Math.Round(transform.position.x, 2).ToString() + " s";
23         yValue.text = "height: " +
24             System.Math.Round(transform.position.y, 2).ToString() + " uL";
25     }
26 }
27

```

Code 23 – Wurfparabel

Die Gravitationskraft wird mit jedem *FixedUpdate* inkrementiert (Zeile 17). Die X- und Y-Koordinaten für die Verschiebung werden in Zeile 18 und 19 berechnet.

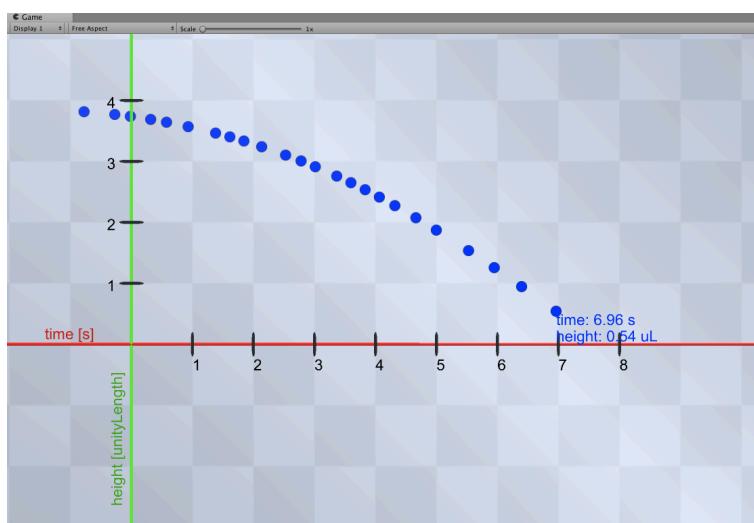


Abbildung 20 - Unity: Wurfparabel

4.4.3 Pendel

Um eine Pendelbewegung zu simulieren sind mehrere Variablen nötig: Die Länge des Pendelseils, der Winkel, die Gravitationskraft und die Winkelgeschwindigkeit. [Jen-Chin Lin, Jeng-Fung Hung, (2009)]

Um ein schwingendes Pendel in Unity zu integrieren, wurden ein Kreis mit einer *LineRenderer* Komponente als Seil zum Ursprung der Winkelfunktion gezogen und dem Skript Code 24 zur Berechnung der Position nach der Zeit mit der Euler-Cromer Methode (Zeile 28-31) hinzugefügt.

```

1  public class pendulum : MonoBehaviour
2  {
3      private float length = 2;    //length of line
4      private float omega, theta, theta_old, omega_old, gravity, timeFrame;
5      private Vector3 p, p0, oldPoint;
6      LineRenderer lineRenderer;
7      private TextMesh velocityText, timeText;
8      Transform ghostPoint, velPoint;
9
10     void Awake()
11     {
12         gravity = 9.81f;
13         omega = 1; //initial velocity
14         theta = 0.2f; //initial angle, must not be zero
15         p0 = transform.position;
16         p0.y += length;
17         lineRenderer = gameObject.AddComponent<LineRenderer>();
18         lineRenderer.widthMultiplier = 0.05f;
19         lineRenderer.SetPosition(0, transform.position + new Vector3(0, length, 0));
20         ghostPoint = GameObject.Find ("ghostPoint").transform;
21         velPoint = GameObject.Find ("velPoint").transform;
22         velocityText = GameObject.Find ("velocityText").GetComponent<TextMesh>();
23         timeText = GameObject.Find ("timeText").GetComponent<TextMesh>();
24     }
25
26     void FixedUpdate()
27     {
28         omega_old = omega;
29         theta_old = theta;
30         omega = omega_old - (gravity / length) * theta_old * Time.deltaTime;
31         theta = theta_old + omega * Time.deltaTime;
32         p.z = p0.z;
33         p.y = p0.y + -length * Mathf.Cos (theta);
34         p.x = p0.x + length * Mathf.Sin (theta);
35         transform.position = p;
36         ghostPoint.position = p;
37         lineRenderer.SetPosition(1, p);
38         velPoint.position = new Vector3(timeFrame, System.Math.Abs(omega), 0);
39         timeFrame += Time.deltaTime;
40         if (timeFrame > 8) { timeFrame = 0; }
41         velocityText.text = "Velocity is: " +
42             System.Math.Round(System.Math.Abs(omega), 2) + " rad/s";
43         timeText.text = "Time is: " + System.Math.Round(timeFrame, 2) + " seconds";
44     }
45 }
```

Code 24 - Implementation Pendel

(Code Vergleich: Rd Gamestudio, <https://rdgamestudio.wordpress.com/2016/05/04/simple-pendulum-in-unity/>, 17.09.2017)

