

Starrkörpersimulation

XMAMan

24.7.2024

Hier wird der Versuch unternommen eine Erklärung dafür zu finden, wie Erin Cattos Box2D eigentlich funktioniert.

Inhaltsverzeichnis

Teil 0: Intro.....	8
Teil 1: Rechteck und Kreis ohne Gravitation und Reibung.....	9
Einleitung.....	9
Ziel dieses Abschnittes.....	9
Was ist ein Starrkörper?.....	9
Zusammenfassung der Symbole/Einheiten von Starrkörpern.....	14
Unbeschränkte Bewegung von Starrkörpern.....	14
Kollision zwischen Starrkörpern.....	15
Kollision zwischen zwei Kreisen.....	16
Fall 1: CircleCircle_Overlapping_FoundsCollisionPoint.....	16
Fall 2: CircleCircle_SamePosition_FoundsCollisionPoint.....	17
Kollision zwischen Rechteck und Kreis.....	17
Fall 1: RectangleCircle_CircleCenterIsInside_FoundCollisionPoint.....	18
Fall 2: RectangleCircle_CircleHitCorner1_FoundCollisionPoint.....	19
Fall 3: RectangleCircle_CircleHitCorner2_FoundCollisionPoint.....	20
Fall 4: RectangleCircle_CircleHitRightFace_FoundCollisionPoint.....	21
Kollision zwischen zwei Rechtecken.....	21
SAT – Separating Axis Theorem.....	21
Support-Punkt für eine Flächennormale.....	22
Die Achse mit der kleinsten Eindringtiefe.....	22
Algorithmus zur Kollisionserkennung zwischen zwei Rechtecken.....	23
Probleme mit dem Rechteck-Kollisionsalgorithmus aus dem „Building a 2D Game Physics Engine“-Buch	24
Überarbeitung des Rechteck-Kollisionsalgorithmus.....	25
Die Kollisionserkennung von Rechteck-Rechteck bei Box2D-Lite.....	27
Kollisionen auflösen.....	28
Collision-Resolution 1 – PositionCorrection.....	28
Collision-Resolution 2 – Iterativer Impuls.....	29
Herleitung der Impuls-Formel.....	29
Erklärung des Impulses am Quelltext.....	33
Anwendung des Impulses als Schleife vs Einmal pro Kontaktpunkt.....	34
So gut funktioniert die Kollisionsauflösung über den iterativen Impulsansatz.....	35
Zusammenfassung von Teil 1.....	36
Quellen von Teil 1.....	37
Teil 2: Bewegungsgleichung als Matrix.....	38
Ziel dieses Abschnitts.....	38
Herleitung der Formel.....	38
Das Problem mit dem Würfel der an der Wand ohne Drehung abprallen soll.....	38
Bewegungsgleichungen als Matrix.....	38
Die Normal-Constraint-Gleichung.....	40
Die Jacobi-Matrix J.....	41
Der Lambda-Spaltenvektor.....	42
Constraint-Bias.....	43
Beschränkung der Constraint-Kraft.....	44
Bestimmung der Constraint-Kräfte.....	44
Bestimmung der neuen Geschwindigkeitswerte unter Nutzung der Constraint-Kraft.....	46
Zusammenfassung wie wir über Matrizen und PGS Lambda und V ² ermitteln.....	46
Was genau ist denn nun ein Constraint? Wie kann man sich das vorstellen?.....	47
Gleichungssystem per (Projected) Gauss-Seidel lösen.....	48
Gauss-Seidel.....	48

Projected Gauss-Seidel (PGS).....	48
Testfälle ohne Schwerkraft.....	49
Würfel gegen die Wand.....	49
Kugel gegen zwei Kugeln.....	49
Würfel gegen zwei Würfel.....	50
Gruppieren ist nicht immer die Lösung.....	51
Das Kontaktpunkt-Gruppierungsproblem.....	51
Erweiterung um Schwerkraft.....	52
Testfälle wo Gegenstände auf ein Tisch fallen.....	52
Korrektur der RectangleRectangle-Collision.....	53
Erweiterung um PositionCorrection.....	54
Ball mit Restitucion von 0.....	54
InvDt*C.Depth.....	55
AllowedPenetration.....	55
PositionalCorrectionRate.....	56
Ball mit Restitution von 1.....	57
DoPositionalCorrection-Schalter.....	57
Zusammenfassung – Wo darf man PositionCorrection einsetzen und wo nicht.....	58
Testfälle, wo PositionCorrection zwingend nötig ist.....	58
Testfälle, wo man Position-Correction nicht einsetzen darf.....	58
Erweiterung um Reibung.....	58
Colliding-Contact vs Resting-Contact.....	59
Die Konvergenzgeschwindigkeit von Projected Gauss-Seidel.....	60
Zusammenfassung von Teil 2.....	61
Offene Punkte / Ausblick.....	61
Quellen für Teil 2.....	61
Teil 3: Sequentielle Impulse.....	62
Ziel dieses Abschnitts.....	62
Matrix um Warmstart erweitern.....	62
Eine J-Zeile als Objekt.....	65
Sequentielle Impulse.....	68
Herleitung der Gesamtformel.....	68
Kraftgleichung für ein einzelnen Körper.....	68
NormalConstraint.....	69
FrictionConstraint.....	70
Constraint-Gleichung als Ebene.....	71
Herleitung des Constraint-Impulses.....	71
Sequentielle Impulse im Vergleich zu Iterative Impulse.....	75
Testfall – CubeAgainstWall – Ohne Reibung.....	76
Testfall – CubeAgainstWall – Mit Reibung.....	78
Accumulated Impulse.....	79
AccumulatedImpulse mal mit und ohne Min-Max-Schranke.....	85
Erweiterung des SI-Solvers um externe Kraft.....	85
Erweiterung des SI-Solvers um WarmStart.....	87
Constraint-Kraft und V2 berechnen – SI vs Matrix/JRow.....	88
Die unterschiedliche Behandlung der externen Kraft.....	89
Vergleich der verschiedenen CollisionResolution-Solver.....	90
Das Newton-Pendel	91
Constraint-Kraft vs Constraint-Impuls.....	92
Zusammenfassung von Teil 3.....	93
Quellen für diesen Abschnitt	93
Teil 4: Soft- und Multiconstraints.....	94

Ziel dieses Abschnitts.....	94
Cleanup & Refactoring.....	94
Die effektive Masse.....	94
Distance-Constraint ohne Feder (Metallstab).....	95
Distance-Constraint als gedämpfte Schwingung (Eisenfeder).....	97
Die Kraftgleichung der gedämpften Feder.....	97
Federsimulation als externe Kraft.....	97
Vergleich von verschiedenen Verfahren zur Federsimulation.....	98
Feder ohne Dämpfung – Simulation über explizite Formel	98
Feder ohne Dämpfung – Simulation über explicit Euler.....	99
Feder ohne Dämpfung – Simulation über implizit Euler.....	99
Feder ohne Dämpfung – Simulation über Semi-Implicit-Euler.....	100
Federsimulation über SoftConstraints.....	102
Softconstraint-Gleichung aus der Feder-Kraftformel herleiten.....	102
Herleitung für Beta und Gamma.....	103
Schritt 1: v2 ermitteln indem für F die Federkraft von Seite 11 eingesetzt wird.....	103
Schritt 2: Die Constraint-Kraft in die v2-Gleichung einsetzen.....	104
Per Parametervergleich Beta und Gamma ablesen.....	104
Die Ermittlung von Lambda bei Softconstraints.....	105
Den Impuls über Gleichung (2.13) ausrechnen.....	106
Den Impuls über Gleichung (2.17) ausrechnen.....	107
Nutzerfreundlichere Konstanten zur Federkonfiguration.....	109
Feder ohne Dämpfung	110
Feder ohne Dämpfung – Simulation über Semi-Implicit-Euler mit Softconstraint.....	111
Feder ohne Dämpfung – Semi-Implicit-Euler - Zusammenfassung.....	114
Zusammenfassung – Die gedämpfte Feder.....	116
Mouse Joint (Objekt per Maus verschieben).....	118
Maus-Constraint ohne Verwendung von SoftConstraint.....	118
Herleitung der Maus-Constraint-Formel.....	118
Umsetzung der MouseConstraint ohne Softconstraints.....	121
Maus-Constraint mit Verwendung von SoftConstraint.....	122
Zusammenfassung von Teil 4.....	123
Quellen für diesen Abschnitt	124
Teil 5: Joints.....	125
5.1 Nutzung der Joints im Editor.....	125
Wheel Joint (Gardinenstange) (Slider bei Unity3D).....	125
Prismatic Joint (Stab in Hülse).....	125
Revolute Joint (Drehgelenk) (Hinge Constraint bei Unity).....	126
Weld Joint (Full Position Constraint).....	126
Distance Joint.....	126
5.2 Herleitung der Joint-Constraints.....	127
Ziel dieses Abschnitts.....	127
Cleanup & Refactoring.....	127
BasisConstraints für die Gelenke.....	128
Mathevorwissen, was für die Constraint-Herleitungen benötigt wird.....	129
PointToLine-Constraint.....	131
Variante 1: $d=x_2+r_2-x_1$	131
Variante 2: $d=x_2+r_2-x_1-r_1$	132
MinMaxTranslation-Constraint.....	133
TranslationMotor-Constraint.....	135
FixAngular-Constraint.....	135
PointToPoint-Constraint.....	137

MinMaxAngular-Constraint.....	139
AngularMotor-Constraint.....	140
Kombination von Basisconstraints.....	141
PointToLine und FixAngular als MultiConstraint.....	141
PointToPoint und FixAngular als Multiconstraint.....	143
Die Joints soft machen.....	145
Der J-Vektor und die Forcedirection.....	147
Zusammensetzung der Joints.....	148
Zusammenfassung von Teil 5.....	148
Teil 6 : Polygone.....	150
Ziel dieses Abschnitts.....	150
Physic-Editor.....	151
Snap Anchor-Points.....	151
Physik-Simulator.....	151
Print-Settings.....	151
Physic-Engine.....	152
Force Visualisation – Zug und Druckkräfte.....	152
Schubdüsen.....	154
Axial Friction.....	155
Friction-Constraint.....	155
CollisionMatrix.....	156
Polygon.....	157
PointIsInside-Check bei Polygonen (Konvex/Konkav).....	158
Fläche eines Polygons über die Trapezformel.....	159
Fläche eines Polygons über das Kreuzprodukt.....	161
Schwerpunkt eines Polygons.....	162
Inertia eines Polygons.....	163
Konkaves Polygon in konvexe Polygone zerlegen.....	166
Prüfen das ein Polygon korrekt erstellt wurde.....	168
Die beiden Polygonarten aus Sicht der Kollisionerkennung.....	168
Schnittpunkt bei ein EdgePolygon.....	169
Schnittpunkt PolygonEdge – Circle.....	169
Schnittpunkt PolygonEdge – konkavses Rigid-Polygon.....	170
Schnittpunkt bei ein konkaven Rigid-Polygon.....	173
Schnittpunkt konkavses Polygon mit Kreis.....	173
Schnittpunkt konkavses Polygon mit Rechteck / anderen konkavsen Polygon.....	174
Broad-Phase Kollisionserkennung.....	177
Zusammenfassung von Teil 6.....	178
Teil 7: Leveleditor.....	179
7.1 Die Teilkomponenten des Editors.....	179
Ziel dieses Abschnitts.....	179
Konzept für den Leveleditor.....	179
7.1.1 Texture Editor.....	181
Für was ist der Editor?.....	181
Anwendungsbeispiel – Davefigur.....	181
7.1.2 Animationseditor.....	187
Beispiel 1: OneTime-Animation.....	187
Beispiel 2: AutoLoop-Animation.....	193
Beispiel 3: Manuelle Animation.....	198
7.2 Spiele direkt im Editor.....	199
Grundaufbau des Editors.....	199
Beispiel 1: Der Ski-Fahrer.....	200

Beispiel 2: Die Abrisskugel.....	221
Beispiel 3: Das Raumschiff.....	228
Beispiel 4: Auto was man von oben sieht.....	235
Einstellmöglichkeiten für die Kamera im Leveleditor.....	240
7.3 Leveleditor aus Programmiersicht.....	242
KeyFrameEditor.....	242
TextureEditor.....	243
Leveleditor.....	245
Erzeugung des LevelEditorViewModels im UnitTest.....	247
Unterprojekte.....	248
Zusammenfassung von Teil 7.....	248
Teil 8: Sprites.....	249
Ziel dieses Abschnitts.....	249
Spriteeditor.....	249
Beispiel: Motorrad-Fahrer.....	249
Anzeige von Sprites.....	253
1 - Objekt im Editor unsichtbar machen.....	253
2 - Anzeige von ein einzelnen Objekt als Sprite.....	254
3 - Anzeige von mehreren Objekten als Sprite.....	254
Teil 9: Dynamische Objekterzeugung.....	256
Ziel dieses Abschnitts.....	256
Objekte zerlegen.....	256
Objekte in Kästchen zerlegen.....	257
Rotierte Bounding-Box.....	258
Schnittmenge zwischen zwei Polygonen.....	259
Objekte in Kästchen oder per Voronoi zerlegen.....	264
Verwendung des Objekt-Destroyer.....	265
Objekte duplizieren.....	265
Teil 10: Example-Games.....	272
Ziel dieses Abschnitts.....	272
Game 1 – Moonlander.....	272
Verwendung der GameSimulator-Klasse.....	272
Schubdüse mit Feuerpartikeln.....	273
Raumschiff zerstören, wenn es zu hart aufsetzt.....	274
Game 2 – Skijumper.....	275
Prüfen, dass der Skispringer verletzungsfrei die Fahne erreicht hat.....	275
Game 3 – Elma.....	276
Sprites über externen Editor erzeugen.....	276
Sprite aus dem Spiel heraus erzeugen.....	277
Sprite anzeigen.....	278
Objekt rein grafisch zerlegen.....	278
Nutzung des Leveleditors innerhalb des Spiels.....	279
Game 4 – Astroids.....	281
Rechteck während der Simulation erstellen.....	281
Polygon dynamisch erzeugen und prüfen ob es mit der Szene kollidiert.....	281
Komplexes Objekt dynamisch erzeugen.....	281
Kollision zwischen dynamisch erzeugten Objekten ermitteln.....	282
Game 5 – CarDrifter.....	283
Erstellung eines TopDown-Games	283
Innerhalb vom Leveleditor das Spiel anzeigen.....	284
Game 6 – Bridge Builder.....	285
Leveleditor-Objekt in Nutzergenerierter Szene platzieren.....	285

Ein WPF-Button auf dem GraphicPanel platzieren.....	285
Dynamisch erzeugtes Level im Leveleditor ansehen.....	286
Game 7 – SpiderBox.....	287
Joints dynamisch erzeugen.....	287
Physiksimulation anstelle von Sprites.....	288
Einzelne Exedatei erstellen.....	289
Teil 11: Zusammenfassung.....	290

Teil 0: Intro

Will man ein Spiel schreiben, wo die Objekte sich physikalisch korrekt bewegen, dann braucht man eine Physikengine. Googelt man nach „2d physics engine“ findet man recht schnell von Erin Catto sein Box2D. Will man verstehen wie Box2D intern funktioniert dann wird man auch recht schnell auf Box2D-Lite stoßen wo man dann solche Sachen hier findet:

<https://github.com/erincatto/box2d-lite/blob/master/src/Joint.cpp>

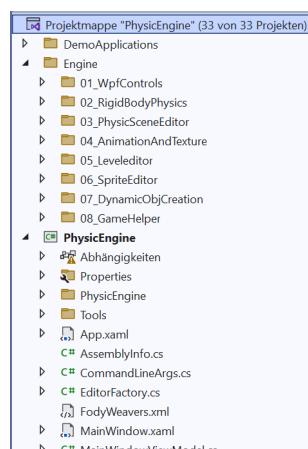
```
44         // deltaV = deltaV0 + K * impulse
45         // invM = [(1/m1 + 1/m2) * eye(2) - skew(r1) * invI1 * skew(r1) - skew(r2) * invI2 * skew(r2)]
46         //      = [1/m1+1/m2      0      ] + invI1 * [r1.y*r1.y -r1.x*r1.y] + invI2 * [r1.y*r1.y -r1.x*r1.y]
47         //      [      0      1/m1+1/m2]           [-r1.x*r1.y r1.x*r1.x]           [-r1.x*r1.y r1.x*r1.x]
70     if (World::positionCorrection)
71     {
72         bias = -biasFactor * inv_dt * dp;
73     }
79     if (World::warmStarting)
80     {
81         // Apply accumulated impulse.
82         body1->velocity -= body1->invMass * P;
83         body1->angularVelocity -= body1->invI * Cross(r1, P);
84
85         body2->velocity += body2->invMass * P;
86         body2->angularVelocity += body2->invI * Cross(r2, P);
87     }
```

Der eigentliche Kern von Box2D hat nicht sonderlich viele Zeilen aber für ein Neuling, der sich dem Thema nähert ist das schwer zu verstehen, was diese Formeln überhaupt bedeuten.

Abschnitt 1 bis 6 erklärt, wie die Formeln von Box2D funktionieren. Der Rest vom Dokument erweitert die hier erstellte Physikengine um Texturen, Animationen und Beispielanwendungen.

Es gibt 10 Abschnitte. Hier sind die Pfade, wo jeweils für jeden Abschnitt aus dem Dokument hier der zugehörige Quelltext liegt:

```
1 -> RigidBodyPhysics\PhysicEngine\Dokumentation\Part1 - Iterative Impulse\ExampleCode
2 -> RigidBodyPhysics\PhysicEngine\Dokumentation\Part2 - Enter the Matrix\ExampleCode
3 -> RigidBodyPhysics\PhysicEngine\Dokumentation\Part3 - Sequential Impulse\ExampleCode
4 -> RigidBodyPhysics\PhysicEngine\Dokumentation\Part4 - Soft- and Multi-Constraints\ExampleCode
5 -> RigidBodyPhysics\PhysicEngine\Dokumentation\Part5 - Joints\ExampleCode
6 -> RigidBodyPhysics\PhysicEngine\Dokumentation\Part6 - Polygons
7 -> RigidBodyPhysics\PhysicEngine\Source          -> Engine\05_Leveleditor
8 -> RigidBodyPhysics\PhysicEngine\Source          -> Engine\06_SpriteEditor
9 -> RigidBodyPhysics\PhysicEngine\Source          -> Engine\07_DynamicObjCreation
10-> RigidBodyPhysics\PhysicEngine\Source         -> DemoApplications
```

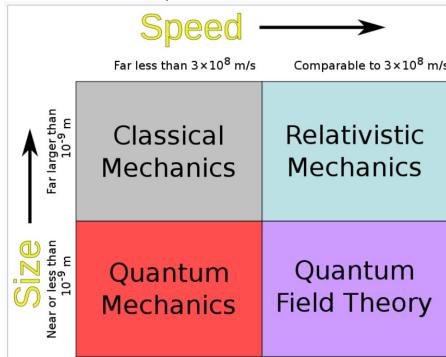


Das Hauptprogramm liegt unter RigidBodyPhysics\PhysicEngine\Source wo dann die Teilprojekte jeweils erklärt werden.

Teil 1: Rechteck und Kreis ohne Gravitation und Reibung

Einleitung

Hier soll die 2D-Starrkörpersimulation erklärt werden. Es grenzt sich ab zu Softbody-Simulation, Partikelsimulation, Fluidsimulation und Federsimulation. Sie funktioniert nach den Gesetzen der klassischen Mechanik (3 Regeln von Newton).



Quelle: https://en.wikipedia.org/wiki/Classical_mechanics

Man versteht das Thema der Starrkörpersimulation (<https://de.wikipedia.org/wiki/Starrk%C3%B6rpersimulation>) besser, wenn man mit ein einfachen Beispiel beginnt und dieses dann Schrittweise immer weiter ausbaut/verbessert. Deswegen beschreiben die ersten 6 Kapitel von diesen Dokument jeweils ein Entwicklungsstand der Physikengine. So versteht man dann besser das Endergebnis, wenn man vorher die Versuche/Irrwege auf den Weg dort hin verstanden hat.

Ziel dieses Abschnittes

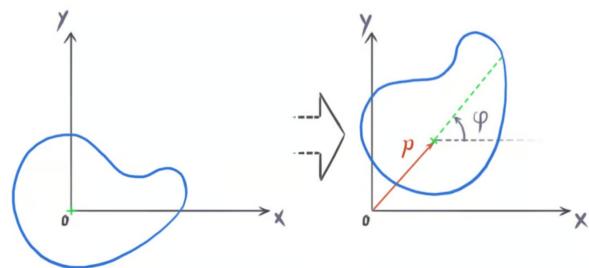
Es soll hier die Bewegung von Rechtecken und Kreisen simuliert werden, welche ohne Reibung und Schwerkraft sich bewegen und dabei kollidieren können. Dazu müssen zuerst eine paar Physik-Grundbegriffe geklärt werden, bevor es an die Algorithmen geht.

Was ist ein Starrkörper?

Ein Starrkörper ist ein Objekt welches sich nicht verformt. Das kann z.B. eine Eisenstange oder ein Steinblock sein. Wenn ich einen Starrkörper in die Luft werfe, dann wird er sich immer nur um sein Schwerpunkt drehen. Wenn ich die Bewegung von einem Starrkörper, der durch die Luft fliegt, simulieren will, dann brauche ich die Position des Schwerpunktes $x(t)$ (oder $p(t)$), die Geschwindigkeit $v(t)$ vom Schwerpunkt, seine Ausrichtung $\varphi(t)$ und seine Winkelgeschwindigkeit $\omega(t)$.

So wird der aktuelle Zustand von einem Starrkörper beschrieben:

```
class RigidBody
{
    Vector2 Position;           //x
    Vector2 LinearVelocity;     //v
    float Angle;                //Phi
    float AngularVelocity;      //Omega
}
```



Ein Starrkörper kann eine beliebige Form haben. Es wäre sogar erlaubt, dass er aus mehreren Einzelobjekten besteht, welche unsichtbar miteinander verbunden sind. So ein Körper kann man sich einfach nur als eine Ansammlung von Massepunkten vorstellen.

Massen vom Körper

Die Gesamtmasse vom Körper wäre die Summe seiner Massepunkte und berechnet sich über

$$M = \sum_{i=1}^N m_i$$

Quelle: Rigid Body Simulation – David Baraff Abschnitt 2.4

Schwerpunkt des Körpers

Der Schwerpunkt ist dann die gewichtete Summe seiner Massepunkte-Positionen:

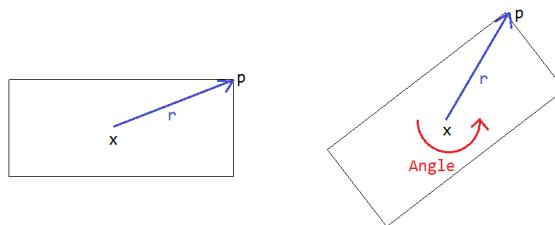
$$\frac{\sum m_i r_i(t)}{M}$$

mi ist hier die Masse eines Punktes und ri seine Position.

Quelle: Rigid Body Simulation – David Baraff Abschnitt 2.6

Die Position-Property von der RigidBody-Klasse muss immer der Schwerpunkt des Körpers sein!
Bei ein Kreis/Rechteck mit konstanter Dichte wäre das die Mitte.

Position eines Punktes von ein Körper



Ich kann die Position eines Partikels im ungedrehten Körper über $p=x+r$ angeben und im gedrehten Körper über $p=x+R(\text{Angle})*r$ mit der Rotationsmatrix:

$$R(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

Quelle1: <https://www.toptal.com/game/video-game-physics-part-iii-constrained-rigid-body-simulation>

Quelle2: Rigid Body Simulation – David Baraff Abschnitt 2.1

Geschwindigkeit eines Punktes von ein Körper

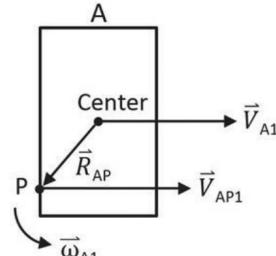
Die Geschwindigkeit eines Partikels ist die Ableitung der Position nach der Zeit. Das ist die Summe aus der Zentrumsgeschwindigkeit v und dem Hebelarm Kreuz Winkelgeschwindigkeit.

$$\dot{p} = v + \omega \times r \quad \text{mit} \quad \omega = (0,0, \text{AngularVelocity})$$

Quelle 1: Rigid Body Simulation – David Baraff Abschnitt 2.3 und 2.5

Quelle 2: <https://www.myphysicslab.com/engine2D/collision-en.html>

Wenn sich der Körper A mit Omega A1 um sein Zentrum dreht, dann bewegt sich der Punkt P mit folgender Geschwindigkeit: $\vec{v}_{AP1} = \vec{v}_{A1} + (\vec{\omega}_{A1} \times \vec{R}_{AP})$



Quelle: Building a 2D Physics Engine Seite 168

Trägheitstensor (Moment of inertia) eines Körpers; Symbol=I; Einheit: [kg*m²]

Wenn ich ein Körper beschleunigen will, dann gibt die Masse an, wie sehr er sich gegen die Beschleunigung wehrt.

Wenn ich einen Körper um eine Achse drehen will, dann gibt sein Trägheitsmoment an, wie sehr er sich gegenüber der Drehung um eine bestimmte Achse wehrt.

Bei 2D-Objekten habe ich nur die Z-Achse über die ich drehen kann und es soll immer nur um den Schwerpunkt gedreht werden. Somit ist das Inertia bei 2D-Körpern eine einzelne skalare Zahl I:

$$I = \int_V \rho(\mathbf{r}) \mathbf{r}^2 dV$$

p(r)=Dichte; r=Kleinster Abstand zur Drehachse

Quelle1: <https://www.toptal.com/game/video-game-physics-part-i-an-introduction-to-rigid-body-dynamics>

Quelle2: https://en.wikipedia.org/wiki/Moment_of_inertia#Motion_in_a_fixed_plane

Beispiel: Ich berechne von einem Rechteck mit konstanter Dichte c dessen Zentrum bei (0,0) liegt und was die Breite w und Höhe h hat dessen Inertia:

$$\begin{aligned}
I &= \int_{-\frac{w}{2}}^{\frac{w}{2}} \int_{-\frac{h}{2}}^{\frac{h}{2}} c * (x^2 + y^2) dx dy & I &= \int_{-\frac{h}{2}}^{\frac{h}{2}} [c * \frac{1}{3} x^3 + c * x * y^2]_{-\frac{w}{2}}^{\frac{w}{2}} dy \\
I &= \int_{-\frac{h}{2}}^{\frac{h}{2}} (c * \frac{1}{3} \frac{w^3}{8} + c * \frac{w}{2} * y^2) - (-c * \frac{1}{3} \frac{w^3}{8} - c * \frac{w}{2} * y^2) dy & I &= c * \int_{-\frac{h}{2}}^{\frac{h}{2}} (\frac{1}{3} \frac{w^3}{4} + w * y^2) dy \\
I &= c [\frac{w^3}{12} * y + \frac{1}{3} w * y^3]_{-\frac{h}{2}}^{\frac{h}{2}} & I &= c ((\frac{w^3}{12} * \frac{h}{2} + \frac{1}{3} w * \frac{h^3}{8}) - (\frac{-w^3}{12} * \frac{h}{2} - \frac{1}{3} w * \frac{h^3}{8})) \\
I &= c (\frac{w^3}{12} * h + \frac{h^3}{12} * w) \quad \text{mit} \quad m = \int_0^w \int_0^h c * dx dy & \frac{m}{c} &= w * h \quad c = \frac{m}{w * h} \\
I &= m * (\frac{w^2}{12} + \frac{h^2}{12}) = m * \frac{w^2 + h^2}{12}
\end{aligned}$$

Hier sind Standardformeln für verschiedene Körper um ihr Inertia zu berechnen:

https://en.wikipedia.org/wiki/List_of_moments_of_inertia

Linearimpuls eines Körpers; Symbol: P; Einheit: [N*s=kg*m*s^-1]

Sowohl für ein Partikel als auch ein Körper berechnet sich der Linearimpuls über

$$P=m*v$$

m ist die Masse des Körpers/Partikels und v die Geschwindigkeit des Zentrums/Partikels

Quelle: <https://de.wikipedia.org/wiki/Impuls>

Eine weitere Definition vom Impuls ist, dass wenn eine Kraft für eine kurze Zeit wirkt, dann ist das auch ein Impuls:

$$P = \int F dt$$

Integral der Kraft über die Zeit:

Wenn die Kraft F konstant während der Zeit h wirkt gilt: $P = hF$

Quelle: <https://www.toptal.com/game/video-game-physics-part-iii-constrained-rigid-body-simulation>

Eigendrehimpuls (Angular Momentum) eines Körpers; Symbol: L; Einheit: [N*m*s=kg*m^2*s^-1]

Ähnlich wie bei der „P=m*v“-Formel ist der Drehimpuls für ein 2D-Körper der um sein Zentrum rotiert wird:

$$L = I\omega$$

I ist der Trägheitstensor bezüglich der Z-Achse die durch den Masseschwerpunkt geht

Omega ist die Winkelgeschwindigkeit des Körpers.

Quelle1: https://en.wikipedia.org/wiki/Angular_momentum#Definition_in_classical_mechanics

Quelle2: https://de.wikipedia.org/wiki/Drehimpuls#Der_Eigendrehimpuls

Linearkraft eines Körpers Symbol: F; Einheit: [N=kg*m*s^-2]

Newton's zweites Gesetz besagt:

$$\vec{F} = m\vec{a}$$

Ich kann es aber auch über die Ableitung des Impulses nach der Zeit schreiben:

$$\vec{F} = \dot{\vec{p}}$$

Quelle: https://de.wikipedia.org/wiki/Newton'sche_Gesetze#Zweites_Newton'sches_Gesetz

Drehmoment (Torque) eines Körpers Symbol: Tau oder M; Einheit: [N*m]

Wenn ich ein 2D-Körper habe, welcher sich mit einer bestimmten Geschwindigkeit um sein Schwerpunkt dreht, dann könnte ich auf der Z-Achse, welche am Schwerpunkt befestigt ist den Körper anhalten. Dazu benötige ich eine Kraft (Drehmoment genannt), welche sich aus der Ableitung des Drehimpulses nach der Zeit ergibt:

$$\tau = I\alpha$$

I=Trägheitstensor im Bezug zur Z-Achse welche durch den Schwerpunkt geht

Alpha = Winkelbeschleunigung des Körpers

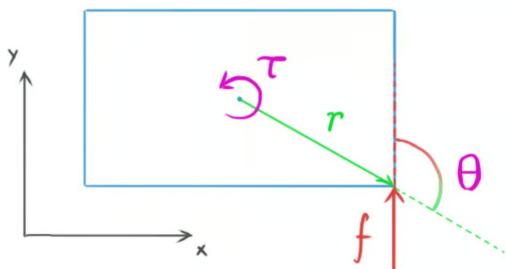
Quelle1: https://en.wikipedia.org/wiki/Moment_of_inertia#Definition

Quelle2: https://de.wikipedia.org/wiki/Drehmoment#Drehmoment_einer_Kraft_bez_%C3%BCglich_einer_Achse

Wenn eine Kraft bei ein Körper an ein bestimmten Punkt angreift, dann lässt dass den Körper rotieren und das Drehmoment was dadurch auf die Z-Zentrumsachse wirkt ist:

$$\tau = \|r\| \|f\| \sin \theta$$

$$\tau = \mathbf{r} \times \mathbf{f}$$



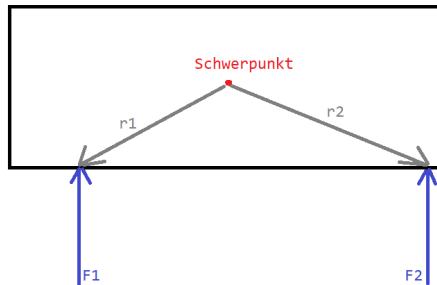
Torque = $r.x * f.y - r.y * f.x$ (Siehe ComputeForceAndTorque-Funktion bei Torque)

Quelle 1: <https://en.wikipedia.org/wiki/Torque>

Quelle 2: <https://www.toptal.com/game/video-game-physics-part-i-an-introduction-to-rigid-body-dynamics>

Kraft auf Punkt bei Körper – Translationskraft und Rotationskraft

Wenn ich mehrere Kräfte habe, die auf ein Körper wirken, dann muss ich die Summe der Kräfte nehmen und lasse das dann direkt aufs Zentrum wirken. Über die Hebelarme zum Zentrum bekomme ich die Drehmomente.



$$\text{RigidBody.Force} = F_1 + F_2$$

$$\text{RigidBody.Torque} = r_1 \times F_1 + r_2 \times F_2$$

Quelle 1 für die Regel, dass jede Kraft auf zwei Arten(Force/Torque) wirkt:

Rigid Body Simulation - David Baraff 2001.pdf Abschnitt 2.7

Quelle 2: <https://www.toptal.com/game/video-game-physics-part-i-an-introduction-to-rigid-body-dynamics> → Funktion ComputeForceAndTorque

Diese Kraft-Regel kann ich dazu benutzen, um ein Starrkörper dadurch zu bewegen, indem ich an ein oder mehreren Angriffspunkten gegen den Körper drücke/ziehe. Das könnte z.B. eine Schubdüse sein, die am Körper an ein Punkt befestigt ist und den Körper damit bewegt.

Wenn die aufsummierte Kraft für einen Zeitraum von dt Sekunden wirkt, dann ergibt sich mit folgender Formel dann die neue Geschwindigkeit und Position des Körpers.

$$\text{RigidBody.LinearVelocity} += \text{RigidBody.Force} / \text{RigidBody.Mass} * dt$$

$$\text{RigidBody.AngularVelocity} += \text{RigidBody.Torque} / \text{RigidBody.Inertia} * dt$$

$$\text{RigidBody.Position} += \text{RigidBody.LinearVelocity} * dt$$

$$\text{RigidBody.Angle} += \text{RigidBody.AngularVelocity} * dt$$

Zusammenfassung der Symbole/Einheiten von Starrkörpern

Translation			Rotation		
Position	$p=(x,y)$ \bar{s}	[m]	Drehwinkel	Angle $\bar{\varphi}$	[0..2 Pi]
Geschwindigkeit	$\bar{v} = \frac{\Delta \bar{s}}{\Delta t}$	[m/s]	Winkelgeschwindigkeit	$\bar{\omega} = \frac{\Delta \bar{\varphi}}{\Delta t}$	[rad/s]
Beschleunigung	$\bar{a} = \frac{\Delta \bar{v}}{\Delta t}$	[m/s ²]	Winkelbeschleunigung	$\bar{\alpha} = \frac{\Delta \bar{\omega}}{\Delta t}$	[rad/s ²]
Masse	m	[kg]	Trägheitsmoment	$I = \int_V \rho(r) r^2 dV$ I=J	[kg*m ²]
Impuls	$\bar{p} = m \cdot \bar{v}$	[kg*m/s]	Drehimpuls	$L = I\omega$	[kg*m ² /s]
Impulssatz / Grundgleichung	$\vec{F} = \dot{\vec{p}} = m\vec{a}$ $P = \int F dt$		Drehimpulssatz	$\bar{M} = \dot{\bar{L}} = J\bar{\alpha}$ $\Delta \bar{L} = \bar{M} \cdot \Delta t$	
Kraft	$\vec{F} = m\vec{a}$	[kg*m/s ²]	Drehmoment	$\tau = I\alpha$ Tau=M	[N*m]
Translationsenergie	$E_{kin} = \frac{1}{2} m \cdot v^2$	[J]	Rotationsenergie	$E_{rot} = \frac{1}{2} I \cdot \omega^2$	[J]

Unbeschränkte Bewegung von Starrkörpern

Bis jetzt wurden die Physik-Grundbegriffe geklärt. Ab hier macht es nun Sinn den für Abschnitt 1 mitgelieferten Beispielquellcode unter *PhysicEngine\Documentation\Part1 - Iterative Impulse\ExampleCode\Source* zu öffnen und jeweils die Funktion zu öffnen, welche jeweils erklärt wird.

Wenn ich die Bewegung von Körpern simulieren will, welche nicht kollidieren und wo auch keine Kräfte wirken dann reicht es, wenn ich einmal initial ein Geschwindigkeitswert für Velocity und AngularVelocity festlege und dann bewegt er sich (Zeile 71/72) einfach nur Schrittweise durch den Raum.

```

76   private void MoveBodys(float dt)
77   {
78     foreach (var body in this.bodies)
79     {
80       body.MoveCenter(dt * body.Velocity);
81       body.Rotate(dt * body.AngularVelocity);
82       body.Force = new Vector2D(0, 0);
83       body.Torque = 0;
84     }
85   }
86 }
```

```

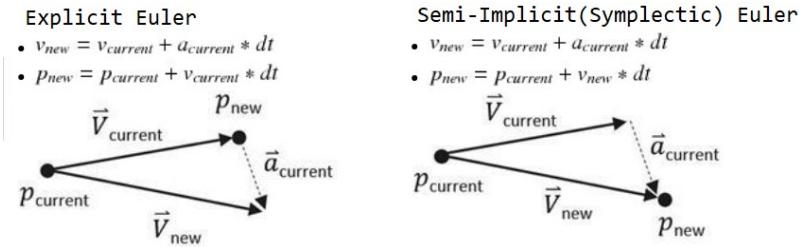
53   public void MoveCenter(Vector2D v)
54   {
55     this.Center += v;
56   }
57   2 Verweise
58   public void Rotate(float angle)
59   {
60     this.Angle += angle;
61   }

```

Siehe: PhysicEngine/PhysicScene.cs

Siehe: PhysicEngine/RigidBody/RigidCircle.cs

Es gibt zwei Möglichkeiten wie man die Position pro Zeitschritt verändern kann:



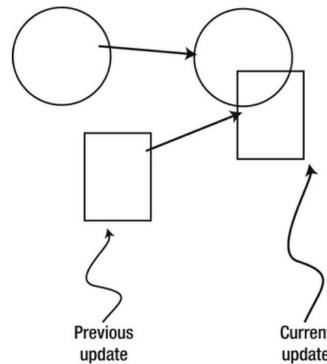
Quelle: Building a 2D Game Physics Engine Seite 129

Explicit Euler ist numerisch instabil. Ein Ball wackelt dann immer leicht, wenn er auf dem Tisch liegt. Deswegen nehmen wir hier Semi-Implicit Euler.

Selbstverständlich wäre eine Kollisionsfreie Bewegung unrealistisch weswegen wir uns nun ansehen wie man eine Kollision zwischen zwei Starrkörpern berechnet.

Kollision zwischen Starrkörpern

Pro Zeitschritt bewegt sich ein Körper um eine gewisse Distanz. Dabei kann es passieren, dass ein Objekt in ein anderes Objekt rein springt. Beispiel: Im ersten Zeitschritt kollidieren der Kreis und das Rechteck noch nicht. Im nächsten Zeitschritt kollidieren sie dann.



Damit zwei Körper bei ihrer Bewegung durch den Raum nicht ineinander eindringen ist es nötig, dass man eine Kollision zwischen zwei Körpern feststellen kann. Stellt man fest, dass zwei Körper sich überschneiden (Kollidieren) dann benötigt man für beide Körper jeweils ein Kontaktspunkt, an welchen dann eine Kraft bzw Impuls wirkt, welcher die Körper dann wieder auseinander drückt und womit die Kollision aufgelöst/beseitigt wird.

Ziel vom Kollisionsalgorithmus zweier Objekte ist es ein Datenobjekt zu erzeugen was so aussieht:

```
public class CollisionInfo
{
    public Vector2D Start; //Collisionpoint from RigidBody1
    public Vector2D End; //Collisionpoint from RigidBody2
    public Vector2D Normal; //Normal from Start-Point
    public float Depth; //Distanz between Start and End
}
```

Es gibt keine allgemeine Regel welche festlegt, wo diese Kontaktspunkte bei einer Kollision sein sollen. Was aber gilt: Die Kollision soll auf kürzesten Weg beseitigt werden. Die Körper werden in Richtung Kontaktnormalen voneinander um die Distanz Depth weggedrückt. Danach berühren sie sich exakt am Randpunkt.

Die Kollisionsabfrage unterteilt sich in eine Broad- und Nearphase. Bei der Broadphase wird geschaut, welches Objekt mit welchen anderen kollidieren könnte. Bei der NearPhase wird dann

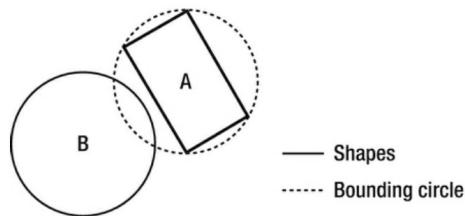
genau geprüft, ob es eine Kollision gab und das CollisionInfo-Objekt erzeugt.

```

6   public static class CollisionHelper
7   {
8       9 Verweise | 7/7 bestanden
9       public static RigidbodyCollision[] GetAllCollisions(List<IRigidBody> bodies)
10      {
11          List<RigidbodyCollision> collisions = new List<RigidbodyCollision>();
12
13          for (int i = 0; i < bodies.Count; i++)
14              for (int j = i + 1; j < bodies.Count; j++)
15              {
16                  var b1 = bodies[i];
17                  var b2 = bodies[j];
18                  if (BoundingCircleTest.Collide(b1, b2))
19                  {
20                      var collision = b1.CollideWith(b2); ← NearPhase
21                      if (collision != null)
22                          collisions.Add(new RigidbodyCollision(collision, b1, b2));
23
24
25      }
26  }
27 }
28 }
```

Siehe: PhysicEngine/CollisionDetection/CollissionHelper.cs

So sieht bei der BroadPhase der BoundingCircleTest aus:



In den nächsten Abschnitten werden nun verschiedene Nearphase-Kollisionsroutinen erklärt. D.h. Der BoundingCircleTest von der BroadPhase hat bereits ermittelt, dass es eine Kollision zwischen zwei Körpern geben könnte und die Nearphase-Funktion sagt nun, ob es wirklich eine Kollision gab und wenn ja, was der Kontaktpunkt und die Eindringtiefe ist.

Kollision zwischen zwei Kreisen

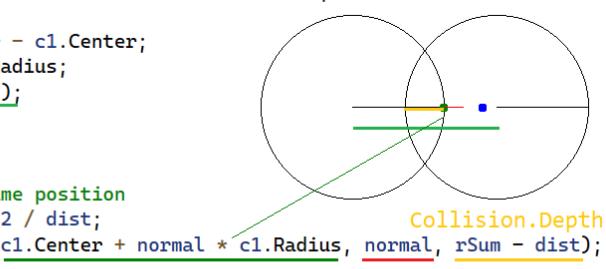
Siehe im Beispielprojekt unter PhysicEngine.UnitTests/CollisionDetection/CircleCircleTests.cs und PhysicEngine/CollisionDetection/NearPhase/NearPhaseTests.cs

Fall 1: CircleCircle_Overlapping_FoundsCollisionPoint

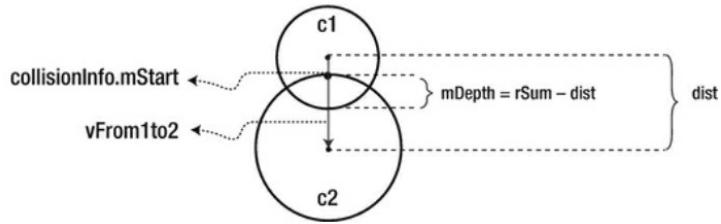
Zwei Kreise überlappen sich. Es wird auf Zeile 38 der grüne Kontaktpunkt zurück gegeben mit der roten Normale und der gelben Eindringtiefe.

```

28  public static CollisionInfo CircleCircle(ICollidableCircle c1, ICollidableCircle c2)
29  {
30      Vector2D from1to2 = c2.Center - c1.Center;
31      float rSum = c1.Radius + c2.Radius;
32      float dist = from1to2.Length();
33
34      if (dist != 0)
35      {
36          // overlapping but not same position
37          Vector2D normal = from1to2 / dist;
38          return new CollisionInfo(c1.Center + normal * c1.Radius, normal, rSum - dist);
39      }
40      else
41      {
42          // same position
43          Vector2D normal = new Vector2D(1, 0);
44          return new CollisionInfo(c1.Center + normal * c1.Radius, normal, rSum);
45      }
46 }
```



So sieht das CollisionInfo-Objekt bei Kreis-Kreis aus:

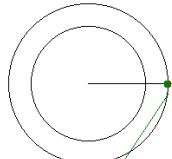


Fall 2: CircleCircle_SamePosition_FoundsCollisionPoint

Die beiden Kreiszentren liegen am gleichen Punkt. Gib Kollisionsnormale (1,0) zurück.

```

28 public static CollisionInfo CircleCircle(ICollidableCircle c1, ICollidableCircle c2)
29 {
30     Vector2D from1to2 = c2.Center - c1.Center;
31     float rSum = c1.Radius + c2.Radius;
32     float dist = from1to2.Length();
33
34     if (dist != 0)
35     {
36         // overlapping but not same position
37         Vector2D normal = from1to2 / dist;
38         return new CollisionInfo(c1.Center + normal * c1.Radius, normal, rSum - dist);
39     }
40     else ← liegen ist dist=0
41     {
42         //same position
43         Vector2D normal = new Vector2D(1, 0);
44         return new CollisionInfo(c1.Center + normal * c1.Radius, normal, rSum);
45     }
46 }
```



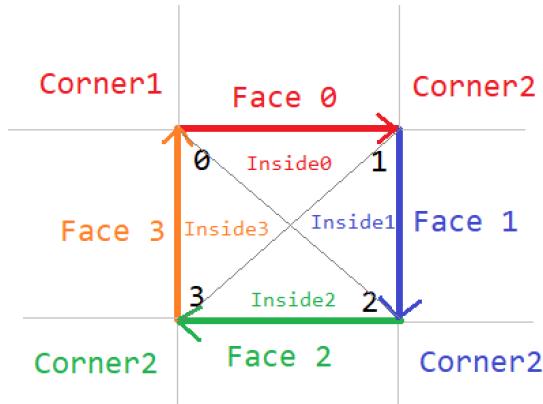
Collision.End liegt außerhalb von den Kreisen da es sich über Start+Normal*Depth errechnet

Kollision zwischen Rechteck und Kreis

Ein Rechteck mit den Eckpunkten 0,1,2,3 ist im Uhrzeigersinn definiert. Um festzustellen, ob es eine Kollision zwischen einem Rechteck und einem Kreis gibt, wird zuerst geschaut, in welchen Facebereich das Kreiszentrum liegt. Wenn es oberhalb der roten Linie liegt, dann liegt es bei Face0. Liegt es rechts neben der blauen Linie, dann ist es Face 1. Liegt es bei keiner der 4 Linien im Außenbereich, dann ist es Inside.

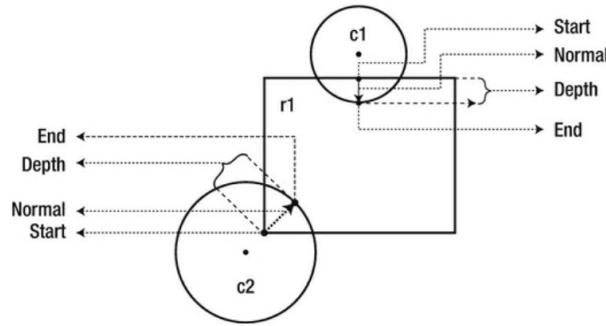
Liegt es außerhalb, dann wird als nächstes geprüft, ob es im Corner-Bereich 1 oder 2 von der jeweiligen Seite liegt.

Der Corner1-Fall tritt nur ein, wenn das Kreiszentrum auf der Face0-Seite liegt. Sollte es z.B. im roten Corner2-Bereich liegen, dann ist das kein blauer Corner1-Fall, da Face0 zuerst geprüft wird und dort dann schon festgestellt wird, dass es im roten Corner2 liegt.



Das Kreiszentrum kann in einer der 12 Bereiche liegen. Liegt es innerhalb oder im Face-Bereich, dann wird die senkrechte Distanz zur nächsten Seite berechnet. Liegt es in einem Corner-Bereich, dann wird die Distanz zwischen Ecke und Kreiszentrum bestimmt.

So sehen zwei CollisionInfo-Objekte für eine Rechteck-Kreis-Kollision aus. Kreis c1 liegt im Face 0-Bereich vom Rechteck und Kreis c2 liegt im Corner2-Bereich.



Die Kontaktpunkte werden hier mit Start(Punkt aus Objekt1) und End(Punkt aus Objekt2) bezeichnet. Die Punkte sollen so gewählt werden, das beim Auseinanderdrücken die Objekte eine möglichst kleine Distanz wandern sollen.

Fall 1: RectangleCircle_CircleCenterIsInside_FoundCollisionPoint

Siehe im Quellcode: PhysicEngine.UnitTests/CollisionDetection/RectangleCircleTests.cs

PhysicEngine/CollisionDetection/NearPhase/NearPhaseTests.cs

Es gibt eine Kollision zwischen ein Kreis und ein Rechteck. Das Kreiszentrum liegt innerhalb vom Rechteck. Es wird durch Vektor-Projektion für jede Rechteckseite geprüft, welchen Abstand das Zentrum zur Rechteckkante hat. Zu allen vier Kanten ist der Abstandswert kleiner Null und der größte negative Abstandswert wird bei Kante 1 gefunden. Also liegt der Kreis im Inside1-Bereich.

```

49  public static CollisionInfo RectangleCircle(ICollidableRectangle r, ICollidableCircle c)
50  {
51      bool inside = true; ← Nach Prüfung aller 4
52      float bestDistance = -99999; Seiten ist
53      int nearestEdge = 0; inside=false
54
55      for (int i=0;i<4;i++)
56      {
57          //find the nearest face for center of circle
58          float projection = (c.Center - r.Vertex[i]) * r.FaceNormal[i];
59          if (projection > 0) ← Bei allen 4 Seiten ist projection < 0
60          {
61              //if the center of circle is outside of rectangle
62              //Bei nearestEdge=1 wird hier festgestellt, dass das Kreiszentrum im
63              //nearestEdge ist
64              nearestEdge = i; Wird für alle 4 Seiten geprüft
65              inside = false;
66              break;
67          }
68          if (projection > bestDistance) ← Bei nearestEdge=1 wird hier festgestellt, dass das Kreiszentrum im
69          {
70              bestDistance = projection; Inside1-Bereich
71              nearestEdge = i; liegt
72          }
73      }
74
75      else
76      {
77          //the center of circle is inside of rectangle
78          Collision.Start
79          return new CollisionInfo(c.Center - r.FaceNormal[nearestEdge] * c.Radius, Collision.Normal
80          r.FaceNormal[nearestEdge], c.Radius - bestDistance);
81      }
82  }

```

Beim Inside1-Bereich ist der kürzeste Weg, wie der Kreis aus dem Rechteck heraus gedrückt wird, indem der Kreis nach Rechts und das Rechteck nach Links bewegt wird. Der Kollisionspunkt liegt am linken grünen Punkt vom Kreis und die Kollisionsnormale zeigt nach Rechts (roter Pfeil / Normale von Kante Face1). Die Eindringtiefe ist der Kreisradius minus bestDistance.

Fall 2: RectangleCircle_CircleHitCorner1_FoundCollisionPoint

Ein Kreis und ein Rechteck kollidiert und diesmal liegt der Kreis im Corner1-Bereich. Schon im ersten Schleifendurchlauf wird bei Zeile 59 festgestellt, dass projection positiv ist und nearestEdge somit 0 ist was zum Abbruch der Schleife bei Zeile 65 führt. Somit ist inside false und wir landen bei Zeile 85 wo festgestellt wird, dass dot kleiner Null ist weil das Kreiszentrum links neben der linken Rechteckkante liegt. Die Kontaktnormalen ergibt sich durch den normalisierten Richtungsvektor v1, welche von der linken oberen Rechteckkante zum Kreiszentrums zeigt. Durch diese Normale wird dann der grüne Kollisionspunkt bestimmt.

```

49  public static CollisionInfo RectangleCircle(ICollidableRectangle r, ICollidableCircle c)
50  {
51      bool inside = true;
52      float bestDistance = -99999;
53      int nearestEdge = 0;
54
55      for (int i=0;i<4;i++)
56      {
57          //find the nearest face for center of circle
58          float projection = (c.Center - r.Vertex[i]) * r.FaceNormal[i];
59          if (projection > 0)
60          {
61              //if the center of circle is outside of rectangle
62              bestDistance = projection;
63              nearestEdge = i;
64              inside = false;
65              break; ←
66          }
67          if (projection > bestDistance)
68          {
69              bestDistance = projection;
70              nearestEdge = i;
71          }
72      }
73
74      if (inside==false)
75      {
76          //the center of circle is outside of rectangle
77
78          //v1 is from left vertex of face to center of circle
79          //v2 is from left vertex of face to right vertex of face
80          Vector2D v1 = c.Center - r.Vertex[nearestEdge];
81          Vector2D v2 = r.Vertex[(nearestEdge + 1) % 4] - r.Vertex[nearestEdge];
82
83          float dot = v1 * v2;   Das Kreiszentrums liegt links von der linken oberen
84                           Ecke und ist somit im Corner1-Bereich von Seite0
85          if (dot < 0) ←
86          {
87              //the center of circle is in corner region of r.Vertex[nearestEdge]
88              float dis = v1.Length();
89
90              // compare the distance with radius to decide collision
91              if (dis > c.Radius) Es wird geprüft, ob die Distanz von der
92                  return null; ← Kreismitte zur linken oberen Ecke kleiner als
93                  der Radius ist
94                  Vector2D normal = v1.Normalize();
95                  return new CollisionInfo(c.Center - normal * c.Radius, normal, c.Radius - dis); //Corner 1
96          }
}

```

Fall 3: RectangleCircle_CircleHitCorner2_FoundCollisionPoint

Wie im Testfall zuvor befindet sich das Kreiszentrum wieder über der oberen Rechteckkante so dass nearestEdge wieder den Wert Null hat. Aber diesmal wird bei Zeile 85 festgestellt, dass dot größer Null ist, was dazu führt, dass wir im Else-Zweig von der Abfrage landen und somit bei Zeile 97 es weiter geht, wo der Face0/Corner2-Fall behandelt wird. Diesmal zeigt v1 nicht von der linken oberen Ecke zum Kreiszentrum sondern es zeigt von der rechten oberen Ecke zum Zentrum. Bei Zeile 104 wird der Abstand des Kreiszentrums zur rechten Rechteckkante ermittelt. Dort wird festgestellt, dass dot negativ ist was bedeutet, dass der Kreis also nicht bei Face0 sondern bei Corner2 liegt. Die Kollisionsnormale ist dann der Richtungsvektor von der rechten oberen Ecke des Rechtecks zum Kreiszentrum.

```

49  public static CollisionInfo RectangleCircle(ICollidableRectangle r, ICollidableCircle c)
50  {
51      bool inside = true;
52      float bestDistance = -99999;
53      int nearestEdge = 0;
54
55      for (int i=0;i<4;i++)
56      {
57          //find the nearest face for center of circle
58          float projection = (c.Center - r.Vertex[i]) * r.FaceNormal[i];
59          if (projection > 0)
60          {
61              //if the center of circle is outside of rectangle
62              bestDistance = projection;
63              nearestEdge = i; Schon bei i=0 ist
64              inside = false; projection>0. D.h. das
65              break; ← Kreiszentrum liegt oberhalb
66              von Seite0
67          if (projection > bestDistance)
68          {
69              bestDistance = projection;
70              nearestEdge = i;
71          }
72      }
73
74      if (inside==false)
75      {
76          //the center of circle is outside of rectangle
77
78          //v1 is from left vertex of face to center of circle
79          //v2 is from left vertex of face to right vertex of face
80          Vector2D v1 = c.Center - r.Vertex[nearestEdge];
81          Vector2D v2 = r.Vertex[(nearestEdge + 1) % 4] - r.Vertex[nearestEdge];
82
83          float dot = v1 * v2; Da dot>0 ist heißt das das Kreiszentrum
84          if (dot < 0)... else ← liegt rechts neben der linken oberen Ecke
85          { und somit nicht im Corner1-Bereich
86              //the center of circle is in corner region of mVertex[nearestEdge+1]
87
88              //v1 is from right vertex of face to center of circle
89              //v2 is from right vertex of face to left vertex of face
90              v1 = c.Center - r.Vertex[(nearestEdge+1) % 4];
91              v2 = -v2; Da dot<0 ist heißt das das
92              dot = v1 * v2; Kreiszentrum liegt rechts neben der
93              if (dot < 0) ← rechten oberen Ecke und somit im
94              { Corner2-Bereich
95                  float dis = v1.Length(); //compare the distance with radius to decide collision
96                  if (dis > c.Radius) ← Es wird der Abstand zur rechten
97                      return null; oberen Ecke bestimmt
98                  Vector2D normal = v1.Normalize();
99                  return new CollisionInfo(c.Center - normal * c.Radius, normal, c.Radius - dis); //Corner 2
100             }else
101         }
102     }
103 }
104 }
```

Fall 4: RectangleCircle_CircleHitRightFace_FoundCollisionPoint

In der Schleife bei Zeile 55 wird bei $i==1$ festgestellt, dass $\text{projection} > 0$ ist. Somit liegt das Kreis im Bereich Rot-Corner2, Blau-Face1 oder Blau-Corner2. Bei Zeile 74 ist $\text{inside}==\text{false}$. Bei Zeile 85 ist $\text{dot} > 0$ was bedeutet, dass der Kreis nicht bei Rot-Corner2 liegen kann sondern er liegt entweder bei Blau-Face1 oder Blau-Corner2. Da bei Zeile 105 $\text{dot} > 0$ ist heißt das der Kreis liegt im Blau-Face1-Bereich. Die Kollisionsnormale ergibt sich aus der Flächennormale von Face1.

```

49  public static CollisionInfo RectangleCircle(ICollidableRectangle r, ICollidableCircle c)
50  {
51      bool inside = true;
52      float bestDistance = -99999;
53      int nearestEdge = 0;
54
55      for (int i=0;i<4;i++)
56      {
57          //find the nearest face for center of circle
58          float projection = (c.Center - r.Vertex[i]) * r.FaceNormal[i];
59          if (projection > 0)
60          {
61              //if the center of circle is outside of rectangle
62              bestDistance = projection;
63              nearestEdge = i;
64              inside = false;
65              break;
66          }
67          if (projection > bestDistance)
68          {
69              bestDistance = projection;
70              nearestEdge = i;
71          }
72      }
73
74      if (inside==false)
75      {
76          //the center of circle is outside of rectangle
77
78          //v1 is from left vertex of face to center of circle
79          //v2 is from left vertex of face to right vertex of face
80          Vector2D v1 = c.Center - r.Vertex[nearestEdge];
81          Vector2D v2 = r.Vertex[(nearestEdge + 1) % 4] - r.Vertex[nearestEdge];
82
83          float dot = v1 * v2;
84          Da dot>0 ist heißt das das Kreiszentrum liegt unterhalb von
85          if (dot < 0)...else ← der rechten oberen Ecke und damit nicht im Corner1-Bereich
86          { von der rechten Seite
87              //the center of circle is in corner region of mVertex[nearestEdge+1]
88
89              //v1 is from right vertex of face to center of circle
90              //v2 is from right vertex of face to left vertex of face
91              v1 = c.Center - r.Vertex[(nearestEdge+1) % 4];
92              v2 = -v2;
93              Da dot>0 ist heißt das das Kreiszentrum liegt oberhalb
94              if (dot < 0)...else ← von der rechten unteren Ecke und somit nicht im
95              { Corner2-Bereich von der rechten Seite
96                  //the center of circle is in face region of face[nearestEdge]
97                  if (bestDistance < c.Radius) ←
98                      return new CollisionInfo(c.Center - r.FaceNormal[nearestEdge] * c.Radius, r.FaceNormal[nearestEdge], c.Radius - bestDistance);
99                  else
100                     return null;
101             }
102         }
103     }
104 }
105 }
```

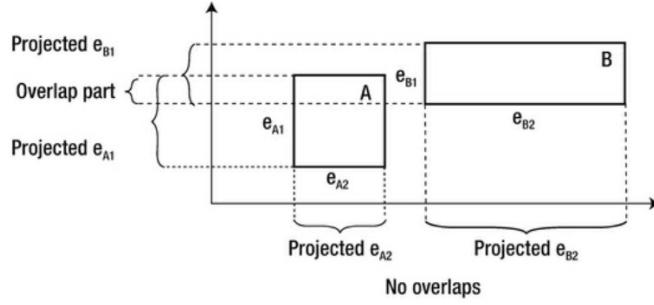
Kollision zwischen zwei Rechtecken

Um den Kollisionsalgorithmus zwischen zwei Rechtecken (welche nicht axial ausgerichtet sind) verstehen zu können benötigt man drei Begriffe: SAT, SupportPunkt und Achse mit der kleinste Eindringtiefe.

SAT – Separating Axis Theorem

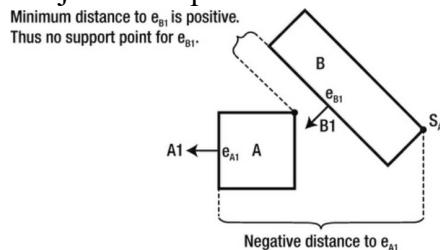
Mit SAT kann ich prüfen, ob zwei konvexe 2D-Polygone sich nicht schneiden. Wenn ich es schaffe zwischen zwei konvexe Polygone eine unendlich lange Linie zu zeichnen, welche von beiden Polygonen nicht berührt wird, dann kollidieren die beiden Polygone nicht. So eine Linie finde ich indem ich von beiden Polygonen alle Flächennormalen nehme und für jede Normale prüfe, ob das eine SAT-Linie ist. Wenn eine SAT-Linie gefunden wurde, dann gibt's keine Kollision und die Suche kann abgebrochen werden.

Beispiel für zwei Rechtecke: Projiziere ich die Rechtecke auf die Y-Achse, überschneiden sich die Bereiche e_A1 und e_B1. Projiziere ich auf die X-Achse, dann überschneiden sich e_A2 und e_B2 nicht. D.h. Die rechte Seite von A oder die linke Seite von B ist eine SAT-Linie. Die horizontalen Kanten von A und B sind keine SAT-Linien, da die Rechtecke sich überlappen.



Support-Punkt für eine Flächen normale

Der Support-Punkt für eine Normale von Objekt 1 ist der Eckpunkt des Objektes 2, welcher den größten negativen Abstand hat. Für die A1-Kante ist das der Punkt SA1. Für B1 gibt es kein Supportpunkt, da alle Punkte von Objekt A ein positiven Abstand haben (vor der B1-Fläche liegen).

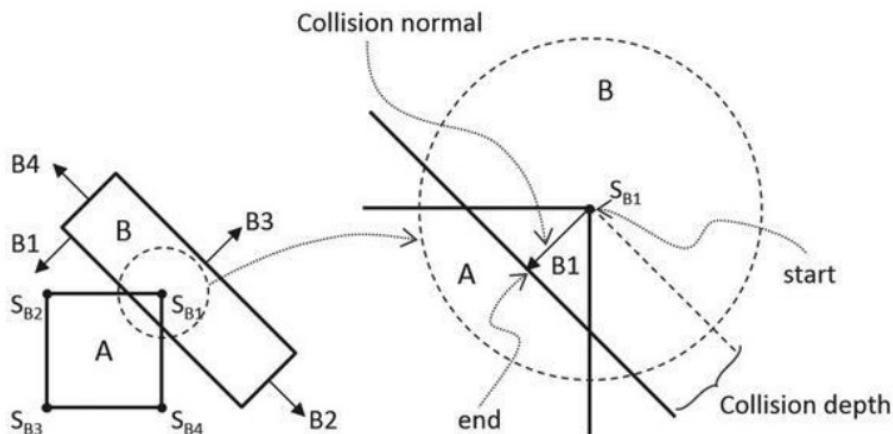


Hat eine Flächen normale von Objekt 1 kein Supportpunkt bei Objekt 2, dann kollidieren diese beiden Objekte nicht. Der Supportpunkt-Abstand ist eine positive Zahl. Er gibt den senkrechten Abstand zwischen einer Würfelseite und einer Würfecke an.

Die Achse mit der kleinsten Eindringtiefe

Der Supportpunkt von einer Fläche ist der Eckpunkt des anderen Objektes, welcher am Meisten in diese Fläche eingedrungen ist. Wenn ich von allen Flächen ihren Supportpunkt berechne, dann ist die Fläche, dessen Supportpunkt-Abstand am geringsten ist die Achse mit der geringsten Eindringtiefe.

Für alle Flächen von B wurden deren Supportpunkte in A berechnet. Die Fläche B1 hat mit Supportpunkt SB1 den kleinsten Supportpunkt abstand.



Die Kollisionsinformation daraus ist der Kontakt punkt SB1, die Kontakt normale B1, die Kollisionstiefe ist der Supportpunkt abstand. Der zweite Kontakt punkt 'end' lässt sich aus SB1, der Normale und der CollisionDepth berechnen.

Algorithmus zur Kollisionserkennung zwischen zwei Rechtecken

Um die Kollision zwischen zwei konvexen Formen A und B zu berechnen tue dieses:

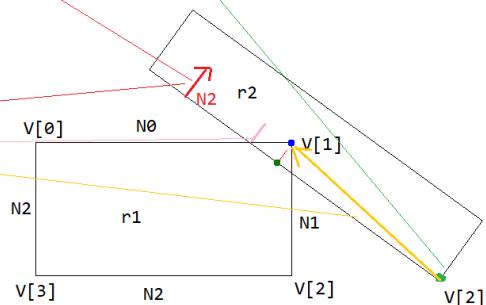
- Berechne für alle Flächen von A deren Supportpunkte
 - Hat eine der Flächen kein Supportpunkt, dann gibt es keine Kollision
 - Wenn alle Flächen ein Supportpunkt haben, dann schaue, wo der Supportpunktabstand am kleinsten ist.
- Berechne für alle Flächen von B deren Supportpunkte
 - Hat eine der Flächen kein Supportpunkt, dann gibt es keine Kollision
 - Wenn alle Flächen ein Supportpunkt haben, dann schaue, wo der Supportpunktabstand am kleinsten ist.
- Die Kollisionsinformation ist die Flächennormale (und deren Supportpunkt) von Objekt A oder B, dessen Supportpunktabstand am kleinsten ist.

Diese Funktion findet für eine Fläche den Eckpunkt vom anderen Rechteck, welcher am Meisten in die Fläche eindringt. FindSupportPoint(r1, -r2.N2, r2.V[2]) liefert hier r1.V[1] zurück.

Siehe im Quelltext: PhysicEngine/CollisionDetection/NearPhase/RectangleRectangleCollision1.cs

```

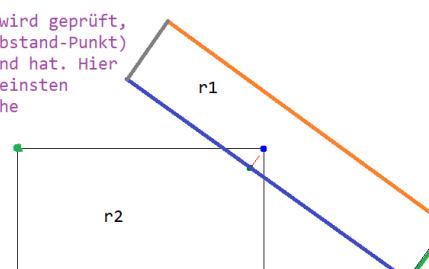
8  struct SupportStruct
9  {
10     1 Verweis
11     public SupportStruct(float supportPointDist)
12     {
13         this.SupportPointDist = supportPointDist;
14     }
15
16     public Vector2D SupportPoint = null;      Für die N2-Fläche von Rechteck r2 wird geprüft,
17     public float SupportPointDist = 0;          welche Punkt aus r1 den größten projection-Wert
18 }                                         hat(am Meisten eindringt). In diesen Fall wird
19  1 Verweis
20  private static SupportStruct FindSupportPoint(ICollidableRectangle r, Vector2D dir, Vector2D ptOnEdge)
21  {
22      var tmpSupport = new SupportStruct(-9999999);
23
24      //check each vector of other object
25      for (int i=0; i<r.Vertex.Length; i++)
26      {
27          //the longest project length
28          float projection = (r.Vertex[i] - ptOnEdge) * dir;
29
30          //find the longest distance with certain edge
31          //dir is -n direction, so the distance should be positive
32          if (projection > 0 && projection > tmpSupport.SupportPointDist)
33          {
34              tmpSupport.SupportPoint = r.Vertex[i];
35              tmpSupport.SupportPointDist = projection;
36          }
37
38      }
39
40      return tmpSupport;
41  }
```



Hier wird für alle Seiten von r1 ermittelt, was der dazu gehörige r2-Supportpunkt ist. SAT ist hier die Abbruchbedingung. Hinweis: Flächenfarbe von r1 = Supportpunktfarbe von r2

```

41  private static CollisionInfo FindAxisLeastPenetration(ICollidableRectangle r1, ICollidableRectangle r2)
42  {
43      float bestDistance = 999999;      Für alle 4 Flächen von r1 wird geprüft,
44      int bestIndex = -1;                welcher SupportPunkt(Max-Abstand-Punkt)
45      Vector2D supportPoint = null;    aus r2 den kleinsten Abstand hat. Hier
46
47      for (int i=0;i<r1.FaceNormal.Length;i++) Abstand zur blauen r1-Fläche
48      {
49          // Retrieve a face normal from A
50          var n = r1.FaceNormal[i];
51
52          // use -n as direction and the vertex on edge i as point on edge
53
54          // find the support on B
55          // the point has longest distance with edge i
56          var tmpSupport = FindSupportPoint(r2, -n, r1.Vertex[i]);
57
58          //SAT says if one side from r1 has no support-Point on r2, then there is no collision
59          if (tmpSupport.SupportPoint == null) return null;
60
61          //get the shortest support point depth
62          if (tmpSupport.SupportPointDist < bestDistance)
63          {
64              bestDistance = tmpSupport.SupportPointDist;
65              bestIndex = i;
66              supportPoint = tmpSupport.SupportPoint;
67          }
68      }
69  }
```



SAT besagt, dass wenn ich für eine Fläche von r1 kein Punkt bei r2 finde, der in diese Fläche eindringt, dann kollidieren r1 und r2 nicht

Es wird für beide Richtungen geprüft, dass SAT die Kollision verhindert.

```

74  public static CollisionInfo RectangleRectangle(ICollidableRectangle r1, ICollidableRectangle r2)
75  {
76      //find Axis of Separation for both rectangle
77      var collisionInfoR1 = FindAxisLeastPenetration(r1, r2);
78
79      if (collisionInfoR1 != null)
80      {
81          var collisionInfoR2 = FindAxisLeastPenetration(r2, r1);
82          if (collisionInfoR2 != null)
83          {
84              //if both of rectangles are overlapping, choose the shorter normal as the normal
85              if (collisionInfoR1.Depth < collisionInfoR2.Depth)
86                  return new CollisionInfo(collisionInfoR1.Start - collisionInfoR1.Normal * collisionInfoR1.Depth, collisionInfoR1.Normal, collisionInfoR1.Depth);
87              else
88                  return new CollisionInfo(collisionInfoR2.Start, -collisionInfoR2.Normal, collisionInfoR2.Depth);
89          }
90      }
91      return null;
92  }
93

```

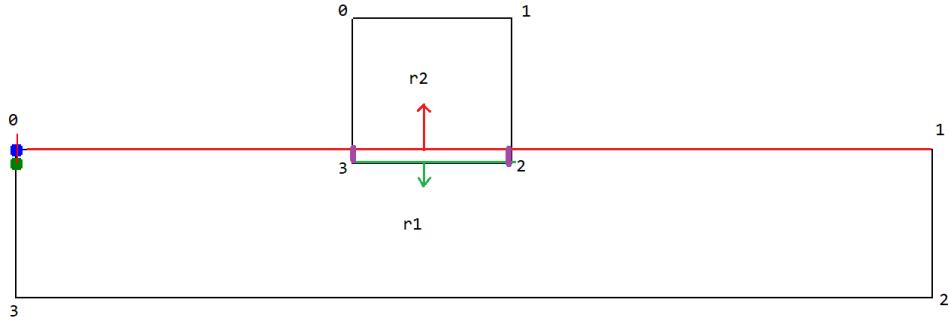
da R1.Start ja der Supportpunkt auf r2 ist und Collision.Start aber auf der Fläche von r1 liegen soll, reche ich hier -Normale*Depth

R2.Start ist bereits ein Eckpunkt von r1.
Somit kann ich den gleich nehmen und brauche nur die Normale drehen

Probleme mit dem Rechteck-Kollisionsalgorithmus aus dem „Building a 2D Game Physics Engine“-Buch

Siehe Unitest: RectangleRectangle_CubeOnTable_FoundTwoCollisionPoints

Wenn ich ein Würfel auf ein Tisch lege, dann findet er nicht die Kollisionspunkte r2.V[3] und r2.V[2] sondern r1.V[0]. Lila= Meine Erwartung; Blaue-Grüne-Punkte: Tatsächlich



Ursache:

Für die rote Linie von r1 müsste er die Punkte 2 und 3 aus r2 finden.

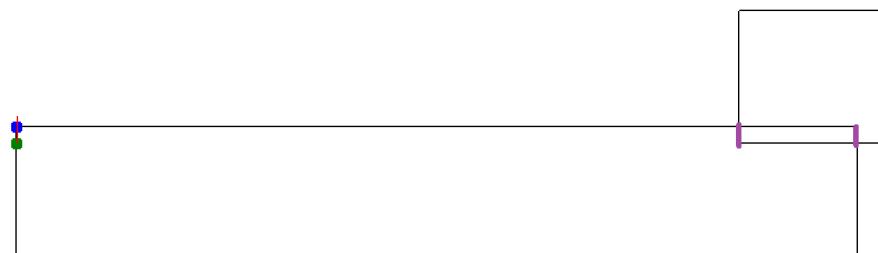
Für die grüne Linie von r2 müsste er die Punkte 0 und 1 aus r1 finden.

Alle 4 Punkte haben die gleichen Collision-Depth aber es kann nur einer von den 4 Punkten zurückgegeben werden. In diesen Fall hat er sich für die linke obere Ecke aus r1 entschieden, weil dieser Punkt zuerst in der Grüne-Linie-Suche gefunden wurde und weil er bei gleicher Kollisionstiefe r2 gegeben über r1 bevorzugt weil hier mit < anstatt <= gearbeitet wird:

```
if (collisionInfoR1.Depth < collisionInfoR2.Depth)
```

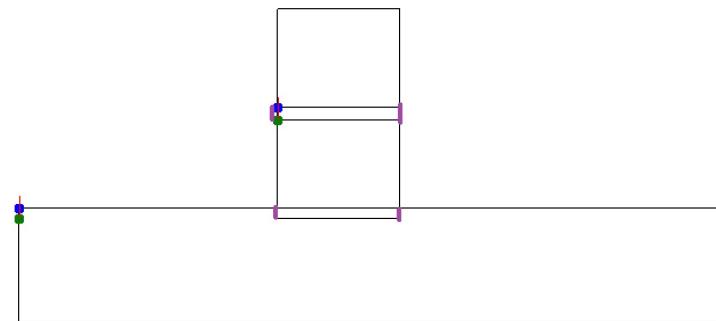
Weiterer Fehlerfall: RectangleRectangle_CubeOnTableCorner_FoundTwoCollisionPoints

Lila: Meine Erwartung; Blaue-Grüne-Punkte: Tatsächlich



Weiterer Fehlerfall: RectangleRectangle_CubeStackOnTable_FoundTwoCollisionPoints

Lila: Meine Erwartung; Blau-Grüne-Punkte: Tatsächlich

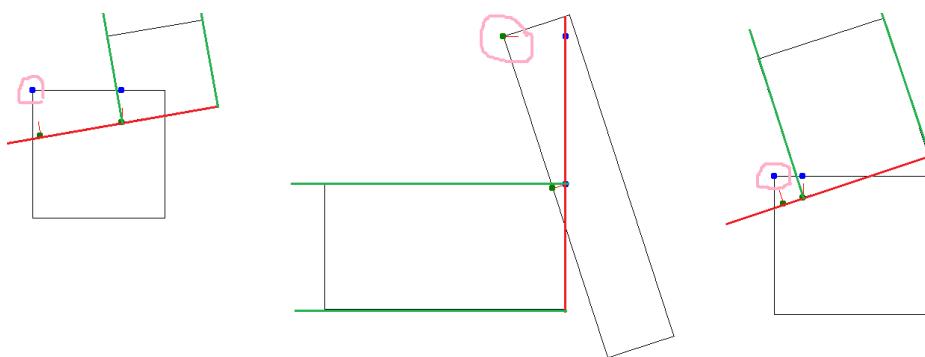


Überarbeitung des Rechteck-Kollisionsalgorithmus

Siehe im Quelltext: PhysicEngine/CollisionDetection/NearPhase/RectangleRectangleCollision2.cs

Da der original-Algorithmus zur Kollisionserkennung zwischen zwei Rechtecken aus dem „Building a 2D physics game engine“-Buch nicht alle Fälle abdeckt habe ich nun diesen Algorithmus an 5 Stellen geändert damit er nun alle Kollisionsfälle beherrscht.

Änderung 1: Left-Right-Clipping beim Support-Punkt



```
private static SupportStruct FindSupportPoint(ICollidableRectangle r, Vector2D dir, Vector2D p1OnEdge, Vector2D p2OnEdge)
{
    var tmpSupport = new SupportStruct(-9999999);

    //check each vector of other object
    for (int i = 0; i < r.Vertex.Length; i++)
    {
        //the longest project length
        Vector2D p1ToRi = r.Vertex[i] - p1OnEdge;
        float normalCheck = p1ToRi * dir;

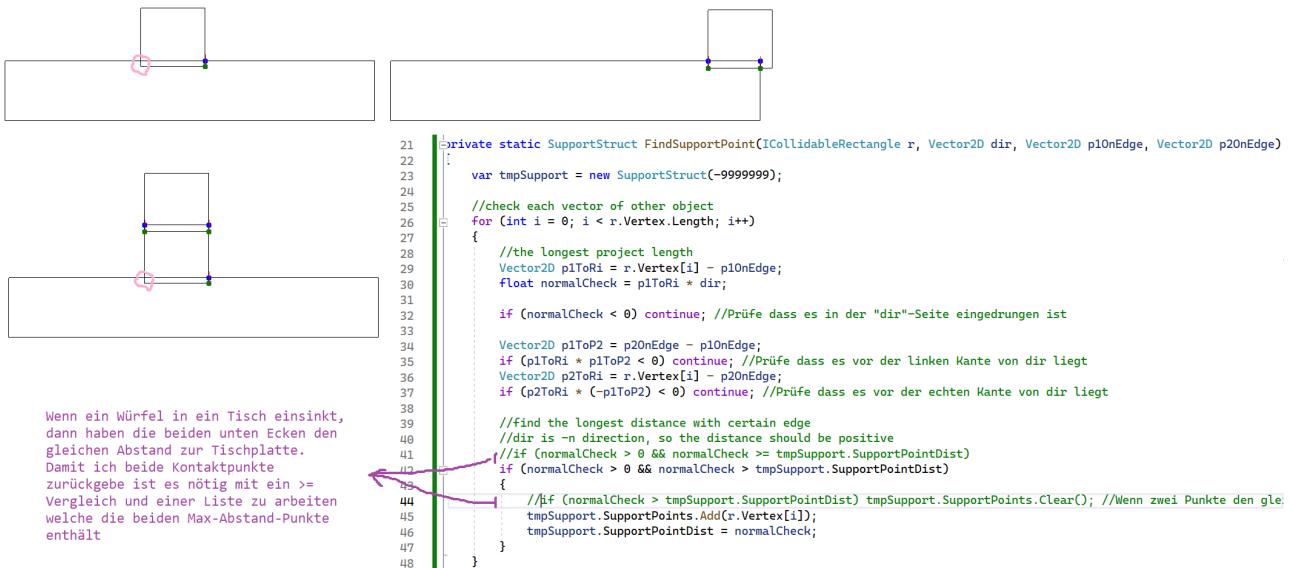
        Das Clipping an der linken und rechten
        roten Face-Seite ist nötig, um keine
        falschen Punkte (Pink eingekreist) zu
        finden

        if (normalCheck < 0) continue; //Prüfe dass es in der "dir"-Seite eingedrungen ist

        Vector2D p1ToP2 = p2OnEdge - p1OnEdge;
        if (p1ToRi * p1ToP2 < 0) continue; //Prüfe dass es vor der linken Kante von dir liegt
        Vector2D p2ToRi = r.Vertex[i] - p2OnEdge;
        if (p2ToRi * (-p1ToP2) < 0) continue; //Prüfe dass es vor der echten Kante von dir liegt

        //find the longest distance with certain edge
        //dir is -n direction, so the distance should be positive
        if (normalCheck > 0 && normalCheck >= tmpSupport.SupportPointDist)
        {
            if (normalCheck > tmpSupport.SupportPointDist) tmpSupport.SupportPoints.Clear(); //Wenn zwei Punkte den gleichen Abstand haben, lösche den alten
            tmpSupport.SupportPoints.Add(r.Vertex[i]);
            tmpSupport.SupportPointDist = normalCheck;
        }
    }
}
```

Änderung 2: Eine Fläche hat nicht nur maximal 1 sondern kann auch 2 Supportpunkte haben

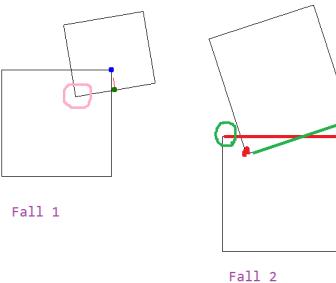


Änderung 3: FindAxisLeastPenetration gibt nun 0, 1 oder 2 Kontaktpunkte zurück

```

52  //Beim Würfel-Auf-Tisch-Fall werden hier zwei Punkte zurück gegeben. Sonst einer
53  2 Verweise
54  private static CollisionInfo[] FindAxisLeastPenetration(ICollidableRectangle r1, ICollidableRectangle r2)
55  {
56      float bestDistance = 999999;
57      int bestIndex = -1;
58      Vector2D[] supportPoints = null;
59
60      for (int i = 0; i < r1.FaceNormal.Length; i++)
61      {
62          // Retrieve a face normal from A
63          var n = r1.FaceNormal[i];
64
65          // use -n as direction and the vertex on edge i as point on edge
66
67          // find the support on B
68          // the point has longest distance with edge i
69          var tmpSupport = FindSupportPoint(r2, -n, r1.Vertex[i], r1.Vertex[(i+1)%4]);
70
71          //SAT says if one side from r1 has no support-Point on r2, then there is no collision
72          if (tmpSupport.SupportPoints == null) return null;
73
74          //get the shortest support point depth
75          if (tmpSupport.SupportPointDist < bestDistance)
76          {
77              bestDistance = tmpSupport.SupportPointDist;
78              bestIndex = i;
79              supportPoints = tmpSupport.SupportPoints.ToArray();
80          }
81
82          //all four directions have support point. That means at least one point from r2 lies inside from r1
83          return supportPoints.Select(x => new CollisionInfo(x + r1.FaceNormal[bestIndex] * bestDistance, r1.FaceNormal[bestIndex]));
    
```

Änderung 4: Anstatt den Kontaktspunkt mit der kleinsten Eindringtiefe werden nun aus beiden Rechtecken die Kontaktspunkte genommen



Fall 1

Fall 2

Anstatt dass ich nur den Kontaktpunkt aus collisionInfoR1 oder collisionInfoR2 (Je nachdem welcher den kleineren Depth-Wert) habe, nehme ich nun von beiden Funktionen ihre Kontaktpunkte. Würde ich nur ein von beiden nehmen, dann würde ich bei Fall 1 den eingekreisten Punkt nicht mit drin haben. Bei Fall 2 könnte garnicht erst zwischen den roten und grünen eingekreisten Punkt der Depth-Wert verglichen werden, da wegen dem Left-Right-Clipping an der grünen Kante der grüne Kontaktspunkte komplett fehlen würde.

```

public static CollisionInfo[] RectangleRectangle(ICollidableRectangle r1, ICollidableRectangle r2)
{
    List<CollisionInfo> contacts = new List<CollisionInfo>();

    var collisionInfoR1 = FindAxisLeastPenetration(r1, r2);
    if (collisionInfoR1 != null)
        contacts.AddRange(collisionInfoR1.Select(x => new CollisionInfo(x.Start - x.Normal * x.Depth, x.Normal, x.Depth)));

    var collisionInfoR2 = FindAxisLeastPenetration(r2, r1);
    if (collisionInfoR2 != null)
        contacts.AddRange(collisionInfoR2.Select(x => new CollisionInfo(x.Start, -x.Normal, x.Depth)));

    if (contacts.Any() == false) return null;

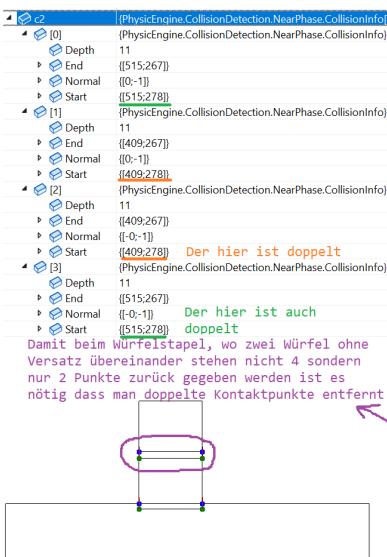
    //Beim Würfel-Stapelbeispiel erhalte ich doppelte einträge welche hier hier rausfiltern will
    var withoutDoubles = contacts
        .GroupBy(x => x.Start.ToString() + x.Depth.ToString())
        .Select(x => x.First())
        .ToArray();

    return withoutDoubles;
}

```

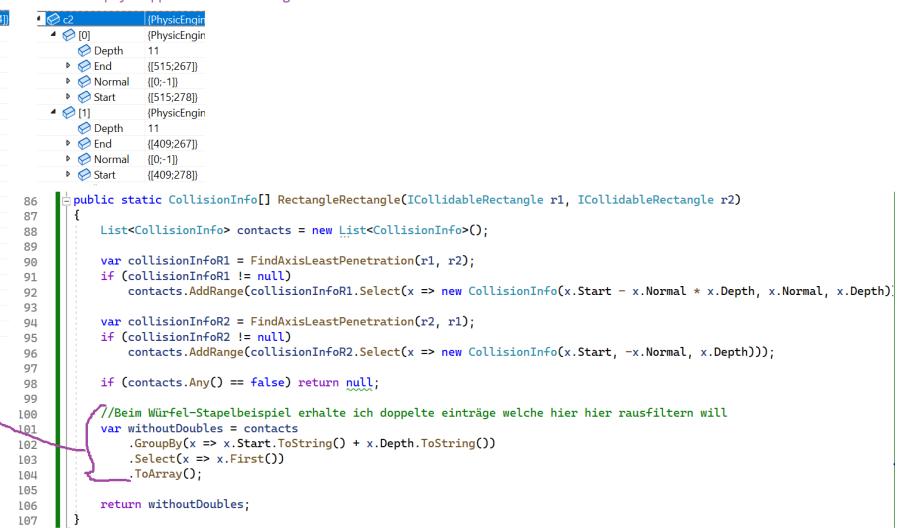
Änderung 5: Doppelte Kontaktspunkte werden entfernt

Ohne die GroupBy-Doppelten-Entfernung



Damit beim Würfelstapel, wo zwei Würfel ohne Versatz übereinander stehen nicht 4 sondern nur 2 Punkte zurück gegeben werden ist es nötig dass man doppelte Kontaktspunkte entfernt

Mit GroupBy-Doppelten-Entfernung



Die Kollisionserkennung von Rechteck-Rechteck bei Box2D-Lite

Siehe im Quelltext: PhysicEngine/CollisionDetection/NearPhase/RectangleRectangleCollision3.cs

Auf der Suche nach einem funktionierenden Rechteck-Kollisionsalgorithmus habe ich mir auch Box2D-Lite angesehen. Allerdings zeigt dieser Algorithmus leider falsche Kontaktspunkte an.

Im Debugger-Vergleich sieht man dass meine C#-Übersetzung das gleiche wie der C++-Originalcode als Kontaktspunkte erzeugt (gelb markiert).

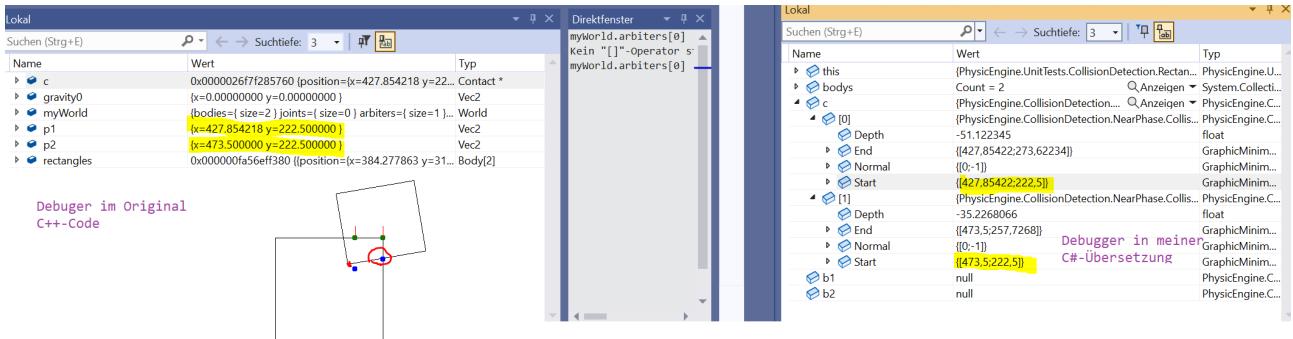
Fall 1: Die Rechtecke berühren sich nicht aber er findet 2 Kontaktspunkte.

Lokal	Direktfenster	Lokal	
Suchen (Strg+E)	Kein "[]"-Operator s... myWorld.arbiters[0]	Suchen (Strg+E)	
Name	Wert	Name	Wert
c	0x0000019bad155540 (position=x=546.500000 y=4... Contact *	this	(PhysicEngine.UnitTests.CollisionDetection.Rectan...
gravity0	{x=0.00000000 y=0.00000000}	bodies	Count = 2
myWorld		arbiter	Q Anzeigen ▾ System.Collections...
p1	[x=546.500000 y=416.548740]	c	(PhysicEngine.CollisionDetection...
p2	[x=546.500000 y=341.220154]	[0]	(PhysicEngine.CollisionDetection.NearPhase.Collis...
rectangles	0x0000000b26ff670 ({position=x=373.748444 y=3... Body[2]}	Depth	-83.00586

Debugger im Original C++-Code

Debugger in meiner C#-Übersetzung

Fall 2: Er findet zwar zwei Kontaktpunkte aber an der falschen stelle (Erwartung wären die roten Punkte).



ToDo: Verstehen wie Box2D-Lite die Kollision macht und dann ihren Quellcode korrigieren.

Kollisionen auflösen

Wenn man verhindern will, dass Starrkörper bei ihrer Bewegung ineinander laufen, dann braucht man im ersten Schritt ein Kollisionserkennungsalgorithmus, welcher zwei Kontaktpunkte und eine Kollisionsnormale liefert. Das haben wir im vorherigen Abschnitt behandelt.

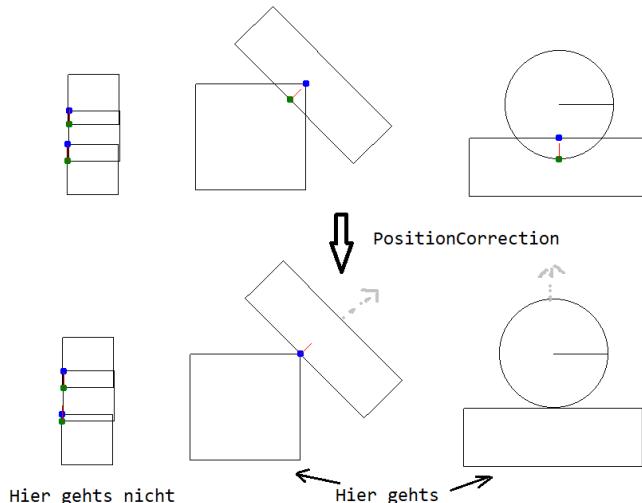
Es gibt nun mehrere Möglichkeiten wie man eine Kollision auflösen (beseitigen) kann. Nach der Kollisionsauflösung sollen die Körper sich dann nicht mehr überlappen.

Collision-Resolution 1 – PositionCorrection

Ich kann die Kollision dadurch auflösen, indem ich die Position so korrigiere, dass die Körper in Richtung Kontaktnormalen voneinander weggedrückt/verschoben werden. Durch die Wichtung mit der Masse kann ich dafür sorgen, dass der leichte Körper z.B. 90% der Wegdrückbewegung macht und der schwere Körper nur 10% so dass sich am Ende dann genau die Ränder der Objekte berühren.

```
81     private void PositionalCorrection(RigidBodyCollision[] collisions, float posCorrectionRate = 0.8f)
82     {
83         foreach(var c in collisions)
84         {
85             Vector2D correctionAmount = c.Normal * (c.Depth / (c.B1.InverseMass + c.B2.InverseMass) * posCorrectionRate);
86             c.B1.MoveCenter(-correctionAmount * c.B1.InverseMass);
87             c.B2.MoveCenter(correctionAmount * c.B2.InverseMass);
88         }
89     }
```

Beispiel: Es wurden Kollisionen festgestellt. Der Würfel ganz unten hat jeweils eine unendliche Masse (InverseMass=0). Das obere Objekt wird nach oben weggedrückt.



Der Vorteil von PositionCorrection ist, dass es leicht zu implementieren ist.

Die Nachteile: Wenn ich eine Würfelstapel habe (links im Bild), dann drückt es bei einmaliger Korrektur den mittleren Würfel in den obersten Würfel rein. Ich benötige mehrere Korrekturschritte damit alle Würfel sich nicht mehr überlappen.

Nächstes Problem: Wenn ein Würfel auf ein Tisch liegt, dann wird die Schwerkraft die Geschwindigkeit nach unten immer weiter erhöhen und die PositionCorrection drückt den Würfel wieder zurück nach oben. Das geht aber nur ein paar TimeSteps gut weil der Würfel mit jedem TimeStep immer schneller wird und irgendwann durch den Tisch hindurch springt.

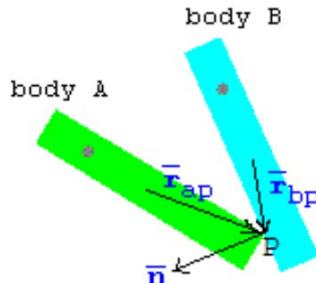
Um eine Kollision korrekt aufzulösen muss also nicht nur die Position sondern auch die Geschwindigkeit korrigiert werden. Wenn sich Körper überlappen aber die Geschwindigkeit Null ist, dann ist es ok, wenn man per PositionCorrection das korrigiert aber wenn die Körper sich auch noch bewegen, dann muss auch die Geschwindigkeit der Körper während der Kollisionsauflösung korrigiert werden. Wie man das macht kommt im nächsten Abschnitt.

Collision-Resolution 2 – Iterativer Impuls

Herleitung der Impuls-Formel

Quelle: <https://www.myphysicslab.com/engine2D/collision-en.html>

Ein Starrkörper wird durch seine Position des Zentrums, seine Ausrichtung, seine Zentrumsgeschwindigkeit und seine Winkelgeschwindigkeit beschrieben.



Körper A und B stoßen am Punkt P zusammen. Der Kollisionsalgorithmus hat ein Kollisionspunkt ap auf Körper A ermittelt und ein Kollisionspunkt bp. ap==bp=P. Die Kollisionsauflösung soll dadurch erfolgen, indem neue Geschwindigkeitswerte $\bar{v}_{a2}, \bar{v}_{b2}$ und ω_{a2}, ω_{b2} für die Körper berechnet werden so dass die Körper dann nicht mehr ineinander laufen.

m_a, m_b = Masse von Körper A und B

\bar{r}_{ap} = Distanzvektor vom Zentrum von A zu P $\bar{r}_{ap} = P - CenterA$

\bar{r}_{bp} = Distanzvektor vom Zentrum von B zu P $\bar{r}_{bp} = P - CenterB$

ω_{a1}, ω_{b1} = Winkelgeschwindigkeit von A und B vor dem Zusammenstoß

ω_{a2}, ω_{b2} = Winkelgeschwindigkeit von A und B nach dem Zusammenstoß

$\bar{v}_{a1}, \bar{v}_{b1}$ = Geschwindigkeit des Zentrums von A und B vor dem Zusammenstoß

$\bar{v}_{a2}, \bar{v}_{b2}$ = Geschwindigkeit des Zentrums von A und B nach dem Zusammenstoß

\bar{v}_{ap1} = Geschwindigkeit des Kollisionspunktes ap=P von Körper A vor dem Zusammenstoß

\bar{v}_{bp1} = Geschwindigkeit des Kollisionspunktes bp=P von Körper B vor dem Zusammenstoß

\bar{n} = Kollisionsnormale. Das ist die Flächennormale von der linken Seite von B

e = Elastizität (0=Unelastisch; 1=perfekt Elastisch)

Die Geschwindigkeit der Kollisionspunkte vor dem Zusammenstoß ist definiert über:

$$\bar{v}_{ap1} = \bar{v}_{a1} + \omega_{a1} \times \bar{r}_{ap} \quad \bar{v}_{bp1} = \bar{v}_{b1} + \omega_{b1} \times \bar{r}_{bp}$$

Die ähnliche Formel gilt auch nach dem Zusammenstoß:

$$\bar{v}_{ap2} = \bar{v}_{a2} + \omega_{a2} \times \bar{r}_{ap} \quad (0)$$

$$\bar{v}_{bp2} = \bar{v}_{b2} + \omega_{b2} \times \bar{r}_{bp} \quad (1)$$

Die Winkelgeschwindigkeit ist zwar eine skalare Zahl aber um damit das Kreuzprodukt bilden zu können definieren wir Omega als ein Vektor in Z-Richtung (Die Körper liegen in der XY-Ebene) wo die Länge angibt, wie schnell er sich dreht.

$$\omega \times \bar{r} = (0, 0, \omega) \times (r_x, r_y, 0) = (-\omega * r_y, \omega * r_x, 0)$$

Wir definieren nun die Geschwindigkeit, mit der sich die Kollisionspunkte vor und nach dem Stoß aufeinander zubewegen:

$$\bar{v}_{ab1} = \bar{v}_{ap1} - \bar{v}_{bp1} \quad (2)$$

$$\bar{v}_{ab2} = \bar{v}_{ap2} - \bar{v}_{bp2} \quad (3)$$

Wir können die Relativgeschwindigkeit auch über diese Formeln angeben:

$$\bar{v}_{ab1} = \bar{v}_{al} + \omega_{al} \times \bar{r}_{ap} - \bar{v}_{bl} - \omega_{bl} \times \bar{r}_{bp} \quad (4)$$

$$\bar{v}_{ab2} = \bar{v}_{a2} + \omega_{a2} \times \bar{r}_{ap} - \bar{v}_{b2} - \omega_{b2} \times \bar{r}_{bp} \quad (5)$$

Die Geschwindigkeit mit der sich die Kontaktpunkte in Normalenrichtung vor dem Zusammenstoß aufeinander zubewegen ist:

$$\bar{v}_{ab1} * \bar{n}$$

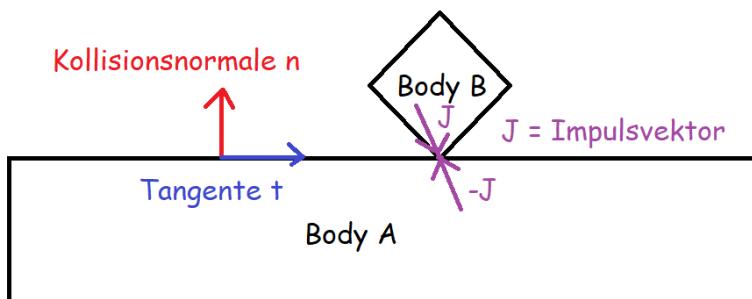
Wir wollen, dass die Körper durch den Stoß in Richtung Normale voneinander abprallen. Für die Relativgeschwindigkeit in Normalenrichtung soll gelten:

$$\bar{v}_{ab2} * \bar{n} = -e \bar{v}_{ab1} * \bar{n} \quad (6)$$

Das Minus kommt daher, weil die Geschwindigkeit in Normalenrichtung reflektiert wird. Das e gibt an, wie viel Prozent der Energie dabei verloren geht.

Normalerweise würde man bei einer Physiksimulation pro Zeitschritt Kräfte auf Objekte wirken lassen, wodurch dann pro Zeitschritt die Geschwindigkeit und Position sich ändert. In unseren Fall, wo zwei Starrkörper kollidieren will ich aber nicht über mehrere Zeitschritte eine Kraft wirken lassen und die Körper voneinander wegdrücken, um die Kollision zu lösen sondern ich will innerhalb von einen Zeitschritt die Geschwindigkeit so ändern, dass sie voneinander weg fliegen. Wenn man sich überlegt, dass ein Körper aus lauter Atomen(Massepunkten) besteht, welche über Federn verbunden sind, dann wäre eine Kollision so, dass die Atome gegeneinander stoßen, wodurch die Federn dann unter Spannung geraten und so würde sich dann die Stoßenergie durch den Körper über die Federn hindurch bewegen und letztendlich würden dann diese Federn den Stoß realisieren. Hier wird also über ein kurzen Zeitraum innerhalb des Körpers etwas mit den Massepunkten gemacht, was den Abprall der Körper dann verursacht.

Bei unserer Starrkörpersimulation wollen wir uns aber den Aufwand sparen dass wir über viele Massepunkte und mehrere TimeSteps Federkräfte wirken lassen. Wir gehen stattdessen davon aus, dass beim Zusammenstoß eine Abprallkraft beim Kollisionspunkt für eine kurze Zeit wirkt, welche die Kollisionspunkte auseinander drückt. Eine Kraft, die für eine kurze Zeit wirkt, nennt man Impuls. Er ist definiert über $P=F*dt$ aber auch über $p=m*v$. Anstelle von P verwenden wir hier aber J, da der Autor von myphysicslab das so gemacht hat. BodyA ist ein Tisch und BodyB ist ein Würfel, der nach rechts unten fällt. An der Würfecke wirkt nun für ein kurzen Zeitraum der Impulsvektor $-J$, wobei J in Würfelflugrichtung zeigt. Der Impuls J wirkt auf beide Körper genau gleichstark und er ist ein Vektor, der aber entgegengesetzt auf beide Körper wirkt.



Dieser Impuls J sorgt dafür, dass der Würfel am Tisch abprallt. Wenn ich den Vektor J mit der Normale n über das Dot-Produkt multipliziere, erhalte ich mit $j = J \cdot \bar{n}$ die Impulsmenge in Normalenrichtung. Groß J ist ein Vektor, klein j ist eine skalare Zahl. Wenn der Normalimpuls $j \cdot \bar{n}$ auf den Würfel nach oben wirkt, dann ändert sich die Würfelsgeschwindigkeit laut der $p=m*v$ -Formel um den v-Wert. Wenn ich den Normalimpuls durch die Masse vom Würfel dividiere, dann sehe ich, um welchen Geschwindigkeitswert der Würfel sich ändert. Das gleiche gilt für den Tisch. Der Normalimpuls ist gleichgroß aber mit unterschiedlichen Vorzeichen und durch die Division mit der Masse erhalte ich den Wert, um den sich die Geschwindigkeiten der Körper in

Normalenrichtung ändert. Somit ergibt sich die Geschwindigkeit der beiden Körper nach der Normalimpulsanwendung über folgende Formeln:

$$\bar{v}_{a2} = \bar{v}_{a1} + j \bar{n} / m_a \quad (7)$$

$$\bar{v}_{b2} = \bar{v}_{b1} - j \bar{n} / m_b \quad (8)$$

Die Impulskraft in Normalenrichtung setzt an den Kollisionspunkten an und über den Hebelarm r wirkt dann die Kraft, welche den Körper drehen lässt. Die neue Winkelgeschwindigkeit ist somit:

$$\omega_{a2} = \omega_{a1} + (\bar{r}_{ap} \times j \bar{n}) / I_a \quad (9)$$

$$\omega_{b2} = \omega_{b1} - (\bar{r}_{bp} \times j \bar{n}) / I_b \quad (10)$$

Alles was ich jetzt noch brauche ist den noch unbekannten Normalimpuls j so dass ich dann über die Gleichungen (7), (8), (9), (10) die neuen Geschwindigkeitswerte ermitteln kann.

Wir starten mit Gleichung (6):

$$\bar{v}_{ab2} * \bar{n} = -e \bar{v}_{ab1} * \bar{n} \quad (6)$$

Wir ersetzen Links die Relativgeschwindigkeit mit (3)

$$(\bar{v}_{ap2} - \bar{v}_{bp2}) * \bar{n} = -e \bar{v}_{ab1} * \bar{n}$$

Nun ersetzen wir Links mit (0) und (1)

$$(\bar{v}_{a2} + \omega_{a2} \times \bar{r}_{ap} - \bar{v}_{b2} - \omega_{b2} \times \bar{r}_{bp}) * \bar{n} = -e \bar{v}_{ab1} * \bar{n}$$

Nun ersetzen wir Links mit (7), (8), (9), (10)

$$((\bar{v}_{a1} + j \bar{n} / m_a) + (\omega_{a1} + (\bar{r}_{ap} \times j \bar{n}) / I_a) \times \bar{r}_{ap} - (\bar{v}_{b1} - j \bar{n} / m_b) - (\omega_{b1} - (\bar{r}_{bp} \times j \bar{n}) / I_b) \times \bar{r}_{bp}) * \bar{n} = -e \bar{v}_{ab1} * \bar{n}$$

Die Klammer beim va1- und vb2-Term kann weg. Das Kreuzprodukt von rap/rbp geht in die Klammer rein.

$$(\bar{v}_{a1} + j \bar{n} / m_a + \omega_{a1} \times \bar{r}_{ap} + (\bar{r}_{ap} \times j \bar{n}) \times \bar{r}_{ap} / I_a - \bar{v}_{b1} + j \bar{n} / m_b - \omega_{b1} \times \bar{r}_{bp} + (\bar{r}_{bp} \times j \bar{n}) \times \bar{r}_{bp} / I_b) * \bar{n} = -e \bar{v}_{ab1} * \bar{n}$$

Ich stelle die Terme in der großen Klammer links etwas um so das (4) am Anfang auftaucht:

$$(\bar{v}_{a1} + \omega_{a1} \times \bar{r}_{ap} - \bar{v}_{b1} - \omega_{b1} \times \bar{r}_{bp} + j \bar{n} / m_a + (\bar{r}_{ap} \times j \bar{n}) \times \bar{r}_{ap} / I_a + j \bar{n} / m_b + (\bar{r}_{bp} \times j \bar{n}) \times \bar{r}_{bp} / I_b) * \bar{n} = -e \bar{v}_{ab1} * \bar{n}$$

Ersetzung der ersten 4 Terme mit (4)

$$(\bar{v}_{ab1} + j \bar{n} / m_a + (\bar{r}_{ap} \times j \bar{n}) \times \bar{r}_{ap} / I_a + j \bar{n} / m_b + (\bar{r}_{bp} \times j \bar{n}) \times \bar{r}_{bp} / I_b) * \bar{n} = -e \bar{v}_{ab1} * \bar{n}$$

Das vab1 multipliziere ich mit n und nehme es aus der Klammer raus

$$\bar{v}_{ab1} * \bar{n} + (j \bar{n} / m_a + (\bar{r}_{ap} \times j \bar{n}) \times \bar{r}_{ap} / I_a + j \bar{n} / m_b + (\bar{r}_{bp} \times j \bar{n}) \times \bar{r}_{bp} / I_b) * \bar{n} = -e \bar{v}_{ab1} * \bar{n}$$

Ich bringe den Term ganz Links auf die rechte Seite

$$(j \bar{n} / m_a + (\bar{r}_{ap} \times j \bar{n}) \times \bar{r}_{ap} / I_a + j \bar{n} / m_b + (\bar{r}_{bp} \times j \bar{n}) \times \bar{r}_{bp} / I_b) * \bar{n} = -e \bar{v}_{ab1} * \bar{n} - \bar{v}_{ab1} * \bar{n}$$

Auf der rechten Seite extrahiere ich vab1*n

$$(j \bar{n} / m_a + (\bar{r}_{ap} \times j \bar{n}) \times \bar{r}_{ap} / I_a + j \bar{n} / m_b + (\bar{r}_{bp} \times j \bar{n}) \times \bar{r}_{bp} / I_b) * \bar{n} = (-e - 1) * \bar{v}_{ab1} * \bar{n}$$

Auf der rechten Seite das Minus aus der Klammer nehmen:

$$(j \bar{n} / m_a + (\bar{r}_{ap} \times j \bar{n}) \times \bar{r}_{ap} / I_a + j \bar{n} / m_b + (\bar{r}_{bp} \times j \bar{n}) \times \bar{r}_{bp} / I_b) * \bar{n} = -(1 + e) * \bar{v}_{ab1} * \bar{n}$$

Ich bringe nun links das n in die Klammer rein. Da n die Länge 1 hat ergibt n*n=1

$$(j 1 / m_a + ((\bar{r}_{ap} \times j \bar{n}) \times \bar{r}_{ap}) * \bar{n} / I_a + j 1 / m_b + ((\bar{r}_{bp} \times j \bar{n}) \times \bar{r}_{bp}) * \bar{n} / I_b) = -(1 + e) * \bar{v}_{ab1} * \bar{n}$$

Ich wende nun folgende Formel

$$(\bar{A} \times \bar{B}) \cdot \bar{C} = (\bar{B} \times \bar{C}) \cdot \bar{A}$$

Mit

$$\bar{A} = (\bar{r}_{ap} \times j \bar{n}) \quad \bar{B} = \bar{r}_{ap} \quad \bar{C} = \bar{n}$$

an

und erhalte

$$((\bar{r}_{ap} \times j \bar{n}) \times \bar{r}_{ap}) * \bar{n} = (\bar{r}_{ap} \times \bar{n}) * (\bar{r}_{ap} \times j \bar{n})$$

$$((\bar{r}_{bp} \times j \bar{n}) \times \bar{r}_{bp}) * \bar{n} = (\bar{r}_{bp} \times \bar{n}) * (\bar{r}_{bp} \times j \bar{n})$$

Der Impulsvektor J ist ein Vektor. Die Impulsmenge j, welche in Normalenrichtung wirkt ist ein

Skalar. Damit kann ich das j auch aus der Klammer auf der rechten Seite raus nehmen:

$$((\bar{r}_{ap} \times j \bar{n}) \times \bar{r}_{ap}) * \bar{n} = (\bar{r}_{ap} \times \bar{n}) * (\bar{r}_{ap} \times \bar{n}) * j$$

$$((\bar{r}_{bp} \times j \bar{n}) \times \bar{r}_{bp}) * \bar{n} = (\bar{r}_{bp} \times \bar{n}) * (\bar{r}_{bp} \times \bar{n}) * j$$

Nun wenden wir die nächste Regel an:

$$(\bar{A} \times \bar{B}) \cdot (\bar{A} \times \bar{B}) = (\bar{A} \times \bar{B})^2$$

$$(\bar{r}_{ap} \times \bar{n}) * (\bar{r}_{ap} \times \bar{n}) * j = (\bar{r}_{ap} \times \bar{n})^2 * j$$

$$(\bar{r}_{bp} \times \bar{n}) * (\bar{r}_{bp} \times \bar{n}) * j = (\bar{r}_{bp} \times \bar{n})^2 * j$$

Ich wende nun diese Ersetzung:

$$((\bar{r}_{ap} \times j \bar{n}) \times \bar{r}_{ap}) * \bar{n} = (\bar{r}_{ap} \times \bar{n})^2 * j \quad ((\bar{r}_{bp} \times j \bar{n}) \times \bar{r}_{bp}) * \bar{n} = (\bar{r}_{bp} \times \bar{n})^2 * j$$

In der Formel an:

$$(j 1/m_a + (\bar{r}_{ap} \times \bar{n})^2 * j / I_a + j 1/m_b + (\bar{r}_{bp} \times \bar{n})^2 * j / I_b) = -(1+e) * \bar{v}_{abl} * \bar{n}$$

Ich nehme das j aus der Klammer

$$j * (1/m_a + 1/m_b + (\bar{r}_{ap} \times \bar{n})^2 / I_a + (\bar{r}_{bp} \times \bar{n})^2 / I_b) = -(1+e) * \bar{v}_{abl} * \bar{n}$$

Ich dividiere durch die Klammer von der linken Seite:

$$j = \frac{-(1+e) * \bar{v}_{abl} * \bar{n}}{(1/m_a + 1/m_b + (\bar{r}_{ap} \times \bar{n})^2 / I_a + (\bar{r}_{bp} \times \bar{n})^2 / I_b)} \quad (11)$$

Mit diesen skalaren j habe ich nun den Impuls in Normalenrichtung. Fehlt noch der Impuls in Tangentenrichtung und schon kenne ich den Impulsvektor J. Dazu starte ich wieder mit Gleichung (6) und diesmal nehme ich nicht die Kollisionsnormale n sondern die Kollisionstangente und erhalte dann die gleiche Gleichung wie (11) nur das n durch die Tangente ersetzt wurde und e durch den Reibungskoeffizienten.