Algorithmen und Visualisierung in 2D

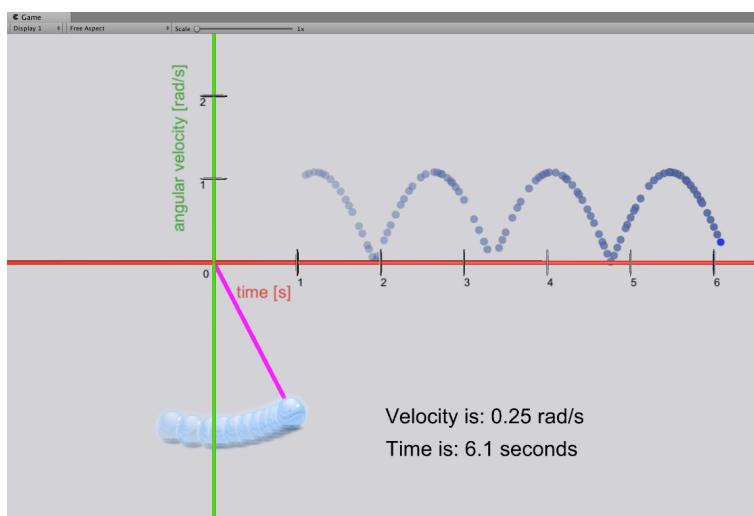


Abbildung 21 - Unity: Pendel

Die Einheit der X-Achse des Graphs (blau) in Abbildung 21 ist die vergangene Zeit, die Einheit der Y-Achse ist die Winkelgeschwindigkeit der Pendelbewegung. Da der Algorithmus eine gleichmäßige Pendelbewegung erzeugt, ist das Ergebnis des Graphen, über die Zeitachse gesehen eine gleichförmige Sinusbewegung mit positiver Amplitude. Die Sinusbewegung wird zur Darstellung alle acht Sekunden zurückgesetzt.

4.4.4 Kreisbewegung

Die Kreisbewegung wurde mit der Parameterdarstellung eines Kreises (Kapitel 2.8) folgendem Beispiel (Code 25) implementiert. Durch die Variable *speed* wird berechnet wieviel Grad pro Sekunde der Punkt an der Kreisoberfläche zurücklegt (Zeile 11). Die X- und Y-Werte werden in Zeile 28 und 29 berechnet.

```

1  public class circleMove : MonoBehaviour
2  {
3      private Vector3 distanceToTarget;
4      private TextMesh xValue, yValue, speedText, radiusText, timeText;
5      private float radius, angle, speed, xNew, yNew, timeFrame;
6
7      void Start ()
8      {
9          radius = 3;
10         angle = 0;
11         speed = (2 * Mathf.PI) / 4; //360 Grad in 4 seconds
12         xValue = GameObject.Find ("xValue").GetComponent<TextMesh>();
13         yValue = GameObject.Find ("yValue").GetComponent<TextMesh>();
14         speedText = GameObject.Find ("speed").GetComponent<TextMesh>();
15         radiusText = GameObject.Find ("radius").GetComponent<TextMesh>();
16         timeText = GameObject.Find ("time").GetComponent<TextMesh>();
17         transform.position = new Vector3(3, 0, -1);
18     }
19
20     void FixedUpdate ()
21     {
22         if (Input.GetKey("up") && speed < 5) speed += 0.1f;
23         if (Input.GetKey("down") && speed > -5) speed -= 0.1f;
24         if (Input.GetKey("right") && radius < 7) radius += 0.1f;
25         if (Input.GetKey("left") && radius > 1) radius -= 0.1f;
26
27         angle += speed * Time.deltaTime;
28         xNew = Mathf.Cos(angle) * radius;
29         yNew = Mathf.Sin(angle) * radius;
30         transform.position = new Vector3(xNew, yNew, -3);
31
32         xValue.text = "x value: " +
33             System.Math.Round(transform.position.x, 2).ToString() + " uL";
34         yValue.text = "y value: " +
35             System.Math.Round(transform.position.y, 2).ToString() + " uL";
36         speedText.text = "angular vel: " + System.Math.Round(speed,2) + " rad/s";
37         radiusText.text = "radius: " + System.Math.Round(radius,2) + " uL";
38         timeText.text = "time: " + System.Math.Round(Time.time, 2) + " s";
39
40         if (System.Math.Round(transform.position.y, 2) == 0)
41             print(Time.time);
42     }
43 }
```

Code 25 - Algorithmus für Kreisbewegung

Neben der Berechnung mit der Parameterdarstellung für eine Kreisbewegung gibt es in Unity die Funktionen *RotateAround* *transform.rotation*, *transform.eulerAngles* und *transform.Rotate* (siehe Kapitel 3.2.2 Rotation).