Wenn man eine Kollision von ein Ball A mit einer Wand B simulieren will, dann setze ich einfach für B die Masse unendlich ein und Gleichung (11) wird zu:

$$j = \frac{-(1+e) * \bar{v}_{abl} * \bar{n}}{(1/m_a + (\bar{r}_{ap} \times \bar{n})^2 / I_a)} \quad (12)$$

Erklärung des Impulses am Quelltext

```

50 // compute and apply response impulses for each object
51 float newRestituation = Math.Min(s1.Restituation, s2.Restituation);
52 float newFriction = Math.Min(s1.Friction, s2.Friction);
53
54 float R1crossN = Vector2D.ZValueFromCross(r1, n);
55 float R2crossN = Vector2D.ZValueFromCross(r2, n);
56
57 // Calc impulse scalar
58 // the formula of jN can be found in http://www.myphysicslab.com/collision.html
59 float jN = -(1 + newRestituation) * velocityInNormal;
60 jN = jN / (s1.InverseMass + s2.InverseMass + Wende Formel 11 aus der
61 R1crossN * R1crossN * s1.InverseInertia + Herleitung an um jN zu
62 R2crossN * R2crossN * s2.InverseInertia); berechnen
63
64 //impulse is in direction of normal ( from s1 to s2)
65 Vector2D impulse = n * jN;
66 // impulse = Integral F dt = m * delta-v
67 // delta-v = impulse / m
68 s1.Velocity -= impulse * s1.InverseMass;  $\bar{v}_{a2} = \bar{v}_{a1} + j \bar{n} / m_a$  (7)
69 s2.Velocity += impulse * s2.InverseMass;  $\bar{v}_{b2} = \bar{v}_{b1} - j \bar{n} / m_b$  (8)
70
71 s1.AngularVelocity -= R1crossN * jN * s1.InverseInertia;
72 s2.AngularVelocity += R2crossN * jN * s2.InverseInertia;
73 Wende aus der Herleitung die Formeln um die Impulskraft in eine Beschleunigung umzurechnen und diese ändert dann die Geschwindigkeit
74 Von der Relativgeschwindigkeit ziehe ich die Geschwindigkeit in Normalenrichtung ab und erhalte so die Tangential-Geschwindigkeit
75
76 Vector2D tangent = relativeVelocity - n * (relativeVelocity * n);
77 float tangentLength = tangent.Length();
78 if (tangentLength == 0) return;
79 //relativeVelocity.dot(tangent) should less than 0
80 tangent = -tangent / tangentLength;
81 Die Tangent-Impulsrichtung soll entgegen der Tangentrichtung wirken. Deswegen wird sie hier negiert.
82
83 float r1CrossT = Vector2D.ZValueFromCross(r1, tangent);
84 float r2CrossT = Vector2D.ZValueFromCross(r2, tangent);
85
86 float jT = -(1 + newRestituation) * (relativeVelocity * tangent) * newFriction;
87 jT = jT / (s1.InverseMass + s2.InverseMass + Wende wieder Formel 11 an aber
88 r1CrossT * r1CrossT * s1.InverseInertia + mit anderen Werten für n und ε
89
90 //friction should less than force in normal direction
91 if (jT > jN)
92     jT = jN;
93
94 //impulse is from s1 to s2 (in opposite direction of velocity)
95 impulse = tangent * jT;
96
97 s1.Velocity -= impulse * s1.InverseMass;
98 s2.Velocity += impulse * s2.InverseMass;
99 s1.AngularVelocity -= r1CrossT * jT * s1.InverseInertia;
100 s2.AngularVelocity += r2CrossT * jT * s2.InverseInertia;
101
102 Lasse die Tangent-Impulskraft
103 wirken um die Geschwindigkeit zu
104 modifizieren

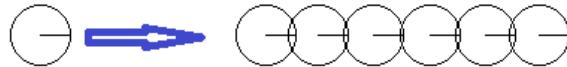
```

$$j = \frac{-(1+e) * \bar{v}_{ab1} * \bar{n}}{(1/m_a + 1/m_b + (\bar{r}_{ap} \times \bar{n})^2/I_a + (\bar{r}_{bp} \times \bar{n})^2/I_b)} \quad (11)$$

Würfel fliegt nach rechts unten

Anwendung des Impulses als Schleife vs Einmal pro Kontaktpunkt

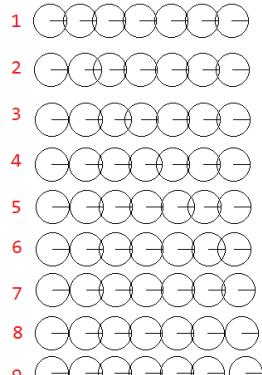
Man könnte jetzt direkt pro Zeitschritt die ApplyImpulse-Funktion für alle Kontaktpunkte aufrufen. Wenn ich eine Reihe von Kugeln habe und dort fliegt nun eine andere Kugel dagegen. Siehe Bild:



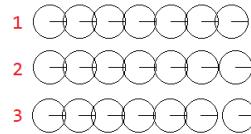
Dann würde es aber bedeuten, dass man ganz viele Zeitschritte braucht, bis der Impuls bis zur letzten Kugel durch geleitet wurde. Aus diesen Grund habe ich eine Schleife, die bis zu 15 mal probiert, ob es noch ein Kontaktspunkt gibt, wo noch ein Impuls anwendbar ist(d.h. Die Kontaktspunkte fliegen aktuell aufeinander zu und der Impuls sorgt nun dafür, dass sie voneinander wegfliegen).

```
9  public static void ApplyImpulsesUntilAllCollisionsAreResolved(RigidBodyCollision[] collisions)
10 {
11     int maxIterationCount = 15;
12
13     //Rufe so oft die ApplyImpulses-Funktion auf, bis alle Kollision-Kontaktspunkte sich voneinander weg bewegen
14     //oder die Max-IterationCount erreicht ist
15     for (int i=0;i< maxIterationCount;i++)
16     {
17         if (ApplyImpulses(collisions) == false)
18             return;
19     }
20
21     //throw new Exception("not all collisions could be resolved");
22 }
23
24 //Returnvalue: true = some Impulse was applied; false = No Impulse was applied
25 1 Verweis
26 public static bool ApplyImpulses(RigidBodyCollision[] collisions)
27 {
28     bool impulseWasApplied = false;
29     foreach (var collision in collisions)
30     {
31         if (ApplyImpulse(collision))
32             impulseWasApplied = true;
33     }
34
35     return impulseWasApplied;
```

Vergleich: Impuls mit/ohne Schleife (Rote Zahlen = TimeSteps)



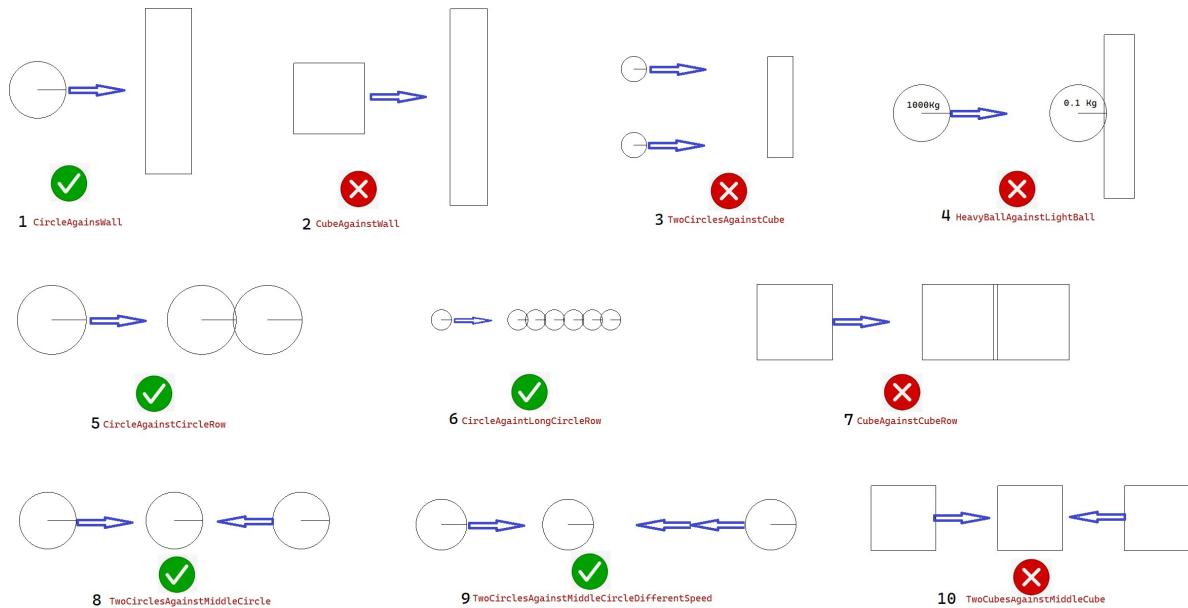
Ein TimeStep = Ein
ApplyImpulses-Aufruf



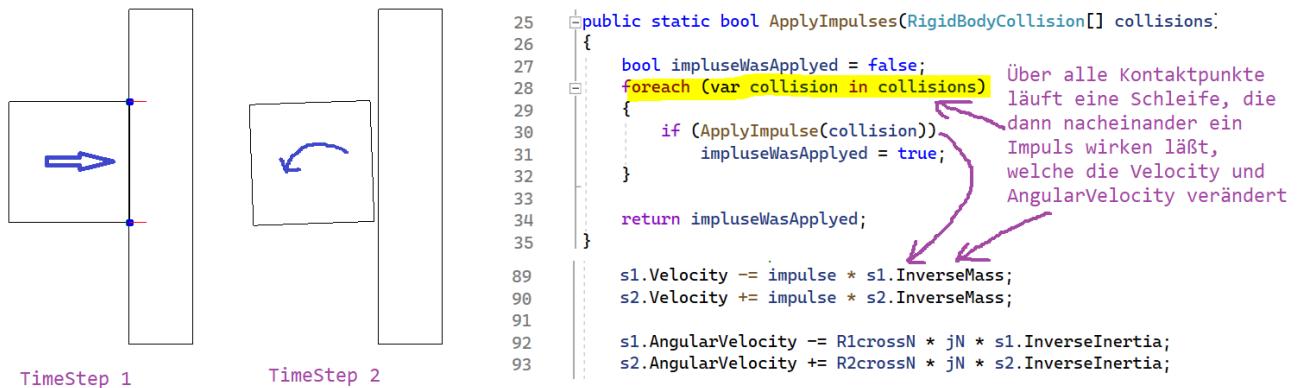
Ein TimeStep = Ein
ApplyImpulsesUntilAllCollis
ionsAreResolved-Aufruf

So gut funktioniert die Kollisionsauflösung über den iterativen Impulsansatz

Hier sind nun 10 Testfälle, wo Objekte (Mit blauen Pfeil) gegen andere Objekte stoßen. Bei 5 Tests funktioniert der Zusammenstoß und die Objekte prallen korrekt voneinander ab. Bei 5 Tests gibt's ein Problem.



Bei Testfall 2 fliegt ein Würfel gegen eine Wand. Erwartung wäre, dass er ohne Rotation abprallt. Stattdessen dreht er sich:



```

25 public static bool ApplyImpulses(RigidbodyCollision[] collisions)
26 {
27     bool impulseWasApplied = false;
28     foreach (var collision in collisions)
29     {
30         if (ApplyImpulse(collision))
31             impulseWasApplied = true;
32     }
33     return impulseWasApplied;
34 }
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93

```

Über alle Kontaktpunkte läuft eine Schleife, die dann nacheinander ein Impuls wirken lässt, welche die Velocity und AngularVelocity verändert

Der Grund dafür ist, weil erst wird der Impuls vom oberen Kontaktspunkt angewendet was dazu führt, dass eine Kraft auf die rechte obere Würfecke nach links wirkt, so dass er Schwung nach links bekommt aber auch eine Linksdrehung. Die untere Ecke wird nach rechts gegen die Wand gedrückt. Nun wird der zweite Impuls am unteren Kontaktspunkt angewendet, welcher aber wegen der schon bestehenden Würfeldrehung nun mit anderer Impulskraft wirkt als am oberen Kontaktspunkt. Diese ungleichen Impulskräfte oben und unten lassen den Würfel dann rotieren.

Bei Testfall 3 ist es das ähnliche Prinzip. Zwei Kugeln fliegen gegen ein Stab. Weil erst der eine Kontaktspunkt wirkt, wird der Stab rotiert und der untere Kontaktspunkt hat einen anderen Impuls, weil der Stab bereits rotiert. Somit fliegt der Stab dann mit Rotation weg anstatt ohne Rotation einfach nur nach rechts. Bei Testfall 7 und 10 kann der Würfel aus gleichen Grund nicht rotationsfrei abprallen.

Beim Testfall 4 fliegt eine schwere Kugel gegen eine leichte Kugel, die vor einer Wand liegt. Dieser Test wird rot, wenn die TimeStep-Zeit zu groß ist oder die maxIterationCount von der ImpulseResolution-Schleife zu klein. Man kann sich den Stoß der schweren Kugel gegen die Leichte so vorstellen, dass man versucht eine rollende Bowlingkugel dadurch zu stoppen, indem man Erbsen dagegen wirft. Jeder Erbsenwurf nimmt nur wenig Schwung von der Bowlingkugel weg. Wenn man nur 150 Erbsen dagegen wirft, dann reicht es nicht aus die Kugel zu stoppen. Hier haben 1500 Impulse (Erbsenwürfe) dagegen dann gereicht die Bowlingkugel zu stoppen und dessen Geschwindigkeit umzukehren.

Was man hier sieht sind die absoluten Geschwindigkeitswerte von der schweren und leichten Kugel über die gelb markierte ImpulseResolution-Schleife.



Die eine Sache, die man also beim Test hier beachten muss ist, dass maxIterationCount groß genug ist. Das andere ist die TimeStep-Schrittweite. Die leichte Kugel prallt innerhalb der Impulseschleife immer wieder nach links und rechts ab und bekommt dabei immer mehr Geschwindigkeit. Ist die Geschwindigkeit zu groß, dann tunnelt sie nach der Impulsschleife in der Position-Move-Funktion durch die Wand hindurch. D.h. sie fliegt durch die Wand weil die Positions-Schrittweite von der Kugel größer ist als wie die Wand dick ist.

Zusammenfassung von Teil 1

Hier wurde die Kollisionserkennung zwischen Körper erklärt und wie man über PositionCorrection und IterativeImpulse die Kollision auflöst. Dabei gab es zwei Problemfälle:

Für den Heavy-Ball-Test gibt es eine Lösung indem man mehr Rechenaufwand pro Zeit macht. Für das Problem mit den Würfel, der sich dreht, wenn er gegen eine Wand fliegt braucht es ein anderen Ansatz. Dieser soll in Teil 2 besprochen werden.

Quellen von Teil 1

-Rigid Body Simulation - David Baraff 2001

-> Erklärt wie man Differentialgleichungssysteme simuliert, indem man von Startpunkt aus immer wieder $dx*dt$ -Wert addiert

-<https://www.toptal.com/game/video-game-physics-part-i-an-introduction-to-rigid-body-dynamics>

-> Grundaufbau einer Physikengine: Update-Position+Velocity; Detect-Collision; Resolve-Collision

-<https://github.com/erincatto/box2d-lite>

-> Grundbausteine: Box-Kollision erkennen; Impuls-Resolution für Kontaktpunkte; Impulse-Resolution für Drehgelenke

-<https://www.youtube.com/watch?v=0qqzGkbLbpM> Box2d-Lite - Box-Kollision

-> Erklärung ist nicht so gut weil in die Formeln zwar Zahlen einsetzt aber nicht die Idee dahinter erklärt

-<https://raphaelpriatama.medium.com/sequential-impulses-explained-from-the-perspective-of-a-game-physics-beginner-72a37f6fea05>

-> Erklärt wie man bei einem Kontaktspunkt einen Impuls wirken lässt. Außerdem was MLCP und PGS bedeutet.

-Iterative Dynamics with Temporal Coherence - Erin Catto 2005

-> Für alle Kontaktspunkte/Gelenke wird eine Velocity-Constraint-Gleichung aufgestellt und damit dann die Gleichung $J^*V = \Lambda$ gebildet (Jede Zeile von J ist eine Velocity-Constraint; V =Geschwindigkeit aller Objekte)

-> Per Projected Gauss Seidel (PGS) wird dann der Vektor Λ berechnet. Dieser Vektor gibt für alle Constraints an, wie stark die Constraint-Kraft wirken muss, um die Kollision aufzulösen

-> Da PGS ein iteratives Verfahren ist und da die Kontaktspunkte vom letzten Zeitschritt nochmal genommen werden heißt das hier Iterative Dynamics mit zeitlicher Kohärenz

-> Offene Fragen: Bei Box2D-Lite sehe ich kein iteratives Gaus-Seidel. Wie wurde dort der Betrag der Constraint-Kraft berechnet?

-Building a 2D Game Physics Engine: Using HTML5 and JavaScript (English Edition)

-> Erklärt wie 2D-Körper miteinander kollidieren können.

-> Zeigt sowohl die PositionCorrection als auch ImpulsResponse

-> Liefert die Formel für jN und jT (Impulsantwort in Kollisionsnormalenrichtung/Tangente) welche ähnlich aussieht wie bei Box2D-Lite

-<https://www.myphysicslab.com/engine2D/collision-en.html>

-> Gute Herleitung für die Impulseantworts-Formel jN

-<https://www.myphysicslab.com/engine2D/collision-methods-en.html>

-> Erklärt was Single Resolve Collision(Iterativ) und Multiple Resolve Collision/Hybrid-Ansatz ist. Was sind die Stärken/Schäden.

-<https://www.myphysicslab.com/engine2D/contact-en.html>

-> Erklärt, wie man die Kontaktkraft mit den Iterativ wachsenden $a=A*f+b$ -Gleichungssystem löst.

-> Offene Frage: Sowohl Gelenke haben ein Velocity-Constraint-Gleichungssystem und die Kontaktkräfte haben ein Acceleration-Constraint-Gleichungssystem -> Werden die separat voneinander betrachtet?

-<https://www.myphysicslab.com/develop/docs/Engine2D.html>

-> Sagt, dass seine Physik-Engine aus folgenden Teilen besteht: ImpulsResolution, Kontaktkraft, Kollisionszeitpunkttermittlung, CollisionDetection(Kante-Ecke, Kante-Kante), High Speed Collision, MultiCollision, Connectors

-> Hier steht der Hinweis, dass sowohl für Kontaktkräfte als auch für CollisionImpulse die A-Matrix aufgestellt wird

-<https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/physicstutorials/8constraintsandsolvers/Physics%20-%20Constraints%20and%20Solvers.pdf>

-> Ein Körper hat mehrere Freiheitsgrade wo er sich aufhalten darf. Bsp: 2D-Kugel darf auf beliebiger XY-Position liegen

-> Ein Position-Constraint ist eine Einschränkung wo der Körper sein darf. Die Constraint gibt den Abstand der Position zur Zielposition an. Bsp: Die 2D-Kugel darf nur auf einer bestimmten Position liegen

-> Die Velocity-Jacobi-Matrix gibt an, in welche Richtung die Kugel wandern muss um zum Zielpunkt zu kommen. Lambda gibt an wo groß die Geschwindigkeitsänderung sein muss um zum Zielpunkt zu kommen.

Teil 2: Bewegungsgleichung als Matrix

Ziel dieses Abschnitts

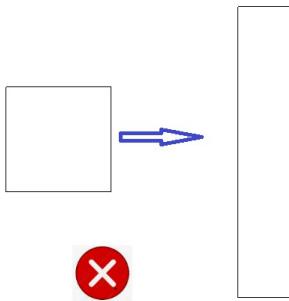
Da in Teil 1 eine einzelne Impulsgleichung für ein Kontaktpunkt nicht gereicht hat, sollen hier nun mithilfe von ein Gleichungssystem mehrere Kontaktpunkte gleichzeitig gelöst werden. Für das Lösen von diesen Gleichungssystem wird Projected Gauss-Seidel verwendet.

Herleitung der Formel

Das Problem mit dem Würfel der an der Wand ohne Drehung abprallen soll

In Teil 1 hatten wir eine Gleichung für ein Impuls hergeleitet, womit wir ein einzelnen Kontaktpunkt lösen können. Wenn ein Würfel aber ohne Drehung gegen eine Wand stößt, dann hat das aber nicht funktioniert, weil erst der eine Impuls gewirkt hat, was eine Drehung verursacht hat. Bei der Berechnung des zweiten Impulses ist dann die Drehung vom ersten Impuls mit in die Formel eingeflossen was dazu führte, dass dort ein anderer Impuls gewirkt hat wodurch dann der Würfel von der Wand mit Drehung abgeprallt ist. Eigentlich muss der Impuls für beide Kontaktpunkte genau gleichgroß sein und genau Richtung $(-1; 0)$ wirken. Hat der Würfel z.B. eine Geschwindigkeit von $(1, 0)$ und ich will mit einer Restitution von 1 (Gummi ohne Reibungsverlust) den Würfel abprallen lassen, dann muss seine Geschwindigkeit nach Anwendung aller Impulse mit der Wand $(-1, 0)$ betragen. Bei zwei Kontaktpunkten heißt das, beide Punkte müssen mit einem Impuls von $(-1, 0)$ gegen die Würfecken drücken um die Geschwindigkeit zu reflektieren.

$$(1,0) - (1,0) - (1,0) = (-1,0)$$



Die Frage lautet nun also woher bekommen wir eine Gleichung für unsere Kontaktpunkte, so dass diese wissen wie viel Impuls angewendet wird so dass der Würfel dann ohne Drehung an der Wand abprallt? Die Antwort dafür liefert Erin Catto im Paper „Iterative Dynamics with Temporal Coherence - Erin Catto 2005“. Der Grundgedanke ist, dass man nicht nur die Geschwindigkeit von einem einzelnen Körper betrachtet und dann daraus sich eine Impulsformel überlegt. Stattdessen nimmt man nun die Geschwindigkeiten von allen Körpern zu einem bestimmten TimeStep-Zeitpunkt und prüft dann für alle Körper gleichzeitig, ob sie mit einem anderen Körper kollidieren und wenn ja, was von allen Körpern gleichzeitig die neuen Geschwindigkeitswerte sind.

Bewegungsgleichungen als Matrix

Wenn ich alle Körper gleichzeitig bewegen will und auch gleichzeitig sicherstellen will, dass ihre Bewegung sich so verhält wie ich es will (Kollisionsvermeidung durch Abprallen), dann muss ich alle Körper als Vektor darstellen. Ich definiere dazu den Vektor V , welcher die aktuellen Geschwindigkeitswerte aller Körper enthält. Außerdem habe ich die Möglichkeit von außen Kräfte auf die Körper wirken zu lassen, indem F_{ext} wirkt. Das sind all die resultierenden Kraftvektoren und Drehmomente auf alle Körper. Eine externe Kraft könnte z.B. die Schwerkraft sein. In unseren Würfel-Wand-Beispiel gibt es keine externe Kraft da wir ohne Gravitation arbeiten.

Wenn unsere Kollisionserkennung mitbekommen hat, dass unser Würfel in die Wand eingedrungen ist, soll die Wand auf den Würfel eine Kraft einwirken lassen, welche den Würfel wieder aus der Wand wegdrückt. Diese Kraft wird hier als F_c (Constraint-Kraft) bezeichnet. F_c ist immer dann ungleich Null, wenn der Würfel sich in die Wand rein bewegt. Das kann dadurch passiert sein, weil jemand dem Würfel initial ein Geschwindigkeitswert gegeben hat, welcher ihn zur Wand bewegen lässt oder eine externe Kraft (z.B: Schwerkraft) hat den Würfel gegen die Wand gedrückt. Die Constraint-Kräfte haben die Aufgabe Kollisionsauflösung und Gelenke/Stabkräfte zu realisieren. Immer wenn der Körper an einer falschen Position oder mit falscher Geschwindigkeit unterwegs ist, dann ist es die Aufgabe der Constraint-Kräfte den Körper zu korrigieren. Wie dieser F_c -Spaltenvektor berechnet wird erklären wir noch. Er enthält so wie der F_{ext} -Vektor auch $3*n$ Floateinträge (pro Körper 2 Floats für die Translationskraft und 1 Float für das Drehmoment).

Die Masse aller Körper wird hier als $3n \times 3n$ -Matrix dargestellt. Es geht darum, dass ich die Bewegungsgleichung ($F = m * a$), welche für einen einzelnen Körper gilt (siehe Bild) nun für alle Körper gleichzeitig ausrechnen will. Deswegen muss die Masse m von jedem Körper in der Matrix zweimal auftauchen und dessen Inertia einmal. Nur so erhalte ich bei der $M * A$ -Multiplikation wieder ein Kraft-Spaltenvektor mit $3n$ Einträgen.

$$V = \begin{pmatrix} v_1 \\ \omega_1 \\ \vdots \\ v_n \\ \omega_n \end{pmatrix} \quad F_c = \begin{pmatrix} f_{c_1} \\ \tau_{c_1} \\ \vdots \\ f_{c_n} \\ \tau_{c_n} \end{pmatrix} \quad F_{ext} = \begin{pmatrix} f_{ext_1} \\ \tau_{ext_1} \\ \vdots \\ f_{ext_n} \\ \tau_{ext_n} \end{pmatrix} \quad M = \begin{bmatrix} m_1 & 0 & 0 \cdots 0 & 0 & 0 \\ 0 & m_1 & 0 \cdots 0 & 0 & 0 \\ 0 & 0 & I_1 & 0 & 0 \\ \vdots & 0 & 0 & m_n & 0 \\ 0 & 0 & 0 \cdots 0 & m_n & 0 \\ 0 & 0 & 0 \cdots 0 & 0 & I_n \end{bmatrix}$$

v1=(Body1.Velocity.X, Body1.Velocity.Y) -> 2 Floats
w1=(Body1.AngularVelocity) -> 1 Float
f=(Force.X, Force.Y) -> 2 Floats
tau=Torque -> 1 Float

n = Body-Anzahl
V = 3*n Floats
F_c = 3*n Floats
F_ext = 3*n Floats
M = [3*n x 3*n]-Matrix

So sieht die Bewegungsgleichung für einen einzelnen Körper aus, wenn man keine Matrix nimmt. $m\ddot{v} = f_c + f_{ext}$
 $I\ddot{\omega} = \tau_c + \tau_{ext}$

$$\dot{V} \approx \frac{V^2 - V^1}{\Delta t} \quad M\dot{V} = F_c + F_{ext}$$

V^1 = Geschwindigkeit von TimeStep1

V^2 = Geschwindigkeit von TimeStep2

$$\begin{array}{c|c|c|c} & 3n & & \\ & \dot{V} & & \\ \hline & 1 & & \\ \hline \begin{matrix} 3n \\ M \end{matrix} & \begin{matrix} 3n \\ 3n \end{matrix} & \begin{matrix} 1 \\ 3n \\ M\dot{V} \end{matrix} & = \begin{matrix} 1 \\ 3n \\ F_c \end{matrix} + \begin{matrix} 1 \\ 3n \\ F_{ext} \end{matrix} \end{array}$$

Nehmen wir mal an ich wüsste wie die Constraint-Kraft F_c aussieht und V^1 , F_{ext} und M sind ja gegeben, dann könnte ich zuerst mit $\dot{V} = M^{-1} * (F_c + F_{ext})$ den \dot{V} (Beschleunigungsvektor) ausrechnen und dann mit $V^2 = V^1 + \dot{V} * dt$ den Geschwindigkeitsvektor für den nächsten TimeStep ausrechnen.

Hinweis: Um die Inverse von der Massematrix, welche ja eine Diagonalmatrix ist, zu bestimmen brauche ich nur von jedem Element aus der Matrix die Inverse bilden.

Beispiel: $A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$ wird zu $A^{-1} = \begin{pmatrix} \frac{1}{1} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{3} \end{pmatrix}$

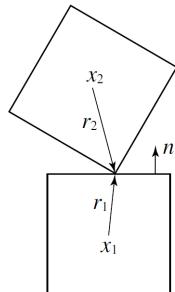
Die Frage ist jetzt also wie kann ich die Constraint-Kraft bestimmen? Die Richtung der Constraint-Kraft wissen wir zumindest. Bei unseren Kontaktpunkten wirkt die Kraft für den Normalimpuls immer in Richtung Kontaktnormalen. Bei Körpern die nicht kollidieren ist die Constraint-Kraft Null.

Somit fehlen uns nur noch die Längen f_c von allen Körpern, die kollidieren.

Über den Constraint-Vektor wissen wir außerdem, dass die resultierende Geschwindigkeit, welche sich aus V_1 und der Constraintkraft ergibt dann so sein muss, dass der Würfel nicht weiter in die Wand rein fliegt. Was wir jetzt also brauchen ist eine Funktion die Prüft, ob der Würfel weiter in Richtung Wand fliegt oder ob er sich davon entfernt. Nur wenn er in Richtung Wand fliegt, soll dann eine Kontaktkraft wirken.

Die Normal-Constraint-Gleichung

Quelle für Normal-Constraint: Iterative Dynamics with Temporal Coherence - Erin Catto 2005
 Ich kann für ein einzelnen Kontaktpunkt prüfen, welchen Abstand die Kontaktpunkte von den beiden Körpern zueinander haben und ich kann auch prüfen, mit welchen Abstand die Kontaktpunkte sich aufeinander zubewegen. Um das zu erklären ist folgendes Bild gegeben:
 Würfel 1 und 2 kollidieren. r_1 und r_2 zeigen von den Zentren zum Kollisionspunkt/Kontaktpunkt.



Schritt 1: Die Position-Constraint C_n misst wie sehr der Kontaktpunkt (rechte untere Ecke) von Würfel 2 in die obere Kante von Würfel 1 eingedrungen ist. Negative Werte bedeuten die Körper kollidieren.

$$C_n = (x_2 + r_2 - x_1 - r_1) \cdot n_1$$

Schritt 2: Die Ableitung nach der Zeit ergibt die Velocity-Constraint:

$$\dot{C}_n = (v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1) \cdot n_1 + (x_2 + r_2 - x_1 - r_1) \cdot \omega_1 \times n_1$$

Für die Ableitungsregel: Siehe Teil 1 'Geschwindigkeit eines Punktes von ein Körper'

Schritt 3: Den V-Vektor extrahieren

Wir ignorieren den zweiten Term, da wir davon ausgehen, dass die Körper sich nur wenig überlappen (zweite Klammer ist Null). Außerdem ersetzen wir n_1 mit n und gehen davon aus, dass n immer von Körper1 zu Körper2 zeigt.

$$\dot{C}_n = (v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1) \cdot n_1 + (x_2 + r_2 - x_1 - r_1) \cdot \omega_1 \times n_1$$

$$\dot{C}_n = (v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1) \cdot n \quad \begin{matrix} n \text{ in die} \\ \text{Klammer rein} \end{matrix}$$

$$\dot{C}_n = v_2 n + (\omega_2 \times r_2) \cdot n - v_1 n - (\omega_1 \times r_1) \cdot n \quad \begin{matrix} a \cdot (b \times c) = (a \times b) \cdot c \end{matrix}$$

$$\dot{C}_n = v_2 n + \omega_2 \cdot (r_2 \times n) - v_1 n - \omega_1 \cdot (r_1 \times n) \quad \begin{matrix} \text{Sortiere die} \\ \text{Plus-Terme um} \end{matrix}$$

$$\dot{C}_n = -v_1 n - \omega_1 \cdot (r_1 \times n) + v_2 n + \omega_2 \cdot (r_2 \times n) \quad \begin{matrix} \text{Extrahiere die} \\ \text{Geschwindigkeits} \\ \text{variablen} \end{matrix}$$

$$J_n V = \begin{pmatrix} -n^T & -(r_1 \times n)^T & n^T & (r_2 \times n)^T \end{pmatrix} \begin{pmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{pmatrix}$$

So wie wir hier jetzt die Position- und Velocity-Constraint-Gleichung hergeleitet haben könnte man das für 2D- als auch 3D-Körper verwenden. Wir wollen uns aber auf 2D konzentrieren. In diesen Fall ergibt das Kreuzprodukt zwischen zwei in der XY-Ebene liegenden Vektoren ein Z-Vektor wo

X und Y 0 ist. Deswegen ist $r_1 \times n$ eine skalare Zahl.

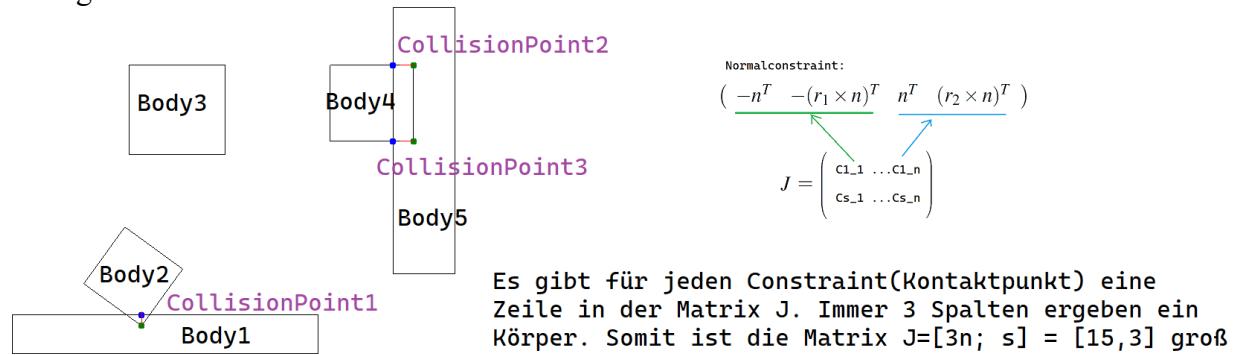
Der V-Vektor enthält 6 Floats ($v_{1.X} \ v_{1.Y} \ w_1 \ v_{2.X} \ v_{2.Y} \ w_2$).

Der J_n -Vektor auch ($-n.X \ -n.Y \ -r_1 \cdot n \ Z \ n.X \ n.Y \ r_2 \cdot n \ Z$).

Dieser J_n -Zeilenvektor hat die Fähigkeit, dass ich als Input die Geschwindigkeit von zwei Körpern, die kollidieren, rein gebe und er ermittelt mir dann durch die Multiplikation von J_n mit V die Relativgeschwindigkeit der beiden Kontaktpunkte in Richtung Kollisionsnormale. Durch das Vorzeichen sehe ich, ob die Kontaktpunkte sich voneinander entfernen oder aufeinander sich zubewegen. Durch den Betrag wie schnell die Kontaktpunkte in Richtung Kollisionsnormale sich bewegen.

Die Jacobi-Matrix J

Wenn ich für alle Körper, die in meiner Simulation vorkommen, den NormalConstraint-Wert ermitteln will, dann kann ich eine Matrix J definieren welche für 5 Körper und 3 Kontaktpunkte wie folgt aussieht:



Body1		Body2		Body3		Body4		Body5	
-n1.X	-n1.Y	-r1N1	n1.X	n1.Y	r2N1	0	0	0	0
0	0	0	0	0	0	0	-n2.X	-n2.Y	-r1N2
0	0	0	0	0	0	0	-n3.X	-n3.Y	-r1N3

← CollisionPoint1
 ← CollisionPoint2
 ← CollisionPoint3

Nun muss ich nur noch den Velocity-Vektor V mit der Velocity-Constraint-Matrix J multiplizieren und erhalte damit ein Spaltenvektor mit s Zeilen, welcher mir pro Kontaktpunkt/Constraint sagt, wie die Relativgeschwindigkeit der jeweiligen Kontaktpunkte ist. N ist die Bodyanzahl. S die Kontaktpunktzahl. V enthält $3 \cdot N$ Floateinträge. Pro Körper: Geschwindigkeit des Zentrums ($v.X, v.Y$) und Rotationsgeschwindigkeit Omega. Für unser Beispiel hier enthält der Spaltenvektor $J \cdot V$ 3 Einträge. Wenn beim $J \cdot V$ -Spaltenvektor Zahlen ungleich 0 stehen heißt das, die pro Zeile angegebenen Kontaktpunkte bewegen sich aufeinander zu oder voneinander weg.

$$V = \begin{pmatrix} v_1 \\ \omega_1 \\ \vdots \\ v_n \\ \omega_n \end{pmatrix}$$

$$J = \begin{pmatrix} -n^T & -(r_1 \times n)^T & n^T & (r_2 \times n)^T \end{pmatrix}$$

$$J = \begin{pmatrix} c_{1,1} & \dots & c_{1,n} \\ c_{2,1} & \dots & c_{2,n} \\ \vdots & & \vdots \\ c_{s,1} & \dots & c_{s,n} \end{pmatrix}$$

3n
 V
 1
 J s
 J · V

Der Lambda-Spaltenvektor

Wenn zwei Körper kollidieren und ich will, dass die Kollision dadurch aufgelöst wird, indem ich die Geschwindigkeit in Richtung Kontaktnormale reflektiere, dann kann ich das wie auch schon in Teil 1 über ein Impuls realisieren. Für unsere erste Überlegung gehen wir davon aus, dass der Restitutionkoeffizient 0 ist. D.h. Der Würfel fliegt gegen die Wand und die Wand absorbiert einfach nur seine gesamte Bewegungsenergie und dann bleibt die Geschwindigkeit 0 und er verbleibt in der Wand stecken. Wenn das unser Ziel ist, dann können wir folgende Überlegung anstellen. Zum Zeitschritt 1 wird die Kollision festgestellt und die Geschwindigkeit aller Körper ist im Spaltenvektor V1. Setzen wir V1 in die Gleichung $J*V1$ ein, dann sehen wir, dass dort ein Vektor ungleich 0 rauskommt da der Würfel sich gerade in Richtung Wand bewegt. Wenn wir nun im nächsten Zeitschritt eine korrigierte Geschwindigkeit V2 wollen, dann muss für V2 gelten: $J*V2=0$. Wenn dort 0 rauskommt bedeutet das der Würfel bewegt sich gegenüber der Kollisionsnormale nicht, er ruht. Damit er in die ruhende Position kommt, müssen wir ein Impuls auf den Würfel anwenden. Folgende Gleichung muss erfüllt werden:
 $J*(V1+A*dt) = 0$. A ist hier der Spaltenvektor mit 3n Einträgen, welcher für jeden Körper die Beschleunigung angibt, welcher wirken muss, damit $J*V2$ dann 0 ergibt. $V2=V1+A*dt$.

Wenn man sich nun die Zeilen von J ansieht und die Gleichung $Zeile_von_J*V = Skalarer_Wert$ anschaut, dann stellt man fest, dass diese Gleichung genau so aussieht wie wenn man eine Ebenengleichung hat, welche durch den Nullpunkt geht und wo man ein Punkt mit der Ebenennormale multipliziert um somit dann festzustellen, welchen Abstand der Punkt zur Ebene hat.

In diesen Fall hat jede Zeile von der J-Matrix 3*n Einträge. Es handelt sich hier also um eine 3n-Dimensionale Hyperebene und auch der Punkt V befindet sich im 3n-Dimensionalen Raum.

Der Gedanke, den ich hier heraus arbeiten will ist der, dass jede J-Zeile als Hyperebene betrachtet werden kann und dass die Normale von dieser Ebenen auch der J-Zeile entspricht. Eine J-Zeile ist also ein Richtungsvektor! Befindet sich der V-Punkt auf der Ebene dann ergibt $J*V$ gleich 0. Befindet sich aber V1 außerhalb der Ebene dann kann ich ein Punkt V2 dadurch auf der Ebene bestimmen, indem ich $V1+J_Zeile * \Lambda$ rechne. Ich verschiebe also den V1-Punkt in Richtung Ebenennormale auf kürzesten Weg zurück zur Ebene zum V2-Punkt. Dieser J-Zeil-Richtungsvektor ist genau die Richtung, in welche unsere Beschleunigung wirken soll, welcher unser V1 korrigiert. Da für die Beschleunigung die Gleichung $F=M*A$ (A ist vielfaches von F) gilt, heißt dass auch, dass nicht nur A parallel zur J_Zeile ist sondern auch die Kraft F, welche am Kontaktspunkt die beiden kollidierten Körper auseinander drückt.

Aus all diesen J-Zeilen-Hyperebene-Überlegungen soll eine wichtige Gleichung hervorgehen:
Die Kontaktkraft/Constraint-Kraft ist das Vielfache von einer J-Zeile. Weil das so ist, ist also ihre Richtung aufgrund der bekannten J-Matrix auch bekannt und nur ihre Länge ist unbekannt. Diese unbekannte Constraint-Kraft-Länge wird Lambda genannt. Lambda ist ein Spaltenvektor der Länge s (Anzahl der Constraints/Kontaktpunkte). Wir definieren den Spaltenvektor Fc dadurch, dass er die Summe aller Constraintkräfte ist. Jede Zeile von der J-Matrix gibt die Richtung für eine Constraintkraft an und jede J-Zeile wird um ein skalaren Wert aus dem Lambda-Spaltenvektor gewichtet. Besteht die J-Matrix aus 3 Zeilen, dann kann ich Fc so ermitteln:

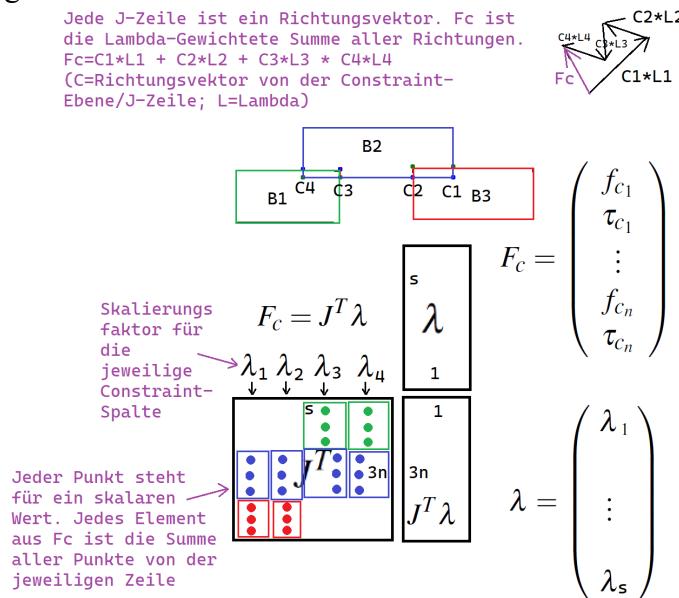
$$J = \begin{bmatrix} Row_1 \\ Row_2 \\ Row_3 \end{bmatrix} \quad \lambda = \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} \quad F_C = Row_1 * \lambda_1 + Row_2 * \lambda_2 + Row_3 * \lambda_3$$

Fc ist also die Summe aller Lambda-Gewichteten Constraint-Ebenennormalen. Da jede J-Zeile ein Richtungsvektor ist, kann ich Fc auch dadurch berechnen, indem ich die J-Matrix transponiere und sie mit den Spaltenvektor Lambda multipliziere.

Somit ergibt sich für F_c nun diese Gleichung: $F_c = J^T \lambda$

Wenn ich die F_c (Constraintkraft) für ein Zeitschritt dt wirken lasse, dann heißt das: Alle Kontaktpunkte stoßen sich gleichzeitig gegenseitig ab. Ob die resultierende Kontaktgeschwindigkeit aller Körper dann aber Null ist hängt von Lambda ab.

Beispiel: Es gibt 3 Körper (Body1=Grün, Body2=Blau, Body3=Rot) und 4 Kontaktpunkte. Die Constraint-Kraft ist die Summe der 4 Kontaktkrafte. Der blaue Würfel bewegt sich wegen der Schwerkraft in den grünen und roten Würfel rein.



Dieses Bild soll erklären: Wenn jede J-Zeile ein Kraftvektor ist und ich will die Summe über alle J-Zeilen bilden, dann muss ich dazu die J-Matrix transponieren und sie dann mit einem Spaltenvektor Lambda multiplizieren um die Summe zu bilden. Jeder Eintrag vom Lambda-Vektor ist dann der Wichtungsfaktor für jeweils eine J-Zeile/Kontaktpunktkraft.

Constraint-Bias

Im vorherigen Beispiel ist ein Würfel gegen eine Wand geflogen wo der Restitutionswert 0 ist. Die Geschwindigkeit des Würfels sollte nach der Kollision somit 0 sein. Deswegen haben wir $V2$ dadurch definiert, indem wir gesagt haben das $J^*V2=0$ ergeben soll. Wenn wir aber eine Kollision von einem Würfel mit einer Wand simulieren wollen, wo der Würfel an der Wand abprallt (Restitution muss größer 0 sein), dann definieren wir $V2$ dadurch, indem J^*V2 gleich ein von mir vorgegebenen Biaswert (Spaltenvektor mit s Einträgen) entsprechen muss:

$$JV^2 = \zeta$$

Wenn wir wollen, dass die Kontaktstellen mit einer Restitution von 1 (0% Reibungsverlust) abprallen, dann muss der Bias-Wert so gewählt werden, dass $Bias=-J^*V1$ gilt. Will ich eine beliebige Restitution e nutzen, dann definiere ich Bias über $Bias=-J^*V1*e$. Mit dem Bias-Wert lege ich fest, welche Relativgeschwindigkeit die Kontaktstellen haben sollen, nachdem der Korrekturimpuls gewirkt hat. Bei einer Normalconstraint will ich, dass die Kontaktstellen voneinander abprallen so dass ich die Formel $Bias=-J^*V1*e$ verwende. Habe ich aber eine Distanzconstraint, dann will ich, dass zwei Kontaktstellen sich vom Abstand her überhaupt nicht verändert. Für so eine Constraint definiere ich $Bias=0$. Je nachdem, was für eine Constraint ich habe, habe ich also eine andere Formel, wie ich Bias definiere.

Um auf das Beispiel mit der Hyperebene zurück zu kommen. Jede Zeile von der J-Matrix ist die Richtung der jeweiligen Constraintkraft/Ebenennormale und jeder Eintrag vom Bias-Spaltenvektor ist der Abstand der Hyperebene zum Nullpunkt. Mit $J_Zeile*V=Bias$ kann ich also prüfen, ob ein

beliebiger V-Hyperpunkt auf der jeweiligen Constraint-Hyperebene liegt. Ziel ist es ein V2-Punkt zu finden, welcher Gleichzeitig auf allen Constraint-Ebenen liegt (gemeinsamer Schnittpunkt aller Constraint-Hyperebenen).

Beschränkung der Constraint-Kraft

Wenn die Kontaktpunkte sich aufeinander zubewegen, dann wollen wir eine Normal-Constraint-Kraft wirken lassen. Bewegen die Kontaktpunkte sich voneinander weg, dann soll keine Kraft wirken, welche der Geschwindigkeit entgegen wirkt. Ich erreiche diese Einschränkung, in welche Richtung und mit welcher Stärke eine Constraint-Kraft wirkt dadurch, indem ich für Lamba Min-Max-Schranken definiere:

$$\lambda_i^- \leq \lambda_i \leq \lambda_i^+, \forall i \in [1, s]$$

Für unser Kontakt-Point-Constraint (Normalconstraint) ist die Min-Max-Schranke dann Lambda=(0..Unendlich)

Diese Schranken erlauben mir zwei Arten von Constraints:

Gleichheits-Constraints: Drehgelenk, Gardinenstange $(\lambda^-, \lambda^+) = (-\infty, \infty)$

Ungleichheits-Constraints: Kontaktstellen, Gelenkwinkel Einschränkungen $(\lambda^-, \lambda^+) = (0, \infty)$

Bestimmung der Constraint-Kräfte

Wir haben Teil 2 damit begonnen, indem wir festgestellt haben, dass wir die Impuls-Kraft von mehreren Kontaktstellen gleichzeitig lösen müssen, wenn wir einen Würfel gegen eine Wand fliegen lassen wollen und ihn ohne Rotation abprallen lassen wollen. Dazu haben wir dann die Bewegungsgleichung in Matrix-Schreibweise aufgestellt und dort in der Gleichung ist dann aber der Constraint-Kraft-Spaltenvektor aufgetaucht, wo wir uns erst mal Gedanken machen mussten, wie wir diesen Spaltenvektor denn berechnen können. Wir kehren nun zur Bewegungsgleichung zurück und unternehmen einen erneuten Anlauf, um nun endlich den Impuls von allen Kontaktstellen gleichzeitig zu bestimmen.

Folgende Dinge haben wir über die Bewegungsgleichung und seine Bestandteile gelernt:

$$\dot{V} \approx \frac{V^2 - V^1}{\Delta t} \quad M\dot{V} = F_c + F_{ext}$$

Bewegungsgleichung
als Matrix

V^1 = Geschwindigkeit vor Anwendung von F_c und F_{ext}

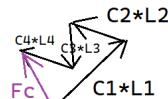
V^2 = Geschwindigkeit nach Anwendung von F_c und F_{ext}

Normalconstraint:
 $\begin{pmatrix} -n^T & -(r_1 \times n)^T & n^T & (r_2 \times n)^T \end{pmatrix}$
 $J = \begin{pmatrix} c_{1,1} & \dots & c_{1,n} \\ \vdots & \ddots & \vdots \\ c_{s,1} & \dots & c_{s,n} \end{pmatrix}$
 Jede Zeile von J steht für
ein Constraint-Hyperebenen-
Normalenvektor

 3 Spalten von J
stehen für ein
Körper

$$F_c = J^T \lambda$$

Mit dieser Formel berechne ich die Lambda-Gewichtete Summe der Constraint-Kräfte



Durch die Min-Max-Lambda-Schranke kann ich dafür sorgen, dass die Impulsgröße immer nur die Kontaktstellen auseinander drückt aber niemals sie zusammen zieht

$$\lambda_i^- \leq \lambda_i \leq \lambda_i^+, \forall i \in [1, s]$$

Mit den Bias-Wert gebe ich vor, welche Relativgeschwindigkeit die Kontaktstellen nach dem Korrekturimpuls haben sollen. Sie sieht für jede Constraint-Art anders aus. Beispiel:

$$\zeta = -J V^1 e \quad \text{Bias bei der Normalconstraint mit Restitution } e$$

$$\zeta = 0 \quad \text{Bias bei der Distanceconstraint}$$

Mit diesen Wissen unternehmen wir nun einen erneuten Versuch die Constraint-Kraft F_c zu bestimmen um somit nun eine drehfreie Würfelabprallung zu realisieren.

Hinweis: Lambda und V2 ist unbekannt. Der Rest ist gegeben.

$$M\dot{V} = F_c + F_{ext}$$

$$\text{Ersetze } F_c \text{ mit: } F_c = J^T \lambda$$

$$M\dot{V} = J^T \lambda + F_{ext}$$

$$\text{Ersetze } \dot{V} \text{ mit } \dot{V} \approx \frac{V^2 - V^1}{\Delta t} \text{ und multipliziere dann mit } \Delta t$$

$$M(V^2 - V^1) = \Delta t(J^T \lambda + F_{ext})$$

$$\text{Links: M in Klammer und dann durch } \frac{1}{\Delta t} \text{ ersetzen}$$

$$\frac{1}{\Delta t} M V^2 - \frac{1}{\Delta t} M V^1 = (J^T \lambda + F_{ext})$$

$$-F_{ext}$$

$$\frac{1}{\Delta t} M V^2 - \frac{1}{\Delta t} M V^1 - F_{ext} = J^T \lambda$$

$$\bullet M^{-1}$$

$$\frac{1}{\Delta t} V^2 - \frac{1}{\Delta t} V^1 - M^{-1} F_{ext} = M^{-1} J^T \lambda$$

$$\text{Links: Klammer um den 2. und 3. Plusterm } \bullet J$$

$$\frac{1}{\Delta t} JV^2 - J\left(\frac{1}{\Delta t} V^1 + M^{-1} F_{ext}\right) = J M^{-1} J^T \lambda$$

$$\text{Ersetze } JV^2 = \zeta \quad \text{Vertausche beide Seiten vom Gleichheitszeichen}$$

$$JM^{-1}J^T \lambda = \frac{1}{\Delta t} \zeta - J\left(\frac{1}{\Delta t} V^1 + M^{-1} F_{ext}\right)$$

A B Ersetze die bekannten Terme durch A und B

A

B

$$\boxed{A \lambda = B}$$

Durch Umstellen wurde die Gleichung A*Lambda=B ermittelt. Um nun zu prüfen, dass ich beim Umstellen der Gleichungen keinen Fehler gemacht habe, prüfe ich bei allen Matrix-Multiplikationen, ob immer gilt: Matrix1-Spaltenanzahl == Matrix2-Zeilenzahl

$$JM^{-1}J^T \lambda = \frac{1}{\Delta t} \zeta - J\left(\frac{1}{\Delta t} V^1 + M^{-1} F_{ext}\right)$$

$$\boxed{A \lambda = B}$$

$$\begin{array}{|c|c|} \hline & \begin{matrix} 3n \\ M^{-1} \\ 3n \end{matrix} \\ \hline \begin{matrix} 3n \\ J \end{matrix} & \begin{matrix} 3n \\ s \\ 3n \\ J M^{-1} \end{matrix} \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline & \begin{matrix} 3n \\ F_{ext} \\ 1 \\ \hline 1 \end{matrix} \\ \hline \begin{matrix} 3n \\ M^{-1} \\ 3n \\ M^{-1} F_{ext} \end{matrix} & \begin{matrix} 3n \\ M^{-1} F_{ext} \\ 1 \end{matrix} \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline & \begin{matrix} 3n \\ J^T \\ s \end{matrix} \\ \hline \begin{matrix} 3n \\ J M^{-1} \\ s \end{matrix} & \begin{matrix} 3n \\ J M^{-1} J^T \\ s \end{matrix} \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline & \begin{matrix} 3n \\ M^{-1} F_{ext} \\ 1 \\ \hline 1 \end{matrix} \\ \hline \begin{matrix} 3n \\ J \\ s \end{matrix} & \begin{matrix} 3n \\ J \\ s \\ J(\cdot) \end{matrix} \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline & \begin{matrix} s \\ \lambda \\ 1 \\ \hline 1 \end{matrix} \\ \hline \begin{matrix} s \\ J M^{-1} J^T \\ s \end{matrix} & \begin{matrix} s \\ A \lambda \\ 1 \\ \hline 1 \end{matrix} \\ \hline \end{array}$$

$$\begin{bmatrix} 1 \\ \frac{1}{\Delta t} \zeta \end{bmatrix} - \begin{bmatrix} 1 \\ s \\ J(\cdot) \end{bmatrix} = \begin{bmatrix} 1 \\ s \\ B \end{bmatrix}$$

$$V = \begin{pmatrix} v_1 \\ \omega_1 \\ \vdots \\ v_n \\ \omega_n \end{pmatrix} \quad J \text{ is the } s \text{-by-} 3n \text{ Jacobian.}$$

$$F_c = J^T \lambda \quad \text{Normalconstraint: } \begin{pmatrix} -n^T & -(r_1 \times n)^T & n^T & (r_2 \times n)^T \end{pmatrix}$$

$$J = \begin{pmatrix} C1_1 & \dots & C1_n \\ Cs_1 & \dots & Cs_n \end{pmatrix}$$

$$F_c = \begin{pmatrix} f_{c_1} \\ \tau_{c_1} \\ \vdots \\ f_{c_n} \\ \tau_{c_n} \end{pmatrix} \quad F_{ext} = \begin{pmatrix} f_{ext_1} \\ \tau_{ext_1} \\ \vdots \\ f_{ext_n} \\ \tau_{ext_n} \end{pmatrix}$$

$$M = \begin{bmatrix} m1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & m1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & I1 & \ddots & 0 & 0 & 0 \\ \vdots & 0 & 0 & \ddots & m_n & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & m_n & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & I_n \end{bmatrix}$$

$$\begin{aligned} C1_1 &= (-n, X, -n, Y, -r1N, Z) \\ C1_n &= (n, X, n, Y, r2N, Z) \\ s &= \text{Constraint-Anzahl} \\ n &= \text{Body-Anzahl} \\ I &= \text{Interia (Skalar)} \\ M &= 3n \times 3n \text{ Matrix} \\ \text{Lambda} &= \text{Spaltenvektor mit } s \text{ Einträgen} \\ F_{ext} / F_c &= \text{Spaltenvektor mit } 3n \text{ Einträgen} \\ \zeta &= \text{Bias-Spaltenvektor mit } s \text{ Einträgen} \end{aligned}$$

Matrizengleichungen für 2D-Körper

Ich stelle fest: Sowohl auf der linken Seite der roten Gleichung als auch auf der rechten Seite kommt ein Spaltenvektor der Länge s raus. Damit kann ich nun also Lambda per Projected Gauss-Seidel (PGS) lösen. Wie man das macht kommt im Praxis-Abschnitt. Wir tun jetzt kurz mal so, dass Lambda nun ermittelt wurde.

Bestimmung der neuen Geschwindigkeitswerte unter Nutzung der Constraint-Kraft

Nachdem ich nun Lambda per Projected Gaus-Seidel ermittelt habe berechne ich noch V^2 indem ich wieder die Ausgangsgleichung nach V^2 umstelle und diesmal das nun bekannte Lambda verwende:

$$M(V^2 - V^1) = \Delta t(J^T \lambda + F_{ext})$$

$$M V^2 - MV^1 = \Delta t(J^T \lambda + F_{ext})$$

$$M V^2 = \Delta t(J^T \lambda + F_{ext}) + MV^1$$

$$V^2 = M^{-1} \Delta t(J^T \lambda + F_{ext}) + V^1$$

Zusammenfassung wie wir über Matrizen und PGS Lambda und V^2 ermitteln

Die 4 Schritte, um alle Kollisionen aufzulösen sind hier nun zu sehen:

$$M(V^2 - V^1) = \Delta t(J^T \lambda + F_{ext})$$

Unbekannt: Lambda und V^2

Schritt 1: Stelle nach Lambda um A λ = B $J M^{-1} J^T \lambda = \frac{1}{\Delta t} \zeta - J(\frac{1}{\Delta t} V^1 + M^{-1} F_{ext})$

Schritt 2: Löse per PGS das Lambda in der roten Gleichung

Schritt 3: Löse V^2 über die grüne Gleichung $V^2 = M^{-1} \Delta t(J^T \lambda + F_{ext}) + V^1$

Schritt 4: Nimm das V^2 Array und weise die Werte daraus den Velocity/AngularVelocity-Werten von allen Körpern zu

So sieht dann die Umsetzung dieser 4 Schritte in der PhysicEngine aus:

```

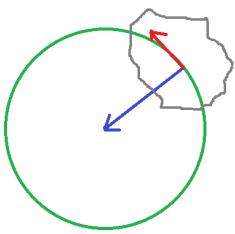
29  public EquationOfMotionData(List<IRigidBody> bodies, RigidbodyCollision[] collisions, float dt, int pgsIterationCount)
30  {
31      float inv_dt = dt > 0.0f ? 1.0f / dt : 0.0f;
32
33      //Schritt 1: Gleichungssystem aufstellen, womit ich Lambda lösen kann
34      this.J = MatrixCreator.GetNormalConstraintMatrixJ(bodies, collisions);
35      this.mInverse = MatrixCreator.GetInverseMassMatrix(bodies);
36      this.jTransposed = J.Transpose();
37      this.bias = MatrixCreator.GetNormalConstraintBias(collisions);
38      this.V = MatrixCreator.GetV(bodies);
39      this.fExt = MatrixCreator.GetFext(bodies);
40
41      //A*Lambda=B -> Gesucht: Lambda
42      this.A = J * mInverse * jTransposed;
43      this.B = inv_dt * bias - J * (inv_dt * V + mInverse * fExt);
44
45      //Schritt 2: Lambda per PGS ermitteln
46      this.initialLambda = Matrix.GetColumnVectorWithZeros(collisions.Length);
47      this.minLambda = MatrixCreator.GetNormalConstraintMinLambda(collisions.Length);
48      this.maxLambda = MatrixCreator.GetNormalConstraintMaxLambda(collisions.Length);
49      this.lambda = ProjectedGaussSeidel.Solve(A, B, initialLambda, minLambda, maxLambda, pgsIterationCount);
50
51      //Schritt 3: Mit Lambda ein V ermitteln, was alle Constraints erfüllt
52      this.vNew = mInverse * dt * (jTransposed * lambda + fExt) + V;
53  }
54
55  //Schritt 4: Die Geschwindigkeitswerte aller Körper korrigieren
56  public void SetVelocityValues(List<IRigidBody> bodies)
57  {
58      float[] vNew = this.vNew.GetColumn(0);
59
60      for (int i = 0; i < bodies.Count; i++)
61      {
62          int roundDecimalPlaces = 7; //Runde 7 Stellen nach dem Komma
63          bodies[i].Velocity.X = MathHelp.Round(vNew[i * 3 + 0], roundDecimalPlaces);
64          bodies[i].Velocity.Y = MathHelp.Round(vNew[i * 3 + 1], roundDecimalPlaces);
65          bodies[i].AngularVelocity = MathHelp.Round(vNew[i * 3 + 2], roundDecimalPlaces);
66      }
67  }
```

Siehe: PhysicEngine/CollisionResolition/EnterTheMatrix/EquationOfMotionData.cs

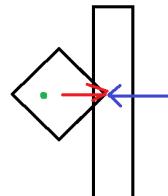
Was genau ist denn nun ein Constraint? Wie kann man sich das vorstellen?

Immer dann, wenn wir für einen vorgegebenen Punkt eines Körpers einen gültigen Bereich festlegen wollen, in dem sich dieser Punkt aufhalten darf, dann brauche ich eine sogenannte Position-Constraint, welche den Abstand vom Punkt zum gültigen Bereich angibt. Leite ich diese Position-Constraint nach der Zeit ab, bekomme ich eine Velocity-Constraint. Extrahiere ich aus dieser Gleichung den V-Vektor, bekomme ich die J-Zeile. Diese J-Zeile ist die Normale von einer Hyperebene. Zusammen mit dem Bias-Term ist $J^*V^2 = \text{Bias}$ eine Ebenengleichung.

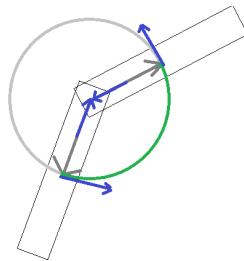
Beispiele für Constraint-Arten.



Distanz-Constraint



Normal-Constraint



Revolute-Joint-Constraint

Das Zentrum vom Körper darf sich nur auf der grünen Kreisbahn aufhalten. Die blaue Constraint-Kraft drückt es in die Kreisbahn, wenn der Abstand nicht stimmt.

Grün: Erlaubter Bereich des Kontaktpunktes laut der Position-Constraint

Der rechte Würfeleckpunkt darf sich nach dem Zusammenstoß mit der Wand nur an dem grünen Punkt aufhalten. Die Normal-Constraint-Kraft drückt die Würfecke dort hin.

Blau: Richtung der Velocity-Constraint-J-Zeile.
In diese Richtung wirkt die korrigierende Kraft.

Der Winkel zwischen den beiden Körper darf nur im grünen Bereich liegen. Eine Distanzkraft drückt einerseits die Drehpunkte zusammen. Drehgelenkskraft sorgt dafür, dass nur im erlaubten Winkelbereich das Gelenk genutzt wird.

Die Position-Constraint beschreibt einen grünen Bereich, indem sich ein von mir festgelegter Punkt eines Körpers aufhalten darf. Diese Beschreibung des grünen Aufenthaltsbereich wird über eine Implizite Funktion (https://de.wikipedia.org/wiki/Satz_von_der_impliziten_Funktion) gemacht. Durch ableiten nach der Zeit und extrahieren von V erhalte ich die blaue Hyperebene, dessen Normale der Constraint-Kraft-Richtung entspricht. Mit dem Constraint-Bias-Wert gebe ich vor, mit welcher Geschwindigkeit die Kontaktpunkte sich aufeinander zu/ voneinander weg bewegen sollen. Beim Distanz-Constraint soll der Abstand gleich bleiben also ist Bias Null. Beim Normalconstraint sollen die Kontaktpunkte voneinander weg fliegen, also ist Bias ungleich Null.

Das Bild, was man im Kopf mitnehmen sollte ist:

- Eine Position-Constraint ist eine grüne Implizite Funktion (Kreis, Punkt, Linie, Kreissegment,...)
- Eine J-Zeile ist ein blauer Constraint-Kraftvektor. Er drückt den Kontaktpunkt in den grünen Bereich rein.
- J-Zeile zusammen mit Bias ergibt eine Hyperebene, welche die gültigen Geschwindigkeitswerte definiert. Wir Projizieren V1 senkrecht auf die Hyperebene, so dass V2 letztendlich nur an genau ein Punkt auf dieser Hyperebene liegt.

Gleichungssystem per (Projected) Gauss-Seidel lösen

Gauss-Seidel

Mit Gaus-Seidel kann ich ein lineares Gleichungssystem lösen indem ich pro Iterationsschritt über alle Zeilen gehe und dann jeweils nach einer gesuchten Variable umstelle. Für all die noch unbekannten Variablen starte ich mit ein Schätzwert, welcher dann pro Schritt verbessert wird.

```
13 ✓ //So funktioniert Gauss-Seidel.
14 | Das Beispiel ist von hier: https://miro.medium.com/v2/resize:fit:828/format:webp/1\*Xh009fKIPaYGhe2wi3wFkg.png
15 | [Fact]
16 |   | 0 Verweise
17 | ✓ public void SolveLinearSystemWithTwoVariables()
18 | {
19 |   | //Gegeben ist folgendes Gleichungssystem
20 |   | // (1) 3x + 1y = 5
21 |   | // (2) 2x + 2y = 6
22 |   | //Gesucht sind x und y
23 |   | //Bei Gauss-Seidel löse ich immer abwechselnd nach einer gesuchten Variable
24 |   | //Die Gleichungen nach x und y umstellen:
25 |   | // (1) x=(5-y)/3      -> Gleichung (1) wurde nach x umgestellt
26 |   | // (2) y=(6-2x)/2    -> Gleichung (2) wurde nach y umgestellt
27 |
28 |   int iterations = 10;
29 |   float startValueX = 0;
30 |   float startValueY = 0;
31 |   float x = startValueX;
32 |   float y = startValueY;
33 |   for (int i = 0; i < iterations; i++)
34 |   {
35 |     x = (5 - y) / 3;
36 |     y = (6 - 2 * x) / 2;
37 |
38 |   //Prüfe ob es stimmt:
39 |   float row1 = 3 * x + y;      //Erwartung für row1: 5
40 |   float row2 = 2 * x + 2 * y; //Erwartung für row2: 6
41 |
42 |   row1.Should().BeApproximately(5, 0.0001F);
43 |   row2.Should().BeApproximately(6, 0.0001F);
44 }
```

Projected Gauss-Seidel (PGS)

PGS macht das Gleiche wie Gauss-Seidel nur dass dort noch x bei jeden Schritt geclampt wird:

```
5 | //Löst das Gleichungssystem A*x=B
6 | | 1 Verweis
7 | internal static class ProjectedGaussSeidel
8 | {
8 | | 1 Verweis
9 | | public static Matrix Solve(Matrix A, Matrix B, Matrix initialX, Matrix minX, Matrix maxX, int iterations)
10 | | {
11 | |   if (initialX.IsColumnVector() == false) throw new ArgumentException("initialX must be a ColumnVector");
12 | |   if (A.ColumnCount != initialX.RowCount) throw new ArgumentException("A.ColumnCount must be equal to initialX.RowCount");
13 | |   if (initialX.Size != B.Size) throw new ArgumentException("RowCount from initialX and B must be equal");
14 |
15 | |   var X = new Matrix(initialX);
16 | |   for (int i = 0; i < iterations; i++)
17 | |   {
18 | |     for (int y=0;y<initialX.RowCount;y++)
19 | |     {
20 | |       //Löse X[y] unter Nutzung von Zeile y
21 | |       float sum = 0;
22 | |       for (int j=0;j<initialX.RowCount;j++)
23 | |       {
24 | |         if (j != y)
25 | |           sum += A[j, y] * X[0, j];
26 | |       }
27 | |       X[0, y] = (B[0, y] - sum) / A[0, y];
28 | |       X[0, y] = MathHelp.Clamp(X[0, y], minX[0, y], maxX[0, y]);
29 | |     }
30 |
31   }
32 }
33 }
```

Bei Zeile 17 geht die Schleife über alle Zeilen vom Gleichungssystem. Die erste Zeile wird nach der ersten unbekannten X-Variable umgestellt. Die zweite Zeile nach der zweiten X-Variable usw.

Testfälle ohne Schwerkraft

Würfel gegen die Wand

Siehe im Quelltext: PhysicEngine.UnitTests.NoGravityTests.cs/CubeAgainstWall(...)

Bei Teil 1 hat der Würfel-Test noch nicht geklappt weil der Impuls für die beiden Kontaktpunkte nicht gleichzeitig gelöst wurde. Beim Matrix-Ansatz geht es nun. Der Würfel fliegt ohne Drehung mit einer Geschwindigkeit von 0.1 gegen die Wand und wird ohne Drehung reflektiert. Bei den gelb markierten Zahlen sieht man, dass der Würfel erst mit 0.1 und dann mit -0.1 fliegt und dass die AngularVelocity 0 bleibt. So sehen die verwendeten Matrizen aus:

V=

```
+0,100000| <- Cube.Velocity.X
+0,000000| <- Cube.Velocity.Y
+0,000000| <- Cube.AngularVelocity
+0,000000| <- Wall.Velocity.X
+0,000000| <- Wall.Velocity.Y
+0,000000| <- Wall.AngularVelocity
```



fExt=

```
| 0,000000|
| 0,000000|
| 0,000000|
| 0,000000|
| 0,000000|
| 0,000000|
```

mInverse=

```
+1,000000 0,000000 0,000000 0,000000 0,000000 0,000000|
| 0,000000 +1,000000 0,000000 0,000000 0,000000 0,000000|
| 0,000000 0,000000 +0,000306 0,000000 0,000000 0,000000|
| 0,000000 0,000000 0,000000 0,000000 0,000000 0,000000|
| 0,000000 0,000000 0,000000 0,000000 0,000000 0,000000|
| 0,000000 0,000000 0,000000 0,000000 0,000000 0,000000|
```

bias=

```
+0,100000|
+0,100000|
```

$$J M^{-1} J^T \lambda = \frac{1}{\Delta t} \zeta - J \left(\frac{1}{\Delta t} V^I + M^{-1} F_{ext} \right)$$

A B

jTransposed=

```
-1,000000 -1,000000 |
| 0,000000 0,000000 |
| -70,000000 +70,000000 |
| +1,000000 +1,000000 |
| 0,000000 0,000000 |
| +87,000000 -53,000000 |
```

J=

```
-1,000000 0,000000 -70,000000 +1,000000 0,000000 +87,000000|
| -1,000000 0,000000 +70,000000 +1,000000 0,000000 -53,000000|
```

lambda=

```
+0,002000|
| +0,002000|
| +0,002000|
```

A=

```
+2,500000 -0,500000 |
| -0,500000 +2,500000 |
```

vNew=

```
-0,100000|
| 0,000000|
| 0,000000|
```

B=

```
+0,004000|
| +0,004000|
| +0,004000|
```

initialLambda=

```
0,000000|
| 0,000000|
| 0,000000|
```

minLambda=

```
0,000000|
| 0,000000|
| 0,000000|
```

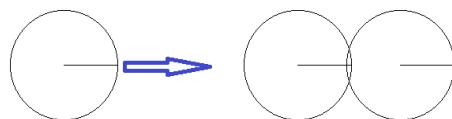
maxLambda=

```
Float.Max0000|
| Float.Max0000|
```

Das rot umrandete Gleichungssystem wurde hier mit PGS gelöst um damit zuerst Lambda zu errechnen und mit diesen Lambda wurde dann vNew bestimmt.

Kugel gegen zwei Kugeln

Wenn ich alle Kontaktpunkte in eine Matrix stecke und dann die resultierende Geschwindigkeit errechne, dann habe ich bei diesen Beispiel ein Problem, wo die linke Kugel gegen die mittlere Kugel stößt:



Siehe: PhysicEngine.UnitTests.NoGravityTests.cs/CircleAgainstCircleRow_...()

Beim Kontakt zwischen der linken und mittleren Kugel ist die Relativgeschwindigkeit ungleich Null was dazu führt, dass die Matrix hier die Geschwindigkeiten reflektiert was so richtig und gewünscht ist. Beim zweiten Kontakt zwischen der mittleren und der rechten Kugel ist die Kontaktgeschwindigkeit zum Zeitpunkt der Kugel-1-2-Kollision Null und die Matrix schlussfolgert daraus, dass die Geschwindigkeit Null bleiben muss, da ja nicht Energie aus dem Nichts entstehen darf. Wenn man das dann simuliert, wird man feststellen, dass die linke Kugel gegen die mittlere Kugel fliegt und dann werden die beiden anderen Kugeln zusammenkleben und dann zusammen nach rechts weg fliegen. Der Impuls wird also nicht korrekt von der linken zuerst auf die mittlere und danach dann auf die rechte Kugel übertragen sondern die Impulsenergie teilt sich Hälfte Hälfte auf die beiden rechten Kugeln auf.

Um dieses Problem zu lösen, muss ich erst den linken Kontaktpunkt anwenden, so dass Kugel Links seine Impulsenergie komplett an die mittlere Kugel überträgt. Danach wird dann ein Gleichungssystem nur für den rechten Kontaktpunkt aufgestellt und nur die mittlere und rechte Kugel werden betrachtet und dessen Geschwindigkeiten korrigiert.

Ich erstelle also für jeden Kontaktspunkt ein Gleichungssystem wo immer nur 2 Körper betrachtet werden.

```

6  //Wie in Part1 wird hier jeder Kontaktspunkt einzeln gelöst
7  1 Verweis
8  □ internal class MatrixImpulseResolverIterativ : IImpulseResolver
9  {
10    2 Verweise
11    □ public void Resolve(List<IRigidBody> bodies, RigidbodyCollision[] collisions, float dt, SolverSettings settings)
12    {
13      if (collisions.Length == 0) return;
14
15      foreach (var col in collisions)
16      {
17        var subBodies = new List<IRigidBody> { col.B1, col.B2 };
18        var solve = new EquationOfMotionData(subBodies, new RigidbodyCollision[] { col }, dt, settings);
19        solve.SetVelocityValues(subBodies);
20
21        string testOutput = solve.GetOutput();
22      }
23    }
24  }

```

Siehe: PhysicEngine/CollisionResolution/MatrixImpulseResolverIterativ.cs

Achtung: Wenn man nur für eine Untergruppe der Constraints eine Lösung finden will, dann darf man auch nur eine Untergruppe der Körper betrachten. Deswegen ist auf Zeile 17 die subBodies-Variable nötig, so dass das ganze Problem dann auf eine Untergruppe von Objekten reduziert wird, auf die dann eine externe Kraft und Constraint-Kräfte wirkt, welche diese Objekte dann so bewegt.

Mit diesen Ansatz kann man dann den Kugel-Stoß-Testfall nun lösen.

Würfel gegen zwei Würfel

Allerdings hilft sowohl der globale als auch der iterative Ansatz nicht beim Würfel-Stoß Problem:



Siehe: PhysicEngine.UnitTesting/NoGravityTests.cs/CubeAgainstCubeRow_...()

Der globale Ansatz würde die rechten beiden Würfel zusammen kleben lassen und der iterative Ansatz würde die Würfel drehen lassen. Hier braucht man eine Kombination aus iterativ und global.

Ich gruppiere nun Kollisionspunkte, wenn beide Körper gleich sind. D.h. In der einen Gruppe sind die beiden Kollisionspunkte zwischen Würfel 1 und 2. In der anderen Gruppe zwischen Würfel 2 und 3.

Damit sieht unser Solver nun so aus:

```

6  //Gruppieren Collisionspunkte, wo B1 und B2 auf das gleiche Objekt zeigen (Würfel ohne Drehung gegen Wand/Tisch)
7  2 Verweise
8  □ internal class MatrixImpulseResolverGrouped : IImpulseResolver
9  {
10    2 Verweise
11    □ public void Resolve(List<IRigidBody> bodies, RigidbodyCollision[] collisions, float dt, SolverSettings settings)
12    {
13      if (collisions.Length == 0) return;
14
15      var collisionGroups = collisions.GroupBy(x => bodies.IndexOf(x.B1) + "_" + bodies.IndexOf(x.B2));
16
17      foreach (var group in collisionGroups)
18      {
19        var subBodies = group.SelectMany(x => new List<IRigidBody> { x.B1, x.B2 }).Distinct().ToList();
20
21        var solve = new EquationOfMotionData(subBodies, group.ToArray(), dt, settings.Gravity, settings.IterationCount);
22        solve.SetVelocityValues(subBodies);
23
24        string testOutput = solve.GetOutput();
25      }
26    }
27  }

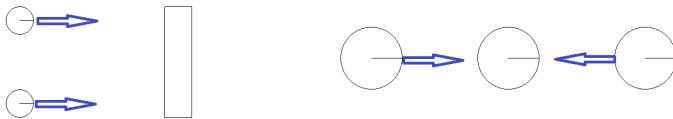
```

Siehe: PhysicEngine/CollisionResolution/MatrixImpulseResolverGrouped.cs

Hiermit geht nun der Kugel- als auch Würfel-Reihentestfall.

Gruppieren ist nicht immer die Lösung

Die Gruppierung klappt leider nicht bei diesen beiden Testfällen:



Siehe: PhysicEngine.UnitTests/NoGravityTests.cs/TwoCirclesAgainstCube_...()

Siehe: PhysicEngine.UnitTests/NoGravityTests.cs/TwoCirclesAgainstMiddleCircle_...()

Beim linken Fall gibt es nichts zu gruppieren. Er würde hier beide Kontaktpunkte iterativ lösen. Leider führt das dazu, dass der Stab dann mit Drehung weggeschwungen wird. Man braucht hier den globalen Ansatz damit er ohne Drehung weg fliegt. Beim rechten Testfall führt gruppieren wieder nur dazu, dass es sich wie der iterative Solver verhält. Die mittlere Kugel bleibt so nicht ruhig. Auch hier muss man beide Kollisionspunkte innerhalb von einem Gleichungssystem lösen.

Das Kontaktgruppierungsproblem

Wir haben jetzt hier 3 Möglichkeiten betrachtet, wie man Kontaktgruppen gruppieren kann: Iterativ, Global, Gruppe laut „B1+_+B2“. Kontaktgruppen bedeutet dass sie zusammen in ein Gleichungssystem gelöst werden. Es wird dann über alle Gruppen iteriert und jede einzeln gelöst. Iterativ war in diesen Beispielen gegenüber der „B1_B2“-Gruppe nicht besser und man kann sich das sparen. Je nach Test braucht man also entweder den Group-Solver oder den Global-Solver.

Hier in Teil 2 habe ich folgende Funktionen in der PhysicScene-Klasse welche von den Impulse-UnitTests dann benutzt werden:

```
12  public class PhysicScene
13  {
14      //Diese Funktionen sind übergangsweise drin bis ich automatisch Kontaktgruppen gruppieren kann
15      3 Verweise
16      public void UseGlobalSolver() => this.impulseResolver = new MatrixImpulseResolverGlobal();
17      2 Verweise
18      public void UseGroupSolver() => this.impulseResolver = new MatrixImpulseResolverGrouped();
19
20      public static NoGravityTestHelper DoTest(string bodysFilePath, float timeStepTickRate, int maxTriesToFindAnCollis
21  {
22          NoGravityTestHelper result = new NoGravityTestHelper();
23
24          var bodys = ExportHelper.ReadFromFile(bodysFilePath);
25
26          result.BodysBeforeCollision = new Body[bodys.Count];
27          result.BodysAfterCollision = new Body[bodys.Count];
28
29          var scene = new PhysicScene(bodys);
30          if (useGlobalSolver) scene.UseGlobalSolver(); else scene.UseGroupSolver();
31
32          return result;
33      }
34
35 }
```

Siehe: PhysicEngine/PhysicScene.cs

Siehe: PhysicEngine.UnitTests/CollisionResolution/NoGravityTestHelper.cs

Wir überlassen jetzt nicht mehr die Physikengine die Entscheidung, welchen Solver sie verwenden soll sondern der Nutzer der Engine muss das durch Aufruf von UseGlobalSolver/UseGroupSolver von außen vorgeben. Damit können wir die automatische Solver-Auswahl erst mal nach hinten schieben und uns auf die Weiterentwicklung des Matrix-Solvers konzentrieren.

Noch eine Anmerkung zum Testfall, wo eine Kugel gegen eine lange Kugelreihe fliegt: Dort habe ich den UnitTest noch so angepasst, dass nach Feststellung der ersten Kollision die TimeStep-Funktion noch 7 mal aufgerufen wird, um somit unter Nutzung des iterativen Ansatzes die Impulsenergie durch die ganze Reihe zu übertragen. Diese Impuls-Reihenschleife war in Teil 1 noch Bestandteil von der Physikengine aber hier habe ich es in den UnitTest genommen da ich schon denke, dass ein Impuls nicht sofort über die ganze Kugelreihe geleitet werden kann sondern ich denke die Ausbreitung der Impulsenergie benötigt genau so viel Zeit, wie auch die Kugel Zeit bräuchte, um die Strecke der Kugelreihe zu durchlaufen. Wenn man sich hier im Video Sekunde 30 ansieht, dann sieht man auch dass die Impulsenergie nicht schneller als die Kugel ist:

<https://www.youtube.com/watch?v=YB7KUGdP7wI>

Erweiterung um Schwerkraft

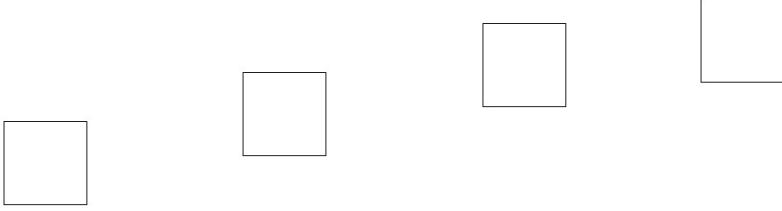
Bei den ganzen Testfällen zuvor gab es keine Schwerkraft und ein Objekt hat sich mit einer Anfangsgeschwindigkeit gegen ein anderes Objekt bewegt. Die PhysicScene-Klasse wird nun an folgenden gelb markierten Stellen erweitert:

```
70  public void TimeStep(float dt)
71  {
72      //1. Weise der externen Kraft einen Wert zu (Der Nutzer darf vor jedem TimeStep-Aufruf auch selber Kraftwerte setzen)
73      if (this.HasGravity)
74          AddGravityForceForAllBodies();
75
76      //2. Ermittle alle Kollisionspunkte
77      var collisionsFromThisTimeStep = CollisionHelper.GetAllCollisions(this.bodies);
78      if (collisionsFromThisTimeStep.Any())
79          this.CollisionOccurred?.Invoke(this, collisionsFromThisTimeStep);
80
81      //3. Constraint-Kraft aufgrund der aktuellen Geschwindigkeit berechnen und zusammen mit der externen Kraft anwenden
82      if (collisionsFromThisTimeStep.Any()) //Abfrage: Gibt es Constraints?
83          this.impulseResolver.Resolve(this.bodies, collisionsFromThisTimeStep, dt, this.settings); //Wende Constraint-Kraft + Externe Kraft an
84      else
85          ApplyExternalForces(dt); //Körper ohne Beschränkung bewegen (Wende nur die externe Kraft an)
86
87      //4. Geschwindigkeit verändert die Position
88      MoveBodies(dt);
89  }
90
91  //Der Nutzer kann pro TimeStep von außen dem Body eine externe Kraft zuweisen. Die Gravity kommt dann hier noch mit als externe Kraft hinzu.
92  private void AddGravityForceForAllBodies()
93  {
94      foreach (var body in this.bodies)
95      {
96          if (body.InverseMass == 0)
97              continue;
98
99          body.Force += new Vector2D(0, this.Gravity) / body.InverseMass;
100     }
101 }
102
103 private void ApplyExternalForces(float dt)
104 {
105     foreach (var body in this.bodies)
106     {
107         if (body.InverseMass == 0)
108             continue;
109         body.Velocity.X += body.InverseMass * body.Force.X * dt;
110         body.Velocity.Y += body.InverseMass * body.Force.Y * dt;
111         body.AngularVelocity += body.InverseInertia * body.Torque * dt;
112     }
113 }
```

Sollte das Schwerkraft-Flag aktiviert sein, dann wird auf Zeile 74 allen Körpern eine Schwerkraft zugewiesen, indem bei Zeil 99 der Force-Property ein Wert zugewiesen wird. Bei MoveBodys wird Force/Torque am Ende der Timestep-Funktion immer wieder auf Null gesetzt. Gibt es keine Kollisionen (Prüfung bei Zeile 82), dann heißt dass die Körper können sich ohne Einschränkung bewegen und es wird bei Zeile 85 die ApplyExternalForces-Funktion aufgerufen, welche die Velocity-Werte laut der Formel aus *Teil 1 - Kraft auf Punkt bei Körper – Translationskraft und Rotationskraft* aktualisiert. Sollte es aber Kollisionspunkte geben, dann taucht die Schwerkraft beim Matrix-Solver im fExt-Term auf so dass er dann sowohl die Schwerkraft als auch Constraint-Kräfte wirken lässt. Es ist wichtig, dass man die Schwerkraft pro TimeStep nicht doppelt anwendet. Entweder der Matrix-Solver wendet sie auf Zeile 83 an oder Zeile 85 macht das. Bei doppelter Anwendung würde es die Körper falsch und mit zu hoher Schwerkraft bewegen.

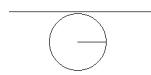
Testfälle wo Gegenstände auf ein Tisch fallen

Um zu prüfen, ob die Schwerkraft funktioniert lasse ich 4 Würfel mit unterschiedlicher Restitutioin aus gleicher Höhe auf ein Tisch fallen und sehe, dass sie ohne Drehung mehrmals hoch und runter springen und dann nach mehreren Sprüngen zur Ruhe kommen.



Siehe: PhysicEngine.UnitTests/CollisionResolution/JumpingTests.cs → JumpgingCubes_()

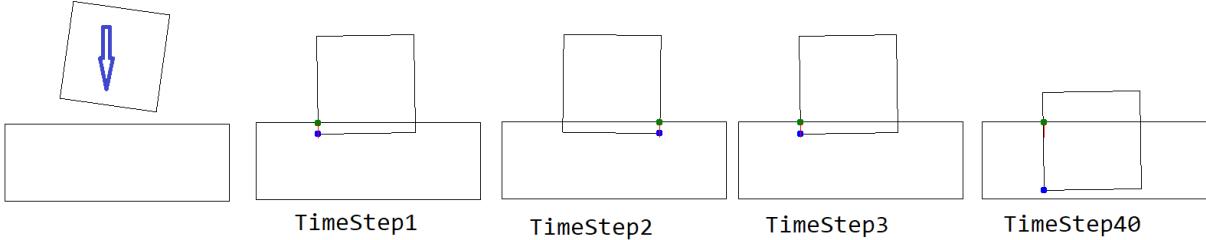
Als nächstes nehme ich ein Ball mit Restitution von 1 und ich sehe, dass auch er unendlich lange springt und immer auf genau die gleiche Ausgangshöhe kommt:



Siehe: PhysicEngine.UnitTests/CollisionResolution/JumpingTests.cs → JumpgingBall_()

Korrektur der RectangleRectangle-Collision

Wenn man ein Würfel auf ein Tisch fallen lässt, der leicht gedreht ist, dann würde ich erwarten, dass er nach dem Aufschlag noch ein paar mal herum wackelt aber dann zur Ruhe kommt.



In Wirklichkeit sieht man pro TimeStep, dass entweder an der unteren linken Ecke ein Kollisionspunkt zu sehen ist und im nächsten Time an der rechten unteren Ecke. Der Würfel hat pro TimeStep immer nur einen Kollisionspunkt mit dem Tisch und er fällt pro TimeStep immer weiter in den Tisch hinein, bis er dann hindurch fällt. Der Grund, warum es immer nur ein Kollisionspunkt zwischen dem Würfel und Tisch gibt ist, weil unsere Kollisionsroutine immer nur den Eckpunkt von Rechteck 1 zurück gibt, welches am meisten in Rechteck 2 eingedrungen ist. Ich verändere deswegen die RectangleRectangleCollision-Klasse wie folgt:

Links: So war es vorher. Es wurde immer nur ein Eckpunkt zurück gegeben

Rechts: Gebe alle Eckpunkte zurück, welche in eine Seite eingedrungen sind.

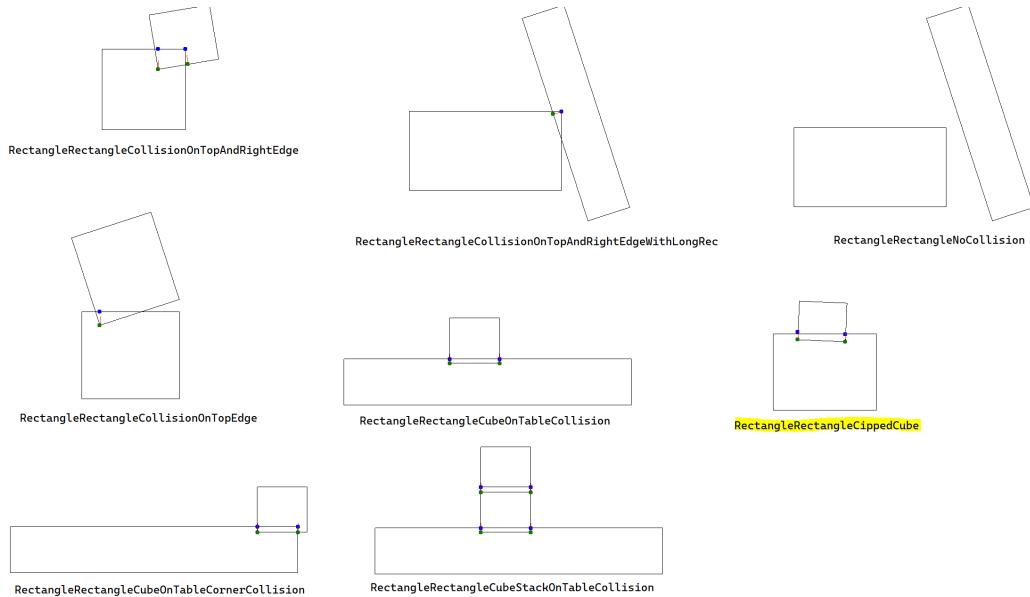
```

21 //Gib all die Punkte aus Rechteck r zurück, welche hinter der Kante p1OnEdge-p2OnEdge liegen
22 private static SupportStruct FindSupportPoint(ICollidableRectangle r, Vector2D dir, Vector2D p1OnEdge, Vector2D p2OnEdge)
23 {
24     var tmpSupport = new SupportStruct(-99999999);
25
26     //check each vector of other object
27     for (int i = 0; i < r.Vertex.Length; i++)
28     {
29         //the longest project length
30         Vector2D p1ToR1 = r.Vertex[i] - p1OnEdge;
31         float normalCheck = p1ToR1 * dir;
32
33         if (normalCheck < 0) continue; //Prüfe dass es in der "dir"-Seite eingedrungen ist
34
35         Vector2D p1ToP2 = p2OnEdge - p1OnEdge;
36         if (p1ToR1 + p1ToP2 < 0) continue; //Prüfe dass es vor der linken Kante von dir liegt
37         Vector2D p2ToR1 = r.Vertex[i] - p2OnEdge;
38         if (p2ToR1 + p1ToP2 < 0) continue; //Prüfe dass es vor der echten Kante von dir liegt
39
40         //find the longest distance with certain edge
41         //dir is "n" direction, so the distance should be positive
42         if (normalCheck > 0 && normalCheck > tmpSupport.SupportPointDist)
43         {
44             if (normalCheck > tmpSupport.SupportPointDist) tmpSupport.SupportPoints.Clear(); //Wenn zwei Punkte den gleich
45             tmpSupport.SupportPointDist = normalCheck;
46         }
47     }
48
49     return tmpSupport;
50 }
```

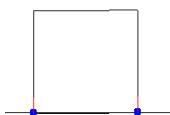
```

21 //Gib all die Punkte aus Rechteck r zurück, welche hinter der Kante p1OnEdge-p2OnEdge liegen
22 private static SupportStruct FindSupportPoint(ICollidableRectangle r, Vector2D dir, Vector2D p1OnEdge, Vector2D p2OnEdge)
23 {
24     var tmpSupport = new SupportStruct(-99999999);
25
26     //check each vector of other object
27     for (byte i = 0; i < r.Vertex.Length; i++)
28     {
29         //the longest project length
30         Vector2D p1ToR1 = r.Vertex[i] - p1OnEdge;
31         float normalCheck = p1ToR1 * dir;
32
33         if (normalCheck < 0) continue; //Prüfe dass es in der "dir"-Seite eingedrungen ist
34
35         Vector2D p1ToP2 = p2OnEdge - p1OnEdge;
36         if (p1ToR1 + p1ToP2 < 0) continue; //Prüfe dass es vor der linken Kante von dir liegt
37         Vector2D p2ToR1 = r.Vertex[i] - p2OnEdge;
38         if (p2ToR1 + p1ToP2 < 0) continue; //Prüfe dass es vor der echten Kante von dir liegt
39
40         if (normalCheck > tmpSupport.MaxDistance) tmpSupport.MaxDistance = normalCheck;
41         tmpSupport.SupportPoints.Add(r.Vertex[i]);
42     }
43
44     return tmpSupport;
45 }
```

Damit ist jetzt noch ein Kollisions-Testfall hinzu gekommen:



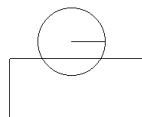
Wenn es immer nur ein Kollisionspunkt gibt, dann wirkt pro TimeStep immer nur links oder rechts ein Korrekturimpuls und der Würfel kommt nie zur Ruhe. Mit zwei Kollisionspunkten gibt es nun gleichzeitig zwei NormalConstraint-Impulse, welche an beiden Ecken den Würfel nach oben drücken und damit liegt der Würfel nun ruhig.



Erweiterung um PositionCorrection

Ball mit Restituion von 0

Ich lasse nun ein Ball mit einer Restitution von 0 auf den Tisch fallen und stelle fest, dass er ohne vom Tisch abzuprallen sofort liegen bleibt er er sinkt dabei je nach Fallhöhe sehr weit in den Tisch ein. Siehe Bild: So bleibt der Ball nach den Aufschlag dann liegen:



Wir haben doch eine NormalConstraint, welche verhindern sollte, dass zwei Körper kollidieren. Warum also sinkt der Ball so weit in den Tisch ein? Antwort: In der TimeStep-Methode wird der Ball mit jeden Schritt um eine gewisse Distanz bewegt. Erst wenn der Ball im Boden ist und die Kollisionsabfrage feststellt, dass es eine Kollision gab wird beim Matrix-Solver ein NormalConstraint-Objekt erstellt, was wegen der Restitution von 0 dann eine Constraint-Kraft errechnet, welche den Ball genau auf eine Geschwindigkeit von 0 bringt. Bei jeden weiteren TimeStep-Aufruf sorgt die NormalConstraint dann dafür, dass die Ballgeschwindigkeit trotz Schwerkraft trotzdem weiter bei 0 bleibt.

Wenn die NormalConstraint den Biaswert nach der Formel

$$\text{Bias} = -\text{restituion} * \text{velocityInNormal}$$

berechnet, dann wird lediglich die Geschwindigkeit vom Ball aber nicht dessen Position verändert.

InvDt*c.Depth

Es ist aber möglich, dass man den Impuls, welcher die Geschwindigkeit korrigiert auch dazu nutzt, um auch die Position der Körper zu verändern. Somit setzt sich der Bias-Wert dann aus dem restitutionBias, welcher die Geschwindigkeit reflektiert und dem positionBias, welcher die Position korrigiert zusammen:

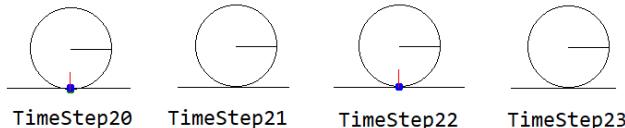
```
76     float restitutionBias = -restitution * velocityInNormal;  
77     float positionBias = invDt * c.Depth;  
78  
79     b[y] = restitutionBias + positionBias;
```

Wenn z.B. der ruhende Ball um 10 Pixel in den Tisch eingedrungen ist, dann wird bei c.Depth der Wert 10 stehen. Wenn wir nun wollen, dass am Ende vom TimeStep der Ball wieder auf den Tisch liegt, dann muss die Position um 10 Pixel verschoben werden. Die Positionformel ist: Position += Velocity*dt (siehe MoveBodys aus der PhysiScene-Klasse). Velocity*dt muss also 10 ergeben. Das tut es nur, wenn Velocity den Wert 10/dt hat. Der Bias-Wert von der Normalconstraint sagt ja, welche Relativgeschwindigkeit die Kontaktpunkte haben sollen. Der Tisch bewegt sich nicht, also bedeutet das, mit dem Bias-Wert gebe ich vor, welche Velocity der Ball haben soll. Wenn der Ball bereits ruht, dann ist der restitutionBias 0 und unser positionBias muss dann den Wert 10/dt haben. InvDt = 1/dt.

*positionBias=invDt*c.Depth bedeutet also wir wollen den Ball um c.Depth Pixel in Richtung Kontaktnormalen verschieben.*

AllowedPenetration

Nach der Erweiterung des NormalConstraint-Bias-Wertes um den invDt*c.Depth-Wert probieren wir erneut, ob der Ball mit Restitution von 0 nun korrekt auf der Tischoberfläche zur Ruhe kommt. Wir sehen, dass er nach den Aufschlag noch mehrmals etwas springt und dann an der Tischoberfläche kurz zur Ruhe kommt. Dort bleibt er dann aber nicht ruhig liegen sondern er macht lauter kleine Minisprünge und der Kontaktpunkte geht immer wieder verloren.

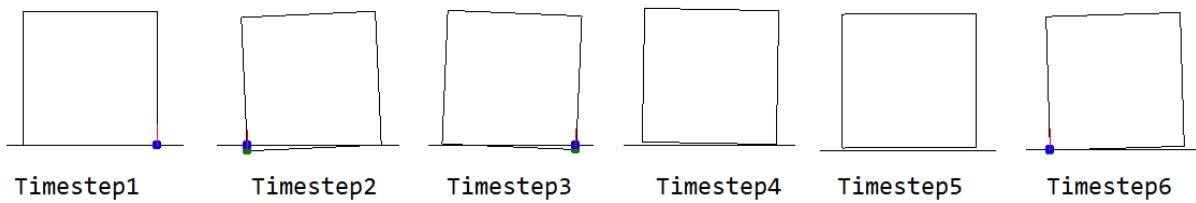


Der Grund dafür ist, weil wir den Ball ja exakt an die Tischoberfläche mit den PositionBias drücken. Dort gibt es dann keine Kollision mehr und die Schwerkraft lässt den Ball für ein TimeStep ungebremst nach unten fallen. Dann wird ein neuer Kontaktpunkt erzeugt, welcher den Ball dann auf die Geschwindigkeit von Null bringt und ihn zurück zur Tischoberfläche verschiebt. So geht das dann unendlich lange weiter und der Ball kommt nie zur Ruhe. Um das zu vermeiden dürfen wir den Ball nicht komplett bis zur Tischkante hoch drücken sondern nur bis kurz davor. Wir erweitern den PositionBias-Term deswegen nun um den AllowedPenetration-Wert:

```
76     float restitutionBias = -restitution * velocityInNormal;  
77     float positionBias = invDt * Math.Max(0, c.Depth - s.AllowedPenetration);  
78  
79     b[y] = restitutionBias + positionBias;
```

AllowedPenetration hat den Wert 1. Ist der Ball um 10 Pixel in den Tisch eingedrungen, so wird er um 9 Pixel nach oben verschoben. Auf diese Weise bleibt nach Anwendung der Positionkorrektur der Kontaktpunkt erhalten und der Ball bleibt ruhig an der Tischoberfläche liegen.

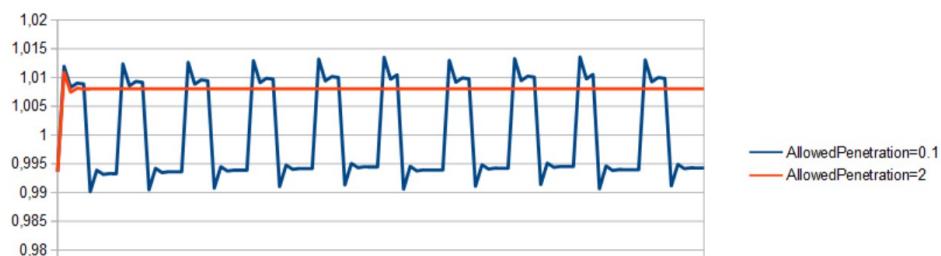
Es gibt beim AllowedPenetration-Faktor noch eine kleine Sache zu beachten. Der Wert darf nicht zu klein sein. Gegeben ist ein Würfel, der auf ein Tisch liegt, und der eine minimale Anfangsdrehgeschwindigkeit hat. Meine Erwartung ist, dass der Würfel nach kurzer Zeit zur Ruhe kommt. Wenn der AllowedPenetration-Wert aber zu klein ist (0.1) dann kommt der Würfel nicht zur Ruhe. Er springt mit lauter kleinen Minisprüngen auf die linke oder rechte Ecke.



Timestep1 Timestep2 Timestep3 Timestep4 Timestep5 Timestep6

Ich muss AllowedPenetration auf 2 stellen damit der Würfel sich beruhigt. Stellt man den Wert zu klein ein, dann wird immer jeweils eine Würfecke vom Tisch heraus gedrückt was den Kontaktspunkt dann löst und was somit dann den Kontaktspunkt löscht.

Beim CubeOnTable_StartsWithMomentum_BecomesQuickResting-Unitest sieht man den Unterschied zwischen AllowedPenetration mal mit ein Wert von 0.1 und mal mit 2. Man sieht hier den Angle-Wert vom Würfel über der Zeit:



Siehe: PhysicEngine.UnitTests/CollisionResolution/RestingContactTests.cs → CubeOnTable_StartsWithMomentum

Bei der roten Linie gibt es am Anfang kurz ein Wackeln und dann läuft sie ohne Bewegung weiter. Die blaue Linie wackelt ständig herum, weil der AllowedPenetration-Wert zu klein ist und es somit immer wieder einen der beiden Kontaktspunkte wegen der Würfeldrehung aus dem Tisch drückt.

PositionalCorrectionRate

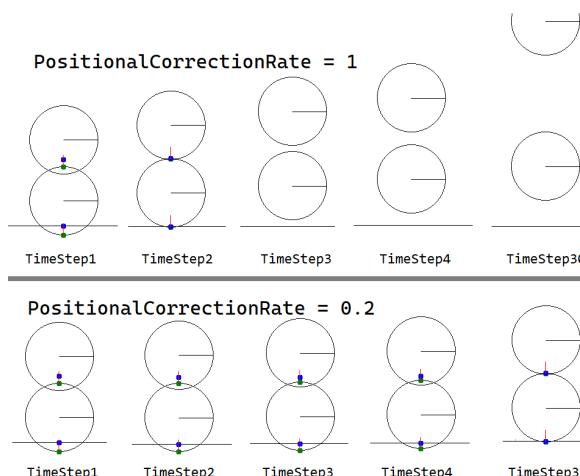
Obwohl wir wegen des Einsatzes von AllowedPenetration den Ball nur bis 1 Pixel unter die Tischoberkante drücken kommt dann Ball nach dem Aufschlag nicht sofort zur Ruhe sondern er springt trotz einer Restitution von 0 trotzdem noch ein paar Mal nach oben. Der Grund dafür ist, weil man den Position-Fehler nicht sofort innerhalb von ein TimeStep-Aufruf korrigieren darf sondern man darf nur ungefähr 20% des Fehlers korrigieren. Wir erweitern den PositionBias um die PositionalCorrectionRate (Baumgarte-Term):

```

76     float restitutionBias = -restitution * velocityInNormal;
77     float positionBias = s.PositionalCorrectionRate * invDt * Math.Max(0, c.Depth - s.AllowablePenetration);
78
79     b[y] = restitutionBias + positionBias;

```

Das ist eine Prozentzahl, die angibt, wie viel Prozent der Überlappung pro Zeitschritt korrigiert werden soll. Ein weiteres Beispiel, wo es wichtig ist, dass die PositionalCorrectionRate verwendet wird ist, wenn ich zwei Kugeln übereinander habe, die initial sich überlappen:



Würde man mit einer PositionalCorrectionRate von 1 die Simulation laufen lassen, dann würde die obere Kugel nach oben weg geschleudert werden, da der Position-Korrekturimpuls viel zu viel Energie in das System bringen würde. Bei einer PositionalCorrectionRate von 0.2 drückt es die Kugeln langsam über mehrere TimeSteps auseinander.

Durch die Erweiterung des NormalConstraint-Bias-Wertes um die PositionCorrection ist es nun möglich, dass ein Ball, der eine Restitution von 0 hat auf ein Tisch fallen kann ohne dass er dabei nach oben springt und er kommt an der Tischoberkante zum stehen. Auch der Kugelstapel kann damit simuliert werden.

Ball mit Restitution von 1

DoPositionalCorrection-Schalter

Wenn ich nun erneut den Testfall laufen lasse, wo ein Ball mit einer Restitution von 1 auf ein Tisch fällt: siehe PhysicEngine.UnitTests/CollisionResolution/JumpingTests.cs → JumpgingBall_()

und diesmal verwende ich aber PositionCorrection, dann stelle ich fest, dass der Ball mit jedem Sprungzyklus immer höher kommt. Der Grund dafür ist, weil der PositionCorrection-Impuls den Ball zusätzliche Energie gibt so dass er dann mehr Geschwindigkeit nach oben bekommt, als wenn die Geschwindigkeit am Tisch einfach nur reflektiert wird. Aus dem Grund wird die PositionCorrection jetzt noch um den DoPositionalCorrection-Schalter erweitert.

```
71 float restitutionBias = -restitution * velocityInNormal;
72
73 float biasFactor = s.DoPositionalCorrection ? s.PositionalCorrectionRate : 0.0f;
74 float positionBias = biasFactor * invDt * Math.Max(0, c.Depth - s.AllowedPenetration);
75
76 b[y] = restitutionBias + positionBias;
```

Siehe: PhysicEngine/CollisionResolution/EnterTheMatrix/Constraints/NormalConstraint.cs

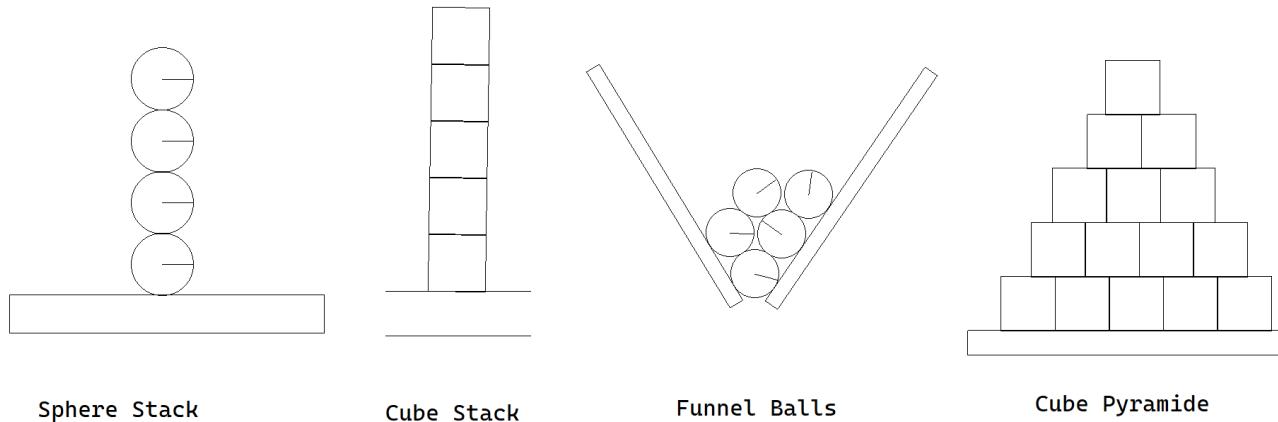
Diesen Schalter stelle ich immer dann auf True, wenn ich ein Ball mit einer Restitution von 1 simulieren will, welcher unendlich lange springen soll ohne das er immer höher kommt. Eine andere Lösung wäre dass man niemals eine Restitution von 1 verwendet sondern z.B. nur von 0.8. Oder ich verwende eine ganz kleine TimeStep-Weite, wodurch der Ball nur ein kleines Stück in den Tisch eindringt, was einen kleinen Korrekturimpuls erzeugt. So darf die Restitution dann auch nahe 1 sein aber man hat mehr Rechenaufwand. Würde man den genauen Kollisionszeitpunkt bestimmen wäre das auch noch eine Lösung.

Zusammenfassung – Wo darf man PositionCorrection einsetzen und wo nicht

Testfälle, wo PositionCorrection zwingend nötig ist

Siehe: PhysicEngine.UnitTests/CollisionResolution/PositionCorrectionTests.cs

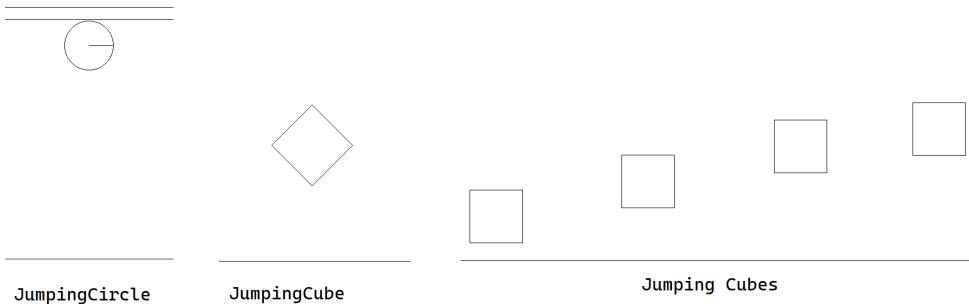
Folgende Testfälle sind Beispiele, wo die PositionCorrection zwingend nötig ist, um zu verhindern, dass die Objekte ineinander rutschen. Der Grund ist, weil die Objekte übereinander liegen und sich gegenseitig ineinander drücken. Bei der Pyramide braucht der Test 14 Sekunden, weil es viele Objekte mit vielen Kontaktpunkten sind, welche eine große Matrix erzeugen. Außerdem benötigt man zwingend 100 PGS-Iterationen, damit das System überhaupt zur Ruhe kommt.



Testfälle, wo man Position-Correction nicht einsetzen darf

Siehe: PhysicEngine.UnitTests/CollisionResolution/JumpingTests.cs

Wenn man eine Restitution von 1 oder nahe 1 hat, dann darf man keine PositionCorrection verwenden, weil das Objekt ansonsten mit jedem Sprung immer höher kommt. Die PositionCorrection gibt dem Objekt zusätzliche Höhe je nach dem, wie tief es in den Boden eingedrungen ist. Man könnte entweder versuchen den exakten Kollisionszeitpunkt zu bestimmen und somit die PositionCorrection überflüssig machen oder man verwendet für solche Superspringbälle keine Positionskorrektur.



Erweiterung um Reibung

Bis jetzt hatten wir nur Fälle betrachtet, wo wir die Bewegung von Kugeln und Würfeln unter Nutzung der Schwerkraft simulieren. Die NormalConstraint sorgt dafür, dass Kollisionen korrekt aufgelöst werden. Was jetzt aber noch fehlt ist die Reibung bei Kontaktpunkten.

Für die Reibung müssen wir wieder eine Constraint-Gleichung aufstellen. Hier beginnen wir sofort

mit der Velocity-Constraint. U1 ist die Kontaktpunktnormale gedreht um 90 Grad. Wir drehen den Vektor indem wir das Kreuzprodukt von der Normale mit Z=(0,0,1) bilden. U1 ist hier also unsere Tangente und wir schauen, mit welcher Geschwindigkeit die Kontaktpunkte in Richtung Tangente sich bewegen.

$$\dot{C}_{u1} = (v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1) \cdot u_1$$

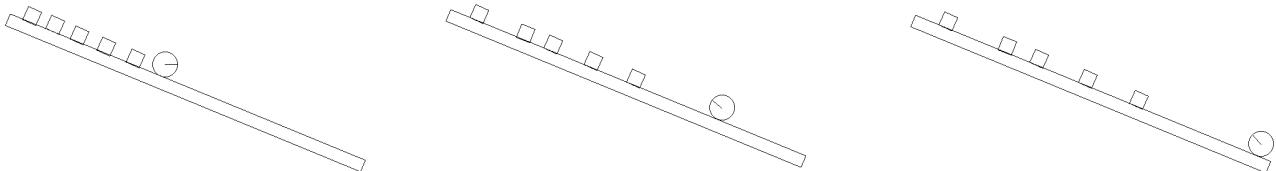
Über umstellen erhalten wir die J*V-Matrix-Zeile.

$$J_u V = \begin{pmatrix} -u_1^T & -(r_1 \times u_1)^T & u_1^T & (r_2 \times u_1)^T \end{pmatrix} \begin{pmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{pmatrix}$$

Die Reibungskraft ist so, dass wenn sich der Körper in Tangenterichtung bewegt, dann soll sie wirken. Ansonsten soll sie nicht wirken. D.h. der Ziel-Velocity-Wert ist 0. Die Reibung will den Körper komplett zum Stillstand bringen. Aus diesen Grund legen wir für den Friction-Biaswert 0 fest.

Die Reibungskraft soll um so stärker wirken um so höher die Schwerkraft ist und um so höher der Friction-Koeffizient ist. Aus dem Grund legen wir für Max-Lambda Friction*Gravity fest. Min-Lambda ist dann einfach MaxLambda negiert.

Ich teste die Reibung nun damit, indem ich unterschiedliche Friction-Koeffizienten den Objekten zuweise, so dass sie unterschiedlich schnell die Schrägen runter rutschen.



Siehe: PhysicEngine.UnitTests/CollisionResolution/RestingContactTests.cs → SlidingCubes()

Colliding-Contact vs Resting-Contact

Aus dem Dokument „Physically Based Modeling Rigid Body Simulation“ Seite 33 von David Baraff habe ich die Begriffe *Colliding Contact* und *Resting Contact*, welche er so definiert:

- **Colliding Contact** = Die Kontaktpunkte bewegen sich in Normalenrichtung aufeinander zu. Beispiel: Ball fliegt gegen Wand. Erwartung: Impuls lässt ihn abprallen.
- **Resting Contact** = Die Relativgeschwindigkeit der Kontaktpunkte in Normalenrichtung ist Null. Beispiel: Ein Würfel oder Ball liegt auf ein Tisch. Erwartung: Der Würfel/Ball bleiben ruhig liegen.

Ich habe mich bei der Benennung der Testfälle im UnitTests-Projekt an diesen Namen orientiert.

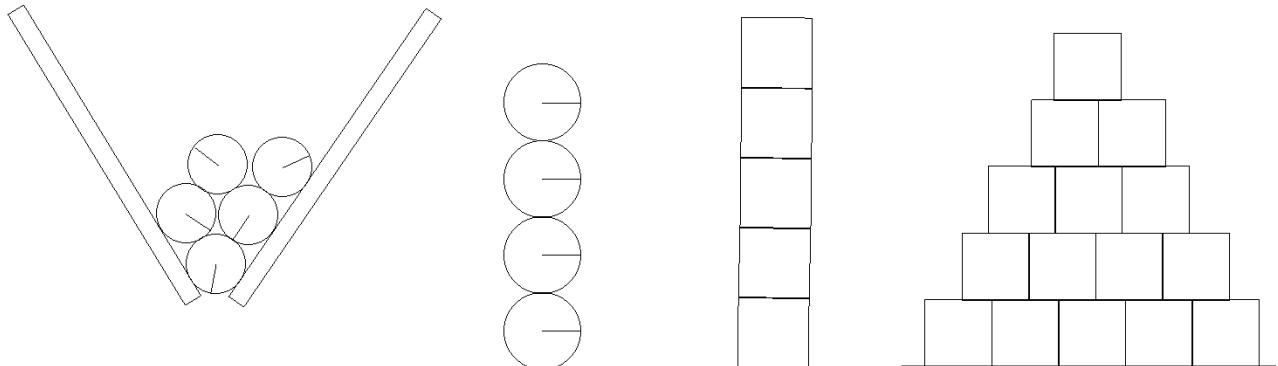
▲	✓	PhysicEngine.UnitTests.CollisionResolution (27)
▷	✓	JumpingTests (4)
▷	✓	NoGravityTests (11)
▷	✓	PGSConvergenceTest (1)
▷	✓	PositionCorrectionTests (4)
▷	✓	RestingContactTests (7)

Bei den RestingContactTests prüfe ich, das ein Würfel oder Ball ruhig auf ein Tisch liegt oder dort dann zur Ruhe kommt. Bei den JumpingTests prüfe ich, das Objekte mit einer unterschiedlichen Restitution unterschiedlich lange springen. Hier wird also ein CollidingContact in ein RestingContact umgewandelt. NoGravityTests prüfen Testfälle, wo Objekte ohne Schwerkraft und mit Anfangsdrehgeschwindigkeit zusammen prallen (CollidingContact-Testfälle). Die PositionCorrectionTests enthalten Testfälle, wo PositionCorrection zwingend nötig ist.

Die Konvergenzgeschwindigkeit von Projected Gauss-Seidel

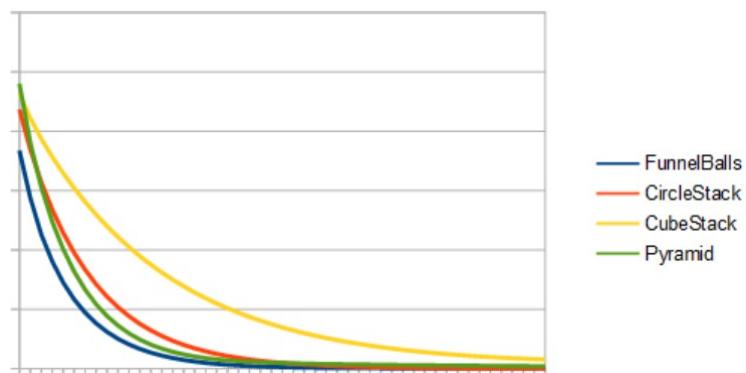
Siehe: PhysicEngine.UnitTests/CollisionResolution/PGSConvergenceTest.cs

Bei diesen Test hat man eine Szene wo alles in Ruhe liegt. Die Schwerkraft bringt nun Unruhe ins System. Die Aufgabe von PGS ist es nun die resultierende Constraint-Kraft möglichst genau zu bestimmen. Macht es da einen Fehler, dann beginnt das System zu wackeln. Der Würfelturm könnte dann z.B. mit Wackeln anfangen oder sogar umkippen.



Ich habe nun zuerst den „perfekten“ Lambda-Wert berechnet, indem ich 1000 PGS-Iterationen laufen lassen habe. Dann habe ich jeden Lambda-Vektor, welcher als Zwischenlösung bei PGS ermittelt wurde gegen den „perfekten“ Lambda-Vektor verglichen, indem ich den Quadrat-Abstand dieser mehrdimensionalen Punkte genommen habe.

Die PGS-Fehlerkurve über 50 Schritte sieht so aus:



Außerdem habe ich noch ermittelt, wie viele TimeSteps nötig waren bis die Körper zur Ruhe kommen:

Test	Matrixgröße von J [BodyCount*3*ConstraintCount]	Timesteps um in Ruhelage zu kommen	Zeit die der UnitTest gebraucht hat
FunnelBalls	$[21*20]=420$	131	358 ms
CircleStack	$[15*8]=120$	27	26 ms
CubeStack	$[18*20]=360$	49	194 ms
Pyramid	$[48*136]=6528$	171	13,3 s

Aus der PGS-Fehlerkurve erkennt man, dass wenn man ein Turm simuliert, dann braucht das mehr PGS-Iterationen, da Fehler am Fuße des Turmes sich auf den oberen Turm auswirken. Das ist eine wenig fehlertolerante Konstruktion. In der Tabelle Spalte 3 sieht man, das viele Kontaktpunkte viele TimeSteps benötigen, bis da Ruhe rein kommt. Tabelle Spalte 4: Die Rechenzeit hängt mit der Größe der Matrix J zusammen. Sie ergibt sich aus:

Körperanzahl*Kontaktpunktanzahl*ConstraintsProKontaktpunkt

Zusammenfassung von Teil 2

7 Schritte, um ein Körper unter Einhaltung der Constraint-Regeln zu bewegen:

$$\text{Gegeben: } M\dot{V} = F_c + F_{ext} \quad \dot{V} \approx \frac{V^2 - V^1}{\Delta t} \quad \text{Gesucht: } F_c \quad V^2$$

1: Constraint-Gleichung aufstellen $f(V2) = \zeta$

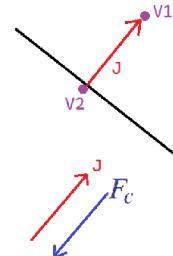
$$J \cdot V2 = \zeta$$

$$\begin{pmatrix} V_{x1} \\ V_{y1} \\ \omega_1 \\ V_{x2} \\ V_{y2} \\ \omega_2 \end{pmatrix}$$

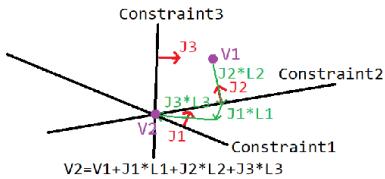
2: Lineare Abbildung f als Matrix schreiben

$$\begin{pmatrix} -n^T & -(r_1 \times n)^T & n^T & (r_2 \times n)^T \end{pmatrix} \zeta$$

3: Da $J \cdot \zeta = 0$ eine Ebenengleichung ist, kann gesuchter Punkt $V2$ über $V1 + J \cdot \lambda$ ermittelt werden



4: $V1$ in eine Richtung zu verschieben heißt, dass auf $V1$ ein Impuls/Eine Kraft in Richtung J wirkt. Diese Kraft ist die Constraint-Kraft F_c welche parallel zu J ist.



5: Da aber von allen Kontaktpunkten der J -Zeilenvektor zu einer J -Matrix zusammengefasst wird, ist also ein $V2$ -Punkt gesucht, welcher auf allen Constraint-Ebenen liegt. Somit heißt das, ich suche die Summe aller J -Zeilen gewichtet nach Lambda was J -Transpose * Lambda bedeutet.

$$F_c = J^T \lambda$$

6: Ausgangsgleichung mit Hilfe von $J \cdot V2 = \zeta$ und $F_c = J^T \lambda$ nach Lambda umstellen und dieses mit PGS lösen.

$$J M^{-1} J^T \lambda = \frac{1}{\Delta t} \zeta - J \left(\frac{1}{\Delta t} V^1 + M^{-1} F_{ext} \right)$$

7: Mit Lambda $V2$ ermitteln und damit alle Körper aktualisieren

```
for (int i = 0; i < bodies.Count; i++)
{
    bodies[i].Velocity.X = V2[i * 3 + 0];
    bodies[i].Velocity.Y = V2[i * 3 + 1];
    bodies[i].AngularVelocity = V2[i * 3 + 2];
}
```

Offene Punkte / Ausblick

Alle Kollisionstestfälle gehen jetzt zwar aber wegen den Kollisionspunkt-Gruppierungsproblem muss in den Tests noch per Schalter festgelegt werden, ob der Globale- oder der Grouped-Solver verwendet wird. Außerdem braucht der Pyramiden-Test sehr lange, weil dort eine große Matrix mit vielen PGS-Iterationen berechnet werden muss. Beim Lambda-Wert starten wir immer nur mit 0. Wir nutzen nicht den Umstand, dass bei Resting-Contacts der Lambda vom vorherigen TimeStep wohl ziemlich ähnlich zum aktuellen Lambda ist.

Ziel für Part 3 ist somit:

- Warmstart (Setze initial Lambda): Erweitere dazu CollisionInfo um StartEdgeIndex/EndEdgeIndex
- Resting-Sphere / Resting-Cube → Untersuche die PGS-Konvergenzgeschwindigkeit mal mit und ohne Warmstart
- Matrizen enthalten viele Nullen → Nutze sequentielle Impulse

Quellen für Teil 2

Iterative Dynamics with Temporal Coherence - Erin Catto 2005

Box2D-Light

Rigid Body Simulation - David Baraff 2001

Teil 3: Sequentielle Impulse

Ziel dieses Abschnitts

Der Matrix-Solver vom vorherigen Abschnitt funktioniert gut aber er ist ineffektiv und langsam. Hier soll der Matrix-Solver verbessert werden und aufbauend auf der fertigen Matrix-Lösung soll hier dann das sequentielle Impulse-Verfahren erarbeitet werden.

Matrix um Warmstart erweitern

In Teil 2 wurde eine Bewegungsgleichung in Matrix-Form aufgestellt welche per PGS gelöst wurde. Dabei musste man ein initialen Lambda-Wert angegeben. Um so näher Lambda an der Ziellösung ist, um so schneller konvergiert PGS auch. Wenn man ein Resting-Contact hat, dann ändert sich ja am Lambda-Wert nichts. Bsp: Ein Würfel liegt ruhig auf dem Boden. Es wirkt die Schwerkraft auf ihn, wodurch dann ein Lambda für seine Normal- und Frictionconstraint berechnet wird. Wenn er ruhig liegt, dann ändert sich auch nicht seine Relativgeschwindigkeit von sein Kontaktunkten und somit hat auch im nächsten TimeStep Lambda wieder den gleichen Wert.

Aus dem Grund wollen wir nun die Kontaktunkte so verändern, dass sie sich die Normal- und Friction-Lambda-Werte vom letzten TimeStep merken. Damit ich weiß, welcher Kontaktunkpt in der Kollisionsabfrage neu ist und welcher schon im letzten TimeStep vorhanden war brauche ich für jeden Punkt einen eindeutigen Schlüssel.

Die RigidBodyCollision-Klasse wird um die Key-Property erweitert:

```
6  public class RigidBodyCollision : CollisionInfo
7  {
8      99+ Verweise | 6/6 bestanden
9      public IRigidBody B1 { get; private set; }
10     99+ Verweise | 4/4 bestanden
11     public IRigidBody B2 { get; private set; }
12     public readonly string Key; //Eindeutiger Schlüssel zwischen Edge/Face von Körper 1 und Edge/Face von Körper 2
13
14     1 Verweis
15     public RigidBodyCollision(CollisionInfo info, IRigidBody b1, IRigidBody b2, int body1Index, int body2Index)
16         :base(info.Start, info.End, info.Normal, info.Depth, info.StartEdgeIndex, info.EndEdgeIndex)
17     {
18         this.B1 = b1;
19         this.B2 = b2;
20         this.Key = body1Index + "_" + body2Index + "_" + info.StartEdgeIndex + "_" + info.EndEdgeIndex;
21     }
22 }
```

Der BodyIndex ist einfach nur die Position des Körpers in der Liste aller Körper:

```
8  public static RigidBodyCollision[] GetAllCollisions(List<IRigidBody> bodies)
9  {
10     List<RigidBodyCollision> collisions = new List<RigidBodyCollision>();
11
12     for (int i = 0; i < bodies.Count; i++)
13         for (int j = i+1; j < bodies.Count; j++)
14         {
15             var b1 = bodies[i];
16             var b2 = bodies[j];
17             if ((b1.InverseMass != 0 || b2.InverseMass != 0) && BoundingCircleTest.Collide(b1, b2))
18             {
19                 var contacts = b1.CollideWith(b2);
20                 if (contacts != null)
21                     collisions.AddRange(contacts.Select(x => new RigidBodyCollision(x, b1, b2, i, j)));
22             }
23         }
24
25     return collisions.ToArray();
26 }
```

Die CollisionInfo-Klasse wird um StartEdgeIndex und EndEdgeIndex erweitert:

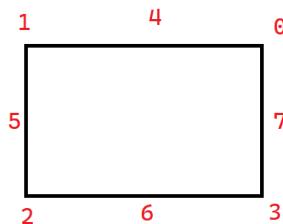
```

7  public class CollisionInfo
8  {
9      public Vector2D Start; //Collisionpoint from RigidBody1
10     public Vector2D End; //Collisionpoint from RigidBody2
11     public Vector2D Normal; //Normal from Start-Point
12     public float Depth; //Distanz between Start and End
13
14     public readonly byte StartEdgeIndex; //Beim Rechteck: Index von der Ecke oder Face-Seite, wo der Kontaktspunkt liegt
15     public readonly byte EndEdgeIndex; //Beim Kreis steht hier immer nur 0
16
17     11 Verweise
18     public CollisionInfo(Vector2D start, Vector2D normal, float depth, byte startEdgeIndex, byte endEdgeIndex)
19         : this(start, start + normal * depth, normal, depth, startEdgeIndex, endEdgeIndex)

```

Der StartEdgeIndex und EndEdgeIndex gibt an welche Kante oder Eckpunkt von ein Körper den Kollisionspunkt definiert.

Bei der RectangleRectangle-Kollision kann es passieren, dass es mehrere Kollisionspunkte zwischen den beiden Objekten gibt. Ein Kollisionspunkt liegt entweder auf der Ecke oder auf einer Kante. Liegt es auf einer Ecke, dann bekommt der Punkt den Index 0..3. Liegt es auf einer Kante, dann 4..7.



```

48     private static CollisionInfo[] FindAxisLeastPenetration(ICollidableRectangle r1, ICollidableRectangle r2)
49     {
50         float bestDistance = 999999;
51         int r1Face = -1;
52         byte[] supportPoints = null;
53
54         for (int i = 0; i < r1.FaceNormal.Length; i++) ...
55
56         //all four directions have support point. That means at least one point from r2 lies inside from r1
57         return supportPoints.Select(r2Edge => new CollisionInfo(r2.Vertex[r2Edge] + r1.FaceNormal[r1Face] * bestDistance, r1.FaceNormal[r1Face], bestDistance, (byte)(r1Face + 4), r2Edge));
58     }

```

Wenn zwei Kreise kollidieren, kann pro Kreis immer nur ein Kollisionspunkt erzeugt werden. Deswegen reicht es hier, wenn ich fest Null als Start/End-EdgeIndex angebe:

```

32     public static CollisionInfo CircleCircle(ICollidableCircle c1, ICollidableCircle c2)
33     {
34         Vector2D from1to2 = c2.Center - c1.Center;
35         float rSum = c1.Radius + c2.Radius;
36         float dist = from1to2.Length();
37
38         if (dist != 0)
39         {
40             // overlapping but not same position
41             Vector2D normal = from1to2 / dist;
42             return new CollisionInfo(c1.Center + normal * c1.Radius, normal, rSum - dist, 0, 0);
43         }
44         else
45         {
46             //same position
47             Vector2D normal = new Vector2D(1, 0);
48             return new CollisionInfo(c1.Center + normal * c1.Radius, normal, rSum, 0, 0);
49         }
50     }

```

Auch bei der RectangleCircle-Kollision kann maximal ein Kontaktspunkt erzeugt werden so dass auch hier fest Null übergeben werden darf.

```

53     public static CollisionInfo RectangleCircle(ICollidableRectangle r, ICollidableCircle c)
54     {
55         return new CollisionInfo(c.Center - normal * c.Radius, normal, c.Radius - dis, 0, 0); //Corner 1
56
57         return new CollisionInfo(c.Center - normal * c.Radius, normal, c.Radius - dis, 0, 0); //Corner 2
58
59         return new CollisionInfo(c.Center - r.FaceNormal[nearrestEdge] * c.Radius, r.FaceNormal[nearrestEdge], c.Radius - bestDistance, 0, 0); //Face
60
61         return new CollisionInfo(c.Center - r.FaceNormal[nearrestEdge] * c.Radius, r.FaceNormal[nearrestEdge], c.Radius - bestDistance, 0, 0);

```

Der CollisionPointsCache nimmt eine Liste von RigidbodyCollision-Objekten entgegen und schaut dann auf die Key-Property um zu sehen, ob es neue Kontaktpunkte gibt. Sollte ein neuer Punkt kommen, dann nimmt er den neuen Punkt und wandelt ihn mit der cToTConverter-Func in ein T um. Sollte der Punkt bereits im letzten TimeStep vorhanden sein, dann wird mit der takeDataFromOldPoint das T in ein T umgewandelt.

```

7  internal class CollisionPointsCache<T> where T : RigidbodyCollision
8  {
9      private List<T> cache = new List<T>();
10     private Func<RigidbodyCollision, T> cToTConverter; //Wandelt ein RigidbodyCollision in ein T um
11     private Action<T, T> takeDataFromOldPoint; //Nimmt vom alten T den Lambdawert und kopiert ihn zum neuen T
12
13     4 Verweise
14     public CollisionPointsCache(Func<RigidbodyCollision, T> cToTConverter, Action<T, T> takeDataFromOldPoint)
15     {
16         this.cToTConverter = cToTConverter;
17         this.takeDataFromOldPoint = takeDataFromOldPoint;
18     }
19
20     4 Verweise
21     public T[] Update(RigidbodyCollision[] collisions)

```

Damit der Kollisionspunkt die Lambda-Werte speichern kann wird folgende Klasse erstellt:

```

7  internal class CollisionPointWithLambda : RigidbodyCollision
8  {
9      public float NormalLambda = 0;
10     public float FrictionLambda = 0;
11
12     5 Verweise
13     public CollisionPointWithLambda(RigidbodyCollision c)
14         :base(c)
15     {
16     }
17
18     4 Verweise
19     public void TakeDataFromOtherPoint(CollisionPointWithLambda c)
20     {
21         this.NormalLambda = c.NormalLambda;
22         this.FrictionLambda = c.FrictionLambda;
23     }

```

Der Konstruktor mit dem RigidbodyCollision-Parameter wird bei der cToTConverter-Funktion benutzt und die TakeDataFromOtherPoint-Methode bei der takeDataFromOldPoint-Action.

Der Grund, warum der PointCache mit ein generischen T arbeitet ist, weil bei der Matrix braucht die WarmStart-Funktion Lambda-Werte (ist die Länge der Constraintkraft). Bei sequentiellen Impulsen wird Impulsenergie (entspricht Lambda*Dt) gespeichert.

Der Solver wird nun um den Cache erweitert:

```

6  //Umsetzung des Papers "Iterative Dynamics with Temporal Coherence – Erin Catto 2005"
7  //Stelle Gleichungssystem über alle Kontaktpunkte auf und Löse dafür Lambda
8  4 Verweise
internal class MatrixImpulseResolver : IImpulseResolver
{
    private CollisionPointsCache<CollisionPointWithLambda> pointCache
        = new CollisionPointsCache<CollisionPointWithLambda>((c) => new CollisionPointWithLambda(c), (oldP, newP) => newP.TakeDataFromOtherPoint(oldP));
13
14    6 Verweise
15    public void Resolve(List<IRigidBody> bodies, RigidbodyCollision[] collisions1, float dt, SolverSettings settings)
16    {
17        if (collisions1.Length == 0) return;
18        var collisions = this.pointCache.Update(collisions1);
19
20        var solve = new EquationOfMotionData(bodies, collisions, dt, settings);
21        solve.SetVelocityValues(bodies);

```

Nun können Dank der gecacheten Punkte nun endlich Lambda-Werte vom letzten TimeStep verwendet werden:

```

53    //Schritt 2: Lambda per PGS ermitteln
54    this.minLambda = constraints.GetMinLambda();
55    this.maxLambda = constraints.GetMaxLambda();
56    this.initialLambda = settings.DoWarmStart ? constraints.GetInitialLambda() : Matrix.GetColumnVectorWithZeros(this.minLambda.RowCount);
57    this.lambda = ProjectedGaussSeidel.Solve(A, B, initialLambda, minLambda, maxLambda, settings.IterationCount);
58    constraints.SaveLambdaInCollisionPoints(this.lambda);

```

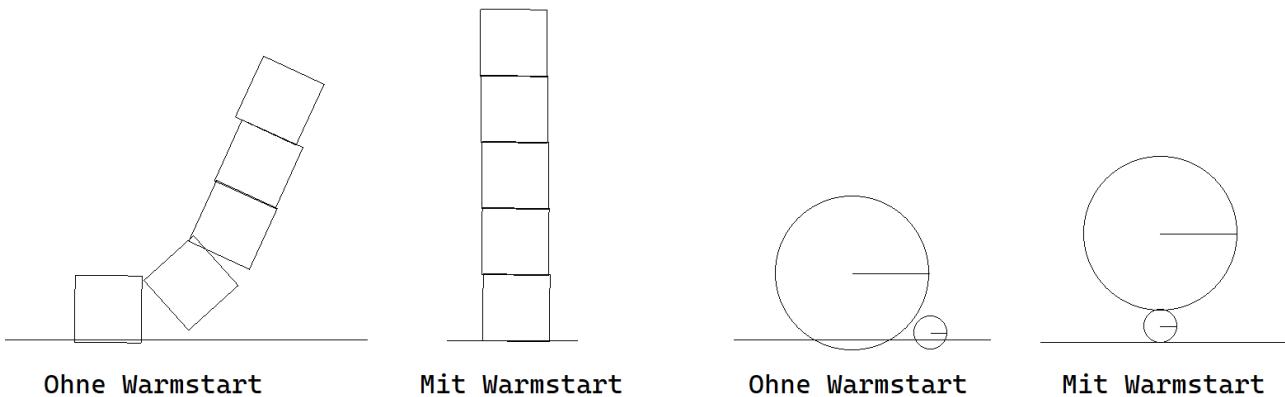
Nur die Constraint-Klasse weiß dann, worauf (Kontaktpunkt/Gelenk) sich das Lambda bezieht und

in welcher Property es dieses speichern muss. Die Normal-Constraint speichert sein Lambda in der NormalLambda-Property beim Kollisionspunkt.

```

95  public Matrix GetInitialLambda()
96  {
97      return Matrix.GetColumnVector(this.data.Collisions.Select(x => x.NormalLambda).ToArray());
98  }
99  3 Verweise
100 public int GetLambdaRowCount()
101 {
102     return this.data.Collisions.Length;
103 }
104 2 Verweise
105 public void SaveLambdaInCollisionPoints(Matrix lambda) //Speichert die Lambdawerte in CollisionPointWithLambda
106 {
107     for (int i=0;i<lambda.RowCount;i++)
108     {
109         this.data.Collisions[i].NormalLambda = lambda[0, i];
110     }
111 }
```

Dank des Warmstarts kann ich nun den Würfel- oder Kugel-Stapel nun mit nur noch 10 PGS-Iterationen simulieren. Wenn ich den Warmstart weglassen, dann fliegt der Stapel um.



Der Nutzen vom Warmstart ist also, das ich mit viel weniger PGS-Schritten Resting-Contact-Fälle simulieren kann.

Eine J-Zeile als Objekt

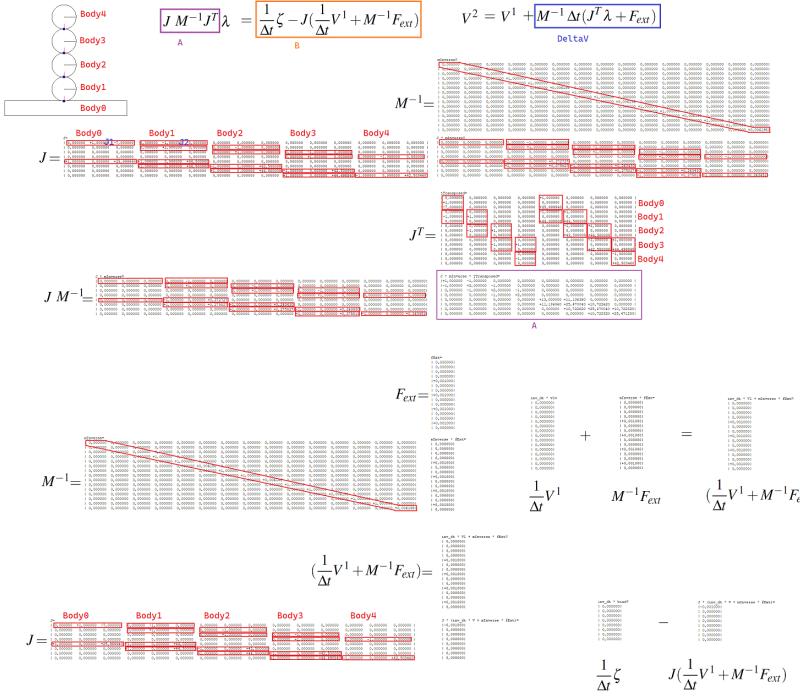
Wenn man beim Matrix-Solver neue Geschwindigkeitswerte für die Körper berechnen will, dann muss man zuerst Matrizen erstellen. Dann ein paar Multiplikationen und Additionen damit machen. Dann per PGS Lambda ermitteln und dann nochmal ein paar Matrizen-Multiplikationen/Additionen, um somit dann die neuen Geschwindigkeitswerte für die Körper zu erhalten.

Sowohl die J-Matrix als auch die mInverse-Matrix enthalten viele Nullen. Man kann diesen Umstand ausnutzen und somit viel Aufwand sparen. Um die Matrizen und ihre ganzen Nullen zu sehen siehe das nächste Bild. Aus diesen Bild wurden folgende Schlussfolgerungen gezogen, wo man Rechenaufwand bei den Matrix-Operationen sparen kann:

- Jedes Constraint-Objekt speichert zwei Kraftvektoren, welche gleichlang und entgegengesetzt auf 2 Körper wirken
- Anstatt die ganze J-Zeile pro Constraint zu speichern, speichere ich nur die beiden 3er-Blöcke in den Variablen J1 und J2 (Richtung der beiden Kraftvektoren)
- Bei der A-Matrix lässt sich die J*mInverse-Multiplikation dadurch vereinfachen, indem ich aus der J-Matrix nur J1 und J2 betrachte und J-Zeile mal mInverse bedeutet, dass J1 mit der Masse von Body1 und J2 mit der Masse von Body2 gewichtet wird
- Jedes Constraint-Objekt speichert eine Zeile aus der A-Matrix. Für die J*mInverse *

jTranspose-Multiplikation nutze ich den Umstand, dass jede Zeile aus j*mInverse und jede Spalte aus jTranpose nur 2 3er-Blöcke hat, wo was steht. Nur wenn zwei Constraints auf den gleichen Körper eine Kraft ausüben, dann muss ich per Vektor-Dot-Produkt zwischen ihnen was rechnen, um ein A-Wert zu erhalten.

- Für den B-Wert wird eine J-Zeile mit einem Spaltenvektor multipliziert. Ich nutze wieder den Umstand, dass nur bei J1 und J2 was steht. Somit muss ich auch nur J1 und J2 jeweils mit den jeweiligen Geschwindigkeits- und Fext-Wert des jeweiligen Körpers per Vektor-Dot verrechnen.
- Für DeltaV nutze ich den Umstand, dass J1/J2*Lambda ein einzelner Constraint-Kraftvektor ist, welcher auf ein einzelnen Körper wirkt. Somit brauche ich nur über alle Constraints iterieren und jedes Constraint-Objekt einen Impuls auf jeweils zwei Körper wirken lassen.
- Hinweis: In der Matrix-Schreibweise taucht der Fext-Term in der DeltaV-Formel mit auf. Die externen Kraft darf nur einmal pro TimeStep wirken. Deswegen habe ich den Constraint-Impuls (Schleife über alle Constraints) und den Force-Extern-Impulse (Schleife über alle Körper) in zwei getrennten Schleifen/Schritten. Da bei der Constraint-Schleife ein Körper mehrmals Constraint-Kräfte erfahren kann, könnte es ansonsten passieren, dass wenn ich dort Fext anwende, der Körper dann mehrmals Fext erfährt.



Siehe im Quelltext unter PhysicEngine/CollisionResolution/JRowAsObject um den neuen Solver zu sehen. Der alte Ansatz befindet sich unter PhysicEngine/CollisionResolution/EnterTheMatrix

Der JRowAsObject-Ansatz funktioniert durch das Einsparen der Plus-Null-Operationen schneller als der Matrixansatz aber was noch nicht so gut ist, dass die Constraint-Objekte aller über die ARow- und B-Property untereinander verknüpft sind. Die Constraint-Objekte müssen in zwei Schritten erzeugt werden. Siehe Bild Schritt 1 und Schritt 2:

```

Schrift 1: Erzeuge J1, J2 und Bias für jedes Constraint-Objekt J1 J2 ζ
Schrift 2: Erzeuge für jedes Constraint die A-Zeile und den B-Spaltenwert
Schrift 3: Ermittle mit PGS für jedes Constraint Lambda
Schrift 4: Verschiebe die Körper aufgrund der Constraintkraft
Schrift 5: Verschiebe die Körper aufgrund der externen Kraft

```

```

private static float CreateLambdaFromConstraint(CConstraint c, Cconstraint[] constraints)
{
    //Berechne die Zeile aus der J1Lambda=Matrix
    float[] J1Lambda = new float[c.constraints.Length];
    Vector3D J1LambdaVec = new Vector3D();
    J1Lambda[0] = c.Bias;
    J1Lambda[1] = c.J1.Z * c.B1.InverseMass;
    J1Lambda[2] = c.J1.Y * c.B1.InverseMass;
    J1Lambda[3] = c.J2.Z * c.B2.InverseMass;
    J1Lambda[4] = c.J2.Y * c.B2.InverseMass;
    for (int i = 0; i < constraints.Length; i++)
    {
        if (constraints[i].J1 == c.J1)
        {
            J1Lambda[5 + i] = constraints[i].Lambda;
        }
    }
    return J1Lambda;
}

private static float CreateLambdaFromBConstraint(CConstraint c, float invc)
{
    //Berechne die Zeile aus der A-Matrix
    float[] J2Lambda = new float[c.constraints.Length];
    Vector3D J2LambdaVec = new Vector3D();
    J2Lambda[0] = c.Bias;
    for (int i = 0; i < constraints.Length; i++)
    {
        if (constraints[i].J2 == c.J2)
        {
            J2Lambda[5 + i] = constraints[i].Lambda;
        }
    }
    return J2Lambda;
}

private static void SolveLambdaFromPGS(CConstraint constraint, int iterations)
{
    for (int i = 0; i < iterations; i++)
    {
        for (int y = 0; y < constraints.Length; y++)
        {
            //Lose constraint(y).Lambda unter Nutzung von Zeile y
            float sum = 0;
            for (int x = 0; x < constraints.Length; x++)
            {
                if (x != y)
                {
                    sum += constraints[x].Lambda * constraints[y].Lambda;
                }
            }
            constraint(y).Lambda = constraint(y).B + sum / constraints[y].Lambda;
            constraint(y).Lambda = constraint(y).Lambda - constraint(y).Lambda * constraints[y].Lambda;
        }
    }
}

private static void MoveBodyWithConstraint(Vector3D constraint, CConstraint constraint, Vector3D body, Vector3D bodyVelocity, float dt)
{
    var force = new Vector3D();
    force.X = constraint.X * body.Mass * dt;
    force.Y = constraint.Y * body.Mass * dt;
    force.Z = constraint.Z * body.Mass * dt;
    body.Force = force;
    body.DerivPosition += force * dt;
    body.Velocity += body.DerivPosition * dt;
    body.AngularVelocity += body.DerivPosition * dt;
}

private static void MoveBodyWithForce(Vector3D constraint, CConstraint constraint, Vector3D body, Vector3D bodyVelocity, float dt)
{
    var force = new Vector3D();
    force.X = constraint.X * body.Mass * dt;
    force.Y = constraint.Y * body.Mass * dt;
    force.Z = constraint.Z * body.Mass * dt;
    body.Force = force;
    body.DerivPosition += force * dt;
    body.Velocity += body.DerivPosition * dt;
    body.AngularVelocity += body.DerivPosition * dt;
}

```

```

10    public static void MoveBodiesWithConstraint(List<IRigidBody> bodies, CollisionPointWithLambda[] collisions,
11    {
12        //Schritt 1: Erzeuge J1, J2 und Bias für jedes Constraint-Objekt
13        float invDt = dt > 0.0f ? 1.0f / dt : 0.0f;
14        var constraints = new ConstraintFactory().CreateConstraints(new ConstraintConstructorData()
15        {
16            Bodies = bodies,
17            Settings = settings,
18            Dt = dt,
19            InvDt = invDt
20        },
21        collisions);
22
23        //Schritt 2: Erzeuge für jedes Constraint die A-Zeile und den B-Spaltenwert
24        foreach (var constraint in constraints)
25        {
26            constraint.ARow = CreateARowForConstraint(constraint, constraints);
27            constraint.B = CreateBValueForConstraint(constraint, invDt);
28        }
29
30        //Schritt 3: Ermittle mit PGS für jedes Constraint Lambda
31        SolveLambdasWithPGS(constraints, settings.IterationCount);

```

Siehe: PhysicEngine/CollisionResolution/JRowAsObject/ResolverHelper.cs

Die Constraint-Objekte kennen sich zwar nicht direkt per Objektverweis untereinander aber zumindest über die ARow- und B-Floatarrays, wo dann jedes Arrayfeld für ein Constraintobjekt steht.

Ein weiterer Nachteil sowohl vom Matrix- als auch JRow-Ansatz ist, dass man gezwungen ist, dass man im Vorhinein angeben muss, wie viel PGS-Schritte man machen will. Erin Catto hat sich mit sein Box2D was überlegt, was er „Sequentielle Impulse“ nennt. Dort sind die Constraint-Objekte dann unabhängig voneinander und man sieht schon Zwischenlösungen für die korrigierten Geschwindigkeitswerte der Körper und könnte dann schon eher die Berechnung der Constraint-Impulse abbrechen, wenn der Geschwindigkeitsfehler der Körper einen bestimmten Wert unterschreitet.

Sequentielle Impulse

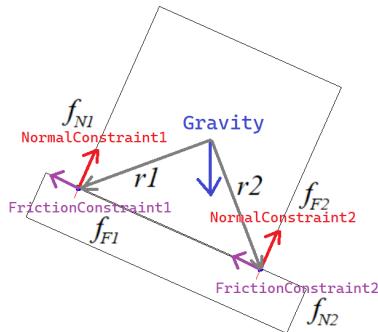
Sowohl der Matrix-Ansatz aus Teil 2 als auch der Ansatz, wo eine J-Zeile ein Constraint-Objekt ist, sind von der Idee her so, dass über den Bias-Wert festgelegt wird, mit welcher Geschwindigkeit zwei Kontaktpunkte aufeinander zu fliegen sollen und dann berechnet man mit PGS genau den Impuls den man braucht, um von V1 zu V2 kommen. Beim sequentiellen Impulsansatz (SI) berechnet man viele kleine Zwischenimpulse und wendet sie sofort auf das aktuelle V an, um somit in mehreren Korrekturschritten das aktuelle V in den gesuchten V2-Punkt umzuwandeln.

Herleitung der Gesamtformel

Kraftgleichung für ein einzelnen Körper

Am Anfang der TimeStep-Methode werden alle Schnittpunkte zwischen allen Körpern ermittelt. Danach muss für jeden Körper die externe Kraft angewendet werden. Aufgrund der angewendeten externen Kraft und der aktuellen Geschwindigkeit resultieren daraus Constraint-Kräfte, welche pro Kontaktspalte auch noch auf die jeweiligen Körper wirken. Somit wirkt auf jeden Körper die Summe all seiner Extern- als auch Constraint-Kräfte womit sich dann ein neuer Geschwindigkeitswert ergibt. Dieser neue Geschwindigkeitswert pro Körper ändert die Position.

Beispiel: Ein Würfel liegt auf einer schrägen Ebene. Die Schwerkraft ist eine Translationskraft, welche auf das Zentrum vom Würfel wirkt. Die beiden unteren Würfecken sind Kollisionspunkte mit der Ebene. Pro Kollisionspunkt gibt es eine NormalConstraint-Kraft, welche den jeweiligen Eckpunkt in Normalenrichtung drückt und die FrictionConstraint-Kraft drückt den Kollisionspunkt genau so stark die Ebene hinauf, dass der Würfel ruhig liegen bleibt.



Zuerst wirkt nur die Schwerkraft (externe Kraft) und das Drehmoment ist Null:

$$f_{ext} = \begin{bmatrix} 0 \\ Gravity \end{bmatrix} \quad \tau_{ext} = 0$$

Daraus resultierend entstehen die Constraint-Kräfte welche an den Kollisionspunkten gegen den Würfel drücken.

Jeder Kraftpfeil, der an ein Punkte außerhalb des Zentrums an einen Körper ansetzt, verschiebt sowohl das Zentrum aber er übt auch ein Drehmoment entsprechend des Hebelarms aus.

Die 4 Constraintkräfte werden summiert und üben eine Translationskraft und ein Drehmoment aus:

$$f_c = f_{N1} + f_{F1} + f_{N2} + f_{F2} \quad \tau_c = r1 \times f_{N1} + r1 \times f_{F1} + r2 \times f_{N2} + r2 \times f_{F2}$$

Laut der „F=m*a“-Formel und Semi Implizit Euler gelten pro Körper folgende Formeln:

$$m \dot{v} = f_c + f_{ext} \quad I \dot{\omega} = \tau_c + \tau_{ext}$$

$$\dot{v} = \frac{v_2 - v_1}{\Delta t}$$

v_1 = Geschwindigkeit des Körpers vor Anwendung von Schwer- und Constraintkraft

v_2 = Geschwindigkeit des Körpers nach Anwendung der Schwer- und Constraintkraft

Die 4 Constraint-Kräfte sind aktuell noch unbekannt: $f_C = f_{NI} + f_{FI} + f_{N2} + f_{F2}$

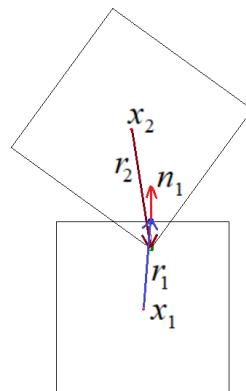
Deswegen kann v_2 aktuell noch nicht ausgerechnet werden. Der Würfel liegt ruhig da ($v_1 = 0$), es wirkt die Schwerkraft, wodurch es 4 Constraint-Kräfte gibt und die resultierende Geschwindigkeit soll dann wieder Null sein ($v_2 = 0$).

Für die unteren Würfelecken (Kontaktpunkte) gilt, dass sie sich sowohl in Richtung Kontaktnormalen als auch in Richtung Kontakttangente nicht bewegen. D.h. Die Relativgeschwindigkeit zwischen dem jeweiligen Kontaktpunkt vom Würfel als auch dem Kontaktpunkt vom Boden soll sich nicht ändern.

NormalConstraint

Die Geschwindigkeit, mit der sich zwei Kontaktpunkte in Richtung Kontaktnormalen bewegen erhalte ich, indem ich eine Funktion aufstelle, welche den Abstand in Normalenrichtung misst und diese Funktion dann nach der Zeit ableite.

Beispiel: Der obere Würfel kollidiert mit den unteren Würfel. R1 zeigt vom Zentrum zum Kollisionspunkt von Würfel 1 und r2 zeigt vom Zentrum zum Kollisionspunkt von Würfel 2.



Mit dieser Funktion messe ich, wie sehr der Kollisionspunkt von Würfel 2 in die obere Kante von Würfel 1 eingedrungen ist:

$$C_n = (x_2 + r_2 - x_1 - r_1) * n_1$$

Die Ableitung davon misst, mit welcher Geschwindigkeit die Kollisionspunkte in Richtung Kontaktnormalen sich aufeinander zu bewegen. Als Input bekommt sie die Geschwindigkeitswerte von beiden Würfeln:

$$\dot{C}_n(v_1, \omega_1, v_2, \omega_2) = (v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1) * n_1 + (x_2 + r_2 - x_1 - r_1) * \omega_1 \times n_1$$

Um diese Funktion zu vereinfachen wird der zweite Plus-Term weggelassen. Es wird davon ausgegangen, dass die beiden Kontaktpunkte nah beieinander liegen und $(x_2 + r_2 - x_1 - r_1)$ somit 0 ergibt. Diese Funktion wird nun so umgestellt, dass sie als Vektor-Produkt geschrieben wird:

$$\dot{C}_n(v_1, \omega_1, v_2, \omega_2) = (v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1) * n_1 \rightarrow n_1 \text{ in die Klammer rein}$$

$$\dot{C}_n(v_1, \omega_1, v_2, \omega_2) = v_2 n_1 + (\omega_2 \times r_2) n_1 - v_1 n_1 - (\omega_1 \times r_1) n_1 \rightarrow a * (b \times c) = (a \times b) * c$$

$$\dot{C}_n(v_1, \omega_1, v_2, \omega_2) = v_2 n_1 + \omega_2 (r_2 \times n_1) - v_1 n_1 - \omega_1 (r_1 \times n_1) \rightarrow \text{sortiere die Plus-Terme}$$

$$\dot{C}_n(v_1, \omega_1, v_2, \omega_2) = -v_1 n_1 - \omega_1 (r_1 \times n_1) + v_2 n_1 + \omega_2 (r_2 \times n_1) \rightarrow \text{schreibe als Vektor-Produkt}$$

$$\dot{C}_n(v_1, \omega_1, v_2, \omega_2) = J * V = \begin{bmatrix} -n_1^T \\ -r_1 \times n_1 \\ n_1^T \\ r_2 \times n_1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix}$$

Ich kann die \dot{C}_n Funktion nun benutzen, um eine Bedingung dafür aufzustellen, wie die Geschwindigkeiten der beiden Würfel sein soll, nachdem die NormalConstraint-Kraft an den Kontaktpunkten gewirkt hat.

Ich definiere dazu V_1 mit $V_1 = \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix}$ → Das ist die Geschwindigkeit von beiden Würfeln zum Kollisionszeitpunkt noch bevor eine externe Kraft oder Constraint-Kraft angewendet wurde. Dieser Spaltenvektor enthält 6 Float-Zahlen.

V_2 ist wie V_1 definiert nur dass es die Geschwindigkeit der beiden Würfel nach Anwendung der externen Kraft und der Constraint-Kraft angibt.

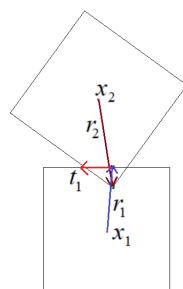
Das noch unbekannte V_2 soll bei ein Kontaktspunkt so sein, das es folgende Gleichung erfüllt:

$$J * V_2 = -J * V_1 * e \rightarrow \text{Das ist unsere NormalConstraint mit Restitution } e (=0..1)$$

FrictionConstraint

Die Kollisionstangente ist definiert über: $t_1 = \begin{bmatrix} -n_{1,X} \\ n_{1,Y} \end{bmatrix}$

Die Reibung soll beim Kollisionspunkt in Tangentenrichtung t1 wirken:



Ich nutze den gleichen Weg zur Herleitung wie bei der Normalconstraint nur mit t_1 anstelle von n_1 und erhalte somit eine Funktion, welche mir sagt, welche Relativgeschwindigkeit zwei Kontaktspunkte in Tangentenrichtung haben:

$$\dot{C}_t(v_1, \omega_1, v_2, \omega_2) = J * V = \begin{bmatrix} -t_1^T \\ -r_1 \times t_1 \\ t_1^T \\ r_2 \times t_1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix}$$

Die Reibung versucht die Relativgeschwindigkeit in Tangentenrichtung auf Null zu bekommen.

Aus dem Reibungskoeffizient ergibt sich, wie viele TimeSteps nötig sind, um die Bewegung zwischen zwei Körpern zu stoppen. Wir wollen innerhalb von ein TimeStep die Bewegung auf 0 bringen, also nutzen wir ein unendlich hohen Reibungskoeffizienten. Somit muss V_2 folgender Bedingung genügen:

$$J * V_2 = 0 \rightarrow \text{Das ist unsere FrictionConstraint}$$

Constraint-Gleichung als Ebene

Pro Kontaktpunkt gibt es jeweils eine Normal- und eine Frictionconstraint, welche vorgibt, welche Geschwindigkeitswerte die beiden kollidierenden Körper nach der Constraint-Kraft-Anwendung haben sollen. Alle Constraint-Gleichungen sehen vom Aufbau her so aus:

$$J * V_2 = \xi$$

ξ ist ein von mir vorgegebener Wert womit ich festlege, welche Relativgeschwindigkeit die Kontaktpunkte nach der Constraint-Kraft-Anwendung haben sollen. Bei der FrictionConstraint definiere ich $\xi = 0$ und bei der NormalConstraint $\xi = -J * V_1$.

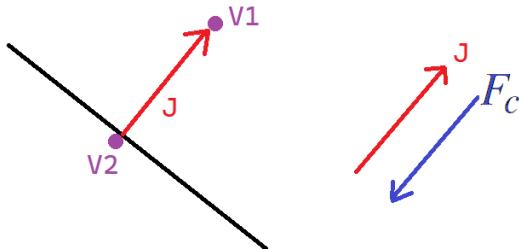
Wenn man sich die Gleichung $J * V_2 = \xi$ ansieht, dann stellt man fest, dass sie eine Ebenengleichung in Hessescher Normalform ist:

https://de.wikipedia.org/wiki/Hessesche_Normalform

$$\vec{x} \cdot \vec{n}_0 - d = 0$$

Das J ist die Normale der Ebene (entspricht n_0) und das ξ ist der Abstand zum Koordinatenursprung (entspricht d). Das V2 sind all die Punkte die auf der Ebene liegen. In unseren Fall bestehen die Vektoren J und V2 aus 6 skalaren Werten. Somit handelt es sich hier um eine 6-dimensionale Ebene.

Wenn ich mir das ganze Grafisch vorstelle, dann ist der schwarze Strich die Constraint-Ebene, welche für eine einzelne Constraint-Gleichung alle gültigen V2-Punkte angibt.



Wenn wir uns am Punkt V1 befinden und wir wollen ein Punkt V2 dadurch auf der Ebene ermitteln, indem wir in Richtung Ebenennormale gehen, dann gilt für V2 somit $V_2 = V_1 + J^T * \lambda$. Lambda ist eine skalare Zahl, welche den Abstand von V1 zur Ebene angibt. Wenn ich zwei Kontaktpunkte in Richtung Kontaktnormalen/Tangente verschieben will, dann heißt das, dass ich muss sie in J-Richtung verschieben. Möchte ich also, dass zwei Kontaktpunkte mit einer bestimmten Geschwindigkeit in J-Richtung beschleunigt werden, dann muss eine Kraft in J-Richtung wirken. Diese Kraft nennen wir Constraint-Kraft F_C und sie ist definiert über: $F_C = J^T \lambda$. Lambda (λ) ist eine skalare Zahl, welche angibt, wie lange die Constraint-Kraft wirken muss, um die Relativgeschwindigkeit von zwei Kontaktpunkten auf den Wert von ξ zu bringen.

Herleitung des Constraint-Impulses

Wenn zwei Würfel kollidieren dann haben beide Würfel vor der Anwendung der externen Kraft und aller Constraint-Kräfte die Geschwindigkeit V1, was ein 6-dimensionaler Vektor ist, der so aussieht:

$$V_1 = \begin{bmatrix} v_{1,X} \\ v_{1,Y} \\ \omega_1 \\ v_{2,X} \\ v_{2,Y} \\ \omega_2 \end{bmatrix}$$

Wenn wir die externe Kraft nun Δt Sekunden lang wirken lassen, ergeben sich die neuen Geschwindigkeitswerte für beide Würfel somit über:

$$v_{1D} = v_1 + \frac{f_{ext}}{m_1} \Delta t \quad \omega_{1D} = \omega_1 + \frac{\tau_{ext}}{I_1} \Delta t \rightarrow \text{Geschwindigkeit von Würfel 1 nach Fext-Anwendung}$$

$$v_{2D} = v_2 + \frac{f_{ext}}{m_2} \Delta t \quad \omega_{2D} = \omega_2 + \frac{\tau_{ext}}{I_2} \Delta t \rightarrow \text{Geschwindigkeit von Würfel 2 nach Fext-Anwendung}$$

Wir definieren V_D (V-Damaged) über $V_D = \begin{bmatrix} v_{1D,X} \\ v_{1D,Y} \\ \omega_{1D} \\ v_{2D,X} \\ v_{2D,Y} \\ \omega_{2D} \end{bmatrix}$.

Das ist die Geschwindigkeit von beiden Würfeln nach der Anwendung der externen Kraft.

Außerdem definieren wir die inverse Masse-Matrix für zwei Körper über:

$$M^{-1} = \begin{bmatrix} m_1^{-1} & 0 & 0 & 0 & 0 & 0 \\ 0 & m_1^{-1} & 0 & 0 & 0 & 0 \\ 0 & 0 & I_1^{-1} & 0 & 0 & 0 \\ 0 & 0 & 0 & m_2^{-1} & 0 & 0 \\ 0 & 0 & 0 & 0 & m_2^{-1} & 0 \\ 0 & 0 & 0 & 0 & 0 & I_2^{-1} \end{bmatrix}$$

V_2 soll dadurch definiert sein, dass es die Geschwindigkeit beider Würfel nach Anwendung der externen Kraft und der Constraint-Kraft ist. Um V_2 bestimmen zu können benötigen wir den Constraint-Impuls p_C . Für diesen Impuls gilt $p_C = M * \Delta V$ was bedeutet:

Wenn der Impuls p_C wirkt, dann ändert sich die Geschwindigkeit um ΔV

Es gelten folgende Ausgangsgleichungen um eine Formel für p_C zu ermitteln:

$$V_2 = V_D + M^{-1} * p_C \quad (1) \rightarrow \text{Ergibt sich aus der } p=m*v\text{-Regel}$$

$$p_C = J^T * \lambda \quad (2) \rightarrow \text{Die Constraint-Normale ist parallel zur Constraint-Kraft}$$

$$J * V_2 = \xi \quad (3) \rightarrow V_2 \text{ ist so definiert, dass es ein Punkt auf der Constraint-Ebene sein soll}$$

Gleichung (1) wird anstelle von V_2 in Gleichung (3) eingesetzt:

$$J * (V_D + M^{-1} * p_C) = \xi \rightarrow \text{Multipliziere mit } J$$

$$J * V_D + J * M^{-1} * p_C = \xi \rightarrow \text{Ersetze } p_C \text{ mit Gleichung (2)}$$

$$J * V_D + J * M^{-1} * J^T * \lambda = \xi \rightarrow \text{Minus } J * V_D$$

$$J * M^{-1} * J^T * \lambda = \xi - J * V_D \rightarrow \text{Durch } J * M^{-1} * J^T$$

$$\lambda = \frac{\xi - J * V_D}{J * M^{-1} * J^T} \rightarrow \text{Setze diese Gleichung für Lambda bei (2) ein}$$

$$p_C = J^T * \frac{\xi - J * V_D}{J * M^{-1} * J^T} \rightarrow \text{(Formel 4) Constraint-Impuls für 2 Körper}$$

Diese Formel die wir hier jetzt haben korrigiert die Geschwindigkeitswerte von zwei Körpern für eine einzelne Constraint-Gleichung. Wenn wir aber an das Beispiel mit den Würfel denken, der auf

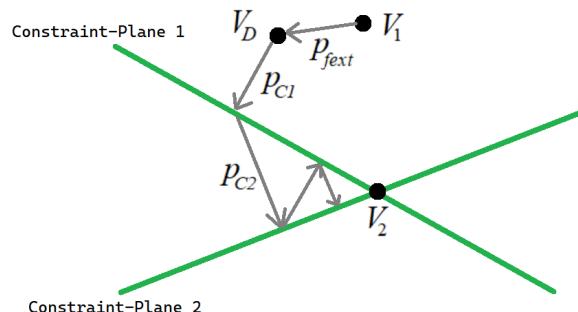
einer schrägen Ebene liegt, dann haben wir dort zwei Kontaktpunkte und pro Kontaktpunkt eine Normal- und eine Friction-Constraint-Gleichung, die eingehalten werden muss. Da alle 4 Constraints sich auf das gleiche Body1-Body2-Paar beziehen heißt dass für alle 4 Constraints wird der gleiche V2-Vektor verwendet und dieser V2-Vektor/Punkt muss allen 4 Constraint-Gleichungen genügen.

Damit man ein gültigen V2-Punkt finden kann ist es nötig, dass sich die Constraint-Ebenengleichungen an ein Punkt schneiden. Dieser Schnittpunkt ist dann der gesuchte V2-Punkt. Man kann diesen Punkt dadurch ermitteln, indem man zuerst für jeden Körper die externe Kraft wirken lässt wodurch sie alle dann alle an ihren individuellen VD-Punkt sind. Dann iteriert man über alle Constraints und für jede Constraint-Ebene wendet man den Constraint-Impuls p_c an, der so aussieht:

$$p_c = J^T * \frac{\xi - J * V}{J * M^{-1} * J^T} \rightarrow \text{Formel 5 } V_D \text{ wurde hier durch } V \text{ ersetzt.}$$

V = Aktuelle Geschwindigkeit von jeweils zwei Körpern, die zum Constraint-Objekt gehören.

Grafisch sieht das ganze für zwei Constraint-Ebenen dann so aus:



Hinweis: Die Idee dafür habe ich von hier <http://allenhou.net/files/slides/2014/constraint-based-physics.pdf> Seite 13 aber ich habe sie noch um den pfext-Impuls erweitert.

Der V1-Punkt beschreibt die Geschwindigkeit von zwei Körpern vor der Korrektur. Dann wird die externe Kraft über den Zeitraum von Δt angewendet was $p_{fext} = f_{ext} * \Delta t$ entspricht so dass man zu den Punkt V_D gelangt. Von dort aus wende ich immer abwechseln für beide Constraints die Constraint-Impulsformel (5) an und komme so den Punkt V_2 immer näher. In diesem Beispielbild hier wird anhand von zwei Constraints gezeigt, wie man den V2-Punkt ermittelt. Wenn man zwischen zwei Körpern noch mehr Constraints hat, dann müssen alle Constraint-Gleichungen bei einem Punkt zusammen laufen damit das Verfahren funktioniert. Wir gehen davon aus, dass das so gegeben ist.

Das Bild soll zeigen: Wenn ich mehrmals über alle Constraints iteriere und den zugehörigen Constraint-Impuls anwende, dann kann ich ein V_2 finden, was alle Constraint-Gleichungen zwischen zwei Körpern erfüllt.

Pro Körper müssen zwei Impulsarten angewendet werden. Zum einen die Summe der externen Kräfte (Schwerkraft) welche über p_{fext} gegeben ist. Außerdem die Summe über alle Constraint-Impulse.

Für die Anwendung des p_{fext} -Impulses ist es leichter, wenn man über alle Starrkörper aus der PhysicScene iteriert und dann einfach p_{fext} anwendet.

Da ein Constraint-Impuls sich immer auf zwei Körper bezieht, ist es leichter, wenn man für die Constraint-Impulsanwendung über alle Constraints iteriert und dann für jede Constraint den p_c -Constraint-Impuls (Formel 5) auf beide zugehörige Körper anwendet.

Für die Normalconstraint lautet der Nenner der Constraint-Formel:

$$J * M^{-1} * J^T = m_1^{-1} + (r_1 \times n_1)^2 * I_1^{-1} + m_2^{-1} + (r_2 \times n_1)^2 * I_2^{-1}$$

Die Herleitung dafür steht hier:

$$\begin{aligned}
 M^{-1} &= \begin{bmatrix} m_1^{-1} & 0 & 0 & 0 \\ 0 & I_1^{-1} & 0 & 0 \\ 0 & 0 & m_2^{-1} & 0 \\ 0 & 0 & 0 & I_2^{-1} \end{bmatrix} & \text{Normalconstraint} \\
 J^T &= \begin{bmatrix} -n_1^T \\ -r_1 \times n_1 \\ n_1^T \\ r_2 \times n_1 \end{bmatrix} & J = \begin{bmatrix} -n_1 & -r_1 \times n_1 & n_1 & r_2 \times n_1 \end{bmatrix} \\
 J * M^{-1} &= \begin{bmatrix} m_1^{-1} & 0 & 0 & 0 \\ 0 & I_1^{-1} & 0 & 0 \\ 0 & 0 & m_2^{-1} & 0 \\ 0 & 0 & 0 & I_2^{-1} \end{bmatrix} \begin{bmatrix} -n_1 m_1^{-1} & -r_1 \times n_1 I_1^{-1} & n_1 m_2^{-1} & r_2 \times n_1 I_2^{-1} \end{bmatrix} \\
 J * M^{-1} * J^T &= \begin{bmatrix} -n_1^T \\ -r_1 \times n_1 \\ n_1^T \\ r_2 \times n_1 \end{bmatrix} \begin{bmatrix} m_1^{-1} + (r_1 \times n_1)^2 I_1^{-1} + m_2^{-1} + (r_2 \times n_1)^2 I_2^{-1} \end{bmatrix} \\
 J * M^{-1} * J^T &= \boxed{\begin{bmatrix} m_1^{-1} + (r_1 \times n_1)^2 I_1^{-1} + m_2^{-1} + (r_2 \times n_1)^2 I_2^{-1} \end{bmatrix}} \quad n_1 n_1^T = 1
 \end{aligned}$$

Der erste vorsichtige Entwurf für das sequentielle Impuls-Verfahren sieht nun so aus:

```

10  public static void MoveBodiesWithConstraint(List<IRigidBody> bodies, RigidbodyCollision[] collisions, SolverSettings settings, float dt)
11  {
12      var constraints = collisions.Select(x => new NormalConstraint0(x)).ToList();
13
14      //1. Apply Gravity-Force
15      foreach (var body in bodies)
16      {
17          body.Velocity.X += body.InverseMass * body.Force.X * dt;
18          body.Velocity.Y += body.InverseMass * body.Force.Y * dt;
19          body.AngularVelocity += body.InverseInertia * body.Torque * dt;
20      }
21
22      //2. Apply Constraint-Forces
23      for (int i = 0; i < settings.IterationCount; i++)
24      {
25          foreach (var c in constraints)
26          {
27              Vector2D v1 = c.B1.Velocity + new Vector2D(-c.B1.AngularVelocity * c.R1.Y, c.B1.AngularVelocity * c.R1.X);
28              Vector2D v2 = c.B2.Velocity + new Vector2D(-c.B2.AngularVelocity * c.R2.Y, c.B2.AngularVelocity * c.R2.X);
29              Vector2D relativeVelocity = v2 - v1;
30              float velocityInForceDirection = relativeVelocity * c.ForceDirection;
31              float impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection);
32              ApplyImpulse(c, impulse);
33          }
34      }
35  }
36  private static void ApplyImpulse(NormalConstraint0 c, float impulse)
37  {
38      Vector2D impulseVector = impulse * c.ForceDirection;
39      c.B1.Velocity -= impulseVector * c.B1.InverseMass;
40      c.B1.AngularVelocity -= Vector2D.ZValueFromCross(c.R1, impulseVector) * c.B1.InverseInertia;
41
42      c.B2.Velocity += impulseVector * c.B2.InverseMass;
43      c.B2.AngularVelocity += Vector2D.ZValueFromCross(c.R2, impulseVector) * c.B2.InverseInertia;
44  }
45  public NormalConstraint0(RigidbodyCollision point)
46  {
47      this.ImpulseMass = 1.0f / (B1.InverseMass + B2.InverseMass +
48      r1crossN * r1crossN * B1.InverseInertia +
49      r2crossN * r2crossN * B2.InverseInertia);
50
51      this.ForceDirection = c.Normal;
52      this.Bias = GetBias(c, this.R1, this.R2);
53  }
54  private float ... GetBias(RigidbodyCollision c, Vector2D r1, Vector2D r2)
55  {
56      Vector2D v1 = c.B1.Velocity + new Vector2D(-c.B1.AngularVelocity * r1.Y, c.B1.AngularVelocity * r1.X);
57      Vector2D v2 = c.B2.Velocity + new Vector2D(-c.B2.AngularVelocity * r2.Y, c.B2.AngularVelocity * r2.X);
58      Vector2D relativeVelocity = v2 - v1;
59      float velocityInNormal = relativeVelocity * c.Normal;
60      float restitution = Math.Min(c.B1.Restitution, c.B2.Restitution);
61      return -restitution * velocityInNormal;
62  }

```

$$P_C = J^T * \frac{[\xi - J * V]}{J * M^{-1} * J^T}$$

$$J^T = \begin{bmatrix} -n_1^T \\ -r_1 \times n_1 \\ n_1^T \\ r_2 \times n_1 \end{bmatrix}$$

$$J * M^{-1} * J^T = \boxed{\begin{bmatrix} m_1^{-1} + (r_1 \times n_1)^2 I_1^{-1} + m_2^{-1} + (r_2 \times n_1)^2 I_2^{-1} \end{bmatrix}}$$

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Examples/ResolverHelper0.cs

Hier wird erst mal nur eine Normalconstraint verwendet. Man könnte aber die FrictionConstraint

laut der oben hergeleiteten Formel für J und dem Bias-Wert von 0 noch einfach hinzufügen aber das Beispiel sollte erst mal möglichst einfach sein.

Wenn man mit diesen Verfahren nun ein Würfelstapel simuliert und der Stapel kippt um, dann wird man sehen, dass die Würfel zusammen kleben. Der Grund dafür ist, weil der Normalimpuls nicht nur drückend sondern auch ziehend wirkt. Wenn beim Normalimpuls auf Zeile 31 eine positive Zahl heraus kommt, dann werden die Kontaktpunkte auseinander gedrückt. Kommt dort eine negative Zahl heraus, dann werden die Kontaktpunkte zusammen gezogen. Der Gedanke erscheint nur naheliegend, dass wir Zeile 31 von so:

```
float impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection);
```

nach so abändern:

```
float impulse = Math.Max(0, c.ImpulseMass * (c.Bias - velocityInForceDirection));
```

Es gibt aber nicht nur die Normalconstraint sondern auch noch die Frictionconstraint. Diese soll schon drückend als auch ziehend wirken. Deswegen erweitern wir die Constraint-Klasse noch um zwei weitere Propertys:

```
public float MinImpulse { get; }  
2 Verweise  
public float MaxImpulse { get; }
```

Und bei der Impuls-Formel fügen wir noch Clamping ein:

```
float impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection);  
impulse = MathHelp.Clamp(impulse, c.MinImpulse, c.MaxImpulse);
```

Um es leichter verständlich zu machen lassen wir außerdem kurz die Schwerkraft weg.

Sequentielle Impulse im Vergleich zu Iterative Impulse

```
11  public static void MoveBodiesWithConstraint(RigidBodyCollision[] collisions, SolverSettings settings)  
12  {  
13      var constraints = collisions.Select(x => new NormalConstraint1(x)).ToArray();  
14  
15      for (int i = 0; i < settings.IterationCount; i++)  
16      {  
17          foreach (var c in constraints)  
18          {  
19              Vector2D v1 = c.B1.Velocity + new Vector2D(-c.B1.AngularVelocity * c.R1.Y, c.B1.AngularVelocity * c.R1.X);  
20              Vector2D v2 = c.B2.Velocity + new Vector2D(-c.B2.AngularVelocity * c.R2.Y, c.B2.AngularVelocity * c.R2.X);  
21              Vector2D relativeVelocity = v2 - v1;  
22              float velocityInForceDirection = relativeVelocity * c.ForceDirection;  
23              float impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection);  
24              impulse = MathHelp.Clamp(impulse, c.MinImpulse, c.MaxImpulse);  
25              ApplyImpulse(c, impulse);  
26          }  
27      }  
28  }  
29  
30  public NormalConstraint1(CollisionPointWithImpulse point)  
31  {  
32      this.ImpulseMass = 1.0f / (B1.InverseMass + B2.InverseMass +  
33      r1crossN * r1crossN * B1.InverseInertia +  
34      r2crossN * r2crossN * B2.InverseInertia);  
35  
36      this.ForceDirection = c.Normal;  
37      this.Bias = GetBias(c, this.R1, this.R2);  
38      this.MinImpulse = 0;  
39      this.MaxImpulse = float.MaxValue;  
40  }  
41  
42  private float GetBias(CollisionPointWithImpulse c, Vector2D r1, Vector2D r2)  
43  {  
44      Vector2D v1 = c.B1.Velocity + new Vector2D(-c.B1.AngularVelocity * r1.Y, c.B1.AngularVelocity * r1.X);  
45      Vector2D v2 = c.B2.Velocity + new Vector2D(-c.B2.AngularVelocity * r2.Y, c.B2.AngularVelocity * r2.X);  
46      Vector2D relativeVelocity = v2 - v1;  
47      float velocityInNormal = relativeVelocity * c.Normal;  
48      float restituion = Math.Min(c.B1.Restituion, c.B2.Restituion);  
49      return -restituion * velocityInNormal;  
50  }
```

Quelltext: PhysicEngine/CollisionResolution/SequentiellImpulse/Examples/ResolverHelper1.cs

Dieses vereinfachte Sequentielle Impuls-Verfahren ist nun in der Lage alle NoGravity-Testfälle zu bestehen. Es hat viel Ähnlichkeit mit dem Iterativen Impuls-Verfahren aus Teil 1 was im Vergleich dazu so aussieht:

Sequentiell Impulse

```

11 public static void MoveBodiesWithConstraint(RigidBodyCollision[] collisions, SolverSettings settings)
12 {
13     var constraints = collisions.Select(x => new NormalConstraint1(x).ToArray());
14
15     for (int i = 0; i < settings.IterationCount; i++)
16     {
17         foreach (var c in constraints)
18         {
19             Vector2D v1 = c.B1.Velocity + new Vector2D(c.B1.AngularVelocity * c.R1.Y, c.B1.AngularVelocity * c.R1.X);
20             Vector2D v2 = c.B2.Velocity + new Vector2D(-c.B2.AngularVelocity * c.R2.Y, c.B2.AngularVelocity * c.R2.X);
21
22             float velocityInForceDirection = relativeVelocity * c.ForceDirection;
23             float impulse = [c.ImpulseMass] * [c.Bias] * velocityInForceDirection;
24             impulse = Math.Min(impulse, c.MinImpulse, c.MaxImpulse);
25             ApplyImpulse(c, impulse);
26         }
27     }
28 }
29
30 public NormalConstraint1(CollisionPointWithImpulse point)
31 {
32     this.ImpulseMass = 1.0f / (B1.InverseMass + B2.InverseMass +
33     rCrossH * rCrossN + B1.InverseInertia +
34     rCrossH * rCrossN + B2.InverseInertia);
35
36     this.bias = GetBias(c, this.R1, this.R2);
37     this.minImpulse = 0f;
38     this.maxImpulse = float.MaxValue;
39 }
40
41 private float GetBias(CollisionPointWithImpulse c, Vector2D r1, Vector2D r2)
42 {
43     Vector2D v1 = c.B1.Velocity + new Vector2D(c.B1.AngularVelocity * r1.Y, c.B1.AngularVelocity * r1.X);
44     Vector2D v2 = c.B2.Velocity + new Vector2D(c.B2.AngularVelocity * r2.Y, c.B2.AngularVelocity * r2.X);
45
46     Vector2D relativeVelocity = v2 - v1;
47     float velocityInNormal = relativeVelocity * c.Normal;
48     float restitution = Math.Min(c.B1.Restitution, c.B2.Restitution);
49     return -restitution * velocityInNormal;
50 }
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88

```

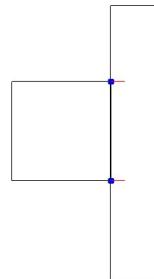
Bei beiden Verfahren hat man eine Iterationsschritt-Schleife und eine Schleife über alle Kontaktpunkte. Bei beiden Verfahren wird dann pro Kontaktspunkt ein Impuls berechnet welcher dann angewendet wird, um jeweils zwei Körper voneinander weg zu drücken. Der Impuls wird bei beiden auch nach der gleichen Formel erstellt, indem die ImpulseMasse (grün) berechnet wird.

Der Impulse wirkt bei beiden in Richtung Kontaktnormalen (rot). Bei beiden wird die Relativgeschwindigkeit der beiden Kontaktspunkte berechnet (orange) und bei beiden wird daraus dann ein velocityInForceDirection/velocityInNormal berechnet (lila).

Der entscheidende Punkt ist der Bias-Wert (gelb markiert). Von der Formel her wird bei beiden „-restitution*velocityInNormal“ als Bias-Wert verwendet aber beim Sequentiellen Impulse-Verfahren wird der Bias-Wert einmal am Anfang vor der Iterationsschleife erstellt. Beim iterativen Verfahren wird der Bias-Wert innerhalb der Kontaktspunkte-Schleife berechnet. Der Bias-Wert ist der Sollwert, wo ich der Impulsformel von außen sage, welche Relativgeschwindigkeit die Kontaktspunkte haben sollen.

Testfall – CubeAgainstWall – Ohne Reibung

Um jetzt besser zu verstehen, wie Sequentielle Impulse funktioniert wollen wir den Testfall betrachten, wo ein Würfel ohne Schwerkraft mit einer gegebenen Anfangsgeschwindigkeit gegen eine Wand fliegt. Der Würfel soll an der Wand abprallen ohne dass er sich dreht. Der aktuelle TimeStep sieht so aus, das der Würfel die Wand berührt und zwei Kontaktspunkte ermittelt wurden:



Auf das Würfel-Gegen-die-Wand-Beispiel bezogen wirkt sich dieser Unterschied in der Bias-Berechnung wie folgt aus.

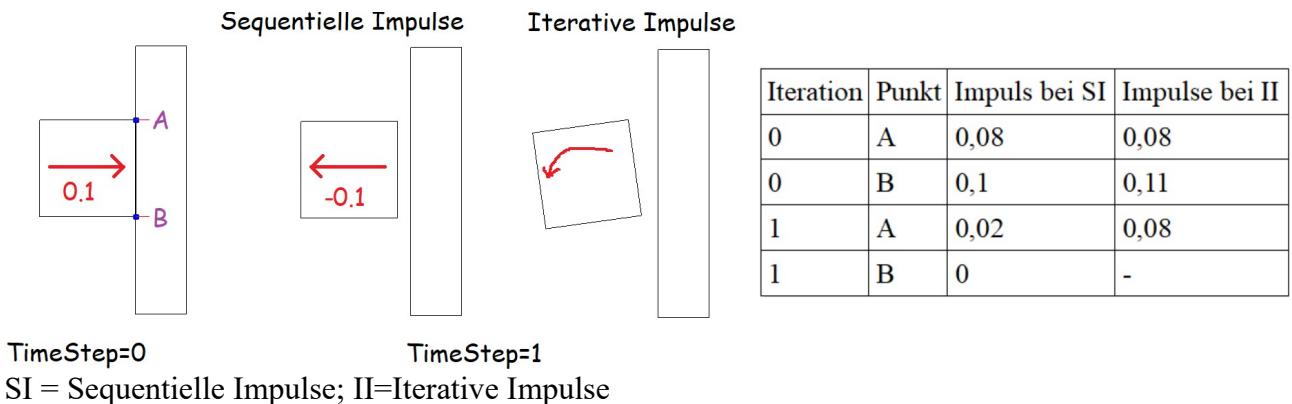
Bei beiden Verfahren wird zuerst der obere Kontaktspunkt A genommen und dann wird ein Impuls von 0.8 angewendet, welcher genau nach links zeigt. Durch diesen Impuls bekommt der Würfel eine Linksdrehung was die untere Ecke stärker gegen die Wand hauft.

Iterativ Impulse

```

9 public static void MoveBodiesWithConstraintUntilAllCollisionsAreResolved(RigidBodyCollision[] collisions, int maxIterationCount)
10 {
11     for (int i = 0; i < maxIterationCount; i++)
12         foreach (var collision in collisions)
13             ApplyImpulse(collision);
14 }
15
16 private static void ApplyImpulse(RigidBodyCollision collision)
17 {
18     var B1 = collision.B1;
19     var B2 = collision.B2;
20     Vector2D n = collision.Normal;
21
22     Vector2D start = collision.Start * (B2.InverseMass / (B1.InverseMass + B2.InverseMass));
23     Vector2D end = collision.End * (B1.InverseMass / (B1.InverseMass + B2.InverseMass));
24     Vector2D p = start + end;
25     Vector2D r1 = p - B1.Center;
26     Vector2D r2 = p - B2.Center;
27
28     Vector2D v1 = B1.Velocity + new Vector2D(-B1.AngularVelocity * r1.Y, B1.AngularVelocity * r1.X);
29     Vector2D v2 = B2.Velocity + new Vector2D(-B2.AngularVelocity * r2.Y, B2.AngularVelocity * r2.X);
30
31     Vector2D relativeVelocity = v2 - v1;
32
33     float velocityInNormal = relativeVelocity * n;
34
35     if (velocityInNormal >= 0) return; //if objects moving apart ignore
36     float restitution = Math.Min(B1.Restitution, B2.Restitution);
37     float rCrossH = Vector2D.ZValueFromCross(r1, n);
38     float rCrossN = Vector2D.ZValueFromCross(r2, n);
39
40     float jn = [restitution * velocityInNormal] - velocityInNormal;
41     jn = jn / (B1.InverseMass + B2.InverseMass +
42     RCrossH * RCrossH * B1.InverseInertia +
43     RCrossN * RCrossN * B2.InverseInertia);
44
45     Vector2D impulse = n * jn;
46
47     B1.Velocity += impulse * B1.InverseMass;
48     B2.Velocity += impulse * B2.InverseMass;
49
50     B1.AngularVelocity -= rCrossH * jn * B1.InverseInertia;
51     B2.AngularVelocity -= rCrossN * jn * B2.InverseInertia;
52 }
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88

```



Nun wird für den unteren Kontaktpunkt B der Impuls angewendet. SI wendet ein Impuls von 0.1 und II ein Impuls von 0.11 an. Der Grund für den Unterschied ist, weil SI hat einmal am Anfang beim Erstellen des NormalConstraint-Objektes über den Bias-Wert sich gemerkt hat, dass sowohl für Punkt A als auch B der Bias 0.1 ist. D.h. Der Würfel fliegt mit 0.1 nach Rechts und nun sage ich mit den Bias von 0.1, dass nun beide Kontaktpunkte mit einer Geschwindigkeit von 0.1 nach der Kollisionsauflösung nach links fliegen sollen. Bei II wird nun für Punkt B ein Bias von 0.1 bestimmt sondern von 0.14. Das kommt daher, dass der Würfel durch den Impuls am Punkt A eine Linksdrehung erhalten hat, was dazu führt, dass die untere rechte Ecke viel stärker gegen die Wand gedrückt wird. II will nun diesen viel schneller sich bewegenden Punkt B an der Wand reflektieren und muss somit ein größeren Impuls anwenden. II berechnet den Biaswert also immer aus der gerade vorhandenen Geschwindigkeit des Würfels und merkt sich nicht, wie die Werte am Anfang vor der Korrektur waren.

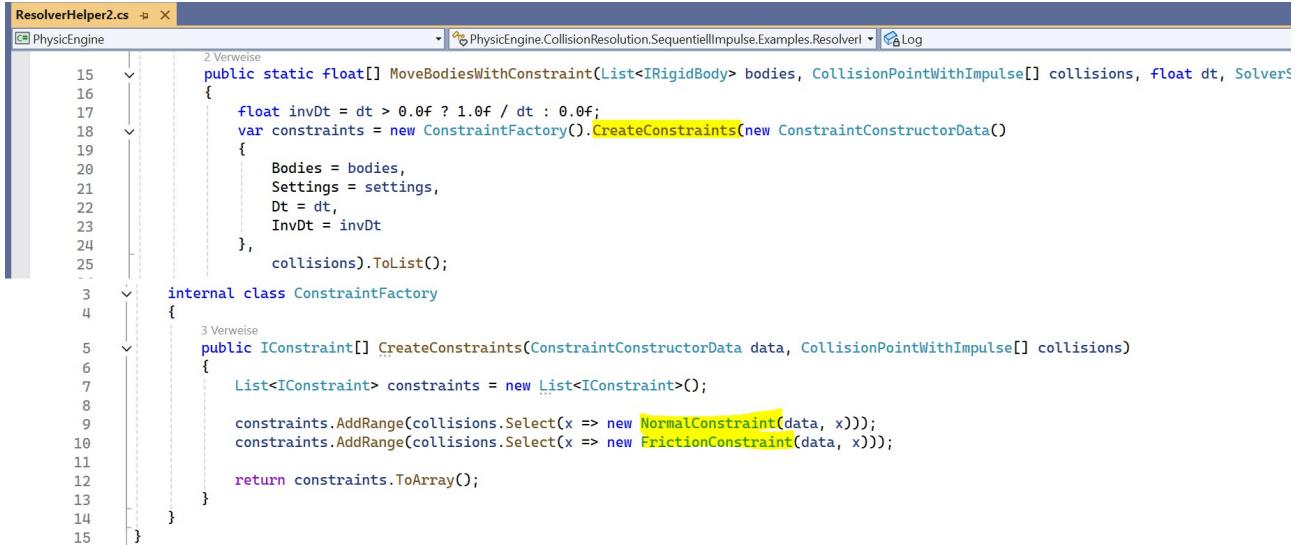
Wenn man die Summe der Impulse berechnet, dann wird bei SI sowohl für den Punkt A als auch B ein Impuls von jeweils 0.1 angewendet. Zweimal 0.1 nach links drücken bedeutet der Würfel fliegt dann mit einer Geschwindigkeit von -0.1 nach links ($0.1-0.2=-0.1$)

Bei II ist die Summe der Impulse auf A 0.16 und auf B 0.11. Da auf die obere und untere Ecke ein unterschiedlicher Impuls wirkt prallt der Würfel mit einer Drehung von der Wand ab.

Dieses Beispiel mit dem Würfel soll eine Sache erklären: Sequentielle Impulse (SI) ist fast das gleiche Verfahren wie Iterative Impulse (II) aus Teil 1. Der Unterschied ist, dass SI sich am Anfang über den Bias-Wert merkt, welche Relativgeschwindigkeit die ganzen Kontaktpunkte nach der Kollisionsauflösung haben sollen. II merkt sich den Bias-Wert nicht. Es geht mehrmals über alle Kontaktpunkte und schaut jeweils, mit welcher Geschwindigkeit die Kontaktpunkte aktuell aufeinander zu fliegen und berechnet daraus den Biaswert (Soll-Geschwindigkeitswert). Es berechnet den Bias-Wert bei jedem Impuls-Anwendungsschritt erneut was dazu führt, dass die Impulse, die bereits während der Kollisionsauflösung angewendet wurden, mit in die Berechnung der nachfolgenden Bias-Werte eingehen.

Testfall – CubeAgainstWall – Mit Reibung

Unser sequentielles Impulsverfahren konnte ohne Einsatz der FrictionConstraint den CubeAgainstWall-Testfall bestehen. Da das so gut läuft erweitern wir das Verfahren nun um die FrictionConstraint. Anstatt für jeden CollisionPoint ein NormalConstraint-Objekt zu erzeugen nutzen wir diesmal die ConstraintFactory, welche für jeden CollisionPoint sowohl ein NormalConstraint- als auch FrictionConstraint-Objekt erzeugt:

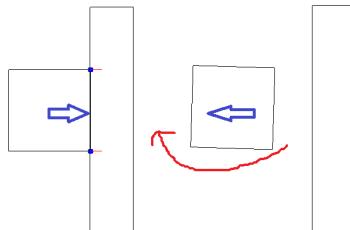


```

15     public static float[] MoveBodiesWithConstraint(List<IRigidBody> bodies, CollisionPointWithImpulse[] collisions, float dt, SolverSettings settings)
16     {
17         float invDt = dt > 0.0f ? 1.0f / dt : 0.0f;
18         var constraints = new ConstraintFactory().CreateConstraints(new ConstraintConstructorData()
19         {
20             Bodies = bodies,
21             Settings = settings,
22             Dt = dt,
23             InvDt = invDt
24         },
25         collisions).ToList();
26
27         internal class ConstraintFactory
28         {
29             3 Verweise
30             public IConstraint[] CreateConstraints(ConstraintConstructorData data, CollisionPointWithImpulse[] collisions)
31             {
32                 List<IConstraint> constraints = new List<IConstraint>();
33
34                 constraints.AddRange(collisions.Select(x => new NormalConstraint(data, x)));
35                 constraints.AddRange(collisions.Select(x => new FrictionConstraint(data, x)));
36
37                 return constraints.ToArray();
38             }
39         }
40     }
41 }

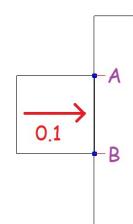
```

Wenn wir jetzt das so probieren, dann sehen wir, dass der Würfel beim Abprallen an der Wand eine Drehung bekommt:



Die Frage ist jetzt, woher kommt der Drehschwung?

Um das zu verstehen erweitere ich das Verfahren um Logging und ich berechne für jedes der 4 Constraint-Objekte außerdem die Summe der Impulse:



```

11     internal class ResolverHelper2
12     {
13         private static string Log = "";
14
15         2 Verweise
16         public static float[] MoveBodiesWithConstraint(List<IRigidBody> bodies, CollisionPointWithImpulse[] collisions, float dt, SolverSettings settings)
17         {
18             float invDt = dt > 0.0f ? 1.0f / dt : 0.0f;
19             var constraints = new ConstraintFactory().CreateConstraints(new ConstraintConstructorData()
20             {
21                 Bodies = bodies,
22                 Settings = settings,
23                 Dt = dt,
24                 InvDt = invDt
25             },
26             collisions).ToList();
27
28             float[] impulseSum = new float[constraints.Count]; //Summe der Impulse pro Constraint
29
30             for (int i = 0; i < settings.IterationCount; i++)
31             {
32                 foreach (var c in constraints)
33                 {
34                     Vector2D v1 = c.B1.Velocity + new Vector2D(-c.B1.AngularVelocity * c.R1.Y, c.B1.AngularVelocity * c.R1.X);
35                     Vector2D v2 = c.B2.Velocity + new Vector2D(-c.B2.AngularVelocity * c.R2.Y, c.B2.AngularVelocity * c.R2.X);
36                     Vector2D relativeVelocity = v2 - v1;
37                     float velocityInForceDirection = relativeVelocity * c.ForceDirection;
38                     float impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection);
39                     impulse = MathHelp.Clamp(impulse, c.MinImpulse, c.MaxImpulse);
40
41                     impulseSum[constraints.IndexOf(c)] += impulse;
42
43                     //Logging für alle 4 Constraints
44                     {
45                         string constraintName = new [] { "NormalA", "NormalB", "FrictionA", "FrictionB" }[constraints.IndexOf(c)];
46                         Log += i + "\t" + constraintName + "\t" + (impulse * 10000) + "\t" + (impulseSum[constraints.IndexOf(c)] * 10000) + "\t" +
47                     }
48
49                     ApplyImpulse(c, impulse);
50
51                 }
52             }
53
54             return impulseSum;
55         }
56     }

```

Der Würfel prallt mit einer Geschwindigkeit von 0.1 nach rechts gegen die Wand und er besitzt eine Restitution von 1. D.h. Nach den Abprallen sollen beide Kontaktpunkte mit 0.1 nach links fliegen. In Spalte E sieht man, welche Relativgeschwindigkeit die jeweiligen Kontaktpunkte haben und bei Spalte B steht, welches Constraint-Objekt das jeweils ist. Auf Zeile 11 sieht man bei der gelb markierten Zelle, dass die Geschwindigkeit von Punkt B größer als 0.1 ist.

	A	B	C	D	E	F	G
1	Iteration	Constraint	Impulse	ImpulseSum	VelocityInForceDirection	Velocity	AngularVelocity
2	0	NormalA	800,00006	800,00006	-0,1 [0,1;0]		0
3	0	NormalB	960,00006	960,00006	-0,14 [0,019999996;0]		-0,0017142858
4	0	FrictionA	-75	-75	0,024000004 [-0,07600001;0]		0,0003428572
5	0	FrictionB	-21,000017	-21,000017	0,005250004 [-0,07600001;-0,007500003]		0,0001821429
6	1	NormalA	134,39995	934,4	0,066400014 [-0,07600001;-0,009600002]		0,0001371429
7	1	NormalB	84,47993	1044,48	0,07888002 [-0,08944001;-0,009600002]		-0,000150857
8	1	FrictionA	29,952017	-45,04799	-0,007488004 [-0,097888;-0,009600002]		3,0171403E-05
9	1	FrictionB	1,8626451E-06	-21,000015	-4,656613E-10 [-0,097888;-0,006604801]		9,435429E-05
10	2	NormalA	34,867195	969,2673	0,0912832 [-0,097888;-0,006604801]		9,43543E-05
11	2	NormalB	0	1044,48	0,102749445 [-0,10137472;-0,006604801]		1,963888E-05
12	2	FrictionA	20,920317	-24,127672	-0,0052300794 [-0,10137472;-0,006604801]		1,963888E-05
13	2	FrictionB	0	-21,000015	-0 [-0,10137472;-0,0045127691]		6,446813E-05

D.h. Punkt B fliegt zu schnell nach links. Eigentlich wäre jetzt ein Impuls nötig, welcher Punkt B wieder etwas zur wand ran zieht. D.h. Es ist ein negativer Impuls nötig. Da aber wegen unseres Clamping der Impuls immer nur positiv sein darf, ist eine Korrektur jetzt nicht mehr möglich.

Die Lösung ist, das man nicht die Zwischenimpulse clampt sondern für die Summe der Impulse pro Constraint muss das Clamping erfolgen. Um diese Aufgabe zu bewerkstelligen wird die Constraint-Klasse jetzt um eine weitere Property erweitert, welche die Summe der Impulse speichert:

```
6   internal class NormalConstraint : IConstraint
7   {
8       14 Verweise
9       public float AccumulatedImpulse { get; set; }
```

Für diese Summe erfolgt das Clamping nun so:

```
58     // Clamp the accumulated impulse
59     float oldSum = c.AccumulatedImpulse;
60     c.AccumulatedImpulse = MathHelp.Clamp(oldSum + impulse, c.MinImpulse, c.MaxImpulse);
61     impulse = c.AccumulatedImpulse - oldSum;
```

Der tatsächlich angewendete Zwischenimpuls ist die Differenz zwischen der aktuellen Summe und den letzten Summenwert.

Accumulated Impulse

Die Frage ist jetzt, warum konnte beim vorherigen Beispiel mit falschen Clamping aber auch ohne FrictionConstraints der Würfel rotationsfrei abprallen? Um besser zu verstehen, wieso das Clamping bei Verwendung der FrictionConstraint mit der AccumulatedImpulse-Variable erfolgen muss und warum man nicht einfach so arbeiten kann wie hier,

```
23     float impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection);
24     impulse = MathHelp.Clamp(impulse, c.MinImpulse, c.MaxImpulse);
```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Examples/ResolverHelper1.cs

wollen wir die beiden Arten des Impulse-Clampings weiter untersuchen und nebeneinander stellen.

Das Impulscamping an sich gab es auch schon beim Matrix-Ansatz in Teil 2. Man will durch das Clamping dafür sorgen, dass z.B. eine NormalConstraint zwei Körper immer nur auseinander drückt aber sie nie zusammen zieht. Oder man will die Reibungskraft laut eines Reibungskoeffizienten eingrenzen.

Die AccumulatedImpulse-Variable ist die Summe aller Zwischenimpulse für einen Kontaktspunkt. Sie entspricht dem Impuls, welcher das PGS-Verfahren beim Matrix-Ansatz aus Teil 2 berechnen würde, um die globale Lösung für eine Menge von Kontaktspunkten zu finden. Nur für diesen global berechneten Impuls soll das Clamping erfolgen.

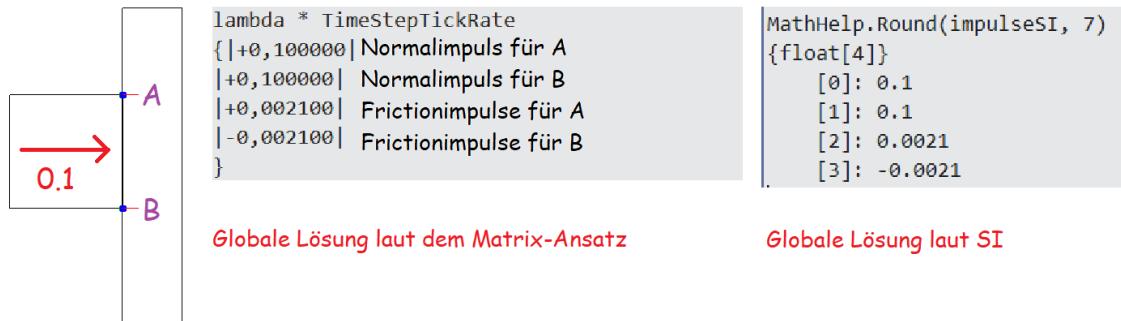
Was ist die globale Lösung überhaupt?

Die globale Lösung ist die Menge aller Impulse, welche angewendet werden muss, um alle Kontaktpunkte gleichzeitig zu lösen. Beispiel: Gegeben ist ein Würfel, der gerade nach rechts fliegt und die Kollisionserkennung hat zwei Kontaktpunkte A und B ermittelt. Damit der Würfel an der Wand abprallt wird sowohl für A als auch B jeweils ein Normal- und Frictionimpuls angewendet. Um nun die globale Lösung zu ermitteln kann ich mit der Matrix-Lösung für jede Constraint das Lambda berechnen. Lambda ist ein Spaltenvektor, welcher die Constraintkraftlänge angibt. Kraft mal Zeitspanne ergibt ein Impuls. Die globale Lösung von der Matrix bekomme ich also über $\text{lambda} * \text{TimeStepTickRate}$. Beim Sequentiellen Impuls-Algorithmus bekomme ich die globale Lösung, indem ich für jede Constraint die Summe der angewendeten Zwischenimpulse ermitte.

```

87 var scene = new PhysicScene(bodys);
88 scene.Solver = solver;
89 scene.DoPositionalCorrection = scene.DoWarmStart = scene.HasGravity = false;
90 scene.IterationCount = 100;
91
92 var lambda = scene.GetProjectedGaussSeidelSteps(TimeStepTickRate, scene.IterationCount).Last();
93 float[] impulseMatrix = (lambda * TimeStepTickRate).GetColumn(0); //Sollwert
94
95 scene.TimeStep(TimeStepTickRate);
96
97 float[] impulseSI = scene.GetLastAppliedImpulsePerConstraint();

```



Siehe: PhysicEngine.UnitTests/CollisionResolution/ExplainSequentiellImpulseTests.cs

So wird die globale Lösung beim Sequentiellen Impuls-Verfahren berechnet (gelb markiert):

```

ResolverHelper2.cs ✎ ×
[PhysicEngine] ⌂ PhysicEngine.CollisionResolution.SequentiellImpulse.Examples.Resolve ⌂ MoveBodiesWithConstraint(List<IRigidBody> bodies, CollisionPointWithImpulse[] collisions, float dt)
20 public static float[] MoveBodiesWithConstraint(List<IRigidBody> bodies, CollisionPointWithImpulse[] collisions, float dt)
21 {
22     float invDt = dt > 0.0f ? 1.0f / dt : 0.0f;
23     var constraints = new ConstraintFactory().CreateConstraints(new ConstraintConstructorData()..., collisions).ToList();
24
25     float[] impulseSum = new float[constraints.Count]; //Summe der Impulse pro Constraint
26
27     for (int i = 0; i < settings.IterationCount; i++)
28     {
29         foreach (var c in constraints)
30         {
31             Vector2D v1 = c.B1.Velocity + new Vector2D(-c.B1.AngularVelocity * c.R1.Y, c.B1.AngularVelocity * c.R1.X);
32             Vector2D v2 = c.B2.Velocity + new Vector2D(-c.B2.AngularVelocity * c.R2.Y, c.B2.AngularVelocity * c.R2.X);
33             Vector2D relativeVelocity = v2 - v1;
34             float velocityInForceDirection = relativeVelocity * c.ForceDirection;
35             float impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection);
36
37             if (useAccumulatedImpulse)
38             {
39                 // Clamp the accumulated impulse
40                 float oldSum = c.AccumulatedImpulse;
41                 c.AccumulatedImpulse = MathHelp.Clamp(oldSum + impulse, c.MinImpulse, c.MaxImpulse);
42                 impulse = c.AccumulatedImpulse - oldSum;
43             }
44             else
45             {
46                 impulse = MathHelp.Clamp(impulse, c.MinImpulse, c.MaxImpulse);
47             }
48
49             impulseSum[constraints.IndexOf(c)] += impulse;
50             ApplyImpulse(c, impulse);
51         }
52     }
53 }
54
55 }
```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Examples/ResolverHelper2.cs

Die globale Lösung ist also ein Vektor, welcher für alle Constraints den nötigen Impuls angibt.

Vergleich – Kollisionsauflösung mal mit und mal ohne AccumulatedImpulse

Nachdem nun geklärt wurde, was die globale Lösung überhaupt ist kommen wir nun darauf zurück, warum die AccumulatedImpulse-Variable nötig ist.

Hat man ein Würfel, der auf ein Tisch steht und eine Normalconstraint drückt über ein Impuls eine Ecke nach oben, dann darf ein Zwischenimpuls auch mal ziehend wirken, um den zu hoch gedrückten Würfel wieder nach unten zu ziehen. Wenn ich das Impulscamping für die Zwischenimpulse anwenden würde, dann könnte ein einmal zu stark wirkender Normalimpuls nicht mehr korrigiert werden. Deswegen darf das Impulscamping nur für die AccumulatedImpulse-Variable getan werden (Zeile 48 im Bild vorherige Seite) aber nicht für die Zwischenimpulse (Zeile 52).

Um besser zu verstehen, was passiert, wenn man einmal das Impulscamping mit AccumulatedImpulse macht und einmal direkt mit dem Zwischenimpuls habe ich die beiden gelb markierten UnitTests erstellt. Beide nutzen den ResolverHelper2 aber mit unterschiedlichen useAccumulatedImpulse-Schalter (Zeile 44 im Bild oben).

```
PhysicEngine.UnitTests (56)
  PhysicEngine.UnitTests (1)
  PhysicEngine.UnitTests.CollisionDetection (18)
  PhysicEngine.UnitTests.CollisionResolution (37)
    Box2DLightTest (1)
    ExplainSequentiellImpulseTests (3)
      Example1_DoltLikeRelativImpulse
      Example2_ExpandWithFriction
      Example3_ExpandWithAccumulatedImpulse

  impulse = MathHelp.Clamp(impulse, c.MinImpulse, c.MaxImpulse);

  float oldSum = c.AccumulatedImpulse;
  c.AccumulatedImpulse = MathHelp.Clamp(oldSum + impulse, c.MinImpulse, c.MaxImpulse);
  impulse = c.AccumulatedImpulse - oldSum;
```

Dort wird getestet, ob der Würfel an der Wand rotationsfrei abprallt und ob die globale Lösung vom SI-Verfahren mit der globalen Matrixlösung übereinstimmt. Der SI-Solver von Example2 macht dabei das gleiche, wie der aus Example1 nur dass es jetzt noch Friction-Constraints gibt. Es gibt pro Kontaktspur also eine Normal- und eine Frictionconstraint und das Clamping erfolgt laut Zeile 52 (siehe im Bild bei ResolverHelper2 oben). Der Würfel prallt mit Rotation ab, da die beiden Frictionimpulse unterschiedliche Werte ermitteln. So sieht die globale Lösung bei Example2 mit falschen Clamping aus:

Sollwert	Istwert
result.ImpulseMatrix[0]: 0.1	result.ImpulsesSI[0]: 0.098888
result.ImpulseMatrix[1]: 0.1	result.ImpulsesSI[1]: 0.104448
result.ImpulseMatrix[2]: 0.0021	result.ImpulsesSI[2]: -0.001236
result.ImpulseMatrix[3]: -0.0021	result.ImpulsesSI[3]: -0.0021

```
ExplainSequentiellImpulseTests.cs  ResolverHelper2.cs
PhysicEngine.UnitTests
  ExplainSequentiellImpulseTests
    Example2_ExpandWithFriction()

35
36  //Das hier macht das gleiche wie SI_Easy1 nur dass zusätzlich noch die FrictionConstraint genutzt wird.
37  //Hier sieht man, dass der Würfel nun wegen falschen Impulscamping falsche Impulse berechnet so dass er
38  //dann mit Drehung abprallt
39  [Fact]
40  public void Example2_ExpandWithFriction()
41  {
42    var result = DoTest(PhysicScene.SolverType.SI_Easy2);
43
44    //Prüfe dass der Würfel rotationsfrei an der Wand abprallt
45    result.CubeVelocity.Should().Be(-result.CubeVelocityBefore);
46    result.CubeAngularVelocity.Should().Be(0);
47
48    result.ImpulseSI[0].Should().Be(result.ImpulseMatrix[0]); //Normal-Impuls für den oberen Kontaktspur
49    result.ImpulseSI[1].Should().Be(result.ImpulseMatrix[1]); //Normal-Impuls für den unteren Kontaktspur
50    result.ImpulseSI[2].Should().Be(result.ImpulseMatrix[2]); //Friction-Impuls für den oberen Kontaktspur
51    result.ImpulseSI[3].Should().Be(result.ImpulseMatrix[3]); //Friction-Impuls für den unteren Kontaktspur
52  }
```

Siehe: PhysicEngine.UnitTests/CollisionResolution/ExplainSequentiellImpulseTests

Warum ist der Frictionimpuls für den Punkt A zu niedrig?

Man sieht im Bild, dass sowohl die Normalimpulse nicht ganz 0.1 ergeben und auch der Frictionimpulse für den Punkt A nicht mit dem Sollwert übereinstimmt (rot Markiert = falscher Wert für FrictionImpuls). Die Frage ist nun, warum kommen dort falsche Werte raus? Um das zu verstehen schauen wir uns zuerst die Zwischenimpulse von der Frictionconstraint am Punkt A näher an.

ResolverHelper2 wird um Logging erweitert, um die Zwischenimpulse für die Frictionconstraint von Punkt A zu sehen:

```

ResolverHelper2.cs
[PhysicalEngine]
[PhysicalEngine.CollisionResolution.SequentialImpulse.Examples.Resolve]
[Log]

20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
}
}

public static float[] MoveBodiesWithConstraint(List<IRigidBody> bodies, CollisionPointWithImpulse[] collisions, float dt,
{
    float invDt = dt > 0.0f ? 1.0f / dt : 0.0f;
    var constraints = new ConstraintFactory().CreateConstraints(new ConstraintConstructorData()...),
        collisions).ToList();

    float[] impulseSum = new float[constraints.Count]; //Summe der Impulse pro Constraint

    for (int i = 0; i < settings.IterationCount; i++)
    {
        foreach (var c in constraints)
        {
            Vector2D v1 = c.B1.Velocity + new Vector2D(-c.B1.AngularVelocity * c.R1.Y, c.B1.AngularVelocity * c.R1.X);
            Vector2D v2 = c.B2.Velocity + new Vector2D(-c.B2.AngularVelocity * c.R2.Y, c.B2.AngularVelocity * c.R2.X);
            Vector2D relativeVelocity = v2 - v1;
            float velocityInForceDirection = relativeVelocity * c.ForceDirection;
            float impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection);

            if (useAccumulatedImpulse)
            {
                // Clamp the accumulated impulse
                float oldSum = c.AccumulatedImpulse;
                c.AccumulatedImpulse = MathHelp.Clamp(oldSum + impulse, c.MinImpulse, c.MaxImpulse);
                impulse = c.AccumulatedImpulse - oldSum;
            }
            else
            {
                impulse = MathHelp.Clamp(impulse, c.MinImpulse, c.MaxImpulse);
            }

            impulseSum[constraints.IndexOf(c)] += impulse;
            ApplyImpulse(c, impulse);

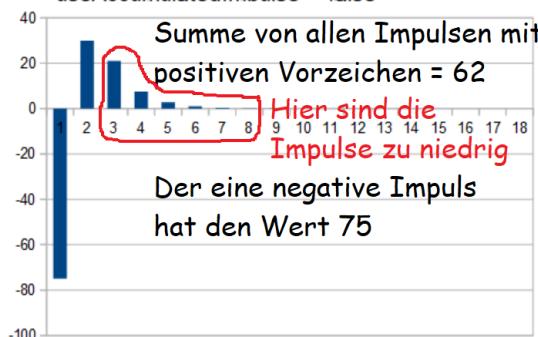
            if (constraints.IndexOf(c) == 2) //Frictionimpuls für den oberen Kollisionspunkt
            {
                float sum = impulseSum[constraints.IndexOf(c)];
                Log += i + "\t" + (impulse * 10000) + "\t" + (sum * 10000) + "\r\n";
            }
        }
    }
}
}

```

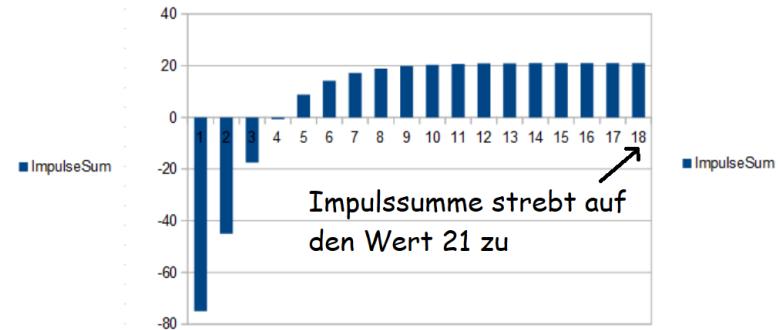
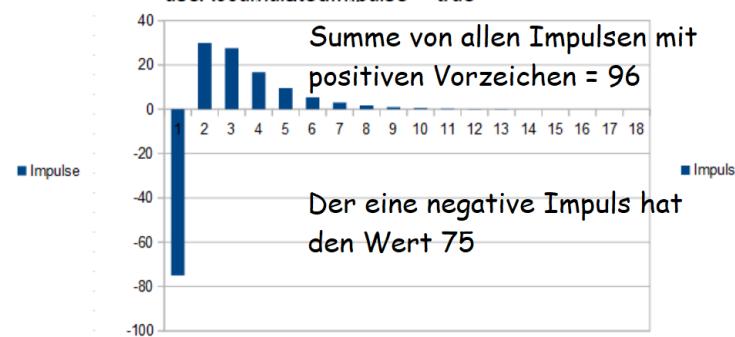
Ich multipliziere hier die Impulse noch mit 10000 da die Zahlen so im Text hier kürzer sind.