Algorithmen und Visualisierung in 2D

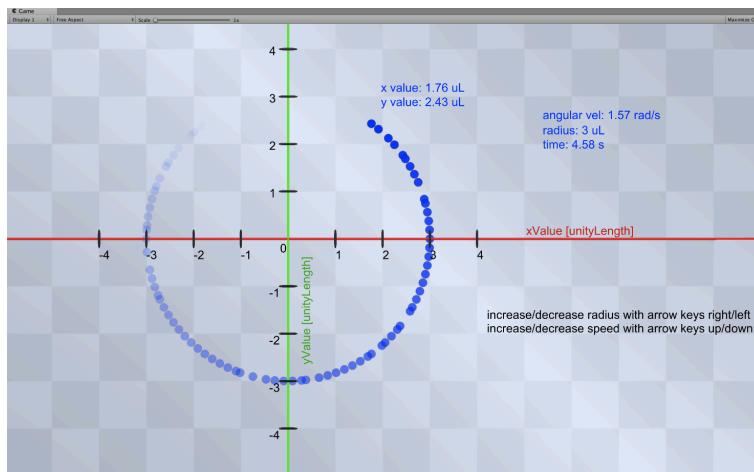


Abbildung 22 - Unity: Darstellung Kreis

Um die Kreisbewegung des Objektes darzustellen (Abbildung 22) wurde das Unity Asset *GhostSprites2D* verwendet, das dem Punkt eine blaue Spur hinterzieht. Der Grund für die ungleichmäßige Verteilung der Punkte liegt nicht an der Unity Engine und der *FixedUpdate* Funktion, sondern an der Implementierung des *GhostSprites2D* Assets.

Die Einheit der X- und Y-Achse ist die Unity-Längeneinheit. Der Winkel und Radius können über die Pfeiltasten gesteuert werden. Erhöht man beispielsweise den Radius, mit einer fixen Winkelgeschwindigkeit gleichmäßig hinauf und hinunter, bewegt sich der Punkt sternförmig um den Nullpunkt.

Mit einem fixen Wert $speed = \frac{2*\pi}{4}$ dauert eine 360° Kreisbewegung, mit der *FixedUpdate* Methode, 4 Sekunden. Im dazugehörigen Skript wird jedes Mal, wenn der berechnete Punkt die X-Achse schneidet, die vergangene Zeit auf der Console (Abbildung 23) ausgegeben. Wie zu sehen ist wird die Zeit rund alle zwei Sekunden ausgegeben und bestätigt die Berechnung mit *FixedUpdate*.

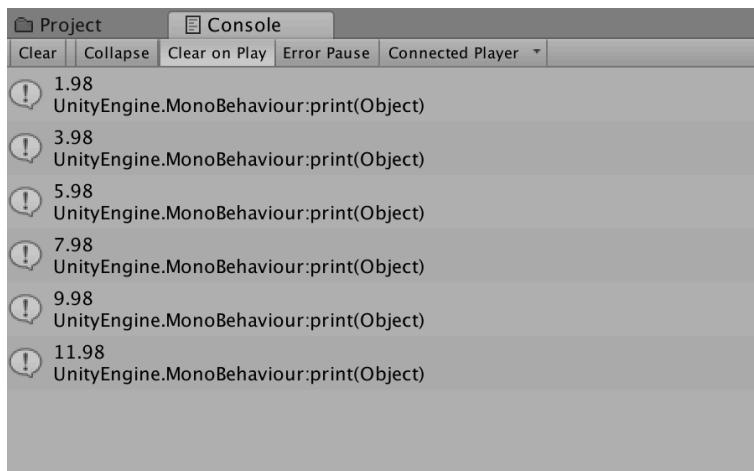


Abbildung 23 - Consolen Output Kreis

4.5 ROBOTIK GELENK

Für dieses Beispiel wurden vier Objekte in Unity erzeugt um die Komponenten des Roboterarms darzustellen (siehe Abbildung 24). Die Gelenke lassen sich mit Pfeiltasten der Tastatur steuern. Dafür wurden einfache Vektoren und Rotationsberechnungen verwendet. Durch die zwei miteinander verbundenen Gelenke wird die Position des angehefteten Gelenkes abhängig von der Position des Ausgangsgelenkes.