Ich vergleiche nun die Zwischenimpulse wo ich einmal mit falschen Clamping arbeite (useAccumulatedImpulse ist false) und einmal, wo ich mit richtigen Clamping arbeite (useAccumulatedImpulse ist true).

useAccumulatedImpulse = false



useAccumulatedImpulse = true



Der Frictionimpuls wird auf den Bereich von -75 bis +75 geclampt. Die Summe aller Frictionimpulse für Punkt A muss 21 ergeben da die Matrix-Lösung das so vorgibt. Im unteren rechten Bild sieht man auch, dass das Verfahren mit korrekten Clamping diesen Wert ermittelt.

Beim unteren linken Bild sieht man, dass beim falschen Clamping die Summe -12 ergibt. Will man verstehen warum dort ein falscher Wert raus kommt muss man die oberen beiden Bilder vergleichen. Man sieht, dass die ersten beiden Zwischenimpulse bei beiden noch gleich sind. Bei den nachfolgenden Zwischenimpulsen fehlt jeweils beim linken oberen Bild immer etwas (rot markiert) wodurch die Summe dann zu klein wird.

Warum ist der FrictionImpuls ab der dritten Iteration zu niedrig?

Um zu verstehen warum der Friction-Impuls für Punkt A ab der dritten Iteration zu klein ist erweitern wir das Logging nun so, dass wir für alle vier Constraints sehen, welcher Impuls jeweils angewendet wurde.

```

ResolverHelper2.cs  ×
PhysicsEngine
PhysicsEngine.CollisionResolution.SequentiellImpulse.Examples.ResolverHelp - ApplyImpulse(Constraint c, float impulse)

40 Vector2D relativeVelocity = v2 - v1;
41 float velocityInForceDirection = relativeVelocity * c.ForceDirection;
42 float impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection);
43
44 if (useAccumulatedImpulse)
45 {
46     // Clamp the accumulated impulse
47     float oldSum = c.AccumulatedImpulse;
48     c.AccumulatedImpulse = MathHelp.Clamp(oldSum + impulse, c.MinImpulse, c.MaxImpulse);
49     impulse = c.AccumulatedImpulse - oldSum;
50 }
51 else
52 {
53     impulse = MathHelp.Clamp(impulse, c.MinImpulse, c.MaxImpulse);
54 }
55
56 impulseSum[constraints.IndexOf(c)] += impulse;
57
58 //Logging für FrictionA
59 //if (constraints.IndexOf(c) == 2) //Frictionimpuls für den oberen Kollisionspunkt
60 //{
61 //    float sum = impulseSum[constraints.IndexOf(c)];
62 //    Log += i + "\t" + (unclampedImpulse * 10000) + "\t" + (impulse * 10000) + "\t" + (sum * 10000) + "\r\n";
63 //}
64
65 //Logging für alle 4 Constraints
66 string constraintName = new [] { "NormalA", "NormalB", "FrictionA", "FrictionB" }[constraints.IndexOf(c)];
67 Log += i + "\t" + constraintName + "\t" + (impulse * 10000) + "\t" + (c.AccumulatedImpulse * 10000) + "\t" + velocityInForceDi
68
69
70 ApplyImpulse(c, impulse);

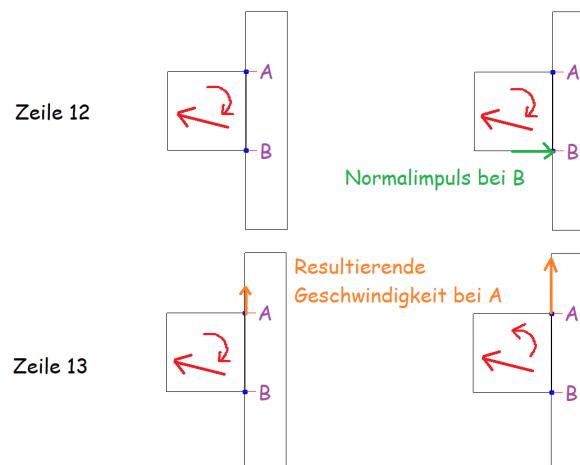
```

Dort sehen wir, dass der erste falsch angewendete Impuls der NormalImpuls für den Punkt B in der Iteration 2 ist (Zeile 12 Spalte C; gelb markierte Zelle). Beim korrekt verwendeten Clamping wird ein Normalimpuls von -10.99 angewendet und beim falschen Clamping ist der Impuls 0.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	useAccumulatedImpulse = false														
2	Iteration	Constraint	Impulse	Acc*VelocityInForceDirection	Velocity		AngularVelocity		useAccumulatedImpulse = true						
3	0	NormalA	800.00006	0	-0.1 [0 1.0]		0		0	NormalA	800.00006	800.00006	-0.1 [0 1.0]		0
4	0	NormalB	960.00006	0	-0.14 [0 0.19999996]		-0.0017142958		0	NormalB	960.00006	960.00006	-0.14 [0.019999996]		-0.0017142958
5	0	FrictionA	-75	0	0.02400004	-0.07600001	0.0003428572		0	FrictionA	-75	-75	0.02400004	-0.07600001	0.0003428572
6	0	FrictionB	-21.000017	0	0.00525004	-0.07600001	0.0001821429		0	FrictionB	-21.000017	-21.000017	0.00525004	-0.07600001	-0.007500003
7	1	NormalA	134.39995	0	0.066400014	-0.07600001	-0.099600002		0	NormalA	134.39998	934.4	0.066400014	-0.07600001	-0.099600002
8	1	NormalB	84.47993	0	0.07888002	-0.08944001	-0.099600002		0	NormalB	84.47997	1044.4801	0.07888001	-0.08944001	-0.099600002
9	1	FrictionA	29.952017	0	-0.007488004	-0.097888002	-0.099600002	3.0171403E-05	0	FrictionA	29.95201	-45.047993	-0.007488002	-0.09788801	-0.099600002
10	1	FrictionB	1.8626451E-10	0	-0.4656613E-10	-0.097888002	-0.099600002	9.435429E-05	0	FrictionB	0	-21.000017	-0.09788801	-0.099600002	9.435429E-05
11	2	NormalA	34.867195	0	0.0912832	-0.097888001	-0.066604801	9.43543E-05	0	NormalA	34.867153	969.2672	0.0912832	-0.09788801	-0.066604801
12	2	NormalB	0	0	0.102749445	-0.10137472	-0.066604801	1.963888E-05	0	NormalB	-10.997802	103.4823	0.10274945	-0.10137472	-0.066604801
13	2	FrictionA	20.920317	0	-0.0052300794	0.10137472	-0.066604801	1.963888E-05	0	FrictionA	27.518972	-17.52902	-0.006879743	0.1027494	-0.066604801

Grafisch kann man sich Tabellenzeile 12 und 13 so vorstellen:

useAccumulatedImpulse=false useAccumulatedImpulse=true



Bei Zeile 12 bewegt sich der Würfel nach links oben und hat eine Rechtsdrehung (rote Pfeile). Punkt B bewegt sich mit einer Geschwindigkeit von 0.1027 nach links. Seine Zielgeschwindigkeit ist aber 0.1. Weil innerhalb der vorherigen Impulse Punkt B etwas zu sehr nach links gedrückt wurde ist es nötig, dass Punkt B ein Impuls nach rechts braucht. Das Verfahren mit korrekten Clamping (rechte Bildseite) führt diesen Impuls am Punkt B nach rechts auch aus (grüner Pfeil) wodurch der Würfel sich dann bei Zeile 13 nach links dreht. Das Verfahren mit falschen Clamping (linke Bildseite) tätigt diesen Impuls nicht wodurch der Würfel sich weiterhin nach rechts dreht.

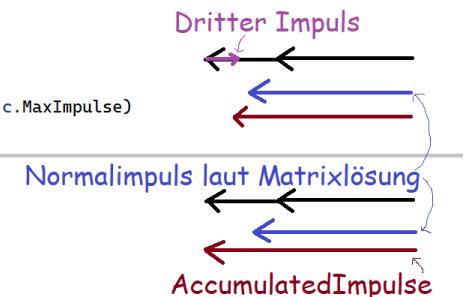
Bei Zeile 13 soll nun bei Punkt A ein Frictionimpuls angewendet werden. Dazu wird ermittelt, mit welcher Geschwindigkeit der Punkt A sich nach oben bewegt. Wenn der Würfel sich einerseits nach oben bewegt und anderseits eine Linksdrehung hat, dann bewegt der Punkt A sich schneller nach oben als wenn er sich zwar nach oben bewegt aber dabei eine Rechtsdrehung hat. Wegen dieser kleineren Geschwindigkeit von A nach oben wird auch ein kleinerer Frictionimpuls auf A angewendet. Das also ist die Antwort, warum der Frictionimpuls bei A in der dritten Iteration zu klein ist.

Normalimpulse als Pfeile im Vergleich

```

44     if (useAccumulatedImpulse)
45     {
46         // Clamp the accumulated impulse
47         float oldSum = c.AccumulatedImpulse;
48         c.AccumulatedImpulse = MathHelp.Clamp(oldSum + impulse, c.MinImpulse, c.MaxImpulse);
49         impulse = c.AccumulatedImpulse - oldSum;
50     }
51     else
52     {
53         impulse = MathHelp.Clamp(impulse, c.MinImpulse, c.MaxImpulse);
54     }

```



Wegen Clamping gibts hier kein dritter Impuls

Man kann sich die Normalimpulse auf einen der beiden Würfelpunkte auch als Vektoraddition vorstellen. Die Matrix-Lösung hat errechnet, dass auf beide Punkte ein Normalimpuls von 0.1 wirken muss damit der Würfel korrekt an der Wand abprallt. Dieser Impuls zeigt nach links und wird hier durch die blauen Pfeile visualisiert. Die ersten beiden Normalimpulse auf den Punkt B haben nach links gezeigt und werden hier mit schwarzen Pfeilen gekennzeichnet. Bei beiden Verfahren sind diese Impulse gleich groß. In der Iteration 3 wird nun festgestellt, dass der Würfel zu schnell nach links fliegt und deswegen ein Impuls nach rechts nötig ist. Beim oberen Verfahren wird deswegen der Lila-Impuls, der nach rechts zeigt, angewendet. Somit entspricht der resultierende Gesamtimpuls=AccumulatedImpulse (rot-braun) schon eher dem blauen Pfeil. Beim unteren Verfahren ist diese Korrektur wegen dem falschen Clamping nicht möglich. Wurde einmal ein zu großer Normalimpuls angewendet, dann kann dieser Fehler nicht mehr korrigiert werden und der Gesamtimpuls bleibt zu groß.

Das Clamping auf die AccumulatedImpuls-Variable bedeutet also, dass der resultierende Gesamtimpuls für eine Normalconstraint zwei Körper immer nur auseinander drücken darf aber nie ziehen. Clampt man aber die Zwischenimpulse, dann versperrt man sich selber den Weg um eine Korrektur vorzunehmen. D.h. Der rot-braune Pfeil darf immer nur nach links zeigen aber die Zwischenimpulse (schwarz oder lila) dürfen auch nach rechts zeigen.

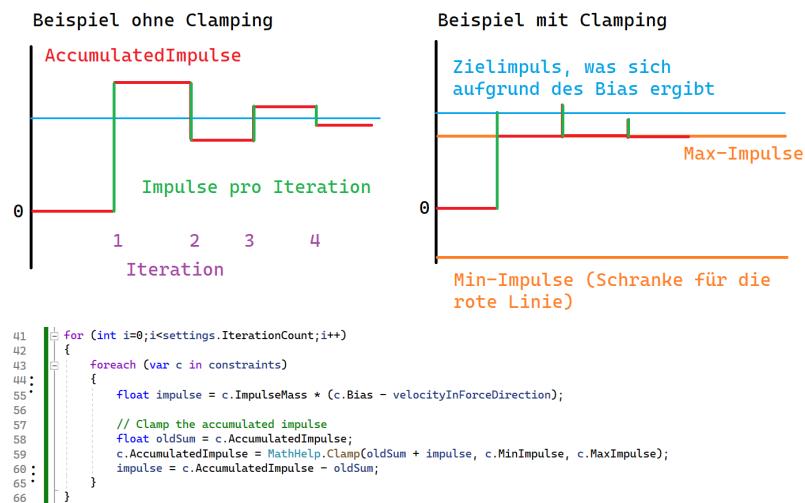
Wenn man das falsche Clamping nutzt aber keine FrictionConstraints, dann wird der Würfel korrekt an der Wand abgeprallt (entspricht ResolverHelper1). Der Grund, warum also die eine Würfecke in der dritten Iteration zu schnell nach links fliegt liegt daran, dass einerseits die Frictionconstraints den Würfel drehen lassen, was dazu führt, dass die eine Ecke schneller von der Wand weg fliegt und anderseits drücken ja auch noch die Normalimpulse am Anfang nach links. Friction-Drehung + Normalimpuls hat die eine Würfecke in der dritten Iteration dann schneller als 0.1 nach links fliegen lassen.

Zusammenfassend kann man also sagen, dass die AccumulatedImpuls-Variable die Summe aller

angewendeten Impulse von ein Constraint-Objekt ist und dass nur für die Summe das Clamping erfolgen darf. Clampst man die Zwischenimpulse, dann kann ein Kontaktpunkt, der zu schnell von einem anderen Kontaktpunkt weg fliegt dann nicht mehr korrigiert werden. Das Würfelbeispiel hat nur erst den Anschein gemacht, dass falsches Clamping lediglich Einfluss auf den Frictionimpuls hat aber in Wirklichkeit haben die Normal- und Frictionimpulse sich gegenseitig beeinflusst und der eigentliche Fehler begann schon ab den Zeitpunkt, wo der Normalimpuls auf den Punkt B nicht mehr nach rechts wirken konnte.

AccumulatedImpulse mal mit und ohne Min-Max-Schranke

Wir haben jetzt geklärt, warum wir nicht direkt die Zwischenimpulse clampen dürfen sondern nur die Summe. Hier ist noch ein Bild, wie man sich das Clamping auf die AccumulatedImpulse-Variable besser vorstellen kann. Beim linken Bild ist MinImpulse=-Unendlich und MaxImpulse=+Unendlich (Bsp: Distance-Constraint). Beim rechten Bild haben die Min-Max-Werte ein Wert in der Nähe von Null (orangene Linien) (Bsp: FrictionConstraint).



Ein Impuls ist eine grüne vertikale Linie, welche die AccumulatedImpulse-Variable (rote horizontale Linie) entweder nach oben oder nach unten verschiebt. Für diese rote Linie gilt die Regel, dass sie sich wegen dem Clamping nur innerhalb der beiden orangenen Linien aufhalten darf. Für die grünen Linien gilt keine Regel, wie lang sie maximal sein dürfen.

Wenn man kein Clamping hat (linkes Bild), dann sieht man, dass die rote AccumulatedImpulse-Variable gegen die blaue Zielimpulse-Linie strebt. Hat man aber Clamping (rechts Bild), dann kann die rote Linie zwar versuchen über die MaxImpulse-Schranke zu kommen aber sie bleibt dort hängen. Das ist so wie wenn man ein Würfel hat, der über ein Tisch rutscht. Die FrictionConstraint von den Würfelkontaktpunkten mit dem Tisch hat ein Bias, wo sie sagt, dass die Zielgeschwindigkeit des Würfels Null sein soll. Sie kann aber innerhalb von ein TimeStep den Würfel nicht auf Null bringen, weil die Reibungskraft begrenzt ist. Der Impuls würde hier also an der MinMax-Impulse-Schranke hängen bleiben und den Würfel zwar bremsen aber es braucht mehrere TimeSteps, bis der Würfel dann ruhig liegen bleibt.

Erweiterung des SI-Solvers um externe Kraft

Um das Sequentielle Impuls-Verfahren besser zu verstehen hatten wir zuerst zwei vereinfachte Versionen davon erstellt (siehe ResolverHelper1, ResolverHelper2). Wir erweitern nun ResolverHelper2 um die Anwendung der externen Kraft (Schwerkraft). Die Funktion, welche die Schwerkraft anwendet sieht so aus:

```
66     private static void ApplyExternalForce(List<IRigidBody> bodies, float dt)
67     {
68         //Wende die externe Kraft für alle Körper an
69         foreach (var body in bodies)
70         {
71             body.Velocity.X += body.InverseMass * body.Force.X * dt;
72             body.Velocity.Y += body.InverseMass * body.Force.Y * dt;
73             body.AngularVelocity += body.InverseInertia * body.Torque * dt;
74         }
75     }
```

Siehe: [PhysicEngine/CollisionResolution/SequentiellImpulse/Examples/ResolverHelper3.cs](#)

So lange es keine Kollisionspunkte gibt, so lange gibt es aktuell auch keine Constraint-Objekte. Da der Matrix-Solver aber nur dann eine externe Kraft anwendet, wenn es auch ein Constraint-Objekt (Kontaktpunkt) gibt, wurde die TimeStep-Methode in der PhysicScene-Klasse so geschrieben, dass sie direkt die ApplyExternalForces-Methode aufruft, wenn es keine Kontaktpunkte gibt:

```

102     public void TimeStep(float dt)
103     {
104         //1. Weise der externen Kraft einen Wert zu (Der Nutzer darf vor jedem TimeStep-Aufruf auch selber Kraftwerte setzen)
105         if (this.HasGravity)
106             AddGravityForceForAllBodies();
107
108         //2. Ermitte alle Kontaktspunkte
109         var collisionsFromThisTimeStep = CollisionHelper.GetAllCollisions(this.bodies);
110         if (collisionsFromThisTimeStep.Any())
111             this.CollisionOccured?.Invoke(this, collisionsFromThisTimeStep);
112
113         //3. Constraint-Kraft aufgrund der aktuellen Geschwindigkeit berechnen und zusammen mit der externen Kraft anwenden
114         if (collisionsFromThisTimeStep.Any()) //Abfrage: Gibt es Constraints?
115             this.ImpulseResolver.Resolve(this.bodies, collisionsFromThisTimeStep, dt, this.settings); //Wende Constraint-Kraft an
116         else
117             ApplyExternalForces(dt); //Körper ohne Beschränkung bewegen (Wende nur die externe Kraft an)
118
119         //4. Geschwindigkeit verändert die Position
120         MoveBodiesAndSetForceToZero(dt);
121     }

```

Siehe: PhysicEngine/PhysicScene.cs

Es geht jetzt also darum den Fall zu überlegen was passiert, wenn es Kontaktspunkte gibt. Wie muss dann der ResolverHelper aussehen, damit er gleichzeitig die externe Kraft als auch die Constraint-Kraft anwendet? Es gibt drei Möglichkeiten wo man den Aufruf von ApplyExternalForce einfügen könnte:

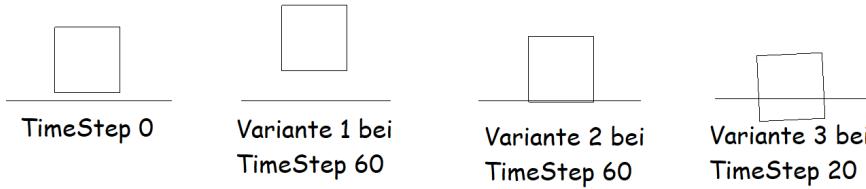
```

19     public static void MoveBodiesWithConstraint(List<IRigidBody> bodies, CollisionPointWithImpulse[] collisions,
20                                                 float dt, SolverSettings settings, Variation variation)
21     {
22         //Möglichkeit 1 -> Geht nicht
23         if (variation == Variation.Variation1) ApplyExternalForce(bodies, dt);
24
25         //Erzeuge ForceDirection, ImpulseMass und Bias für jedes Constraint-Objekt
26         float invDt = dt > 0.0f ? 1.0f / dt : 0.0f;
27         var constraints = new ConstraintFactory().CreateConstraints(new ConstraintConstructorData[]...
28             , collisions);
29
30         //Möglichkeit 2: -> Geht
31         if (variation == Variation.Variation2) ApplyExternalForce(bodies, dt);
32
33         //Finde per PGS Relative-Kontaktpunktgeschwindigkeitswerte, welche dem Bias-Wert entsprechen
34         for (int i = 0; i < settings.IterationCount; i++)...
35
36         //Möglichkeit 3: -> Geht nicht
37         if (variation == Variation.Variation3) ApplyExternalForce(bodies, dt);
38
39     }

```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Examples/ResolverHelper3.cs

Wenn ich nun ein Würfel mit Restitution von 0.9 und ohne PositionCorrection aus geringer Höhe auf ein Tisch fallen lasse, dann würde ich erwarten, dass er erst ein paar mal auf dem Tisch abprallt und mit jedem Sprung aber weniger hoch kommt bis er zur Ruhe kommt. Wenn ich alle drei Möglichkeiten der ApplyExternalForce probiere, dann sehe ich, dass nur Variante 2 funktioniert.



Bei Variante 1 springt der Würfel immer höher und bei Variante 3 sinkt er in den Tisch ein und fällt hindurch. Der Unterschied zwischen Variante 1 und 2 ist, dass bei Variante 1 die externe Kraft die Geschwindigkeit des Würfels ändert und somit auch der Bias-Wert bei Zeile 27 einen anderen Wert bekommt, als wenn man die externe Kraft erst nach der Erstellung der Constraint-Objekte anwendet (Variante 2 und 3). Es wird also nicht nur die aktuelle Würfelgeschwindigkeit zum Zeitpunkt der Aufschlags reflektiert sondern die aktuelle Geschwindigkeit Plus das Geschwindigkeitsdelta, was der Würfel innerhalb von ein TimeStep wegen der Schwerkraft bekommt. Somit bekommt der Würfel bei Variante 1 bei jeden Aufschlag immer noch etwas extra Energie.

Bei Variante 3 wird beim Aufschlag 90% der Energie reflektiert aber danach kommt dann nochmal die Schwerkraft und beschleunigt ihn nach unten. Wenn der Würfel das erste Mal aufschlägt hat er noch genug Energie um etwas nach oben zu fliegen aber schon beim zweiten Aufschlag reicht der Schwung nicht mehr aus und er fällt durch den Tisch hindurch. Hier wurde zwar ein korrekter Bias-Wert berechnet aber die Schwerkraft wurde nicht innerhalb der PGS-Schleife auf Zeile 40 beachtet so dass die Constraint-Kraft zu klein ist.

Aus dem Grund muss die externe Kraft nach der Constraint-Erstellung und vor der PGS-Schleife erfolgen (Variante 2).

Erweiterung des SI-Solvers um WarmStart

Wenn man ein Würfelstapel hat, dann hat man viele Resting-Contact-Punkte, wo Schwerkraft, Normalconstraintkraft und Frictionkraft im gegenseitigen Gleichgewicht sind. Damit der Stapel ruhig steht ist es wichtig, dass die errechneten Normal- und Friction-Impulse genau berechnet werden. Beim WarmStart merkt man sich, welche Impulse man pro Kontaktspunkt beim letzten TimeStep angewendet hat und man wendet diese beim aktuellen TimeStep wieder an. Der Vorteil vom Warmstart-Einsatz ist, dass man viel weniger Rechenschritte benötigt, um den Würfelstapel ruhig stehen zu lassen.

Damit wir wissen, welchen Normal- und Frictionimpuls wir pro Kontaktspunkt beim letzten TimeStep angewendet hatten, erzeugen wir die CollisionPointWithImpulse-Klasse, welche sich den NormalImpulse und FrictionImpulse merken kann.

```

5   //Speichert den Normal- und Friction-Impulse-Wert für ein Kollisionspunkt
14 Verweise
6   internal class CollisionPointWithImpulse : RigidBodyCollision
7   {
8     public float NormalImpulse = 0;
9     public float FrictionImpulse = 0;
10
11   1 Verweis
12   public CollisionPointWithImpulse(RigidBodyCollision c)
13     : base(c)
14   {
15   }
16
17   1 Verweis
18   public void TakeDataFromOtherPoint(CollisionPointWithImpulse c)
19   {
20     this.NormalImpulse = c.NormalImpulse;
21     this.FrictionImpulse = c.FrictionImpulse;
22   }

```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/CollisionPointWithImpulse.cs

Wir nutzen nun den CollisionPointsCache, welchen wir auch schon beim Matrix- und JRow-Solver verwendet haben, um die RigidBodyCollision-Objekte in CollisionPointWithImpulse-Objekte umzuwandeln.

```

36   0 Verweis
37   public void Resolve(List<IRigidBody> bodies, RigidBodyCollision[] collisions1, float dt, SolverSettings s
38   {
39     if (collisions1.Length == 0) return;
40     var collisions = this.pointCache.Update(collisions1);

```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/SequentiellImpulseResolver.cs

Sowohl die NormalConstraint als auch FrictionConstraint hat eine Referenz auf das CollisionPointWithImpulse-Objekt, wo es durch Aufruf von SaveImpulse die AccumulatedImpulse-Variable beim Kontaktpunkt speichert (Zeile 80).

```

23   public float AccumulatedImpulse { get; set; }
24
25   private CollisionPointWithImpulse point;
26   1 Verweis
27   public NormalConstraint(ConstraintConstructorData data, CollisionPointWithImpulse point)
28   {
29     this.AccumulatedImpulse = data.Settings.DoWarmStart ? point.NormalImpulse : 0;
30
31   1 Verweis
32   private float GetBias(ConstraintConstructorData data, CollisionPointWithImpulse c, Vector2D r1, Vector2D r2)...
33
34   2 Verweise
35   public void SaveImpulse()
36   {
37     this.point.NormalImpulse = this.AccumulatedImpulse;
38   }

```

Diese Methode wird immer dann aufgerufen, wenn die AccumulatedImpulse-Variable einen neuen Wert bekommen hat (Zeile 64).

```

10  public static void MoveBodiesWithConstraint(List<IRigidBody> bodies, CollisionPointWithImpulse[] collisions, float dt)
11  {
12      //Schritt 1: Erzeuge ForceDirection, ImpulseMass und Bias für jedes Constraint-Objekt
13      float invDt = dt > 0.0f ? 1.0f / dt : 0.0f;
14      var constraints = new ConstraintFactory().CreateConstraints(new ConstraintConstructorData()..., collisions);
15
16      //Schritt 2: Wende die externe Kraft für alle Körper an
17      foreach (var body in bodies)...
18
19      //Schritt 3: Wende den ersten Impuls an, welcher bereits den Großteil der Korrektur erreichen sollte
20      if (settings.DoWarmStart)
21      {
22          foreach (var c in constraints)
23          {
24              ApplyImpulse(c, c.AccumulatedImpulse);
25          }
26      }
27
28      //Schritt 4: Finde per PGS Relative-Kontaktpunktgeschwindigkeitswerte, welche dem Bias-Wert entsprechen
29      for (int i=0;i<settings.IterationCount;i++)
30      {
31          foreach (var c in constraints)
32          {
33              float impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection);
34
35              // Clamp the accumulated impulse
36              float oldSum = c.AccumulatedImpulse;
37              c.AccumulatedImpulse = MathHelp.ClampOldSum + impulse, c.MinImpulse, c.MaxImpulse);
38              impulse = c.AccumulatedImpulse - oldSum;
39
40              ApplyImpulse(c, impulse);
41
42              c.SaveImpulse();
43          }
44      }
45
46  }
47
48  }
49
50  }
51
52  }
53
54  }
55
56  }
57
58  }
59
60  }
61
62  }
63
64  }
65
66  }
67

```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/ResolverHelper.cs

Um nun den WarmStart-Impuls anzuwenden wird der gespeicherte AccumulatedImpulse-Wert im Normal- und Frictionconstraintkonstruktor ausgelesen (Zeile 53 im vorherigen Bild) und dann bei Zeile 36 wird dieser ausgelesene Impuls angewendet. Es wäre auch erlaubt, dass man Schritt 2 und Schritt 3 vertauscht. Wichtig ist nur, dass der WarmStart-Impuls nach der Erstellung der Constraint-Objekte und vor der PGS-Schleife (Schritt 4) erfolgt.

Constraint-Kraft und V2 berechnen – SI vs Matrix/JRow

Es wurden jetzt alle Schritte vom Sequentiellen Impulsverfahren Einzeln besprochen. Nun geht es noch mal um den Gesamtablauf und wie er im Vergleich zum MatrixSolver sich verhält.

Sowohl bei SI als auch der Matrix will man eine Geschwindigkeit V2 für alle Körper berechnen, welche die Constraint-Bedingungen erfüllt. Dazu ist es nötig die Constraint-Kraft zu berechnen. Beim Matrix-Solver habe ich eine Gleichung, mit der ich die Constraint-Kraft berechnen kann und ich habe eine Gleichung, mit der ich V2 berechnen kann. Beide Gleichungen ergeben sich aus Newtons zweiten Gesetz und der Implizit-Euler-Formel. Beide kann ich getrennt verstehen und getrennt berechnen. Die Schwierigkeit bei SI ist, dass er das Berechnen der Constraint-Kraft und das Berechnen von V2 miteinander vermischt. Innerhalb der PGS-Schleife korrigiert er auch schon die Velocity-Werte was aber deswegen geht, weil er einerseits den Bias-Wert nicht mit verändert und andererseits fragt er pro PGS-Schritt nicht: Wie muss die Constraint-Kraft lauten, damit ich von V1 nach V2 komme? Sondern SI fragt: Wie muss die Constraint-Kraft lauten, damit ich vom aktuellen V zu V2 komme? Ich springe mit mehreren Schritten mit den V-Punkt von V1 nach V2 hin. Deswegen wird bei der SI-Formel der aktuelle V-Abstand zum Bias genommen und bei der Matrix wird fix für den Abstand von V1 zu V2 die Constraintkraft berechnet.

Sequentielle Impulse

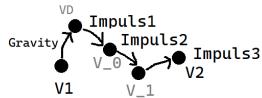
```

41   for (int i=0;i<settings.IterationCount;i++)
42   {
43     foreach (var c in constraints)
44     {
45       float impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection);
46       ApplyImpulse(c, impulse);
47     }
48   }

```

Schritt 1:
Constraintkraft per PGS ermitteln

Schritt 2: Unter Nutzung der Constraintkraft die Velocity-Werte korrigieren



Matrix/JRow

```

30 // Ermittle mit PGS für jedes Constraint Lambda
31 for (int i = 0; i < iterations; i++)
32 {
33   for (int y = 0; y < constraints.Length; y++)
34   {
35     // Löse constraints[y].Lambda unter Nutzung von Zeile y
36     float sum = 0;
37     for (int j = 0; j < constraints.Length; j++)
38     {
39       if (j != y)
40         sum += constraints[y].ARow[j] * constraints[j].Lambda;
41     }
42     constraints[y].Lambda = (constraints[y].B - sum) / constraints[y].ARow[y];
43     constraints[y].Lambda = MathHelp.Clamp(constraints[y].Lambda, constraints[y].MinLambda, constraints[y].MaxLambda);
44   }
45 }
46 // Verschiebe die Körper aufgrund der Constraintkraft
47 foreach (var constraint in constraints)
48 {
49   MoveBodyWithDeltaV(constraint.J1, constraint.Lambda, constraint.B1, dt);
50   MoveBodyWithDeltaV(constraint.J2, constraint.Lambda, constraint.B2, dt);
51 }

```

Constraint-Kraft = Wie komme ich von V1 nach V2 mit genau ein Sprung?



Anstatt dass man beim PGS eine feste Matrix A und B hat, um damit dann ein Lambda zu berechnen habe ich bei SI lediglich die Bias-Werte (Ziel-Relativegeschwindigkeit) fest aber der Velocity-Wert wird zusammen mit dem Impulswert pro Constraint-Iterationsschritt berechnet und verändert.

Diese Vermischung macht es so schwierig mit den Matrix-Wissen SI zu verstehen. Stattdessen sollte man sich SI eher so vorstellen, dass ich eine Tischdecke gerade rücken will. Am Anfang überlege ich mir, an welcher Position die Eckpunkte der Decke liegen sollen (Bias-Wert). Dann ziehe ich jeweils immer nur an einer Ecke und das mache ich so oft, bis alle Ecken an der richtigen Position liegen. Während des Ziehens wende ich also eine Kraft für eine Zeit an (Impuls) und ich verändere auch gleich die Position (V2-Anpassung).

Beim Matrix-Ansatz würde ich mir vorher theoretisch überlegen, wie stark ich an jeder Ecke genau ziehen müsste (Lambda-Berechnung) und dann würde ich pro Ecke genau einmal nur ziehen müssen (V2 zuweisen), weil ich mir vorher alles genau überlegt habe.

Die unterschiedliche Behandlung der externen Kraft

Der Grund warum ich über das Thema hier nochmal spreche ist, weil alle Solver unterscheiden sich in der Art, ab wann und wo sie die externe Kraft anwenden (D.h. Den Velocity-Wert wegen Gravity verändern).

Matrix / JRow

Wenn es keine Kontaktpunkte gibt, dann gibt es auch keine Constraint-Matrix/Constraint-Kraft und somit muss die externe Kraft direkt ohne Solver auf die Körper angewendet werden. Sollte es aber Kontaktpunkte geben, dann wird zuerst die Constraint-Kraft berechnet und danach wird dann die Constraint- und Externe-Kraft angewendet. Es ist egal, in welcher Reihenfolge ich die Constraint- und Extern-Kraft anwende.

Serielle Impulse

Hier muss die externe Kraft (Fext) genau zwischen der Berechnung der Bias-Werte und der PGS-Schleife angewendet werden. Erfolgt Fext vor der Bias-Berechnung, dann verfälscht es die Biaswerte. Kommt es nach der PGS-Schleife, dann zerstört man den einmal gefundenen V2-Punkt.

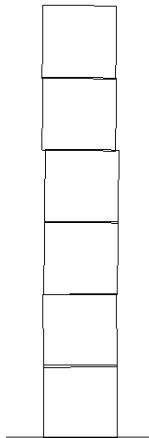
Die Gemeinsamkeit von allen Solvoren ist, dass die externe Kraft immer erst nach der Bias-Wert-Erstellung angewendet werden darf. Wendet man die externe Kraft vor der Bias-Erstellung an, bekommt man falsche Bias-Werte.

Der Unterschied ist, dass bei Matrix/JRow die Reihenfolge der Constraint- und Extern-Kraft-Anwendung egal ist und bei SI muss erst die externe Kraft angewendet werden und danach erst die Constraint-Kraft.

Vergleich der verschiedenen CollisionResolution-Solver

Die Geschwindigkeit der Solver soll durch die Laufzeit vom UnitTest PositionCorrectionTests.LongCubeStack_FallingDown_BecomeCalm verglichen werden.

Dort hat man ein langen Würfelstapel, wo die Würfel aufeinander fallen. Ich simuliere 500 TimeSteps und schaue am Ende ob der Stapel noch steht. Wenn nicht gilt das als Komplettversagen (So etwas wird hier nicht geduldet).



Solver	PGS-Iterationen=100; TestRuns=10	PGS-Iterationen=20; TestRuns=20	PGS-Iterationen=10; TestRuns=30
Matrix	5.37s (1.98)	4.83s (1.81)	4.94s (1.82)
Sequentielle Impulse (SI)	3.39s (1.25)	3.28s (1.23)	3.46s (1.28)
JRow	2.71s (1)	2.67s (1)	2.71s (1)

Ich habe die Simulation TestRuns mal hintereinander ausgeführt und davon dann die Zeit gemessen. Einmal hatte ich mit 100 PGS-Iterationen gearbeitet, was all den Aufwand, den ich vor dem PGS-Iterieren mache weniger schlimm ins Gewicht fallen lässt. Dann habe ich mit 20 PGS-Iterationen gearbeitet, so dass die Rechenschritte innerhalb der PGS-Iterationen mehr ins Gewicht fallen.

Ergebnis: Am schnellsten ist JRow. Dann kommt SI und als letztes der Matrix-Solver. In Klammern habe ich dahinter geschrieben, um welchen Faktor die Verlierer langsamer waren als JRow.

Wenn man den Quelltext miteinander vergleicht (Siehe SolverCompareTest.CompareLinesOfCode) dann hat die Matrix am meisten Quelltext und SI/II am wenigsten.

Hier sieht man die Anzahl der Quelltextzeilen und in Klammern die Prozentzahl gegenüber dem Maxwert.

- Matrix: 831 (100%)
- JRow: 385 (46%)
- SI: 370 (44%)
- IterativeImpulse: 59 (7%)

Vergleicht man den Einarbeitungsaufwand, so benötigt man folgende Wissensbausteine für die jeweiligen Verfahren:

	Iterative Impulse	Matrix	JRow	SI
Normalimpulsformel	X			X
Impuls anwenden	X	X	X	X
Bewegungsgleichung als Matrix		X	X	
NormalConstraint		X	X	X
Constraint-Bias		X	X	X
Projected Gauss Seidel		X	X	X
PositionCorrection		X	X	X
WarmStart		X	X	X
AccumulatedImpulse-Clamping				X

Iterative Impulse benötigt lediglich die Herleitung der Normalimpulsformel+Anwendung wohin gehend SI 8 Dinge braucht, um das zu verstehen. Da es leichter war den Matrix-Solver zu verstehen, haben wir uns zuerst damit in Teil 2 beschäftigt. SI benötigt das Wissen von Teil 1 und 2, um es verstehen zu können. Der JRow-Solver war ein Versuch von mir und ist nicht unbedingt nötig, um SI zu verstehen.

Der JRow-Solver ist zwar am schnellsten aber er hat halt den Nachteil, dass die Constraint-Objekte untereinander sich kennen/eine Abhängigkeit haben. SI hat die Eleganz, dass es wenig Quelltextzeilen benötigt und man eine klare Trennung zwischen den einzelnen Constraint-Objekten hat. Lediglich der Einarbeitungsaufwand ist dort am höchsten.

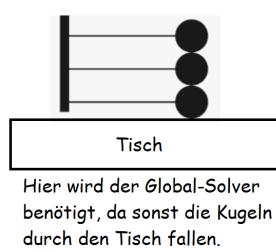
Mein Fazit wäre also vom Quellcode und von der Laufzeit her ist SI recht gut aber vom Einarbeitungsaufwand ist es am schwersten. Will man eine einfache Lösung, nimmt man die Matrix. Will man eine schnelle Lösung, nimmt man JRow. Will man eine recht schnelle und kompakte Lösung und ist bereit mehr Einarbeitungsaufwand zu spendieren, nimmt man SI.

Das Newton-Pendel

Will ich ein Newton-Pendel mit der Physikengine simulieren, dann muss ich ihr durch Aufruf von UseGlobalSolver sagen, dass sie die Kontaktpunkte laut BodyIndex1-BodyIndex2 gruppieren soll.

```
45 | var scene = new PhysicScene(bodies);
46 | if (useGlobalSolver) scene.UseGlobalSolver(); else scene.UseGroupSolver();
```

Will ich aber ein Kugel-Stapel simulieren, dann muss ich den globalen Solver nutzen, da sonst die Kugeln durch den Tisch hindurch fallen.



Es wäre ja nun schön, wenn man der Physikengine nicht von außen noch extra sagen müsste, ob sie nun den Grouped- oder Global-Solver nutzen soll sondern sie das automatisch selber erkennt. Das Problem ist, was macht man, wenn man ein Pendel hat, was am Anfang noch wie im linken Bild aussieht und dann drehe ich das um 90 Grad. Soll der Engine dann das erkennen und automatisch die Kontaktpunkte anders gruppieren? Weil mir das wie ein billiger Trick erscheint lasse ich die Engine so und man muss halt weiterhin ihr sagen, wenn sie ein Newton-Pendel simulieren soll.

Constraint-Kraft vs Constraint-Impuls

Wenn man sich die Herleitung vom Matrix-Solver aus Teil 2 ansieht, dann wird dort eine Constraint-Kraft ausgerechnet. In der Projected Gaus-Seidel-Schleife (PGS) stellt Lambda die Länge der Constraint-Kraftvektoren dar. Beim sequentiellen Impuls-Verfahren wird in der PGS-Schleife die Impulsformel aus Teil 3 (Formel 5) verwendet. Lambda entspricht beim SI-Verfahren Constraint-Kraft * Impuls-Anwendungszeit (beides unbekannte Zahlen beim Colliding Contact). D.h. Der Matrixsolver arbeitet mit Constraintkräften und der SI-Solver mit Constraint-Impulsen. Da beim Matrix-Solver die Constraint-Kraft Dt Sekunden lang wirkt, kann der Constraint-Impuls dadurch ermittelt werden, indem die Constraint-Kraft mit Dt multipliziert wird.

Bei ein RestingContact weiß man ja, dass die Constraint-Kräfte und die Extern-Kräfte beide Dt Sekunden lang wirken. Also kann die Constraintkraft über $F_c = \frac{P_c}{dt}$ ermittelt werden. Beim CollidingContact weiß man aber nicht, wie lange der NormalImpuls wirkt, um zwei Körper auseinander zu bekommen. Deswegen kann man also bei ein gegebenen ConstraintImpuls bei ein CollidingContact nicht die Constraintkraft ausrechnen.

Der Grund, warum ich hier nochmal Constraint-Kräfte und Constraint-Impulse erwähne ist, weil mir bei den Studien der Paper von Erin und Marijn aufgefallen ist, dass $J^T \lambda$ mal eine Kraft und mal ein Impuls ist.

Hier ist es mal eine Kraft:

Iterative Dynamics with Temporal Coherence - Erin Catto 2005 Seite 5 $F_c = J^T \lambda$

Constraint based physics solver - Marijn 2015 Seite 12 $\mathbf{f}_c = \mathbf{J}^T \boldsymbol{\lambda}$

Hier ist es dann ein Impuls:

Modeling and Solving Constraints - Erin Catto 2009 Seite 54 $\mathbf{P}_c = \mathbf{J}^T \boldsymbol{\lambda}$

3D Constraint Derivations for Impulse Solvers - Marijn 2015 $\mathbf{p}_c = \mathbf{J}^T \boldsymbol{\lambda}$

Ich denke je nachdem, ob man als Ausgangsgleichung die Matrix-Gleichung aus Teil 2 oder die Impulsformel aus Teil 3 verwendet, kommt man dann zu dem Schluss, dass $J^T \lambda$ ein Impuls oder eine Kraft ist.

Zusammenfassung von Teil 3

Wir haben uns nun den sequentiellen Impulse-Solver angesehen, welcher die Normal- und Friction-Kraft berechnet. Damit kann nun die Bewegung von Kugeln und Würfeln simuliert werden. Im nächsten Abschnitt geht es um die Simulation von Gummibändern und Objekten, der mit der Maus festgehalten werden.

Quellen für diesen Abschnitt

<https://github.com/erincatto/box2d-lite>

Iterative Dynamics with Temporal Coherence - Erin Catto 2005

Constraint based physics solver - Marijn 2015

Comparison between Projected Gauss-Seidel and Sequential Impulse Solvers for Real-Time Physics Simulations - Marijn 2015

3D Constraint Derivations for Impulse Solvers - Marijn 2015

Modeling and Solving Constraints - Erin Catto 2009

<https://kevinyu.net/2018/01/17/understanding-constraint-solver-in-physics-engine/>

<https://kevinyu.net/2018/02/01/improving-the-stability-of-your-physics/>

http://mft-spirit.nl/files/MTamis_ConstraintBasedPhysicsSolver.pdf

<https://www.youtube.com/watch?v=SHinxAhv1ZE>

Global solvers versus local solvers

Global

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Local (iterative)

```
for k = 1 to num_iterations
    for i = 1 to 3
        solve  $a_i \lambda_i = b_i$ 
    end
end
```

Solving position errors using pseudo velocity

$$m\ddot{\mathbf{v}} = \mathbf{n}\lambda$$

Pseudo Newton's law

$$I\ddot{\boldsymbol{\omega}} = (\mathbf{r} \times \mathbf{n})\lambda$$

Pseudo velocity constraint

$$\dot{C} = (\ddot{\mathbf{v}} + \ddot{\boldsymbol{\omega}} \times \mathbf{r}) \cdot \mathbf{n} - \beta s$$

Clamping the accumulated impulse

```
oldImpulse = constraint.impulse;
delta = constraint.ComputeImpulse();
constraint.impulse += delta;
constraint.impulse = max(0, constraint.impulse);
delta = constraint.impulse - oldImpulse;
constraint.ApplyImpulse(delta);
```

Warm starting requires us to accumulate impulses

```
void Solve()
{
    delta = constraint.ComputeImpulse();
    constraint.impulse += delta;
    constraint.ApplyImpulse(delta);
}
```

Beginning of next time step:

```
void WarmStart()
{
    constraint.ApplyImpulse(constraint.impulse);
}
```

Teil 4: Soft- und Multiconstraints

Ziel dieses Abschnitts

Hier soll am Beispiel der Distanz-Constraint (Eisenfeder) eine SoftConstraint erklärt werden und am Maus-Constraint-Beispiel (Objekt per Maus verschieben) was eine Multiconstraint ist.

Cleanup & Refactoring

Ich entferne den Matrix-, JRow- und Iterative-Resolver da nur noch SI verwendet wird.
CollisionPointsCache ist nun nicht mehr generisch.

Die effektive Masse

In Teil 3 hatten wir für den Sequentiellen Impuls-Algorithmus die Impulsformel hergeleitet, welche so aussieht:

$$p_c = J^T * \frac{\xi - J * V}{J * M^{-1} * J^T}$$

Im Quellcode haben wir den $\frac{1}{J * M^{-1} * J^T}$ -Term dann *ImpulseMass* genannt:


```

    float impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection);

```

Dieser Nenner der Impuls-Formel wird von Marijn im Dokument '3D Constraint Derivations for Impulse Solvers - Marijn 2015' auf Seite 7 effektive Masse genannt.

Expanding the products gives us the effective mass:

$$\begin{aligned} \mathbf{JM}^{-1} &= [-\mathbf{L}_a m_a^{-1}, -\mathbf{A}_a \mathbf{I}_a^{-1}, \mathbf{L}_b m_b^{-1}, \mathbf{A}_b \mathbf{I}_a^{-1}] \\ \boxed{\mathbf{JM}^{-1} \mathbf{J}^T} &= \mathbf{L}_a \mathbf{L}_a^T m_a^{-1} + \mathbf{A}_a \mathbf{I}_a^{-1} \mathbf{A}_a^T + \mathbf{L}_b \mathbf{L}_b^T m_b^{-1} + \mathbf{A}_b \mathbf{I}_a^{-1} \mathbf{A}_b^T \end{aligned} \quad (2.6)$$

Er orientiert sich dabei an 'Soft Constraints - Erin Catto 2011' Seite 42:

$$m_{eff} = \frac{1}{JM^{-1} J^T}$$

Um den Zeilenvektor \mathbf{J} zu erhalten hatten wir bei der Normal- und Frictionconstraint zuerst eine PositionConstraint-Funktion aufgestellt, welche den Abstand zwischen zwei Ankerpunkte misst und die Ableitung von der PositionConstraint hat die VelocityConstraint ergeben. Beide Funktionen haben eine skalare Zahl zurück gegeben und bei der $\mathbf{J}^* \mathbf{V}$ -Umstellung hat sich dann ein \mathbf{J} -Zeilenvektor ergeben, welcher aus einer Zeile mit 6 Zahlen besteht.

Wir werden später in Teil 4 dann eine PositionConstraint-Funktionen kennen lernen, welche ein 2D-Spaltenvektor als Rückgabetyp hat. Das \mathbf{J} , was daraus dann entsteht ist eine Matrix mit 6 Spalten und 2 Zeilen.

Wenn \mathbf{J} ein Zeilenvektor ist, dann ist die effektive Masse eine skalare Zahl. Sollte \mathbf{J} aber eine Matrix mit 2 Zeilen sein, dann ergibt $J M^{-1} J^T$ eine 2*2-Matrix und die effektive Masse erhalte ich dann durch die Inverse von dieser 2*2-Matrix.

Die Effektivmasse ist der Umrechnungsfaktor zwischen ein Impuls und vrel $v_{rel} = b - J * V$:

$$m_{eff} = \frac{p}{v_{rel}}$$

Quelle für diese Formel: Soft Constraints - Erin Catto 2011 Seite 41

Über $p=mEff * vRel$ kann ich den Constraint-Impuls (Skalar oder Spaltenvektor) ausrechnen.

Wie berechne ich die effektive Masse

Marijn schreibt auf Seite 7 in „3D Constraint Derivations for Impulse Solvers - Marijn 2015“:

$$\begin{aligned} \mathbf{J} &= [-\mathbf{L}_a, -\mathbf{A}_a, \mathbf{L}_b, \mathbf{A}_b] \\ \mathbf{JM}^{-1} \mathbf{J}^T &= \mathbf{L}_a \mathbf{L}_a^T m_a^{-1} + \mathbf{A}_a \mathbf{I}_a^{-1} \mathbf{A}_a^T + \mathbf{L}_b \mathbf{L}_b^T m_b^{-1} + \mathbf{A}_b \mathbf{I}_a^{-1} \mathbf{A}_b^T \end{aligned}$$

L ist die lineare Komponente und A die angular Komponente vom Jacobi-Vektor. Das kleine m ist die Masse und groß I der Inertia-Wert/Matrix.

$$\mathbf{M}^{-1} = \begin{bmatrix} m_a^{-1} \mathbf{D} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_a^{-1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & m_b^{-1} \mathbf{D} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{I}_b^{-1} \end{bmatrix}$$

Mit dieser Formel kann ich sowohl für 2D als auch 3D-Körper die effektive Masse berechnen.

Für den 2D-Fall gilt:

Wenn \mathbf{J} nur eine Zeile enthält, dann ist \mathbf{A}_a und \mathbf{A}_b jeweils eine Float-Zahl und \mathbf{L}_a und \mathbf{L}_b eine 2D-Vektor. Wenn das normierte Richtungsvektoren sind, dann ist $\mathbf{L}_a * \mathbf{L}_a^T = 1$.

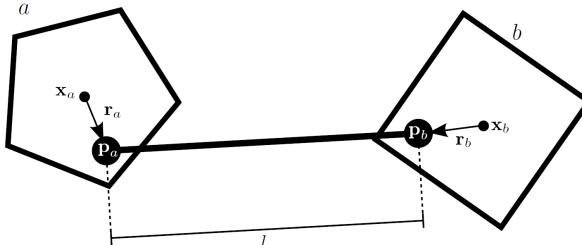
In diesen Abschnitt ging es um folgende Dinge: Bei der Impulsformel taucht ein Faktor auf, der sich

Effektivmasse nennt. Das ist eine skalare Zahl oder eine 2×2 -Matrix. Man kann beim Ausrechnen des $J M^{-1} J^T$ -Terms entweder Matrix-Multiplikationen anwenden oder man nutzt die Formel von Marijn wo man dann mit $\mathbf{L}_a^* \mathbf{L}_a^T = 1$ und $\mathbf{L}_b^* \mathbf{L}_b^T = 1$ vereinfachen kann.

Distance-Constraint ohne Feder (Metallstab)

Quelle für die Formeln und das Bild von diesen Abschnitt: 3D Constraint Derivations for Impulse Solvers - Marijn 2015 Seite 10 und 11

Wenn ich möchte, dass zwei Anker-Punkte von zwei Körpern immer den gleichen Abstand zueinander haben, dann nutze ich eine Distance-Constraint.



Quelle für das Bild: 3D Constraint Derivations for Impulse Solvers - Marijn 2015 Seite 10

Die Körper a und b sind dann wie durch ein Metallstab an den Punkten p_a und p_b verbunden. In Richtung des Stabes wirkt dann eine Zug- oder Druckkraft, welche die Körper immer auf den Abstand l hält.

Schritt 1: Positionconstraint

Um eine Constraint zu definieren, muss ich zuerst die Position-Constraint aufstellen, welche prüft, um wie viel der Abstand zwischen p_a und p_b gegenüber der Zielvorgabe l abweicht:
Ich definiere dazu den Richtungsvektor \mathbf{n} :

$$\mathbf{n} = \frac{\mathbf{p}_b - \mathbf{p}_a}{|\mathbf{p}_b - \mathbf{p}_a|}$$

Die Distance-PositionConstraint ist somit:

$$c(\mathbf{x}_a, \mathbf{q}_a, \mathbf{x}_b, \mathbf{q}_b) = (\mathbf{p}_b - \mathbf{p}_a) \cdot \mathbf{n} - l$$

Schritt 2: Velocity-Constraint

Ich leite nach der Zeit ab um die Velocity-Constraint zu erhalten:

$$\begin{aligned} \dot{c}(\mathbf{x}_a, \mathbf{q}_a, \mathbf{x}_b, \mathbf{q}_b) &= (\mathbf{v}_b + \boldsymbol{\omega}_b \times \mathbf{r}_b - \mathbf{v}_a - \boldsymbol{\omega}_a \times \mathbf{r}_a) \cdot \mathbf{n} \\ &= \mathbf{v}_b \mathbf{n} + (\boldsymbol{\omega}_b \times \mathbf{r}_b) \mathbf{n} - \mathbf{v}_a \mathbf{n} - (\boldsymbol{\omega}_a \times \mathbf{r}_a) \mathbf{n} \\ &= \mathbf{v}_b \mathbf{n} + \boldsymbol{\omega}_b (\mathbf{r}_b \times \mathbf{n}) - \mathbf{v}_a \mathbf{n} - \boldsymbol{\omega}_a (\mathbf{r}_a \times \mathbf{n}) \end{aligned}$$

Schritt 3: J- und V-Vektor extrahieren

Ich extrahiere nun \mathbf{v} um somit den J-Vektor zu erhalten:

$$\mathbf{Jv} = \begin{bmatrix} -\mathbf{n} \\ -(\mathbf{r}_a \times \mathbf{n}) \\ \mathbf{n} \\ (\mathbf{r}_b \times \mathbf{n}) \end{bmatrix}^T \begin{bmatrix} \mathbf{v}_a \\ \boldsymbol{\omega}_a \\ \mathbf{v}_b \\ \boldsymbol{\omega}_b \end{bmatrix}$$

Schritt 4: Die effektive Masse berechnen

$$\mathbf{J} = [-\mathbf{L}_a, -\mathbf{A}_a, \mathbf{L}_b, \mathbf{A}_b]$$

$$\mathbf{JM}^{-1}\mathbf{J}^\top = \mathbf{L}_a\mathbf{L}_a^\top m_a^{-1} + \mathbf{A}_a\mathbf{I}_a^{-1}\mathbf{A}_a^\top + \mathbf{L}_b\mathbf{L}_b^\top m_b^{-1} + \mathbf{A}_b\mathbf{I}_b^{-1}\mathbf{A}_b^\top \quad (2.6)$$

$$\mathbf{Jv} = \begin{bmatrix} -\mathbf{n} \\ -(\mathbf{r}_a \times \mathbf{n}) \\ \mathbf{n} \\ (\mathbf{r}_b \times \mathbf{n}) \end{bmatrix}^\top \begin{bmatrix} \mathbf{v}_a \\ \boldsymbol{\omega}_a \\ \mathbf{v}_b \\ \boldsymbol{\omega}_b \end{bmatrix} \quad (3.3)$$

$$\mathbf{L}_a = \mathbf{n} \quad \mathbf{L}_a\mathbf{L}_a^\top = \mathbf{1}$$

$$\mathbf{A}_a = (\mathbf{r}_a \times \mathbf{n}) \quad \mathbf{L}_b\mathbf{L}_b^\top = \mathbf{1}$$

$$\mathbf{L}_b = \mathbf{n}$$

$$\mathbf{A}_b = (\mathbf{r}_b \times \mathbf{n}) \quad \boxed{\mathbf{JM}^{-1}\mathbf{J}^\top = m_a^{-1} + \mathbf{A}_a^2 \mathbf{I}_a^{-1} + m_b^{-1} + \mathbf{A}_b^2 \mathbf{I}_b^{-1}}$$

Schritt 5: Constraint-Klasse erstellen

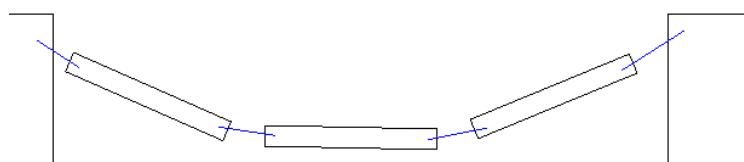
```

28  public DistanceJointConstraint(ConstraintConstructorData data, DistanceJoint joint)
29  {
30      var s = data.Settings;
31
32      this.B1 = joint.B1;
33      this.B2 = joint.B2;
34      this.R1 = joint.Anchor1 - joint.B1.Center;
35      this.R2 = joint.Anchor2 - joint.B2.Center;
36
37
38      this.MinImpulse = float.MinValue;
39      this.MaxImpulse = float.MaxValue;
40      this.AccumulatedImpulse = s.DoWarmStart ? joint.AccumulatedImpulse : 0;
41
42      Vector2D a1Toa2 = joint.Anchor2 - joint.Anchor1;
43      float length = a1Toa2.Length();
44      Vector2D n = length > 0.0001f ? a1Toa2 / length : new Vector2D(1, 0);
45
46      this.ForceDirection = n;
47
48      float r1crossN = Vector2D.ZValueFromCross(this.R1, n);
49      float r2crossN = Vector2D.ZValueFromCross(this.R2, n);
50
51      float invMass = B1.InverseMass + B1.InverseInertia * r1crossN * r1crossN + B2.InverseMass + B2.InverseInertia * r2crossN * r2crossN;
52      this.ImpulseMass = 1f / invMass;
53
54      float biasFactor = s.DoPositionalCorrection ? s.PositionalCorrectionRate : 0.0f;
55      this.Bias = biasFactor * data.InvDt * (joint.Length - length);

```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Constraints/DistanceJointConstraint.cs

Schritt 6: Beispielanwendung für Distanzjoint: Hängebrücke



Die blauen Linien sind Distanzjoints.

Distance-Constraint als gedämpfte Schwingung (Eisenfeder)

Die Kraftgleichung der gedämpften Feder



Wenn man ein Raum ohne Schwerkraft und Luftreibung hat, in welchen man eine Feder an einer Wand befestigt, an der ein Gewicht hängt, dann wirken zwei Kräfte auf dieses Gewicht:

- Die Kraft aufgrund der Auslenkung der Feder wird über das Hookesche Gesetz beschrieben: $F=-k*x$ (x = Auslenkung; k =Federkonstante)
- Eine Dämpfungskraft aufgrund der Reibung innerhalb der Feder. Laut dem Gesetz von Stokes ist sie proportional zur Geschwindigkeit: $F=-c * dx/dt$ (c =Dämpfungskonstante; $dx/dt=v(t)$)

Die Kraft, die auf das Massestück wirkt wird laut Wikipedia über diese Gleichung beschrieben:

$$F = -kx - c \frac{dx}{dt} = m \frac{d^2x}{dt^2}$$

https://en.wikipedia.org/wiki/Harmonic_oscillator

Federsimulation als externe Kraft

Ich kann eine Feder simulieren, indem ich der Force-Property von den Massestück, welches an der Feder hängt, einen Wert laut der Feder-Kraft-Gleichung zuweise. Die Force-Property wird auch für die Zuweisung der Schwerkraft genutzt. Die Anwendung der Kräfte aus der Force-Property erfolgt beim SI-Solver nach der Constraint-Objekterstellung und vor der Constraint-Schleife.

So sieht dann die Berechnung der Federkraft aus:

```

14  public static void MoveBodiesWithConstraint(List<IRigidBody> bodies, List<IJoint> joints, CollisionPointWithImpulse[] collision
15  {
16      //Schritt 0: Setze die Body.Force-Property laut Federkraftformel
17      foreach (var joint in joints)
18          GetForceFromDistanceJoint(joint as DistanceJoint);
19
20      //Diese Funktion zeigt, das man eine Feder auch darüber simulieren kann, indem man sie als externe Kraft betrachtet
21      1 Verweis
22      private static void GetForceFromDistanceJoint(DistanceJoint joint)
23  {
24
25          Vector2D a1Toa2 = joint.Anchor2 - joint.Anchor1;
26          float length = a1Toa2.Length();
27          Vector2D ForceDirection = length > 0.0001f ? a1Toa2 / length : new Vector2D(1, 0);
28
29          Vector2D R1 = joint.Anchor1 - joint.B1.Center;
30          Vector2D R2 = joint.Anchor2 - joint.B2.Center;
31
32          //VelocityAtContactPoint = V + mAngularVelocity cross R
33          Vector2D v1 = joint.B1.Velocity + new Vector2D(-joint.B1.AngularVelocity * R1.Y, joint.B1.AngularVelocity * R1.X);
34          Vector2D v2 = joint.B2.Velocity + new Vector2D(-joint.B2.AngularVelocity * R2.Y, joint.B2.AngularVelocity * R2.X);
35          Vector2D relativeVelocity = v2 - v1;
36
37          // Relative velocity in Force direction
38          float velocityInForceDirection = relativeVelocity * ForceDirection;
39
40          float force = -joint.Damping * velocityInForceDirection - joint.Stiffness * (length - joint.Length);
41
42          joint.B1.Force -= ForceDirection * force;
43          joint.B2.Force += ForceDirection * force;
44
45      }
46
47  }

```

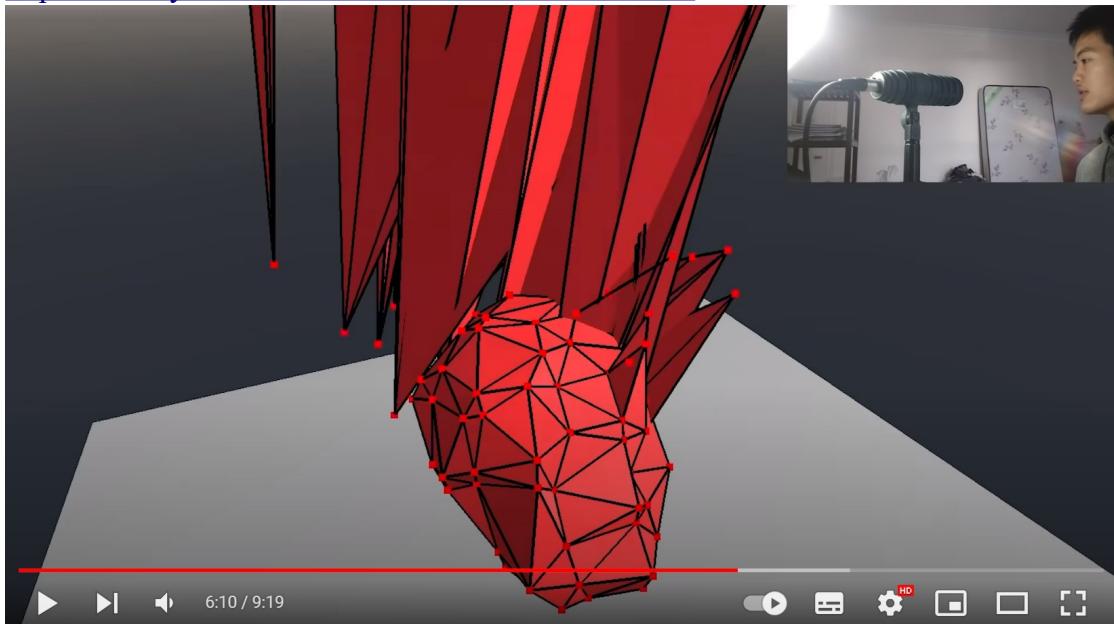
Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Examples/ResolverHelper1.cs

Der Vorteil bei diesen Weg der Simulation ist, dass ich lediglich die Formel aus Wikipedia direkt in meine Physikengine übertragen brauche und schon habe ich eine gedämpfte Feder in meine Engine eingebaut.

Der Nachteil ist aber, das bei dieser Art der Federsimulation es passieren kann, dass die Feder explodiert, wenn der Stiffness- und TimeStep-Wert zu hoch ist. Diese Art der Simulation ist bei Verwendung von Semi-Implicit Euler numerisch nicht stabil.

Hier ist ein Beispiel für eine explodierende Feder-Simulation. Der rote Ball sieht überhaupt nicht mehr wie ein runder Ball aus. Gonkee (oben rechts im Bild) ist darüber überhaupt nicht erfreut.

<https://www.youtube.com/watch?v=iSMbRGTBOHU>



Vergleich von verschiedenen Verfahren zur Federsimulation

Um das Problem mit der Feder, die explodiert, besser verstehen zu können, betrachten wir ein ein Kilo schweres Gewicht, was ohne Dämpfung schwingen soll. Es startet mit einer Auslenkung von 100 und soll im Bereich von -100 bis +100 für 3 Sekunden lang sich bewegen. Alle 50 ms Sekunden wird ein Abtastwert berechnet.

Man kann eine ungedämpfte Schwingung entweder über die explizite Formel angeben oder man nutzt ein numerischen Integrator (ODE Solver), so wie in *Soft Constraints - Erin Catto 2011* Seite

14 angegeben. In UnitTest PhysicEngine.UnitTests/SoftConstraints/SpringMassTest.cs sind die nachfolgend genannten Verfahren zu sehen.

Feder ohne Dämpfung – Simulation über explizite Formel

Laut den Wikipedia-Artikel https://en.wikipedia.org/wiki/Simple_harmonic_motion kann eine ungedämpfte Schwingung über folgende Gleichungen angegeben werden:

$$x(t) = A \cos(\omega t - \varphi) \quad \omega = 2\pi f$$

```

34  //m = Masse in Kg; xStart = Auslenkung der Feder zum Zeitpunkt t=0
35  1 Verweis | 0/1 bestanden
36  public float[] UndampedSpringWithExplicitFormular(float frequency = 1, float xStart = 100, float m = 1, float h = 50)
37  {
38  }
39  float omega = 2 * (float)Math.PI * frequency / 1000; // Durch 1000, da ich die Zeit in ms und nicht s angebe
40  List<float> xValues = new List<float>();
41  for (float t = 0; t <= 3000; t += h)
42  {
43      float x = xStart * (float)Math.Cos(omega * t);
44      xValues.Add(x);
45  }
46  return xValues.ToArray();
47 }
```

Feder ohne Dämpfung – Simulation über explicit Euler

Ich nutze nun als Ausgangsgleichung die Kraftgleichung der ungedämpften Feder von https://en.wikipedia.org/wiki/Simple_harmonic_motion

$$F_{\text{net}} = m \frac{d^2 x}{dt^2} = -kx \quad \omega = \sqrt{k/m} \quad \omega^2 m = k$$

Das Massestück wird mit $\frac{F}{m} = \frac{-kx}{m} = -\omega^2 * x$ beschleunigt. $v_2 = v_1 + h * \frac{F}{m} * x$

Als numerischen Integrator nutze ich explicit Euler von *Soft Constraints - Erin Catto 2011 Seite 15*:

$$v_2 = v_1 - h\omega^2 x_1$$

$$x_2 = x_1 + hv_1$$

Die v2-Formel enthält kein Bezug zu x2 und die x2-Formel enthält kein v2. Somit kann ich beide Formeln ohne vorher was umzustellen direkt verwenden.

```

81  public float[] UndampedSpringWithExplicitEuler(float frequency = 1, float xStart = 100, float m = 1, float h = 50)
82  {
83      //Soft Constraints - Erin Catto 2011 -> Seite 15
84      //v2 = v1 - h * k / m * x1
85      //x2 = x1 + h * v1
86      float omega = 2 * (float)Math.PI * frequency / 1000;
87      float k = m * omega * omega;
88      float x = xStart;
89      float v = 0;
90
91      List<float> xValues = new List<float>();
92      for (float t = 0; t <= 3000; t += h)
93      {
94          float v2 = v - h * k / m * x; //Explicit-Euler
95          float x2 = x + h * v;
96
97          x = x2;
98          v = v2;
99
100         xValues.Add(x);
101     }
102
103     return xValues.ToArray();
104 }
```

$$v_2 = v_1 - h\omega^2 x_1$$

$$x_2 = x_1 + hv_1$$

Soft Constraints - Erin Catto 2011 Seite 15

Feder ohne Dämpfung – Simulation über implizit Euler

Das nächste Verfahren ist implicit Euler von *Soft Constraints - Erin Catto 2011 Seite 16*:

$$x_2 = x_1 + hv_2$$

$$v_2 = v_1 - h\omega^2 x_2$$

In der x2-Formel taucht v2 auf und in der v2-Formel taucht x2 auf. Am Anfang ist aber nur x1=100 und v1=0 gegeben. Ich kann also nicht x2 ausrechnen, ohne v2 zu kennen und v2 kann ich auch nicht ausrechnen, da ich nicht x2 kenne. Aus dem Grund leite ich eine v2-Formel her, was ohne x2-Bezug auskommt, um dieses Verfahren nutzen zu können. Ich nehme Erins Gleichungen und ersetze Omega-Quadrat mit $\omega^2 = \frac{k}{m}$

$$1: \quad x_2 = x_1 + hv_2$$

$$2: \quad v_2 = v_1 - h * \frac{k}{m} * x_2$$

Ich setze für x2 in Gleichung 2 Gleichung 1 ein:

$$v_2 = v_1 - h * \frac{k}{m} * (x_1 + hv_2) \rightarrow \text{Klammer ausmultiplizieren}$$

$$v_2 = v_1 - h * \frac{k}{m} * x_1 - h * \frac{k}{m} * h * v_2 \rightarrow \text{v2-Terme auf die linke Seite bringen}$$

$$v_2 + h^2 * \frac{k}{m} * v_2 = v_1 - h * \frac{k}{m} * x_1 \rightarrow \text{v2 ausklammern}$$

$$v_2 * (1 + h^2 * \frac{k}{m}) = v_1 - h * \frac{k}{m} * x_1 \rightarrow \text{Durch die Klammer dividieren}$$

$$v_2 = \frac{v_1 - h * \frac{k}{m} * x_1}{1 + h^2 * \frac{k}{m}} \rightarrow \text{v2-Formel ohne x2-Bezug}$$

Ich kann diese neue v2-Formel ohne x2-Bezug nun beim v2-Startwert und in der t-Schleife nutzen.

```

118     public float[] UndampedSpringWithImplicitEuler(float frequency = 1, float xStart = 100, float m = 1, float h = 50)
119    {
120        float omega = 2 * (float) Math.PI * frequency / 1000;
121        float k = m * omega * omega;
122        float x = xStart;
123        float v = 0;
124        float v2 = v; //Wenn man das als Startwert nimmt, dann erhält man Werte von -100 ... +100
125        //float v2 = (v - h * k / m * x) / (1 + h * h * k / m); //Mit diesen Startwert erhält man Werte von -90 ... +90
126
127        List<float> xValues = new List<float>();
128        for (float t = 0; t <= 3000; t += h)
129        {
130            //Soft Constraints - Erin Catto 2011 -> Seite 16 -> Hier steht die Formel für Implicit Euler
131            float x2 = x + h * v2;
132            v2 = (v - h * k / m * x) / (1 + h * h * k / m); //v2-Formel ohne x2-Bezug -> Funktioniert
133            //v2 = v - h * k / m * x2; //Original-V2-Formel -> Funktioniert nur wenn die frequency nicht zu hoch ist
134
135            x = x2;
136            v = v2;
137            xValues.Add(x);
138        }
139
140        return xValues.ToArray();
141    }

```

$$x_2 = x_1 + hv_2$$

$$v_2 = v_1 - h\omega^2 x_2$$

Soft Constraints - Erin Catto 2011 Seite 16

Wenn ich sie für die Startwertberechnung nutze, dann erhalte ich nur noch X-Werte im Bereich von -90 bis +90. Anscheinend muss man als Startwert für v2 den v1-Wert nutzen damit das Verfahren funktioniert. In der t-Schleife muss ich die v2-Formel ohne x2-Bezug verwenden, wenn die Feder auch bei hohen Frequenzen funktionieren soll. Wenn man niedrige Frequenzen hat, dann funktioniert auch die v2-Formel von Zeile 133.

Feder ohne Dämpfung – Simulation über Semi-Implicit-Euler

Semi-Implicit-Euler ist das nächste Verfahren von Erin auf Seite 17:

$$x_2 = x_1 + hv_2$$

$$v_2 = v_1 - h\omega^2 x_1$$

oder auch hier https://en.wikipedia.org/wiki/Semi-implicit_Euler_method

$$v_{n+1} = v_n - \omega^2 x_n \Delta t$$

$$x_{n+1} = x_n + v_{n+1} \Delta t.$$

beschrieben wird. Erin gibt die x_2 und v_2 -Formel aber in verdrehter Reihenfolge an. Ich nutze sie aber so, dass ich zuerst v_2 ausrechne und danach erst x_2 , da in der x_2 -Formel das v_2 auftaucht und da es in dieser Reihenfolge auch im Wikipedia-Artikel so steht. Die Semi-Implicit-Euler-Funktion sieht so aus:

```

97 public float[] UndampedSpringWithSemiImplicitEuler(float frequency = 1, float xStart = 100, float m = 1, float h = 50)
98 {
99     //v2 = v1 - h * k / m * x1
100    //x2 = x1 + h * v2
101    float omega = 2 * (float)Math.PI * frequency / 1000;
102    float k = m * omega * omega;
103    float x = xStart;
104    float v = 0;
105
106    List<float> xValues = new List<float>();
107    for (float t = 0; t <= 3000; t += h)
108    {
109        //https://en.wikipedia.org/wiki/Semi-implicit_Euler_method -> Hier steht die Formel für x2/v2
110        //Soft Constraints - Erin Catto 2011 -> Seite 17 -> Hier steht die Formel auch
111        float v2 = v - h * k / m * x; //Semi-Implicit-Euler
112        float x2 = x + h * v2;
113
114        x = x2;
115        v = v2;
116
117        xValues.Add(x);
118    }
119
120    return xValues.ToArray();
121 }
```

https://en.wikipedia.org/wiki/Semi-implicit_Euler_method

$$v_{n+1} = v_n - \omega^2 x_n \Delta t$$

$$x_{n+1} = x_n + v_{n+1} \Delta t.$$

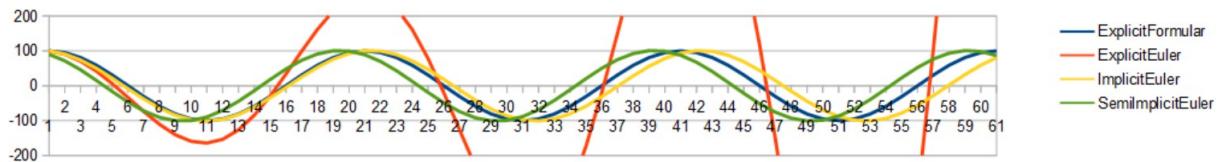
Soft Constraints -
Erin Catto 2011
Seite 17

$$x_2 = x_1 + hv_2$$

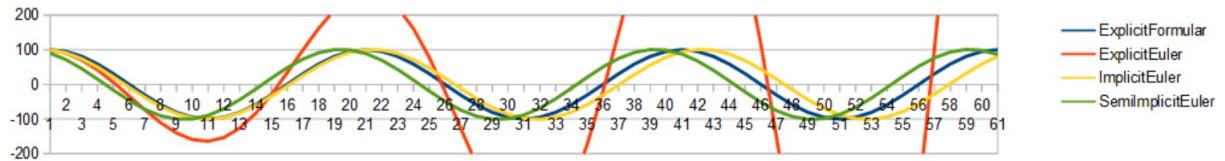
$$v_2 = v_1 - h\omega^2 x_1$$

Die Verfahren zeigen nun die Auslenkung (Y-Achse) der Feder über der Zeit (X-Achse):

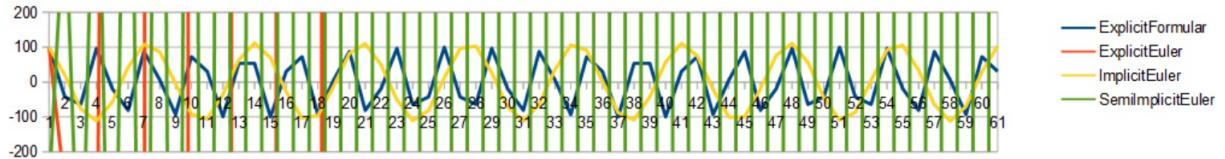
Frequency=1



Frequency=2



Frequency=6.4



Siehe: PhysicEngine.UnitTests/SoftConstraints/SpringMassTest.cs → UndampedStiffSpring_()

Was man hier sieht: Die explizite Formel und Implicit Euler erzeugen immer eine Sinuskurve mit richtiger Auslenkung von -100 bis +100 selbst wenn die Frequenz höher wird. Wegen der ungenauen Abtastung sieht die blaue Kurve aber bei höheren Frequenzen etwas eckig aus. Bei der gelben Kurve (Implicit Euler) stimmt die Anzahl der Schwingungen nicht mehr, wenn die Frequenz zu hoch wird.

ExplicitEuler und SemilimplicitEuler erzeugen Auslenkungswerte, die weit höher als wie

vorgegeben im Bereich von -100 bis +100 liegen. Diese viel zu hohen Auslenkungswerte habe ich hier mit Federexplosion bezeichnet. ExplicitEuler explodiert eher als SemiImplicitEuler.

Unsere PhysicEngine verwendet SemiImplicitEuler. Da dieses Verfahren numerisch nicht stabil ist, ist also die Feder explodiert, als wir die Force-Property von den Starrkörpern mit der Federkraft gefüttert hatten. Die explizite Formel kann in unserer PhysikEngine nicht verwendet werden, da unserer Kollisionsroutine schrittweise arbeitet. Es bleibt also nur noch ImplizitEuler als numerisch stabiles Verfahren zur Federsimulation.

Erin hat nun aber eine Möglichkeit im Dokument „*Soft Constraints - Erin Catto 2011*“ beschrieben, wie man eine Velocity-Constraint-Gleichung so ändert, dass es auch mit Semi Implicit Euler möglich ist eine stabile Feder zu simulieren.

Federsimulation über SoftConstraints

Immer wenn ich in diesen Abschnitt Erin zitiere, dann beziehe ich mich auf folgendes Dokument von ihm: Soft Constraints - Erin Catto 2011

Softconstraint-Gleichung aus der Feder-Kraftformel herleiten

In Teil 3 hatten wir eine Velocity-Constraint-Gleichung definiert, welche so aussieht:

$$J * V_2 + \frac{\beta}{h} C + \xi = 0$$

Das unbekannte V2 muss diese Gleichung erfüllen. Damit kann festgelegt werden, welchen Abstand und welche Relativgeschwindigkeit zwei Ankerpunkte zueinander haben sollen. Groß C ist der PositionError. Beta ist die PositionCorrectionRate (Baumgart-Stabilisierungsfaktor). Über C lege ich fest, um welche Distanz die Ankerpunkte in J-Richtung verschoben werden sollen. Über Xi lege ich fest, welche Relativgeschwindigkeit die Ankerpunkte haben sollen.

Die Korrektur der Geschwindigkeit erfolgt innerhalb von ein TimeStep und die Korrektur des PositionErros erfolgt in $\frac{1}{\beta}$ TimeSteps.

Wenn ich nun die beiden Ankerpunkte mit einer gedämpften Feder verbinden will, dann kann ich dazu eine so genannte SoftConstraint-Gleichung verwenden. Diese Gleichung beschreibt Erin auf Seite 36 so:

$$Jv + \frac{\beta}{h} C(x) + \gamma \frac{\lambda}{h} = 0$$

h=TimeStep; Lambda=Constraint-Impuls; Beta/Gamma=Konstanten

Achtung: Ich habe das Lambda noch durch h geteilt. Das mache ich, weil ich Lambda als Constraint-Impuls auffasse und Erin hat aber Lambda als Constraint-Kraft interpretiert. Um aber in diesen Dokument eine einheitliche Interpretation von Lambda zu erhalten sage ich immer: Lambda ist der Constraint-Impuls!

Die gedämpfte Federkraft, wo x die Auslenkung der Feder beschreibt, lautet:

$$F = -kx - c \frac{dx}{dt}$$

Siehe: https://en.wikipedia.org/wiki/Harmonic_oscillator#Damped_harmonic_oscillator

Wenn ich alles auf eine Seite bringe und durch c teile erhalte ich:

$$\frac{dx}{dt} + \frac{k}{c} x + \frac{F}{c} = 0$$

Wenn ich eine Feder betrachte, die nur in Richtung der X-Achse schwingt und wo am Nullpunkt ein Massestück befestigt ist, dann sieht die SoftConstraint dafür so aus, dass das J=1 weggelassen werden kann. Außerdem gilt für eine Constraint: $F = \frac{\lambda}{h}$ und beim PositionError wird wegen

SemiImplicitEuler x1 verwendet. Außerdem soll eine Constraint sich immer auf v2 beziehen so dass gilt: $\frac{dx}{dt} = v_2$

Die SoftConstraint-Gleichung erhalte ich, indem ich die Feder-Kraft-Formel nehme und dann die blauen Terme von oben durch die blauen Constraint-Terme (v2, x1, Lambda) ersetze:

$$\begin{array}{l} \frac{dx}{dt} + \frac{k}{c}x + \frac{F}{c} = 0 \quad \text{Harmonic oscillator} \\ \downarrow \\ v_2 + \frac{\beta}{h}x_1 + \frac{\gamma}{h}\lambda = 0 \quad \text{SoftConstraint} \end{array}$$

Wir wissen von der Federkraft-Gleichung, dass x die Auslenkung beschreibt und die Ableitung von x die Geschwindigkeit eines Massestücks, was an einer gedämpften Feder hängt. Bei der Federkraft-Formel steht vor dem x- und F-Term noch ein konstanter Faktor. Bei der SoftConstraint-Formel wurde das auch so gemacht. Es wurden aber nicht 1 zu 1 die konstanten Faktoren von der Federkraftformel zu der SoftConstraint übertragen sondern es wurden hier die neuen noch unbekannten Faktoren Beta und Gamma (grün markiert) eingeführt. Diese grünen Faktoren steuern die Steifigkeit und Dämpfung der Feder.

Die so erstellte SoftConstraint ist eine Gleichung, welche v2 nun so steuert, dass es den Anschein hat, dass die beiden Ankerpunkte von zwei Körpern dann mit einer gedämpften Feder verbunden sind.

$$v_2 + \frac{\beta}{h}x_1 + \gamma\frac{\lambda}{h} = 0 \rightarrow \text{SoftConstraint mit noch unbekannten Beta, Gamma und Lambda}$$

Wir beginnen mit der Herleitung für Beta und Gamma und danach kommt Lambda.

Herleitung für Beta und Gamma

Wenn ich mit Semi-Implicit-Euler eine Differentialgleichung lösen/simuliere will, dann nutze ich folgende beide Gleichungen in genau der Reihenfolge:

- 1: $v_2 = v_1 + h * \frac{F_1}{m}$ → F = Kraft welche sich aus den x1/v1-Variablen ergibt
 - 2: $x_2 = x_1 + h * v_2$
- h=TimeStep

Für Implicit-Euler brauche ich diese Gleichungen:

- 1: $v_2 = v_1 + h * \frac{F_2}{m}$ → F = Kraft welche sich aus den x2/v2-Variablen ergibt
- 2: $x_2 = x_1 + h * v_2$

Schritt 1: v2 ermitteln indem für F die Federkraft von Seite 11 eingesetzt wird

Wenn man eine Feder über Semi-Implicit-Euler (wird von unserer PhysikEngine verwendet) unter Verwendung der Federkraftformel simuliert, dann kann die Feder bei zu großer Steifigkeit explodieren. Wenn man aber Implicit-Euler zur Simulation nutzt, dann explodiert die Feder nicht. Wir wollen nun eine Formel für v2 finden, welche man erhält, wenn man eine Feder per Implicit-Euler simuliert. Dazu nutzen wir die Federkraft F von Erins Dokument Seite 11:

$$F = -c * \frac{dx}{dt} - k * x \rightarrow \text{Wegen Implicit Euler ersetzen wir } x \text{ mit } x_2 \text{ und } \frac{dx}{dt} \text{ mit } v_2$$

Unsere 3 Ausgangsgleichungen für die Herleitung von v2 sind:

$$1: v_2 = v_1 + h * \frac{F_2}{m}$$

$$2: x_2 = x_1 + h * v_2$$

$$3: F_2 = -c * v_2 - k * x_2 \rightarrow \text{Nimm Gleichung 1 und ersetze } F_2 \text{ mit der Federkraft}$$

$$1: v_2 = v_1 - \frac{h * c * v_2}{m} - \frac{h * k * x_2}{m} \rightarrow \text{Setze für } x_2 \text{ Gleichung 2 ein: } x_2 = x_1 + h * v_2$$

$$v_2 = v_1 - \frac{h * c * v_2}{m} - \frac{h * k * (x_1 + h * v_2)}{m} \rightarrow \text{Klammer aus multiplizieren}$$

$$v_2 = v_1 - \frac{h * c * v_2}{m} - \frac{h * k * x_1 + h^2 * k * v_2}{m} \rightarrow v_2\text{-Terme auf die linke Seite}$$

$$v_2 + \frac{h * c * v_2}{m} + \frac{h^2 * k * v_2}{m} = v_1 - \frac{h * k * x_1}{m} \rightarrow v_2 \text{ ausklammern}$$

$$v_2 * (1 + \frac{h * c}{m} + \frac{h^2 * k}{m}) = v_1 - \frac{h * k * x_1}{m} \rightarrow \text{durch die Klammer auf linker Seite teilen}$$

$$v_2 = \frac{v_1 - \frac{h * k * x_1}{m}}{1 + \frac{h * c}{m} + \frac{h^2 * k}{m}} \rightarrow \text{Formel für } v_2, \text{ wenn man eine Feder per Implicit Euler simuliert}$$

Schritt 2: Die Constraint-Kraft in die v2-Gleichung einsetzen

Wenn ich eine Differentialgleichung per Semi-Implicit-Euler unter Nutzung einer Softconstraint-Gleichung lösen will, dann nutze ich folgende 4 Ausgangsgleichungen:

$$1: v_2 = v_1 + h * \frac{F}{m}$$

$$2: x_2 = x_1 + h * v_2$$

$$3: v_2 + \frac{\beta}{h} * x_1 + \gamma * \frac{\lambda}{h} = 0 \rightarrow \text{Das ist unsere Softconstraint}$$

$$4: F = \frac{\lambda}{h}$$

Zuerst stelle ich Gleichung 3 nach der Constraint-Kraft $F = \frac{\lambda}{h}$ um und erhalte:

$$F = \frac{\lambda}{h} = \frac{-(v_2 + \frac{\beta}{h} * x_1)}{\gamma} = -\frac{v_2}{\gamma} - \frac{\beta * x_1}{h * \gamma}$$

Ich setze nun in Gleichung 1 die Constraintkraft ein:

$$v_2 = v_1 + h * \frac{-\frac{v_2}{\gamma} - \frac{\beta * x_1}{h * \gamma}}{m} = v_1 - \frac{h * v_2}{\gamma * m} - \frac{h * \beta * x_1}{h * \gamma * m} = v_1 - \frac{h * v_2}{\gamma * m} - \frac{\beta * x_1}{\gamma * m} \rightarrow v_2 \text{ auf die linke Seite}$$

$$v_2 + \frac{h*v_2}{\gamma*m} = v_1 - \frac{\beta*x_1}{\gamma*m} \rightarrow v2 \text{ aus klammern}$$

$$v_2 * \left(1 + \frac{h}{\gamma*m}\right) = v_1 - \frac{\beta*x_1}{\gamma*m} \rightarrow \text{Durch die Klammer auf der linken Seite teilen}$$

$$v_2 = \frac{v_1 - \frac{\beta}{\gamma*m} * x_1}{1 + \frac{h}{\gamma*m}} \rightarrow \text{Formel für v2, wenn man Semi-Implicit-Euler und SoftConstraint nutzt}$$

Per Parametervergleich Beta und Gamma ablesen

Wir haben eine Formel für v2 nun einmal dadurch erhalten, indem wir bei Implicit Euler die Federkraft eingesetzt haben und dann einmal dadurch, indem wir bei Semi-Implicit Euler die Soft-Constraint-Kraft verwendet haben.

Stellt man die beiden Gleichungen nebeneinander dann bekommt man folgendes Bild (Quelle: Seite 33):

Spring-Damper $v_2 = \frac{v_1 - \frac{hk}{m} x_1}{1 + \left[\frac{hc}{m} + \frac{h^2 k}{m} \right]}$	Soft Constraint $v_2 = \frac{v_1 - \frac{\beta}{m\gamma} x_1}{1 + \frac{h}{m\gamma}}$
Implicit Euler	Semi-implicit Euler

Man sieht dass die Formeln sehr ähnlich sind und wenn man die Bereiche mit gleicher Farbe gleichsetzt, dann erhält man:

$$1: \frac{h*k}{m} = \frac{\beta}{m*\gamma}$$

$$2: \frac{h*c}{m} + \frac{h^2*k}{m} = \frac{h}{m*\gamma}$$

Ich stelle nun die grüne Gleichung (1) nach Gamma um:

$$3: \gamma = \frac{\beta}{h*k}$$

Auch Gleichung (2) stelle ich nach Gamma um:

$$4: \gamma = \frac{h}{h*c + h^2*k} \quad \gamma = \frac{1}{c + h*k}$$

Wenn ich nun Gleichung 3 und 4 gleichsetze, kann ich sie nach Beta umstellen:

$$\frac{\beta}{h*k} = \frac{1}{c + h*k} \quad \beta = \frac{h*k}{c + h*k}$$

Durch den Parametervergleich der beiden v2-Formeln habe ich nun für Gamma und Beta folgende Formeln ermittelt (Siehe Seite 34/46):

$$\gamma = \frac{1}{c + hk}$$

$$\beta = \frac{hk}{c + hk}$$

h=TimeStep; c=Dämpfungskonstante; k=Federkonstante

Diese Parameter können wir nun in unsere Softconstraint-Gleichung einsetzen:

$$Jv + \frac{\beta}{h} C(x) + \gamma \frac{\lambda}{h} = 0$$

Wenn man sich die Gleichung ansieht, dann ist jetzt nur noch Lambda unbekannt. Dieses braucht man aber, um ein Korrekturimpulse anwenden zu können.

Die Ermittlung von Lambda bei Softconstraints

Quelle für diesen Abschnitt: 3D Constraint Derivations for Impulse Solvers - Marijn 2015 Seite 8/9.

Ausgangspunkt für die Herleitung des noch unbekannten Constraint-Impulses Lambda sind folgende beide Gleichungen:

Newton's zweites Gesetz in der Impulsform: (v -Strich= v_1 ; $v=v_2$)

$$\mathbf{p} = \mathbf{M}\dot{\mathbf{v}}\Delta t \approx \mathbf{M}(\mathbf{v} - \bar{\mathbf{v}}) = \mathbf{J}^T \boldsymbol{\lambda} \quad (2.8)$$

Die Softconstraint-Gleichung, bei der Lambda aktuell noch unbekannt ist:

$$\dot{c}(\mathbf{x}_a, \mathbf{q}_a, \mathbf{x}_b, \mathbf{q}_b) = \mathbf{J}\mathbf{v} + \gamma \cdot \frac{\boldsymbol{\lambda}}{\Delta t} + \frac{\beta \cdot \mathbf{c}}{\Delta t} = 0 \quad (2.9)$$

Gleichung 2.8 wird nach v umgestellt:

$$\begin{aligned} \mathbf{v} - \bar{\mathbf{v}} &= \mathbf{M}^{-1} \mathbf{J}^T \boldsymbol{\lambda} \\ \mathbf{v} &= \mathbf{M}^{-1} \mathbf{J}^T \boldsymbol{\lambda} + \bar{\mathbf{v}} \end{aligned} \quad (2.10)$$

Das v in Gleichung (2.9) wird durch die rechte Seite von Gleichung (2.10) ersetzt und dann wird Lambda freigestellt:

$$\begin{aligned} \dot{c}(\dots) &= \mathbf{J} \left(\mathbf{M}^{-1} \mathbf{J}^T \boldsymbol{\lambda} + \bar{\mathbf{v}} \right) + \gamma \cdot \frac{\boldsymbol{\lambda}}{\Delta t} + \frac{\beta \cdot \mathbf{c}}{\Delta t} \\ &= \mathbf{J} \mathbf{M}^{-1} \mathbf{J}^T \boldsymbol{\lambda} + \mathbf{J} \bar{\mathbf{v}} + \frac{\gamma}{\Delta t} \cdot \boldsymbol{\lambda} + \frac{\beta \cdot \mathbf{c}}{\Delta t} \\ &= \left(\mathbf{J} \mathbf{M}^{-1} \mathbf{J}^T + \frac{\gamma}{\Delta t} \right) \cdot \boldsymbol{\lambda} + \mathbf{J} \bar{\mathbf{v}} + \frac{\beta \cdot \mathbf{c}}{\Delta t} \end{aligned} \quad (2.11)$$

Die Klammer vor Lambda wird zur neu definierten Konstante K zusammengefasst:

$$K = \mathbf{J} \mathbf{M}^{-1} \mathbf{J}^T + \frac{\gamma}{\Delta t} \quad (2.12)$$

Somit erhalten wir die Gleichung (2.13) wo ich nur noch durch K dividieren muss um Lambda zu erhalten:

$$K \boldsymbol{\lambda} = -\mathbf{J} \bar{\mathbf{v}} - \frac{\beta \cdot \mathbf{c}}{\Delta t} \quad (2.13)$$

Den Impuls über Gleichung (2.13) ausrechnen

Wenn ich Lambda so ermitte und den Impuls anwende, dann funktioniert die Warm-Start-Funktion nicht mehr. So sieht die Lambda-Berechnung und Impulsanwendung aus. Der obere Teil ist aus der DistanceJointConstraint-Klasse. Dort berechne ich Beta, Gamma, K und all die Terme aus der (2.13)-Formel. Im unteren Teil wende ich den Impuls an. Aus Schritt 4 von der Distancejoint-Constraint-Herleitung wissen wir noch: $\mathbf{J} \mathbf{M}^{-1} \mathbf{J}^T = m_a^{-1} + \mathbf{A}_a^2 \mathbf{I}_a^{-1} + m_b^{-1} + \mathbf{A}_b^2 \mathbf{I}_b^{-1}$

3D Constraint Derivations
for Impulse Solvers - Marijn
2015 Seite 8

$$\mathbf{K}\lambda = -\mathbf{J}\bar{\mathbf{v}} - \frac{\beta \cdot \mathbf{c}}{\Delta t} \quad (2.13)$$

$$\gamma = \frac{1}{c + hk}$$

$$\beta = \frac{hk}{c + hk}$$

Soft Constraints -
Erin Catto 2011
Seite 34

```

53     float invMass = B1.InverseMass + B1.InverseInertia * r1crossN * r1crossN + B2.InverseMass + B2.InverseInertia * r2crossN * r2crossN;
54
55     float h = data.Dt;
56
57     //So könnte man laut der Formel (2.13) aus "3D Constraint Derivations for Impulse Solvers - Marijn 2015" auch den Impuls berechnen
58     float gamma = 1 / (joint.Damping + h * joint.Stiffness);
59     float beta = (h * joint.Stiffness) / (joint.Damping + h * joint.Stiffness);
60     this.K = invMass + gamma / h;
61     this.BetaDeltaT = beta * (Length - joint.Length) / h;
62     this.GammaDt = gamma / h;
63
64     //Schritt 4: Finde per PGS Relative-Kontaktpunktgeschwindigkeitswerte, welche dem Bias-Wert entsprechen
65     for (int i = 0; i < settings.IterationCount; i++)
66     {
67         foreach (var c in constraints)
68         {
69             //VelocityAtContactPoint = V + mAngularVelocity cross R
70             Vector2D v1 = c.B1.Velocity + new Vector2D(-c.B1.AngularVelocity * c.R1.Y, c.B1.AngularVelocity * c.R1.X);
71             Vector2D v2 = c.B2.Velocity + new Vector2D(-c.B2.AngularVelocity * c.R2.Y, c.B2.AngularVelocity * c.R2.X);
72             Vector2D relativeVelocity = v2 - v1;
73
74             // Relative velocity in Force direction
75             float velocityInForceDirection = relativeVelocity * c.ForceDirection;
76
77             var d = (c as DistanceJointConstraint);
78             float impulse = (-velocityInForceDirection - d.BetaDeltaT) / d.K / settings.IterationCount;
79
80             c.AccumulatedImpulse += impulse;
81
82             ApplyImpulse(c, impulse);
83
84             c.SaveImpulse();
85         }
86     }

```

$$\mathbf{K} = \mathbf{JM}^{-1}\mathbf{J}^T + \frac{\gamma}{\Delta t} \quad (2.12)$$

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Examples/ResolverHelper2.cs
Hinweis zum Bild: Das c im roten Kasten links bei Gleichung (2.13) steht für den Position-Error, also currentLength-joint.Length. Im roten Kasten rechts: c=Federdämpfung; k=Federkonstante; h=TimeStep

Wenn ich Warmstart nutze, dann bekommt die Feder immer mehr Energie und kommt nicht zur Ruhe oder schwingt gleichmäßig. Als Grund gibt es wage Hinweise von Erin und Marijn.

Erin schreibt dazu unter: <https://pybullet.org/Bullet/phpBB3/viewtopic.php?f=4&t=1354#p5044>

$$J * (v1 + invM * JT * lambda) + softness * lambda + bias = 0$$

Now collect coefficients of lambda:

$$(J * invM * JT + softness) * lambda = - J * v1 - bias$$

Now we define:

$$K = J * invM * JT + softness$$

and $M_{eff} = invK$ is the effective mass.

Now keep in mind that v1 contains the velocities for both bodies:

$$v1 = column_vector(vb1, omegab1, vb2, omegab2)$$

Now here is where things get a bit hairy. Each iteration we are not computing the whole impulse, we are computing an increment to the impulse and we are updating the velocity. Also, as we solve each constraint we get a perfect v2, but then some other constraint will come along and mess it up. So we want to patch up the constraint while acknowledging the accumulated impulse and the damaged velocity. To help with that we use P for the accumulated impulse and dP as the update. Mathematically we have:

$$M * (v2new - v2damaged) = JT * dP$$

$$J * v2new + softness * (P + dP) + bias = 0$$

Marijn schreibt als Grund, warum er Gleichung (2.13) nicht direkt nutzt:

3D Constraint Derivations for Impulse Solvers - Marijn 2015 Seite 8:

We define \mathbf{K} :

$$\mathbf{K} = \mathbf{JM}^{-1}\mathbf{J}^T + \frac{\gamma}{\Delta t} \quad (2.12)$$

Which gives:

$$\mathbf{K}\lambda = -\mathbf{J}\bar{\mathbf{v}} - \frac{\beta \cdot \mathbf{c}}{\Delta t} \quad (2.13)$$

We calculate the delta impulse to satisfy the constraint, instead of calculating the whole impulse in each iteration. We use λ as the accumulated impulse and $\Delta\lambda$ as the incremental impulse update. This gives us the following formula:

$$\mathbf{M}(\mathbf{v}_{\text{new}} - \mathbf{v}_{\text{damaged}}) = \mathbf{J}^T \Delta\lambda \quad (2.14)$$

Durch ausprobieren habe ich ja gesehen, dass ich (2.13) nicht direkt nutzen darf, wenn ich auch Warmstart noch nutzen will. Auch ist es komisch, dass ich den Impuls durch die Iterationcount dividieren muss. Normalerweise macht man das sonst auch nicht.

Ich folge nun weiter der Erklärung von Marijn/Erin und zeige nun, wie man eine Impulsformel für Softconstraints findet, welche mit und ohne Warmstart funktioniert.

Den Impuls über Gleichung (2.17) ausrechnen

Wenn man ein Impuls dadurch ausrechnen will, indem man mit den Sequentielle-Impulse-Algorithmus arbeitet, dann startet man mit ein V1-Start-Wert und ein AccumulatedImpuls von 0 (oder vom vorherigen TimeStep) und dann ermittelt man mit jeder Iterationsschleife ein Zwischenimpuls, welcher das aktuelle v immer mehr Richtung v2 bewegt.

Vnew ist hier der v2-Wert, wo man hin will. Vdamaged ist das aktuelle V. Delta-Lambda ist der Zwischenimpuls, welcher innerhalb der Iterationsschleife berechnet wird.

Für den Zwischenimpuls gilt laut der „p=m*v“-Regel:

$$\mathbf{M}(\mathbf{v}_{\text{new}} - \mathbf{v}_{\text{damaged}}) = \mathbf{J}^T \Delta\lambda \quad (2.14)$$

Ich stelle (2.14) nach vnew um:

$$\mathbf{v}_{\text{new}} = \mathbf{M}^{-1}\mathbf{J}^T \Delta\lambda + \mathbf{v}_{\text{damaged}} \quad (2.15)$$

Jetzt brauchen wir noch die SoftConstraint-Gleichung. Lambda ist der AccumulatedImpuls und DeltaLambda der (Zwischen-)Impuls aus der aktuellen Iteration.

$$\dot{c}(\dots) = \mathbf{J}\mathbf{v}_{\text{new}} + \gamma \cdot \frac{\lambda + \Delta\lambda}{\Delta t} + \frac{\beta \cdot \mathbf{c}}{\Delta t} = 0 \quad (2.16)$$

Wir setzen nun in Gleichung (2.16) für vnew (2.15) ein und stellen nach Delta-Lambda um:

$$\begin{aligned} \mathbf{J} \left(\mathbf{M}^{-1}\mathbf{J}^T \Delta\lambda + \mathbf{v}_{\text{damaged}} \right) + \gamma \cdot \frac{\lambda + \Delta\lambda}{\Delta t} + \frac{\beta \cdot \mathbf{c}}{\Delta t} &= 0 \\ \mathbf{JM}^{-1}\mathbf{J}^T \Delta\lambda + \mathbf{J}\mathbf{v}_{\text{damaged}} + \frac{\gamma}{\Delta t} \cdot \lambda + \frac{\gamma}{\Delta t} \cdot \Delta\lambda + \frac{\beta \cdot \mathbf{c}}{\Delta t} &= 0 \\ \left(\mathbf{JM}^{-1}\mathbf{J}^T + \frac{\gamma}{\Delta t} \right) \cdot \Delta\lambda &= -\mathbf{J}\mathbf{v}_{\text{damaged}} - \frac{\gamma}{\Delta t} \cdot \lambda - \frac{\beta \cdot \mathbf{c}}{\Delta t} \\ \mathbf{K}\Delta\lambda &= -\mathbf{J}\mathbf{v}_{\text{damaged}} - \frac{\gamma}{\Delta t} \cdot \lambda - \frac{\beta \cdot \mathbf{c}}{\Delta t} \\ \Delta\lambda &= \frac{-\mathbf{J}\mathbf{v}_{\text{damaged}} - \frac{\gamma}{\Delta t} \cdot \lambda - \frac{\beta \cdot \mathbf{c}}{\Delta t}}{\mathbf{K}} \end{aligned} \quad (2.17)$$

So sieht das ganze im Code aus. Der oberere Teil ist wieder aus dem DistanceJointConstraint-Konstruktor und der untere vom SI-Solver.

$$\Delta \lambda = \frac{-\mathbf{J} \mathbf{v}_{\text{damaged}} - \frac{\gamma}{\Delta t} \cdot \lambda - \frac{\beta \cdot \mathbf{c}}{\Delta t}}{\mathbf{K}} \quad (2.17)$$

3D Constraint Derivations for Impulse
Solvers - Marijn 2015 Seite 9

$$\gamma = \frac{1}{c + hk}$$

$$\beta = \frac{hk}{c + hk}$$

Soft Constraints -
Erin Catto 2011
Seite 34

```

53 float invMass = B1.InverseMass + B1.InverseInertia * r1crossN * r1crossN + B2.InverseMass + B2.InverseInertia * r2crossN * r2crossN;
54
55 float h = data.Dt; 
$$\mathbf{JM}^{-1} \mathbf{J}^T = m_a^{-1} + \mathbf{A}_a^2 \mathbf{I}_a^{-1} + m_b^{-1} + \mathbf{A}_b^2 \mathbf{I}_b^{-1}$$
 Schritt 4 - Distancejoint-Constraint
56 //So könnte man laut der Formel (2.13) aus "3D Constraint Derivations for Impulse Solvers - Marijn 2015" auch den Impuls berechnen
57 float gamma = 1 / (joint.Damping + h * joint.Stiffness);
58 float beta = (h * joint.Stiffness) / (joint.Damping + h * joint.Stiffness);
59 this.K = invMass + gamma / h;
60 this.BetaCDeltaT = beta * (Length - joint.Length) / h;
61 this.GammaDt = gamma / h;
62
63 //Schritt 4: Finde per PGS Relative-Kontaktpunktgeschwindigkeitswerte, welche dem Bias-Wert entsprechen
64 for (int i = 0; i < settings.IterationCount; i++)
65 {
66     foreach (var c in constraints)
67     {
68         //VelocityAtContactPoint = V + mAngularVelocity cross R
69         Vector2D v1 = c.B1.Velocity + new Vector2D(-c.B1.AngularVelocity * c.R1.Y, c.B1.AngularVelocity * c.R1.X);
70         Vector2D v2 = c.B2.Velocity + new Vector2D(-c.B2.AngularVelocity * c.R2.Y, c.B2.AngularVelocity * c.R2.X);
71         Vector2D relativeVelocity = v2 - v1;
72
73         // Relative velocity in Force direction
74         float velocityInForceDirection = relativeVelocity * c.ForceDirection;
75
76         // (2.17)-Formel aus "3D Constraint Derivations for Impulse Solvers - Marijn 2015"
77         var d = (c as DistanceJointConstraint);
78         float impulse = (-velocityInForceDirection - d.GammaDt * c.AccumulatedImpulse - d.BetaCDeltaT) / d.K;
79
80         c.AccumulatedImpulse += impulse;
81
82         ApplyImpulse(c, impulse);
83
84         c.SaveImpulse();
85     }
86 }

```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Examples/ResolverHelper2.cs

Mit dieser (2.17)-Formel funktioniert nun die Simulation einer Feder mit und ohne Warmstart. Es gibt aber noch zwei Punkte, die es im Zusammenhang mit der Federsimulation zu klären gibt: Die Federdämpfung c und Federkonstante k müssen von außen vorgegeben werden um festzulegen, wie steif und gedämpft die Feder ist. Das sind recht abstrakte Variablen. Es gibt nutzerfreundlichere Variablen zur Federkonfiguration. Das zweite Thema, was dann noch kommen soll, ist die Simulation einer Feder ohne Dämpfung.

Nutzerfreundlichere Konstanten zur Federkonfiguration

Wenn man die Federgleichung (Differentialgleichung 2. Ordnung mit Anfangswert und konstanten Koeffizienten) (z.B. per https://de.wikipedia.org/wiki/Charakteristische_Gleichung) löst, dann erhält man laut Wikipedia (https://en.wikipedia.org/wiki/Harmonic_oscillator) für die Gleichung:

$$F = ma = m \frac{d^2x}{dt^2} = m\ddot{x} = -kx.$$

Die Lösung:

$$x(t) = A \cos(\omega t + \varphi)$$

mit

$$\omega = \sqrt{\frac{k}{m}}$$

Omega ist also dadurch entstanden, indem man die Konstanten in der Lösung der Differentialgleichung zusammengefasst hat.

In diesen Wikipediaartikel steht auch, dass wenn man die gedämpfte Schwingung löst, dann erhält man für

$$F = -kx - c \frac{dx}{dt} = m \frac{d^2x}{dt^2}$$

die Konstanten:

- $\omega_0 = \sqrt{\frac{k}{m}}$ is called the "undamped angular frequency" of the oscillator",
- $\zeta = \frac{c}{2\sqrt{mk}}$ is called the "damping ratio".

$$f = \frac{\omega_0}{2\pi}$$

Erin schreibt unter Soft Constraints - Erin Catto 2011 auf Seite 11

$$\frac{d^2x}{dt^2} + 2\zeta\omega \frac{dx}{dt} + \omega^2 x = 0$$

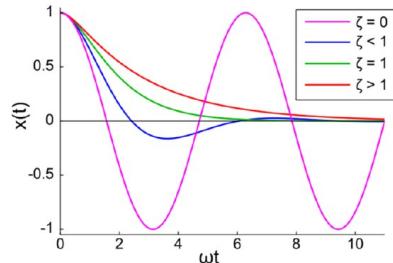
$$2\zeta\omega = \frac{c}{m} \quad \omega^2 = \frac{k}{m}$$

Und auf Seite 45 hat er nach c und k umgestellt:

$$k = m_{eff}\omega^2$$

$$c = 2m_{eff}\zeta\omega$$

Die Idee ist nun, dass der Nutzer zur Beschreibung einer Feder nicht die Konstanten c und k festlegt, sondern er überlegt sich mit welcher Frequenz f soll die Feder schwingen und er überlegt sich eine Dämpfungszahl Zeta, welche von 0 bis 1 geht. Seite 13: So sieht die Schwingung unter Verwendung verschiedener Dämpfungszahlen Zeta aus:



Unter https://github.com/erincatto/box2d/blob/main/src/dynamics/b2_joint.cpp

sieht man wie Erin die Dämpfungszahl Zeta und die Frequenz f in die Federkonstante k (Stiffness bei ihm genannt) und die Federdämpfung (damping genannt) umrechnet. Ich habe sein Code bei mir an folgender Stelle verwendet:

```

14  static internal class JointSoftnessHelper
15  {
16      // Verweis
17      public static void LinearStiffness(float frequencyHertz, float dampingRatio, IRigidBody bodyA, IRigidBody bodyB, out float stiffness, out float damping)
18      {
19          float massA = bodyA.InverseMass != 0 ? 1.0f / bodyA.InverseMass : 0;
20          float massB = bodyB.InverseMass != 0 ? 1.0f / bodyB.InverseMass : 0;
21          float mass;
22          if (massA > 0 && massB > 0)
23          {
24              mass = massA * massB / (massA + massB);
25          }
26          else if (massA > 0)
27          {
28              mass = massA;
29          }
29          else
30          {
31              mass = massB;
32          }
33          // Mit der 1000-Division rechne ich die Frequenz-Pro-Millisekundenzahl in eine Frequenz-Pro-Sekunden-Zahl um
34          float omega = 2 * (float)Math.PI * frequencyHertz / 1000;
35          stiffness = mass * omega * omega; // /k
36          damping = 2 * mass * dampingRatio * omega; // /c
37      }
}

```

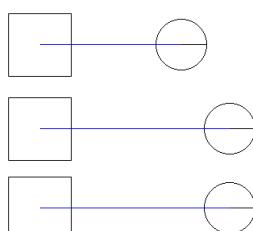
Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/JointSoftnessHelper.cs

Da meine Physicengine die Zeit nicht in Sekunden sondern in Millisekunden als Input in der TimeStep-Methode bekommt, muss ich auch die Frequenz von Schwingungen pro Sekunde in Schwingungen pro Millisekunde umrechnen, indem ich durch 1000 teile. Hängt die Feder an ein unbeweglichen Objekt, dann wird für die mass-Variable die Masse des beweglichen Objektes genommen. Wenn beide Objekte beweglich sind, dann wird auf Zeile 23 die effektive Masse errechnet. Ziel ist es, dass der Nutzer für die Feder festlegen soll, wie oft sie in der Sekunde schwingt ohne sich darüber Gedanken machen zu müssen, welches Gewicht an der Masse hängt. Hängt dort viel Gewicht, dann führt das zu einer höheren Steifigkeit k und zu höherer Dämpfung c. Die Feder passt sich also automatisch an das verwendete Gewicht (zum Definitionszeitpunkt der Feder) an.

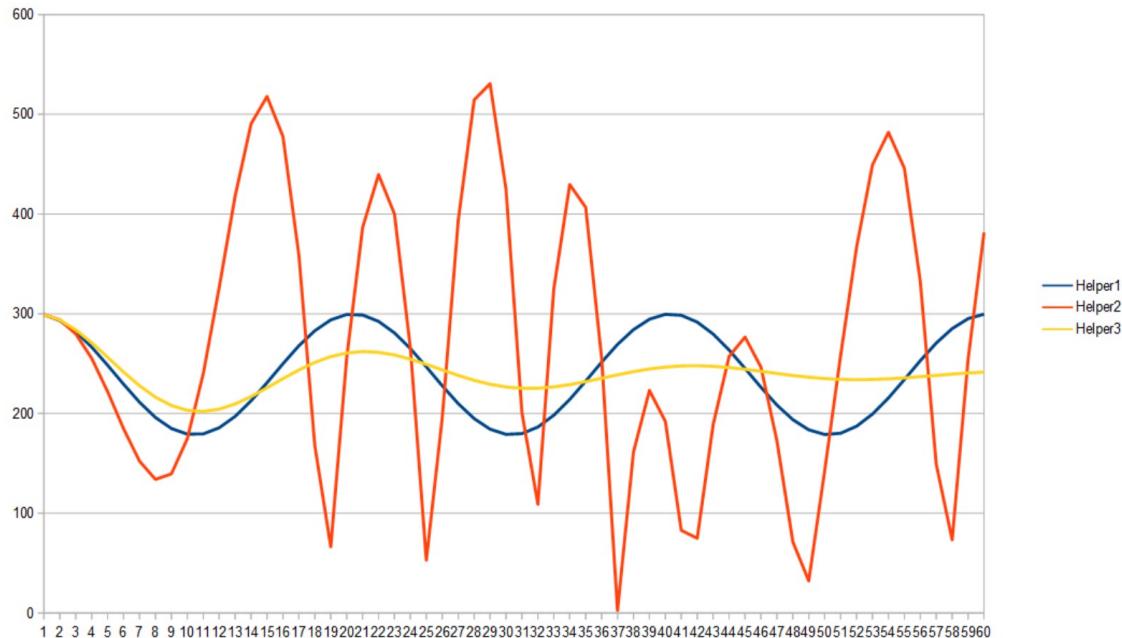
Ziel von diesen Abschnitt war zu erklären, woher die Formeln zur Umrechnung von Zeta und frequenz nach c und k her kommen. Sie kommen also nur daher, dass jemand die Differentialgleichungen gelöst hat und die Konstanten zusammengefasst hat.

Feder ohne Dämpfung

Ich habe folgende Szene gebaut wo es keine Schwerkraft gibt. Es gibt 3 Würfel mit unendlicher Masse und dort hängt per Feder eine Kugel dran. Die oberste Feder hat ein Dampingratio-Wert von 0 (keine Dämpfung) und alle Federn haben eine Frequenz von 1. Da ich mit 50ms pro Timestep arbeite, heißt dass, eine Feder braucht 20 TimeSteps für einen Zyklus.



Wenn ich nun die 3 ResolverHelper-Variationen zur Simulation von 60 TimeSteps nutze, dann bekomme ich folgende Auslenkungswerte für die ungedämpfte Feder:



Siehe: PhysicEngine.UnitTests/Constraints/DistanceJointTest.cs

Helper1 = Modifiziert die Body.Force-Property laut Federkraftformel

Helper2 = Nutzt eine DistanceJoint und SoftConstraint mit Gleichung (2.13)

Helper3 = Nutzt eine DistanceJoint und SoftConstraint mit Gleichung (2.17)

Helper1 verhält sich so wie erwartet. Ich nutze eine Dämpfung von 0 und erwarte eine Sinuskurve als Auslenkungsfunktion. Bei Helper2 wurde ja bereits gesagt, dass das nicht gehen kann, da Formel (2.13) laut Aussage von Erin und Marijn ja der Impuls und nicht wie vom SI-Solver benötigt, der Delta-Impuls ist. Bei Helper3 nutze ich nun die richtige Formel und simuliere ein Distanzjoint über eine SoftConstraint. Die Frage ist nun, warum dort die Kurve eine gedämpfte Schwingung zeigt obwohl ich doch eine DampingRatio von 0 verwende?

Um das Problem besser zu verstehen, nutze ich wieder die SpringMassTest-Klasse, welche ich schon bei der Untersuchung der explodierenden Feder genutzt habe. Dort untersuche ich eine ungedämpfte Schwingung von einem Kilo schweren Gewicht, was mit einer Schwingung pro 1000 ms schwingen soll. Es startet mit einer Auslenkung von 100 und soll im Bereich von -100 bis +100 die ganze Zeit sich bewegen. Ich nutze wieder die explizite Formel und die numerischen Integratoren, welche ich schon bei der Federexplosion genutzt habe. Die SpringMassTest-Klasse wird nun noch um ein Semi-Implizite Euler-Verfahren erweitert, welches eine SoftConstraints nutzt.

Feder ohne Dämpfung – Simulation über Semi-Implicit-Euler mit Softconstraint

Für die Federsimulation per Semi-Implicit-Euler unter Nutzung einer SoftConstraint nutze ich folgende Formeln aus *Soft Constraints - Erin Catto 2011 auf Seite 30*:

$$\begin{aligned}
 v_2 &= v_1 + \frac{h}{m} \lambda \\
 x_2 &= x_1 + h v_2 \\
 v_2 + \frac{\beta}{h} x_1 + \gamma \lambda &= 0
 \end{aligned}$$

Achtung: Das Lambda steht hier für die Constraint-Kraft. Das sieht man daran, dass wenn man die erste Gleichung über die allgemeinere Formel $v_2 = v_1 + h * \frac{F}{m}$ ausdrückt, dann sieht man per Parametervergleich, dass F hier Lambda entspricht. Wenn wir aber Lambda als Impuls auffassen wollen, dann gilt $\lambda = F * h$ und $F = \frac{\lambda}{h}$. Wir schreiben somit Gleichung 1 in der allgemeinen Form und bei Gleichung 3 ersetzen wir den Kraft-Lambda-Wert durch den Impuls-Lambda-Wert.

$$1: \quad v_2 = v_1 + h * \frac{F}{m}$$

$$2: \quad x_2 = x_1 + h * v_2$$

$$3: \quad v_2 + \frac{\beta}{h} x_1 + \gamma \frac{\lambda}{h} = 0 \rightarrow \text{Softconstraint-Gleichung mit Constraintimpuls Lambda}$$

Um nun mit den Gleichungen arbeiten zu können brauchen wir Gleichung 1 ohne Lambda oder F-Bezug. Wir stellen dazu Gleichung 3 nach Lambda um:

$$\lambda = -(v_2 + \frac{\beta}{h} x_1) \frac{h}{\gamma} \rightarrow \text{Ersetze } \lambda \text{ mit } F * h$$

$$F * h = -(v_2 + \frac{\beta}{h} x_1) \frac{h}{\gamma} \rightarrow \text{Teile beide Seiten durch } h$$

$$F = -(v_2 + \frac{\beta}{h} x_1) \frac{1}{\gamma} \rightarrow \text{Das ist die SoftConstraint-Kraft}$$

Wir haben nun die SoftConstraint-Kraft ermittelt und setzen diese für F in Gleichung 1 ein:

$$v_2 = v_1 - h(v_2 + \frac{\beta}{h} x_1) \frac{1}{(\gamma m)} \rightarrow \text{Klammer ausmultiplizieren}$$

$$v_2 = v_1 - v_2 * \frac{h}{(\gamma m)} - \frac{\beta}{h} x_1 * \frac{h}{(\gamma m)} \rightarrow v2-Terme auf die linke Seite bringen$$

$$v_2 + v_2 * \frac{h}{(\gamma m)} = v_1 - \frac{\beta}{h} x_1 * \frac{h}{(\gamma m)} \rightarrow v2 ausklammern; h auf der rechten Seite kürzen$$

$$v_2(1 + \frac{h}{\gamma m}) = v_1 - \frac{\beta x_1}{\gamma m} \rightarrow \text{Durch die Klammer dividieren}$$

$$v_2 = \frac{v_1 - \frac{\beta x_1}{\gamma m}}{1 + \frac{h}{\gamma m}} \rightarrow \text{Semi-Implicit-Softconstraint-Formel für v2}$$

Um nun eine ungedämpfte Schwingung zu simulieren, nutzen wir nun diese beiden Gleichungen:

$$1: \quad v_2 = \frac{v_1 - \frac{\beta x_1}{\gamma m}}{1 + \frac{h}{\gamma m}}$$

$$2: \quad x_2 = x_1 + h * v_2$$

Außerdem hatten wir im Abschnitt „Herleitung für Beta und Gamma“ bereits noch diese Formeln für Beta und Gamma ermittelt:

$$\gamma = \frac{1}{c + hk}$$

$$\beta = \frac{hk}{c + hk}$$

Für die Federkonstante k hatten wir im Abschnitt „Nutzerfreundlichere Konstanten zur Federkonfiguration“ $k = \omega^2 * m$ ermittelt.

So sieht nun die Funktion aus, welche per Semi-Implicit Euler simuliert, wo die Federkraft per SoftConstraint-Kraft bestimmt wird:

```

149 public float[] UndampedSpringWithSemiImplicitEulerAndSoftConstraint(float frequency = 1, float xStart = 100, float m = 1, float h
150 {
153     float omega = 2 * (float)Math.PI * frequency / 1000;
154     float k = m * omega * omega;
155     float dampingRatio = 0;
156     float c = 2 * m * dampingRatio * omega;
157     float gamma = 1 / (c + h * k);
158     float beta = (h * k) / (c + h * k);
159
160     float x = xStart;
161     float v = 0;
162     float v2 = v;
163
164     List<float> xValues = new List<float>();
165     for (float t = 0; t <= 3000; t += h)
166     {
167         v2 = (v - beta * x / (gamma * m)) / (1 + h / (gamma * m)); //Semi-Implicit-Euler mit Softconstraint
168         float x2 = x + h * v2;
169         x = x2;
170         v = v2;
171
172         xValues.Add(x);
173     }
}

```

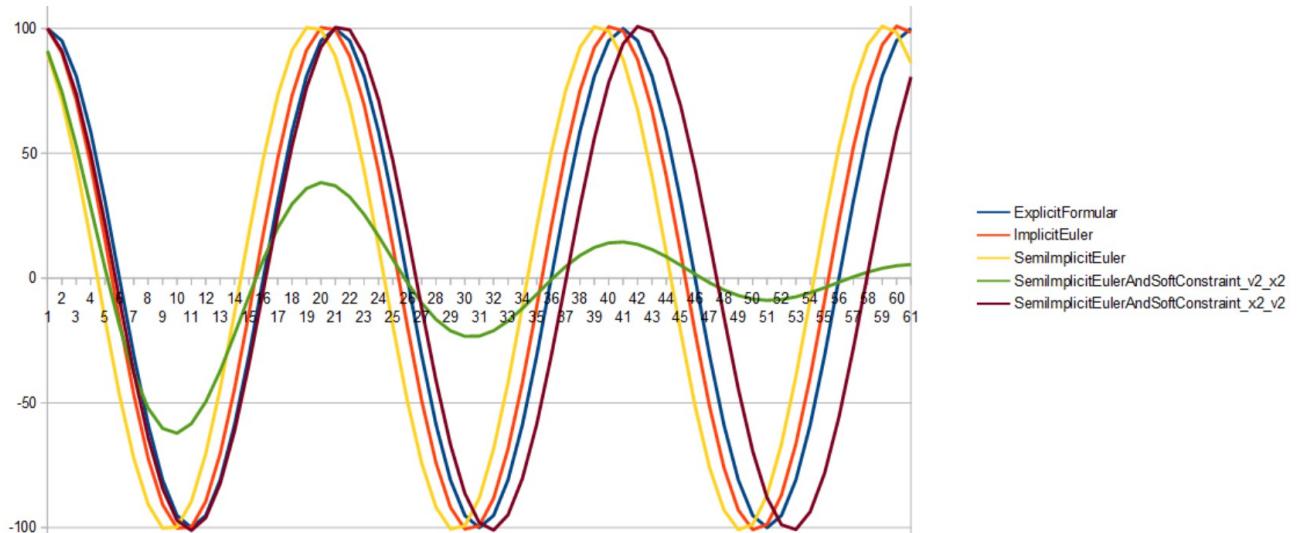
Soft Constraint

$$v_2 = \frac{v_1 - \frac{\beta}{m\gamma} x_1}{1 + \frac{h}{m\gamma}}$$

Semi-implicit Euler

Soft Constraints - Erin Catto 2011 Seite 32

Um nun zu sehen, ob die ganzen Funktionen aus SpringMassTest.cs ungedämpft sind visualisiere ich sie: Man sieht, dass alle Schwingungen außer die grüne Linie ungedämpft sind.



Die grüne Linie entspricht der SoftConstraint-Funktion, wo zuerst das v2 und danach dann das x2 ermittelt wurde (Siehe im Quelltext-Bild Zeile 167/168). Wenn diese beiden Zeilen aber vertauscht werden, dann ist die Kurve wieder ungedämpft (Siehe braune Kurve).

So wird die braune Kurve ermittelt:

```

183     x2 = x + h * v2;
184     v2 = (v - beta * x / (gamma * m)) / (1 + h / (gamma * m)); //Semi-Implicit-Euler mit Softconstraint

```

In der v2-Formel taucht kein x2 auf. Nur in der x2-Formel taucht v2 auf. Es ist also erlaubt, dass ich die braune Kurve auch dadurch erzeuge, indem ich erst v2 berechne und in der x2-Formel nutze ich dann den alten v2-Wert:

```

191     float v2Old = v2;
192     v2 = (v - beta * x / (gamma * m)) / (1 + h / (gamma * m)); //Semi-Implicit-Euler mit Softconstraint
193     x2 = x + h * v2Old;

```

Innerhalb der TimeStep-Methode von der PhysicScene-Klasse erfolgt erst die Berechnung von v2 unter Nutzung von SoftConstraints und danach dann erst x2. Die Frage ist, was passiert, wenn ich in der TimeStep-Methode zuerst bei Schritt 3 das neue v2 berechne und dann bei Schritt 4 den v2Old-Wert nutze?

```

108     public void TimeStep(float dt)
109     {
110         //1. Weise der externen Kraft einen Wert zu (Der Nutzer darf vor jedem TimeStep-Aufruf auch selber Kraftwerte setzen)
111         if (this.HasGravity)
112             AddGravityForceForAllBodies();
113
114         //2. Ermittle alle Kollisionspunkte
115         var collisionsFromThisTimeStep = CollisionHelper.GetAllCollisions(this.bodies);
116         if (collisionsFromThisTimeStep.Any())
117             this.CollisionOccurred?.Invoke(this, collisionsFromThisTimeStep);
118
119         var oldV = this.bodies.Select(x => x.Velocity).ToArray();
120         var oldA = this.bodies.Select(x => x.AngularVelocity).ToArray();
121
122         //3. Constraint-Kraft aufgrund der aktuellen Geschwindigkeit berechnen und zusammen mit der externen Kraft anwenden
123         if (collisionsFromThisTimeStep.Any() || this.joints.Any()) //Abfrage: Gibt es Constraints?
124             this.impulseResolver.Resolve(this.bodies, this.joints, collisionsFromThisTimeStep, dt, this.settings); //Wende Constraint-Kraft + Externe Kraft an
125         else
126             ApplyExternalForces(dt); //Körper ohne Beschränkung bewegen (Wende nur die externe Kraft an)
127
128         //4. Geschwindigkeit verändert die Position
129         //MoveBodiesAndSetForceToZero(dt);
130
131         //Nutze oldV und oldA um die neue Position zu bestimmen um somit Semi-Implicit-Euler zu simulieren, wo erst x2 und dann v2 errechnet wird
132         for (int i=0;i<bodies.Count;i++)
133         {
134             var body = bodies[i];
135             body.MoveCenter(dt * oldV[i]);
136             body.Rotate(dt * oldA[i]);
137
138             //Resette die externe Kraft
139             body.Force = new Vector2D(0, 0);
140             body.Torque = 0;
141         }
142
143         foreach (var joint in this.joints)
144         {
145             joint.UpdateAnchorPoints();
146         }
147     }

```

Hier funktioniert nun die unendliche Schwingung. Aber leider funktionieren dann andere Tests nicht mehr. So sinken dann Objekte in den Boden ein oder der JumpingBall-Test mit unendlichen Sprüngen geht nicht.

Feder ohne Dämpfung – Semi-Implicit-Euler - Zusammenfassung

Meine Physikengine nutzt Semi-Implicit-Euler so dass ich versuche mit diesen Verfahren eine ungedämpfte unendliche Federschwingung zu simulieren.

Durch probieren bin ich zu folgenden Erkenntnissen gekommen:

- Semi-Implizit-Euler ohne Softconstraints und der Reihenfolge: v2;x2 schwingt unendlich. Die Feder kann aber explodieren.
- Semi-Implizit-Euler mit Softconstraint und der Reihenfolge x2;v2 schwingt auch unendlich. Hier sinken aber Objekte in den Boden ein und der JumpingBall-Test mit Restitution=1 geht nicht.
- Semi-Implizit-Euler mit Softconstraint und der Reihenfolge v2;x2 schwingt nicht unendlich. Dafür explodiert die Feder nicht.

Da bei Variante 2 die Objekte im Boden versinken, ist diese Option ein NoGo. Ich habe also nur die Wahl zwischen Variante 1, wo die Feder explodieren kann wenn die Steifigkeit zu hoch ist oder zwischen Variante 3, wo die Feder niemals explodiert aber wo keine unendliche Federschwingung möglich ist. Ich nutze Variante 3, da es Federn ohne Reibung nicht gibt (Hypothese von mir!).

Meines Wissens arbeitet Box2D auch mit Variante 3. Laut diesen Beitrag scheinen sie auch das Problem zu haben:

<https://gamedev.stackexchange.com/questions/98679/in-box2d-how-do-i-create-an-infinitely-oscillating-spring>

In Box2D, how do I create an infinitely oscillating spring?

Asked 8 years, 3 months ago Modified 8 years, 3 months ago Viewed 1k times

- How do I create an ideal spring that can oscillate indefinitely using Box2D?
- 3 A `b2DistanceJoint` works almost perfectly, but it eventually slows to a stop, even with damping set to 0. I think it is because `b2DistanceJoint` is not meant for creating springs, but for preserving the distance between two bodies by correction the violated distance with a spring like behaviour. Hence, it should have a damping to eventually bring back the bodies to initial state and stop at some point.

box2d physics spring

Share Improve this question Follow

edited Apr 19, 2015 at 18:58

Anko - inactive in protest
13.4k 10 54 26

asked Apr 19, 2015 at 17:41

Narek
1,327 3 12 26

I saw a physics presentation by the author of box2d and his talk was specifically talking about how to make sure physics systems are convergent so they will slow down and come to a rest instead of gaining energy over time and things "exploding" mathematically. The talk showed that since physics simulations are just approximations, the methods either dampen or amplify. Because of this, I think it's possible there might not be an easy way to make this happen, and that its by design that things dampen over time, due to how box2d does integration. May be a way but I could see not being a way too – Alan Wolfe Apr 19, 2015 at 18:21

@AlanWolfe The integrator that uses Box2D (semi-implicit Euler) is Symplectic integrator. Semi-implicit Euler method almost conserves the energy. Hence, I think it is more about what is the task of the Distance Joint, rather than how the integration is being done, and does system loses energy because of the simulation.

– Narek Apr 19, 2015 at 18:47

Dieser Beitrag bestätigt einerseits, dass es kein Problem von Semi-Implizit-Euler ist und andererseits wird hier spekuliert, dass Erin selber gesagt hat es „müsste“ so sein.

Ich habe über eine Box2D-C#-Portierung getestet, wie sich dort die ungedämpfte Feder verhält:

<https://github.com/codingben/box2d-netstandard>

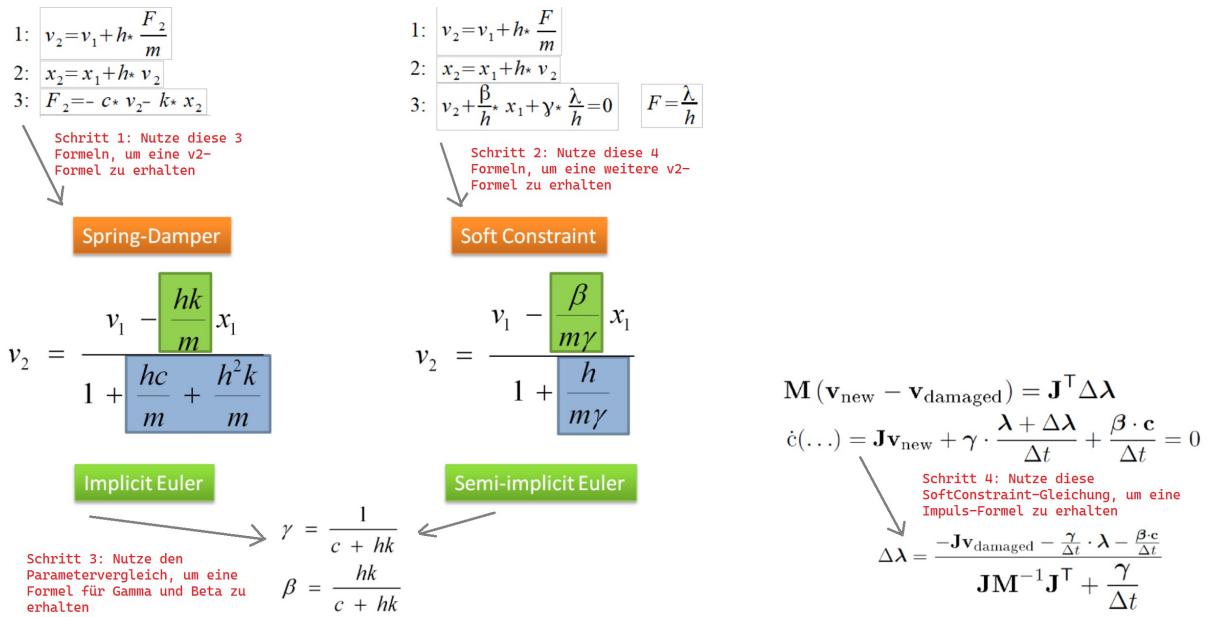
The screenshot shows the Visual Studio IDE with the following components:

- Code Editor:** Displays the `Box2D.WorldTests.cs` file. The code defines a `CreateWorld` method that creates a world, adds several bodies, and creates a `DistanceJointDef` to connect them. The damping ratio is set to 0.0f, which is highlighted in yellow.
- Project Explorer:** Shows the project structure for "box2d-examples-window". It includes subfolders like "examples" containing "Box2D.Window" and "Box2D.WindowTests", and other files like "Program.cs".
- Output Window:** Shows the output of the physics simulation, indicating FPS: 60.0 and Frames: 1225.
- Simulator Window:** A black window titled "Physics Simulation" showing the results of the simulation.

Auch dort wird die Feder gedämpft, obwohl die DampingRatio 0 ist. Ich gehe deswegen davon aus, dass es mit SoftConstraints und Semi-Implizit-Euler schwer umsetzbar ist eine unendliche Federschwingung zu simulieren da selbst Erin Catto an dieser Aufgabe gescheitert ist. Will man unbedingt eine unendliche Federschwingung simulieren, dann muss man die Feder über die externe Kraft so wie bei ResolverHelper1 umsetzen. Der Nachteil ist dann aber, dass diese Feder explodiert, wenn die Steifigkeit zu hoch ist und zu viele Kräfte auf sie wirken.

Zusammenfassung – Die gedämpfte Feder

Um die Bewegung von einer gedämpften Feder zu simulieren haben wir zuerst ein Massestück genommen und dort dann die Force-Property laut der Federkraft-Formel modifiziert. Bei dieser Feder haben wir dann aber festgestellt, dass sie explodiert, wenn die Steifigkeit zu hoch ist. Dann haben wir festgestellt, dass beim Impliciten Euler-Verfahren die Feder nicht explodiert. Da unsere Engine aber Semi Implicit Euler nutzt, haben wir aus der Federkraftformel die SoftConstraint-Formel hergeleitet, mit der es möglich ist, eine stabile Feder bei Semi Implicit Euler zu simulieren. Allerdings tauchen in der SoftConstraint die unbekannten Faktoren Beta und Gamma auf, welche wir dann über die Schritte 1 bis 3 (siehe Bild) hergeleitet haben. Für die Herleitung der Impulsformel haben wir Schritt 4 verwendet:



Die Formel für k und c stammt aus Wikipedia.

Somit hat man nun folgende Formeln:

$m \frac{d^2x}{dt^2} + c \frac{dx}{dt} + kx = 0$  acceleration velocity position	$\gamma = \frac{1}{c + hk}$ $\beta = \frac{hk}{c + hk}$ $f = \frac{\omega}{2\pi}$	$k = m_{eff}\omega^2$ $c = 2m_{eff}\zeta\omega$ $\omega = \text{Undamped angular frequency of the oscillator}$ $m_{eff} = \text{Effective Mass}$
m Mass c viscous damping coefficient (Damping) k Force Constant (Stiffness) x Auslenkung	h TimeStep $\gamma \beta$ SoftConstraint-Parameter	f Frequency ζ DampingRatio Vom Nutzer vorgegeben

Der Nutzer definiert nun zwei Objekte und zwei Ankerpunkte. Die Hebelarmvektoren r1/r2 gehen vom Zentrum zum Ankerpunkt.



Der Abstand der beiden Ankerpunkte zum Simulationsstart-Zeitpunkt bestimmt die Länge der Feder (joint.Length).

Außerdem legt der Nutzer über die Frequenz fest, wie oft in der Sekunde sie schwingen soll und über die DampingRatio (0..1) wie stark sie gedämpft ist.

$$f \quad \text{Frequency}$$

$$\zeta \quad \text{DampingRatio}$$

Nun gibt es zwei Wege eine Feder zu simulieren:

Weg 1: Die Force-Property wird entsprechend der Federkraftformel manipuliert:

```

47 // Relative velocity in Force direction
48 float velocityInForceDirection = relativeVelocity * ForceDirection;
49
50 float force = -joint.Damping * velocityInForceDirection - joint.Stiffness * (length - joint.Length);
51
52 joint.B1.Force -= ForceDirection * force;
53 joint.B2.Force += ForceDirection * force;
  
```

→ Hier kann die Feder bei zu hoher Steifigkeit explodieren aber dafür ist eine unendlich lange ungedämpfte Schwingung möglich.

Weg 2: Distance-Constraint per SoftConstraint weich machen:

Um die Distance-Constraint um eine SoftConstraint zu erweitern werden zusätzlich zur Effektivmasse noch die Parameter Gamma und Beta im Constraint-Konstruktur berechnet (Zeile 81, 82). Für die Impulsformel wird anstelle von Zeile 78 nun Zeile 87 verwendet:

```

62     float invMass = B1.InverseMass + B1.InverseInertia * r1crossN * r1crossN + B2.InverseMass + B2.InverseInertia * r2crossN * r2crossN;
63     this.ImpulseMass = if / invMass;                                     Bias legt fest:
64
65     float biasFactor = s.DoPositionalCorrection ? s.PositionalCorrectionRate : 0.0f; vRel soll 0 sein
66     this.Bias = biasFactor * data.InvDt * (joint.Length - length);           Abstand der Ankerpunkte soll joint.Length sein
67
68     if (joint.ParameterType != ExportData.Joints.DistanceJointExportData.SpringParameter.NoSoftness)
69     {
70         this.IsSoftConstraint = true;
71         float h = data.Dt;
72         //So könnte man laut der Formel (2.13) oder (2.17) aus "3D Constraint Derivations for Impulse Solvers - Marijn 2015" auch den Impu
73         float gamma = 1 / (joint.Damping + h * joint.Stiffness);
74         float beta = (h * joint.Stiffness) / (joint.Damping + h * joint.Stiffness);
75         this.K = invMass + gamma / h;
76         this.BetaCDeltaT = beta * (length - joint.Length) / h;
77         this.GammaDt = gamma / h;
78
79     }
80
81     float impulse = float.NaN;
82     if (c.IsSoftConstraint == false)
83     {
84         impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection); Distanzjoint ohne Dämpfung = Eisenstab
85     }
86     else
87     {
88         var d = (c as DistanceJointConstraint);                         Distanzjoint mit Dämpfung = Eisenfeder
89         ///(2.17)-Formel aus "3D Constraint Derivations for Impulse Solvers - Marijn 2015"
90         impulse = (-velocityInForceDirection - d.GammaDt * c.AccumulatedImpulse - d.BetaCDeltaT) / d.K;
91     }
92
93 }
```

Wenn bei einer Distance-Constraint die „normale“ Constraint-Formel verwendet wird, dann haben die Ankerpunkte immer einen fest definierten Abstand zueinander (obere Formeln im Bild). Will man eine Feder zwischen die Ankerpunkte spannen, nutze man die unteren Formeln:

$$\text{Constraint} \quad J_V + \frac{\beta}{h} C(x) = 0$$

Bias legt den Zielwert für v_{rel} fest und den Zielabstand der Ankerpunkte

$$m_{\text{eff}} = \frac{1}{JM^{-1}J^T}$$

$$m_{\text{eff}} v_{\text{rel}} = p$$

$$\text{SoftConstraint} \quad J_V + \frac{\beta}{h} C(x) + \gamma \frac{\lambda}{h} = 0 \quad \text{Gamma legt den Dämpfungsfaktor fest}$$

$$\mathbf{K} = \mathbf{JM}^{-1}\mathbf{J}^T + \frac{\gamma}{\Delta t}$$

$$\Delta \lambda = \frac{-\mathbf{Jv}_{\text{damaged}} - \frac{\gamma}{\Delta t} \cdot \lambda - \frac{\beta \cdot c}{\Delta t}}{\mathbf{K}}$$

Man kann mit SoftConstraints nicht nur eine DistanceConstraint „weich“ machen sondern auch noch andere Constraints, die noch kommen werden.

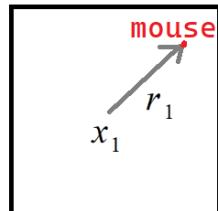
Mouse Joint (Objekt per Maus verschieben)

Maus-Constraint ohne Verwendung von SoftConstraint

Herleitung der Maus-Constraint-Formel

Es soll ein Objekt per Mauszeiger verschoben werden. Der Nutzer klickt auf das Objekt drauf, wodurch der Ankerpunkt festgelegt wird. Nun soll dieser Ankerpunkt immer den Mauszeiger folgen indem Kräfte ihn dahin ziehen.

Gegeben ist ein Objekt und ein roter Mausklickt-Punkt. Der Hebelarm r_1 zeigt vom Zentrum x_1 zum Mausklick-Punkt. Der Ankerpunkt ergibt sich aus $x_1 + r_1$



Schritt 1: Die Positionconstraint misst den Abstand vom Ankerpunkt zum Mauspunkt.

$$C: (x_1 + r_1) - \text{mouse} = 0$$

Schritt 2: Die Ableitung ergibt die Velocity-Constraint

$\dot{C}: v_1 + \omega_1 \times r_1 = 0$ Da der mouse-Punkt als konstant betrachtet wird, ist dessen Ableitung nach der Zeit 0.

Schritt 3: Den J- und V-Vektor extrahieren

Ich nutze die Cross Product Matrix von „3D Constraint Derivations for Impulse Solvers - Marijn 2015“ Seite 32

$$\begin{aligned} \mathbf{a} \times \mathbf{b} &= [\mathbf{a}]_{\times} \mathbf{b} \\ [\mathbf{a}]_{\times} &= \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} \\ [\mathbf{a}]^T_{\times} &= -[\mathbf{a}]_{\times} \\ -\mathbf{a} \times \mathbf{b} &= \mathbf{b} \times \mathbf{a} = -[\mathbf{a}]_{\times} \mathbf{b} = [\mathbf{a}]^T_{\times} \mathbf{b} \end{aligned}$$

Und davon diese Regel: $\mathbf{b} \times \mathbf{a} = [\mathbf{a}]^T_{\times} \mathbf{b}$ um Omega1 aus dem Kreuzprodukt zu lösen

$$\omega_1 \times r_1 = [r_1]_{\times}^T \omega_1$$

Alternative Quelle für die Cross-Produkt Matrix (Dort wird es Skew-Symmetric Matrix genannt)

$$\begin{aligned} \mathbf{a} \times \mathbf{b} &= [\mathbf{a}]_{\times} \mathbf{b} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \\ \mathbf{a} \times \mathbf{b} &= [\mathbf{b}]^T_{\times} \mathbf{a} = \begin{bmatrix} 0 & b_3 & -b_2 \\ -b_3 & 0 & b_1 \\ b_2 & -b_1 & 0 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \end{aligned}$$

https://en.wikipedia.org/wiki/Cross_product

Diese Cross-Produkt-Matrix-Regeln beziehen sich auf 3D-Vektoren. Bei unserer 2D-Physik-Engine

wollen wir aber 2D-Vektoren nutzen wo das Kreuzprodukt so definiert ist, dass für die Z-Komponente beim Hebelarm r Null eingesetzt wird und die Winkelgeschwindigkeit Omega ist eine skalare Zahl und kein 3D-Vektor.

Die Skew-Matrix definiere ich, indem ich für a1/a2/b3 Null einsetze (rote Nullen), so dass die Skew-Matrix wie auf der rechten Seite vereinfacht werden kann:

$$\omega_1 \times r_1 = \mathbf{a} \times \mathbf{b} \quad \omega_1 = \mathbf{a} \quad r_1 = \mathbf{b}$$

$$\mathbf{a} \times \mathbf{b} = [\mathbf{a}]_{\times} \mathbf{b} = \begin{bmatrix} 0 & -a_3 \\ a_3 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \quad [\mathbf{a}]_{\times} = \begin{bmatrix} 0 & -a_3 \\ a_3 & 0 \end{bmatrix}$$

$$\mathbf{a} \times \mathbf{b} = [\mathbf{b}]^T_{\times} \mathbf{a} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ b_2 & -b_1 \end{bmatrix} \begin{bmatrix} 0 \\ b_1 \\ 0 \end{bmatrix} \quad [\mathbf{b}]^T_{\times} = \begin{bmatrix} -b_2 & b_1 \end{bmatrix}$$

Wegen der orange markierten Null kann ich die Skew-Matrix auch wie auf der rechten Seite definieren. Es erzeugt das gleiche Ergebnis. Diese verkürzte Skew-Matrix-Definition gilt nur für unseren Fall, wo wir das Kreuzprodukt zwischen ein 2D-Hebelarm und einer skalaren Drehgeschwindigkeit bestimmen wollen.

Für unseren Fall brauchen wir die untere von den Regeln, weil r1 auf der linken Seite stehen soll und Omega auf der rechten von der Multiplikation, wenn wir nach J*V umstellen wollen. (J enthält r; V enthält Omega).

Ich nutze nun die Regel $\omega_1 \times r_1 = [r_1]_x^T \omega_1$ und stelle $\dot{C}: v_1 + \omega_1 \times r_1 = 0$ um zu:

$$\dot{C}: v_1 + [r_1]_x^T \omega_1 + v_2 * 0 + \omega_2 * 0 = 0$$

$$\dot{C}: J * V = \begin{bmatrix} 1 \\ [r_1]_x^T \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0$$

Der Linke Vektor ist der gesuchte J-Vektor. Allerdings ist $[r_1]_x^T$ entweder ein 2D-Vektor wenn ich die verkürzte Skew-Matrix-Definition nehme oder eine 3*3-Matrix. Wir wollen die verkürzte Skew-Matrix-Definition verwenden wo $[r_1]_x^T$ ein 2D-Vektor ist. Das führt dazu, dass J nun nicht mehr ein Zeilenvektor mit 6 Floats ist sondern J besteht nun aus zwei Zeilen, da $[r_1]_x^T$ 2 Floats enthält.

So sieht J und V ausgeschrieben aus:

$$J = \begin{bmatrix} 1 & 0 & -r_{1y} & 0 & 0 & 0 \\ 0 & 1 & r_{1x} & 0 & 0 & 0 \end{bmatrix} \quad V = \begin{bmatrix} v_{1x} \\ v_{1y} \\ \omega_1 \\ v_{2x} \\ v_{2y} \\ \omega_2 \end{bmatrix}$$

Wenn ich J*V rechne, dann ist die erste Zeile von J die X-Geschwindigkeit vom Mauszeiger zum Ankerpunkt und die zweite J-Zeile ist die Y-Geschwindigkeit.

Das ist nun die Velocity-Constraint-Gleichung unter Nutzung der 2*1-Skewmatrix:

$$\dot{C} : J_* V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -r_{1y} & r_{1x} \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}^T \begin{bmatrix} v_{1x} \\ v_{1y} \\ \omega_1 \\ v_{2x} \\ v_{2y} \\ \omega_2 \end{bmatrix} = 0$$

Hinweis: Für den 3D-Fall muss man die 3*3-Skewmatrix nehmen was dann so aussieht:

$$J = \begin{bmatrix} 1 & 0 & 0 & 0 & r_z & -r_y & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -r_z & 0 & r_x & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & r_y & -r_z & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Quelle: <http://allenchou.net/files/slides/2014/constraint-based-physics.pdf> Seite 25

Schritt 4: Den K-Faktor ausrechnen

Für die Berechnung des Impulses müssen wir den Faktor/die Matrix K ausrechnen, welche laut „3D Constraint Derivations for Impulse Solvers - Marijn 2015“ Seite 8 laut (2.12) / (2.4) definiert ist.

Hinweis: Die effektive Masse ist über $m_{\text{Effektive}}=1/K$ definiert. Wenn K eine skalare Zahl ist, kann ich das direkt ausrechnen. Wenn K eine Matrix ist, dann muss ich die inverse von K bestimmen. Wenn J nur aus einer Zeile besteht, dann ist K eine skalare Zahl. Enthält J mehrere Zeilen, dann handelt es sich um eine Multiconstraint und K ist eine Matrix.

$$K = JM^{-1}J^T + \frac{\gamma}{\Delta t} \quad (2.12)$$

$$K = JM^{-1}J^T$$

$$K\lambda = -J\bar{v} - \frac{\beta \cdot c}{\Delta t} \quad (2.13)$$

$$\lambda = \frac{-J\bar{v} - \zeta}{K} \quad (2.4)$$

$$\Delta\lambda = \frac{-Jv_{\text{damaged}} - \frac{\gamma}{\Delta t} \cdot \lambda - \frac{\beta \cdot c}{\Delta t}}{K} \quad (2.17)$$

$$\zeta = \text{Bias}$$

c = PositionConstraint-Error

Mit SoftConstraints

Ohne SoftConstraints

Quelle: 3D Constraint Derivations for Impulse Solvers - Marijn 2015 Seite 8

In unseren Beispiel hier ist K also eine Matrix, da J mehrere Zeilen enthält. So berechnet sich K bei der MouseConstraint, wenn ich ohne SoftConstraints arbeite:

$$J = \begin{bmatrix} 1 & 0 & -r_{1y} & 0 & 0 & 0 \\ 0 & 1 & r_{1x} & 0 & 0 & 0 \end{bmatrix} \quad V = \begin{bmatrix} v_{1x} \\ v_{1y} \\ \omega_1 \\ v_{2x} \\ v_{2y} \\ \omega_2 \end{bmatrix} \quad \dot{C} : J_* V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -r_{1y} & r_{1x} \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}^T \begin{bmatrix} v_{1x} \\ v_{1y} \\ \omega_1 \\ v_{2x} \\ v_{2y} \\ \omega_2 \end{bmatrix} = 0 \quad M = \begin{bmatrix} m_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & m_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & I_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & m_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & I_2 \end{bmatrix} \quad M^{-1} = \begin{bmatrix} m_1^{-1} & 0 & 0 & 0 & 0 & 0 \\ 0 & m_1^{-1} & 0 & 0 & 0 & 0 \\ 0 & 0 & I_1^{-1} & 0 & 0 & 0 \\ 0 & 0 & 0 & m_2^{-1} & 0 & 0 \\ 0 & 0 & 0 & 0 & m_2^{-1} & 0 \\ 0 & 0 & 0 & 0 & 0 & I_2^{-1} \end{bmatrix}$$

$m_2 = I_2 = \infty$
Masse des Mauszeigers ist unendlich

$$J^T M^{-1} = \begin{bmatrix} m_1^{-1} & 0 & 0 & 0 & 0 & 0 \\ 0 & m_1^{-1} & 0 & 0 & 0 & 0 \\ 0 & 0 & I_1^{-1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$K = J^T M^{-1} J^T =$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -r_{1y} & r_{1x} \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & -r_{1y} & 0 & 0 & 0 \\ 0 & 1 & r_{1x} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} m_1^{-1} & 0 & -r_{1y} * I_1^{-1} & 0 & 0 & 0 \\ 0 & m_1^{-1} & r_{1x} * I_1^{-1} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} m_1^{-1} & 0 & -r_{1y} * I_1^{-1} & 0 & 0 & 0 \\ 0 & m_1^{-1} & r_{1x} * I_1^{-1} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} m_1^{-1} + r_{1y}^2 * I_1^{-1} & -r_{1x} * r_{1y} * I_1^{-1} \\ -r_{1x} * r_{1y} * I_1^{-1} & m_1^{-1} + r_{1x}^2 * I_1^{-1} \end{bmatrix}$$

Ich kann K auch über die Formel (2.6) von Seite 7 „3D Constraint Derivations for Impulse Solvers - Marijn 2015“ berechnen:

$$\mathbf{J} = [-\mathbf{L}_a, -\mathbf{A}_a, \mathbf{L}_b, \mathbf{A}_b]$$

$$\mathbf{JM}^{-1}\mathbf{J}^T = \mathbf{L}_a\mathbf{L}_a^T m_a^{-1} + \mathbf{A}_a\mathbf{I}_a^{-1}\mathbf{A}_a^T + \mathbf{L}_b\mathbf{L}_b^T m_b^{-1} + \mathbf{A}_b\mathbf{I}_b^{-1}\mathbf{A}_b^T \quad (2.6)$$

$$-\mathbf{L}_a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad m_a^{-1} \text{ Inverse Masse des angeklickten Objektes (Skalare Zahl)}$$

$$-\mathbf{A}_a = \begin{bmatrix} -r_{ly} \\ r_{lx} \end{bmatrix} \quad \mathbf{I}_a^{-1} \text{ Inverse Inertia des angeklickten Objektes (Skalare Zahl)}$$

$$\mathbf{L}_b = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad m_b^{-1} \text{ Inverse Masse/Inertia des Mauszeigers. Hat unendliche Masse und ist somit 0.}$$

$$\mathbf{A}_b = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \mathbf{I}_b^{-1}$$

$$\dot{\mathbf{C}} : J_* V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -r_{ly} & r_{lx} \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v_{1x} \\ v_{1y} \\ \omega_1 \\ v_{2x} \\ v_{2y} \\ \omega_2 \end{bmatrix} = 0$$

$$\mathbf{L}_a\mathbf{L}_a^T m_a^{-1} = \begin{bmatrix} m_a^{-1} & 0 \\ 0 & m_a^{-1} \end{bmatrix}$$

$$\mathbf{L}_b\mathbf{L}_b^T m_b^{-1} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$\mathbf{A}_a \mathbf{A}_a^T = \begin{bmatrix} -r_{ly} & r_{lx} \\ r_{ly}^2 & -r_{lx}^2 \\ r_{lx} & -r_{lx}^2 r_{ly} \end{bmatrix}$$

$$\mathbf{A}_b \mathbf{I}_b^{-1} \mathbf{A}_b^T = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$\mathbf{JM}^{-1}\mathbf{J}^T = \begin{bmatrix} m_a^{-1} & 0 \\ 0 & m_a^{-1} \end{bmatrix} + \begin{bmatrix} r_{ly}^2 & -r_{lx}^2 r_{ly} \\ -r_{lx}^2 r_{ly} & r_{lx}^2 \end{bmatrix} \mathbf{I}_a^{-1} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$\boxed{\mathbf{K} = \mathbf{JM}^{-1}\mathbf{J}^T = \begin{bmatrix} m_a^{-1} + r_{ly}^2 \mathbf{I}_a^{-1} & -r_{lx}^2 r_{ly} \mathbf{I}_a^{-1} \\ -r_{lx}^2 r_{ly} \mathbf{I}_a^{-1} & m_a^{-1} + r_{lx}^2 \mathbf{I}_a^{-1} \end{bmatrix}}$$

Erst muss dazu J bestimmt werden. Dann unterteile ich J in die 4 farblich markierten Bereiche und ermittle so -La, -Aa, Lb, Ab (rechts oben im Bild). Dann berechne ich die 4 Summenglieder aus der K-Formel (rechts unten im Bild). Jedes Summenglied muss entweder ein skalarer Wert oder eine Matrix mit gleicher Dimension sein. Die Summe dieser 4 Elemente ergibt K (links unten im Bild).

Umsetzung der MouseConstraint ohne Softconstraints

Schritt 5: Die MouseConstraint-Klasse erstellen

```

53  public MouseConstraint(ConstraintConstructorData data, MouseConstraintData mouseData)
54  {
55      this.mouseData = mouseData;
56
56      this.B2 = mouseData.RigidBody;
57      this.B1 = null;
58      this.R2 = MathHelp.GetWorldDirectionFromLocalDirection(mouseData.RigidBody, mouseData.LocalAnchorDirection);
59      this.R1 = null;
60
61      this.AccumulatedMultiConstraintImpulse = data.Settings.DoWarmStart ? mouseData.AccumulatedImpulse : new Vector2D(0, 0);
62
63      var b = mouseData.RigidBody;
64      var r = this.R2;
65      this.InverseK = Matrix2x2.FromScalars(
66          b.InverseMass + r.Y * r.Y * b.InverseInertia,
67          -r.X * r.Y * b.InverseInertia,
68          -r.X * r.Y * b.InverseInertia,
69          b.InverseMass + r.X * r.X * b.InverseInertia
70      )
71      .Invert();
72
73      this.MaxImpulse = data.Dt * mouseData.MaxForce;
74
75  }
76
77  public Vector2D GetCDot() //Gibt CDot=J*V zurück (Geschwindigkeit in Richtung jeder J-Zeile)
78  {
79      var b = this.mouseData.RigidBody;
80      Vector2D v = b.Velocity + new Vector2D(-b.AngularVelocity * this.R2.Y, b.AngularVelocity * this.R2.X);
81      return v;
82  }
83
84  2 Verweise
85  public Vector2D GetC()
86  {
87      var b = this.mouseData.RigidBody;
88      Vector2D positionError = b.Center + this.R2 - this.mouseData.MousePosition;
89
90      return positionError;
91  }

```

$$\boxed{\mathbf{K} = \mathbf{JM}^{-1}\mathbf{J}^T = \begin{bmatrix} m_a^{-1} + r_{ly}^2 \mathbf{I}_a^{-1} & -r_{lx}^2 r_{ly} \mathbf{I}_a^{-1} \\ -r_{lx}^2 r_{ly} \mathbf{I}_a^{-1} & m_a^{-1} + r_{lx}^2 \mathbf{I}_a^{-1} \end{bmatrix}}$$

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Constraints/MouseConstraint.cs

Und so die Impuls-Berechnung und Anwendung:

```

105    private static void SolveMultiConstraint(IConstraint c, float invDt)
106    {
107        float biasFactor = 1.0f; // 0..1 -> Wie schnell folgt das Objekt dem Mauszeiger
108        Vector2D bias = biasFactor * invDt * c.GetC();
109        Vector2D impulse = c.InverseK * (-bias - c.GetCDot());
110
111        Vector2D oldSum = c.AccumulatedMultiConstraintImpulse;
112        c.AccumulatedMultiConstraintImpulse += impulse;
113
114        //Max-Clamping
115        if (c.MaxImpulse != float.MaxValue && c.AccumulatedMultiConstraintImpulse.SquareLength() > c.MaxImpulse * c.MaxImpulse)
116        {
117            c.AccumulatedMultiConstraintImpulse *= c.MaxImpulse / c.AccumulatedMultiConstraintImpulse.Length();
118        }
119
120        impulse = c.AccumulatedMultiConstraintImpulse - oldSum;
121
122        ApplyImpulse(c, impulse);
123
124        c.SaveImpulse();
125    }
126
127    private static void ApplyImpulse(IConstraint c, Vector2D impulse)
128    {
129        if (c.B1 != null)
130        {
131            c.B1.Velocity -= impulse * c.B1.InverseMass;
132            c.B1.AngularVelocity -= Vector2D.ZValueFromCross(c.R1, impulse) * c.B1.InverseInertia;
133        }
134
135        if (c.B2 != null)
136        {
137            c.B2.Velocity += impulse * c.B2.InverseMass;
138            c.B2.AngularVelocity += Vector2D.ZValueFromCross(c.R2, impulse) * c.B2.InverseInertia;
139        }
140    }
141
142    private static void SaveImpulse(IConstraint c)
143    {
144        if (c.B1 != null)
145        {
146            c.B1.Velocity = Vector2D.Zero;
147            c.B1.AngularVelocity = 0.0f;
148        }
149
150    }

```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/ResolverHelper.cs

Test der Mausconstraint



Ich teste die Maus-Constraint indem ich Würfel an verschiedenen Punkten anklicke und sie dann um 10 Pixel nach oben ziehe. Erwartung ist, dass der Ankerpunkt des Würfels an der Maus kleben bleibt. Die grünen Würfel haben unterschiedliches Gewicht. Der rote Boden ist nicht verschiebbar.

Maus-Constraint mit Verwendung von SoftConstraint

Wenn ich ein Objekt mit der Maus anklicken will, was per Gummiband festgehalten wird, so muss die Maus-Constraint-Formel um die SoftConstraint-Parameter erweitert werden.

Bei SoftConstraints muss Formel (2.12) und (2.17) verwendet werden. Wenn ich Multiconstraints habe, dann steht im Zähler von (2.17) ein 2D-Vektor und aus der K-Division wird eine K-Inverse-2*2-Matrix-Multiplikation. Die SoftConstraint-Formel (2.12) zur K-Berechnung ändert sich wegen des Multiconstraint-Einsatzes dahingehend, dass die Einheitsmatrix noch vor den skalaren (roten) Faktor geschrieben werden muss, da nur so eine 2*2-Matrix-Addition erfolgen kann.

$$\gamma = \frac{1}{c + hk}$$

$$\beta = \frac{hk}{c + hk}$$

$$h \text{ TimeStep}$$

$$\gamma, \beta \text{ SoftConstraint-Parameter}$$

$$k = m_{\text{eff}} \omega^2$$

$$c = 2m_{\text{eff}} \zeta \omega$$

$$f = \frac{\omega}{2\pi}$$

$$\zeta \text{ DampingRatio}$$

$$\omega \text{ Undamped angular frequency of the oscillator}$$

$$m_{\text{eff}} \text{ Effective Mass}$$

$$\gamma, \beta \text{ SoftConstraint-Parameter}$$

Da K eine Matrix ist bedeutet durch K dividiert das man die Inverse von K bildet und dann mit dieser multipliziert.

Skalare Zahlen

2*2-Matrix

2D-Vektoren

Formeln:

$$\mathbf{K} = \mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \frac{\gamma}{\Delta t} \quad (2.12)$$

$$\Delta \lambda = \frac{-\mathbf{J}\mathbf{v}_{\text{damaged}} - \frac{\gamma}{\Delta t}(\lambda - \frac{\beta c}{\Delta t})}{\mathbf{K}} \quad (2.17)$$

$$\mathbf{c} = \text{PositionConstraint-Error}$$

$$\mathbf{J} = \begin{bmatrix} 1 & 0 & -r_{ly} & 0 & 0 & 0 \\ 0 & 1 & r_{lx} & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{J}^{-1} = \begin{bmatrix} m_a^{-1} + r_{ly}^2 \cdot I_a^{-1} & -r_{ly} \cdot r_{ly} \cdot I_a^{-1} \\ -r_{lx} \cdot r_{ly} \cdot I_a^{-1} & m_a^{-1} + r_{lx}^2 \cdot I_a^{-1} \end{bmatrix}$$

$$\mathbf{v}_{\text{RelX}} = \begin{bmatrix} v_{\text{RelX}} \\ v_{\text{RelY}} \end{bmatrix}$$

$$\lambda = \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix}$$

$$\mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

So sieht die Maus-Constraint-Klasse und der Impulse-Solver bei Multi-Soft-Constraints aus:

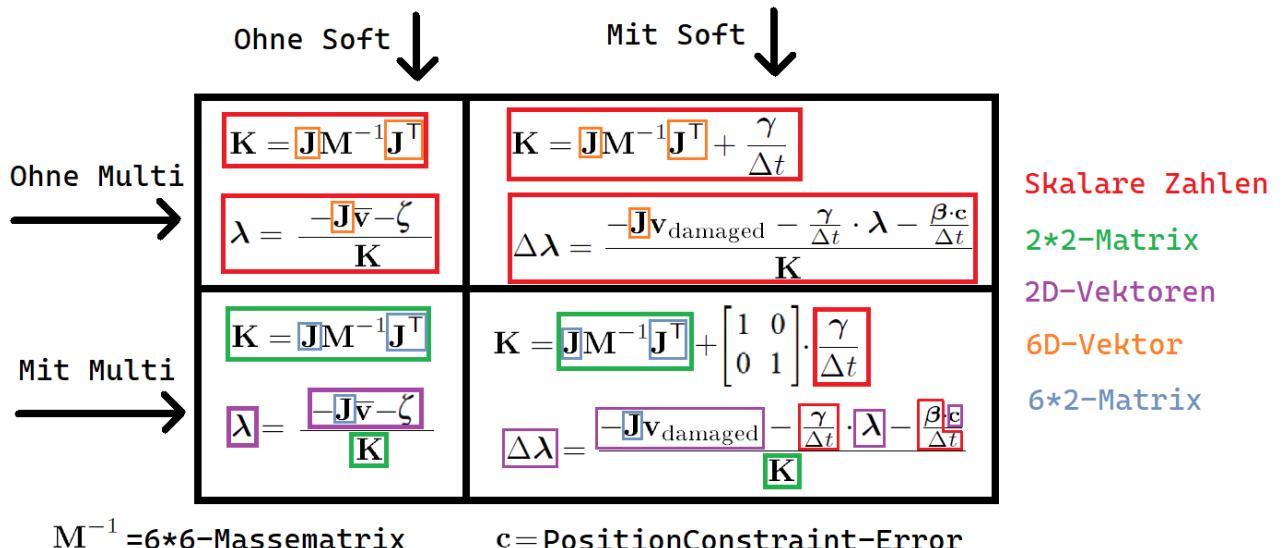
```

80   {
81     this.IsSoftConstraint = true;
82
83     float h = data.Dt;
84     float d = mouseData.Damping;
85     float k = mouseData.Stiffness;
86     this.Gamma = h * (d + h * k);
87     if (this.Gamma != 0)
88       this.Gamma = 1.0f / this.Gamma; //Gamma / h
89     this.Beta = h * k * this.Gamma; //Beta / h
90
91     this.InverseK = Matrix2x2.FromScalars(
92       b.InverseMass + r.Y * r.Y * b.InverseInertia + this.Gamma,
93       -r.X * r.Y * b.InverseInertia,
94       -r.X * r.Y * b.InverseInertia,
95       b.InverseMass + r.X * r.X * b.InverseInertia + this.Gamma
96     );
97     .Invert();
98   }
99
100  private static void SolveMultiConstraint(IConstraint c, float invDt)
101  {
102    Vector2D impulse = null;
103
104    if (c.IsSoftConstraint == false)
105    {
106      float biasFactor = 1.0f; //0..1 -> Wie schnell folgt das Objekt dem Mauszeiger
107      Vector2D bias = biasFactor * invDt * c.GetC();
108      impulse = c.InverseK * (-bias - c.GetCDot());
109    }
110    else
111    {
112      impulse = c.InverseK * (-c.GetCDot() - c.Gamma * c.AccumulatedMultiConstraintImpulse - c.Beta * c.GetC());
113    }
114
115    Vector2D oldSum = c.AccumulatedMultiConstraintImpulse;    $\Delta\lambda = \frac{-\mathbf{J}\mathbf{v}_{\text{damaged}} - \frac{\gamma}{\Delta t} \cdot \boldsymbol{\lambda} - \frac{\beta \cdot c}{\Delta t}}{\mathbf{K}}$  (2.17)
116    c.AccumulatedMultiConstraintImpulse += impulse;
117
118    //Max-Clamping
119    if (c.MaxImpulse != float.MaxValue && c.AccumulatedMultiConstraintImpulse.SquareLength() > c.MaxImpulse * c.MaxImpulse)
120    {
121      c.AccumulatedMultiConstraintImpulse *= c.MaxImpulse / c.AccumulatedMultiConstraintImpulse.Length();
122    }
123
124    impulse = c.AccumulatedMultiConstraintImpulse - oldSum;
125
126    ApplyImpulse(c, impulse);
127
128    c.SaveImpulse();
129
130  }
131
132  }
133

```

Zusammenfassung von Teil 4

Eine Constraint hat Soft- und Multi-Schalter. Dadurch ergeben sich 4 unterschiedliche Constraint-Arten beim Sequentiell-Impulse-Solver. Gemeinsamkeit von allen 4 Constraintarten ist, dass sie feste Variablen wie K, Gamma, Lambda und PositionError haben, welche vorher berechnet werden können und in der SI-Schleife wird dann der neue Impuls mit diesen festen Variablen, den akkumulierten Impuls und der aktuellen Kontaktgeschwindigkeit berechnet.



$$\bar{\mathbf{v}} = \mathbf{v}_{\text{damaged}} = 6\text{D-Vektor} \quad \gamma = \frac{1}{c + hk} \quad \beta = \frac{hk}{c + hk}$$

Eine Constraint wird dann zur Multi-Constraint, wenn die Position-Error-Funktion ein Vektor anstelle einer skalaren Zahl zurück gibt. Bei der Mouse-Constraint haben wir den Abstand zwischen den Mausklickpunkt und dem Objekt-Ankerpunkt dadurch gemessen, indem wir den X- und Y-Abstand genommen haben und ihn mit 0 gleichgesetzt haben. Hätte man stattdessen den

euklidischen Abstand als Maß genommen, dann wäre die MouseConstraint ohne Multi gewesen und es wäre dann so etwas wie eine DistanceConstraint mit ein Soll-Abstandswert von Null geworden.

Die Entscheidung, ob eine Constraint Soft ist oder nicht hängt davon ab, ob ich eine gedämpfte Feder simulieren will oder ob ich eine steife Eisenstange/Drehgelenk simulieren will.

Quellen für diesen Abschnitt

Soft Constraints - Erin Catto 2011

3D Constraint Derivations for Impulse Solvers - Marijn 2015

https://github.com/erincatto/box2d/blob/main/src/dynamics/b2_distance_joint.cpp

https://github.com/erincatto/box2d/blob/main/src/dynamics/b2_joint.cpp

https://github.com/erincatto/box2d/blob/main/src/dynamics/b2_mouse_joint.cpp

<http://allenhou.net/files/slides/2014/constraint-based-physics.pdf>

Teil 5: Joints

Die Physikengine soll neben dem Distanzjoint um weitere Joints erweitert werden. Jedes Joint-Objekt ist dadurch definiert, dass zwei Körper mit jeweils ein Ankerpunkt gegeben sind. Das Joint-Objekt lässt dann auf die Ankerpunkte eine Kraft wirken. Bevor aber über den internen Aufbau der Joints gesprochen werden kann soll zuerst hier erklärt werden, welche Joints es gibt.

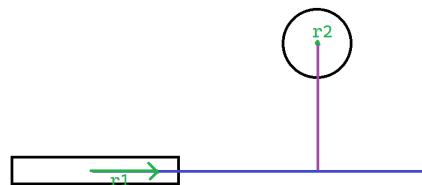
5.1 Nutzung der Joints im Editor

Wheel Joint (Gardinenstange) (Slider bei Unity3D)

Beim Wheel-Joint soll eine Rolle simuliert werden, welche in einer Schiene läuft. Die Rolle darf sich drehen aber sie muss immer auf der Schiene bleiben.

Die Schiene wird dadurch definiert, indem ich bei Körper 1 ein Ankerpunkt definiere. Die blaue Achse, welche durch das Zentrum von Körper 1 und den Ankerpunkt definiert ist, gibt nun die Schiene an. Dreht sich Körper 1, dann dreht sich auch die Schiene mit. Bei Körper 2 wird auch ein Ankerpunkt definiert. Dieser Punkt muss dann immer auf der Schiene bleiben.

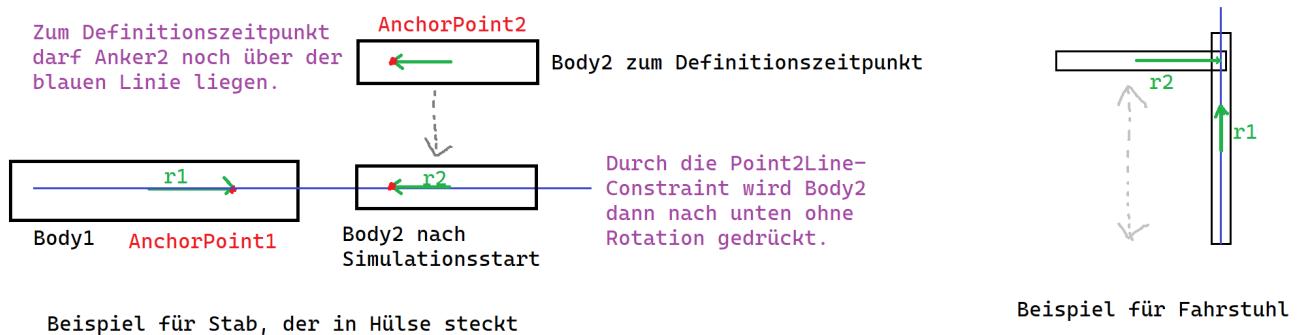
Die Aufgabe vom WheelJoint ist es den Ankerpunkt von Körper 2 auf die blaue Schiene von Körper 1 zu bringen. In diesem Bild hier hat Körper 2 ein senkrechten Abstand zur blauen Schiene (Lila-Linie). Das Wheeljoint wird nun den r2-Ankerpunkt zurück auf die blaue Linie drücken so dass die Lila-Linie dann eine Länge von Null hat.



Grün=Richtungsvektor vom Körperzentrum zum Ankerpunkt.

Prismatic Joint (Stab in Hülse)

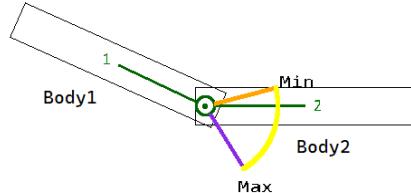
Beim Prismatic-Joint wird ein Stab simuliert, der in einer Hülse steckt. Da wir hier in 2D sind, heißt dass, der Stab kann sich nicht um seine Achse rotieren. Er hat nur ein Freiheitsgrad. Er kann sich in entlang der Hülse nur vor oder zurück bewegen. Beispiele dafür wären eine Luftpumpe, eine Teleskopstange oder ein Fahrstuhl (siehe Bild).



Das PrismaticJoint macht hier zwei Dinge. Es drückt den Ankerpunkt von Body2 auf die blaue Linie und es sorgt dafür, dass die relative Ausrichtung beider Körper sich nicht ändert. D.h. Beim Fahrstuhlbeispiel sorgt es dafür, dass der Winkel zwischen der Body1-Stange und Body2-Stange immer 90 Grad bleibt, weil die Körper zum Definitionszeitpunkt einen Winkelabstand von 90 Grad hatten.

Revolute Joint (Drehgelenk) (Hinge Constraint bei Unity)

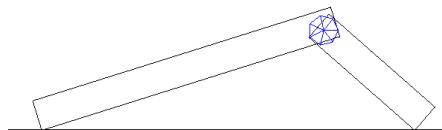
Bei diesen Gelenk sind zwei Körper über ein gemeinsamen Ankerpunkt verbunden. An den Hebelarm vom Zentrum von Körper 1 zum Ankerpunkt (hier grün 1) hängen die Min-Max-Schrankenarme. Dreht sich Körper 1, dann drehen sich auch die Schrankenarme mit. Hebelarm 2 darf nur innerhalb der Min-Max-Schranken sich aufhalten. Beispiel für so ein Gelenk wäre bei einer 2D-Figur der Ober- und Unterarm, die über ein RevoluteJoint verbunden sind.



Das Gelenk macht hier zwei Dinge. Es drückt die beiden Ankerpunkte zusammen und es dreht die Körper so, dass der grüne Hebelarm von Körper 2 innerhalb der Min-Max-Schranke liegt.

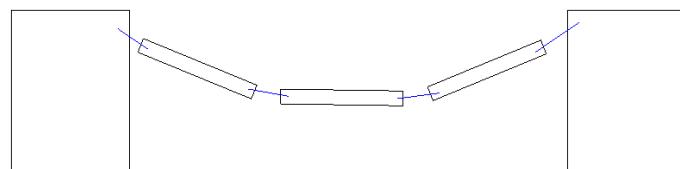
Weld Joint (Full Position Constraint)

Zwei Körper werden bei diesen Gelenk am Kontaktspunkt miteinander verschweißt so dass sie sich dann wie ein einzelner Körper verhalten. Jeder Körper hat hier einen Kontaktspunkt und diese Punkte werden durch das Weld-Joint zusammengedrückt und das WeldJoint verhindert hat, dass sich der Relativwinkel zwischen den Körpern ändert.



Distance Joint

Der Abstand von zwei Ankerpunkten bleibt immer gleich. Man kann somit entweder Hängebrücken oder Pendel bauen.



Oder man nutzt die Soft-Option und baut sich eine Feder/Gummiband.

5.2 Herleitung der Joint-Constraints

Ziel dieses Abschnitts

Da der Quellcode mittlerweile viel Code enthält, der nicht mehr gebraucht wird, soll es zuerst darum gehen denn Quellcode aufzuräumen, die IConstraint-Schnittstelle an die 4 Ausprägungen anzupassen und die externe Abhängigkeit zur Grafik-Bibliothek wegen der Vector-Klasse zu beseitigen. Nach dem Refactoring sollen dann weitere Constraints (Gelenke) implementiert werden.

Cleanup & Refactoring

Schritt 1: Ich entferne den Code, wo eine Feder als externe Kraft simuliert wird, da ich nur über SoftConstraints das machen will.

```

Constraintfactory.cs
166 public void TimeStep(float dt)
167 {
168     //1. Weise den externen Kraft einen Wert zu (Der Nutzer darf vor jedem TimeStep-Aufruf auch selber
169     //if (this.HasGravity)
170     //    AddGravityForceForAllBodies();
171 
172     //2. Emittiere alle Kollisionen
173     var collisionsFromThisTimeStep = CollisionHelper.GetAllCollisions(this.bodies);
174     if (collisionsFromThisTimeStep.Any())
175         this._collisionOccurred?.Invoke(this, collisionsFromThisTimeStep);
176 
177     //So kann man eine unendliche Federbeschleunigung simulieren
178     //var oldV = this.bodies.Select(x => x.Velocity).ToArray();
179     //var oldA = this.bodies.Select(x => x.AngularVelocity).ToArray();
180 
181     //3. Constraint-Kraft aufgrund der aktuellen Geschwindigkeit berechnen und zusammen mit der externen
182     //Kraft addieren
183     if (collisionsFromThisTimeStep.Any() || this.joints.Any() || this.mouseData != null) //Abfrage: Gib
184         this._impulseResolver.Resolve(new SolverInputData(this.bodies, this.joints, collisionsFromThisT
185     else
186         ApplyExternalForces(dt); //Wörde ohne Beschränkung bewegen (Wende nur die externe Kraft an)
187 
188     //4. Geschwindigkeit verändert die Position
189     MoveBodiesAndSetForceToZero(dt);
190 
191     /*
192     //So kann man eine unendliche Federbeschleunigung simulieren
193     for (int i = 0; i < bodies.Count; i++)
194     {
195         var body = bodies[i];
196         body.MoveCenter(dt * oldV[i]);
197         body.Rotate(dt * oldA[i]);
198 
199         //Resette die externe Kraft
200         body.Force = new Vector2D(0, 0);
201         body.Torque = 0;
202     }

```

```

PhysicsEngine.cs
49 private static void GetJointDistance(Distance joint)
50 {
51     Vector2D a1To2 = joint.Anchor2 - joint.Anchor1;
52     float length = a1To2.Length();
53 
54     Vector2D R1 = joint.Anchor1 - joint.B1.Center;
55     Vector2D R2 = joint.Anchor2 - joint.B2.Center;
56 
57     //VelocityAtContactPoint = V + mangularVelocity cross R
58     Vector2D v1 = joint.B1.Velocity + new Vector2D(-joint.B1.AngularVelocity * R1.Y, joint.B1.AngularVelocity * R1
59     Vector2D v2 = joint.B2.Velocity + new Vector2D(-joint.B2.AngularVelocity * R2.Y, joint.B2.AngularVelocity * R2
60     Vector2D relativeVelocity = v2 - v1;
61 
62     //Relative velocity in Force direction
63     float velocityInForceDirection = relativeVelocity * ForceDirection;
64 
65     float force = -joint.Damping * velocityInForceDirection - joint.Stiffness * (length - joint.Length);
66 
67     joint.B1.Force += ForceDirection * force;
68     joint.B2.Force += ForceDirection * force;
69 }

```

Schritt 2: Das IConstraint-Interface wird in ILinear1DConstraint, ILinear2DConstraint, IAngularConstraint, ISoftConstraint, ISoftConstraint1D und ISoftAngular aufgeteilt:

```

IConstraint.cs
6 internal interface IConstraint
7 {
8     //1 Verweise
9     Rigidbody B1 { get; }
10    //2 Verweise
11    Rigidbody B2 { get; }
12    //3 Verweise
13    Vec2D R1 { get; } //Mittelwert vom B1.Center zum Kontaktpunkt
14    //4 Verweise
15    Vec2D R2 { get; } //Mittelwert vom B2.Center zum Kontaktpunkt
16    //5 Verweise
17    float MinImpulse { get; }
18    //6 Verweise
19    float MaxImpulse { get; }
20    //7 Verweise
21    void SaveImpulse(); //Speichert den Impuls im CollisionPointWithImpulse oder IJoint
22    //8 Verweise
23    void ApplyWarmStartImpulse();
24    //9 Verweise
25    void DoSingleSISetup();
26 
27    //Es wird eine Kraft zwischen zwei Ankerpunkten / Kontaktpunkten entlang der Richtung ForceDirection (linearer 1D-Impuls)
28    //10 Verweise
29    internal interface ILinear1DConstraint : IConstraint
30    {
31        //11 Verweise
32        Vec2D ForceDirection { get; } //In diese Richtung wird B2 gedrückt (B1 wird entgegengesetzt gedrückt)
33        //12 Verweise
34        float Bias { get; } //Voreingabe für die Relativgeschwindigkeit der Kontaktpunkte
35        //13 Verweise
36        float ImpulseMass { get; } //Umrechnungsvektor von der Relativ-Kontaktpunktgeschwindigkeitswert in ein Impuls (Entspricht Inver
37        //14 Verweise
38        float AccumulatedImpulse { get; set; }
39    }
40 
41    //Lineares 2D-Impuls
42    //15 Verweise
43    internal interface ILinear2DConstraint : IConstraint
44    {
45        //16 Verweise
46        Matrix2x2 InverseK { get; } //=>J*M^-1*I*T
47        //17 Verweise
48        void GetDot(); //Gibt CDot=J*V zurück (Geschwindigkeit in Richtung jeder J-Zeile)
49        //18 Verweise
50        Vec2D Bias { get; }
51        //19 Verweise
52        Vec2D AccumulatedImpulse { get; set; }
53    }
54 }

```

```

InternalInterfaces.cs
38 internal interface IAngularConstraint : IConstraint
39 {
40     //10 Verweise
41     float Bias { get; } //Voreingabe für die relative Winkelgeschwindigkeit zwischen Body1 und Body2
42     //11 Verweise
43     float ImpulseMass { get; } //Umrechnungsvektor vom Relative-Winkelgeschwindigkeit in ein Drehimpuls
44     //12 Verweise
45     float AccumulatedImpulse { get; set; }
46 
47     //Wenn eine Constraint das ISoftConstraint-Interface implementiert, dann heißt das, sie kann Soft sein. Muss es aber nicht.
48     internal interface ISoftConstraint
49     {
50         //13 Verweise
51         bool IsSoftConstraint { get; } //Nur wenn hier true steht, ist die Constraint wirklich soft. Ansonsten gilt sie als Stiff
52         //14 Verweise
53         float Gamma { get; }
54         //15 Verweise
55         float Beta { get; }
56     }
57 
58     //16 Verweise
59     internal interface ISoftConstraint1D : ISoftConstraint, ILinear1DConstraint
60     {
61         //7 Verweise
62         float PositionError { get; }
63     }
64 
65     //17 Verweise
66     internal interface ISoftConstraint2D : ISoftConstraint, ILinear2DConstraint
67     {
68         //4 Verweise
69         Vec2D PositionError { get; } //Entspricht dem PositionConstraint-Wert
70     }
71 
72     //18 Verweise
73     internal interface ISoftConstraint3D : ISoftConstraint, IAngularConstraint
74     {
75         //4 Verweise
76         Vec3D PositionError { get; } //Entspricht dem PositionConstraint-Wert
77     }
78 
79     //19 Verweise
80     internal interface ISoftAngular : ISoftConstraint, IAngularConstraint
81     {
82         //7 Verweise
83         float PositionError { get; }
84     }
85 }

```

Schritt 3: PhysicSceneConstructorData wird internal. Dafür bekommt PhysicScene.Reload nun PhysicSceneExportData als Input. Damit die UnitTests noch laufen muss ich das InternalsVisibleToAttribute setzen

er (C:) > Desktop > Data > C# > 58 PhysikEngine Tutorial > Part5 - Joints > Source > PhysicEngine

Name	Änderungsdatum	Typ	Größe
bin			
CollisionDetection			
CollisionResolution			
ExportData			
Joints			
MathHelper			
MouseBodyClick			
obj			
RigidBody			
PhysicEngine.csproj			
PhysicScene.cs			

```

<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <TargetFramework>net7.0</TargetFramework>
        <ImplicitUsings>enable</ImplicitUsings>
        <Nullable>enable</Nullable>
    </PropertyGroup>

    <ItemGroup>
        <Reference Include="GraphicMinimal">
            <HintPath>..\..\51 GraphicEngine8\GraphicEngine8\Source\Tools\bin\Debug\GraphicMinimal.dll</HintPath>
        </Reference>
        <AssemblyAttribute Include="System.Runtime.CompilerServices.InternalsVisibleToAttribute">
            <_Parameter1>PhysicEngine.UnitTests</_Parameter1>
        </AssemblyAttribute>
    </ItemGroup>

</Project>

```

Zeile 1, Spalte 1 | 100% | Windows (CRLF) | UTF-8

Schritt 4: Alle IRigidBody/IJoint-Interface werden internal. Es gibt jetzt nur noch eine eingeschränkte public-Sicht:

The screenshot shows three open code files in Visual Studio:

- IPublicRigidBody.cs** (Left):


```

5  public interface IPublicRigidBody
6  {
7  }
8
9  1 Verweis
10 public interface IPublicRigidBody : IPublicRigidBody
11 {
12     14 Verweise
13     Vector2D[] Vertex { get; }
14
15     11 Verweis
16     public interface IPublicRigidRectangle : IPublicRigidBody
17     {
18         8 Verweise
19         Vector2D Center { get; }
      
```
- IPublicJoint.cs** (Middle):


```

5  public interface IPublicJoint
6  {
7  }
8
9  5 Verweise
10 public interface IPublicDistanceJoint : IPublicJoint
11 {
12     8 Verweise
13     Vector2D Anchor1 { get; }
14
15     8 Verweise
16     Vector2D Anchor2 { get; }
      
```
- PhysicScene.cs** (Right):


```

58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
      
```

Schritt 5: Ich habe mit FindAll nach „public“ gesucht und überall dort, wo es möglich war es auf internal umgestellt

Schritt 6: Ich nutze nicht mehr die Vector2D-Klasse aus der Grafik-Bibliothek sondern die eigene Vec2D-Klasse um somit die Physik-Engine unabhängig von der Grafik-Bibliothek zu bekommen. Nur beim Zeichnen im Testbed-Projekt nutze ich noch die Vector2D-Grafik-Klasse.

Schritt 7: Ich nenne das Hauptprojekt von Part5 nach Testbed um, da das fester Bestandteil der Physik-Engine ist.

Schritt 8: Ich lagere aus dem Testbed-Projekt die Controls SimulatorControl und EditorControl aus.

Schritt 9: Der Simulator bekommt ein ManipulateJointsControl

Schritt 10: Ich erstelle im Editor und Simulator die Joints: Revolute, Prismatic, Weld und Wheel

Schritt 11: RigidBody um die Property CollideExcludeList erweitert. Anpassung der CollisionHelper-Klasse.

BasisConstraints für die Gelenke

Alle Gelenke setzen sich aus mehreren Einzelconstraints zusammen. So gibt es z.B. beim prismatischen Gelenk eine Kraft, welche den Ankerpunkt von ein Körper auf die Bewegungssachse

von ein anderen Körper drückt (erste Constraint) und es gibt eine weitere Constraint, welche verhindert, dass sich der Körper dreht. Ich möchte hier 5 unterschiedliche Gelenke simulieren: Distanz, Prismatic, Wheel, Weld, Revolute. All diese Gelenke bestehen aus Constraints, die teilweise gleich sind und teilweise nur bei den einen Gelenk vorkommen. All die Constraints, die bei mehr als ein Gelenk vorkommen habe ich habe unter den Name *BasisConstraint* zusammen gefasst. Bevor ich im Detail auf die einzelnen Gelenke eingehe, möchte ich zuerst die Basisconstraints erklären. Mit diesen Wissen im Hinterkopf kann man dann die Gelenke verstehen.

Mathevorwissen, was für die Constraint-Herleitungen benötigt wird

Ableitung von Vektoren nach der Zeit

Wenn man den Schwerpunkt x_1 eines Körpers nach der Zeit ableitet dann gilt:

$$\frac{\partial(x_1)}{\partial t} = v_1$$

Wenn man ein Hebelarm nach der Zeit ableiten will, dann muss man laut „Rigid Body Simulation - David Baraff 2001“ (2-7) folgende Regel anwenden:

$$\frac{\partial(r_1)}{\partial t} = \omega_1 \times r_1$$

Kreuzprodukt

Wir arbeiten in unserer 2D-Physikengine die ganze Zeit immer mit 2D-Vektoren. In Wirklichkeit sind das aber 3D-Vektoren, wo bei allen Vektoren die Z-Komponente 0 ist. Für den AngularVelocity (Omega)-Vektor gilt aber die Regel, dass X und Y Null sind und dafür beim Z-Wert ein Wert ungleich 0 steht.

Das bedeutet das Kreuzprodukt zwischen ein Omega und ein Hebelarmvektor sieht ausgeschrieben so aus:

$$\omega_1 \times r_1 = \begin{bmatrix} 0 \\ 0 \\ \omega_1 \end{bmatrix} \times \begin{bmatrix} r_{1x} \\ r_{1y} \\ 0 \end{bmatrix} = \begin{bmatrix} -\omega_1 * r_{1y} \\ \omega_1 * r_{1x} \\ 0 \end{bmatrix}$$

Will ich stattdessen das Kreuzprodukt zwischen zwei XY-Vektoren bilden dann sieht das so aus:

$$a \times b = \begin{bmatrix} a_x \\ a_y \\ 0 \end{bmatrix} \times \begin{bmatrix} b_x \\ b_y \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ a_x * b_y - a_y * b_x \end{bmatrix}$$

D.h. Es gibt nur zwei Arten von Vektoren mit denen wir rechnen. Entweder steht bei X und Y etwas oder nur bei Z. Wenn mit dem Kreuzprodukt gerechnet wird, dann entsteht ein XY-Vektor oder ein Z-Vektor. Es passiert aber nie, dass ein XYZ-Vektor entsteht.

Wenn ich die Reihenfolge beim Kreuzprodukt ändere, dann ändert sich das Vorzeichen:

$$a \times b = -b \times a$$

Skew-Matrix für Winkelgeschwindigkeit und Hebelarm

Wenn man eine Gleichung umstellen will, wo ein Kreuzprodukt vorkommt, dann kann man mit der Skew-Matrix sich die Arbeit erleichtern. Dazu gibt es in Wikipedia folgende Hilfe:

https://en.wikipedia.org/wiki/Cross_product#Conversion_to_matrix_multiplication

Wenn ich $\omega_1 \times r_1$ rechen will, dann kann ich anstelle der 3*3-Skew-Matrix auch ein 2D-Vektor verwenden, da sowohl r1-Z Null ist als auch w1-X und w1-Y.

$$\omega_1 \times r_1 = \mathbf{a} \times \mathbf{b} \quad \omega_1 = \mathbf{a} \quad r_1 = \mathbf{b}$$

$$\mathbf{a} \times \mathbf{b} = [\mathbf{b}]_{\times}^T \mathbf{a} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ b_2 & -b_1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad [\mathbf{b}]_{\times}^T = \begin{bmatrix} -b_2 & b_1 \\ a_3 & 0 \end{bmatrix}$$

Somit kann ich folgende Umstellungen machen:

$$\overline{\omega}_1 \times r_1 = \begin{bmatrix} 0 \\ 0 \\ \omega_1 \end{bmatrix} \times \begin{bmatrix} r_{1x} \\ r_{1y} \\ 0 \end{bmatrix} = \begin{bmatrix} -\omega_1 * r_{1y} \\ \omega_1 * r_{1x} \\ 0 \end{bmatrix} = [r_1]_x^T * \omega_1 = \begin{bmatrix} -r_{1y} \\ r_{1x} \\ 0 \end{bmatrix} * \omega_1$$

Ganz am Anfang ist das Omega noch ein 3er Vektor. Um das zu zeigen, habe ich ein Strich drüber gemacht. Alle anderen Omegas ohne Strich sind skalare Zahlen.

Wenn man also einfach nur mit der Skew-Matrix der Kreuzprodukt in eine Multiplikation umwandeln will: $\overline{\omega}_1 \times r_1 = [r_1]_x^T * \omega_1$ dann ist das Omega beim Kreuzprodukt auf der linken Seite noch ein 3er Vektor und bei der Multiplikation auf der rechten Seite ist es dann ein skalarer Wert.

Skew-Matrix für zwei XY-Vektoren

Wende ich die Skew-Matrix auf den linken Produkt-Operant an erhalte ich:

$$[a]_x^T * b = \begin{bmatrix} -a_y \\ a_x \\ 0 \end{bmatrix}^T * \begin{bmatrix} b_x \\ b_y \\ 0 \end{bmatrix} = a_x * b_y - a_y * b_x$$

Außerdem wissen wir über das Kreuzprodukt:

$$a \times b = \begin{bmatrix} a_x \\ a_y \\ 0 \end{bmatrix} \times \begin{bmatrix} b_x \\ b_y \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ a_x * b_y - a_y * b_x \end{bmatrix}$$

Da die Z-Komponente von $a \times b$ das gleiche ist wie $[a]_x^T * b$, gilt

$$[a]_x^T * b = a \times b$$

Ich kann die Skew-Matrix auch auf den rechten Produkt-Operanten anwenden:

$$a * [b]_x^T = \begin{bmatrix} a_x \\ a_y \\ 0 \end{bmatrix}^T * \begin{bmatrix} -b_y \\ b_x \\ 0 \end{bmatrix} = a_y * b_x - a_x * b_y$$

Wir wissen außerdem das gilt:

$$b \times a = \begin{bmatrix} b_x \\ b_y \\ 0 \end{bmatrix} \times \begin{bmatrix} a_x \\ a_y \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ a_y * b_x - a_x * b_y \end{bmatrix}$$

Da auch hier die Z-Komponente von $b \times a$ das gleiche wie $a * [b]_x^T$ ist gilt:

$$a * [b]_x^T = b \times a$$

Wir fassen unser Wissen zusammen:

$$1: a \times b = -b \times a$$

$$2: a \times b + a \times c = a \times (b + c)$$

$$3: \overline{\omega}_1 \times r_1 = [r_1]_x^T * \omega_1 \rightarrow \text{Hier wird ein 3D-Omega in ein Skalar-Omega umgewandelt}$$

4: $[a]_x^T * b = a \times b \rightarrow a \text{ und } b \text{ sind XY-Vektoren}$

5: $a * [b]_x^T = b \times a$

Gerade Regel 4 und 5 sind wichtig. Man darf nicht einfach die Reihenfolge von den Produkt operanten ändern, da hier ein Vektor mit ein Skew-Vektor multipliziert wird. Will ich eine Skew-Matrix wieder zurück in ein Kreuzprodukt umwandeln, dann ist es entscheidend, ob der erste oder zweite Operant die Skew-Matrix ist.

Bei Regel 3 auf der rechten Seite dagegen ist es erlaubt, dass die Reihenfolge sich ändert, da Omega ein Skalar ist. Ich habe mit der Skew-Matrix das Omega quasi befreit.

PointToLine-Constraint

Bei diesen Constraint geht es darum, dass man ein Körper hat, welcher eine Achse/Schiene definiert. Der Ankerpunkt von einem zweiten Körper soll sich dann nur auf der vorgegebenen Achse bewegen dürfen.

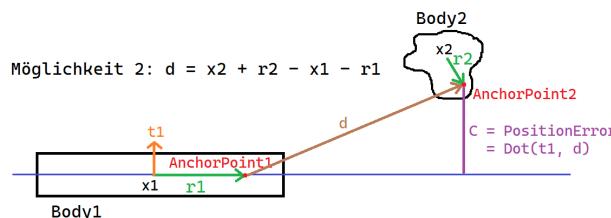
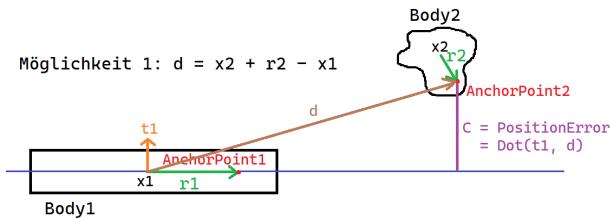
Die blaue Achse im Bild ist dadurch definiert, dass sie eine Linie ist, welche durch x_1 und AnchorPoint1 verläuft. Dreht sich Körper1, dann dreht sich auch die blaue Achse. Der Nutzer legt durch Anchorpunkt1 fest, in welche Richtung die Achse verlaufen soll und durch Anchorpunkt2, welcher Punkt von Körper2 auf dieser blauen Linie laufen soll.

Die Aufgabe von der PointToLine-Constraint ist es den Ankerpunkt2 auf die blaue Linie zu drücken.

Über folgende Schritte erfolgt die Herleitung für diese Constraint:

Schritt 1: Die PositionConstraint misst den senkrechten Abstand von Ankerpunkt2 zur blauen Linie

Um zu sehen, welchen Abstand AnchorPoint2 zur blauen Linie hat reicht es, wenn ich den Richtungsvektor d mit den normierten Tangentvektor zu r_1 multipliziere. Dadurch projiziere ich d auf t_1 und erhalte somit den PositionError für unsere Constraint. Dabei gibt es zwei Möglichkeiten (Siehe Bild), wie ich d definieren kann:



Wir werden beide Möglichkeiten herleiten. Dabei werden wir sehen, dass am Ende das gleiche Ergebnis raus kommt. Nur der Weg dort hin ist etwas anders.

Variante 1: $d=x_2+r_2-x_1$

Die Positionconstraint sieht so aus:

$$C: t_1 * (x_2 + r_2 - x_1) = 0 \quad d = x_2 + r_2 - x_1 \quad t_1 = \text{Normalize}(r_1).Spin90()$$

Schritt 2: Velocity Constraint für Variante 1

Die Ableitung der Position x_1 ergibt die Geschwindigkeit v_1 . Die Ableitung des Hebelarms r_1 ergibt das Kreuzprodukt zwischen den Hebelarm und die Winkelgeschwindigkeit. Die Ableitung von t_1 ergibt Cross(w_1, t_1). Für die Ableitung des Dot-Produktes wende ich die Produktregel an.

$$\dot{C} : (\omega_1 \times t_1) * d + t_1 * (v_2 + \omega_2 \times r_2 - v_1) = 0$$

Schritt 3: Den J- und V-Vektor extrahieren (Variante 1)

Über die Skew-Matrix bekomme ich das Kreuzprodukt weg: $\overline{\omega_1} \times r_1 = [r_1]_x^T * \omega_1$

$$\dot{C} : [t_1]_x^T \omega_1 * d + t_1 * v_2 + t_1 * [r_2]_x^T \omega_2 - t_1 * v_1 = 0$$

Beim vorherigen Schritt waren die Omegas noch Vektoren. Nun sind es skalare Werte. Ich nutze nun diese beiden Regeln um die Skew-Vektor mal Vektor-Multiplikation in eine Kreuzprodukt zurück zu verwandeln: $[a]_x^T * b = a \times b$ $a * [b]_x^T = b \times a$

$$\dot{C} : t_1 \times d * \omega_1 + t_1 * v_2 + r_2 \times t_1 * \omega_2 - t_1 * v_1 = 0 \rightarrow \text{Ich sortiere nach } v1, w1, v2, w2$$

$\dot{C} : -t_1 * v_1 + t_1 * d * \omega_1 + t_1 * v_2 + r_2 \times t_1 * \omega_2 = 0 \rightarrow \text{Ich ersetze } t_1 \times d = -d \times t_1 \text{ da der W1-Term wegen der K-Formel aus Schritt 4 negativ sein muss. Nun kann ich den V-Vektor extrahieren:}$

$$\dot{C} : J * V = \begin{bmatrix} -t_1 \\ -d \times t_1 \\ t_1 \\ r_2 \times t_1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0 \rightarrow \text{Ich ersetze } d = x_2 + r_2 - x_1$$

$$\dot{C} : J * V = \begin{bmatrix} -t_1 \\ -(x_2 + r_2 - x_1) \times t_1 \\ t_1 \\ r_2 \times t_1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0$$

Variante 2: $d=x_2+r_2-x_1-r_1$

Die Positionconstraint sieht hier so aus:

$$C : t_1 * (x_2 + r_2 - x_1 - r_1) = 0 \quad d = x_2 + r_2 - x_1 - r_1$$

Die VelocityConstraint erhalte ich über die Ableitung:

$$\dot{C} : (\omega_1 \times t_1) * d + t_1 * (v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1) = 0$$

$$\dot{C} : [t_1]_x^T \omega_1 * d + t_1 * v_2 + t_1 * [r_2]_x^T \omega_2 - t_1 * v_1 - t_1 * [r_1]_x^T \omega_1 = 0$$

Omega wurde durch Skew befreit. Nun zurück: $[a]_x^T * b = a \times b$ $a * [b]_x^T = b \times a$

$$\dot{C} : t_1 \times d * \omega_1 + t_1 * v_2 + r_2 \times t_1 * \omega_2 - t_1 * v_1 - r_1 \times t_1 * \omega_1 = 0 \rightarrow \text{Sortiere nach } v1, w1, v2, w2$$

$$\dot{C} : -t_1 * v_1 + (t_1 \times d - r_1 \times t_1) * \omega_1 + t_1 * v_2 + r_2 \times t_1 * \omega_2 = 0$$

$$(t_1 \times d - r_1 \times t_1) = t_1 \times d + t_1 \times r_1 = t_1 \times (d + r_1) = -(d + r_1) \times t_1$$

$$\dot{C} : J * V = \begin{bmatrix} -t_1 \\ -(d + r_1) \times t_1 \\ t_1 \\ r_2 \times t_1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0 \rightarrow \text{Ersetze } d = x_2 + r_2 - x_1 - r_1$$

$$\dot{C}: J * V = \begin{bmatrix} -t_1 \\ -(x_2 + r_2 - x_1) \times t_1 \\ t_1 \\ r_2 \times t_1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0$$

Man sieht, dass diese Variante genau gleich wie Variante 1 aussieht. Sie ist von der Herleitung nur umständlicher. Der Grund, warum ich überhaupt auf die Idee gekommen bin das so zu machen ist, weil Erin Catto das d so definiert (siehe Zeile 29):

https://github.com/erincatto/box2d/blob/main/src/dynamics/b2_wheel_joint.cpp#L33C7-L33C7

```

28     // Linear constraint (point-to-line)
29     // d = pB - pA = xB + rB - xA - rA
30     // C = dot(ay, d)
31     // Cdot = dot(d, cross(wA, ay)) + dot(ay, vB + cross(wB, rB) - vA - cross(wA, rA))
32     //      = -dot(ay, vA) - dot(cross(d + rA, ay), wA) + dot(ay, vB) + dot(cross(rB, ay), vB)
33     // J = [-ay, -cross(d + rA, ay), ay, cross(rB, ay)]

```

Schritt 4: Den K-Faktor ausrechnen

Ich nutze die Formel (2.6) von Seite 7 aus „3D Constraint Derivations for Impulse Solvers - Marijn 2015“ um K zu berechnen:

$$\boxed{\mathbf{J} = [-\mathbf{L}_a, -\mathbf{A}_a, \mathbf{L}_b, \mathbf{A}_b] \quad \mathbf{JM}^{-1}\mathbf{J}^T = \mathbf{L}_a \mathbf{L}_a^T m_a^{-1} + \mathbf{A}_a \mathbf{I}_a^{-1} \mathbf{A}_a^T + \mathbf{L}_b \mathbf{L}_b^T m_b^{-1} + \mathbf{A}_b \mathbf{I}_b^{-1} \mathbf{A}_b^T \quad \mathbf{K} = \mathbf{JM}^{-1}\mathbf{J}^T}$$

Ich definiere $L_a = L_b = t_1$ $A_a = (x_2 + r_2 - x_1) \times t_1$ $A_b = r_2 \times t_1$ Da t1 normiert ist, ergibt $t_1 * t_1 = 1$

$$K = m_a + A_a^2 * I_a + m_b + A_b^2 * I_b$$

Schritt 5: Die PointToLine-Klasse erstellen

Da die Position- und Velocity-Constraint eine einzelne Float-Zahl zurück gibt, verwenden wir hier eine ILinear1DConstraint. Der ResolverHelper benötigt für diese Constraintart R1, R2 und ForceDirection. Diese 3 Angaben können wir aus dem J-Vektor ablesen (siehe Markierung). T1 haben wir so definiert, dass es die Tangente vom Richtungsvektor ist, welcher von Body1.Center nach Anchor1 zeigt.

```

29     public PointToLine(ConstraintConstructorData data, IPuntoLineJoint joint)
30     {
31         this.joint = joint;
32         var s = data.Settings;
33
34         B1 = joint.B1;
35         B2 = joint.B2;
36
37         //d = x2+r2 - x1
38         R1 = joint.Anchor2 - joint.B1.Center;           //R1=d=x1+r2-x1
39         R2 = joint.Anchor2 - joint.B2.Center;           //R2=r2
40
41         AccumulatedImpulse = s.DoWarmStart ? joint.AccumulatedPointToLineImpulse : 0;
42
43         ForceDirection = (joint.Anchor1 - B1.Center).Spin90().Normalize();
44
45         Vec2D d = joint.Anchor2 - B1.Center;
46         float s1 = Vec2D.ZValueFromCross(d, ForceDirection);
47         float s2 = Vec2D.ZValueFromCross(R2, ForceDirection);            $K = m_a + A_a^2 * I_a + m_b + A_b^2 * I_b$ 
48
49         float invMass = B1.InverseMass + s1 * s1 * B1.InverseInertia + B2.InverseMass + s2 * s2 * B2.InverseInertia;
50         ImpulseMass = 1f / invMass;
51
52         float biasFactor = s.DoPositionalCorrection ? s.PositionalCorrectionRate : 0.0f;
53         Bias = -biasFactor * data.InvDt * (d * ForceDirection);
54     }

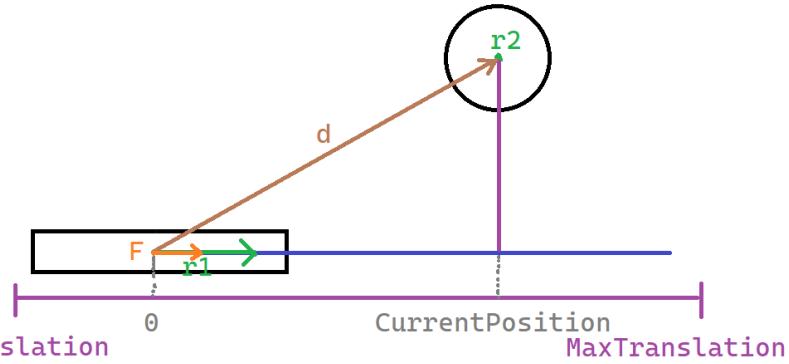
```

Siehe: PhysicEngine/CollisionResolution/SequentielImpulse/Constraints/BasicConstraints/PointToLine.cs

$$\dot{C}: J * V = \begin{bmatrix} -t_1 \\ -(x_2 + r_2 - x_1) \times t_1 \\ t_1 \\ r_2 \times t_1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0$$

MinMaxTranslation-Constraint

Bei dieser Constraint soll die CurrentPosition (siehe Bild) sich nur innerhalb der MinTranslation/MaxTranslation-Werte bewegen dürfen. CurrentPosition erhalte ich, indem ich d, was über $d = x_2 + r_2 - x_1$ definiert ist, auf die blaue Linie projiziere, indem ich $CurrentPosition = F * d$ rechne. Die blaue Linie ist durch das Zentrum von Body1 und die Richtung r1 (dreht sich mit Body1 mit) definiert.



Schritt 1: Die PositionConstraint misst den Abstand von CurrentPosition zum Schrankenwert

$$C: F \cdot d = Bias \quad \text{mit} \quad d = x_2 + r_2 - x_1 \quad F = \text{Normalize}(Anchor_1 - x_1)$$

Wenn $F \cdot d > \text{MaxTranslation}$ ist, dann ist $\text{Bias} = (\text{MaxTranslation} - F \cdot d)$. Wenn $F \cdot d < \text{MinTranslation}$, dann ist $\text{Bias} = (\text{MinTranslation} - F \cdot d)$.

Schritt 2: Die Ableitung ergibt die Velocity-Constraint

Unter Nutzung der Regel $\frac{\partial(x_1 + r_1)}{\partial t} = v_1 + \omega_1 \times r_1$ leiten wir nach der Zeit ab:

$$\dot{C}: (\omega_1 \times F) \cdot d + F \cdot (v_2 + \omega_2 \times r_2 - v_1) = 0$$

Schritt 3: Den J- und V-Vektor extrahieren

Unter Nutzung der Skew-Matrix $\omega_1 \times r_1 = [r_1]_x^T \omega_1$ stelle ich etwas um:

$$\dot{C}: [F]_x^T \omega_1 \cdot d + F \cdot v_2 + F \cdot [r_2]_x^T \omega_2 - F \cdot v_1 = 0 \rightarrow \text{Zurück: } [a]_x^T \cdot b = a \cdot b \quad a \cdot [b]_x^T = b \cdot a$$

$$\dot{C}: F \times d \cdot \omega_1 + F \cdot v_2 + r_2 \times F \cdot \omega_2 - F \cdot v_1 = 0 \rightarrow \text{sortiere nach } v1, w1, v2, w2$$

$$\dot{C}: -F \cdot v_1 + F \times d \cdot \omega_1 + F \cdot v_2 + r_2 \times F \cdot \omega_2 = 0 \rightarrow V \text{ extrahieren}$$

$$\dot{C}: J \cdot V = \begin{bmatrix} -F \\ -d \times F \\ F \\ r_2 \times F \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0$$

Schritt 4: Den K-Faktor ausrechnen

Unter Nutzung von Marijns-Regel

$$\mathbf{J} = [-\mathbf{L}_a, -\mathbf{A}_a, \mathbf{L}_b, \mathbf{A}_b] \quad \mathbf{J} \mathbf{M}^{-1} \mathbf{J}^\top = \mathbf{L}_a \mathbf{L}_a^\top m_a^{-1} + \mathbf{A}_a \mathbf{I}_a^{-1} \mathbf{A}_a^\top + \mathbf{L}_b \mathbf{L}_b^\top m_b^{-1} + \mathbf{A}_b \mathbf{I}_a^{-1} \mathbf{A}_b^\top \quad \mathbf{K} = \mathbf{J} \mathbf{M}^{-1} \mathbf{J}^\top$$

erhalten wir: $F \cdot F = 1 \quad s_1 = d \times F \quad s_2 = r_2 \times F$

$$K = m_a + s_1^2 * I_a + m_b + s_2^2 * I_b$$

Schritt 5: Die MinMaxTranslation-Klasse erstellen

Während der Herleitung habe ich die PositionConstraint mit Bias gleichgesetzt, was entweder für den MinTranslation oder MaxTranslation-Wert steht. Da die Angabe in Pixeln erfolgen muss und ich bei der Definition aber den Min/Max-Translation-Wert in Verhältnis zur DistanceOnStart angebe, muss ich das mit der DistanceOnStart-Multiplikation bei Zeile 57/58) wieder in ein Pixelwert umrechnen. Über PositionCorrection erfolgt nun die Korrektur. Wenn currentDistance z.B. 10 Pixel hinter der Max-Schranke liegt, dann wirkt der Impuls nur ziehend und er zieht dann um 10 Pixel den Ankerpunkt2 zurück. Sollte aber die currentDistance innerhalb der Min-Max-Schranke liegen, dann ist die ImpulseMass Null, was dazu führt, dass gar kein Impuls angewendet wird.

```

34     public MinMaxTranslation(ConstraintConstructorData data, IMinMaxTranslationJoint joint)
35     {
36         this.joint = joint;
37         var s = data.Settings;
38
39         Vec2D d = joint.Anchor2 - joint.B1.Center;
40
41         B1 = joint.B1;
42         B2 = joint.B2;
43         R1 = d;
44         R2 = joint.Anchor2 - joint.B2.Center;
45
46         AccumulatedImpulse = s.DoWarmStart ? joint.AccumulatedMinMaxImpulse : 0;
47
48         ForceDirection = (joint.Anchor1 - joint.B1.Center).Normalize();
49         float s1 = Vec2D.ZValueFromCross(d, ForceDirection);
50         float s2 = Vec2D.ZValueFromCross(R2, ForceDirection);
51
52         float effectiveMass = 1.0f / (B1.InverseMass + s1 * s1 * B1.InverseInertia + B2.InverseMass + s2 * s2 * B2.InverseInertia);
53
54
55         float biasFactor = s.DoPositionalCorrection ? s.PositionalCorrectionRate : 0.0f;
56
57         float min = joint.DistanceOnStart * joint.MinTranslation; //Umrechnung der Min-Max-Werte in Pixel
58         float max = joint.DistanceOnStart * joint.MaxTranslation;
59         float currentDistance = ForceDirection * d;
60
61         if (currentDistance > max)
62         {
63             Bias = biasFactor * data.InvDt * (max - currentDistance);
64             ImpulseMass = effectiveMass;
65             MaxImpulse = 0; //Impuls soll nur ziehend wirken
66         }
67
68         if (currentDistance < min)
69         {
70             Bias = biasFactor * data.InvDt * (min - currentDistance);
71             ImpulseMass = effectiveMass;
72             MinImpulse = 0; //Impuls soll nur drückend wirken
73         }
74     }

```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Constraints/BasicConstraints/MinMaxTranslation.cs

$$\dot{C} : J \cdot V = \begin{bmatrix} -F \\ -d \times F \\ F \\ r_2 \times F \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0$$

$$K = m_a + s_1^2 * I_a + m_b + s_2^2 * I_b$$

TranslationMotor-Constraint

Bei diesen Constraint geht es darum, dass ein Ankerpunkt per Motorkraft zu einer angegebenen Position bewegt wird (Variante1) oder das der Ankerpunkt mit einer bestimmten Geschwindigkeit bewegt wird

(Variante 2). Ich nutze hier die gleiche Klasse wie MinMaxTranslation nur dass ich diesmal den Bias-Wert bei Variante 1 nicht auf den Min- oder Max-Wert sondern auf den MotorPosition-Wert setze, wenn ich per PositionCorrection eine bestimmte Auslenkung einstellen will.

Will ich Body2 mit einer bestimmten Geschwindigkeit einfach nur bewegen, dann setze ich den Bias-Wert auf den Bias-Velocity-Wert 'MotorSpeed'.

```

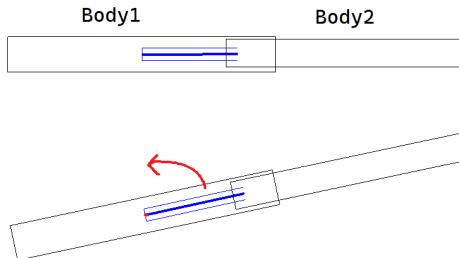
34     public TranslationMotor(ConstraintConstructorData data, IITranslationMotorJoint joint)
35     {
36         this.joint = joint;
37         var s = data.Settings;
38
39         Vec2D d = joint.Anchor2 - joint.B1.Center;
40
41         B1 = joint.B1;
42         B2 = joint.B2;
43         R1 = d;
44         R2 = joint.Anchor2 - joint.B2.Center;
45
46         AccumulatedImpulse = s.DoWarmStart ? joint.AccumulatedTranslationMotorImpulse : 0;
47
48         ForceDirection = (joint.Anchor1 - joint.B1.Center).Normalize();
49         float s1 = Vec2D.ZValueFromCross(d, ForceDirection);
50         float s2 = Vec2D.ZValueFromCross(R2, ForceDirection);
51
52         this.ImpulseMass = 1.0f / (B1.InverseMass + s1 * s1 * B1.InverseInertia + B2.InverseMass + s2 * s2 * B2.InverseInertia);
53
54
55         float maxImpulse = data.Dt * joint.MaxMotorForce;
56         this.MinImpulse = -maxImpulse;
57         this.MaxImpulse = maxImpulse;
58
59         if (joint.Motor == IPublicPrismaticJoint.TranslationMotor.GoToReferencePosition)
60         {
61             float biasFactor = s.DoPositionalCorrection ? s.PositionalCorrectionRate : 0.0f;
62             float currentDistance = ForceDirection * d;
63             float pos = joint.DistanceOnStart * joint.MotorPosition;
64             Bias = biasFactor * data.InvDt * (pos - currentDistance);
65         }
66
67         if (joint.Motor == IPublicPrismaticJoint.TranslationMotor.IsMoving)
68         {
69             Bias = joint.MotorSpeed;
70         }
71     }

```

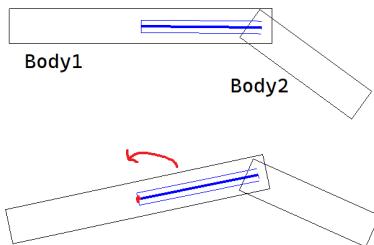
Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Constraints/BasicConstraints/TranslationMotor.cs

FixAngular-Constraint

Bei dieser Constraint darf Body2 sich nicht gegenüber Body1 drehen können. Wenn Body1 sich um sein Zentrum um 20 Grad dreht, dann muss sich Body2 auch um 20 Grad drehen. In diesen Beispiel haben beide Körper zum Definitionszeitpunkt einen Winkel von 0 Grad. D.h. Die Winkeldifferenz zwischen den Körper muss durch eine Constraint steht bei 0 Grad gehalten werden.



Das gleiche gilt auch für Körper, die initial eine Winkeldifferenz von z.B. 40 Grad haben. Am Anfang hat Body1 eine Ausrichtung von 0 Grad und Body2 von -40 Grad. Dreht sich Body1 nun um 20 Grad, dann muss Körper 2 nun eine Ausrichtung von $-40+20=-20$ Grad bekommen.