```

1  public class robo : MonoBehaviour
2  {
3      private float speed;
4      Transform joint1, joint2, arm1, arm2;
5
6      void Start ()
7      {
8          speed = 40;
9          joint1 = GameObject.Find ("joint1").transform;
10         joint2 = GameObject.Find ("joint2").transform;
11
12         joint2.position = joint1.position + joint1.up * 5;
13     }
14
15     void FixedUpdate ()
16     {
17         if (Input.GetKey("right"))
18             joint1.Rotate(Vector3.right * speed * Time.deltaTime);
19
20         if (Input.GetKey("left"))
21             joint1.Rotate(Vector3.left * speed * Time.deltaTime);
22
23         joint2.position = joint1.position + joint1.up * 5;
24
25         if (Input.GetKey("up"))
26             joint2.Rotate(Vector3.right * speed * Time.deltaTime);
27
28         if (Input.GetKey("down"))
29             joint2.Rotate(Vector3.left * speed * Time.deltaTime);
30     }
31 }
```

Code 26 - Robo Arm

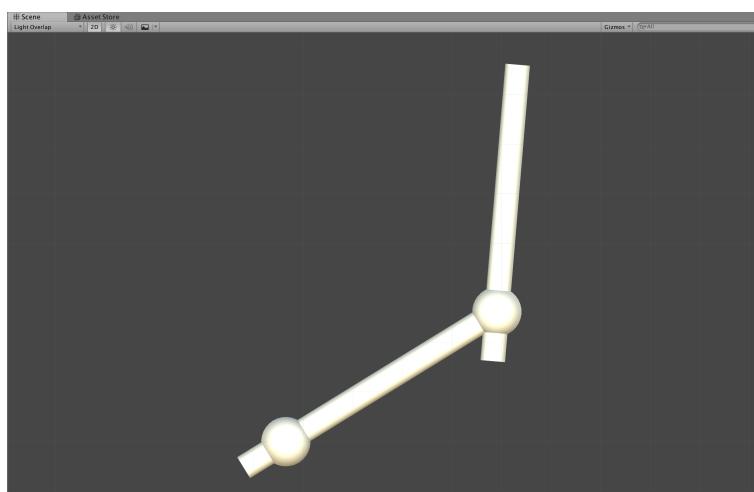


Abbildung 24 - Unity: Robo Arm

4.6 ELASTISCHE BEWEGUNG

Dieses Beispiel (Code 27) simuliert Gravitation und Elastizität mit einem Ball der auf einer Oberfläche springt. Mit einem *Rigidbody* und einem *Physics-Material* können diese Eigenschaften für eine *GameObject* ohne Code simuliert werden. Das Erkennen der Kollision regelt die Funktion *OnCollisionEnter* (Zeile 41) und verwendet um den Richtungsvektor der Bewegung zu ändern. Die Verschiebung der Position (Zeile 20) wird mit der berechneten Geschwindigkeit (Zeile 25 / 31), der Richtung (direction, Zeile 6) und einer Gravitationskraft (Zeile 9) berechnet. Weiters wird mit der Variable *bounciness* (Zeile 10) die Elastizität simuliert. Erhöht man nun die Gravitationskraft, steigt die Geschwindigkeit, mit der der Ball fällt. Erhöht man *bounciness* springt der Ball öfters auf und ab.

```

1 void Start ()
2 {
3     upOrDown = true;
4     stop = false;
5     velocity = 0;
6     direction = Vector3.down;
7     g = 9.81f;
8     gravityMultiplier = 2.5f;
9     gravityForce = g * gravityMultiplier;
10    bounciness = 0.2f;
11    startHeight = transform.position + Vector3.up;
12    oldHeight = startHeight;
13 }
14
15 void FixedUpdate ()
16 {
17     if (!stop)
18     {
19         float distance = direction * velocity * gravityForce;
20         transform.position += distance * Time.deltaTime;
21
22         if (direction == Vector3.down)
23         {
24
25             velocity += (startHeight - transform.position).magnitude
26                         * Time.deltaTime * bounciness;
27         }
28         else
29         {
30             if (velocity >= 0)
31                 velocity -= Time.deltaTime * transform.position.magnitude;
32             else
33             {
34                 direction = Vector3.down;
35                 oldHeight = transform.position;
36             }
37         }
38     }
39 }
40
41 void OnCollisionEnter(Collision col)
42 {
43     if (oldHeight.y > 1.1)
44     {
45         upOrDown = false;
46         direction = Vector3.up;
47         if (bounciness >= 0.2f) bounciness -= 0.1f;
48     }
49     else { stop = true; }
50 }
```

Code 27 - Springender Ball

4.7 PINGPONG SPIEL

Diese Demonstration eines Pingpongspiels ist eine gute Anwendung für Vektorberechnungen. Um den Spielball von den Barrieren abprallen zu lassen kann die Klasse *Vector2.Reflect* verwendet werden. Um die Reflexion in einem eigenen Skript (Code 28) selbst zu berechnen kommt das Reflexionsgesetz zum Einsatz und wird in Zeile 30 berechnet. Es wurde eine Steuerung eingebaut (Zeile 22-25), mit der man die beiden Tor-Blockaden steuern kann.

```

1  public class pingpong : MonoBehaviour
2  {
3      Transform player1, player2;
4      private float velocity;
5      private Vector3 direction;
6
7      void Start ()
8      {
9          player1 = GameObject.Find ("player1").transform;
10         player2 = GameObject.Find ("player2").transform;
11
12         velocity = 0.1f;
13         direction = new Vector3(velocity, 0.1f, 0);
14     }
15
16     void FixedUpdate ()
17     {
18         transform.position += direction;
19
20         if (Input.GetKey(KeyCode.Q)) player1.position += Vector3.up * 0.3f;
21         if (Input.GetKey(KeyCode.A)) player1.position += Vector3.down * 0.3f;
22         if (Input.GetKey("up")) player2.position += Vector3.up * 0.3f;
23         if (Input.GetKey("down")) player2.position += Vector3.down * 0.3f;
24     }
25
26     void OnCollisionEnter(Collision col)
27     {
28         direction = direction - 2 * (Vector3.Dot(direction, col.transform.forward))
29                     * col.transform.forward;
30     }
31 }
32

```

Code 28 - Pingpong Spiel

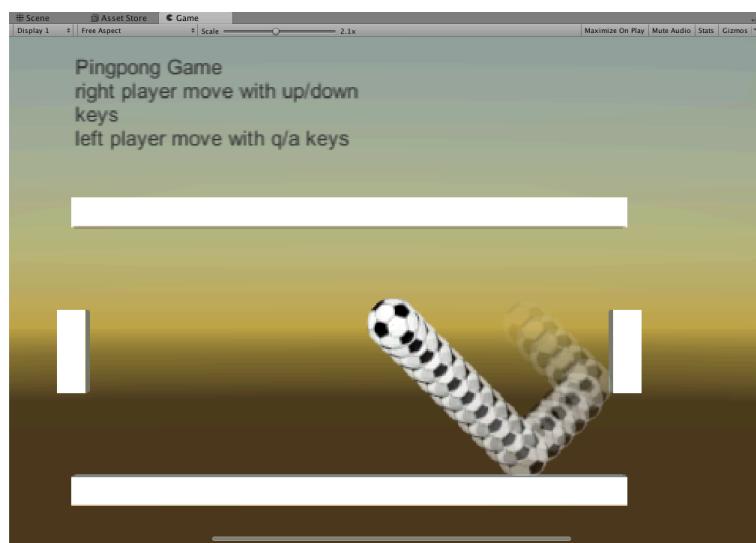


Abbildung 25 - Unity: Pingpong Spiel

4.8 BILLARD SPIEL

Dieses Beispiel veranschaulicht die Vektorreflexion, Rotation, Reibung, Drall und andere Berechnungen aus vorhergehenden Beispielen. Je nachdem wie stark und in welchem Winkel der Queue auf den Ball trifft, ändern sich Bewegungsgeschwindigkeit, Bewegungsrichtung, und Rotation des Balls.

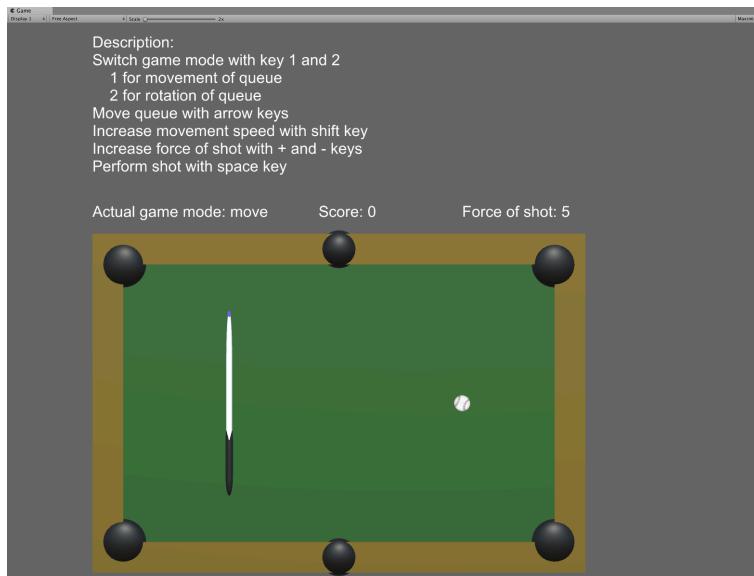


Abbildung 26 - Unity: Billiard Spiel

Da das ganze Skript ca. 250 Zeilen (Datei *billard.cs* auf der beigelegten CD) fasst, ist in Code 29 der wichtigste Teil der Berechnungen zu sehen. Die Variable *angle* (Zeile 2) bestimmt wie stark die Rotation des Balls, in Abhängigkeit des Impulses des Queue, zu- und abnimmt. Mit *driftSpeed* (Zeile 9) wird geregelt, wie stark der Drall, in Abhängigkeit der Kraft und des Winkels, der eigentlichen Richtung des Balls entgegenwirkt und einen Bogenstoß darstellt.