Wir definieren: $A = Body2.Angle - Body1.Angle$ zum Definitionszeitpunkt. D.h. A ist ein fester Wert, der sich beim simulieren dann nicht mehr ändert.

Schritt 1: Die PositionConstraint misst, um wie weit sich der Winkelabstand gegenüber der Startkonfiguration geändert hat.

$$C: \phi_2 - \phi_1 = A$$

Schritt 2: Die Ableitung ergibt die Velocity-Constraint

$$\dot{C}: \omega_2 - \omega_1 = 0$$

Schritt 3: Den J- und V-Vektor extrahieren

$$\dot{C}: J * V = \begin{bmatrix} 0 \\ -1 \\ 0 \\ 1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0$$

Schritt 4: Den K-Faktor ausrechnen

Unter Nutzung von Marijns-Regel

$$J = [-L_a, -A_a, L_b, A_b] \quad JM^{-1}J^T = L_a L_a^T m_a^{-1} + A_a I_a^{-1} A_a^T + L_b L_b^T m_b^{-1} + A_b I_b^{-1} A_b^T \quad K = JM^{-1}J^T$$

erhalten wir: $L_a * L_a^T = 0 \quad A_a * A_a^T = 1 \quad L_b * L_b^T = 0 \quad A_b * A_b^T = 1$

$$K = I_a + I_b$$

Schritt 5: Die Erstellung der Constraint-Klasse

Bis jetzt hatten wir immer nur lineare Impulse, die auf Ankerpunkte wirken. In diesen Fall wollen wir einen Drehimpuls wirken lassen. Um das zu bewerkstelligen habe ich das IAngularConstraint-Interface erstellt, welches für Constraints verwendet werden soll, welche ein Drehimpuls wirken lassen.

Im Konstruktor berechne ich die ImpulseMass-Variable, was unserem inversen K entspricht und mit dem PositionError-Wert das Bias per PositionCorrection-Formel.

```

30  public FixAngular(ConstraintConstructorData data, IAngularJoint joint)
31  {
32      this.joint = joint;
33      var s = data.Settings;
34
35      B1 = joint.B1;
36      B2 = joint.B2;
37
38      AccumulatedImpulse = s.DoWarmStart ? joint.AccumulatedAngularImpulse : 0;
39
40      ImpulseMass = 1f / (B1.InverseInertia + B2.InverseInertia);
41
42      float positionError = B2.Angle - B1.Angle - joint.AngularDifferenceOnStart;
43
44      float biasFactor = s.DoPositionalCorrection ? s.PositionalCorrectionRate : 0.0f;
45      Bias = -biasFactor * data.InvDt * positionError;
46  }

```

Siehe: PhysicEngine/CollisionResolution/SequentielImpulse/Constraints/BasicConstraints/FixAngular.cs

In der SIStep-Funktion rechne ich den Velocity-Constraint-Wert in ein Impuls um.

```

52  public void DoSingleSIStep(float invDt)
53  {
54      float angularVelocity = B2.AngularVelocity - B1.AngularVelocity;
55      float angularImpulse = ImpulseMass * (Bias - angularVelocity);
56
57      ResolverHelper.ApplyAngularImpulse(this, angularImpulse);
58
59      SaveImpulse();
60  }

```

Und so wende ich den Drehimpuls an:

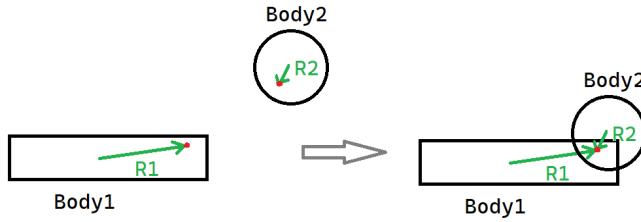
```

220  internal static void ApplyAngularImpulse(IConstraint c, float angularImpulse)
221  {
222      c.B1.AngularVelocity -= angularImpulse * c.B1.InverseInertia;
223      c.B2.AngularVelocity += angularImpulse * c.B2.InverseInertia;
224  }

```

PointToPoint-Constraint

Bei dieser Constraint geht es darum, dass zwei Ankerpunkte die gleiche Position haben. Gegeben sind zwei Objekte mit jeweils ein Ankerpunkt. Die Constraint drückt die beiden Punkte zusammen.



Schritt 1: Die PositionConstraint misst den Abstand der beiden Ankerpunkte

$$C: (x_2 + r_2) - (x_1 + r_1) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Schritt 2: Die Ableitung ergibt die Velocity-Constraint

$$\dot{C}: v_2 + \omega_2 \times r_2 - v_1 - \omega_1 \times r_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Schritt 3: Den J- und V-Vektor extrahieren mit der Regel $\overline{\omega_1} \times r_1 = [r_1]_x^T * \omega_1$

$$\dot{C}: v_2 + [r_2]_x^T \omega_2 - v_1 - [r_1]_x^T \omega_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow \text{sortiere nach } v1, w1, v2, w2$$

$$\dot{C}: -v_1 - [r_1]_x^T \omega_1 + v_2 + [r_2]_x^T \omega_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow [r_1]_x^T = \begin{bmatrix} -r_{1y} \\ r_{1x} \end{bmatrix}$$

$$\dot{C}: -\begin{bmatrix} v_{1x} \\ v_{1y} \end{bmatrix} - \begin{bmatrix} -r_{1y} \\ r_{1x} \end{bmatrix} \omega_1 + \begin{bmatrix} v_{2x} \\ v_{2y} \end{bmatrix} + \begin{bmatrix} -r_{2y} \\ r_{2x} \end{bmatrix} \omega_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\dot{C}: J^* V = \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ r_{1y} & -r_{1x} \\ 1 & 0 \\ 0 & 1 \\ -r_{2y} & r_{2x} \end{bmatrix} \begin{bmatrix} v_{1x} \\ v_{1y} \\ \omega_1 \\ v_{2x} \\ v_{2y} \\ \omega_2 \end{bmatrix} = 0 \quad J = \begin{bmatrix} -1 & 0 & r_{1y} & 1 & 0 & -r_{2y} \\ 0 & -1 & -r_{1x} & 0 & 1 & r_{2x} \end{bmatrix}$$

Schritt 4: Den K-Faktor ausrechnen

$$J^* M^{-1} = \begin{bmatrix} m_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & m_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & I_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & m_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & I_2 \end{bmatrix} \quad K = J^* M^{-1} * J^T = \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ r_{1y} & -r_{1x} \\ 1 & 0 \\ 0 & 1 \\ -r_{2y} & r_{2x} \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 & r_{1y} & 1 & 0 & -r_{2y} \\ 0 & -1 & -r_{1x} & 0 & 1 & r_{2x} \end{bmatrix} \begin{bmatrix} -m_1 & 0 & r_{1y} I_1 & m_2 & 0 & -r_{2y} I_2 \\ 0 & -m_1 & -r_{1x} I_1 & 0 & m_2 & r_{2x} I_2 \end{bmatrix} \begin{bmatrix} -m_1 & 0 & r_{1y} I_1 & m_2 & 0 & -r_{2y} I_2 \\ 0 & -m_1 & -r_{1x} I_1 & 0 & m_2 & r_{2x} I_2 \end{bmatrix} \begin{bmatrix} K1 & K2 \\ K3 & K4 \end{bmatrix}$$

$$K1 = m_1 + r_{1y}^2 * I_1 + m_2 + r_{2y}^2 * I_2$$

$$K2 = -r_{1x} * r_{1y} * I_1 - r_{2x} * r_{2y} * I_2$$

$$K3 = -r_{1x} * r_{1y} * I_1 - r_{2x} * r_{2y} * I_2$$

$$K4 = m_1 + r_{1x}^2 * I_1 + m_2 + r_{2x}^2 * I_2$$

Schritt 5: Die PointToPoint-Klasse erstellen

Da das eine ILinear2DConstraint ist, muss ich im Konstruktor die inverse K-Matrix und den PositionError-2D-Vektor definieren:

```

48     public PointToPoint(ConstraintConstructorData data, IPointToPointJoint joint)
49     {
50         this.joint = joint;
51
52         this.B1 = joint.B1;
53         this.B2 = joint.B2;
54         this.R1 = joint.Anchor1 - joint.B1.Center;
55         this.R2 = joint.Anchor2 - joint.B2.Center;
56
57         var s = data.Settings;
58         this.AccumulatedImpulse = s.DoWarmStart ? joint.AccumulatedPointToPointImpulse : new Vec2D(0, 0);
59
60         this.PositionError = joint.Anchor2 - joint.Anchor1;
61         this.MaxImpulse = float.MaxValue;
62
63         float x = -R1.X * R1.Y * B1.InverseInertia - R2.X * R2.Y * B2.InverseInertia;
64
65         this.InverseK = Matrix2x2.FromScalars(
66             B1.InverseMass + B2.InverseMass + R1.Y * R1.Y * B1.InverseInertia + R2.Y * R2.Y * B2.InverseInertia,
67             x,
68             x,
69             B1.InverseMass + B2.InverseMass + R1.X * R1.X * B1.InverseInertia + R2.X * R2.X * B2.InverseInertia
70         )
71         .Invert();
72     }

```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Constraints/BasicConstraints/PointToPoint.cs

Außerdem schreibt das ILinear2DConstraint-Interface die CDot-Funktion vor welche so aussieht:

```

29     public Vec2D GetCDot() //Gibt CDot=J*V zurück (Geschwindigkeit in Richtung jeder J-Zeile)
30     {
31         //Variante 1: Relative Ankerpunktgeschwindigkeit über Drehhebelarmformel: V + Cross([0, 0, AngularVelocity], [R.X, R.Y, 0])
32         Vec2D v1 = B1.Velocity + new Vec2D(-B1.AngularVelocity * R1.Y, B1.AngularVelocity * R1.X);
33         Vec2D v2 = B2.Velocity + new Vec2D(-B2.AngularVelocity * R2.Y, B2.AngularVelocity * R2.X);
34         Vec2D cDot = v2 - v1;
35
36         //Variante 2: J*V-Formel
37         float cDotX = -B1.Velocity.X + R1.Y*B1.AngularVelocity + B2.Velocity.X - R2.Y*B2.AngularVelocity;
38         float cDotY = -B1.Velocity.Y - R1.X*B1.AngularVelocity + B2.Velocity.Y + R2.X*B2.AngularVelocity;
39
40         return cDot;
41     }

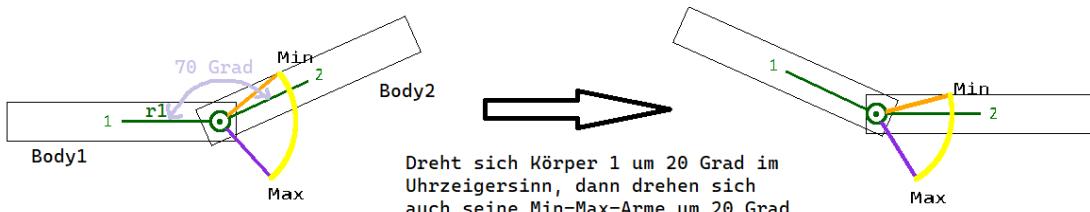
```

Ich kann hier entweder die Drehhebelarm-Formel verwenden oder die J*V-Formel.

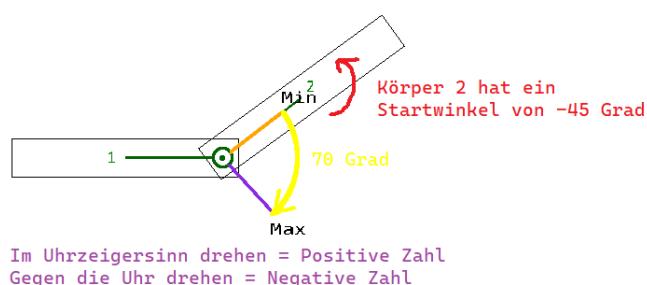
MinMaxAngular-Constraint

Wenn ich zwei Stäbe habe, welche durch ein Revolute-Joint verbunden sind, dann ist es die Aufgabe vom MinMaxAngular-Constraint den Bewegungsbereich einzuschränken. Die Definition erfolgt so, dass am Körper 1 ein Min (Orange)- und ein Max (Lila)-Schranken-Arm hängt. Der Hebelarm von Körper 2 soll sich dann nur innerhalb der Schranke (gelb markiert) bewegen können.

Dreht sich Körper 1 und somit sein grüner r1-Hebelarm, dann drehen sich im gleichen Maße auch seine Min-Max-Schrankenarme mit.



Nehmen wir mal an Body1 ist fix und die Winkeldifferenz zwischen Min und Max sind 70 Grad. Außerdem liegt zum Definitionszeitpunkt Hebelarm2 genau auf der Min-Schranke. Dann heißt das, Körper 2 darf sich um bis zu 70 Grad im Uhrzeigersinn drehen.



Ich muss mir also nur die Information merken, welchen Winkel die Körper zum Start der Simulation hatten. Hier im Beispiel hat Körper 1 die Ausrichtung von 0 Grad und Körper 2 -45 Grad. Körper 1 ist fix und dreht sich nicht. Für Körper 2 heißt das, dass seine Ausrichtung im Bereich von -45 bis (-45+70) Grad liegen darf.

Wenn Körper 1 nun nicht fix ist, dann muss ich die Winkeldifferenz zum Simulationsstartzeitpunkt mir merken. Hier z.B: ist Angle2-Angle1=-45-0=-45 Grad. Dreht sich nun Körper 1 z.B. um 10 Grad im Uhrzeigersinn, dann muss der Angle2-Wert im Bereich von (-45+10) bis (-45+10+70) Grad liegen. D.h. Für (Angle2-Angle1-StartAngleDifference) muss gelten, dass es im Bereich von 0 bis 70 Grad liegt.

Wenn der Hebelarm von Körper 2 zum Definitionszeitpunkt nun nicht genau auf der Min-Schranke liegt sondern z.B. bei Min+10 Grad = 35 Grad. Dann heißt das, dass Körper 2 darf sich um 10 Grad gegen die Uhr drehen oder um 70-10=60 Grad mit der Uhr.

Sollte zum Definitionszeitpunkt Hebelarm 2 aber schon außerhalb der Min-Max-Schranke liegen, dann muss die MinMaxAngular-Constraint zuerst einmal das Gelenk korrigieren. Auch hier ist es wichtig zu wissen, um wie viel Grad liegt es außerhalb der Schranke.

Im Konstruktor der RevoluteJoint-Klasse merke ich mir die Winkeldifferenz zwischen Body1 und Body2, den Abstand von r2 zur Min-Schranke und wie viel Min- und Max auseinander liegen.

```

61     UpdateAnchorPoints(); //Aktualisiere Anchor1/Anchor2
62
63     Vec2D r1 = (this.Body1.Center - this.Anchor1).Normalize();
64     Vec2D r2 = (this.Body2.Center - this.Anchor2).Normalize();
65     float angle = Vec2D.Angle360(r1, r2); //Winkel von r2 im Bezug zu r1
66
67
68     thisAngularDifferenceOnStart = B2.Angle - B1.Angle;
69     thisDiffToMinOnStart = (float)((angle - thisLowerAngle) * Math.PI / 180); //LowerAngle=Winkel im Bezug zu r1
70
71     float min = thisLowerAngle;
72     float max = thisUpperAngle;
73     if (thisLimitIsEnabled == false)
74     {
75         min = 0;
76         max = 360;
77     }
78     thisMinMaxDifference = max - min;
79     if (thisMinMaxDifference < 0) thisMinMaxDifference += 360; //Sorge dafür, dass min<max gilt
80     thisMinMaxDifference = (float)(MinMaxDifference * Math.PI / 180);
81
82     UpdateAnchorPoints(); //Aktualisiere CurrentPosition
83
84     thisMotorPosition = Math.Min(1, Math.Max(0, thisCurrentPosition)); //Soll-Startwert = Istwert zum Start
85
86
87     4 Verweise
88     public void UpdateAnchorPoints()
89     {
90         thisAnchor1 = MathHelp.GetWorldPointFromLocalDirection(thisB1, thisr1);
91         thisAnchor2 = MathHelp.GetWorldPointFromLocalDirection(thisB2, thisr2);
92
93         float a = B2.Angle - B1.Angle - thisAngularDifferenceOnStart + thisDiffToMinOnStart;
94         thisCurrentPosition = a / thisMinMaxDifference; //Wenn r2 im Min-Max-Bereich liegt, dann steht hier 0..1
    }
```

Siehe: PhysicEngine/Joints/RevoluteJoint.cs

Die MinMaxAngular-Constraint-Klasse orientiert sich an der FixAngular-Klasse. Nur dass der Korrekturdrehimpuls eben nur dann gemacht wird, wenn angle außerhalb der Schranke liegt. Liegt es innerhalb, dann ist die ImpulseMass Null, was dafür sorgt, dass kein Impuls angewendet wird.

```

24     public MinMaxAngular(ConstraintConstructorData data, IMinMaxAngularJoint joint)
25     {
26         this.joint = joint;
27         var s = data.Settings;
28
29         B1 = joint.B1;
30         B2 = joint.B2;
31         R1 = joint.Anchor1 - joint.B1.Center;
32         R2 = joint.Anchor2 - joint.B2.Center;
33
34         AccumulatedImpulse = s.DoWarmStart ? joint.AccumulatedMinMaxAngularImpulse : 0;
35
36         float impulseMass = 1f / (B1.InverseInertia + B2.InverseInertia);
37
38         float biasFactor = s.DoPositionalCorrection ? s.PositionalCorrectionRate : 0.0f;
39
40         float angle = B2.Angle - B1.Angle - joint.AngularDifferenceOnStart + joint.DiffToMinOnStart;
41         if (angle < 0)
42         {
43             this.ImpulseMass = impulseMass;
44             MinImpulse = 0; //Drehe nur so, dass angle erhöht wird
45             Bias = -biasFactor * data.InvDt * angle;
46         }
47         if (angle > joint.MinMaxDifference)
48         {
49             this.ImpulseMass = impulseMass;
50             MaxImpulse = 0; //Drehe nur so, dass angle verringert wird
51             Bias = -biasFactor * data.InvDt * (angle - joint.MinMaxDifference);
52         }
53     }

```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Constraints/Revolute/MinMaxAngular.cs

AngularMotor-Constraint

Mit der AngularMotor-Constraint kann ein Revolutejoint gesagt werden, welchen Sollwinkel es hat. So könnte man dann ein Roboterarm bauen, wo ich für jedes Gelenk vom Arm einmalig im Editor den erlaubten MinMax-Bereich festlege. Während der Simulation kann ich dann über den MotorPosition-Wert der PhysikEngine sagen, welche Sollwerte die Gelenke vom Roboterarm haben. Die AngularMotor-Constraint versucht den Arm laut der vorgegebenen Sollwinkel auszurichten, indem sie auf die Armsegmente Rotationskräfte wirken lässt.

Sie schaut dazu, wo innerhalb des MinMax-Bereichs der Hebelarm von Body2 liegt und wie viel Grad Abstand der Hebelarm zum Sollwinkelwert hat. Wenn MotorPosition den Wert 0 hat, dann heißt dass, der Hebelarm2 soll auf die Min-Schranke bewegt werden und wenn sie den Wert 1 hat, dann soll der Hebelarm auf der Max-Schranke liegen.

Die AngularMotor-Klasse orientiert sich an der FixAngular-Klasse. Hier wird beachtet, dass der r2-Hebelarm zum Definitionszeitpunkt nicht auf der Min-Schranke liegen muss. Sollte er z.B. 10 Grad drüber liegen, wird über die DiffToMinOnStart-Addition angle so korrigiert, dass man weiß, dass er ja bereits um 10 Grad gedreht wurde. Wenn der Motor-Sollwert nun auch 10 Grad ist, dann muss ich also keine Korrektur vornehmen. MotorPosition ist eine Zahl von 0 bis 1.

```

33     public AngularMotor(ConstraintConstructorData data, IAngularMotorJoint joint)
34     {
35         this.joint = joint;
36         var s = data.Settings;
37
38         B1 = joint.B1;
39         B2 = joint.B2;
40         R1 = joint.Anchor1 - joint.B1.Center;
41         R2 = joint.Anchor2 - joint.B2.Center;
42
43         AccumulatedImpulse = s.DoWarmStart ? joint.AccumulatedAngularMotorImpulse : 0;
44
45         this.ImpulseMass = 1f / (B1.InverseInertia + B2.InverseInertia);
46
47         float angle = B2.Angle - B1.Angle - joint.AngularDifferenceOnStart;//angle=0..joint.MinMaxDifference
48
49         float maxImpulse = data.Dt * joint.MaxMotorTorque;
50         this.MinImpulse = -maxImpulse;
51         this.MaxImpulse = maxImpulse;
52
53         if (joint.Motor == IPublicJointAngularMotor.GoToReferenceAngle)
54         {
55             float biasFactor = s.DoPositionalCorrection ? s.PositionalCorrectionRate : 0.0f;
56
57             float setAngle = joint.MotorPosition * joint.MinMaxDifference;
58
59             Bias = biasFactor * data.InvDt * (setAngle - angle);
60         }
61
62         if (joint.Motor == IPublicJointAngularMotor.SpinAround)
63         {
64             Bias = joint.MotorSpeed;
65         }
66     }

```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Constraints/Revolute/AngularMotor.cs

Kombination von Basisconstraints

Anstatt mehrere Constraints hintereinander innerhalb des SI-Solvers zu lösen kann man die J-Zeilen aus mehreren Constraints auch nehmen und per inverse Matrix dann auf einmal lösen. Auf diese Weise sind weniger SI-Schritte nötig.

PointToLine und FixAngular als MultiConstraint

Da Erin Catto beim Prismatic-Joint die PointToLine und FixAngular-Constraint zusammen gefasst hat, will ich hier zeigen, wie er das gemacht hat. So kann man dann vergleichen ob zwei Einzelconstraints besser als eine Multiconstraint ist.

Hier ist Erins Prismatic-Joint wo er den Multi-Ansatz verwendet:

https://github.com/erincatto/box2d/blob/main/src/dynamics/b2_prismatic_joint.cpp

```

28 // Linear constraint (point-to-line)
29 // d = p2 - p1 = x2 + r2 - x1 - r1
30 // C = dot(perp, d)
31 // Cd = dot(d, cross(w1, perp)) + dot(perp, v2 + cross(w2, r2) - v1 - cross(w1, r1))
32 //     = -dot(perp, v1) - dot(cross(d + r1, perp), w1) + dot(perp, v2) + dot(cross(r2, perp), v2)
33 // J = [-perp, -cross(d + r1, perp), perp, cross(r2, perp)]
34 //
35 // Angular constraint
36 // C = a2 - a1 + a_initial
37 // Cd = w2 - w1
38 // J = [0 0 -1 0 0 1]
39 //
40 // K = J * invM * JT
41 //
42 // J = [-a -s1 a s2]
43 //     [0 -1 0 1]
44 // a = perp
45 // s1 = cross(d + r1, a) = cross(p2 - x1, a)
46 // s2 = cross(r2, a) = cross(p2 - x2, a)
47 // Block Solver
48 // We develop a block solver that includes the angular and linear constraints. This makes the limit stiff.
49 //
50 // The Jacobian has 2 rows:
51 // J = [-uT -s1 uT s2] // linear
52 //     [0 -1 0 1] // angular

```

Schritt 1: Aus den beiden Einzel-J-Vektoren wird eine J-Matrix erzeugt

Wir nehmen den J-Vektor aus der PointToLine-Constraint und das J von der FixAngular-Constraint.

$$\text{PointToLine: } \dot{C}: J * V = \begin{bmatrix} -t_1 \\ -(x_2 + r_2 - x_1) \times t_1 \\ t_1 \\ r_2 \times t_1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0$$

$$\text{FixAngular: } \dot{C}: J * V = \begin{bmatrix} 0 \\ -1 \\ 0 \\ 1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0$$

Zur kompakteren Schreibweise definieren wir: $s_1 = -(x_2 + r_2 - x_1) \times t_1$ $s_2 = r_2 \times t_1$

Unser J bekommt nun zwei Zeilen:

$$\text{Schreibweise 1: } v_1/v_2 \text{ als 2er-Vektor } \dot{C}: J * V = \begin{bmatrix} -t_1 & 0 \\ s_1 & -1 \\ t_1 & 0 \\ s_2 & 1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0$$

$$\begin{bmatrix} -t_{1x} & 0 \\ -t_{1y} & 0 \\ s_1 & -1 \\ t_{1x} & 0 \\ t_{1y} & 0 \\ s_2 & 1 \end{bmatrix}^T \begin{bmatrix} v_{1x} \\ v_{1y} \\ \omega_1 \\ v_{2x} \\ v_{2y} \\ \omega_2 \end{bmatrix} = 0$$

Schreibweise 2: Alle Vektorelemente als skalare Zahl ausgeschrieben: $\dot{C}: J * V =$

Schritt 2: K mit den neuen J ausrechnen:

$$\dot{C} : J \cdot V = \begin{bmatrix} -t_{1x} & 0 & v_{1x} \\ -t_{1y} & 0 & v_{1y} \\ s_1 & -1 & 0 \\ t_{1x} & 0 & v_{2x} \\ t_{1y} & 0 & v_{2y} \\ s_2 & 1 & 0 \end{bmatrix} = 0$$

$$J^T = \begin{bmatrix} -t_{1x} & 0 & v_{1x} \\ -t_{1y} & 0 & v_{1y} \\ s_1 & -1 & 0 \\ t_{1x} & 0 & v_{2x} \\ t_{1y} & 0 & v_{2y} \\ s_2 & 1 & 0 \end{bmatrix}$$

$$V = \begin{bmatrix} v_{1x} \\ v_{1y} \\ 0 \\ v_{2x} \\ v_{2y} \\ 0 \end{bmatrix}$$

$$M^{-1} = \begin{bmatrix} m_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & m_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & I_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & m_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & I_2 \end{bmatrix}$$

$$J \cdot M^{-1} = \begin{bmatrix} m_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & m_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & I_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & m_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & I_2 \end{bmatrix}$$

$$K = J \cdot M^{-1} \cdot J^T = \begin{bmatrix} -t_{1x} & 0 & s_1 & -1 & t_{1x} & t_{1y} \\ -t_{1y} & 0 & t_{1x} & 0 & -I_1 & 0 \\ s_1 & -1 & t_{1y} & 0 & 0 & I_2 \\ t_{1x} & 0 & 0 & I_1 & 0 & 0 \\ t_{1y} & 0 & 0 & 0 & I_2 & 0 \\ s_2 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boxed{\begin{array}{l} K1 = m1*t1x^2 + m1*t1y^2 + I1*s1^2 + m2*t1x^2 + m2*t1y^2 + I2*s2^2 \\ m1 * t1^2 + I1*s1^2 + m2*t1^2 + I2*s2^2 \rightarrow t1^2=1 \\ K2 = -s1*I1 + s2*I2 \\ K3 = -s1*I1 + s2*I2 \\ K4 = I1 + I2 \end{array}}$$

Schritt 3: Die MultiConstraint-Klasse erzeugen

Im Konstruktor definiere ich die inverse K-Matrix und die Hebelarme R1/R2. Diese benötige ich für die Anwendung des linearen PointToLine-Impulses. Beim Angular-Impuls brauche ich kein Hebelarm.

```

37 public PointToLineAndFixAngular(ConstraintConstructorData data, PrismaticJoint joint)
38 {
39     this.joint = joint;
40
41     B1 = joint.B1;
42     B2 = joint.B2;
43
44     //R1/R2 wird für die ResolverHelper.ApplyLinearImpulse benötigt. Deswegen setze ich hier die Werte wie bei PointToLine
45     R1 = joint.Anchor2 - joint.B1.Center;
46     R2 = joint.Anchor2 - joint.B2.Center;
47
48     this.t1 = (joint.Anchor1 - B1.Center).Spin90().Normalize();
49     Vec2D d = joint.Anchor2 - B1.Center;
50
51     this.AccumulatedImpulse = data.Settings.DoWarmStart ? new Vec2D(joint.AccumulatedPointToLineImpulse, joint.AccumulatedAngularI
52
53     float pointToLinePositionError = d * this.t1;
54     float angularPositionError = B2.Angle - B1.Angle - joint.AngularDifferenceOnStart;
55
56     this.PositionError = new Vec2D(pointToLinePositionError, angularPositionError);
57
58     float s1 = -Vec2D.ZValueFromCross(d, this.t1);
59     float s2 = Vec2D.ZValueFromCross(R2, this.t1);
60
61     float k2 = -s1*B1.InverseInertia + s2*B2.InverseInertia;
62     float k4 = B1.InverseInertia + B2.InverseInertia;
63     if (k4 == 0) k4 = 1;// For bodies with fixed rotation. //Quelle: https://github.com/erincatto/box2d/blob/main/src/dynamics/b2\_.
64
65     this.InverseK = Matrix2x2.FromScalars(
66         B1.InverseMass + B2.InverseMass + s1 * s1 * B1.InverseInertia + s2 * s2 * B2.InverseInertia,
67         k2,
68         k2,
69         k4
70     ).Invert();
71 }
```

Um ein Impuls berechnen zu können benötige ich CDot. Diesen erhalte ich, indem ich die erste J-Zeile mit V multipliziere was die pointToLineVelocity ergibt und mit der zweiten J-Zeile berechne ich angularVelocity.

```

22     public Vec2D GetCDot() //Gibt CDot=J*V zurück (Geschwindigkeit in Richtung jeder J-Zeile)
23     {
24         float pointToLineVelocity = -t1 * B1.Velocity - Vec2D.ZValueFromCross(B2.Center + R2 - B1.Center, t1) * B1.AngularVelocity
25         float angularVelocity = B2.AngularVelocity - B1.AngularVelocity;
26
27         return new Vec2D(pointToLineVelocity, angularVelocity);
28     }

```

Der Impuls, den ich über CDot und die inverse K-Matrix erhalte ist ein 2D-Vektor. Der erste Wert ist der PointToLine-Impuls und der zweite der FixAngular-Impuls. Wichtig bei der Anwendung des Impulses ist hier, dass R1 und R2 von den Werten her so gesetzt werden, wie sie in der PointToLine-Klasse gesetzt wurden. D.h. Sie orientiert sich an der ersten J-Zeile. Nur so arbeitet ApplyLinearImpulse dann korrekt.

```

84     public void DoSingleSISStep(float invDt)
85     {
86         float biasFactor = 1.0f; //0..1
87         Vec2D bias = biasFactor * invDt * this.PositionError;
88
89         Vec2D impulse = this.InverseK * (-bias - this.GetCDot());
90
91         AccumulatedImpulse += impulse;
92
93         ResolverHelper.ApplyLinearImpulse(this, this.t1 * impulse.X);
94         ResolverHelper.ApplyAngularImpulse(this, impulse.Y);
95
96         SaveImpulse();
97     }

```

Um nun zu testen, ob die Verwendung der Einzelklassen PointToLine und FixAngular (Variante 1) oder PointToLineAndFixAngular (Variante 2) besser ist habe ich die IterationCount des SI-Solvers auf 1 gestellt und AccumulateImpulse auf false gestellt. Nun habe ich etwas am prismatischen Gelenk mit der Maus rumgewackelt. Es war bei beiden Varianten zu sehen, dass das Gelenk zittert. Bei Variante 2 war das Zittern nach kürzerer Zeit weg. D.h. Der Impulswert durch Matrix-Invertierung (Variante 2) ist numerisch genauer als Projected Gaus-Seidel (Variante 1).

Rein von der Umsetzung/Klasse finde ich die Einzellösung aber leichter zu verstehen. Bei der Matrix-Lösung muss man auf einmal wissen, welche J-Zeile für welche Art von Impuls steht und wie man sie anwenden muss. In diesen Fall werden die Hebelarme R1/R2 für die PointToLine-Impulsanwendung verwendet. Wenn ich aber zwei lineare Impulse habe, dann müssen entweder beide Constraints die gleichen Ankerpunkte benutzt haben oder es werden zwei R1/R2-Paare in der MultiConstraint-Klasse benötigt.

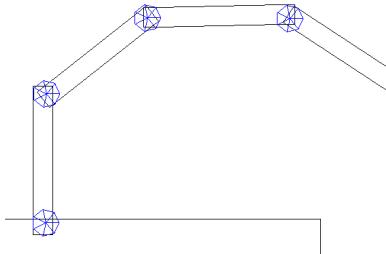
PointToPoint und FixAngular als Multiconstraint

Wenn ich den steifen Weld-Arm simulieren will, dann benötige ich 40 SI-Iterationen, damit er zur Ruhe kommt. Ich will sehen, ob und wie viel SI-Iterationen ich sparen kann, wenn ich PointToPoint und FixAngular als Multiconstraint nutze. Außerdem macht Erin das bei seinen Weld-Joint auch so.

https://github.com/erincatto/box2d/blob/main/src/dynamics/b2_weld_joint.cpp

```
86     // J = [-I -r1_skew I r2_skew]
87     //      [ 0       -1 0       1]
88     // r_skew = [-ry; rx]
```

Ich will sehen, ob sein Aufwand gerechtfertigt ist.



Schritt 1: Aus den beiden Einzel-J-Vektoren wird eine J-Matrix erzeugt

$$\text{PointToPoint: } \dot{C} : J * V = \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ r_{1y} & -r_{1x} \\ 1 & 0 \\ 0 & 1 \\ -r_{2y} & r_{2x} \end{bmatrix}^T \begin{bmatrix} v_{1x} \\ v_{1y} \\ \omega_1 \\ v_{2x} \\ v_{2y} \\ \omega_2 \end{bmatrix} = 0 \quad J = \begin{bmatrix} -1 & 0 & r_{1y} & 1 & 0 & -r_{2y} \\ 0 & -1 & -r_{1x} & 0 & 1 & r_{2x} \end{bmatrix}$$

$$\text{FixAngular: } \dot{C} : J * V = \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ 1 \end{bmatrix}^T \begin{bmatrix} v_{1x} \\ v_{1y} \\ \omega_1 \\ v_{2x} \\ v_{2y} \\ \omega_2 \end{bmatrix} = 0$$

$$\text{PointToPointAndFixAngular: } \dot{C} : J * V = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ r_{1y} & -r_{1x} & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ -r_{2y} & r_{2x} & 1 \end{bmatrix}^T \begin{bmatrix} v_{1x} \\ v_{1y} \\ \omega_1 \\ v_{2x} \\ v_{2y} \\ \omega_2 \end{bmatrix} = 0 \quad J = \begin{bmatrix} -1 & 0 & r_{1y} & 1 & 0 & -r_{2y} \\ 0 & -1 & -r_{1x} & 0 & 1 & r_{2x} \\ 0 & 0 & -1 & 0 & 0 & 1 \end{bmatrix}$$

Schritt 2: K mit den neuen J ausrechnen:

$$J^* M^{-1} = \begin{bmatrix} m_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & m_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & I_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & m_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & I_2 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 & r_{ly} & 1 & 0 & -r_{2y} \\ 0 & -1 & -r_{lx} & 0 & 1 & r_{2x} \\ 0 & 0 & -1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -m_1 & 0 & r_{ly}I_1 & m_2 & 0 & -r_{2y}I_2 \\ 0 & -m_1 & -r_{lx}I_1 & 0 & m_2 & r_{2x}I_2 \\ 0 & 0 & -I_1 & 0 & 0 & I_2 \end{bmatrix}$$

$$K = J^* M^{-1} * J^T = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ r_{ly} & -r_{lx} & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ -r_{2y} & r_{2x} & 1 \end{bmatrix} \begin{bmatrix} K1 & K2 & K3 \\ K4 & K5 & K6 \\ K7 & K8 & K9 \end{bmatrix}$$

$$\begin{aligned} K1 &= m_1 + r_{ly}^2 * I_1 + m_2 + r_{2y}^2 * I_2 \\ K2 &= -r_{lx} * r_{ly} * I_1 - r_{2x} * r_{2y} * I_2 \\ K3 &= -r_{ly} * I_1 - r_{2y} * I_2 \\ K4 &= -r_{lx} * r_{ly} * I_1 - r_{2x} * r_{2y} * I_2 = K2 \\ K5 &= m_1 + r_{lx}^2 * I_1 + m_2 + r_{2x}^2 * I_2 \\ K6 &= r_{lx} * I_1 + r_{2x} * I_2 \\ K7 &= -r_{ly} * I_1 - r_{2y} * I_2 = K3 \\ K8 &= r_{lx} * I_1 + r_{2x} * I_2 = K6 \\ K9 &= I_1 + I_2 \end{aligned}$$

Schritt 3: Umsetzung als Constraint-Klasse

Im Konstruktor lege ich die inverse K-Matrix an:

```

26  public PointToPointAndFixAngular(ConstraintConstructorData data, IPointToPointAndFixAngularJoint joint)
27  {
28      this.joint = joint;
29
30      this.B1 = joint.B1;
31      this.B2 = joint.B2;
32      this.R1 = joint.Anchor1 - joint.B1.Center;
33      this.R2 = joint.Anchor2 - joint.B2.Center;
34
35      var s = data.Settings;
36      this.accumulatedImpulse = s.DoWarmStart ? new Vec3D(joint.AccumulatedPointToPointImpulse.X, joint.AccumulatedPointToPointImpulse.Y,
37
38      Vec3D positionError = new Vec3D(
39          joint.Anchor2.X - joint.Anchor1.X,
40          joint.Anchor2.Y - joint.Anchor1.Y,
41          B2.Angle - B1.Angle - joint.AngularDifferenceOnStart);
42
43      this.bias = s.PositionalCorrectionRate * data.InvDt * positionError;
44
45      float m1 = B1.InverseMass, m2 = B2.InverseMass, I1 = B1.InverseInertia, I2 = B2.InverseInertia;
46
47      float k2 = -R1.X * R1.Y * I1 - R2.X * R2.Y * I2;
48      float k3 = -R1.Y * I1 - R2.Y * I2;
49      float k6 = R1.X * I1 + R2.X * I2;
50
51      this.inverseK = new Matrix3x3(new float[] {
52          m1 + R1.Y * R1.Y * I1 + m2 + R2.Y * R2.Y * I2,
53          k2,
54          k3,
55          k2,
56          m1 + R1.X * R1.X * I1 + m2 + R2.X * R2.X * I2,
57          k6,
58          k3,
59          k6,
60          I1 + I2
61      })
62      .Invert();
63  }

```

Und den Impuls berechne ich über die Standardformel von „3D Constraint Derivations for Impulse Solvers - Marijn 2015“ (2.17).

```

70  public void ApplyWarmStartImpulse()
71  {
72      ResolverHelper.ApplyLinearImpulse(this, this.accumulatedImpulse.XY);
73      ResolverHelper.ApplyAngularImpulse(this, this.accumulatedImpulse.Z);
74  }
75  2 Verweise
76  public void DoSingleSIStep()
77  {
78      Vec3D impulse = this.inverseK * (-this.bias - this.GetCDot());
79
80      this.accumulatedImpulse += impulse;
81
82      ResolverHelper.ApplyLinearImpulse(this, impulse.XY);
83      ResolverHelper.ApplyAngularImpulse(this, impulse.Z);
84
85      this.SaveImpulse();
86  }
87  1 Verweis
88  private Vec3D GetCDot() //Gibt CDot=J*V zurück (Geschwindigkeit in Richtung jeder J-Zeile)
89  {
90      float cDotX = -B1.Velocity.X + R1.Y*B1.AngularVelocity + B2.Velocity.X - R2.Y*B2.AngularVelocity;
91      float cDotY = -B1.Velocity.Y - R1.X*B1.AngularVelocity + B2.Velocity.Y + R2.X*B2.AngularVelocity;
92      float cDotZ = -B1.AngularVelocity + B2.AngularVelocity;
93
94      return new Vec3D(cDotX, cDotY, cDotZ);
95  }

```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/Constraints/Weld/PointToPointAndFixAngular.cs

Fazit für die 3*3-Matrix-Constraint:

Bei 30 SI-Iterationen braucht die Variante, wo ich Point2Point und FixAngular als zwei getrennte Constraints benutze ungefähr 1000 TimeSteps bis der WeldArmStiff sich nicht mehr bewegt. Die 3*3-Matrix-Variante braucht nur 100-200 TimeSteps. D.h. Sie ist effektiver. Nutze ich die Soft-Variante beim WeldArm mit 30 SI-Iterationen, dann sind beide Varianten nicht in der Lage den Arm zur Ruhe zu bekommen aber nach 100 TimeSteps ist die Matrix-Variante ruhiger als die Einzelconstraint-Variante.

Die Joints soft machen

Im Editor wurden alle PropertyViewModel von den Gelenken um die Soft-Property erweitert:

```

internal class WeldJointPropertyViewModel : ReactiveObject
{
    2 Verweise
    [Reactive] public bool CollideConnected { get; set; } = false;

    2 Verweise
    [Reactive] public SoftPropertyViewModel Soft { get; set; } = new SoftPropertyViewModel();
}

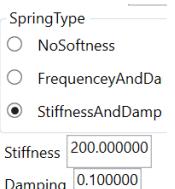
```

Bei den SoftPropertyViewModel kann man entweder die Soft-Option komplett deaktivieren oder man aktiviert es und nutzt entweder die Frequenz und DampingRatio oder Stiffness und Damping.

```

8  internal class SoftPropertyViewModel : ReactiveObject
9  {
10     2 Verweise
11     [Reactive] public SpringParameter SpringParameter { get; set; } = SpringParameter.StiffnessAndDamping;
12     2 Verweise
13     [Reactive] public float FrequencyHertz { get; set; } = 100; //Wie oft in der Sekunde soll die Feder schwingen
14     2 Verweise
15     [Reactive] public float DampingRatio { get; set; } = 0.5f; //0 = Keine Dämpfung (Unendliche Schwingung); 1=Komplette Dämpfung
16     2 Verweise
17     [Reactive] public float Stiffness { get; set; } = 200; //ForceFactor k
18     2 Verweise
19     [Reactive] public float Damping { get; set; } = 0.1f; //Damping coefficient c

```



Jede Joint-Klasse (IJoint) hat ein SoftConstraintData-Objekt was diesen Parameter intern speichert.

```

9  internal class SoftConstraintData
10 {
11     10 Verweise
12     internal SpringParameter ParameterType { get; set; } = SpringParameter.FrequencyAndDampingRatio;
13     4 Verweise
14     internal float FrequencyHertz { get; set; } = 0; //Wie oft in der Sekunde soll die Feder schwingen
15     4 Verweise
16     internal float DampingRatio { get; set; } = 0; //0 = Keine Dämpfung (Unendliche Schwingung); 1=Komplette Dämpfung
17     8 Verweise
18     internal float Stiffness { get; set; } = 0; //ForceFactor k: 0=Keine Feder   -> Nur diese Parameter werden intern zur Berechnung verwendet
19     7 Verweise
20     internal float Damping { get; set; } = 0; //Damping coefficient c           -> Nur diese Parameter werden intern zur Berechnung verwendet

```

Diese Klasse hat eine Methode, um Gamma und Beta laut der Formel von „Soft Constraints - Erin Catto 2011“ Seite 34 auszurechnen.

```

54 //k=J*M^-1*J^T
55 // 6 Verweise
56 public void GetSoftConstraintParameters(float dt, float k, Action<float> gammaOut, Action<float> betaOut, Action<float> ImpulseMassOut)
57 {
58     float h = dt;
59
60     //Quelle1: https://github.com/erincatto/box2d/blob/main/src/dynamics/b2\_distance\_joint.cpp#L138C27-L138C27
61     //Quelle2: 3D Constraint Derivations for Impulse Solvers - Marijn 2015" Seite 9 Formel (2.17)
62     float gamma = 1.0f / (h * (Damping + h * Stiffness)); //Gamma entspricht Gamma/dt aus (2.17) "3D Constraint Derivations for Impulse Solve
63     float beta = h * Stiffness * gamma; //Beta entspricht Beta/dt-Wert aus (2.17)
64     float ImpulseMass = 1.0f / (k + gamma); //ImpulseMass entspricht K aus der Formel (2.12)
65
66     //Da Propertys nicht als Out-Parameter verwendet werden dürfen und das IConstraint-Interface aber sagt, dass das Propertys sein müssen nu
67     gammaOut(gamma);
68     betaOut(beta);
69     ImpulseMassOut(ImpulseMass);
70 }

```

Siehe: PhysicEngine/Joints/SoftConstraintData.cs

Alle Constraints, welche Soft gemacht werden sollen, rufen diese Methode auf, um damit Beta und Gamma zu berechnen:

```

65 if (joint.Soft.ParameterType != SpringParameter.NoSoftness)
66 {
67     IsSoftConstraint = true;
68     joint.Soft.GetSoftConstraintParameters(data.Dt, invMass, x => Gamma = x, x => Beta = x, x => ImpulseMass = x);
69 }

```

Mit diesen beiden Parametern wird dann laut der Formel (2.17) von Marijn die Constraint soft gemacht. Bei Constraints, die ein linearen Impuls in ForceDirection ausüben (Normalconstraint, Distanz,...), sieht die Soft-Impulsformel so aus:

```

105 //Quelle1: https://github.com/erincatto/box2d/blob/main/src/dynamics/b2\_distance\_joint.cpp#L191
106 //Quelle2: 3D Constraint Derivations for Impulse Solvers - Marijn 2015" Seite 9 Formel (2.17)
107 float impulse = -c.ImpulseMass * (velocityInForceDirection + c.Beta * c.PositionError + c.Gamma * c.AccumulatedImpulse);

```

Bei 2D/3D-XY-Constraints (Maus, Point2Point) muss das Gamma mit der Einheitsmatrix multipliziert auf die K-Matrix addiert werden (Siehe (2.12 Marijn)

```

51
52     this.InverseK = Matrix2x2.FromScalars(
53         b.InverseMass + r.Y * r.Y * b.InverseInertia + this.Gamma,
54         -r.X * r.Y * b.InverseInertia,
55         -r.X * r.Y * b.InverseInertia,
56         b.InverseMass + r.X * r.X * b.InverseInertia + this.Gamma
57     )
58     .Invert();

```

Der Soft-Impuls ist nun ein 2D/3D-Vektor:

```

148     Vec2D impulse = c.InverseK * (-c.GetCDot() - c.Gamma * c.AccumulatedImpulse - c.Beta * c.PositionError);
149
150     Vec3D impulse = inverseK * (-GetCDot() - Gamma * accumulatedImpulse - Beta * this.PositionError);

```

Beim Angular-Impuls sieht die Formel so wie beim 1D-Linearimpuls aus, nur das anstelle von Velocity nun AngularVelocity steht:

```

196     float angularVelocity = c.B2.AngularVelocity - c.B1.AngularVelocity;
197
198     float angularImpulse = -c.ImpulseMass * (angularVelocity + c.Beta * c.PositionError + c.Gamma * c.AccumulatedImpulse);

```

Ein Constraint soft zu machen heißt also Gamma und Beta ausrechnen und damit dann eine andere Impulsformel nehmen. Der Datentyp des Impulses hängt davon ab, ob die PositionConstraint eine Distanz (1D-Linear), ein XY-Abstand (2D-Linear) oder ein Winkelabstand (AngularConstraint) misst. Interessanter Weise wird ein Gelenk durch Softconstraints sogar härter, als wenn ich kein Soft verwende, wenn der Stiffness-Wert bei 200 und höher liegt. Erin Catto hatte glaube mal erwähnt, dass Softconstraints ähnlich wie eine Feder die Federenergie speichern. Durch diese Energie scheint das Gelenk dann härter zu werden.

Der J-Vektor und die Forcedirection

Bis jetzt war es so, dass alle Constraint, dessen J-Vektor nur aus einer Zeile besteht, waren vom

Aufbau her so: $\dot{C}: J \cdot V = \begin{bmatrix} -F \\ -r_1 \times F \\ F \\ r_2 \times F \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0 \rightarrow F=\text{ForceDirection}; r1/r2=\text{Hebelarme}$

$$\text{oder so: } \dot{C} : J * V = \begin{bmatrix} 0 \\ -1 \\ 0 \\ 1 \end{bmatrix}^T \begin{bmatrix} v_1 \\ \omega_1 \\ v_2 \\ \omega_2 \end{bmatrix} = 0$$

Ich denke das ist kein Zufall da eine Kraft entweder zwischen zwei Ankerpunkten besteht, was ein Linearimpuls erzeugt oder beide Ankerpunkte liegen an der gleichen Stelle und von dort aus wirkt ein Drehimpuls. Wäre das J-Muster nicht so, würde das bedeuten dass auf ein Körper eine Kraft wie aus dem Nichts wirkt. Eine Kraft kann aber immer nur von ein anderen Körper aus wirken so dass sie sich dann gegenseitig beeinflussen. Deswegen ist dieses Muster auch eine gute Kontrollmöglichkeit für die J*V-Herleitung.

Zusammensetzung der Joints

Die Hälfte der Constraints wird von mehr als ein Joint genutzt.

	Distance	Prismatic	Wheel	Weld	Revolute
FixAngular (Soft)		X		X	
MinMaxTranslation			X		
PointToLine		X	X		
PointToPoint				X	X
TranslationMotor (Soft)		X	X		
DistanceJointConstraint (Soft)	X				
MinMaxDistance	X				
PointToLineAndFixAngular		X			
AngularMotor (Soft)					X
MinMaxAngular					X

So legt z.B. die PrismaticJoint-Klasse fest, welche Constraints sie nutzt:

```
94  public List<IConstraint> BuildConstraints(ConstraintConstructorData data)
95  {
96      List<IConstraint> list = new List<IConstraint>();
97      //Möglichkeit 1: PointToLine und Angular-Constraint als getrennte Klassen
98      //list.Add(new PointToLine(data, this));
99      //list.Add(new FixAngular(data, this));
100
101
102      //Möglichkeit 2: PointToLine kommt in die erste J-Zeile und Angular in die zweite J-Zeile. Über
103      //die inverse K-Matrix bekomme ich zwei Impulswerte: Linear-Impuls in t1-Richtung und AngularImpuls
104      list.Add(new PointToLineAndFixAngular(data, this));
105
106      if (this.LimitIsEnabled)
107          list.Add(new MinMaxTranslation(data, this));
108      if (this.Motor == IPublicPrismaticJoint.TranslationMotor.Disabled)
109          this.AccumulatedTranslationMotorImpulse = 0;
110      else
111          list.Add(new TranslationMotor(data, this));
112      return list;
113 }
```

Jedes Joint-Objekt hat die BuildConstraints-Methode, wo sie dann die benötigten Constraints als Liste zurück gibt.

Zusammenfassung von Teil 5

Alle Joints sind so, dass immer jeweils genau zwei Körper miteinander eine Kraft an den Ankerpunkten wirken lassen. Jeder Körper hat genau ein Ankerpunkt. Über die Position- und Velocityconstraint erfolgt die Herleitung des J-Vektors und der K-Matrix/Skalar. Der Impuls zeigt dann von Ankerpunkt1 zu Ankerpunkt2 dessen Länge über J*V/K ausgerechnet werden kann.

Es gibt nur zwei Möglichkeiten, wie die beiden Körper untereinander Kraft ausüben:

Variante 1: Ihre Ankerpunkte ziehen sich an/stoßen sich ab.

Variante 2: Beide Ankerpunkte liegen an der gleichen Stelle und es wirkt eine Drehkraft.

Folgende 6 Schritte sind nötig, wenn ich die Bewegung von Starrkörper aufgrund von selbst aufgestellten Regeln (Abstoßregel, Distanzregel, FixAngular-Regel) beeinflussen will:

1: Ankerpunkte definieren

2: PositionConstraint aufstellen

3: VelocityConstraint durch Ableiten nach der Zeit ermitteln und dann nach J^*V umstellen

4: Die Inverse von K ermitteln

5: Impulslänge Lambda ermitteln

6: Impuls in J-Richtung bei beiden Ankerpunkten wirken lassen



Schritt 2: Ich möchte dass die Körper sich abstoßen oder ein Gelenk Kräfte auf die Körper ausübt. Dazu definiere ich die PositionConstraint welche nur auf die Variablen $x_1, r_1, w_1, x_2, r_2, w_2$ als Input benötigt. 2

Beispiel für lineare PositionConstraint: $C: t_1 \cdot (x_2 + r_2 - x_1) = 0 \quad d = x_2 + r_2 - x_1 \quad t_1 = \text{Normalize}(r_1).Spin90()$

Beispiel für angulare PositionConstraint: $C: \phi_2 - \phi_1 = A$

Schritt 3: Ich leite die PositionConstraint nach der Zeit ab und erhalte die VelocityConstraint, welche ich nach J^*V umstellen kann.

$$\dot{C}: J^* V = \begin{bmatrix} -t_1 \\ -(x_2 + r_2 - x_1) \times t_1 \\ t_1 \\ r_2 \times t_1 \end{bmatrix} = 0$$

3

$$\dot{C}: J^* V = \begin{bmatrix} 0 \\ -1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_0 \\ v_2 \\ v_0 \end{bmatrix} = 0$$

Beispiel für lineare J^*V -VelocityConstraint

Beispiel für angulare J^*V -VelocityConstraint

Schritt 4: Ich berechne K über $K = JM^{-1}J^T$ und die Inverse davon.

4

$$K = J \cdot M^{-1} \cdot J^T = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ r_{1y} & -r_{1x} & 0 \\ 0 & 1 & 0 \\ -r_{2y} & r_{2x} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$K = J \cdot M^{-1} \cdot J^T = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ r_{1y} & -r_{1x} & 0 \\ 0 & 1 & 0 \\ -r_{2y} & r_{2x} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Wenn der Rückgabetyp der PositionConstraint ein Skalar ist, dann enthält J ein Spaltenvektor und K ist ein Skalar.

Gibt PositionConstraint ein 2D-Vektor zurück, dann enthält J 2 Spaltenvektoren und K ist eine 2×2 -Matrix.

Gibt es ein 3D-Vektor zurück, enthält J 3 Spaltenvektoren und K ist eine 3×3 -Matrix.

Schritt 5: Ich berechne über J^*V den CDot-Wert (Skalar/2D/3D-Vektor) und berechne damit die Impulslänge (Lambda)

J enthält eine Zeile \rightarrow CDot, Bias und Lambda sind skalare Zahlen; K=Skalar
J enthält zwei Zeilen \rightarrow CDot, Bias und Lambda sind 2D-Vektoren K=2x2 Matrix
J enthält drei Zeilen \rightarrow CDot, Bias und Lambda sind 3D-Vektoren K=3x3 Matrix

5

$$\lambda = \frac{-\mathbf{J} \cdot \mathbf{v} - \zeta}{K}$$

Ohne Soft Mit Soft

Schritt 6: Über $J^T \lambda$ erhalte ich den Impulsvektor womit ich die Geschwindigkeit der beiden Körper korrigieren kann.

$$J^T = \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ r_{1y} & -r_{1x} \\ 1 & 0 \\ 0 & 1 \\ -r_{2y} & r_{2x} \end{bmatrix} \begin{bmatrix} \text{ImpulseX1} \\ \text{ImpulseY1} \\ \text{Drehimpuls1} \\ \text{ImpulseX2} \\ \text{ImpulseY2} \\ \text{Drehimpuls2} \end{bmatrix}$$

6

Ausgangspunkt, wie wir überhaupt darauf gekommen sind, Starrkörperbewegungen per Constraints zu steuern sind folgende Formeln:

A: Newton hat sich die $F=m \cdot a$ -Formel überlegt was eine Differentialgleichung wegen $a=\dot{x}$ ist

B: Um sie numerisch zu lösen nutzen wir Semi-Implizit-Euler so dass pro Zeitschritt h der Körper nun wie folgt bewegt wird:

$$1: \quad v_2 = v_1 + h * \frac{F}{m}$$

$$2: \quad x_2 = x_1 + h * v_2$$

C: Diese Formel lässt den Körper nur in XY-Richtung geradlinig laufen. Um nun aber zu verhindern, dass er durch Wände fliegt oder um ein Türscharnier zu simulieren, so brauche ich noch eine 3. Gleichung:

$$3: \quad J^* V_2 = Bias$$

Das J habe ich dadurch erhalten, indem ich über die PositionConstraint-Gleichung gesagt habe, welche Bedingung für 2 Ankerpunkte erfüllt sein muss. Diese $J^*V_2 = Bias$ -Formel ist nur eine Umstellung von dieser Ankerpunkt-Bedingung. Wenn ich diese 3 Gleichungen nun ineinander einsetze, dann ergeben sich daraus dann alle Regeln zur beschränkten Starrkörperbewegung.

Zusammenfassung der Zusammenfassung

Beschränkte Bewegung = Newton-Formel+SemiImplizit-Euler+Position/VelocityConstraint

Teil 6 : Polygone

Ziel dieses Abschnitts

In diesen Abschnitt soll die Physikengine soweit vorbereitet werden, dass mit der Erstellung des Leveleditors (kommt in Teil 7) begonnen werden kann. Es fehlen aktuell noch Polygone (z.B. für den Levelrand), Schubdüsen (für Raumschiffe), Kraftvisualisierung von Stabkräften (für BridgeBuilder) und eine Kollisionsmatrix (um eine Person zu modellieren, welche ihre Arme neben ihren Körper entlang führt).

Physic-Editor

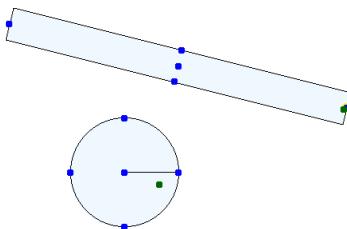
Snap Anchor-Points

Damit man beim Bau eines Greifarms oder Motorrads die Gelenke genau auf den Hauptachsen platzieren kann, soll die PlaceTwoAnchorPointsFunction-Klasse so umgeändert werden, dass bei Drücken der Shift-Taste der Ankerpunkt nun nicht mehr nur im Body-Center platziert wird, sondern dass man noch weitere Punkte zur Auswahl hat.

IEditorShape wird dazu um folgende Methode erweitert:

```
10  interface IEditorShape
11 {
12     Vec2D[] GetAnchorPoints();
```

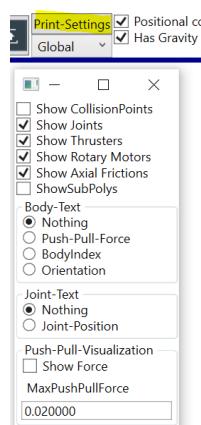
Wenn ich nun beim Ankerpunkt-Platzieren die Shift-Taste halte, dann bekomme ich die blauen Snap-Points angezeigt an denen die Maus dann kleben bleibt, wenn sie drüber fährt:



Physik-Simulator

Print-Settings

Für die Testausgabe wird der Simulator um ein Print-Settings-Dialog erweitert:



Somit können nun Stabkräfte, Thruster oder Unterpolygone (entstehen, wenn ein konkaves Polygon in konvexe Polygone zerlegt wird) angezeigt werden.

Physic-Engine

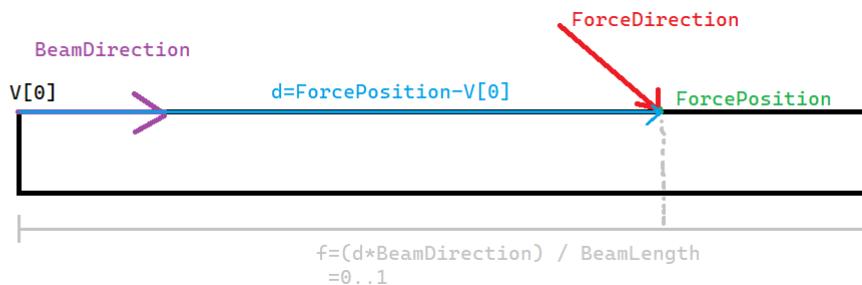
Force Visualisation – Zug und Druckkräfte

Bei BridgeBuilder sollen die Stäbe von der Brücke rot angezeigt werden, wenn sie unter starken Druckkräften stehen und blau, wenn sie auseinander gezogen werden. Grün bedeutet geringe Kraft.

Ein Rechteck gilt dann als Stab, wenn es 5 mal so lang wie breit ist. Immer dann, wenn eine Kraft auf ein Körper wirkt, dann wird berechnet, zu wie viel Prozent die Kraft auf das linke Stabende, und zu wie viel Prozent auf das rechte Stabende geht. Die Differenz zwischen den Stabend-Punkten ergibt dann die Zug-Druck-Kraft.

Eine Kraft wirkt immer dadurch auf ein Körper, dass sie ein Angriffspunkt (Ankerpunkt) hat und eine Richtung.

Zuerst berechne ich f , welche angibt, an welcher Stelle des Stabes die Kraft wirkt. Wenn f Null ist, dann liegt die ForcePosition am linken Stabende. Wenn f Eins ist, dann liegt es ganz rechts:



Die RigidRectangle-Klasse wird um die AddTrackForce-Methode erweitert. Diese speichert die Kräfte, welche aufs Stabende wirken. Zuerst wird mit f bestimmt, an welcher Stelle die Kraft ansetzt. Da wir nur die Zug- und Druck-Kräfte in Stabrichtung messen wollen, projiziere ich die Kraft bei Zeile 183 noch in Stabrichtung so dass ich nur noch skalare Kraftwerte für die Stabenden mir merken muss.

```
165 #region IBeamForceTracker
166 private bool rectangleIsBeam = false; //Nur wenn das Rechteck 5 mal so lang wie hoch ist gilt es als Stab
167 private Vec2D beamDirectionLocal; //Wenn das Rechteck wie ein Stab aussieht, dann zeigt dieser Vektor in Stabrichtung
168 private Vec2D beamDirection;
169 private float inverseBeamLength;
170 private float forceOnLeftBeam = 0;
171 private float forceOnRightBeam = 0;
172 2 Verweise
173 public void ResetTrackForce()
174 {
175     this.forceOnLeftBeam = 0;
176     this.forceOnRightBeam = 0;
177 }
178 3 Verweise
179 public void AddTrackForce(Vec2D forcePosition, Vec2D forceDirection)
180 {
181     if (this.InverseMass == 0 || this.rectangleIsBeam == false) return; //Trage die Kräfte, wenn das Rechteck eine Stabform hat
182     Vec2D d = forcePosition - this.Vertex[0];
183     float f = (d * this.beamDirection) * this.inverseBeamLength; //f=0 -> Kraft wirkt an linker Balkenecke; 1=Kraft wirkt an rechter Balkenecke
184     float forceInBeamDirection = forceDirection * this.beamDirection;
185
186     this.forceOnLeftBeam += (1 - f) * forceInBeamDirection;
187     this.forceOnRightBeam += f * forceInBeamDirection;
188 }
189 4 Verweise
190 public float GetPushPullForce()
191 {
192     return this.forceOnLeftBeam - this.forceOnRightBeam;
193 }
#endregion
```

Siehe: PhysicEngine/RigidBody/RigidRectangle.cs

Der ResolverHelper wird nun an zwei Stellen erweitert. Die Gravitationskraft lasse ich aufs Zentrum des Körpers wirken. D.h. die Hälfte dieser Kraft geht jeweils auf die Stabenden.

```

26 //Schritt 2: Wende die externe Kraft für alle Körper an
27 foreach (var body in bodies)
28 {
29     body.Velocity.X += body.InverseMass * body.Force.X * dt;
30     body.Velocity.Y += body.InverseMass * body.Force.Y * dt;
31     body.AngularVelocity += body.InverseInertia * body.Torque * dt;
32
33 //Wende Schwerkraft zu 50% aufs linke Stabende und zu 50% aufs rechte Stabende an
34 if (body is IBeamForceTracker)
35 {
36     var beam = (IBeamForceTracker)body;
37     beam.ResetTrackForce();
38     beam.AddTrackForce(body.Center, body.Force);
39 }
40 }

```

Siehe: PhysicEngine/CollisionResolution/SequentiellImpulse/ResolverHelper.cs

Die Kräfte von den Constraints (Normalkraft, Gelenke, Maus) entnehme ich der AccumulatedImpulse-Variable. Ich rechne den Impuls durch dt-Division in eine Kraft um.

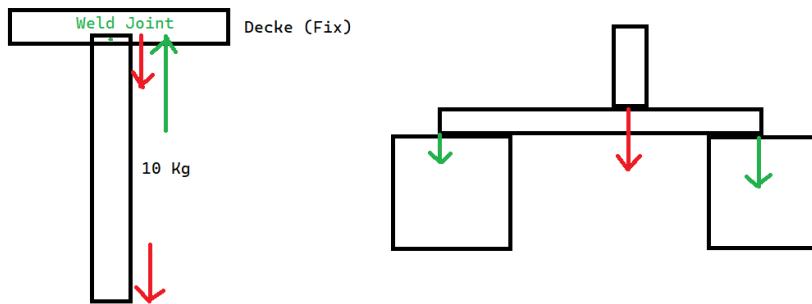
```

61 //Schritt 5: Berechne, welche Zug- und Druck-Kräfte auf die Stäbe (längliche Rechtecke) gewirkt haben
62 // Wird nur für die Anzeige genutzt. Hat keine Auswirkung auf die Bewegung der Körper!
63 foreach (var c in constraints)
64 {
65     if (c is ILinear1DConstraint)
66     {
67         var cLin = (ILinear1DConstraint)c;
68         Vec2D impulse = cLin.ForceDirection * cLin.AccumulatedImpulse;
69         TrackBeamForce(c.B1, invDt, c.R1, -impulse);
70         TrackBeamForce(c.B2, invDt, c.R2, impulse);
71     }
72     else if (c is ILinearImpulse)
73     {
74         var cLin = (ILinearImpulse)c;
75         Vec2D impulse = cLin.GetAppliedLinearImpulse();
76         TrackBeamForce(c.B1, invDt, c.R1, -impulse);
77         TrackBeamForce(c.B2, invDt, c.R2, impulse);
78     }
}

```

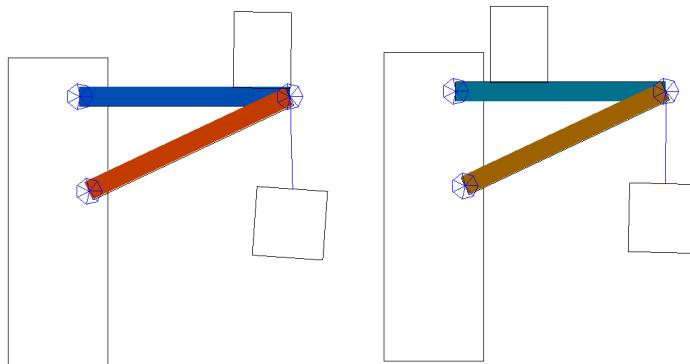
Beispiel Links: Ein 10Kg-Stab hängt an der Decke. Es wirken jeweils 5Kg Gravitationskraft auf die Stabenden und das Weld-Joint zieht den Stab mit 10Kg nach oben. Somit entsteht am oberen Stabende eine Zugkraft von 5Kg und am unteren Ende ziehen 5Kg nach unten. Die Differenz zwischen -5Kg und +5Kg ergibt -10Kg. D.h. Auf den Stab wirkt eine Zugkraft von 10Kg.

Beispiel Rechts: Auf den Stab wirken nur Kräfte die Senkrecht zur Stabrichtung sind. Somit erfolgt auf den Stab überhaupt keine Zug-Druck-Kraft. Die inneren Kräfte, die entstehen, weil in der Mitte was drauf steht werden nicht betrachtet.



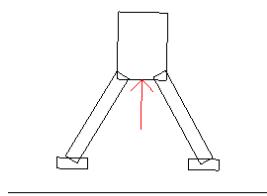
D.h. Diese Art der Kraftvisualisierung eignet sich für Fachwerke, wo die Stäbe nur an den Enden mit Gelenken verbunden sind.

So sieht das ganze dann aus: Links steht der Würfel oben weiter am Stabende. Somit wird der obere Stab mehr auf Zug (Farbe ist mehr Blau) und der untere Stab mehr auf Druck (mehr Rot) belastet. Beim linken Beispiel steht der Würfel weiter links so dass die Kräfte auf die Stäbe weniger stark sind. Deswegen gehen ihre Farben mehr Richtung grün.



Schubdüsen

Für das Moonlander-Spiel werden Schubdüsen (Thruster) benötigt. Eine Schubdüse hat einen Ankerpunkt bei einem Körper, wo sie dran befestigt ist und sie hat eine Richtung, in welche die Düse zeigt. Im Simulator wird die Düse als Pfeil angezeigt. Sie drückt das Raumschiff in diesen Beispiel nach oben:



Die Kraft der Schubdüse wird ähnlich wie die Schwerkraft behandelt nur das diese Kraft nicht beim Zentrum ansetzt sondern am Ankerpunkt der Düse. Beim ResolverHelper wird die ApplyExternalForces-Methode noch um den Schubdüsenblock (blau markiert) erweitert:

```
private static void ApplyExternalForces(List<IRigidBody> bodies, List<IThruster> thrusters, float dt)
{
    //Gravity-Force
    foreach (var body in bodies)
    {
        body.Velocity.X += body.InverseMass * body.Force.X * dt;
        body.Velocity.Y += body.InverseMass * body.Force.Y * dt;
        body.AngularVelocity += body.InverseInertia * body.Torque * dt;
    }

    //Wende Schwerkraft zu 50% aufs linke Stabende und zu 50% aufs rechte Stabende an
    if (body is IBeamForceTracker)
    {
        var beam = (IBeamForceTracker)body;
        beam.ResetTrackForce();
        beam.AddTrackForce(body.Center, body.Force);
    }
}

//Thruster-Force
foreach (var thruster in thrusters)
{
    var body = thruster.B1;

    if (thruster.IsEnabled == false || body.InverseMass == 0)
        continue;

    body.Velocity.X += body.InverseMass * thruster.Force.X * dt;
    body.Velocity.Y += body.InverseMass * thruster.Force.Y * dt;
    body.AngularVelocity += Vec2D.ZValueFromCross(thruster.R1, thruster.Force) * body.InverseInertia;

    if (body is IBeamForceTracker)
    {
        var beam = (IBeamForceTracker)body;
        beam.AddTrackForce(thruster.Anchor, thruster.Force);
    }
}
```

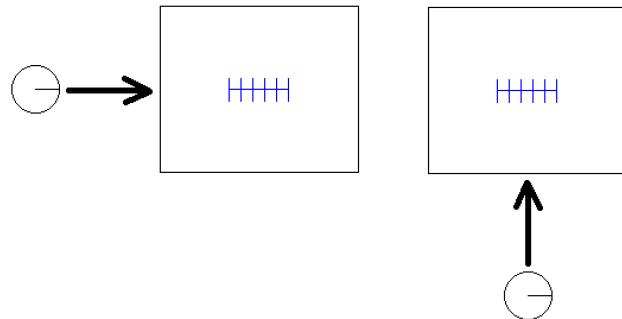
Die `PhysicsScene` speichert die Thruster-Objekte ähnlich wie die Joints in einer Liste, welche über `PublicInterface` nach außen steuerbar sind:

```
18 Verweise
public enum SolverType { Global, Grouped }

private List<IRigidBody> bodies = new List<IRigidBody>();
private List<IJoint> joints = new List<IJoint>();
private List<IThruster> thrusters = new List<IThruster>();
private MouseConstraintData mouseData = null; //Wenn der Nutzer einen Körper angeklickt hat, welchen er festhalten will
```

Axial Friction

Wenn man ein 2D-Autospiel erzeugen will, wo man von oben auf das Auto schaut, dann müssen die Räder in Fahrtrichtung sich bewegen können aber zur Seite dürfen sie nicht rutschen. Für diese Aufgabe erstelle ich eine AxialFriction-Constraint, welche Reibung für einen Körper nur in eine angegebene Achsenrichtung wirken lässt. Beispiel: Zu sehen ist ein Rad von oben. Wenn nun ein Ball von links gegen das Rad in die Seite stößt (linkes Bild), dann bremst die AxialFriction, welche hier nur horizontal wirkt den Stoß aus. Stößt der Ball in Fahrtrichtung gegen das Rad (rechtes Bild), dann macht die AxialFriction nichts und das Rad rollt nach oben davon.



Die AxialFriction wird ähnlich wie eine Schubdüse dadurch definiert, dass man ein Punkt hat, an dem die Reibungskraft wirkt und es wird eine Richtung definiert, in welche die Reibung arbeitet. Die Constraint-Klasse orientiert sich dabei an der FrictionConstraint-Klasse vom Kontaktpunkt. Der Unterschied ist, dass Body2 hier der Boden ist, welche im Hintergrund ist, und welcher eine unendliche Masse hat. Die inverse Masse von Body2 und auch die Hebelarmlänge ist hier Null. Somit sieht die Constraint dann so aus:

```
25 public AxialFrictionConstraint(ConstraintConstructorData data, IAxialFriction axialFriction)
26 {
27     this.axialFriction = axialFriction;
28     B1 = axialFriction.B1;
29     B2 = null;
30
31     R1 = axialFriction.R1;
32     R2 = null;
33     float r1crossT = Vec2D.ZValueFromCross(R1, axialFriction.ForceDirection);
34
35     ImpulseMass = 1.0f / (B1.InverseMass +
36         r1crossT * r1crossT * B1.InverseInertia);
37
38     ForceDirection = axialFriction.ForceDirection;
39     Bias = 0;
```

Man kann diese AxialFriction aber auch benutzen, um ein Pendel zu dämpfen. Links ungedämpft und rechts wurde es durch die AxialFriction gedämpft:



Friction-Constraint

Ich ersetze in der FrictionConstraint diese Zeile:

```
MaxImpulse = data.Settings.Gravity * friction * data.Dt;
```

durch diese Zeile hier:

```
MaxImpulse = c.NormalImpulse * friction;
```

und orientiere mich dabei an der Box2D-lite-Formel:

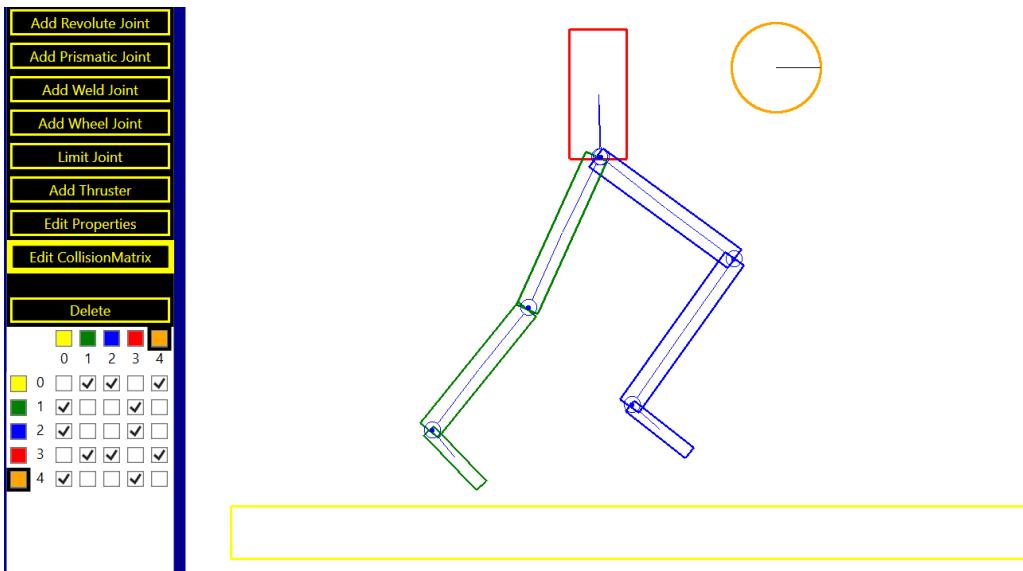
```
174 // Compute friction impulse
175 float maxPt = friction * c->Pn;
```

<https://github.com/erincatto/box2d-lite/blob/master/src/Arbiter.cpp>

da das realistischer ist, dass z.B. bei ein Würfelstapel der untere Würfel mehr Reibung mit den Boden hat, weil er mehr wegen der anderen Würfel in den Boden gedrückt wird. Für diese Änderung war die NormalImpulse-Variable nötig, die es erst seit Teil 3 mit dem WarmStart gibt.

CollisionMatrix

Wenn ich eine Figur mit zwei Beinen bauen will, dann muss ich verhindern, dass die beiden Beine miteinander kollidieren da sonst keine Schrittbewegung möglich ist. Außerdem müssen aber beide Beine den Boden berühren können. Ich löse diese Aufgabe indem ich jeden Körper eine CollisionCategory als Attribut mitgebe, was eine Zahl von 0..4 ist. Über die Matrix definiere ich dann, welche Kategorie mit welcher anderen kollidiert. Die Kategorie wird als Farbe dargestellt. In diesen Beispiel kollidieren die beiden Beine nicht miteinander aber sie können den Boden berühren. Außerdem darf der orangene Ball zwar mit dem roten Torso kollidieren aber nicht mit den Beinen.



Ich habe dazu den CollisionPairManager, welcher in der PhysicScene genutzt wird, um die CollisionMatrix erweitert:

```
7 internal class CollisionPairManager<T> where T : class, ICollidable, IBoundingCircle
8 {
9     private List<CollidablePair<T>> pairs;
10    private List<T> collidables;
11    public bool[,] CollisionMatrix { get; private set; }
12 }
```

Der CollisionPairManager verhindert dann, das Objekte miteinander kollidieren wo die Matrix auf false steht:

```
54 //Das ist der BroadPhase-Filter. All diese ICollidable-Eigenschaften ändern sich beim Objekt über seine Laufzeit nicht.
55 1 Verweis
56 private static bool ShouldCollide(ICollidable b1, ICollidable b2, bool[,] collisionMatrix)
57 {
58     if (b1.IsNotMoveable && b2.IsNotMoveable) return false;
59     if (collisionMatrix[b1.CollisionCategory, b2.CollisionCategory] == false) return false;
60     if (b1.CollideExcludeList.Contains(b2)) return false;
61 
62     return true;
}
```

Der Editor speichert all seine erzeugten Physik-Daten im FunctionData-Objekt. Dort wird dann auch festgelegt, wie groß die Matrix ist.

```
internal class FunctionData
{
    private static int MatrixSize = 5;

    public List<IEditorShape> Shapes = new List<IEditorShape>();
    public List<IEditorJoint> Joints = new List<IEditorJoint>();
    public List<IEditorThruster> Thrusters = new List<IEditorThruster>();
    public bool[,] CollisionMatrix = new bool[MatrixSize, MatrixSize];

    2 Verweise
    public FunctionData()
    {
        this.CollisionMatrix[0, 0] = true; //Objekte mit Kategorie 0 sollen Defaultmäßig miteinander kollidieren
    }
}
```

Polygon

Wenn ich bei Elma mit Polygonen das Level bauen will, dann muss ich mich innerhalb eines Polygons mit dem Motorrad aufhalten dürfen. Das Polygon ist aus Kollisionssicht innen hohl. Will ich aber eine Figur aus ein Polygon bauen, dann muss das Polygon innen gefüllt sein. Wenn das Polygon eine konvexe Form hat, dann kann ich SAT für die Kollisionsberechnung nehmen. Sollte es konkav sein, dann gibt es entweder ein Paper aus dem Jahre 2004 „Efficient Collision Detection between 2D Polygons“ oder ich zerlege das Polygon in konvexe Teil-Polygone. Dazu gibt es hier: <https://stackoverflow.com/questions/20529899/algorithms-for-collision-detection-between-concave-polygons>

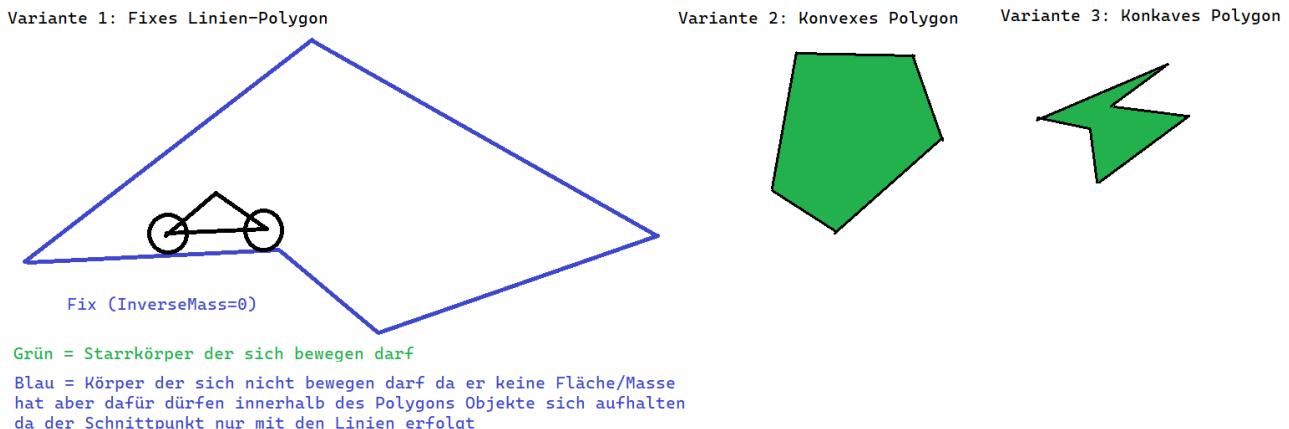
eine Auflistung von Algorithmen.

Es gibt folgende 3 Polygon-Arten aus Sicht der Kollisionsberechnung:

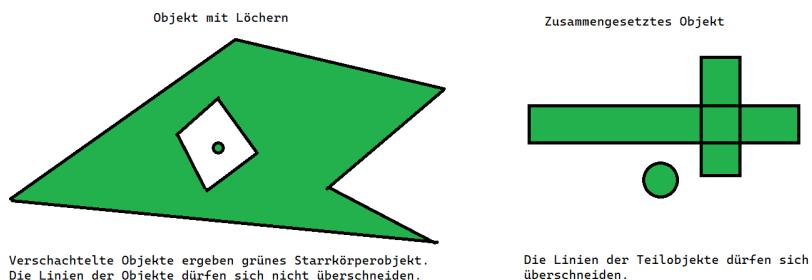
1. Polygonlinienzug = Innen hohl. Wird für den Levelrand benötigt. Bei Box2D heißt diese Polygonart Edge/Chain:
https://github.com/erincatto/box2d/blob/main/src/collision/b2_collide_edge.cpp
2. Konvexes Polygon = Kollision erfolgt per SAT. Beispiel:
https://github.com/erincatto/box2d/blob/main/src/collision/b2_collide_polygon.cpp
3. Konkaves Polygon = Muss entweder in konvexe Polygone zerlegt werden oder man findet einen direkten Weg zur Kollisionsberechnung

Will ich eine Starrkörpersimulation für ein Polygon durchführen, dann ist es wichtig, dass das Objekt eine definierte Fläche, Masse und Schwerpunkt hat. Wenn ich das Linien-Polygon (Variante 1) als fixes Objekt habe, dann muss ich mir über dessen Masse/Schwerpunkt keine Gedanken machen.

So sehen die 3 Polygonarten aus, die für die Starrkörpersimulation verwendet werden können:



Beim Elma-Spiel kann man Polygone auch (mehrfach) ineinander verschachteln so dass man Löcher heraus schneiden kann. Diese Art die Objektdefinition ist aber nicht zwangsläufig nur auf Polygone begrenzt. Man könnte auf zwei Arten aus Grundobjekten (Kreis, Rechteck, Polygon) ein Merged-Objekt erzeugen:



Ich erwähne das Merged-Objekt hier nur um es von den Polygonen abgrenzen. Das ist ein eigenes Thema für sich. Hier soll es nur um die 3 Polygonarten gehen: Linienpolygon, konvex, konkav.

Folgende Funktionen benötige ich für diese 3 Polygonarten:

- PointIsInside-Check → Wird für den Editor benötigt

- Masseschwerpunkt, Masse und Inertia → Zur Simulation

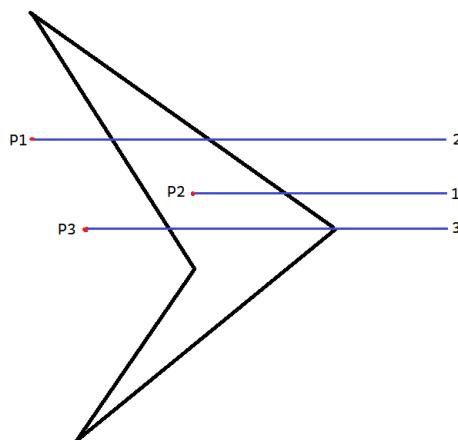
- Kollisionspunkt mit Normale, wo zwei Objekte mit kürzesten Weg auseinander gedrückt werden

PointIsInside-Check bei Polygonen (Konvex/Konkav)

Für diese Aufgabe habe ich folgenden Link gefunden:

<https://www.eecs.umich.edu/courses/eecs380/HANDOUTS/PROJ2/InsidePoly.html>

Ich nutze davon Solution1. Gegeben sind hier 3 rote Beispelpunkte für die geprüft werden soll, ob sie innerhalb des Polygons sind:



Ich berechne die Schnittpunkte mit den blauen Linien mit dem Polygon und zähle wie oft es ein Schnittpunkt gab. Ist die Anzahl gerade, dann liegt der Punkt außerhalb. Punkt 3 ist aber ein Randfall, weil die blaue Linie hier durch eine Polygon-Ecke geht. Um das zu verhindern, steht auf Zeile 18 nur `>` anstatt `>=`.

```

6 //Variante 1 von Solution 1 -> Funktioniert
7 0 Verweise
8 public static bool CheckInside1(Vec2D[] polygon, Vec2D p)
9 {
10     int counter = 0;
11     int i;
12     double xinters;
13     Vec2D p1, p2;
14
15     p1 = polygon[0];
16     for (i = 1; i <= polygon.Length; i++)
17     {
18         p2 = polygon[i % polygon.Length];
19         if (p.Y > Math.Min(p1.Y, p2.Y))      //Hier steht > und nicht >= um Ecken nicht doppelt zu zählen
20         {
21             if (p.Y <= Math.Max(p1.Y, p2.Y))
22             {
23                 if (p.X <= Math.Max(p1.X, p2.X)) //Wenn ich rechts neben der Linie starte, kann es kein Schnittpunkt geben
24                 {
25                     if (p1.Y != p2.Y)//Suche kein Schnittpunkt zwischen zwei horizontalen Linien
26                     {
27                         xinters = (p.Y - p1.Y) * (p2.X - p1.X) / (p2.Y - p1.Y) + p1.X; //X-Koordinante vom Schnittpunkt
28                         if (p1.X == p2.X || p.X <= xinters)
29                             counter++;
30                     }
31                 }
32             }
33         }
34         p1 = p2;
35
36         if (counter % 2 == 0)
37             return false;
38         else
39             return true;
40     }
}

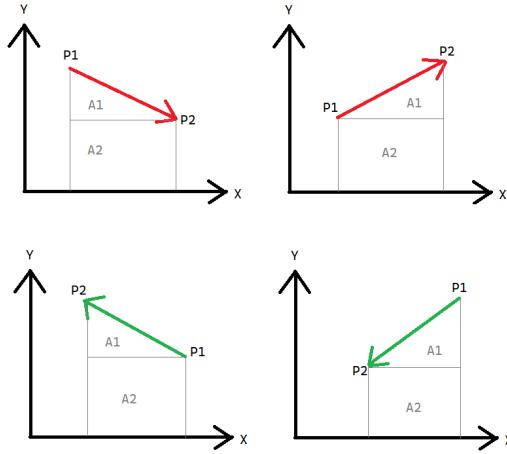
```

Siehe: PhysicEngine/MathHelper/PolygonHelper.cs

Fläche eines Polygons über die Trapezformel

Man kann über die Gaußsche Trapezformel den Flächeninhalt von ein Polygon berechnen. Siehe hier: https://de.wikipedia.org/wiki/Gau%C3%9Fsche_Trapezformel

Wenn ich den Flächeninhalt unter den roten und grünen Linien berechnen will, dann muss ich A1 + A2 rechnen.



Das sind die Flächenformeln für die 4 Fälle:

Fall 1 (Rot fallend):

$$A1 + A2 = \frac{1}{2} (x_2 - x_1) * (y_1 - y_2) + (x_2 - x_1) * y_2 \rightarrow \text{Klammere } (x_2 - x_1) \text{ aus}$$

$$A1 + A2 = \left(\frac{1}{2} * (y_1 - y_2) + y_2 \right) * (x_2 - x_1) \rightarrow \text{Multipliziere die innere Klammer mit } 1/2$$

$$A1 + A2 = \left(\frac{1}{2} * y_1 + \frac{1}{2} * y_2 \right) * (x_2 - x_1) \rightarrow \frac{1}{2} \text{ aus der linken Klammer raus nehmen}$$

$$A1 + A2 = \frac{1}{2} (y_1 + y_2) (x_2 - x_1)$$

Fall 2 (Rot steigend):

$$A1 + A2 = \frac{1}{2} (x_2 - x_1) * (y_2 - y_1) + (x_2 - x_1) * y_1 \rightarrow \text{Klammere } (x_2 - x_1) \text{ aus}$$

$$A1 + A2 = \left(\frac{1}{2} (y_2 - y_1) + y_1 \right) * (x_2 - x_1) \rightarrow \text{Innere Klammer } * 1/2 \text{ und danach dann } \frac{1}{2} \text{ raus nehmen}$$

$$A1 + A2 = \frac{1}{2} (y_1 + y_2) (x_2 - x_1)$$

Fall 3 (Grün steigend):

$$A1 + A2 = \frac{1}{2} (x_1 - x_2) * (y_2 - y_1) + (x_1 - x_2) * y_1 \rightarrow \text{Klammere } (x_1 - x_2) \text{ aus}$$

$$A1 + A2 = \left(\frac{1}{2} (y_2 - y_1) + y_1 \right) * (x_1 - x_2) \rightarrow \text{Innere Klammer } * 1/2 \text{ und danach dann } \frac{1}{2} \text{ raus nehmen}$$

$$A1 + A2 = \frac{1}{2} (y_1 + y_2) (x_1 - x_2)$$

Fall 4 (Grün fallend):

$$AI + A2 = \frac{1}{2}(x_1 - x_2) * (y_1 - y_2) + (x_1 - x_2) * y_2 \rightarrow \text{Klammere } (x_1 - x_2) \text{ aus}$$

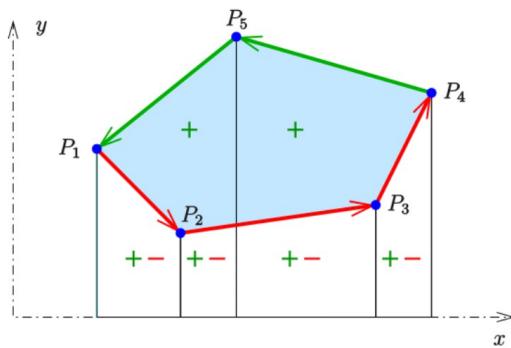
$$AI + A2 = \left(\frac{1}{2}(y_1 - y_2) + y_2 \right) * (x_1 - x_2) \rightarrow \text{Innere Klammer } * 1/2 \text{ und danach dann } 1/2 \text{ raus nehmen}$$

$$AI + A2 = \frac{1}{2}(y_1 + y_2) * (x_1 - x_2)$$

Für die beiden roten Linien ist die Flächenformel gleich und somit: $AI + A2 = \frac{1}{2}(y_1 + y_2)(x_2 - x_1)$

Für die beiden grünen Linien gilt die Formel: $AI + A2 = \frac{1}{2}(y_1 + y_2)*(x_1 - x_2)$

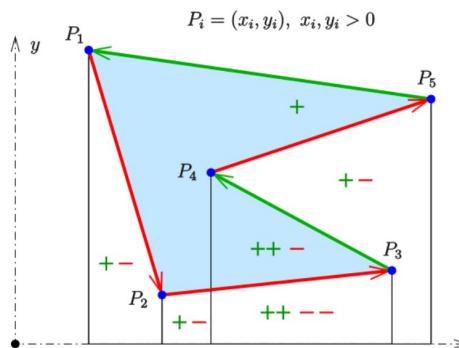
Wenn ich ein Polygon gegen den Uhrzeigersinn gegeben habe, dann kann ich dessen Fläche dadurch berechnen, indem ich die Flächen unter den grünen Linien nehme und davon die roten Flächen abziehe.



Wenn ich ein Minus eins in die rechte Klammer von der roten Flächenformel nehme, dann bekomme ich die gleiche Formel, wie ich es für die grüne Linie habe. Somit muss ich nur über alle Linien iterieren und die Summe von den Linien-Flächen mit folgender Formel bilden:

$$A = \frac{1}{2} \sum_{i=1}^n (y_i + y_{i+1})(x_i - x_{i+1})$$

Diese Formel funktioniert auch bei konkaven Polygonen:



Sollte das Polygon nicht CCW angegeben sein, dann kommt bei der Fläche eine negative Zahl raus. Es reicht, wenn man den Betrag von der Zahl nimmt so erhält man für alle möglichen Polygonarten die Fläche.

```

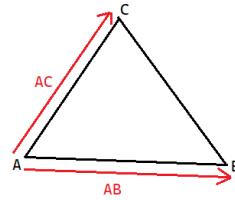
12
13     public static float GetAreaFromPolygon(Vec2D[] polygon)
14     {
15         float area = 0;
16         for (int i=0; i<polygon.Length; i++)
17         {
18             //area += polygon[i].X * (polygon[(i + 1) % polygon.Length].Y - polygon[(i - 1 + polygon.Length) % polygon.Length].Y)
19             area += (polygon[i].Y + polygon[(i + 1) % polygon.Length].Y) * (polygon[i].X - polygon[(i + 1) % polygon.Length].X);
20         }
21         return Math.Abs(area * 0.5f);
    }
```

Siehe: PhysicEngine/MathHelper/PolygonHelper.cs

Fläche eines Polygons über das Kreuzprodukt

Um zu verstehen, wie man den Schwerpunkt bei ein Polygon berechnet ist es nötig, dass ich hier noch eine andere Formel zur Berechnung der Polygonfläche zeige.

Ich kann den Flächeninhalt von ein Dreieck über das Kreuzprodukt von zwei Kanten berechnen:

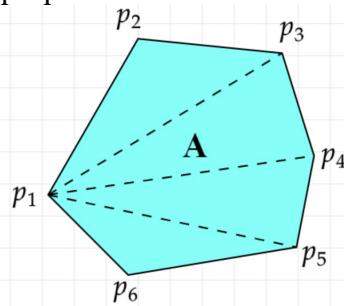


$$\text{TriangleArea} = \frac{1}{2} |\mathbf{AB} \times \mathbf{AC}|$$

Quelle: https://en.wikipedia.org/wiki/Area_of_a_triangle#Using_vectors

Außerdem kann ein konvexes Polygon in lauter Dreiecke zerlegt werden, indem ich ein beliebigen Polygoneckpunkt nehme und von dort aus dann zu jeder Kante ein Dreieck bilde:

p1-p2-p3 p1-p3-p4 p1-p4-p5 p1-p5-p6

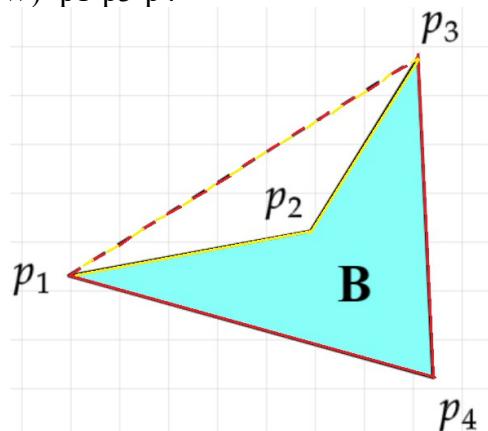


Quelle für das Bild: <https://fotino.me/moment-of-inertia-algorithm/>

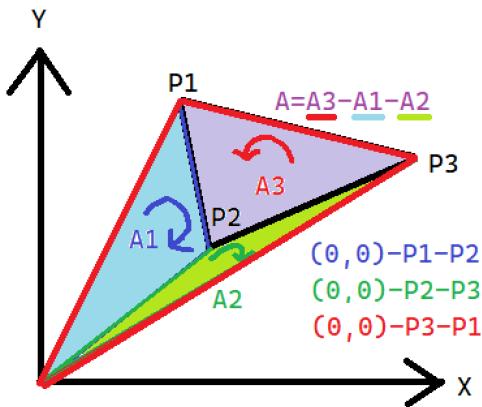
Die Fläche vom Polygon ist hier die Summe der Dreiecksflächen.

Wenn ich für das folgende konkaves Polygon diese Zerlegung mache, dann bekomme ich zwei Dreiecke:

Gelb(CCW)=p1-p2-p3 Rot(CW)=p1-p3-p4



Die Polygonfläche ergibt sich aus der Fläche vom roten Dreieck minus dem gelben Dreieck. Wenn ich die Fläche des Polygons über das Kreuzprodukt berechne und nicht den Betrag davon nehme, dann bekommt automatisch das rote Dreieck ein positives Vorzeichen und das gelbe Dreieck wird negativ. In diesen Beispiel ist p1 der gemeinsame Punkt von beiden Dreiecken. Es ist aber auch erlaubt, dass ich ein beliebigen Punkt als gemeinsamen Dreieckspunkt nehme. Wenn ich den Null-Punkt nehme, dann sehen die Dreiecke so aus:



Die Fläche vom Dreieck P1-P2-P3 ergibt sich aus den roten Dreieck minus Blau und Grün. Das Kreuzprodukt sorgt automatisch durch die Reihenfolge der Operanden dafür, dass das Vorzeichen stimmt. So sieht nun die Funktion zur Polygonflächenberechnung aus, wenn ich das Kreuzprodukt verwende und als gemeinsamen Punkt für alle Dreiecke den Nullpunkt nehme.

```

73     public static float GetAreaFromPolygon1(Vec2D[] polygon)
74     {
75         float area = 0;
76         for (int i = 0; i < polygon.Length; i++)
77         {
78             var p1 = polygon[i];
79             var p2 = polygon[(i + 1) % polygon.Length];
80
81             area += Vec2D.ZValueFromCross(p1, p2); //Area from Triangle p1-p2-[0;0] = 1/2*|Cross(p1,p2)|
82         }
83         return Math.Abs(area * 0.5f);
84     }

```

Schwerpunkt eines Polygons

Suche ich nach CenterOfGravity für ein Polygon, dann finde ich das hier:

https://en.wikipedia.org/wiki/Centroid#Of_a_polygon

<https://demonstrations.wolfram.com/CenterOfMassOfAPolygon/>

$$C_x = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i),$$

$$C_y = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i),$$

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i).$$

Da die Formeln stimmen sehe ich dadurch, dass ich eine Kontrollfunktion geschrieben habe, welche alle Pixel einer Shape als Punktmassen betrachtet.

```

61     public static Vec2D CenterOfGravity(IEditorShape shape)
62     {
63         bool[,] map = ShapeToBoolMap(shape);
64         long xSum = 0, ySum = 0;
65         int counter = 0;
66
67         for (int x = 0; x < map.GetLength(0); x++)
68             for (int y = 0; y < map.GetLength(1); y++)
69             {
70                 if (map[x, y])
71                 {
72                     counter++;
73                     xSum += x;
74                     ySum += y;
75                 }
76             }
77
78         return shape.GetBoundingBox().Min + new Vec2D(xSum / (float)counter, ySum / (float)counter);
79     }

```

$$M = \sum_{i=1}^N m_i$$

$$\frac{\sum m_i r_i(t)}{M}$$

M=counter=Masse des Körpers
mi=Masse des Pixels
ri=Position (x,y) des Pixels

Wenn ich den Schwerpunkt von ein Dreieck mit konstanter Dichte berechnen will, dann ist das die Summe der 3 Eckpunkt geteilt durch Drei:

$$C = \frac{1}{3}(L + M + N)$$

Centroid
=Schwerpunkt
=Geometrischer Schwerpunkt

Quelle: https://en.wikipedia.org/wiki/Centroid#Of_a_triangle

Wenn man die Schwerpunktfunction mit der Flächenfunktion vergleicht dann sieht man, dass das Polygon in lauter Dreiecke zerlegt wird, wo der gemeinsame Punkt der Nullpunkt ist. Dann wird von jeden dieser Dreiecke der Schwerpunkt und dessen vorzeichenbehaftete Fläche berechnet. Das Gewicht von den Dreieck ergibt sich aus der Fläche (mal Dichte von Eins). Nun wird so getan, als ob sich das gesamte Gewicht von diesen Dreieck an dessen Schwerpunkt befindet. Das gesamte Polygon besteht nun also als lauter Massepunkten, welche an den Schwerpunkten seiner Dreiecke liegt. Der Schwerpunkt vom Polygon ist dann die Summe der Dreiecksmassepunkte (Laut der Formel wie es bei der Kontrollfunktion steht).

```

86 //Quelle1: https://en.wikipedia.org/wiki/Centroid#Of_a_polygon
87 //Quelle2: https://demonstrations.wolfram.com/CenterOfMassOfAPolygon/
88 public static Vec2D GetCenterOfMassFromPolygon(Vec2D[] polygon)
89 {
90     Vec2D pos = new Vec2D(0, 0);
91     float area = 0;
92     for (int i = 0; i < polygon.Length; i++)
93     {
94         var p1 = polygon[i];
95         var p2 = polygon[(i + 1) % polygon.Length];
96
97         float s = p1.X * p2.Y - p2.X * p1.Y; //s=Vec2D.ZValueFromCross(p1, p2);
98         pos += (p1 + p2) * s;//Center from Triangle p1-p2-[0;0] = 1/3*(p1 + p2 + new Vec2D(0,0))
99
100        area += s;
101    }
102
103    float f = 1 / (3 * area);
104
105    return pos * f;
106 }
```

```

73     public static float GetAreaFromPolygon1(Vec2D[] polygon)
74     {
75         float area = 0;
76         for (int i = 0; i < polygon.Length; i++)
77         {
78             var p1 = polygon[i];
79             var p2 = polygon[(i + 1) % polygon.Length];
80
81             area += Vec2D.ZValueFromCross(p1, p2); //Area from Triangle p1-p2-[0;0] = 1/2*|Cross(p1,p2)|
82         }
83     }
84 }
```

Inertia eines Polygons

Der Inertia-Wert von ein beliebigen 2D-Objekt berechnet sich über $I = \int_V \rho(\mathbf{r}) \mathbf{r}^2 dV$

$\rho(\mathbf{r})$ =Dichte; \mathbf{r} =Kleinster Abstand zur Drehachse.

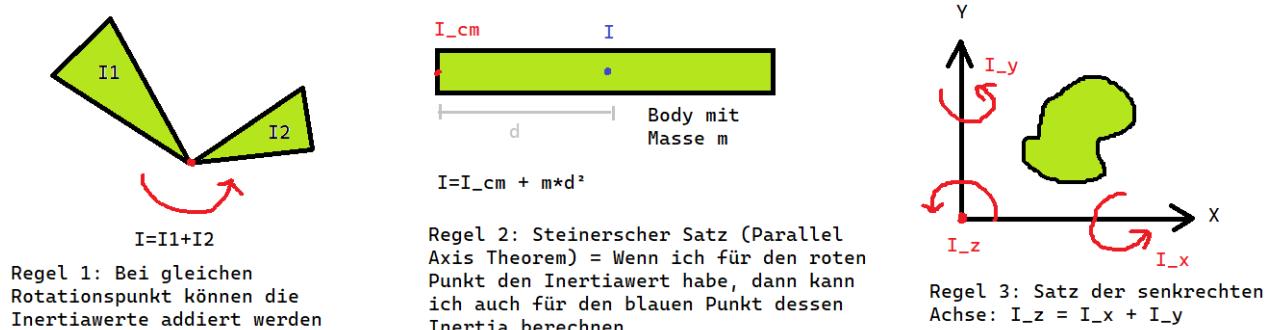
Quelle: <https://www.toptal.com/game/video-game-physics-part-i-an-introduction-to-rigid-body-dynamics>

Es gibt folgende Grundregeln für den Inertia-Wert:

Quelle für Inertia-Summenregel: <https://fotino.me/moment-of-inertia-algorithm/>

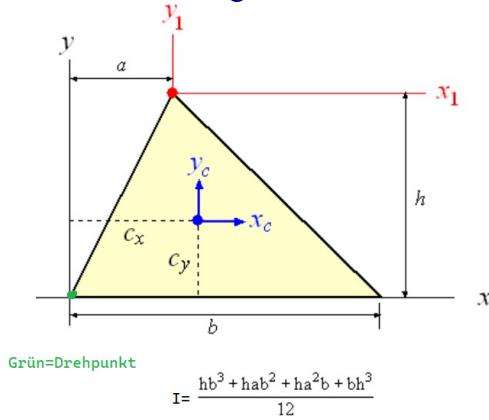
Quelle für Parallel Axis Theorem: https://en.wikipedia.org/wiki/Parallel_axis_theorem

Quelle für senkrechte Achse: https://en.wikipedia.org/wiki/Perpendicular_axis_theorem

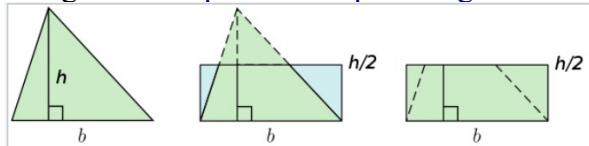


Wenn ich den Inertia-Wert für ein Dreieck berechnen will, was über den linken unteren grünen Punkt gedreht werden soll, dann brauche ich folgende Formel:

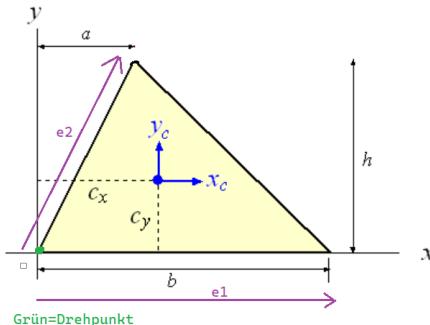
Quelle: <https://www.efunda.com/math/areas/triangle.cfm>



Wenn ich den Flächeninhalt von ein beliebigen Dreieck berechnen will, dann gilt $A=b*h/2$
Hier ist die Grafische Herleitung dafür: https://en.wikipedia.org/wiki/Area_of_a_triangle



Ich nehme nun das gelbe Dreieck und erweitere es noch um die Vektoren e_1 und e_2 , welche unnormiert vom grünen Eckpunkt zu den anderen Eckpunkten zeigen:



Wenn ich den Flächeninhalt vom gelben Dreieck berechnen will, dann kann ich entweder das Kreuzprodukt nehmen: $A = \frac{|e_1 \times e_2|}{2}$

Oder ich nehme die Formel vom grünen Dreieck: $A = \frac{b * h}{2}$

Ich definiere die Variable D wie folgt: $D = (e_1 \times e_2)_z$ D ist der Z-Wert vom Kreuzprodukt.

Wenn ich die beiden Flächenformeln gleichsetze, bekomme ich: $\frac{D}{2} = \frac{b * h}{2}$ $D = b * h$

Außerdem sehe ich, dass die Länge vom e_1 -Vektor b entspricht. Also gilt für das Quadrat davon: $e_1 * e_1 = b^2$

Wenn ich den Satz von Pythagoras anwende, dann gilt für die Quadratlänge von e_2 :

$$e_2 * e_2 = h^2 + a^2$$

Will ich a ausrechnen, dann muss ich dazu nur e_1 auf das normierte e_2 projizieren. Dazu schreibe ich: $b = |e_1|$ $a = e_2 * \frac{e_1}{b}$

Ich stelle das um indem ich a und b auf die linke Seite bringe: $a * b = e_1 * e_2$

Ich nehme nun die Interiaformel vom gelben Dreieck was um den grünen Punkt gedreht werden soll:

$$I = \frac{h * b^3 + h * a * b^2 + h * a^2 * b + b * h^3}{12}$$

Bei dieser Formel sehe ich, dass in allen 4 Termen im Zähler $h*b$ auftaucht. Ich schreibe das vor den Bruch:

$$I = h * b \frac{b^2 + a * b + a^2 + h^2}{12}$$

Nun nutze ich folgende 4 Ersetzungsregeln:

$$D = b * h \quad e_1 * e_1 = b^2 \quad e_2 * e_2 = h^2 + a^2 \quad a * b = e_1 * e_2$$

Und setze das in die Inertia-Formel ein:

$$I = D \frac{e_1 * e_1 + e_1 * e_2 + e_2 * e_2}{12}$$

Auf diese Weise habe ich nun die Inertia-Formel so umgeändert, dass ich nur noch mit den Vektoren e_1 und e_2 arbeite. Ich schreibe den Zähler jetzt noch so aus, dass ich keine Vektoren dort stehen habe sondern nur noch skalare Zahlen. Dazu definiere ich:

$$ex1 = e1.X \quad ey1 = e1.Y \quad ex2 = e2.X \quad ey2 = e2.Y$$

$$I = D \frac{ex1 * ex1 + ey1 * ey2 + ex1 * ex2 + ey1 * ey2 + ex2 * ex2 + ey2 * ey2}{12}$$

Ich sortiere die Terme im Zähler nun so um, dass alles, was mit x zu tun kommt zuerst hin und danach alles was mit y zu tun hat.

$$I = D \frac{(ex1 * ex1 + ex1 * ex2 + ex2 * ex2) + (ey1 * ey2 + ey1 * ey2 + ey2 * ey2)}{12}$$

Ich definiere nun die Variablen $intx2$ und $inty2$:

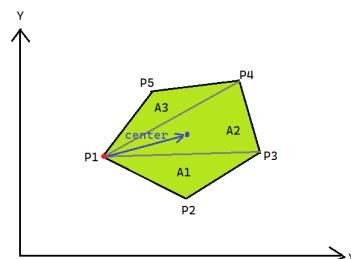
$$intx2 = ex1 * ex1 + ex1 * ex2 + ex2 * ex2$$

$$inty2 = ey1 * ey2 + ey1 * ey2 + ey2 * ey2$$

Somit wird die Dreiecks-Inertiaformel nun kürzer:

$$I = D \frac{intx2 + inty2}{12} \quad \text{mit} \quad D = (e_1 \times e_2)_z$$

Ich nehme nun ein konvexes Polygon und zerlege es in 3 Dreiecke, indem ich von $P1$ aus zu jedem Eckpunkt Kanten erzeuge:



Wenn ich von diesen Polygon den Inertia-Wert wissen will, dann berechne ich zuerst für jedes der 3 Dreiecke dessen Inertia-Wert, wo $P1$ der gemeinsame Drehpunkt ist. Dann bilde ich die Summe dieser 3 Inertia-Werte und dann nutze ich den Satz von Steiner um aus den Inertia-Wert für $P1$ den Inertia-Wert für den Schwerpunkt vom Polygon zu berechnen. Den Schwerpunkt erhalte ich, indem ich von jedem Dreieck dessen Schwerpunkt über die $C=1/3*(P1+P2+P3)$ -Formel berechne und mit dessen Flächeninhalt Wichte und am Ende die gewichtete Schwerpunktsumme durch die Gesamtfläche dividiere.