```

1 //controls speed of spinn motion
2 float angle = (-1) * this.spinSpeed * 500 * (1 - this.spinFriction)
3           * this.moveForce * 5;
4
5 //rotates the ball
6 transform.eulerAngles += new Vector3(0, 0, angle);
7
8 //value of drift dependent on spin and force
9 this.driftSpeed = (-150) * this.spinSpeed * this.moveForce;
10
11 //claculation for movement on x and y axis
12 this.transform.position += (this.moveDirection + new Vector3(this.driftSpeed, 0, 0)
13           * (this.driftFriction)) * this.moveForce * (1- this.friction);

```

Code 29 - Wichtigste Berechnungen für Billiard Beispiel

5 ANWENDUNGSBEREICHE

5.1 ANIMATION

Eine Anwendung für Algorithmen in 2D spiegelt sich in Animationen wieder, da eine Animation nichts anderes ist, als eine oder mehrere berechnete Bewegungen eines oder mehrerer Objekte. Dabei gibt es viele Techniken, von denen einige in den folgenden Punkten näher erläutert werden.

5.1.1 Keyframing

Durch Keyframing gibt der Animator mehrere Parameter für ein Objekt zu einer gewissen Position zu einer gewissen Zeit an, zu denen Zwischenschritte in Form von Position und/oder Winkel vom Computer berechnet werden. Dadurch wird ein flüssiger Ablauf der Bewegung sichergestellt. (vgl. [Leo Hackstein] [1])

5.1.2 Inverse Kinematik

Diese Techniken werden beispielsweise für Gelenkbewegungen in der Robotik angewendet. Bei der Vorwärtsskinematik wird von den Parametern der Gelenke (Winkel) die Position der Endeffektoren berechnet. Bei der inversen Kinematik berechnet man die Gelenkparameter ausgehend von den Endeffektor-Positionen. (siehe Code 26 - Robo Arm)

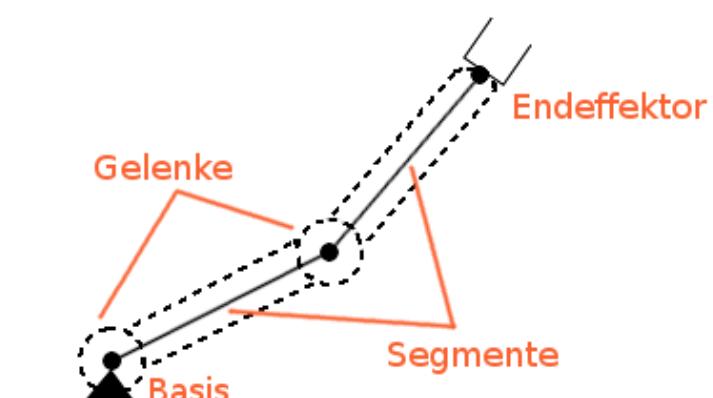


Abbildung 27 - Bestandteile eines Robo Arms [<http://wiki.ifs-tud.de/>] []

Eine einzelne Gelenksbewegung in der Robotik ist nichts anderes als eine zweidimensionale Bewegung. Eine Überlagerung mehrerer Bewegungen führt meist zu einer dreidimensionalen Bewegung, abhängig von der Position und dem Bewegungswinkel der Gelenke. (vgl. [Daniela Steidl][1])

5.1.3 Motion Capture

Mit Motion Capture wird das Skelett eines Schauspielers über Sensoren auf den Computer als 2D oder 3D Skelett übertragen. Dabei liefert jeder Sensor je nach Anwendung zwei oder drei Koordinatenpunkte, die so vom Schauspieler auf das digitale Model übertragen werden (Abbildung 28). So können reale Bewegungen auf ein digitales Model übertragen werden.