Mir ist bei der Inertia-Funktion fürs Polygon bei Erin Catto aufgefallen, dass er nach den gelb markierten Teil mit drin hat

https://github.com/erincatto/box2d/blob/main/src/collision/b2_polygon_shape.cpp#L274

```
// Shift to center of mass then to original body origin.
massData->I += massData->mass * (b2Dot(massData->center, massData->center) - b2Dot(center, center));
```

und bei mir auf Zeile 165 ist das nicht mit drin. Da ich meine Funktion übers Pixelzählen mit PhysicSceneEditorControl/Controls/Editor.Model/ShapeHelper.cs / GetInertia geprüft habe, bin ich mir sicher, dass meine Funktion so aber funktioniert.

```

124     public static float GetInertiaFromPolygon(float density, Vec2D[] polygon)
125     {
126         Vec2D center = new Vec2D(0, 0);
127         float area = 0;
128         float I = 0;
129         Vec2D s = polygon[0]; // Get a reference point for forming triangles. Use the first vertex to reduce round-off errors.
130         float k_inv3 = 1f / 3f;
131
132         for (int i=0;i<polygon.Length;i++)
133         {
134             // Triangle vertices.
135             Vec2D e1 = polygon[i] - s;
136             Vec2D e2 = i + 1 < polygon.Length ? polygon[i + 1] - s : polygon[0] - s;
137
138             float D = Vec2D.ZValueFromCross(e1, e2);
139
140             float triangleArea = 0.5f * D;
141             area += triangleArea;
142
143             // Area weighted centroid
144             center += triangleArea * k_inv3 * (e1 + e2);
145
146             float ex1 = e1.X, ey1 = e1.Y;
147             float ex2 = e2.X, ey2 = e2.Y;
148
149             float intx2 = ex1 * ex1 + ex2 * ex1 + ex2 * ex2;
150             float inty2 = ey1 * ey1 + ey2 * ey1 + ey2 * ey2;
151
152             I += (0.25f * k_inv3 * D) * (intx2 + inty2);
153
154             // Total mass
155             float mass = density * area;
156
157             // Center of mass (Shows from point s to local CenterOfMass; GlobalCenterOfMass=center+s)
158             center *= 1.0f / area;
159
160             // Inertia tensor relative to the local origin (point s).
161             I *= density;
162
163             // Shift to center of mass then to original body origin by using the parallel axis theorem
164             I -= mass * (center * center);
165
166         }
167
168         return Math.Abs(I); //I=Positiv -> Polygon is CW; Negativ -> Polygon is CCW
    }

```

Diese Funktion funktioniert nicht nur für konvexe Polygone sondern auch für konkave. Der Grund warum das so ist ist weil ähnlich wie bei der Funktion für die Flächenberechnung ist das Vorzeichen von D (Zeile 138) mal positive und mal negativ je nachdem, ob das Dreieck innerhalb des Polygons liegt oder außerhalb. Dadurch zieht man dann vom ursprünglich zu großen Dreieck das kleine äußere Dreieck dann ab.

Eine weitere Abweichung zur Schwerpunktfunction vom vorherigen Abschnitt ist, das hier die Dreiecke als gemeinsamen Punkt nicht den Nullpunkt nutzen sondern den Polygon[0]-Punkt. Dadurch ist center dann kein Richtungsvektor vom Nullpunkt zum Polygon-Schwerpunkt sondern es zeigt vom Polygon[0]-Punkt zum Schwerpunkt. Genau um diese Strecke muss dann der Inertia-Wert auf Zeile 165 dann korrigiert werden.

Konkaves Polygon in konvexe Polygone zerlegen

Da ich für die Schnittpunktberechnung SAT nutzen will, zerlege ich das konkave Polygon in konvexe. In diesen Link <https://github.com/ivanfratric/polypartition> sehe ich, dass der Polygon-Decomposition-Algorithmus von Mark Keil und Jack Snoeyink aus dem Jahre 1998 der beste Weg dafür ist. Der Algorithmus wird hier https://www.ime.usp.br/~cris/aulas/14_2_331/projetos/keil-snoeyink-long.pdf mit langen Worten, und hier https://www.researchgate.net/publication/2880601_On_the_Time_Bound_for_Convex_Decomposition_of_Simple_Polygons mit kurzen Worten beschrieben.

Sie haben für ihr Paper ein Java-Programm mitgeliefert:

<http://www.cs.unc.edu/~snoeyink/papers/convdecomp.ps.gz>

Wofür es auch eine C++ Übersetzung gibt:

<https://github.com/ivanfratric/polypartition/blob/master/src/polypartition.cpp#L968>

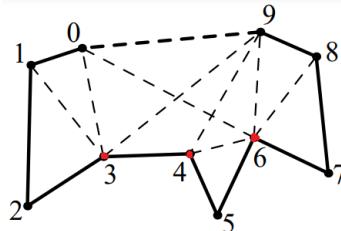
und in Haxe [https://de.wikipedia.org/wiki/Haxe_\(Programmiersprache\)](https://de.wikipedia.org/wiki/Haxe_(Programmiersprache)):

<https://github.com/azrafe7/hxGeomAlgo/blob/master/src/hxGeomAlgo/SnoeyinkKeil.hx>

Der Algorithmus funktioniert so, dass er zuerst für jeden Polygon-Punkt prüft, ob man eine Linie zu einem anderen Punkt innerhalb des Polygons zeichnen kann ohne dass ich dabei eine Polygonkante schneide. Diese Linien werden Diagonalen im Paper genannt.

Außerdem ermitteln sie für jeden Polygonpunkt, ob der innerhalb des Polygons gemessene Winkel größer als 180 Grad ist. Wenn das so ist, dann wird dieser Polygonpunkt als Reflex bezeichnet. Ansonsten Convex.

Beispiel: Bei diesen Polygon gibt es 3 Reflexpunkte. Ich erkenne sie daran, dass wenn ich in CCW-Richtung entlang des Polygons laufe, dann muss ich ein Rechtsknick an den Punkten 3, 4 und 6 machen. Bei den anderen Convpunkten mache ich immer ein Linksknick. Außerdem wurden hier einige Diagonalen als gestrichelte Linien gezeichnet. Die Linie 0 nach 9 ist keine Diagonale sondern eine Polygonkante.



Wenn ich nun dieses konkave Polygon in konvexe Polygone zerlegen will, dann muss ich die Reflexpunkte beseitigen indem ich dort eine Diagonale ansetze, die dann zu einem anderen Punkt (Konvex oder Reflex) führt. An dieser Diagonale teile ich das Polygon dann in zwei Unterpolygone auf. Die Unterpolygone teile ich an den Reflex-Punkten dann wieder auf, bis es keine Reflexpunkte mehr gibt. Der Algorithmus hier versucht nun mit minimaler Diagonalenanzahl alle Reflexpunkte zu beseitigen. Bei zwei Reflex-Punkten wären das Minimum eine Diagonale, die beide Reflexpunkte verbindet. In dem Beispiel hier mit den 3 Punkten kann ich die Reflexpunkte nicht direkt verbinden. Punkt 3 und 4 darf ich nicht verbinden, da sie per Polygonkante schon verbunden sind. Verbinde ich Punkt 4 und 6, dann entsteht ein Dreieck und ein Restpolygon, für dass ich dann weitere Diagonalen finden muss, um es Konvex zu machen. Es wäre aber besser vom Punkt 3 aus nur die Punkte 1, 0 und 9 als Diagonalendpunkte zu betrachten, da dort das linke Polygon dann konvex ist. Von 3 nach 8 wäre nicht gut, weil das linke Teilstück dann konkav ist.

Die Idee von den Algorithmus ist es nun für jede mögliche Diagonale ein Gewicht zu berechnen, was angibt, wie viele weiteren Diagonalen sind für die beiden Teilstücke nötig, um beide konvex zu machen. Wie aber genau diese Diagonalen-Gewichtsberechnung erfolgt und habe ich leider nicht verstanden. So ist mein Verständnis grob:

Die PolygonComposer-Klasse besteht aus 4 Teilen:

Siehe: PhysicEngine/MathHelper/PolygonComposition/PolygonComposer.cs

Zeile 131-226: Hier wird jeder Polygonpunkt in Convex/Reflex eingeteilt. Das wird in vertices[i].IsConvex gespeichert. Außerdem wird für jeden Polygonpunkt geprüft, zu welchen anderen Polygonpunkten es eine Sichtlinie gibt. Das wird in dpstates[i,j].Visible gespeichert.

Zeile 227-274: Hier wird für jede mögliche Diagonale zwischen i und j (dpstates[i,j].Visible ist true) das Gewicht dpstates[i, j].Weight berechnet und welche weiteren Folgediagonalen dpstates[i,j].Pairs dann nötig werden. Je nachdem ob der Punkt i Reflex (TypA) oder Convex (TypB) ist, gibt es eine andere Methode zur Gewichtsberechnung.

Zeile 277-351: Aus den dpstates[i,j].Pairs-Diagonalen werden nun die Diagonalen mit den kleinsten Gewicht genommen und in diagonals gespeichert.

Zeile 353-417: Das Polygon wird anhand der im vorherigen Schritt ausgewählten Diagonalen in konvexe Teilstücke zerlegt.

Prüfen das ein Polygon korrekt erstellt wurde

Wenn im Editor ein Polygon erzeugt wird, dann soll die AddPolygonFunction-Klasse auf drei Dinge achten:

- Keine Linie schneidet eine andere Linie
- Keine zwei Punkte liegen an der gleichen Stelle
- Das Polygon ist in CCW-Richtung angegeben

Während man mit der Maus den nächsten Polygonpunkt platziert, prüfe ich diese Dinge und nur wenn currentLineIntersects false ist, darf ein neuer Punkt erzeugt werden.

```
23 public override void HandleMouseMove(MouseEventArgs e)
24 {
25     currentPosition = new Vec2D(e.X, e.Y);
26 
27     this.currentLineIntersects = false;
28     for (int i=0;i<points.Count - 2;i++)
29     {
30         if (PolygonHelper.IntersectLines(points[i], points[i+1], points.Last(), currentPosition))
31         {
32             this.currentLineIntersects = true;
33             break;
34         }
35     }
36 
37     if (this.points.Any() && (this.points.Last() - currentPosition).Length() < MinPointDistance)
38         this.currentLineIntersects = true;
39 }
40 
41 private void CreatePolygon()
42 {
43     if (this.points.Count >= 3)
44     {
45         //Stelle sicher, dass das Polygon immer in CCW-Richtung angegeben ist
46         Vec2D[] polygon = PolygonHelper.OrderPointsCCW(this.points.ToArray());
47 
48         this.shapes.Add(new EditorPolygon(polygon));
49     }
50 }
```

Siehe: EditorControl/Model/Function/Shapes/AddPolygonFunction.cs

Die beiden Polygonarten aus Sicht der Kollisionerkennung

Aus Sicht der Kollisionserkennung gibt es zwei Polygonarten: EdgePolygon und RigidPolygon. Beim EdgePolygon soll geprüft werden, ob ein Objekt mit dem Rand kollidiert. Beim RigidPolygon soll geprüft werden, ob ein Objekt innerhalb vom Polygon liegt und dann soll es dort raus gedrückt werden. Ich nutze für beide Polygonarten die gemeinsame Basisklasse *ConcavePolygon*, welche all die Interfaces implementiert, welche das IRigidBody-Interface vorschreibt außer das ICollidable-Interface.

Die Ableitungen der ConcavePolygon-Klasse erweitern diese Klasse um ICollidable und ICollidableContainer.

```
internal abstract class ConcavePolygon : IBoundingCircle, IForceable, IMoveable, IExportableBody, IClickable, IPublicRigidBody, IPublicRigidPolygon
{
    internal class EdgePolygon : ConcavePolygon, IRigidBody, ICollidableContainer
    {
        7 Verweise
        public ICollidable[] Collidables { get; }
        private PolygonEdge[] edges;
    }

    internal class RigidPolygon : ConcavePolygon, IRigidBody, ICollidableContainer
    {
        7 Verweise
        public ICollidable[] Collidables { get; }
        private CollidableConvexPolygon[] subPolys;
    }

    private static CollisionInfo[] GetCollisions(ICollidable c1, ICollidable c2)
    {
        if ((c1 is ICollidableContainer)==false && (c2 is ICollidableContainer) == false)
        {
            return NearPhaseTests.Collide(c1, c2); //Single with Single
        }
        else if ((c1 is ICollidableContainer) && (c2 is ICollidableContainer)==false) //Container with Single
        {
            return (c1 as ICollidableContainer).Collidables.SelectMany(x => NearPhaseTests.Collide(x, c2)).ToArray();
        }
        else if ((c1 is ICollidableContainer) == false && (c2 is ICollidableContainer)) //Single with Container
        {
            return (c2 as ICollidableContainer).Collidables.SelectMany(x => NearPhaseTests.Collide(c1, x)).ToArray();
        }
        else //Container with Container
        {
            List<CollisionInfo> collisions = new List<CollisionInfo>();
            var container1 = c1 as ICollidableContainer;
            var container2 = c2 as ICollidableContainer;
            foreach (var single1 in container1.Collidables)
            {
                foreach (var single2 in container2.Collidables)
                {
                    collisions.AddRange(NearPhaseTests.Collide(single1, single2));
                }
            }
            return collisions.ToArray();
        }
    }
}
```

```
internal interface ICollidable
{
    49 Verweise
    Vec2D Center { get; } //Massezentrum des Objekts
    8 Verweise
    CollidableType TypeId { get; }
    15 Verweise
    List<ICollidable> CollideExcludeList { get; }
    18 Verweise
    int CollisionCategory { get; } //Die Kollisionskategorie
}

12 Verweise
internal interface ICollidableContainer
{
    10 Verweise
    ICollidable[] Collidables { get; }
}
```

Der ICollidableContainer enthält entweder eine Menge von PolygonEdges oder CollidableConvexPolygons. In der GetCollisions-Funktion wird für jedes Containerelement geprüft, ob es eine Kollision mit anderen ICollidables gab.

Schnittpunkt bei ein EdgePolygon

Beim EdgePolygon geht es darum, dass ein Schnittpunkt nur dann mit ein anderen Objekt erfolgen soll, wenn eine Polygon-Kante berührt wird. So ist es erlaubt, dass ein Objekt innerhalb des Polygons sich ohne Kollision aufhalten darf. Diese Polygonart wird für den Levelrand benötigt. Ein EdgePolygon besteht aus einer Liste von PolygonEdges. Bei ein Inside-EdgePolygon zeigen all Normalen von dessen Kanten nach innen und bei ein Outside-EdgePolygon zeigen sie nach außen. Ein Schnittpunkt zwischen ein EdgePolygon und ein anderen Objekt wird ermittelt, indem für jede Kante der Schnittpunkt ermittelt wird.

Schnittpunkt PolygonEdge – Circle

Jeder Linienpunkt P, welcher bei einer Linie, die von P1 nach P2 geht kann mit $t=0..1$ so beschrieben werden:

$$V = P2 - P1 \quad P = P1 + V * t$$

Jeder Kreispunkt P, der auf dem Rand von ein Kreis mit Zentrum m und Radius r liegt, kann implizit so beschrieben werden:

$$(P_x - m_x)^2 + (P_y - m_y)^2 = r^2$$

Den Schnittpunkt zwischen der Linie und dem Kreis erhalte ich durch einsetzen des Linienpunktes in die Kreisgleichung:

$$(P1_x + V_x * t - m_x)^2 + (P1_y + V_y * t - m_y)^2 = r^2$$

Wenn ich diese Gleichungen nach t umstelle, dann erhalte ich die Schnittpunktkoordinaten.

Ich ersetze die Konstanten: $a = P1_x - m_x \quad b = P1_y - m_y$

$$(a + V_x * t)^2 + (b + V_y * t)^2 = r^2 \rightarrow 1. \text{ binomische Formel jeweils für beide Klammern anwenden}$$

$$a^2 + 2*a*V_x*t + V_{x^2}*t^2 + b^2 + 2*b*V_y*t + V_{y^2}*t^2 = r^2 \rightarrow t \text{ auf die linke Seite}$$

$$(2*a*V_x + 2*b*V_y)*t + (V_{x^2} + V_{y^2})*t^2 = r^2 - a^2 - b^2 \rightarrow \text{Ersetze mit c,d,e}$$

$$c = 2*a*V_x + 2*b*V_y \quad d = V_{x^2} + V_{y^2} \quad e = a^2 + b^2 - r^2$$

$$d*t^2 + c*t + e = 0 \rightarrow \text{Anwendung der Mitternachtsformel:}$$

Die Lösungen einer quadratischen Gleichung $ax^2 + bx + c = 0$ lauten:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Quelle: <https://de.serlo.org/mathe/1549/mitternachtsformel-quadratische-l%C3%B6sungsformel>

$$t_{1,2} = \frac{-c \pm \sqrt{c^2 - 4de}}{2d} \quad \text{mit} \quad f_1 = \sqrt{c^2 - 4de} \quad f_2 = \frac{1}{2d}$$

$$t_1 = (-c + f_1) * f_2 \quad t_2 = (-c - f_1) * f_2$$

Der Kollisionspunkt ist so, dass dessen Normale immer der Edge-Normale entspricht. Der Kollisionspunkt liegt vom Kreiszentrum aus in Richtung Normale. Auf diese Weise habe ich immer nur ein Kollisionspunkt selbst wenn der Kreis an zwei Stellen die Linie schneidet.

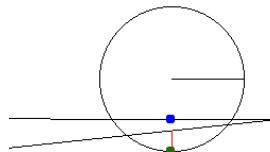
```

8     internal static CollisionInfo[] EdgeCircle(ICollideablePolygonEdge edge, ICollidableCircle circle)
9     {
10        Vec2D p1ToC = circle.Center - edge.P1;
11        float normalDistance = p1ToC * edge.Normal;
12        if (normalDistance > circle.Radius) return new CollisionInfo[0];
13        if (normalDistance < 0) return new CollisionInfo[0]; //Keine Kollision mit der Linie, wenn das Kreiszentrum hinter der Linie liegt
14
15        Vec2D p1 = edge.P1;
16        Vec2D p2 = edge.P2;
17        Vec2D m = circle.Center;
18        float radius = circle.Radius;
19
20        Vec2D V = p2 - p1;
21        float a = p1.X - m.X, b = p1.Y - m.Y, c = 2 * a * V.X + 2 * b * V.Y, d = V.X * V.X + V.Y * V.Y, e = a * a + b * b - radius * radius;
22        float f1 = (float)Math.Sqrt(c * c - 4 * d * e), f2 = 1.0f / (2 * d);
23        float t1 = (-c + f1) * f2, t2 = (-c - f1) * f2;
24        if (t1 >= 0 && t1 <= 1 || t2 >= 0 && t2 <= 1)
25        {
26            return new CollisionInfo[] { new CollisionInfo(circle.Center - edge.Normal * circle.Radius, edge.Normal, circle.Radius - normalDistance, 0, 0) };
27        }
28
29    }
30

```

Siehe: PhysicEngine/CollisionDetection/NearPhase/Polygon/EdgeOtherCollision.cs

Der Grund, warum ich bei Zeile 13 nur dann eine Kollision mit der Kante berechne, wenn das Kreiszentrum vor der Kante liegt ist, weil ich verhindern will, dass ein Kreis durch eine spitze Ecke hindurchgedrückt wird, wenn er zwei Kanten schneidet. In diesen Beispiel hier soll der Kreis nur nach oben gedrückt werden auch wenn er die untere Kante dessen Normale nach unten zeigt, mit durchschneidet.

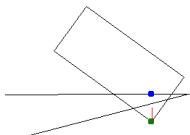


Schnittpunkt PolygonEdge – konvexes Rigid-Polygon

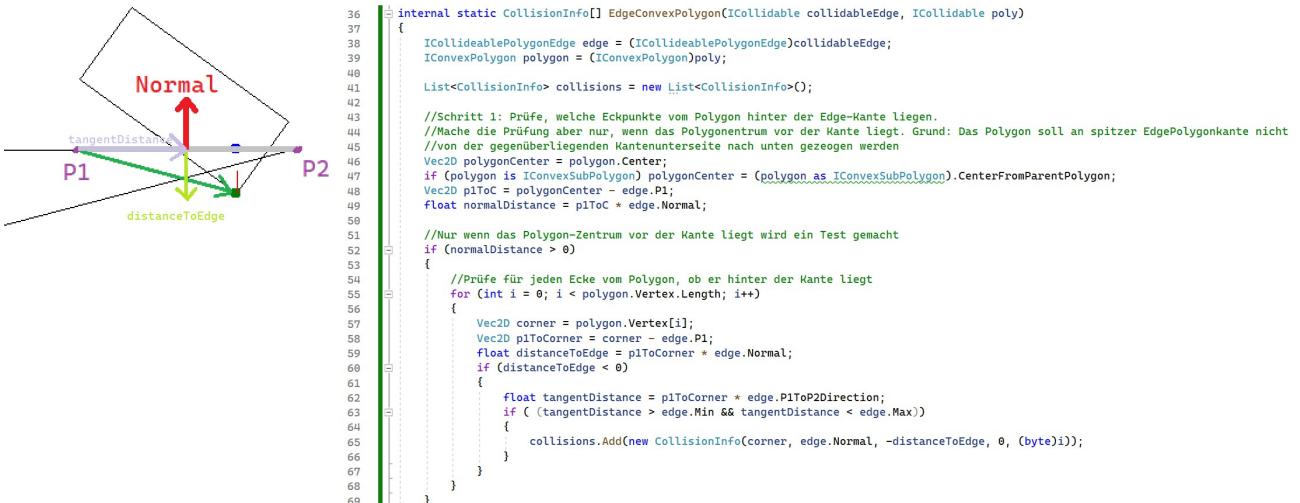
Hinweis: Die PolygonEdge wird hier Edge genannt. Sie sind Teilobjekte vom EdgePolygon.

Es gibt zwei Möglichkeiten, wie es zu einen Schnittpunkt zwischen einer Edge und ein konvexen Polygon (z.B. Rechteck) kommen kann: Einer der beiden Edge-Endpunkte liegt innerhalb des Polygons oder einer der Polygonecken liegt hinter der Edge.

Beispiel: Das konvexe Rigid-Polygon ist ein Rechteck. Vom EdgePolygon sieht man zwei Kanten auf dem Bild. Die rechte untere Rechteckecke durchschneidet sowohl die obere als auch die untere Edge. Es darf aber nur für die oberen Kante ein Kollisionspunkt erzeugt werden, da die Normale von der unteren Kante nach unten zeigt. Sie würde das Rechteck noch weiter nach unten in das Polygon rein drücken.

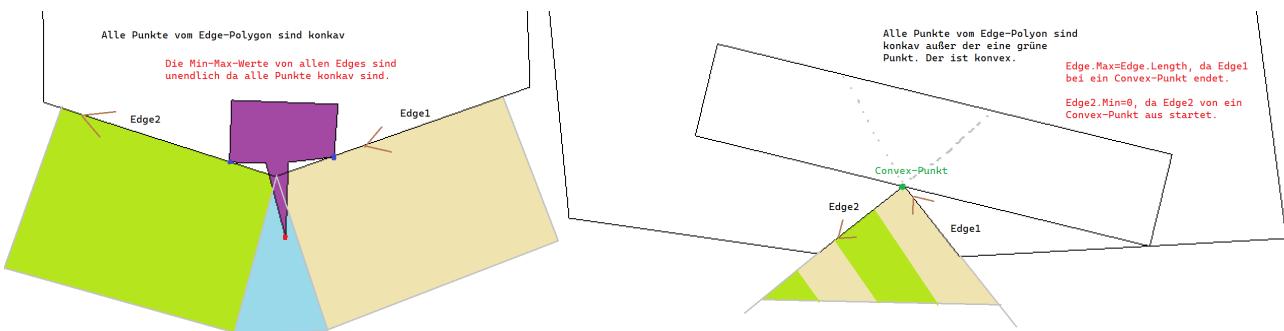


Der erste Schritt ermittelt all die Polygonpunkte, welche die Edge (geht von P1 nach P2) durchschneiden. Dazu muss die distanceToEdge kleiner Null sein und die tangentDistance zwischen P1 und P2 liegen.



Siehe: PhysicEngine/CollisionDetection/NearPhase/Polygon/EdgeOtherCollision.cs

Auf Zeile 63 sieht man, dass tangentDistance nicht von 0 bis edge.Length sondern von edge.Min bis edge.Max. Der Grund dafür wird mir folgenden Bild erklärt:



Linkes Bild: Die Spitze vom Lila-Polygon befindet sich beim EdgePolygon zwischen Edge1 und Edge2. Würde man bei der Edge2-Kante die tangentDistance von 0 bis edge.Length kontrollieren, dann käme es nur dann zur Kollision, wenn ein Punkt im grünen Bereich liegt. Das gleiche gilt für Edge1. Der blaue Bereich würde nicht zur Kollision führen. Aus dem Grund ist edge.Min im linken Bild Float.MinValue und edge.Max ist Float.MaxValue.

Ich darf aber nicht immer mit den Float.Min/MaxValues arbeiten da das EdgePolygon auch eine Ecke haben kann, welche konvex ist (Huckel). Würde ich das im rechten Bild so machen, dann würde das Rechteck nicht auf dem grünen Konvex-Punkt aufliegen sondern auf der gestrichelten Verlängerungslinie von den Edge-Kanten. Damit im rechten Bild die Kollisionsabfrage richtig funktioniert, darf Edge1 als Max-Wert nur bis Edge-Length gehen und Edge2 darf als Min-Wert nur von 0 aus starten.

Um zu ermitteln, was die Edge-Min/Max-Werte sind, muss ich schauen, ob die Punkte des EdgePolygons konvex sind oder nicht.

Ich definiere ein konvexen Punkt dadurch, dass der Winkel zu sein angrenzenden Kanten größer als 180 Grad ist. Beim EdgePolygon, wo die Normalen nach innen zeigen (linkes Polygon im Bild) messe ich den Winkel auf der Innenseite. Beim EdgePolygon mit nach außen zeigenden Edgenormalen messe ich außen (rechtes Polygon).

In diesen Bild sind die Konvex-Punkte blau eingezzeichnet:

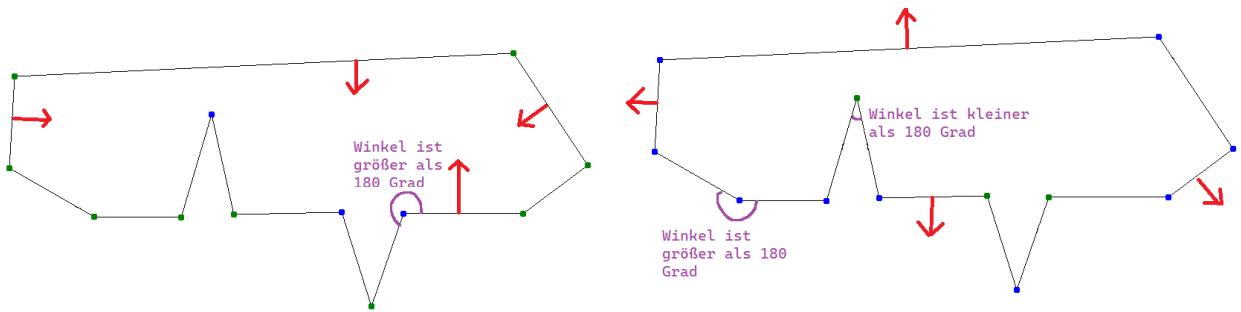


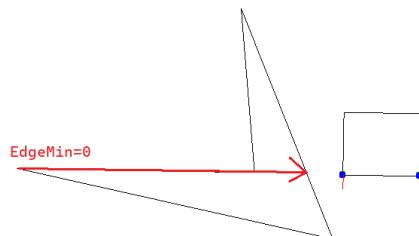
Bild: So wird beim EdgePolygon definiert, welche Punkte konvex sind.

Wenn eine Edge auf ein konvexen Punkt endet, dann ist Edge.Max = Edge.Length. Ansonsten ist es Edge.MaxValue. Wenn eine Edge von einem konvexen Punkt startet, dann gilt Edge.Min = 0. Ansonsten gilt Edge.Min = Edge.MinValue.

Die erste Prüfung hat nun all die Punkte vom Polygon/Rechteck ermittelt, welche hinter der Edge liegen.

Achtung: Man darf bei der Edge aber nicht einfach Edge.MinValue/Edge.MaxValue nutzen sondern nur bis zum nächsten Schnittpunkt mit der Kante, die nicht direkter Nachbar ist. Wenn man die rote Edge von 0 bis Unendlich gehen lassen würde, dann würde der Würfel mit ihr kollidieren (siehe Bild).

Deswegen darf sie nur bis zum Pfeilende gehen (Schnittpunkt mit der rechten Polygonkante)



Ich ermittle für jede Kante die Schnittpunkte mit allen anderen Kanten und schaue dann, was ist der kleinste Max-Wert, welcher länger als die EdgeLength ist und was ist der die größte negative Min-Zahl:

```

35 //Schritt 2: Berechne wie lang die Edges maximal sein dürfen indem von jeder Edge alle Schnittpunkte mit allen Nicht-Nachbarkanten berechnet
36 float[] minValues = new float[this.Vertex.Length];
37 float[] maxValues = new float[this.Vertex.Length];
38 for (int i = 0; i < this.Vertex.Length; i++)
39 {
    //Prüfe für Edge [i]-[i+1] und ermittle seine Min/Max-Werte
40     float min = float.MinValue; //Größte negative Zahl ist gesucht
41     float max = float.MaxValue; //Kleinste Zahl, die größer als edgeLength ist, ist gesucht
42
43     float edgeLength = (this.Vertex[(i + 1) % this.Vertex.Length] - this.Vertex[i]).Length();
44
45     //Gehe durch alle Edges durch, die nicht den Punkt [i] oder [i+1] enthalten
46     for (int j = 0; j < this.Vertex.Length; j++)
47     {
48         if (j != i && j != i + 1 && (j + 1) != i && (j + 1) != (i + 1)) //Kante [j]-[j+1] ist kein Nachbar von [i]-[i+1]
49         {
50             var p1l = this.Vertex[i];
51             var p1z = this.Vertex[i + 1] % this.Vertex.Length;
52             var p2l = this.Vertex[j];
53             var p2z = this.Vertex[j + 1] % this.Vertex.Length;
54
55             PolygonHelper.IntersectionTwoRays(p1l, (p1z - p1l).Normalize(), p1l, (p2z - p2l).Normalize(), out float t1, out float t2);
56             if (float.IsNaN(t1) == false && float.IsInfinity(t1) == false)
57             {
58                 if (t1 < 0)
59                     min = Math.Max(min, t1);
60
61                 if (t1 > edgeLength)
62                     max = Math.Min(max, t1);
63             }
64         }
65     }
66
67     minValues[i] = min - edgeLength;
68     maxValues[i] = max;
69 }
70
71
72

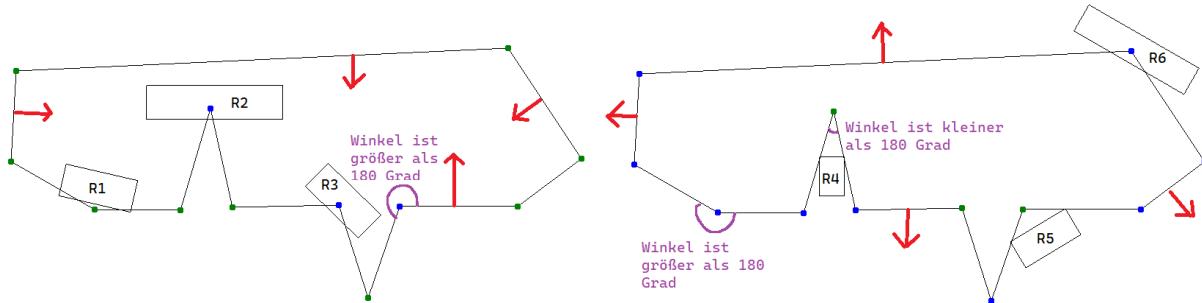
```

Siehe: PhysicEngine/RigidBody/Polygon/EdgePolygon.cs

Die zweite Möglichkeit, wie es zum Schnittpunkt zwischen einem EdgePolygon/Polygonkante und einem Konvex-Polygon/Rechteck kommen kann ist, dass ein Edge-P1/P2-Punkt innerhalb des konvexen Polygons liegt.

Dazu gehe ich durch alle Seiten vom konvexen Polygon und prüfe, ob der P1-Punkt innerhalb des Polygons liegt. Ich muss diese Prüfung nicht für P2 machen, da ja für jede Edge die

Kollisionsabfrage gemacht wird. Somit ergeben alle Edge.P1-Punkte das vollständige EdgePolygon. Der Edge.P1-Punkt kann allerdings nur dann innerhalb vom Rechteck/Konvexpolygon liegen, wenn er konvex (Huckel) ist. Beispiel: Es sind hier 6 Rechtecke gegeben wo geprüft werden soll, ob sich innerhalb des Rechtecks EdgePolygon-Punkte befinden. Nur die blauen konvex-Punkte können in das Rechteck eindringen (Siehe R2,R2,R6). Die grünen Konkav-Punkte können nicht ins Rechteck, da vorher schon die Eckpunkte vom Rechteck mit der Kante kollidieren (siehe R1, R4).



Zeile 73: Nur wenn der Punkt Konvex (Blau) ist, dann wird geprüft, ob er im Rechteck/Polygon liegt.

Zeile 89: Da das ein konkav polygon ist, kann ich vorzeitig die Schleife wegen SAT beenden, wenn Edge.P1 außerhalb von nur einer Seite liegt.

```

71 //Schritt 2: Prüfe per SAT, ob der Edge.P1-Punkt innerhalb des Polygons liegt.
72 //Mach das aber nur, wenn er eine nach außen zeigende Spitze ist.
73 if (edge.IsP1Convex)
74 {
75     bool p1IsInsideFromRectangle = true;
76     int minFaceIndex = -1;
77     float maxFaceDistance = float.MinValue;
78
79     for (int i = 0; i < polygon.Vertex.Length; i++)
80     {
81         Vec2D corner = polygon.Vertex[i];
82         Vec2D p1ToCorner = edge.P1 - corner;
83
84         //Prüfe für den P1-Edgepunkt, ob er innerhalb des Polygons liegt
85         float faceDistance = p1ToCorner * polygon.FaceNormal[i];
86         if (faceDistance > 0)
87         {
88             p1IsInsideFromRectangle = false;
89             break; //Abbruch da es kein Schnittpunkt geben kann
90         }
91         else
92         {
93             if (faceDistance > maxFaceDistance)
94             {
95                 maxFaceDistance = faceDistance;
96                 minFaceIndex = i;
97             }
98         }
99     }
100
101    if (p1IsInsideFromRectangle)
102    {
103        collisions.Add(new CollisionInfo(edge.P1, -polygon.FaceNormal[minFaceIndex], -maxFaceDistance, (byte)minFaceIndex, 0));
104    }
}

```

Schnittpunkt bei einem konkaven Rigid-Polygon

Ein konkav polygon wird in konvexe Polygone zerlegt. Der Schnittpunkt mit einem konkaven Polygon mit einem anderen Objekt erfolgt dadurch, indem für jedes konvexe Unterpolygon die Schnittpunkte ermittelt werden.

Schnittpunkt konvexas Polygon mit Kreis

Ich finde zwei Quellen für die Polygon-Kreis-Kollision:

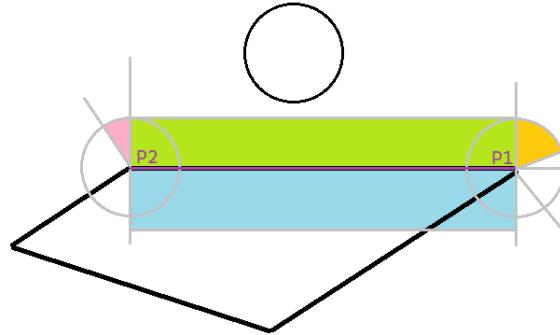
https://github.com/erincatto/box2d/blob/main/src/collision/b2_collide_circle.cpp#L55

<https://github.com/tutsplus/ImpulseEngine/blob/master/Collision.cpp#L68>

Beide sind gleich implementiert und beide sind sehr ähnlich wie die Kreis-Rechteck-Kollision.

Wenn man prüfen will, ob der schwarze Kreis innerhalb vom schwarzen Polygon liegt, dann ermittle ich zuerst, zu welcher Polygonseite das Kreiszentrum den kleinsten Abstand hat. Sollte bei der

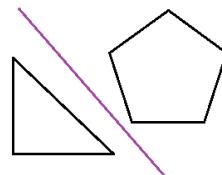
Prüfung schon rauskommen, dass der Kreisabstand größer als dessen Radius ist, dann gibt es keine Kollision. In diesen Beispiel wurde für die obere Polygonkante z.B. ermittelt, dass das Kreiszentrum zu dieser Kante den kleinsten Abstand hat. Nun gibt es 4 Bereiche, wo das Kreiszentrum sein kann. Ist es im blauen Bereich, dann wird er nach oben gedrückt. Liegt es über der P1-P2-Linie und rechts neben der Linien-Normale im orangenen Bereich, dann ergibt sich die Kollisionsnormale aus dem Vektor von P1 zum Kreiszentrums. Liegt es links neben dem P2-Punkt, dann zeigt die Normale von P2 zum Kreiszentrums. Ansonsten muss es im grünen Bereich liegen und es wird in Richtung P1-P2-Linien-Normale gedrückt.



Siehe: PhysicEngine/CollisionDetection/NearPhase/Polygon/ConvexPolygonOtherCollision.cs

Schnittpunkt konvexes Polygon mit Rechteck / anderen konvexen Polygon

Wenn es eine Linie gibt, die den Raum so in zwei Teile unterteilt, dass das eine Objekt auf der einen Seite liegt und das andere auf der anderen, dann kollidieren die Objekte nicht (Separating Axis Theorem). Beispiel: Weil ich die Möglichkeit habe die Lila-Linie so zu platzieren, dass beide Objekte die Linie nicht schneiden und auf unterschiedlichen Seiten liegen bedeutet dass, die Objekte kollidieren nicht:



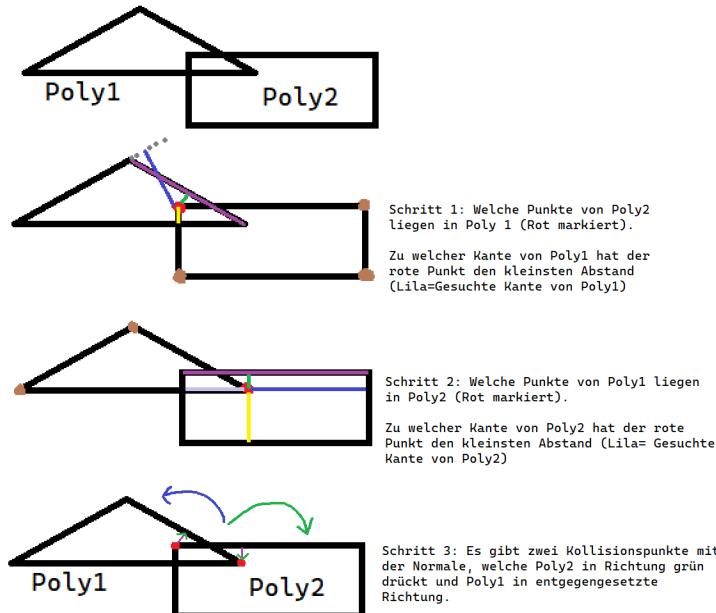
Wenn ich vor der Aufgabe stehe, dass ich einfach nur Wissen will, ob denn zwei konvexe Polygone kollidieren, dann kann ich für jede Polygonkante von beiden Objekten so eine Prüflinie erzeugen und dann schauen, ob sie eine SAT-Linie ist. Wenn ja, kann ich die Suche abbrechen und weiß nun, dass die Objekte nicht kollidieren.

Für unsere Physikengine nutzt uns die Information, ob zwei Objekte einfach nur kollidieren aber wenig. Wir brauchen Kollisionspunkte welche Kollisionsnormalen haben. An dieser Stelle kann ich dann Kräfte wirken lassen, welche in Richtung Kollisionsnormale die Objekte auseinander drücken.

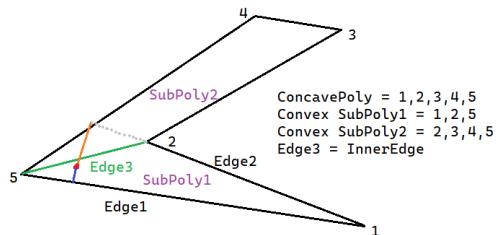
Um zu prüfen, ob zwei Objekte kollidieren und wenn ja, was ihre Kollisionspunkte sind, nutze ich folgenden Grundgedanke:

Ein Punkt liegt innerhalb von ein konvexen Objekt, wenn er hinter allen Polygonkanten liegt. Ein Kollisionspunkt soll immer nur auf einer Polygonecke liegen.

Wenn ich diese Idee in ein Algorithmus zur Kollisionspunkttermittlung umwandle, dann bekomme ich folgenden Ablauf: Erst schaue ich, welche Punkte von Poly2 sich innerhalb von Poly1 liegen. Außerdem ermittle ich die Kante von Poly1, wo all die Poly2-Punkte die kleinste Eindringtiefe haben. In diesen Beispiel hat die linke obere Rechteckecke den kleinsten Abstand zur rechten oberen Dreiecksseite. Also ist dieser Punkt ein Kollisionspunkt mit der Normale von der Dreiecksseite. Das gleiche muss ich auch in die andere Richtung machen da es sein kann, dass Eckpunkte von Poly1 sich in Poly2 befinden.



Wenn ich die Kollisionspunkte zwischen zwei konkaven Polygonen ermitteln will, dann muss ich zuerst beide Polygone in konvexe Teilstücke zerlegen. In diesem Beispiel habe ich ein konkav polygon mit 5 Ecken gegeben, was in 2 konvexe Teilstücke zerlegt wurde. Nun habe ich festgestellt, dass der rote Punkt sich innerhalb von SubPoly1 befindet.



Der rote Punkt hat die kürzeste Distanz zu Edge3. Würde ich nun ein Kollisionspunkt an der Stelle vom roten Punkt erzeugen und die Edge3-Normale als Kollisionsnormale nehmen, dann würde der rote Punkt Richtung grüner Linie gedrückt werden und dort dann kleben bleiben. Der Punkt soll aber aus dem konkavem Polygon raus gedrückt werden. Damit das geht, darf man innen liegende Kanten (grün), welche durch Polygonzerlegung entstehen, nicht mit in Betracht ziehen, wenn man wissen will, zu welcher Polygonkante ein Punkt den kleinsten Abstand hat. In diesem Beispiel hat Edge1 den kleineren Abstand (Blau markiert) als Edge2 (Orange). Also ist die Normale von Edge1 die gesuchte Kollisionsnormale.

So sieht die Umsetzung aus: Zuerst ermittle ich die Kollisionspunkte aus Poly2 und dessen außen liegende Kante von Poly1 mit kleinsten Abstand laut den beiden Erklärbildern:

```

7  private static CollisionInfo[] GetAllPointsFromPoly2WhichAreInsideInPoly1(IConvexPolygon poly1, IConvexPolygon poly2, bool swap)
8  {
9      List<int> p2Indizes = new List<int>(); //Liste all der Poly2-Punkte, welcher innerhalb von Poly1 liegen
10     for (int i = 0; i < poly2.Vertex.Length; i++)
11         p2Indizes.Add(i);
12
13     float maxFaceDistance = float.MinValue;
14     int minFaceIndex = -1;
15
16     for (int i = 0; i < poly1.Vertex.Length; i++)
17     {
18         Vec2D faceNormal = poly1.FaceNormal[i];
19         Vec2D faceP1 = poly1.Vertex[i];
20
21         List<int> p2PointsNotInPoly1 = new List<int>();
22         float minDistance = float.MaxValue; //Wie weit ist der entfernteste Punkt von (Poly1-Kante i) weg?
23         foreach (int j in p2Indizes)
24         {
25             float faceDistance = (poly2.Vertex[j] - faceP1) * faceNormal; //faceDistance ist negativ, wenn der Poly2-Punkt im Poly1
26
27             if (faceDistance > 0) //Punkt j liegt außerhalb der Poly1-Kante i?
28             {
29                 p2PointsNotInPoly1.Add(j); //Wenn ja, dann entferne ihn aus p2Indizes.
30             }
31             else
32             {
33                 if (faceDistance < minDistance) //Liegt Punkt j am Meisten innerhalb der aktuellen Poly1-Seite i?
34                 {
35                     minDistance = faceDistance;
36                 }
37             }
38         }
39         foreach (int j in p2PointsNotInPoly1)
40             p2Indizes.Remove(j);
41
42         if (p2Indizes.Any() == false) return new CollisionInfo[0]; //Kein Punkt von Poly2 liegt in Poly1
43
44         //Bei welche Face-Seite von Poly1 dringen die Punkte am wenigsten ein?
45         //Nimm nur dann eine Polygonkante als potenzielle Wegstöcke, wenn sie eine OutsideEdge ist
46         bool faceIsInnerFace = poly1 is IConvexSubPolygon && (poly1 as IConvexSubPolygon).IsOutsideEdge[i] == false;
47         if (faceIsInnerFace == false && minDistance > maxFaceDistance)
48         {
49             maxFaceDistance = minDistance;
50             minFaceIndex = i;
51         }
52     }
}

```

Siehe: PhysicEngine/CollisionDetection/NearPhase/Polygon/PolygonPolygonCollision.cs

Diese Funktion rufe ich für beide Richtungen auf. Da ich aber am Ende will, dass alle Kollisionsnormalen von allen Punkten von Poly1 in Richtung Poly2 zeigen, muss ich die Normalen der Kollisionspunkte von Zeile 95 noch umdrehen. Das mache ich mit den Swap-Schalter (3. Parameter)

```

83     internal static CollisionInfo[] PolygonPolygon(ICollidable polygon1, ICollidable polygon2)
84     {
85         IConvexPolygon poly1 = (IConvexPolygon)polygon1;
86         IConvexPolygon poly2 = (IConvexPolygon)polygon2;
87
88         List<CollisionInfo> contacts = new List<CollisionInfo>();
89
90         //Es ist möglich, dass Poly2-Punkte in Poly1 sich aufhalten aber in Poly2 befindet sich kein Poly1-Punkt
91         //Deswegen ist es kein Abbruchgrund, wenn p2Points oder p1Points keine Punkte enthält ist.
92         var p2Points = GetAllPointsFromPoly2WhichAreInsideInPoly1(poly1, poly2, false);
93         contacts.AddRange(p2Points);
94
95         var p1Points = GetAllPointsFromPoly2WhichAreInsideInPoly1(poly2, poly1, true);
96         contacts.AddRange(p1Points);
97
98         return contacts.ToArray();
99     }
}

```

So sieht der zweite Teil von der GetAllPointsFromPoly2WhichAreInsideInPoly1-Funktion aus. Hier sieht man wie mit Swap die Normale gedreht wird:

```

56     //Die Punkte von Poly2, welcher innerhalb von Poly1 liegen haben die kleinste Eindringtiefe bei Poly1-Seite minFaceIndex
57     Vec2D faceNormal = poly1.FaceNormal[minFaceIndex]; //In diese Richtung werden all die gefundenen Poly2-Punkte gedrückt
58     Vec2D faceP1 = poly1.Vertex[minFaceIndex];
59
60     List<CollisionInfo> collisions = new List<CollisionInfo>();
61     if (swap == false) //Normale zeigt von Poly1 so, dass Poly2 in diese Richtung gedrückt werden soll
62     {
63         foreach (int i in p2Indizes)
64         {
65             Vec2D poly2Point = poly2.Vertex[i];
66             float depth = (faceP1 - poly2Point) * faceNormal; //Positive Number
67             collisions.Add(new CollisionInfo(poly2Point, faceNormal, depth, (byte)(poly1.Vertex.Length + minFaceIndex), (byte)i));
68         }
69     }
70     else //Normale zeigt von Poly2 nach Poly1. Also muss sie hier noch getauscht werden
71     {
72         foreach (int i in p2Indizes)
73         {
74             Vec2D poly2Point = poly2.Vertex[i];
75             float depth = (faceP1 - poly2Point) * faceNormal; //Positive Number
76             collisions.Add(new CollisionInfo(poly2Point + faceNormal * depth, -faceNormal, depth, (byte)(poly1.Vertex.Length + minFa
77         }
78     }
79
80     return collisions.ToArray();
}

```

Wenn ich die Polygon-Kollisionsfunktion von Erin mir ansehe, dann stelle ich fest, dass er Kollisionspunkte dadurch erzeugt, indem er Kanten des einen Polygons mit Kanten des anderen Polygons schneidet und der Schnittpunkt ist dann ein potenzieller Kollisionspunkt. Warum er das so kompliziert macht weiß ich aber nicht. Es könnte sein, dass er Dinge weiß, die über unser normales Verständnis hinaus gehen^^

https://github.com/erincatto/box2d/blob/main/src/collision/b2_collide_polygon.cpp

Broad-Phase Kollisionserkennung

Wenn ich nach Broad-Phase-Algorithmen suche, dann finde ich:

- Sort & Sweep (mit InsertionSort oder QuickSort)
- QuadTree/BSPTree für statische Objekte; CircleTree für dynamische Objekte
- kD-Tree
- Grid
- Dynamic Bounding Volume Tree(Dbct)
 - Bsp: https://github.com/erincatto/box2d/blob/main/src/collision/b2_dynamic_tree.cpp
- <https://github.com/bulletphysics/bullet3/tree/master/src/Bullet3Collision/BroadPhaseCollision>

Es gibt eine Quelle die sehr oft genannt wird: Real-Time Collision Detection von Christer Ericson.

Wenn man den Umstand ausnutzen will, dass die Objekte zwischen zwei Frames ungefähr die gleiche Position haben, dann muss man die Objekte sortieren. Das kann entweder eine flache Liste wie bei Sort & Sweep (oder auch Sweep and Prune genannt) sein oder ein dynamischer Baum, wie es Erin bei Box2D macht.

Um erst mal ein einfachen Einstieg in das Thema zu bekommen, möchte ich eine ShouldCollide-Paare-Liste erzeugen, welche immer nur dann aktualisiert werden muss, wenn Objekte hinzukommen oder entfernt werden. So kann ich verhindern, der Levelrand von einer Szene gegen sich selbst getestet wird. Diese Paare-Liste ist dann der Input für den NearPhase-Test.

Um die Broadphase bei Box2D oder BulletPhysics verstehen zu können muss ich wohl erst das Buch von Ericson mir ansehen. Es scheint, dass das ein eigenes Kapitel für sich ist und für den Anfang hoffe ich, dass erst mal die ShouldCollide-Liste reicht.

So sieht nun die Kollisionsabfrage aus. Der CollisionPairManager erstellt eine Liste von Objekte-Paaren, für die ein Kollisionstest erfolgen soll. Diese Paar-Liste ist fix über alle Frames und ändert sich nur, wenn Objekte hinzukommen oder entfernt werden.

```
54 //Das ist der BroadPhase-Filter. All diese ICollidable-Eigenschaften ändern sich beim Objekt über seine Laufzeit nicht.  
55 // Verweis  
56 private static bool ShouldCollide(ICollidable b1, ICollidable b2, bool[,] collisionMatrix)  
57 {  
58     if (b1.IsNotMoveable && b2.IsNotMoveable) return false;  
59     if (collisionMatrix[b1.CollisionCategory, b2.CollisionCategory] == false) return false;  
60     if (b1.CollideExcludeList.Contains(b2)) return false;  
61     return true;  
62 }
```

Der CollisionManager ermittelt alle Kollisionspunkte indem er die vor gefilterte Liste nutzt, dann den BoundingCircleTest für die Broadphase und danach die Kontaktpunkte mit der Nearphase-Funktion ermittelt.

```
20 // Verweis  
21 public RigidBodyCollision[] GetAllCollisions()  
22 {  
23     List<RigidBodyCollision> collisions = new List<RigidBodyCollision>();  
24     foreach (var pair in this.pairManager.GetPairs()) //BroadPhase-Filter: Gib nur die Paare zurück, die laut CollisionMatrix kollidieren könne  
25     {  
26         var b1 = pair.C1;  
27         var b2 = pair.C2;  
28  
29         if (BoundingCircleTest.Collide(b1, b2)) //Broadphase-Test  
30         {  
31             var contacts = GetCollisions(b1, b2); //Nearphase-Test  
32             if (contacts.Any())  
33                 collisions.AddRange(contacts.Select(x => new RigidBodyCollision(x, b1, b2, pair.Index1, pair.Index2)));  
34         }  
35     }  
36  
37     return collisions.ToArray();  
38 }
```

Zusammenfassung von Teil 6

Die Physikengine wurde um die Funktionen erweitert, welche noch nötig waren, um ein Level bauen zu können.

Weitere Ideen für das RigidPhysics-Projekt:

- Polygone mit Löchern (Merged-Objekt aus OutsidePolygon wo drinnen ein InsidePolygon ist)
- Merged-Objekte, um somit WeldJoints zu sparen
- Broadphase verbessern indem ich das Buch von Christer Ericson mir ansehe
- Kollisionserkennung um Tunneling erweitern

In dieser Dokumentation haben die Teile 1 bis 6 den Kern der Physikengine besprochen. In den nächsten 2 Abschnitten werden jetzt noch Editoren entwickelt, welche die Physiksimulation um Texturen, Animationen und Sprits erweitern.

Teil 7: Leveleditor

7.1 Die Teilkomponenten des Editors

Ziel dieses Abschnitts

In den Abschnitte 1 bis 6 wurde der Kern der Physikengine (siehe PhysicEngine/Engine/02_RigidBodyPhysics) erklärt und dabei gab es zu jeden Abschnitt ein eigenes Projekt innerhalb des Dokumentationsordners. Ab jetzt geht es darum das Hauptprojekt zu erklären. In diesen Abschnitt soll der Projektunterordner Engine/05_Leveleditor erklärt werden. Das Vorgehen ist so, dass hier anhand von Beispielen erklärt wird, wie der Leveleditor benutzt wird. Es wird auch kurz erklärt, wie er grob vom internen Aufbau her funktioniert aber diese grobe Einführung soll eher dazu dienen, dass man dann selber sich anschaut, wie der Leveleditor intern arbeitet. Dieser Wechsel im Erklärungsstil wurde deswegen gemacht, da der Hauptfokus von diesen Dokument auf den Physikformeln liegen soll und es dem Leser eher ermöglichen soll, mit diesen Projekt zu arbeiten und nicht schrittweise zu dokumentieren, wie der Leveleditor erstellt wurde.

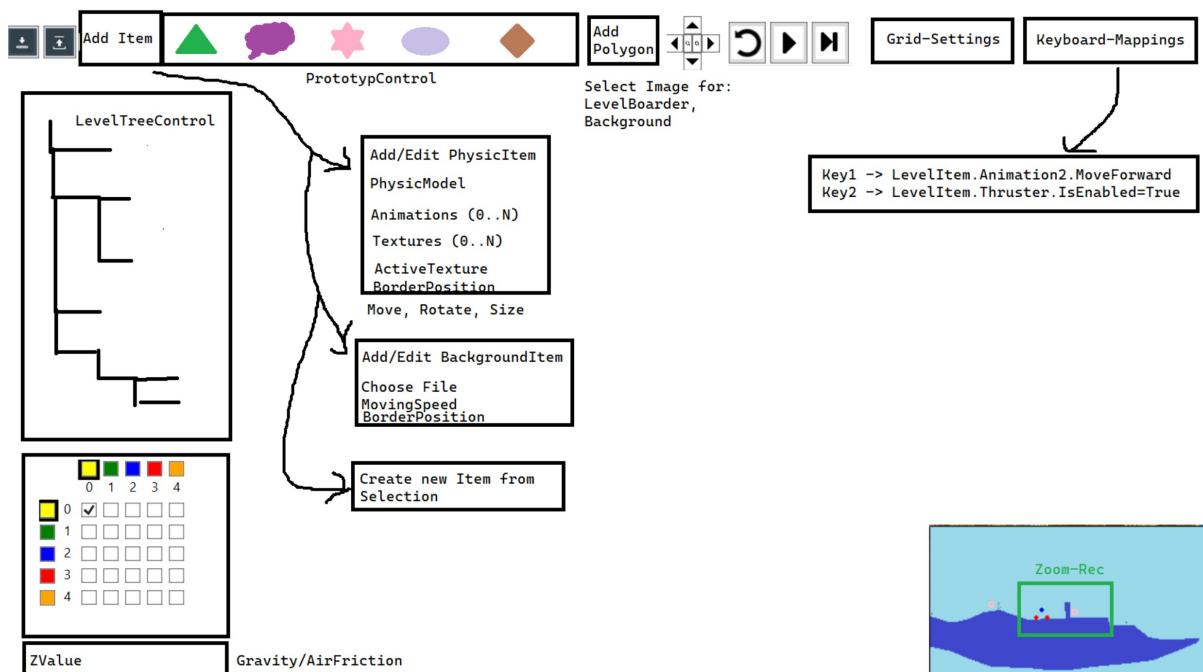
Konzept für den Leveleditor

Es gibt zwar bereits ein Editor, um ein PhysicScene-Objekt zu erzeugen und auch zu testen aber die Szenen, die mit diesen Editor erstellt werden haben noch keine Texturen und man kann auch nicht per Tastendruck mehrere Gelenke gleichzeitig und kontrolliert ansteuern. Außerdem kann ich ein erstelltes Fahrzeug/Person dann nicht verschieben/skalieren. Wenn ich eine Kopie von einem Körper erstelle und dann was ändern will, so muss ich dann alle Kopien einzeln anpassen.

Mit dem Leveleditor soll man zuerst ein PhysicItem erzeugen können, was eine texturierte PhysicScene ist, wo die Gelenke dann über eine KeyFrame-Animation gesteuert werden können. Dieses PhysicItem kann man dann in einer Szene platzieren. Wenn ich mehrere Kopien von diesem Item erstelle, dann soll eine Änderung im PhysicItem sich dann auf alle Kopien auswirken.

Die Platzierung soll entweder frei oder per Raster gehen. Es soll Hintergrundobjekte geben, wo der Z-Abstand definiert wird und der Levelrand wird über Polygone erzeugt.

Um all das umzusetzen sieht das Konzept des Leveleditors so aus:



Oben Links gibt es das Prototyp-Control. Dort erzeuge ich ein Fahrzeug/Skifahrer den ich texturiere und animiere. Dieses Prototyp-Item wird dann im PrototypControl aufgelistet und von dort kann ich es dann per Drag & Drop ins Level ziehen um es dort dann zu platzieren/rotieren/skalieren. Links im Baum sehe ich dann welche LevelItems es alle gibt.

Der Editor zur Erstellung von ein Prototyp-Item besteht aus drei Teilen:

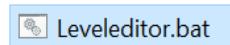
PhysicEditor = Das ist der Editor, den wir bis jetzt die ganze Zeit genutzt haben

TextureEditor = Hier kann ich Bilder an Starrkörper dran hängen

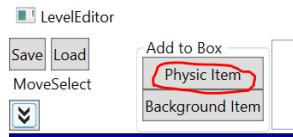
AnimationEditor = Festlegen der Gelenksollwerte, um eine Bewegung zu definieren

Abschnitt 7.1.1 behandelt den Textureditor und Abschnitt 7.1.2 den Animationseditor.

Um die Beschreibung in den Abschnitten 7.1.1 und 7.1.2 nachvollziehen zu können starte den Leveleditor unter Batch-Files:



Und klicke dann auf den „Physic Item“-Button:



Dort ist dann der PhysicEditor, TexturEditor und Animationseditor zu sehen.

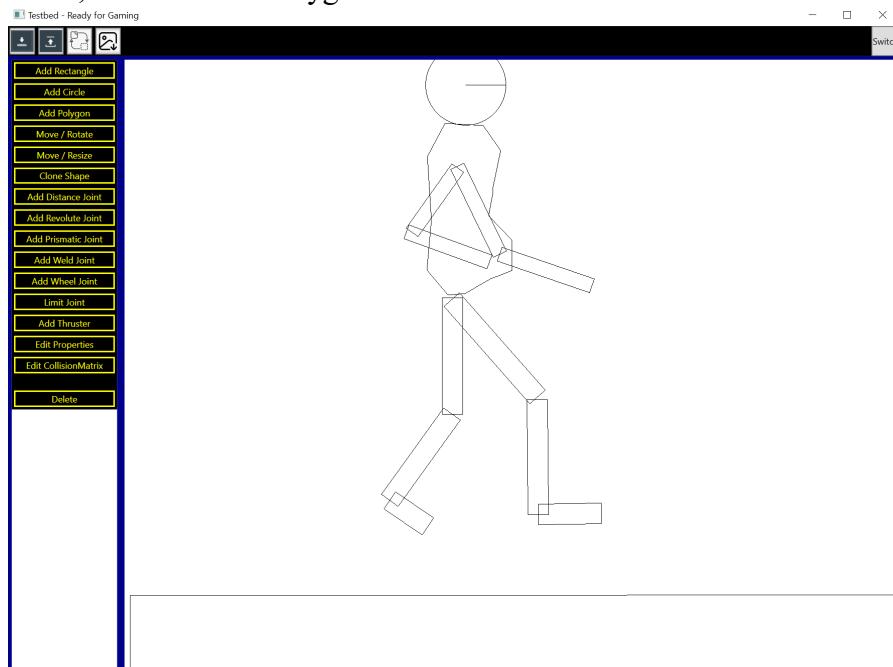
7.1.1 Texture Editor

Für was ist der Editor?

Die Physikengine erzeugt Rechtecke, Kreise und Polygone. Der Textur-Editor nutzt diese drei Formen als Input und zeichnet eine Textur über diese Objekte. Der Nutzer kann im Editor festlegen, an welcher Stelle die Textur an den Objekt befestigt ist und welche Ausrichtung/Größe sie hat.

Anwendungsbeispiel – Davefigur

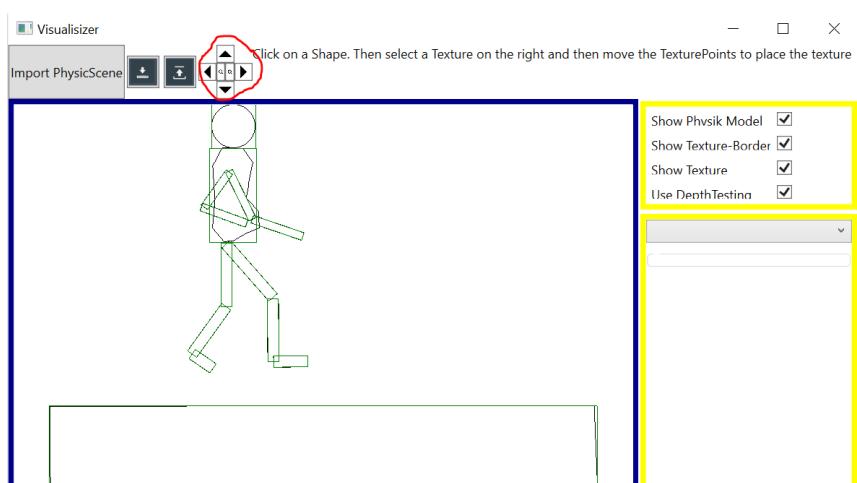
Es soll eine Figur texturiert werden. Dazu baue ich im Physik-Editor zuerst eine Figur, welche als Kopf einen Kreis hat, als Torso ein Polygon und die Gliedmaßen bestehen aus Rechteckecken.



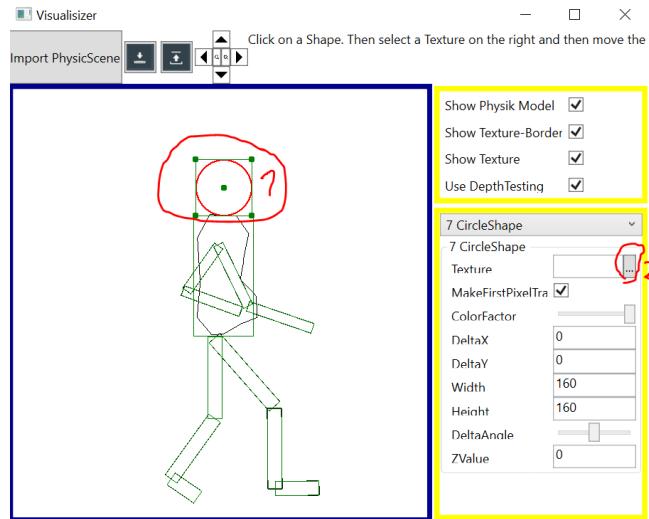
Dann klicke ich auf „Add Texture“ und gehe in den Textur-Editor:



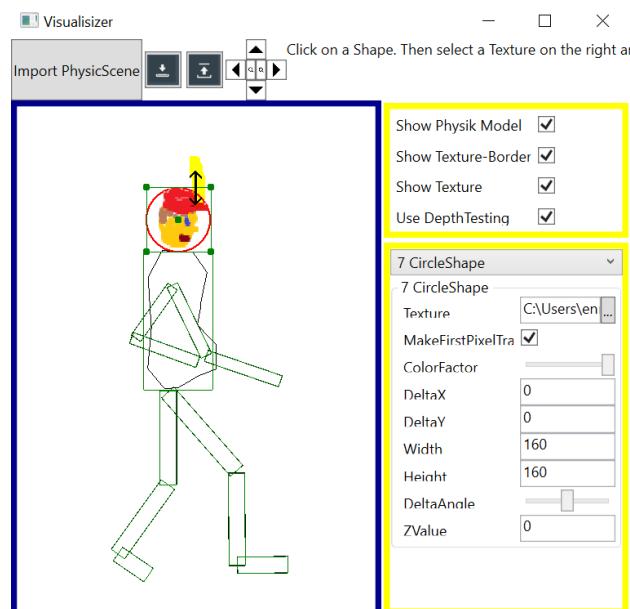
Dort kann ich die Kamera-Position einstellen wenn ich näher ran zoomen will:



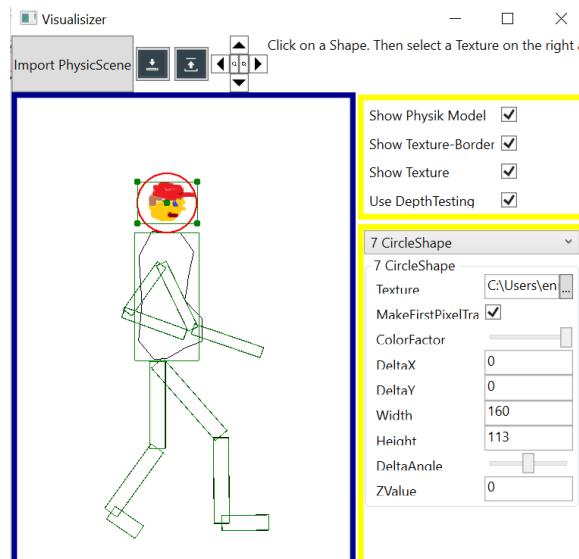
Nun klicke ich auf den Kopf der Figur und wähle eine Textur dafür aus:



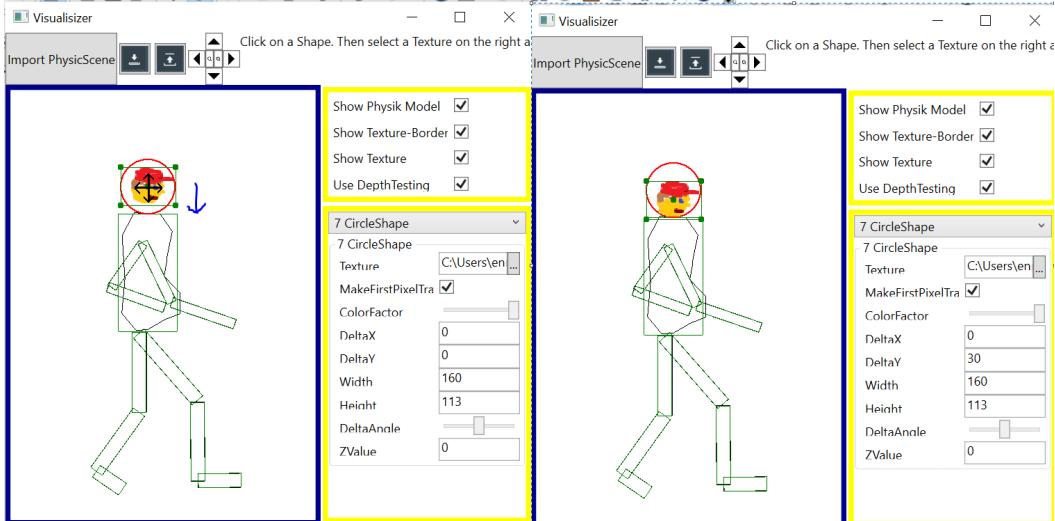
Die Textur sieht etwas verzerrt aus. Also gehe ich mit der Maus auf den oberen Rand vom Textur-Border und verkleinere die Höhe.



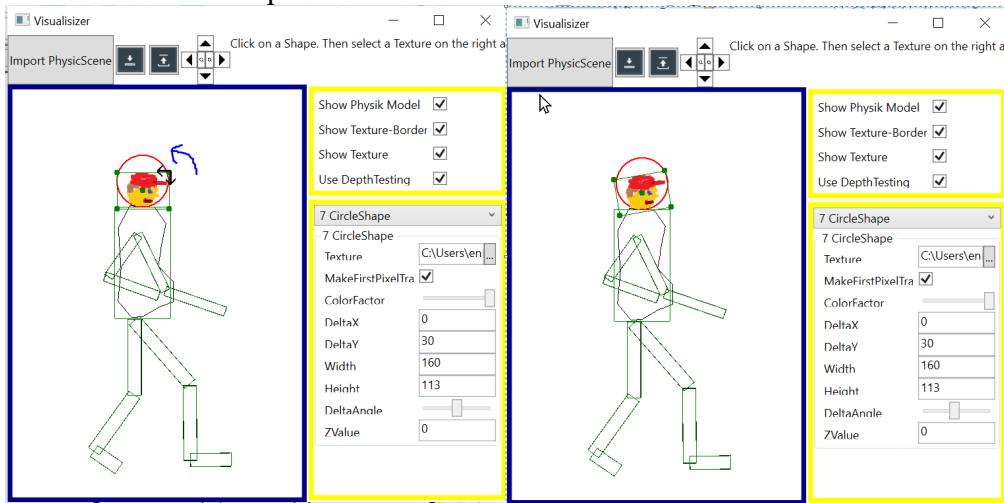
Jetzt ist der Kopf zwar nicht mehr so verzerrt aber er schwebt noch in der Luft:



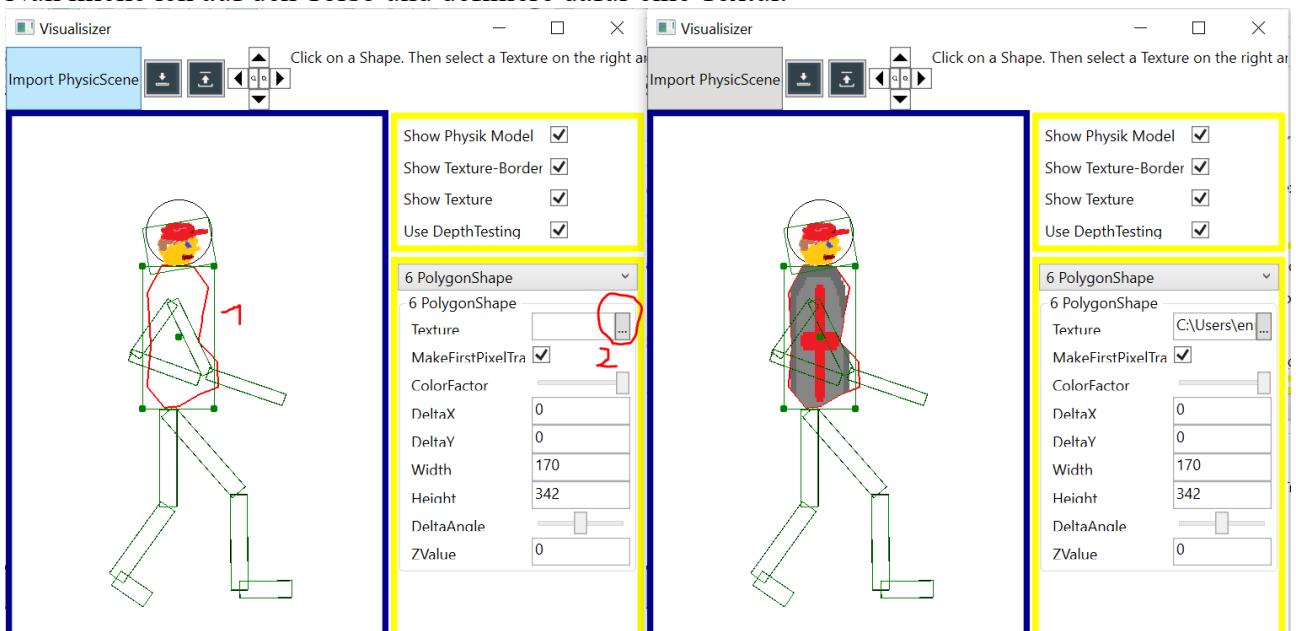
Ich gehe nun mit der Maus über die Mitte vom Textur-Rechteck und verschiebe die Textur nach unten:



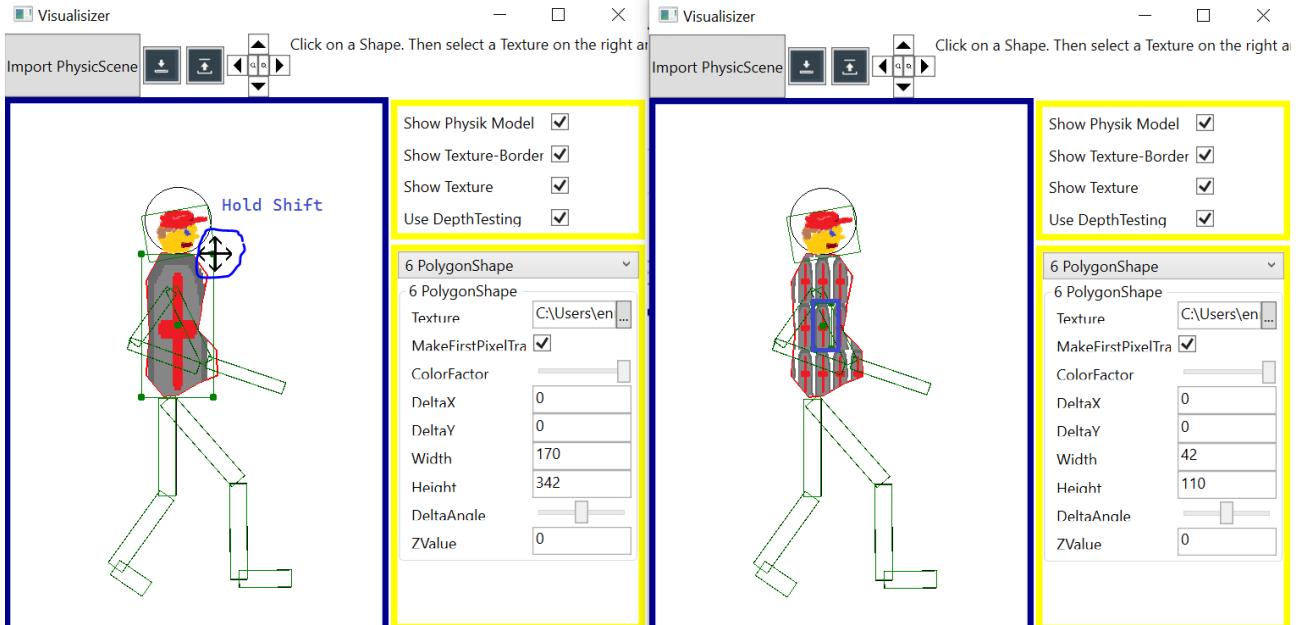
Jetzt sitzt der Kopf schon mal auf dem Torso aber er soll jetzt noch etwas mehr nach oben schauen. Dazu gehe ich auf eine der Eckpunkte vom Rechteck und drehe damit das Rechteck:



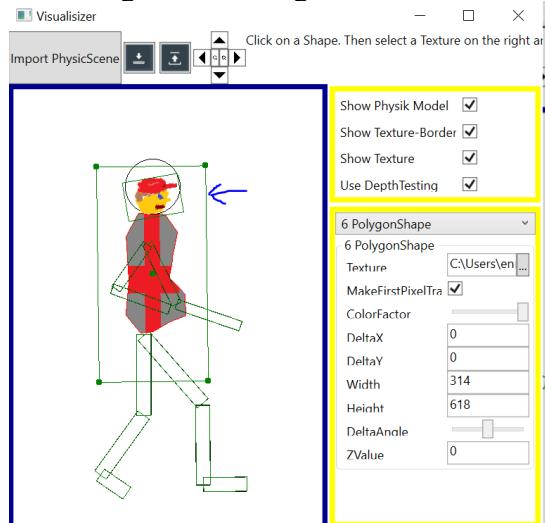
Nun klicke ich auf den Torso und definiere dafür eine Textur.



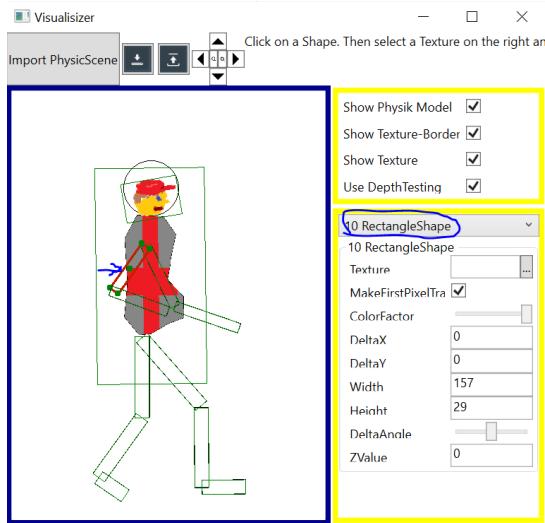
Bei ein Polygon kann ich die Textur kleiner und größer machen, indem ich Shift gedrückt halte und dann auf eine Textur-Ecke klicke und dann den Punkt verschieben. In diesen Bild wurde das Textur-Rechteck kleiner gemacht (blau markiert im rechten Bild). Dadurch kommt es dann zur Kachelung von der Textur:



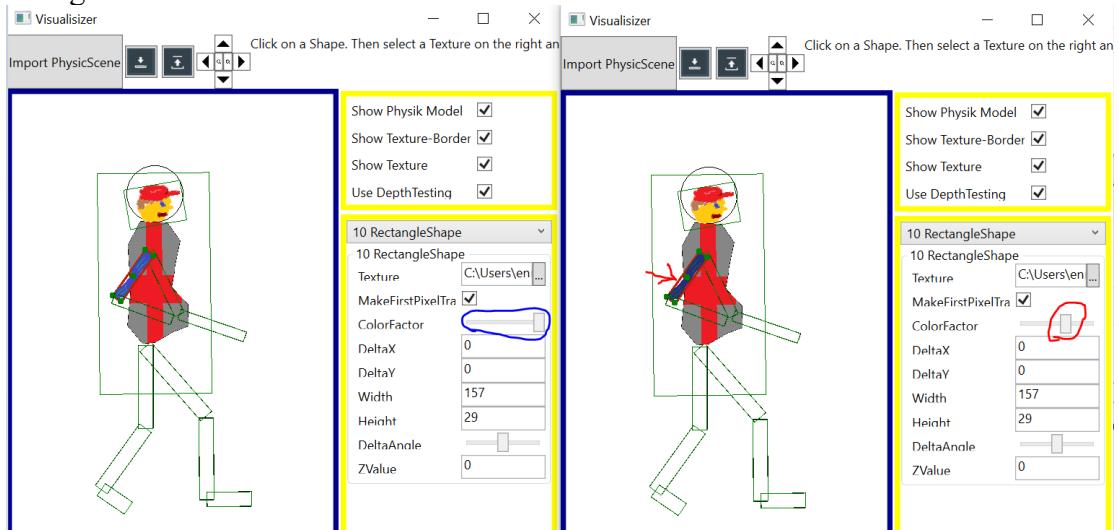
Wenn ich das Textur-Rechteck aber größer mache, dann kann ich nur ein bestimmten Teil der Textur anzeigen. Im Bild: Der blaue Pfeil zeigt auf das vergrößerte Texturrechteck.



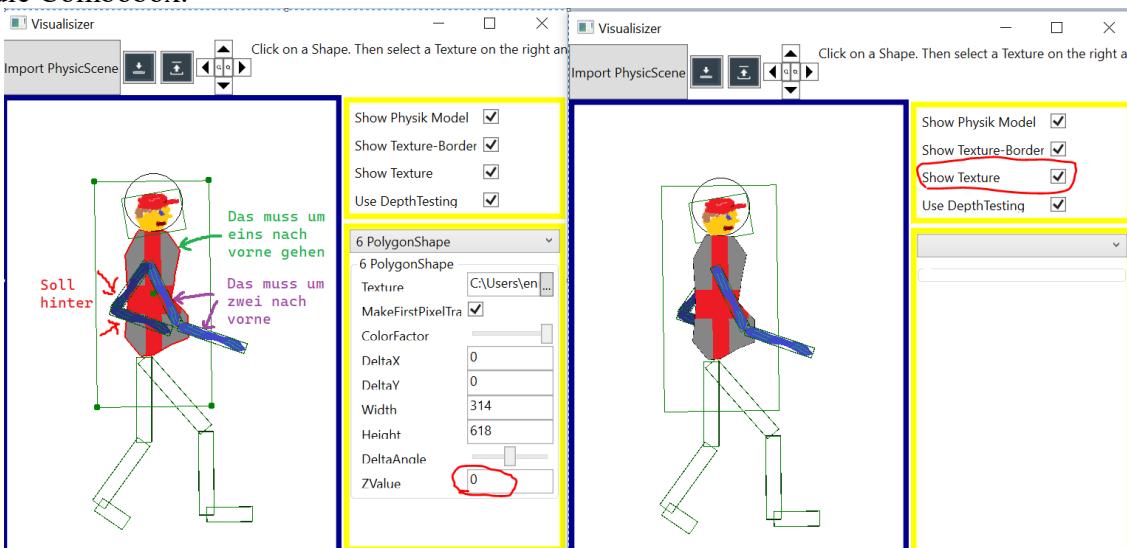
Als nächstes sollen die Arme texturiert werden. Wenn man mit der Maus auf den Oberarm klickt, dann befinden sich an der Stelle sowohl das Textur-Rechteck vom Arm als auch vom Torso. In diesen Fall wird das kleinere Objekt selektiert. Hier wurde der linke Oberarm selektiert. Man sieht auch in der Combobox (blau markiert), welches Objekt gerade selektiert wurde. Man kann auch über die Combobox die Selektion ändern.



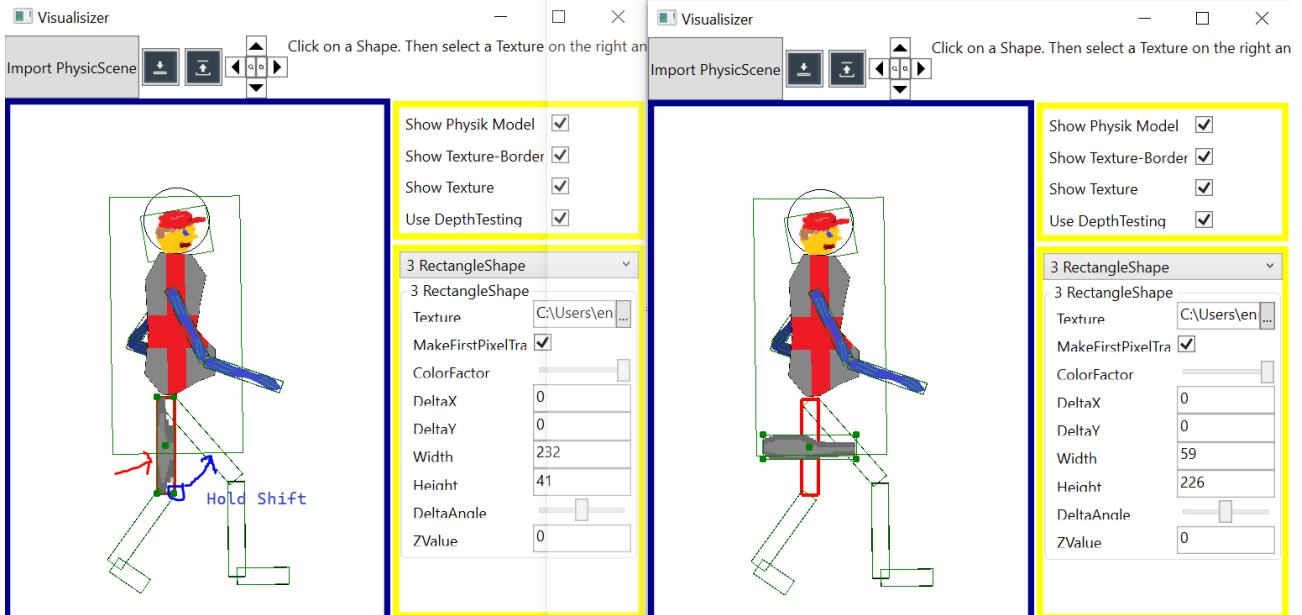
Wir ändern nun die Textur vom linken Oberarm. Da der Arm hinten liegt, soll er etwas dunkler sein. Dazu verringere ich etwas den ColorFactor:



Nun werden alle 4 Arm-Teile texturiert. Allerdings stellen wir nun fest, dass der Arm, der eigentlich hinten liegen sollte vorne angezeigt wird. Wir ändern nun den ZValue vom Torso auf 1 und vom rechten Arm auf 2. Wenn man den Torso zuerst nach vorne holt und danach dann erst den rechten Arm, dann verdeckt der Torso den Arm so, dass man ihn nicht mehr sieht. Ich kann ihn dadurch selektieren, indem ich kurz mal die Texturen ausblende (ShowTexture=false) oder ich selektiere über die Combobox.



Nun sollen noch die Beine Texturen bekommen. Wenn ich eine Textur dafür auswähle, dann sieht die erst mal komisch aus (linkes Bild). Mit gedrückter Shift-Taste ziehe ich nun den unteren rechten Textur-Punkt nach oben rechts und das Bild sieht auf einmal ok aus. Über „Flip90“ geht das auch.



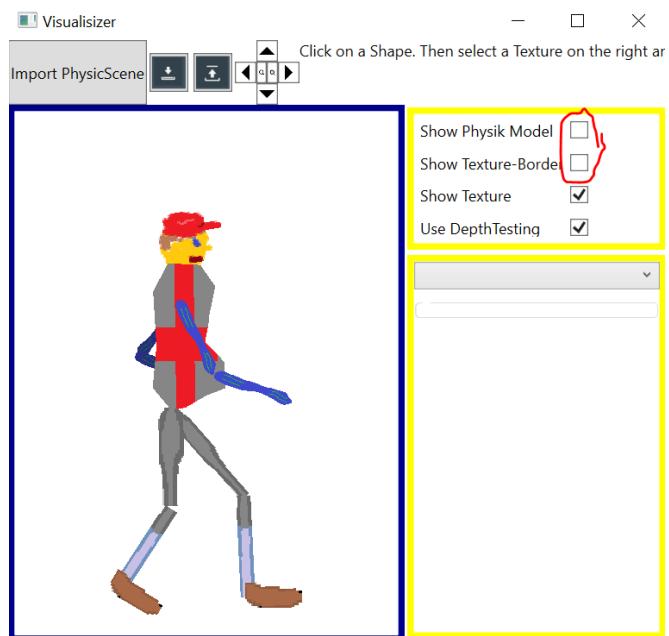
Der Grund für die Anfängliche Verzerrung ist, weil die Textur so aussieht:



Hätte man sie direkt gleich so gespeichert kann man sich diesen Korrekturschritt sparen:



Die Beine und Füße bekommen auch noch eine Textur. Ohne die Textur-Rechtecke sieht das Objekt nun so aus:



7.1.2 Animationseditor

Die Aufgabe vom Animationseditor ist es die Soll-Gelenkwerte von ein Physik-Objekt festzulegen. Damit wird eine Bewegung (von einer Person/Maschine) simuliert. Es gibt drei Arten wie eine Animation abgespielt werden kann:

OneTime: Animation wird einmal per Timer abgespielt. Beispiel: Person die einmal mit ein Stock schlägt.

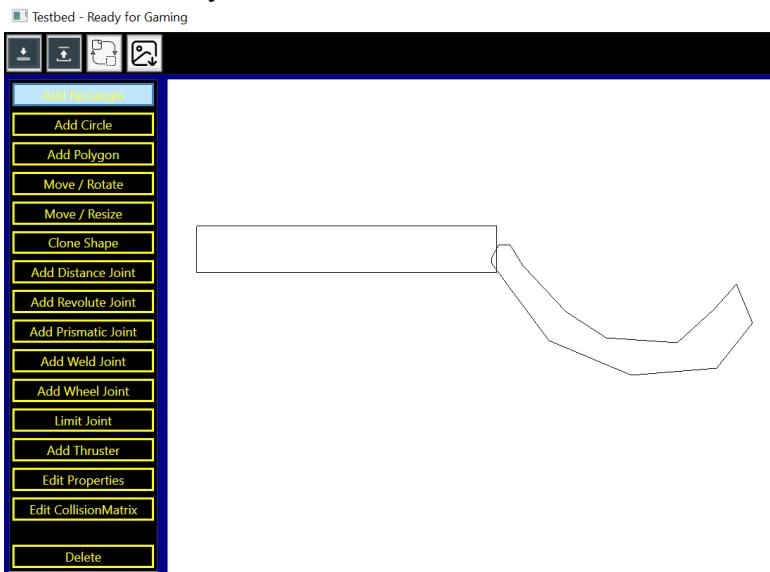
AutoLoop: Animation beginnt automatisch immer wieder von vorne. Beispiel: Person die so lange läuft, bis die Animation gestoppt wird

Manual: Animation läuft nur dann wenn die MoveForward oder MoveBackward-Taste gedrückt ist. Beispiel: Ski-Springer geht in die Hocke, wenn MoveForward gehalten wird. Er stoppt die Bewegung, wenn keine Taste gedrückt wird. Über Halten von MoveBackward steht er wieder auf.

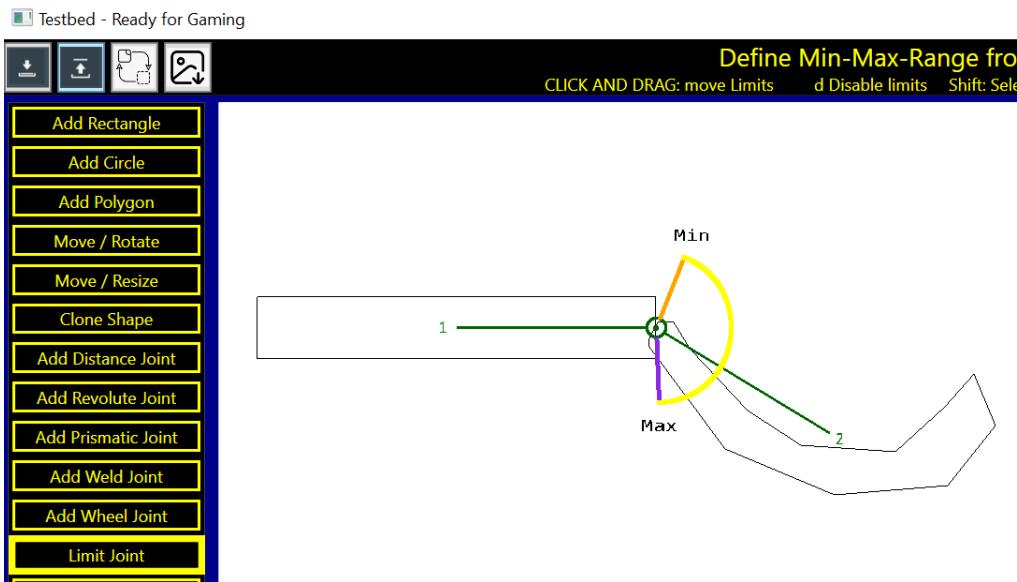
Beispiel 1: OneTime-Animation

So wird ein Hebelarm erstellt, der sich einmal bewegt.

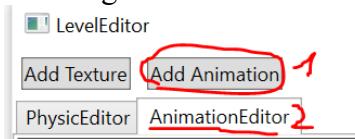
Schritt 1: Im PhysikEditor wird zuerst ein Rechteck und ein Polygon definiert:



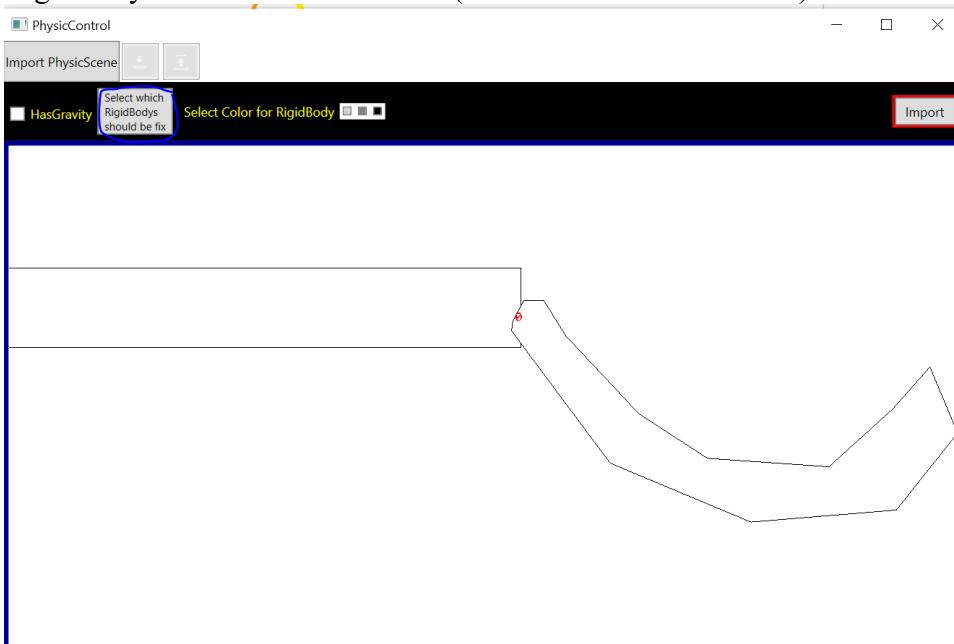
Schritt 2: Über ein Revolute Joint werden diese beiden Objekte verbunden. Der grüne Hebelarm2 darf sich nur noch in den gelben markierten Min-Max-Bereich aufhalten.



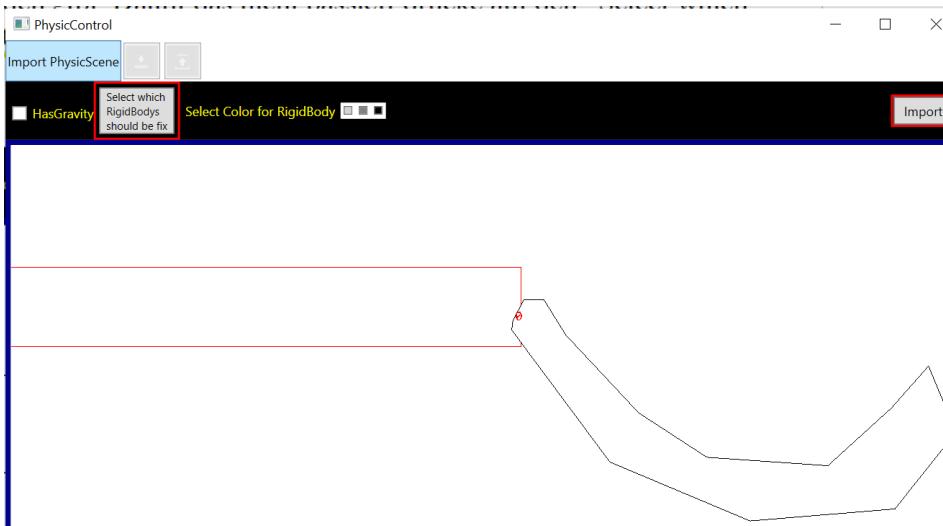
Schritt 3: Klicke auf „Add Animation“ und gehe dann in den AnimationEditor:



Schritt 4: Würde man jetzt die PhysicScene so direkt simulieren, so würde das Objekt einfach nur runter fallen, da es kein Fußboden gibt. Damit das nicht passiert drücke auf den „Select which RigidBodies should be fix“-Button (Im Bild hier blau markiert).

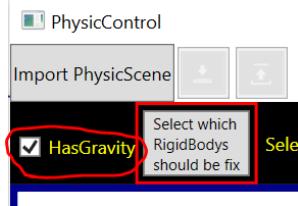


Klicke dann auf das Rechteck. Dadurch wird es rot. Ab jetzt hat dieses Rechteck eine unendliche Masse und kann sich nicht mehr bewegen. Damit trotzt es nun der Schwerkraft.



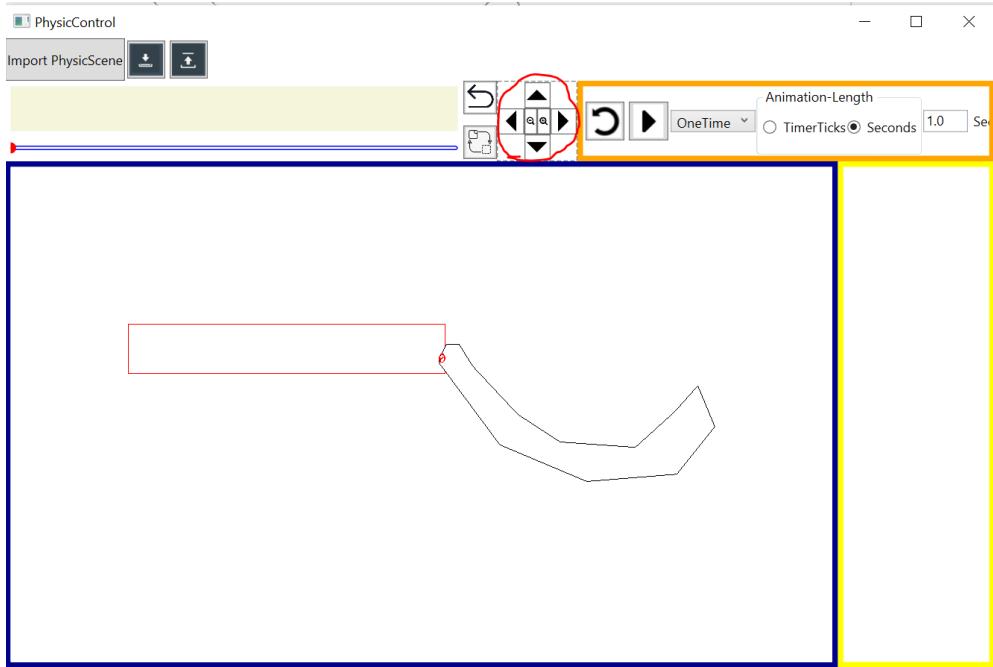
Rechteck hat roten Rand = Es hat eine unendliche Masse und bewegt sich nicht

Schritt 5: Wir wollen während der Physik-Simulation Schwerkraft haben. Also aktiviere sie im Importer-Fenster:

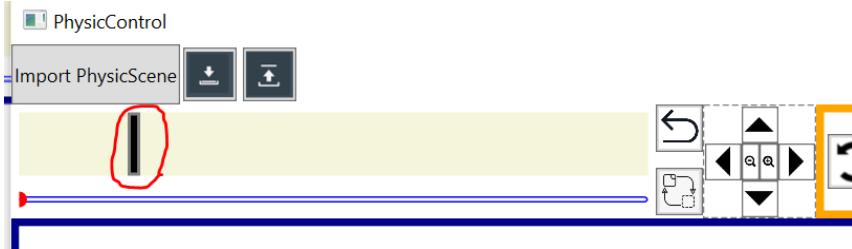


Schritt 6: Klicke auf Import.

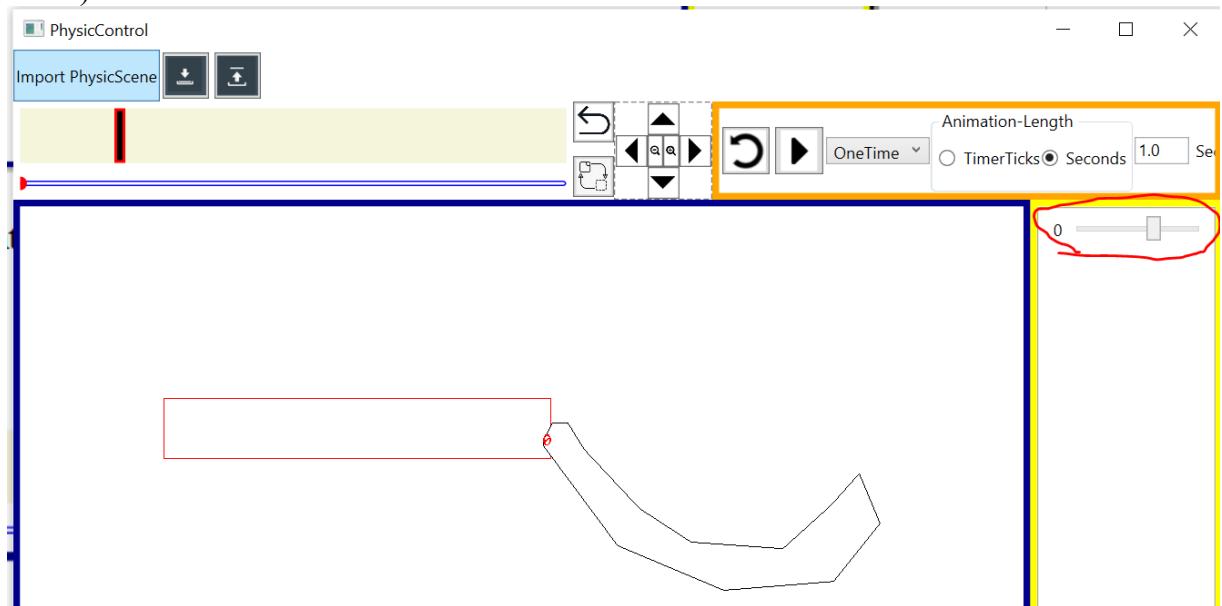
Schritt 7: Platziere nun die Kamera um das Objekt besser sehen zu können. Über die kleinen Buttons in der Mitte kann man rein- und raus zoomen.



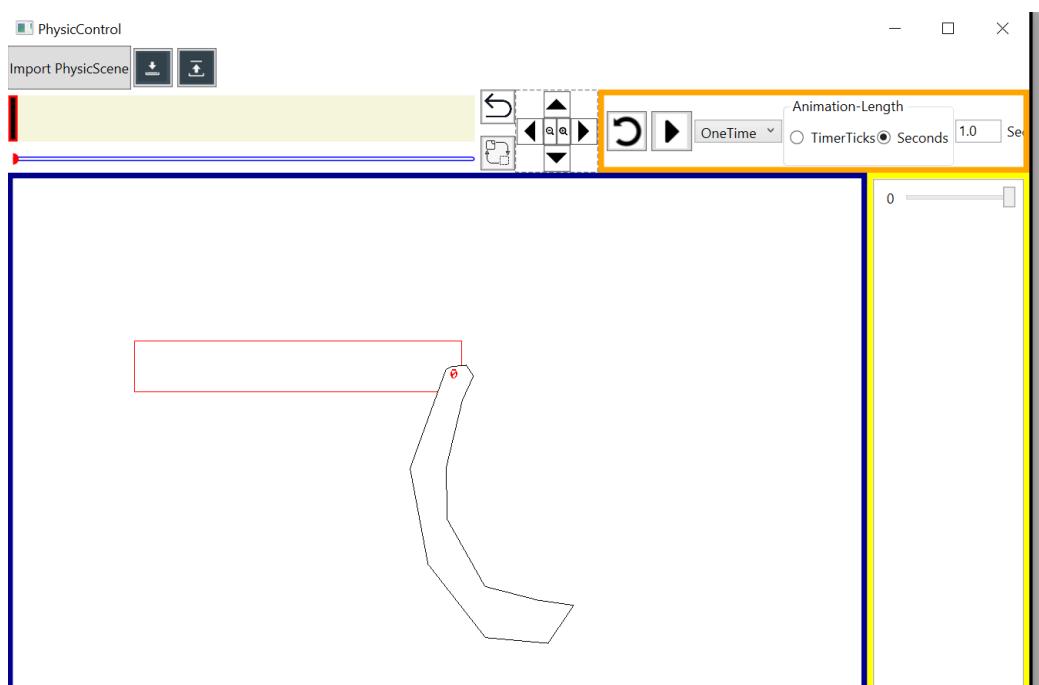
Schritt 8: Klicke nun per Rechtsklick in das KeyFrame-Control und erstelle damit einen neuen KeyFrame:



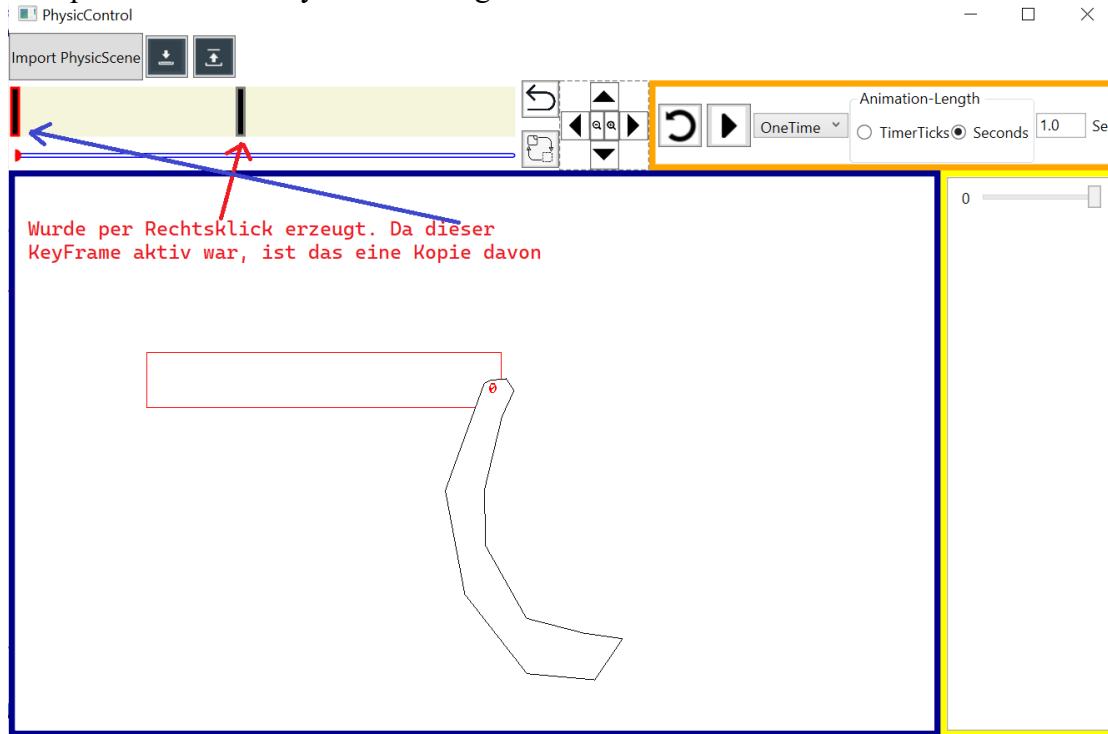
Schritt 9: Der KeyFrame wurde erstellt aber ist noch nicht aktiv. Um ihn bearbeiten zu können klicke mit der linken Maustaste drauf. Das er aktiv ist sieht man nun daran, dass er ein roter Rahmen hat. Außerdem werden im linken gelben Bereich nun alle Gelenke des Physik-Objektes angezeigt. Wir haben nur ein Revolute-Joint. Dessen Position können wir nun über den Slider (rot markiert) einstellen:



Schritt 10: Definiere den Soll-Wert von Gelenk Nummer 0 und verschiebe den KeyFrame per MouseDown nach links. Damit wird gesagt, dass die Animation zum Zeitpunkt 0 so starten soll, dass das Gelenk Nummer 0 den Wert 1 hat. D.h. Das Polygon hängt nun unten am Rechteck dran:

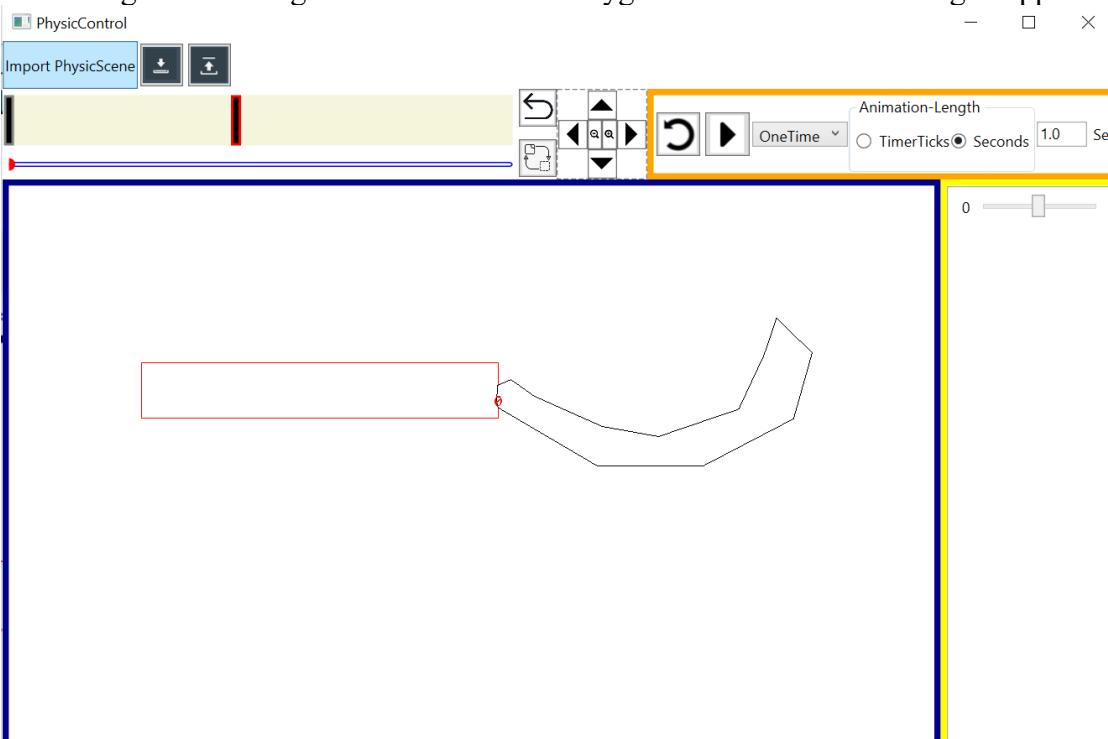


Schritt 11: Erzeuge nun ein neuen KeyFrame, indem wieder per Rechtsklick in das KeyFrame-Control geklickt wird. Der neu erstellte KeyFrame bezieht seine Werte direkt von den Soll-Werten, die gerade das Physik-Objekt hat. Weil gerade der ganz linke KeyFrame aktiv ist, bedeutet dass, es wird ein Kopie von diesen KeyFrame erzeugt:

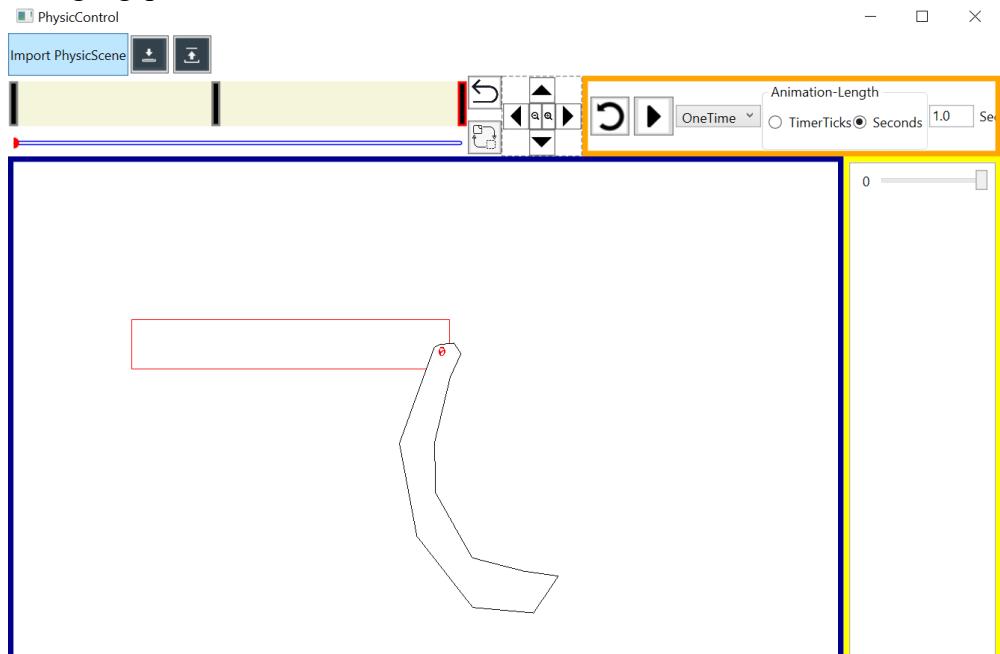


Der neu erstellte KeyFrame muss noch per Linksklick aktiviert werden. Erst dann werden dann im gelben Fenster rechts seine Soll-Gelenkwerte angezeigt.