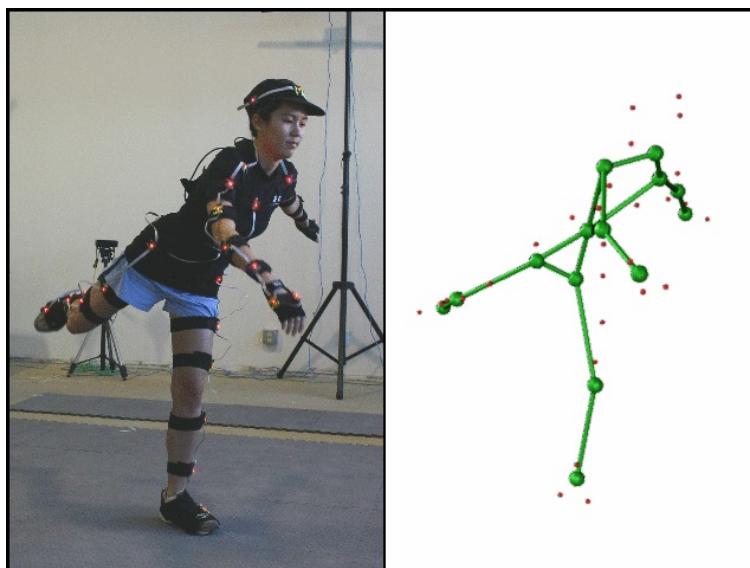


Abbildung 28 - Motion Capture: Reales Skelett zu digitalem Skelett. [<http://graphics.berkeley.edu/>] [1]

Motion Capturing wird neben Animationen vorläufig für Computerspiele, Virtual Reality und die Filmproduktion verwendet.

5.2 SPIELE

Algorithmen für 2D Bewegungen werden sehr häufig für 2D Spiele und auch 3D Spiele verwendet. Dabei hat der Spieler selbst in der Regel großen Einfluss auf die Bewegungen bestimmter Objekte im Spiel. Durch Eingabegräte wie Maus, Tastatur und Gaming Controller werden die Eingabeparameter im Spiel in Bewegung umgesetzt.

5.2.1 Jump and Run

So findet man beispielsweise in jedem „Jump and Run“ 2D Spiel Algorithmen, die Geschwindigkeit, Sprünge, Schüsse, Rotationen von Objekten etc. berechnen.



Abbildung 29 - 2D Jump and Run: Super Mario ([\[http://nintendo-online.de/\]](http://nintendo-online.de/)) [1]

5.2.2 Shooter

In 2D Shootern wie zum Beispiel Space Shootern finden vor allem Algorithmen für konstante und beschleunigte Bewegungen sowie Winkelbewegungen und Vektor bzw. Matrizenberechnungen Anwendung.

5.2.3 Vogelperspektive

2D Action bzw. Strategiespiele werden oftmals aus der Vogelperspektive dargestellt und bieten so zahlreiche Anwendungen für Bewegungsalgorithmen in 2D.

5.3 GESTENERKENNUNG

In vielen Anwendungen wie Virtual Reality und Augmented Reality gibt es eine Gestenerkennung, die als Interface zur Steuerung gewisser Elemente oder Ereignisse zwischen dem Menschen und der virtuellen Welt dient. Dafür werden Kamera und Infrarot basierte Systeme verwendet, die den Ablauf der Bewegung durch Algorithmen der Positionserkennung, Richtung und Geschwindigkeit berechnen. Für diese Verfahren ist die Algorithmik der Bewegungserkennung sehr gefragt, da die von den Sensoren gelieferten Ergebnisse richtig interpretiert werden müssen. Beispielsweise muss für Kopfbewegungen zwischen einem Nicken und Kopfschütteln klar unterschieden werden.

6 RESÜMEE

Eine anfänglich simple Bewegung in der Realität kann im digitalen Raum wesentlich komplexer erscheinen. Um Bewegungen in einem Computermodell realistisch darstellen zu können, kommen noch einige Faktoren hinzu, die einen Algorithmus noch komplizierter erscheinen lassen. Für ein erfolgreiches Arbeiten an dieser Thematik sind gewisse Grundlagen, wie Formeln zur mathematischen Berechnung und grundlegende Programmierkenntnisse unumgänglich, um die Formeln für Bewegungen im zweidimensionalen Raum in Code umzusetzen.

Während das Implementieren von Bewegungen mit den vorintegrierten Funktionen von dem Unity – C# Framework sehr einfach zu erreichen ist, wie zum Beispiel die Implementierung einer Pendelbewegung (Kapitel 4.4.3) mit der Physics engine, stellt es eine größere Herausforderung dar, diesen Algorithmus selbstständig zu berechnen. Ebenfalls kann die selbstständige Implementierung von Bewegungsalgorithmen sehr aufwendig sein, da zum Beispiel in der Unity API nur die von Haus aus integrierten Funktionen für diese Algorithmen zur Verfügung gestellt werden und es immer wieder notwendig ist, in Quellen einer anderen Programmiersprache nach Algorithmen zu suchen und gefundene Code Elemente in die eigene Programmiersprache zu übersetzen. Zur Ideenfindung ist das Recherchieren für Algorithmen in anderen Programmiersprachen zu empfehlen, da der Ablauf der Berechnungen eines Algorithmus in vielen Programmiersprachen gleich umgesetzt werden kann.

Durch die Visualisierung mit Unity5 wird der Vorgang für den Aufbau des Verständnisses beschleunigt, da man durch vielseitiges Testen und Erweitern der Algorithmen schnell Erfahrung sammeln und diese auf weitere Beispiele ausbauen kann.

In modernen Entwicklungsumgebungen wie Unity werden sehr viele Funktionen bereitgestellt, die das Bewegen von Game Objects sehr einfach machen. Ist jedoch der Punkt erreicht, an dem man die Bewegung selbst programmieren muss, ist es von Vorteil, Grundlagen der Algorithmik und Naturwissenschaften verstanden zu haben. Die Inhalte dieser Arbeit, spiegeln genau diese Grundlagen wider und können Anfängern in der Thematik helfen, ihre eigenen Algorithmen für Bewegungen im zweidimensionalen Raum zu implementieren.

Verfügt man über gute Programmierkenntnisse können bestehende mathematische Formeln für einfache Bewegungen problemlos in Code umgesetzt werden. Bei komplexeren Bewegungen, für die es im verwendeten Framework keine vorgefertigten Funktionen gibt, ist es jedoch notwendig, selbstständig einen Algorithmus dafür zu entwickeln.

LITERATURVERZEICHNIS

Bücher

[Seifert 2014]

Carsten Seifert: Spiele entwickeln mit Unity 5. Carl Hanser Verlag 2015

[Papula 2009]

Lothar Papula: Mathematische Formelsammlung. 10. Auflage. Vieweg + Teubner Verlag 2009

Whitepaper

Paul E. Dickson, Jeremy E. Block, Gina N. Echevarria, and Kristina C. Keenan (2017); An Experience-based Comparison of Unity and Unreal for a Stand-alone 3D Game Development Course; <https://dl.acm.org/>; 23.09.2017

Lubomir Ivanov (2015); 3D Game Development with Unity in the computer science curriculum; <https://dl.acm.org/>; 25.09.2017

Jen-Chin Lin, Jeng-Fung Hung, (2009); The Integration of Simulation Modeling System for Student Experiment and Modeling Ability Evaluation; <https://dl.acm.org/>; 27.09.2017

Internet-Referenzen

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

[1] Monobehaviour; <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>;

26.08.2017

[2] Vector2 - magnitude; <https://docs.unity3d.com/ScriptReference/Vector2-magnitude.html>; 26.08.2017

[3] Vector2 – Distance; <https://docs.unity3d.com/ScriptReference/Vector2.Distance.html>; 26.08.2017

[4] Vector2 – normalized; <https://docs.unity3d.com/ScriptReference/Vector2-normalized.html>; 26.08.2017

[5] LineRenderer; <https://docs.unity3d.com/ScriptReference/LineRenderer.html>; 26.08.2017

[6] Debug - DrawRay; <https://docs.unity3d.com/ScriptReference/Debug.DrawRay.html>; 26.08.2017

[7] Gizmos; <https://docs.unity3d.com/ScriptReference/Gizmos.html>; 26.08.2017

[8] Transform; <https://docs.unity3d.com/ScriptReference/Transform.html>; 26.08.2017

[Grundwissen Physik]

Grotz Bernhard: Grundwissen Physik. <https://www.grund-wissen.de/physik/index.html>, 12.09.2017

[Computational Physics]

(B. Ydri, A. Bouchareb, R. Chemam: Lectures on Computational Physics, 2013;
https://homepages.dias.ie/ydri/COMPUTATIONALNOTES_FINAL.pdf; 21.09.2017)

Literaturverzeichnis

[Leo Hackstein]

[1]

https://wwwcg.in.tum.de/fileadmin/user_upload/Lehrstuehle/Lehrstuhl_XV/Teaching/WS07_08/Seminar/talks/Leo_Hackstein.pdf; 21.09.2017)

[Daniela Steidl]

[1]

https://wwwcg.in.tum.de/fileadmin/user_upload/Lehrstuehle/Lehrstuhl_XV/Teaching/SS11/Proseminar-PIXAR/Daniela_Steidl.pdf, 21.09.2017)

[<https://math.stackexchange.com/>]

(<https://math.stackexchange.com/questions/13261/how-to-get-a-reflection-vector>;
25.09.2017)

[<http://wiki.ifs-tud.de/>]

[1] http://wiki.ifs-tud.de/_media/biomechanik/projekte/ws2012/roboarm_naming.png;
23.09.2017

[<http://graphics.berkeley.edu/>]

[1] <http://graphics.berkeley.edu/papers/Kirk-SPE-2005-06/thumb.jpg>; 24.09.2017

[<http://nintendo-online.de/>]

[1] http://nintendo-online.de/upload/images/2014/01/15/i_70997175252d66928a46313-92791043.jpg; 24.09.2017