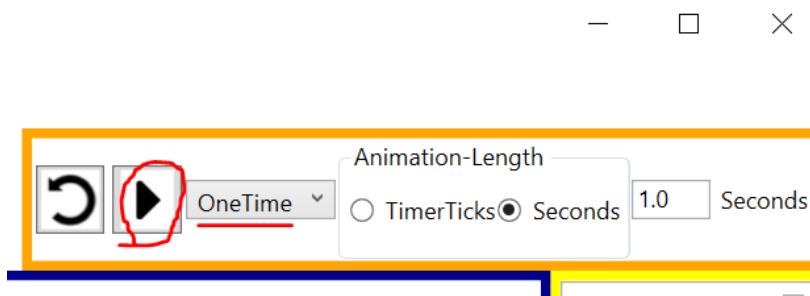
Der zweite KeyFrame hat hier ein roten Rahmen. Also ist er selektiert. Hier soll der Sollwert für Gelenk 0 bei ungefähr 0.5 liegen. Dadurch ist das Polygon nun etwas nach oben geklappt.



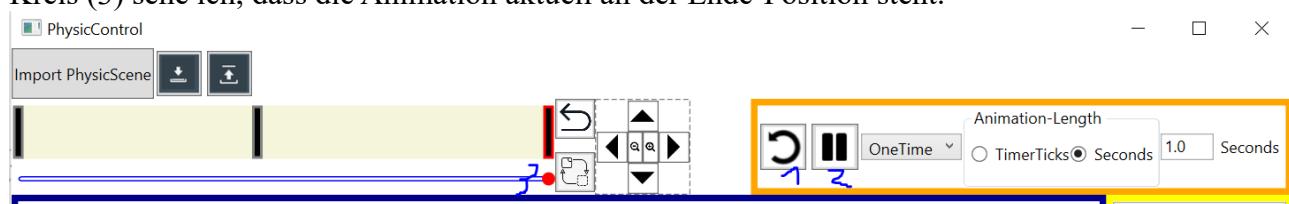
Schritt 12: Ich erzeuge nun eine Kopie vom ersten KeyFrame (liegt bei Time 0) indem ich ihn zuerst per Linksklick selektiere und dann klicke ich per Rechtsklick in ein freien Bereich und verschiebe den neu erstellten KeyFrame nach ganz rechts. Auf diese Weise beginnt die Animation, indem das Polygon nach unten hängt. Dann bewegt es sich kurz etwas nach oben und dann geht es zurück zum Ausgangspunkt.



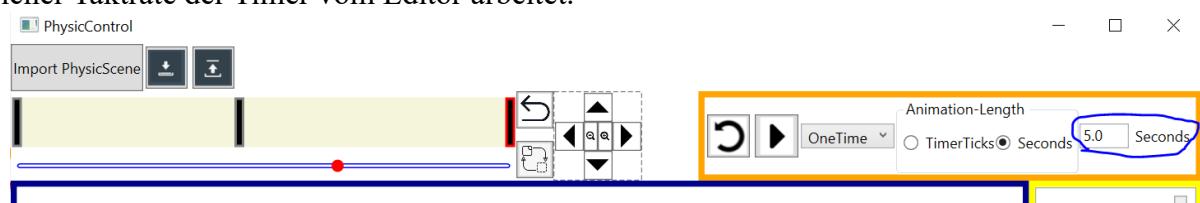
Schritt 13: Ich kann die Animation nun abspielen, indem ich auf den Play-Button drücke. Die Animation wird genau einmal abgespielt und dann bleibt sie am Ende stehen.



Drücke ich auf (1) startet die Animation von vorne. Drücke ich auf (2) pausiert sie. An den roten Kreis (3) sehe ich, dass die Animation aktuell an der Ende-Position steht.



Die Länge der Animation kann entweder in Sekunden eingestellt werden oder in der Anzahl der TimerTicks. Bei der TimerTick-Anzahl hängt die Animationsgeschwindigkeit dann davon ab, mit welcher Taktrate der Timer vom Editor arbeitet.



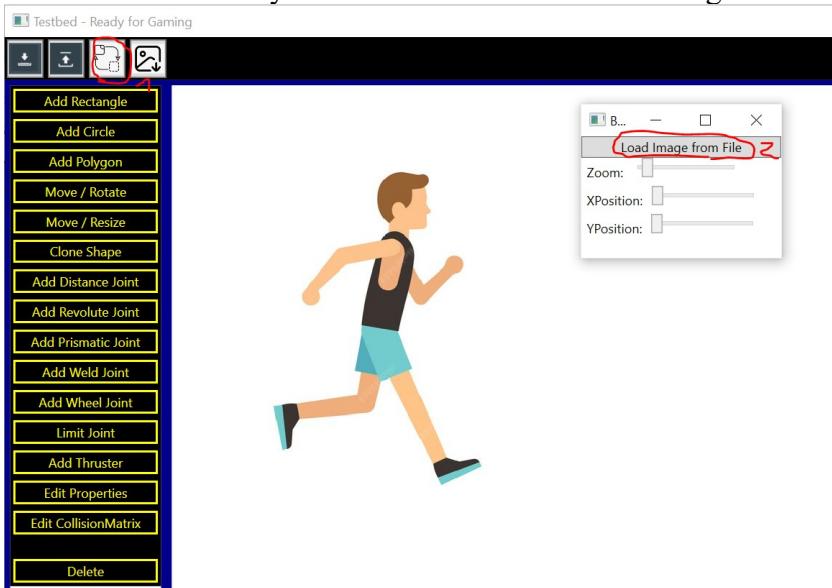
Beispiel 2: AutoLoop-Animation

Diesmal soll eine laufend Person animiert werden. Als Vorlage für die Animation wird eine Sprite-Datei verwendet:

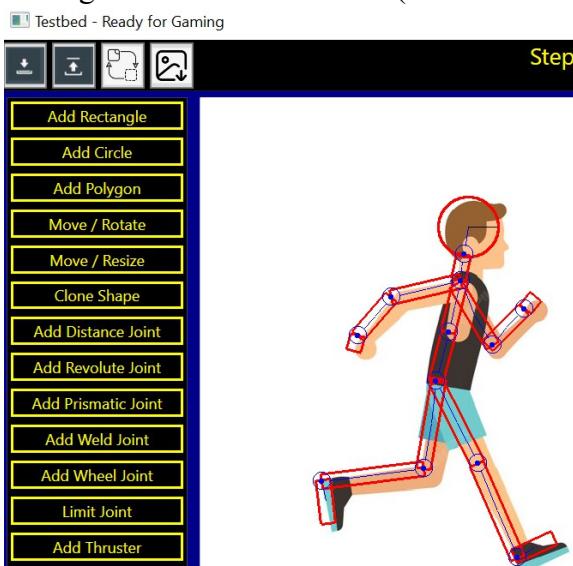


Wir kopieren die erste Figur aus der Sprite-Datei in eine extra Bilddatei und laden diese Datei im Physik-Editor ein.

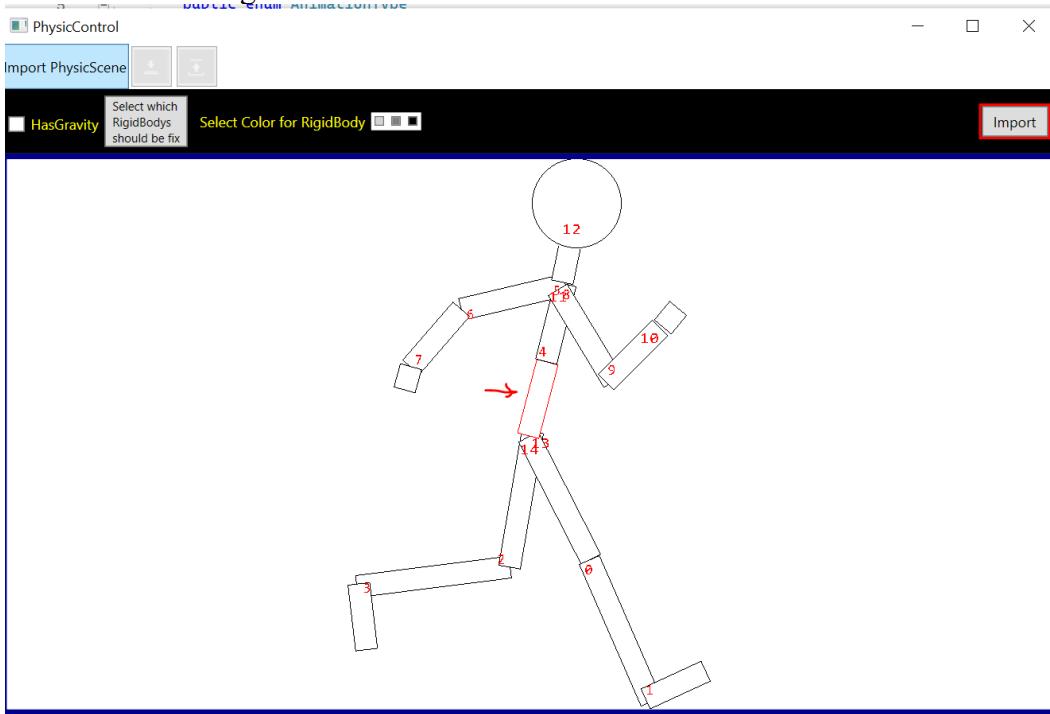
Gehe dazu in den Physik-Editor und ändere den Hintergrund.



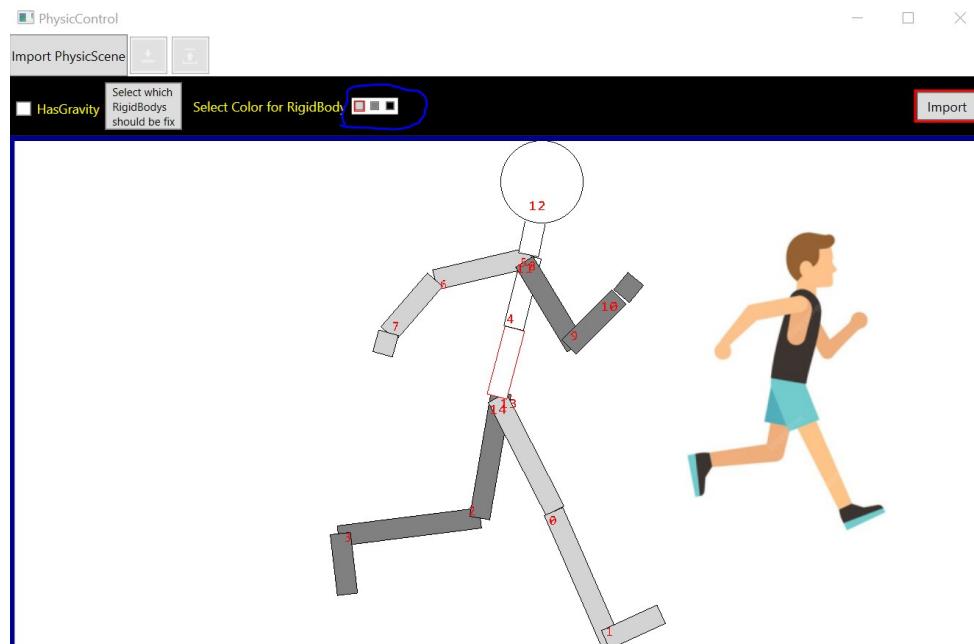
Erzeuge nun lauter Rechtecke (über den Tab-Mode) und bau die Figur nach:



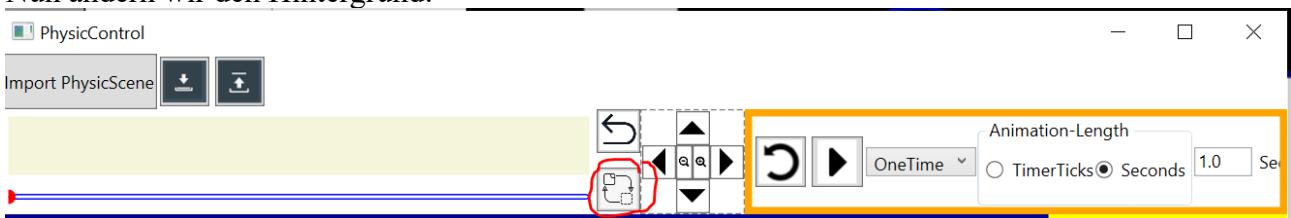
Importiere das Physik-Objekt in den KeyFrameAnimator. Diesmal legen wir fest, dass das Rechteck in der Mitte der Figur fix sein soll:



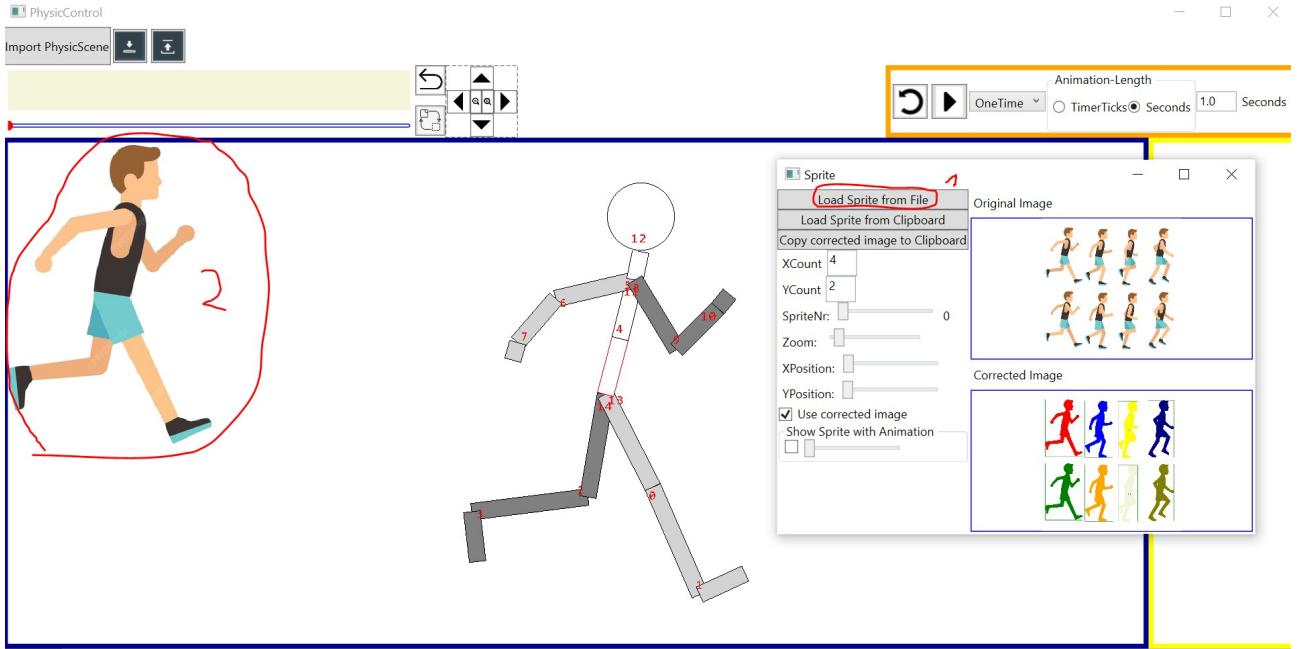
Damit man nun weiß, welcher Arm und welches Bein nun vorne ist, und welches hinten, gibt es die Möglichkeit die Rechtecke einzufärben. Vorlage ist das Bild, was auch schon im Physik-Editor genommen wurde. Gliedmaßen, die hinten liegen bekommen dunkles Grau und die vorderen Teile bekommen ein helleres Grau.



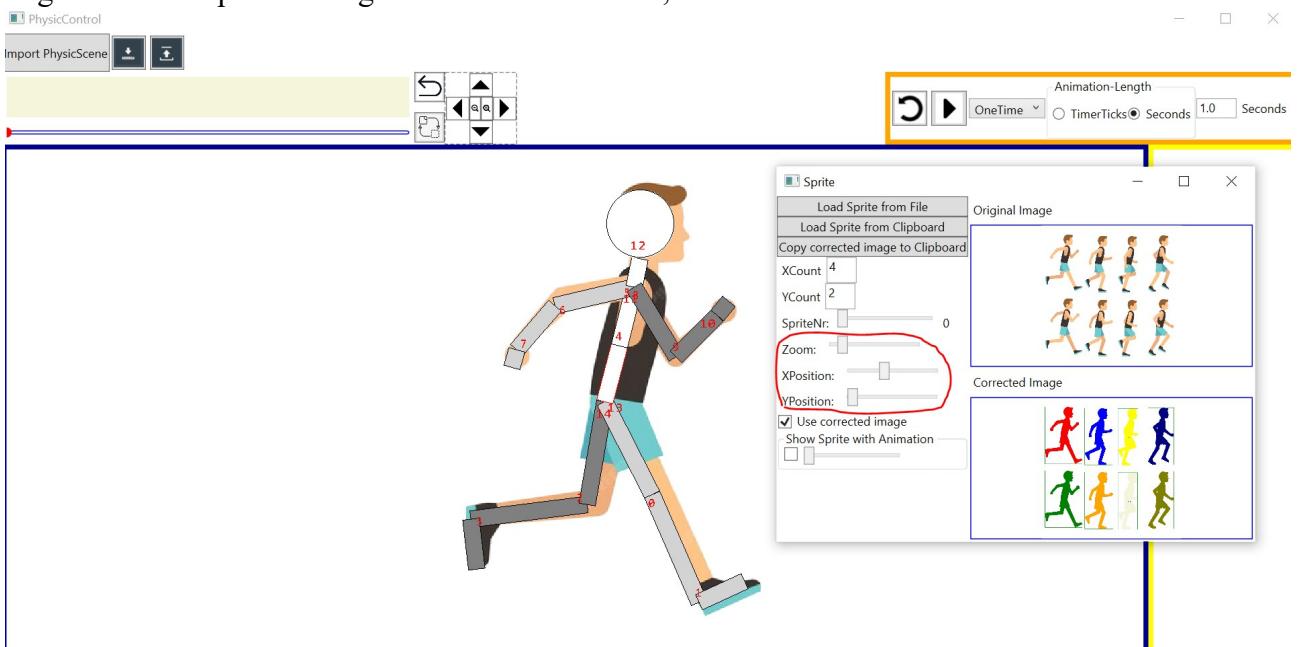
Nun ändern wir den Hintergrund:



Über „Load Sprite from File“ laden wir die Sprite-Datei ein (1) wodurch sie dann als Hintergrundbild (2) angezeigt wird.

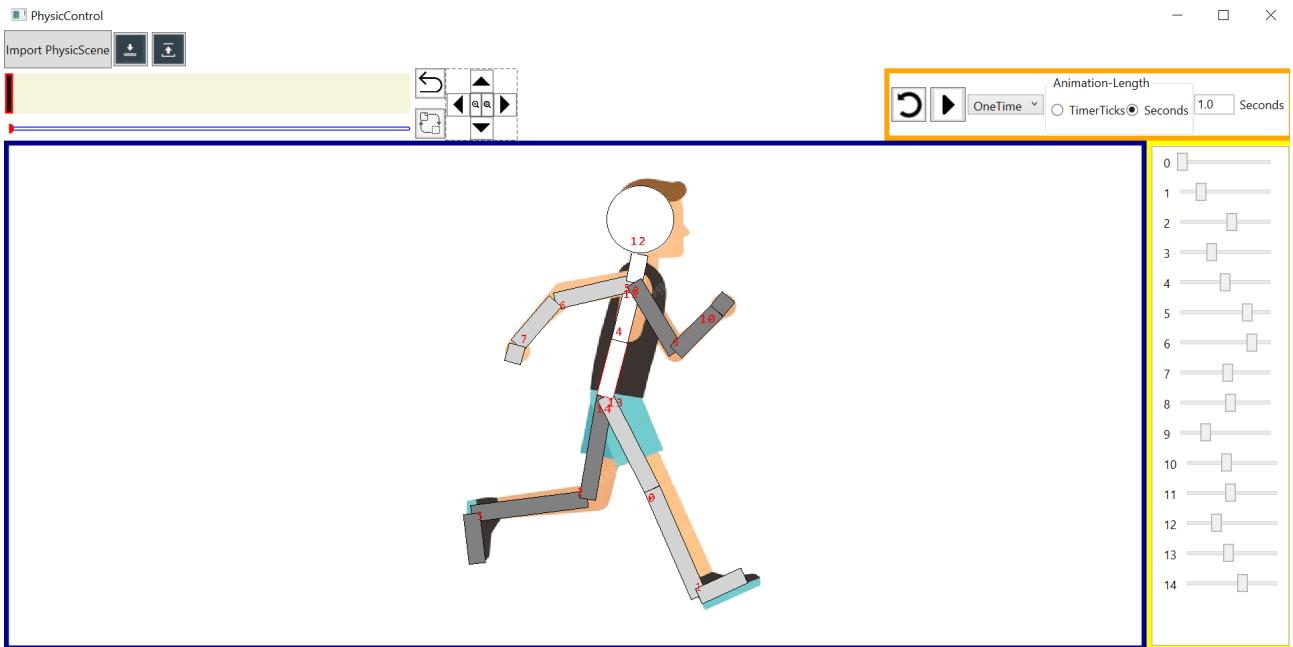


Nun muss das Hintergrundbild so verschoben werden, dass es mit dem Physik-Model übereinander liegt indem im Sprite-Dialogfenster über XPosition, YPosition und Zoom das Bild verschoben wird:

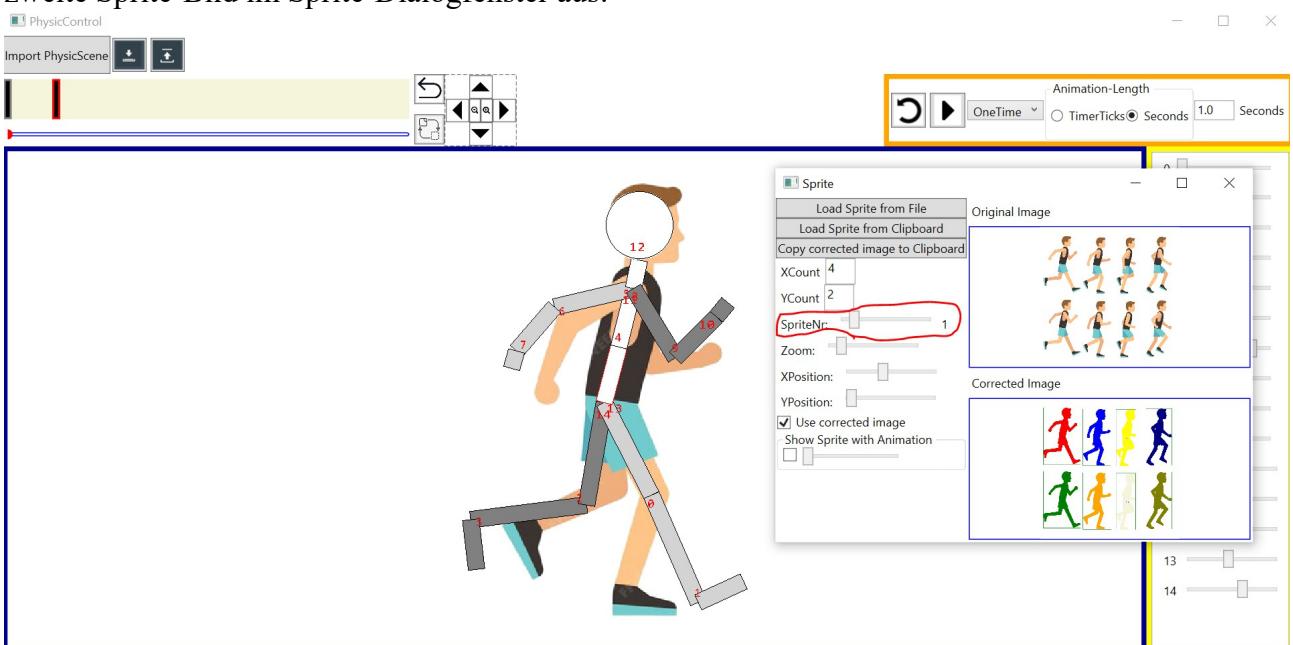


Über „SpriteNr“ kann eingestellt werden, welches Teilbild aus dem Sprite gezeigt wird. Aktuell wird Bild 0 gezeigt (linkes oberes Bild).

Nun erzeugen wir per Rechtsklick unseren ersten Key-Frame, und schieben ihn nach links:

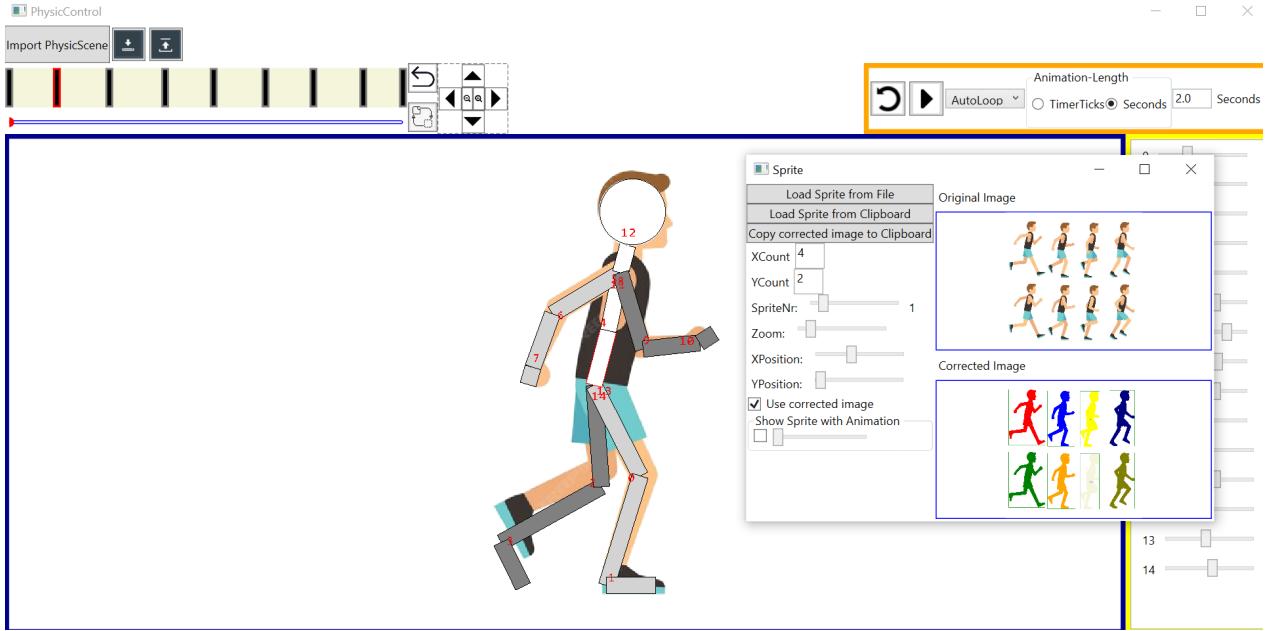


Dann erzeugen wir unseren zweiten KeyFrame daneben und selektieren ihn. Dann wählen wir das zweite Sprite-Bild im Sprite-Dialogfenster aus:



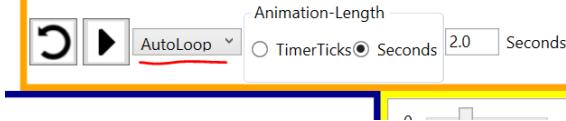
Hier sieht man, dass das grau/weiße Physikmodel nun nicht mehr mit dem Hintergrundbild übereinstimmt. Nun müssen im gelben rechten Bereich alle Gelenke so angepasst werden, dass sie mit dem Hintergrund wieder übereinstimmen.

Das ganze macht man nun für alle Sprite-Teilbilder und erstellt somit alle KeyFrames:



Wichtig: Der erste und der letzte Key-Frame müssen übereinstimmen wenn man eine AutoLoop-Animation nutzt, da die Bewegung sonst beim Übergang zwischen zwei Sequenzen nicht flüssig aussieht. Das macht man dadurch, indem an eine Kopie vom ersten KeyFrame erzeugt und sie dann ans Ende verschiebt.

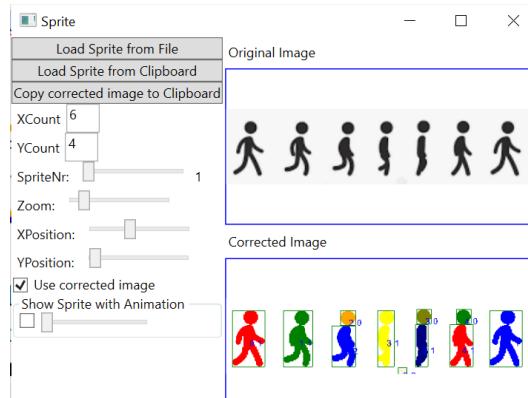
Diesmal nutzen wir die AutoLoop-Einstellung im Play-Control.



Hinweis zum Sprite-Dialogfenster: Der Hintergrund vom Spritebild muss entweder weiß oder schwarz sein. Die Objekte müssen aus einer zusammenhängenden Pixelfläche bestehen. Wenn also so ein Bild hier genommen wird, wo der Kopf in der Luft schwebt:



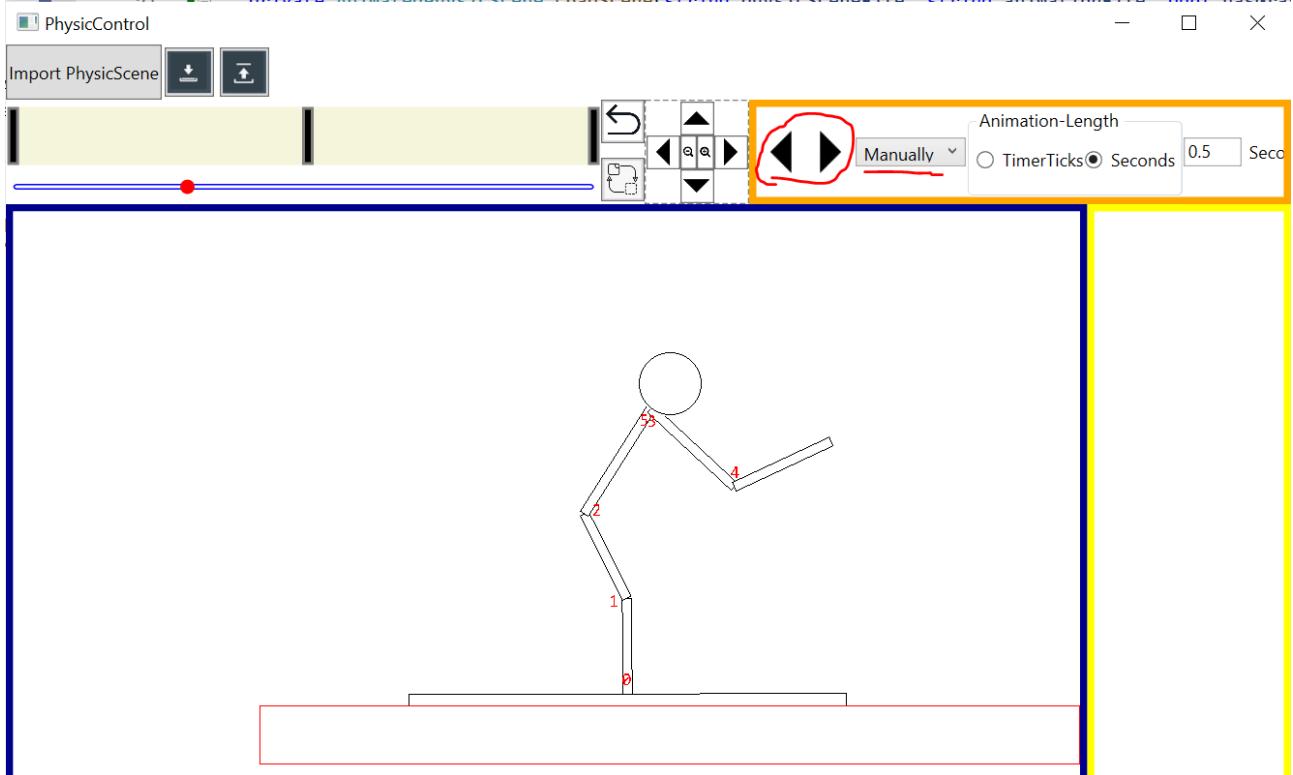
Dann denkt der Sprite-Importer, dass das zwei Zeilen in der Bilddatei sind. Im unteren Fenster sieht man welche Bereiche er gefunden hat. Dort ist der Kopf teilweise einzeln grün umrandet und teilweise nicht:



Man kann auch auf das Parsing der Spritedatei verzichten und die „Use corrected image“-Option deaktivieren. Allerdings müssen dann alle Sprite-Teilbilder gleich groß sein!

Beispiel 3: Manuelle Animation

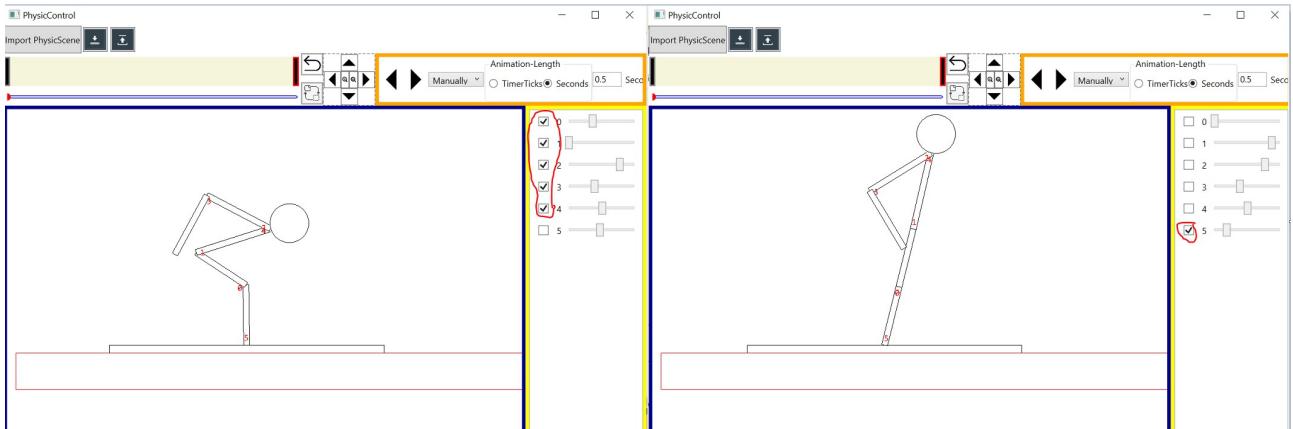
Um ein Ski-Springer zu animieren, welcher sich hinhocken und aufstellen kann, wird wieder eine Person im Physik-Editor gebaut und dann wird (diesmal ohne Sprite-Hintergrund) eine Animation erstellt, wo er von komplett stehend zu komplett hockend sich bewegt. Nun nutzen wir den „Manually“-Animationstyp und halten die MoveForward/MoveBackward-Tasten mit der Maus gedrückt. Nur dann, wenn die Maus gedrückt ist, bewegt er sich. Über die Animation-Length kann eingestellt werden, wie schnell er diese Bewegung ausführt.



Es ist erlaubt, dass man pro PhysicScene auch mehrere Animationsdateien erstellt. Die erste Animation von unseren Ski-Springer lässt den Springer sich hocken oder hinstellen. Die zweite Animation soll den Springer nach vorne oder zurück lehnen lassen, indem sein Fußgelenk sich bewegt.

Wichtig wenn man mehrere Animationen gleichzeitig pro Objekt laufen lässt: Jedes Gelenk darf immer nur von einer Animation gesteuert werden.

Links wurde eine Animation erzeugt, welche die Gelenke 0 bis 4 steuert, damit der Ski-Springer sich aufstellt/hinhockt. Rechts wurde nur Gelenk 5 animiert, um ihn nach vorne/hinten zu bewegen.

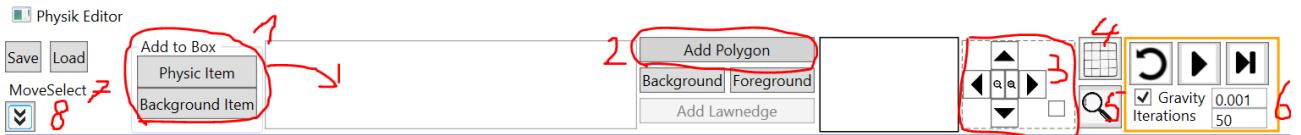


Im LevelEditor kann dann für jede Animation eine Taste hinterlegt werden. Hält man diese gedrückt, dann wird die Animation dann nach vorne oder hinten abgespielt.

7.2 Spiele direkt im Editor

Bis jetzt wurde nur der Textur- und Animationseditor allgemein erklärt. Diese beiden Editoren sind ein Bestandteil vom Leveleditor. Hier soll nun anhand von Beispielen die Nutzung des Leveleditors erklärt werden.

Grundaufbau des Editors



1:

Wenn man auf „Physic Item“ oder „Background Item“ klickt, dann kann man ein neues Objekt erzeugen, was dann in der Auswahlbox landet. Von dort aus kann man dann per Drag & Drop das Objekt in die Szene ziehen. Die Idee ist, dass man komplexe Objekte wie Autos oder Personen einmal erzeugt und dann können sie aber mehrmals im Level vorkommen.

2:

Der Levelrand wird über „Add Polygon“ erzeugt. Diese Objekte landen nicht in der Box. Die Textur der Levelrandpolygone kann über „Background“ und „Foreground“ definiert werden.

3:

Hier kann die Position/Zoom der Kamera eingestellt werden. Klickt man auf die Checkbox, wird der Autozoom aktiviert, so dass das gesamte Level dann genau in den Bildschirm passt.

4:

Mit diesen Schalter kann ein Gitter im Editor-Hintergrund eingeblendet werden. Das Gitter hat eine Snap-Funktion so dass Objekte genau an den Gitterpunkten/Linien angeordnet werden können.

5:

Mit diesen Schalter kann ein kleines Vorschaufenster rechts unten eingeblendet werden. Das gelbe Rechteck zeigt den Sichtbereich. Man kann es mit der Maus auch verschieben und mit den Mausrad die Größe ändern.

6:

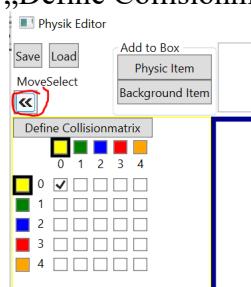
Hiermit kann der Simulator gesteuert werden. Man kann die Simulation im Einzelschritt oder Timergesteuert ablaufen lassen.

7:

Hier steht immer die Funktion, welche im Editor gerade aktiv ist.

8:

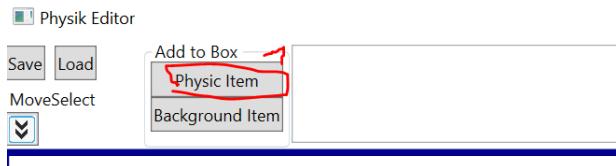
Mit diesen Schalter kann links vom Editor noch eine weitere Menüleiste aufgeklappt werden. Klickt man da drauf sieht man noch die „Define Collisionmatrix“-Funktion und den Level-Baum.



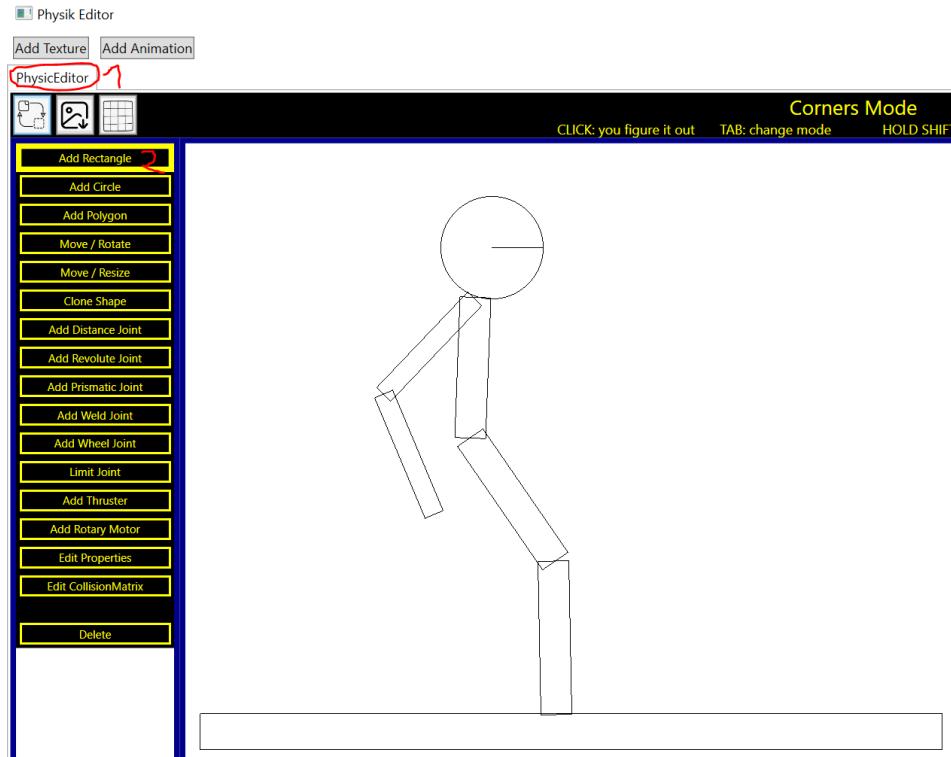
Beispiel 1: Der Ski-Fahrer

Um eine Szene zu bauen, wo ein Skifahrer über eine Schanze springt müssen folgende Schritte getan werden:

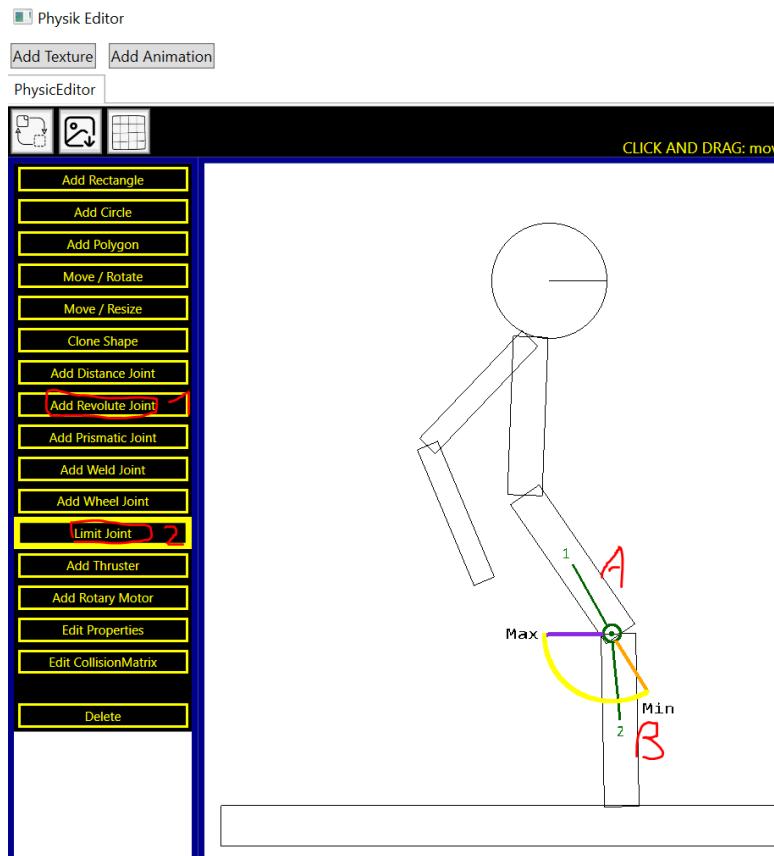
Schritt 1: Den Skifahrer erzeugen. Klicke dazu auf „Physic Item“



Du bist nun im PhysicEditor (1) und kannst nun über lauter Rechtecke und ein Kreis den Fahrer bauen:



Sein Kniegelenk definierst du indem du auf „Add Revolute Joint“ klickst und dann zuerst auf sein Oberschenkel (Rechteck A) klickst und dann auf sein Unterschenkel (Rechteck B). Dann musst du die Ankerpunkte definieren wo du sagst, an welcher Stelle vom Rechteck soll das Gelenk befestigt sein. Über Limit Joint (2) definierst du dann, in welchen Winkelbereich das Knie bewegt werden darf. Der grüne Hebelarm mit der 2 (Unterschenkel) darf dann nur im gelben Kreissegment sich aufhalten.



Schritt 2: Nachdem der Skifahrer im PhysicEditor erzeugt wurde klickst du nun auf „Add Texture“ (1) und definierst dann im TextureEditor (2) die Texturen.

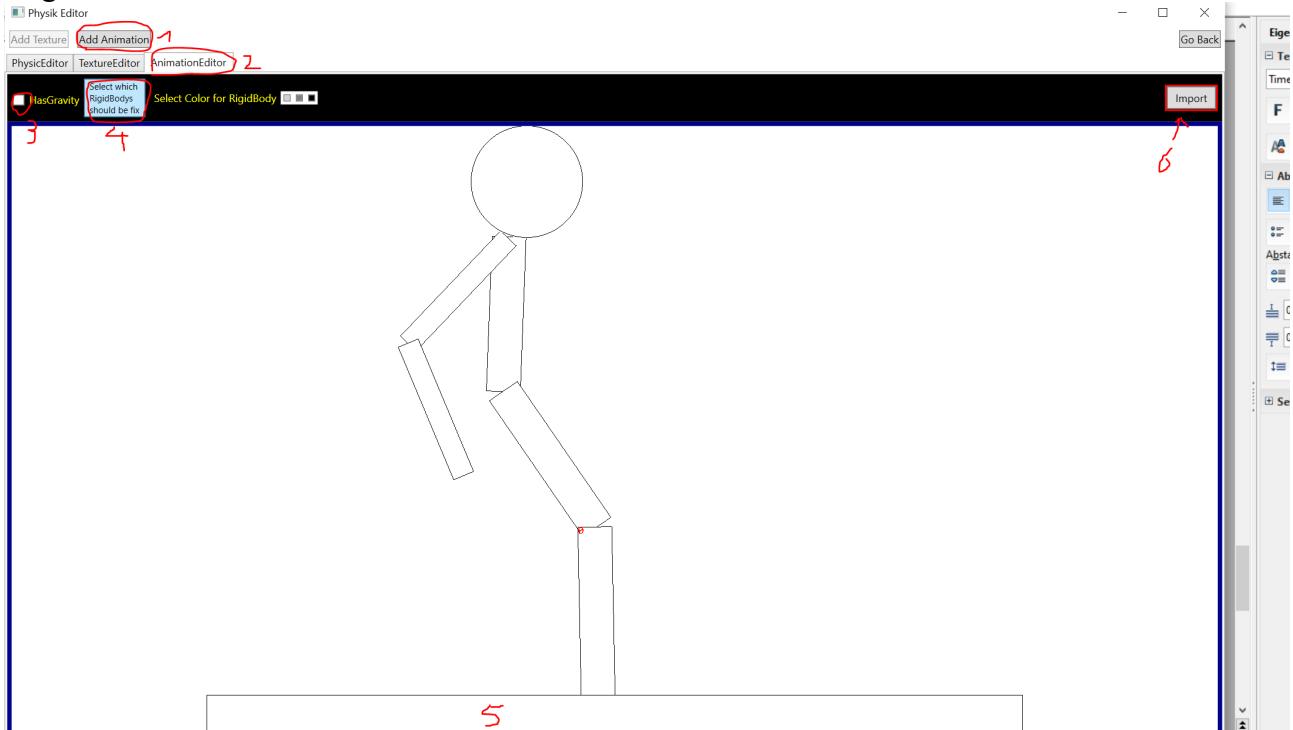


Den Kopft texturierst du indem du zuerst auf den Kreis klickst (1) und dann eine Textur dafür auswählst (2)

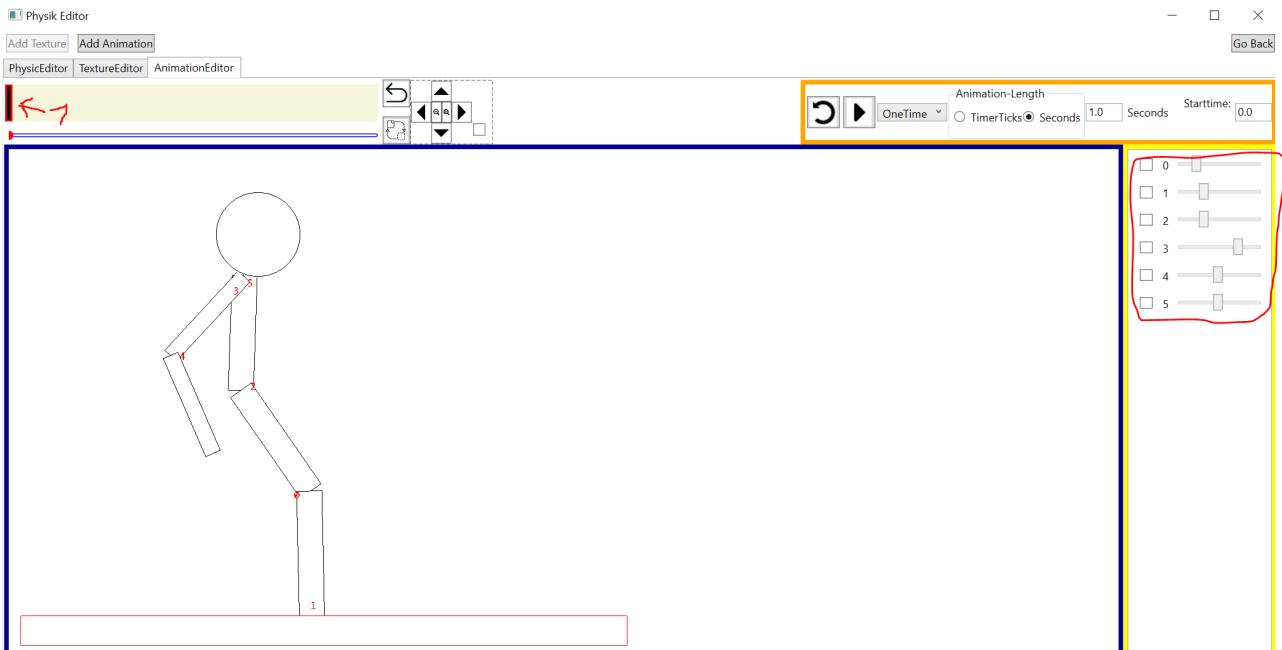


Du kannst diese Textur dann noch über den grünen Punkt (3) verschieben und über die Eckpunkte (4) kannst du sie drehen und skalieren (Shift halten).

Schritt 3: Nachdem alle Gliedmaßen eine Textur haben klickst du auf „Add Animation“ (1) und gehst dann in das neu erstellte Animation-Tab (2). Da der Skifahrer auf der Erde und nicht im Weltraum sich aufhalten soll, aktivierst du die Schwerkraft (3) und dann musst du noch festlegen, welcher Teil vom Physikmodel soll unbeweglich sein. Dazu gehst du auf (4) und klickst dann auf den Ski (5). Dieser Schritt ist nötig, da sonst der Skifahrer beim Erstellen der Animation nach unten wegfallen würde.



Im Animationseditor klickst du zuerst auf den ersten Keyframe wodurch dann rechts alle Gelenkpositionen eingeblendet werden:

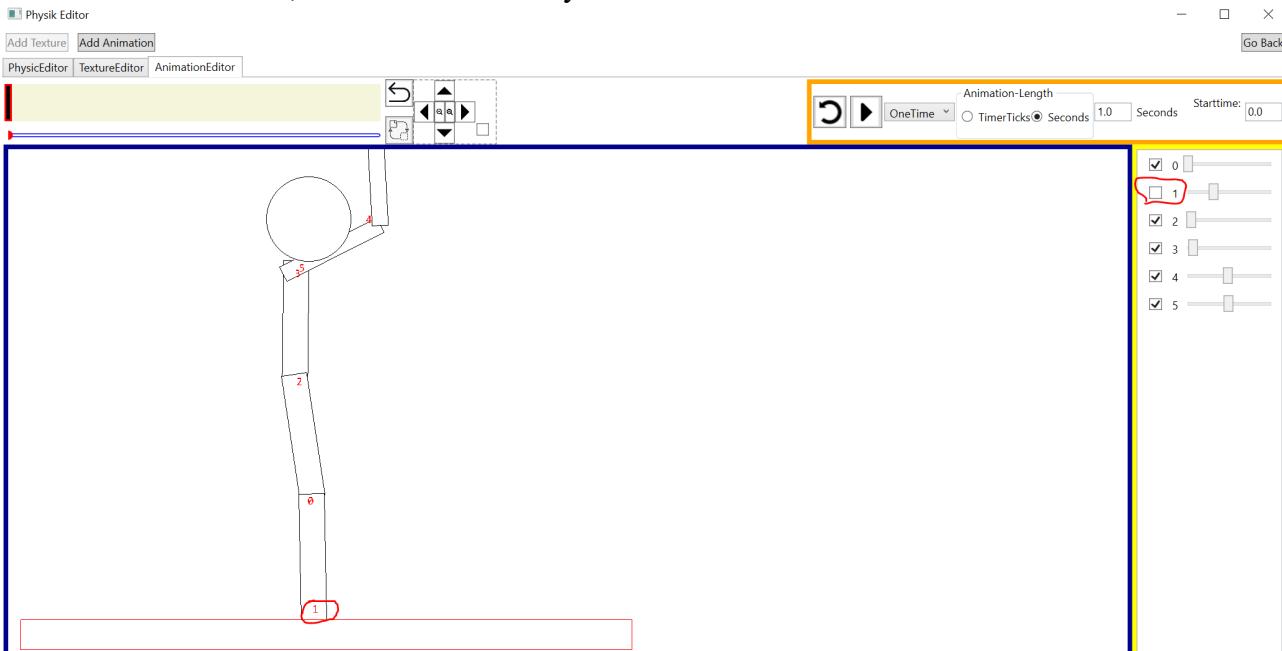


Der Skifahrer soll zwei Bewegungen können.

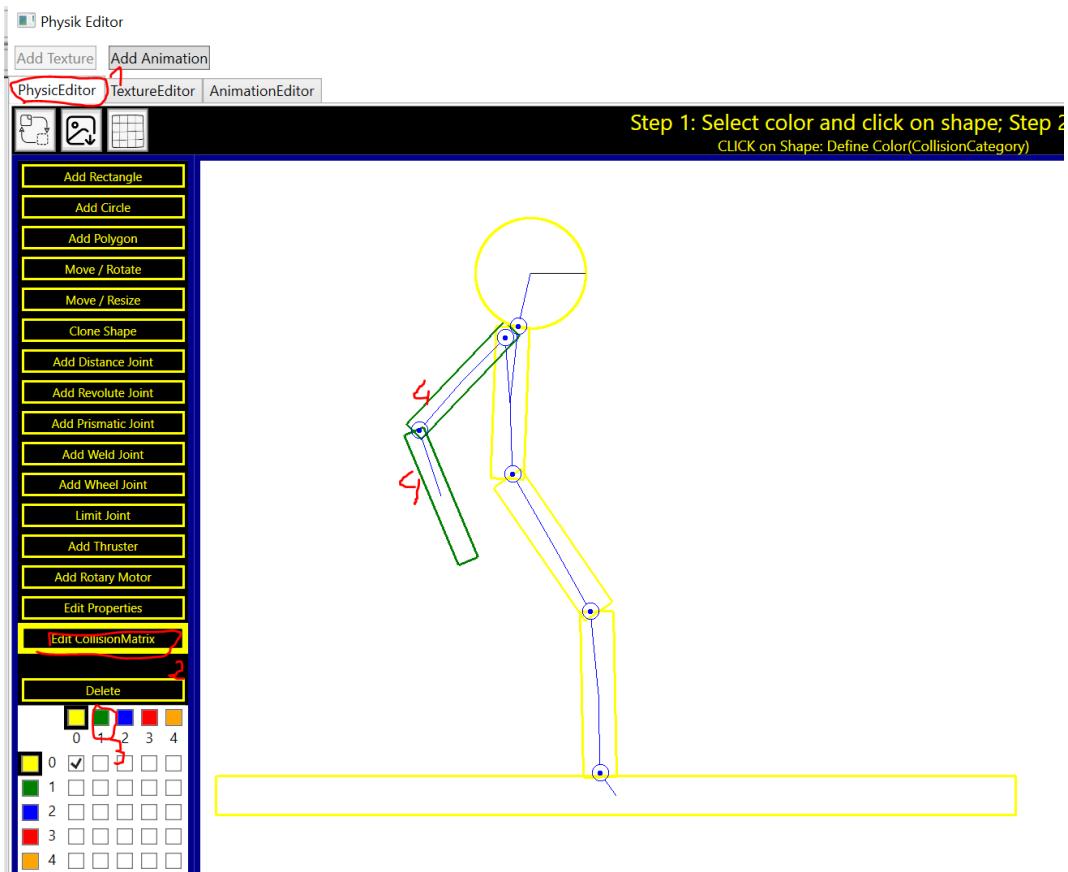
Bewegung 1: Er stellt sich hin oder hockt sich hin

Bewegung 2: Er lehnt sich nach vorne oder hinten

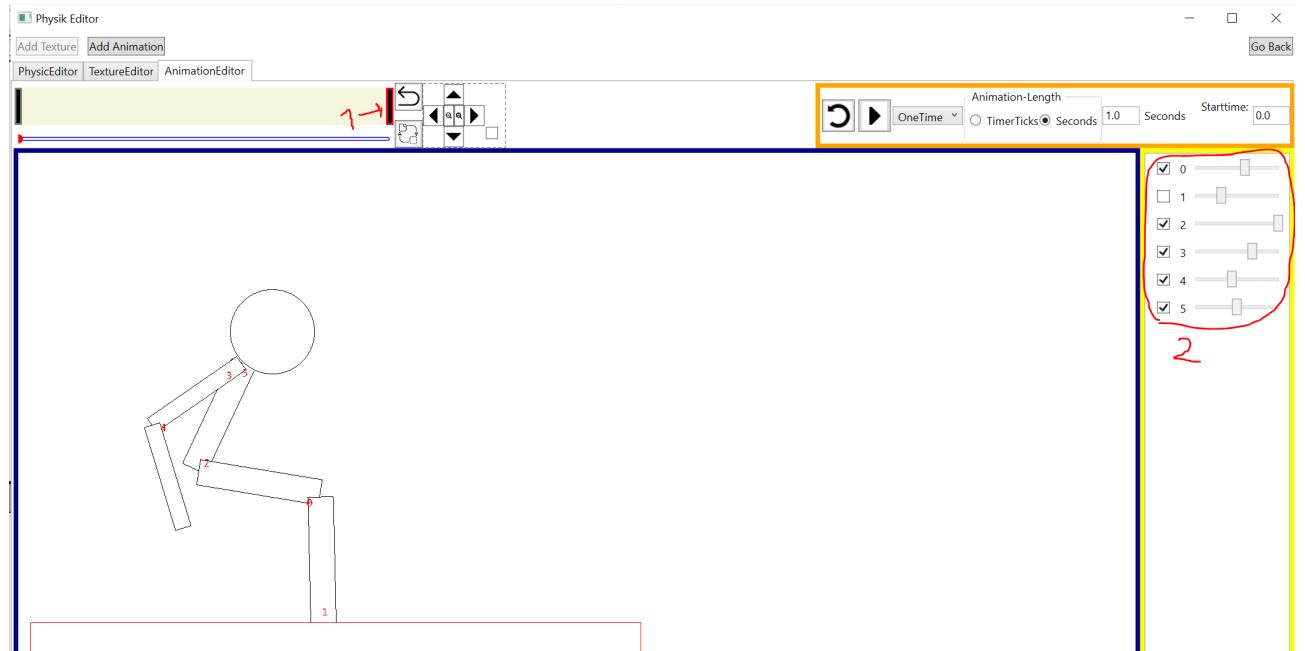
Für die erste Bewegung müssen alle Gelenke außer das Fußgelenk animiert werden. An den roten Zahlen sieht man, welches Gelenk welche Nummer hat. Gelenk 1 soll hier nicht bewegt werden also wird es nicht mit an der rechten Seite aktiviert aber alle anderen Gelenke schon. Nun stellen wir die Gelenke so ein, dass unser erster Keyframe den Fahrer aufrichtet.



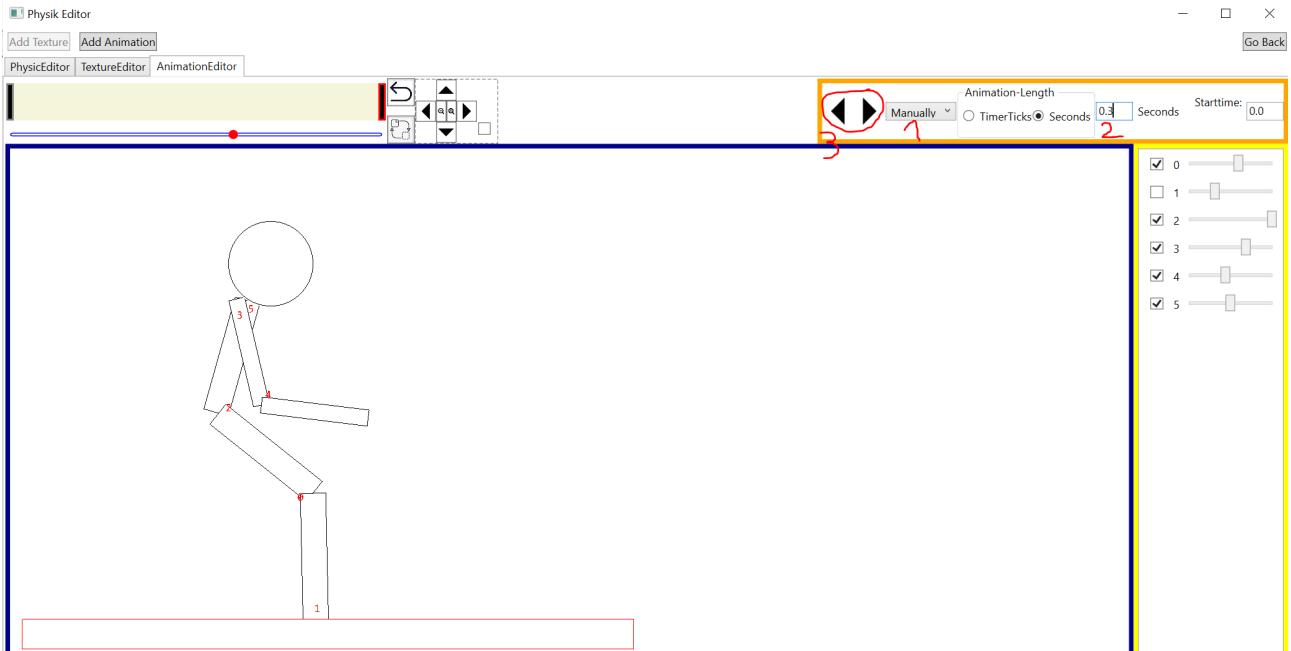
Wir stellen aber fest, dass seine Arme gegen sein Kopf und sein Unterarm gegen sein Körper stößt. Damit das nicht passiert gehen wir nochmal zum PhysikEditor-Tab zurück und sagen, dass die Arme die grüne Kollisionskategorie bekommen. In der Kollisionsmatrix kollidiert nur Gelb mit Gelb. D.h. die grünen Arme können nun am Körper vorbei gehen.



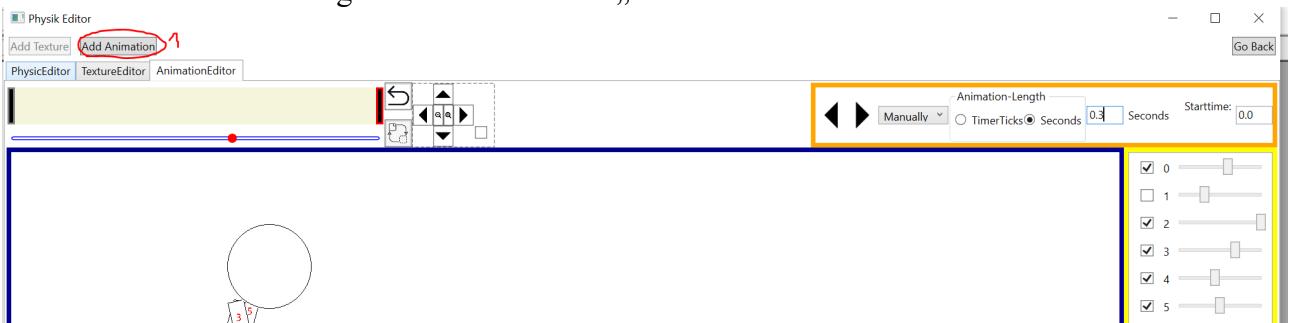
Zurück in Animations-Tab erzeugen wir nun ein zweiten Keyframe (1) und lassen den Fahrer nun hocken:



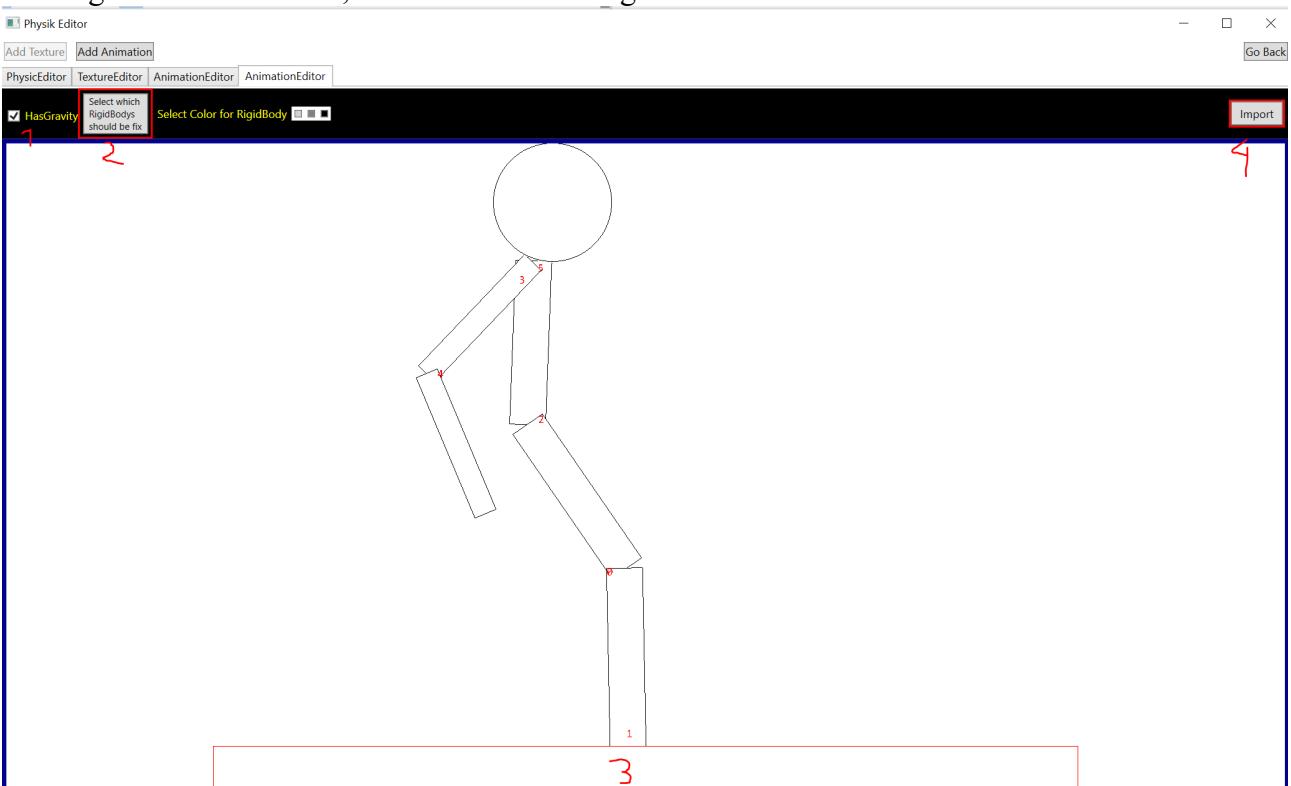
Nun sagen wir noch, dass diese Animation per Tastendruck gesteuert werden soll indem wir sie auf „Manually“ stellen. Dann definieren wir wie schnell die Animation ablaufen soll indem wir bei (2) die Zeit in Sekunden einstellen. Wir testen die Animation über die Playforward/Playbackward-Buttons (3).



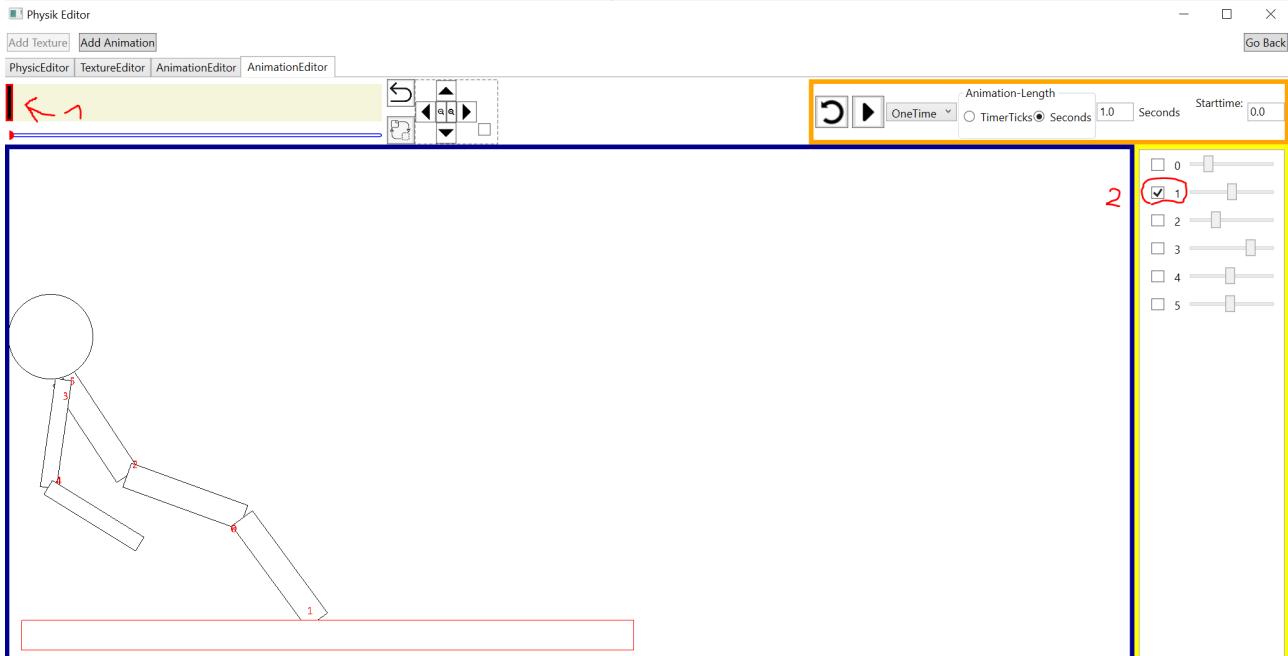
Für die zweite Animation gehen wir wieder auf „Add Animation“



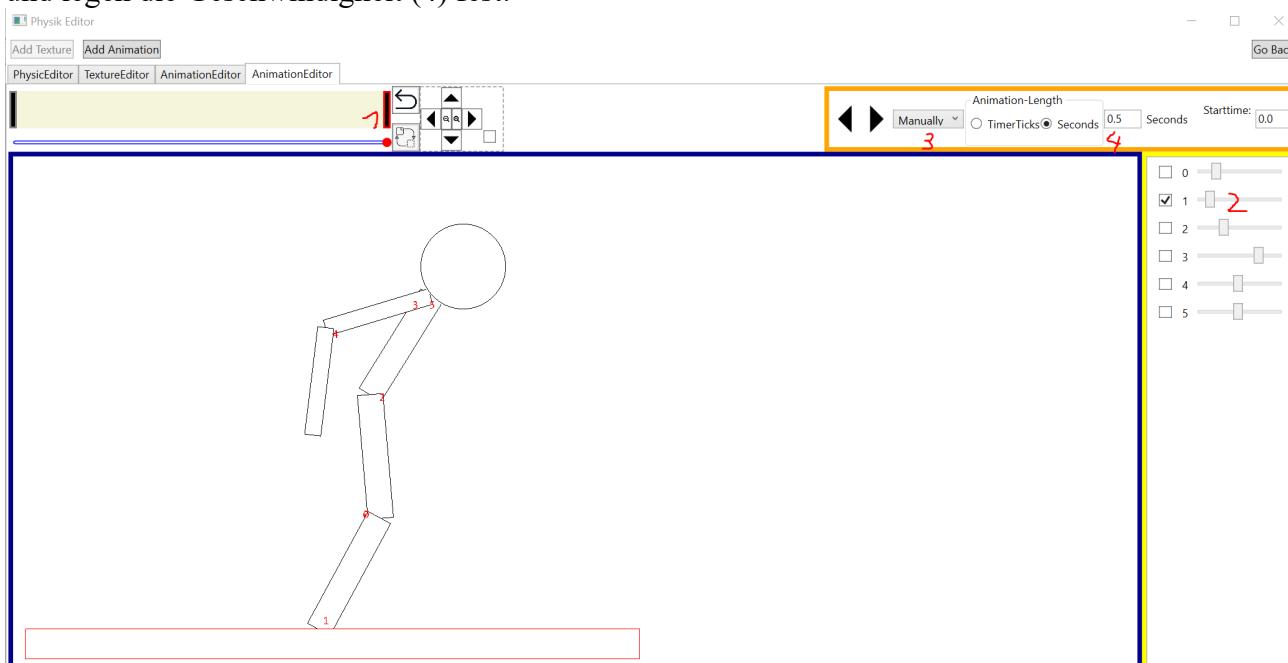
Und legen hier wieder fest, dass es Schwerkraft gibt und die Skier fix sind:



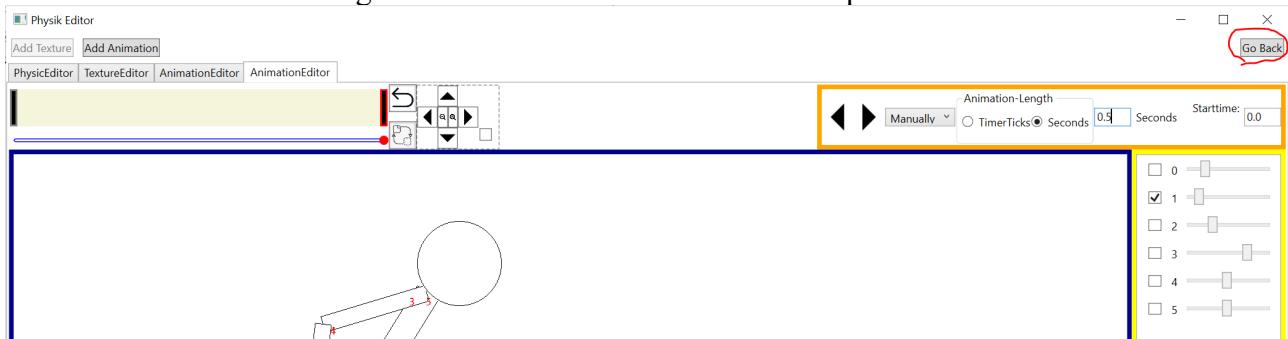
Wir erzeugen den Keyframe an Time-Position 0 und sagen dieses mal, dass nur Gelenk 1 animiert werden soll.



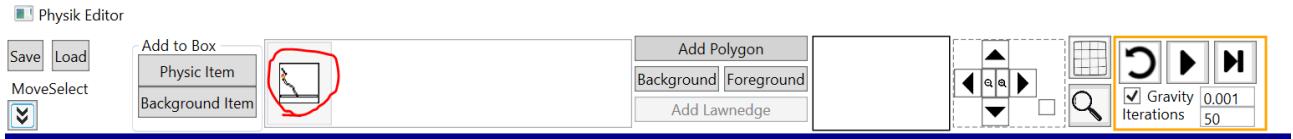
Wir definieren den zweiten Keyframe (1,2) und stellen die Animationsart wieder auf manuell (3) und legen die Geschwindigkeit (4) fest:



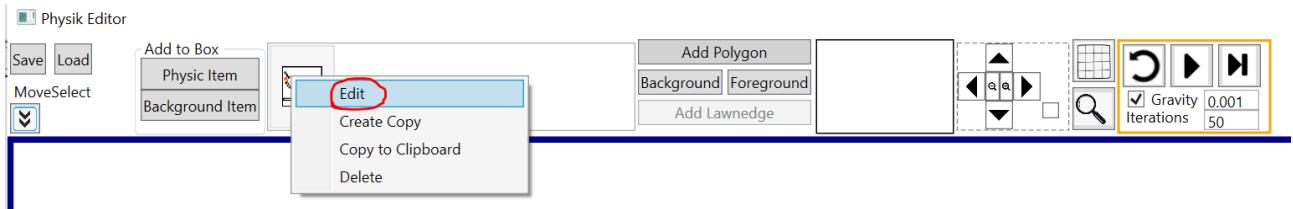
Wenn wir nun auf GoBack gehen kommen wir zurück in den Haupteditor und



dort sehen wir, dass unser Skifahrer nun in der Auswahlbox oben erscheint:

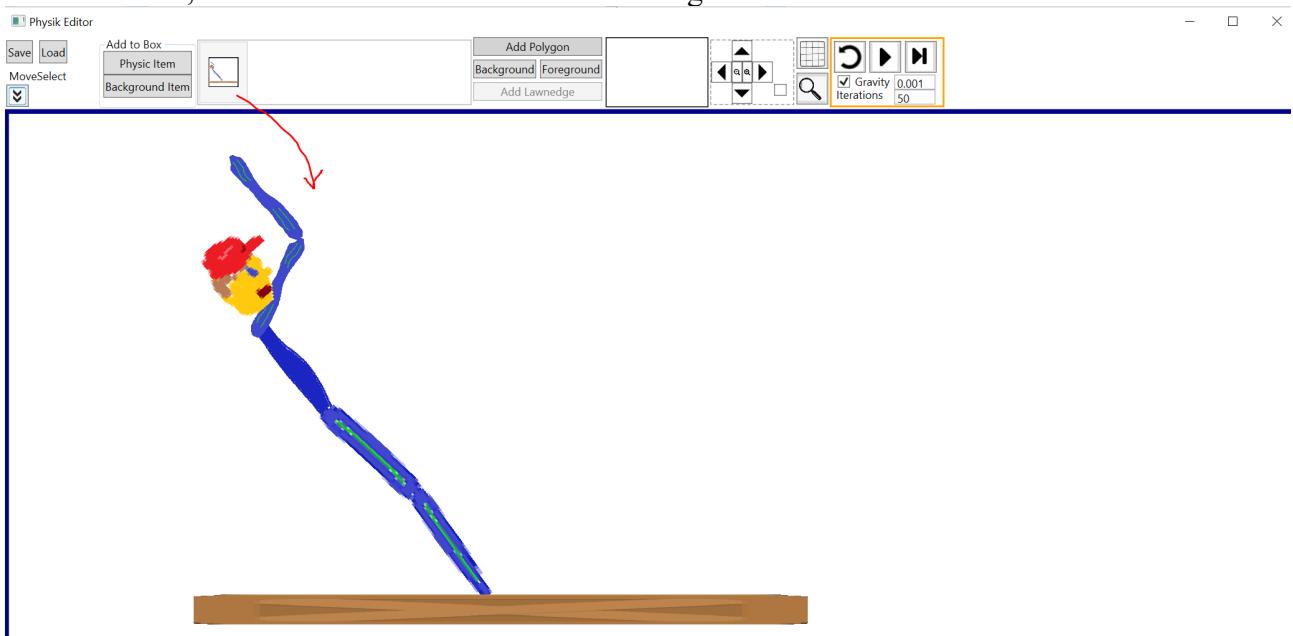


Über rechtsklick->Edit können wir den Skifahrer dann weiter bearbeiten:



Wenn ich jetzt per Drag & Drop den Skifahrer in den Editor ziehe:
Mausklick zum platzieren. Esc um die Platzierungsfunktion zu beenden.

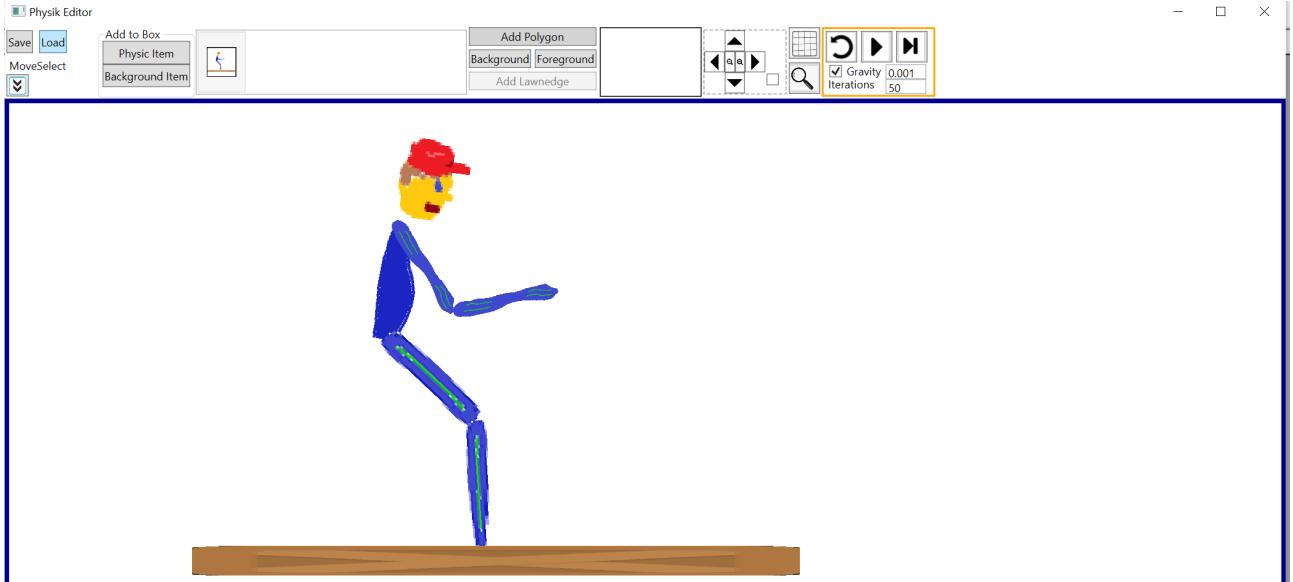
Dann sehe ich, dass der Fahrer sehr weit nach hinten gelehnt ist.



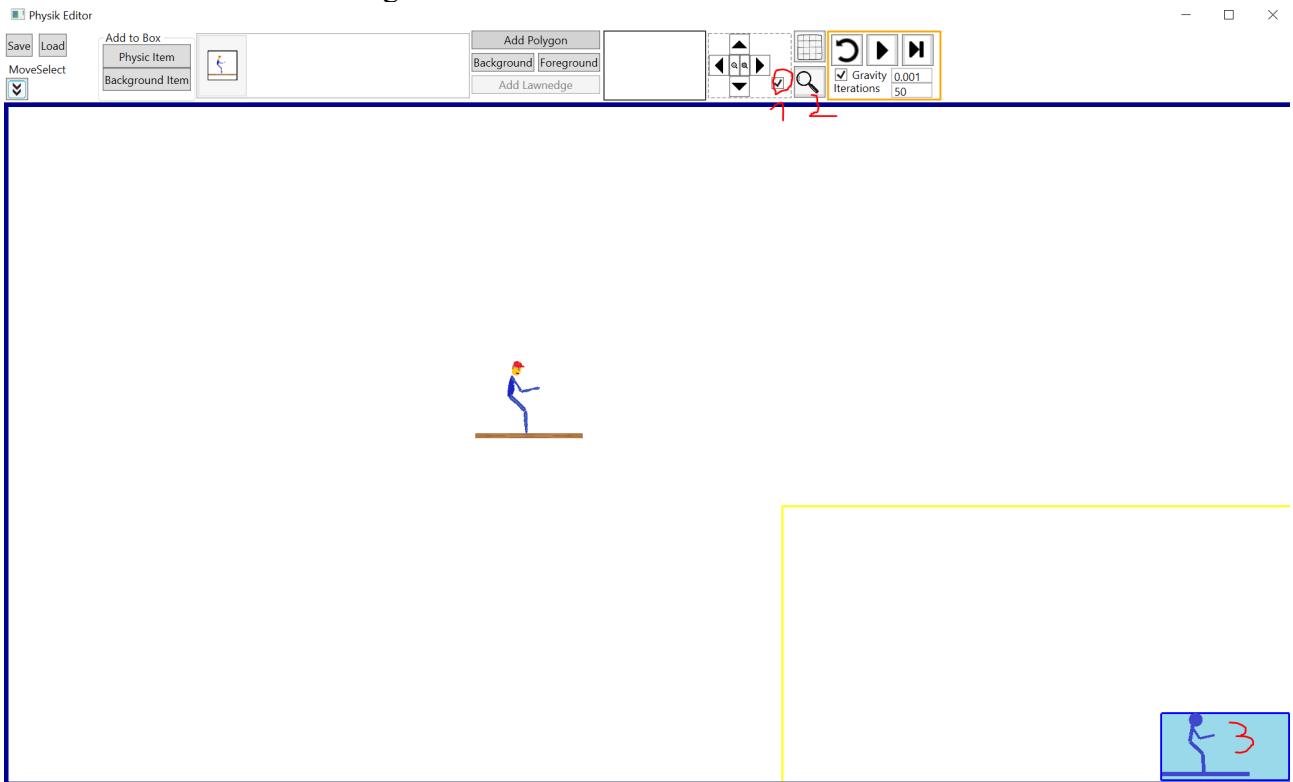
Der Grund dafür ist weil die Animations-Start-Zeit auf 0 eingestellt ist. Wir editieren den Fahrer indem wir mit Rechtsklick->Edit in der Auswahlbox bei beiden Animationen eine Starttime von 0.5 festlegen:



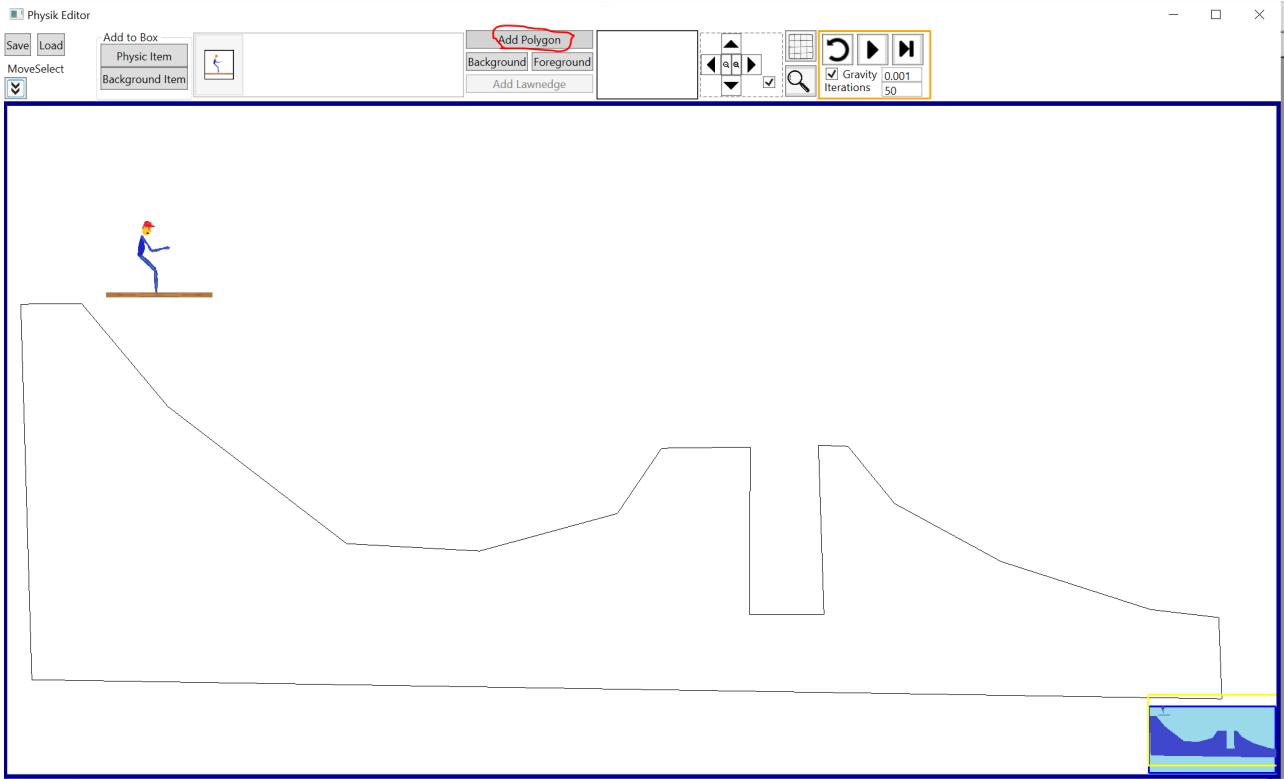
So dass der Fahrer nun so aussieht:



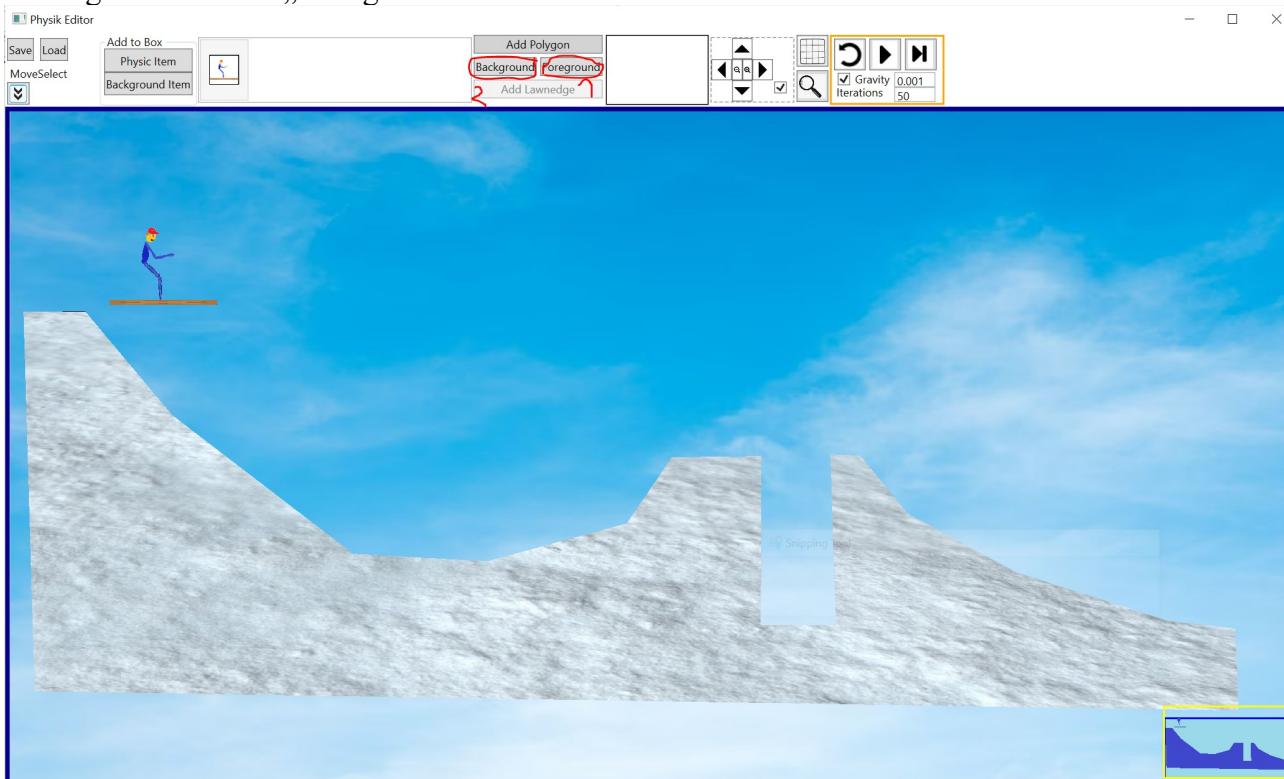
Nun zoomen wir etwas heraus indem wir bei der Kamera die Checkbox aktivieren (ab jetzt ist die Kamera nutzbar) und dann blenden wir das Vorschaufenster ein und zoomen per Mausrand raus, während wir die Maus in das gelbe Rechteck halten:



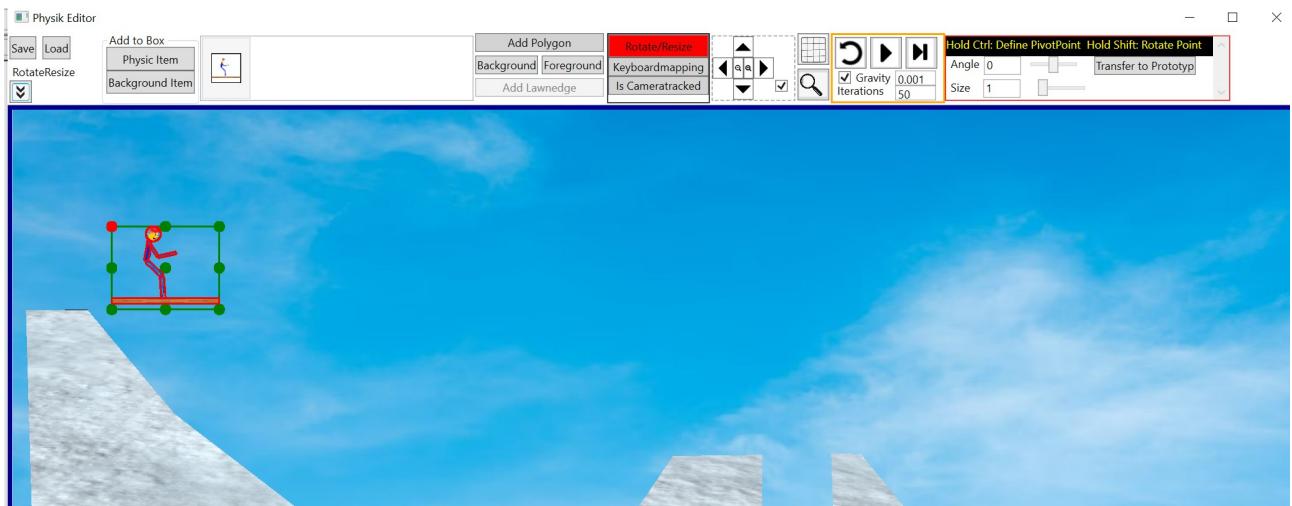
Über die „Add Polygon“-Funktion bauen wir nun eine Schanze.



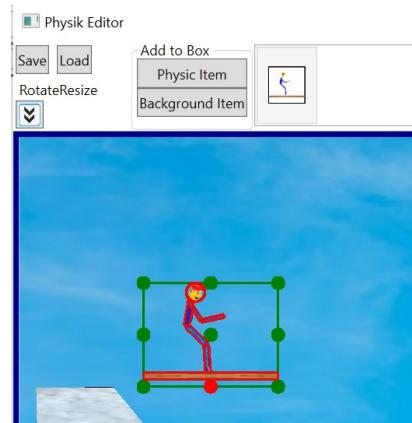
Die Textur von der Schanze (und allen Levelpolygone) legen wir über „Foreground“ fest und das Hintergrundbild über „Background“



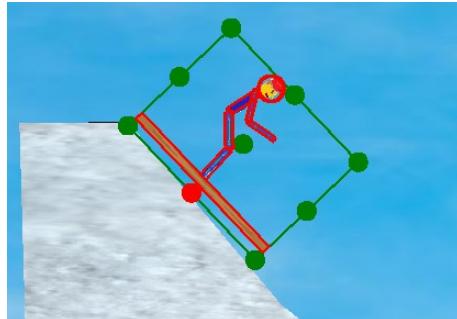
Um den Fahrer nun noch richtig zu platzieren klicken wir einmal mit der Maus drauf wodurch seine Kontrollpunkte angezeigt werden. Der rote Punkt ist sein Rotationspunkt.



Wenn wir die Strg-Taste gedrückt halten und dann auf einen der Kontrollpunkte klicken, dann wird dass das neue Rotationszentrum. In diesen Beispiel wurde nun der untere mittlere Punkt als Zentrumspunkt gewählt:



Wenn wir nun die Shift-Taste gedrückt halten und ein grünen Kontrollpunkt mit der Maus verschieben, dann dreht sich der Fahrer um den roten Punkt:

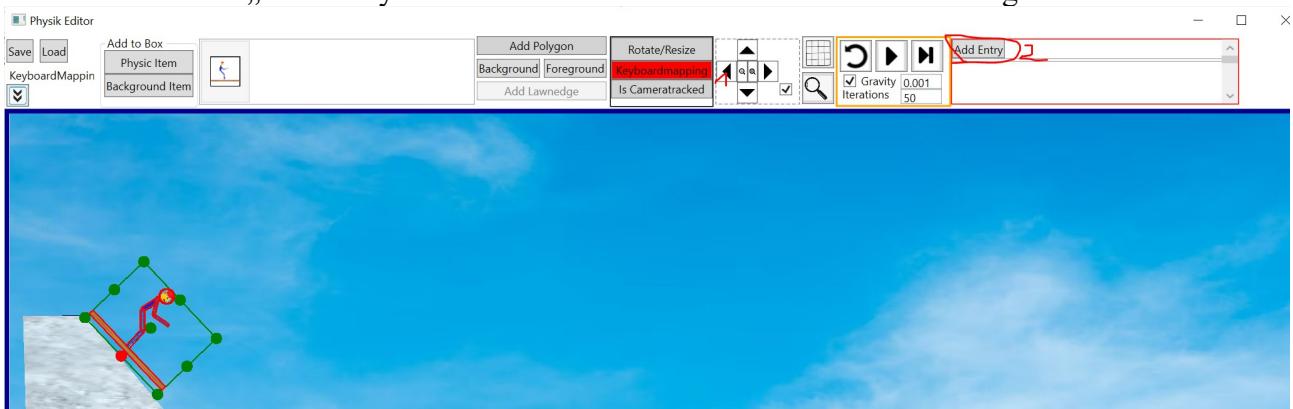


Wenn wir ohne Strg- und Shift-Taste ein grünen Punkt mit der Maus verschieben, dann können wir die Größe vom Fahrer skalieren.

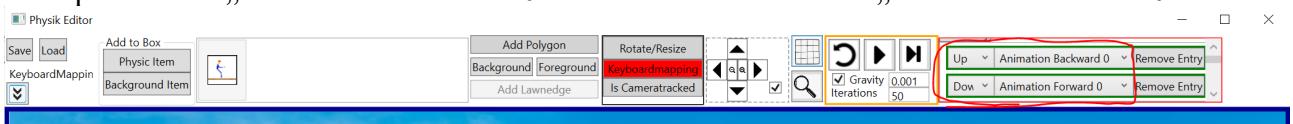
Nun wollen wir festlegen, dass mit den Kursortasten der Fahrer gesteuert werden kann. Dazu muss der Fahrer selektiert sein (ist er bereits) und dann gehen wir auf die „Keyboardmapping“-Funktion:



Und dort dann auf „Add Entry“ um ein neuen Tastendruck-Handler hinzuzufügen:

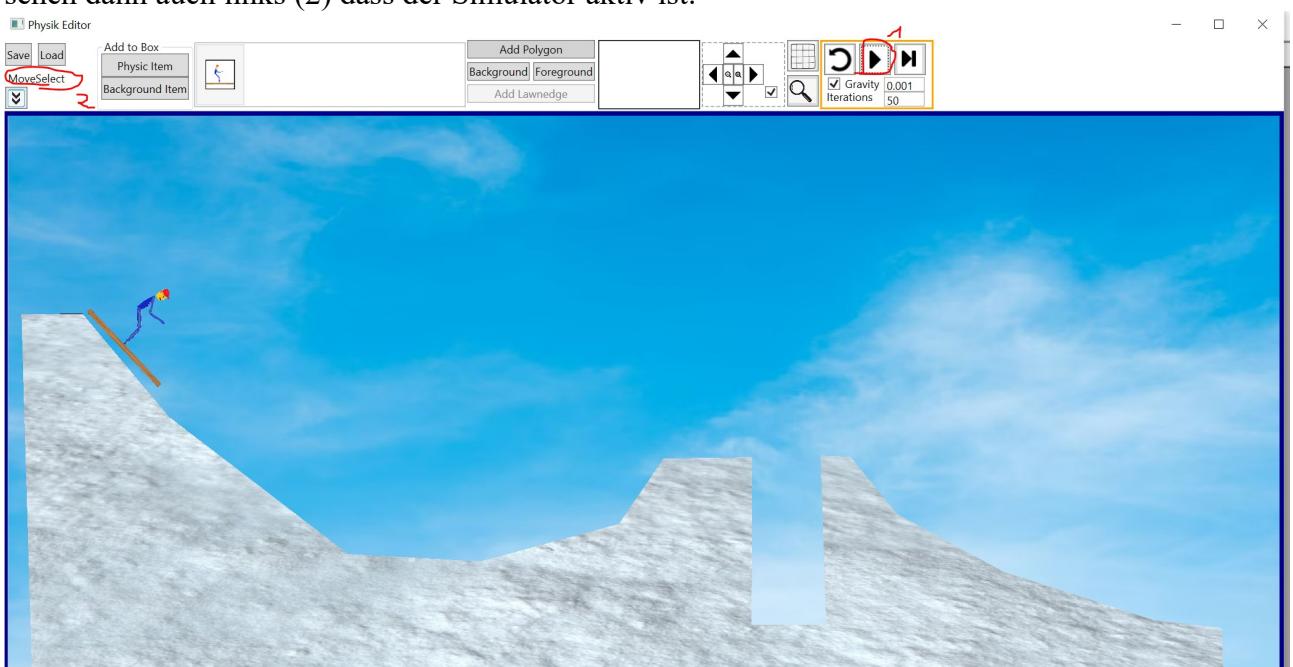


Die erste Animation war dafür da, dass der Fahrer sich hinstellt oder in die Hocke geht. Wir mappen die Up-Taste auf „Animation Backward 0“ und die Down-Taste auf „Animation Forward 0“

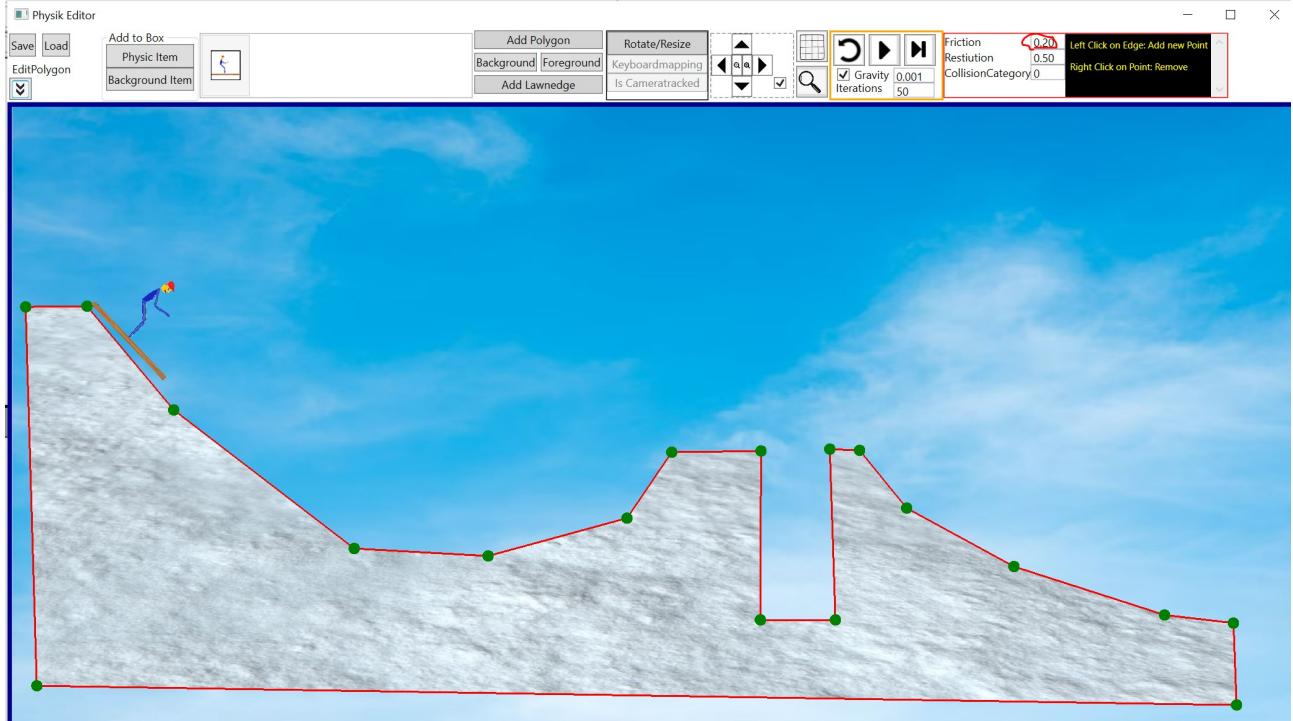


Die zweite Animation steuern wir über die Left- und Right-Taste.

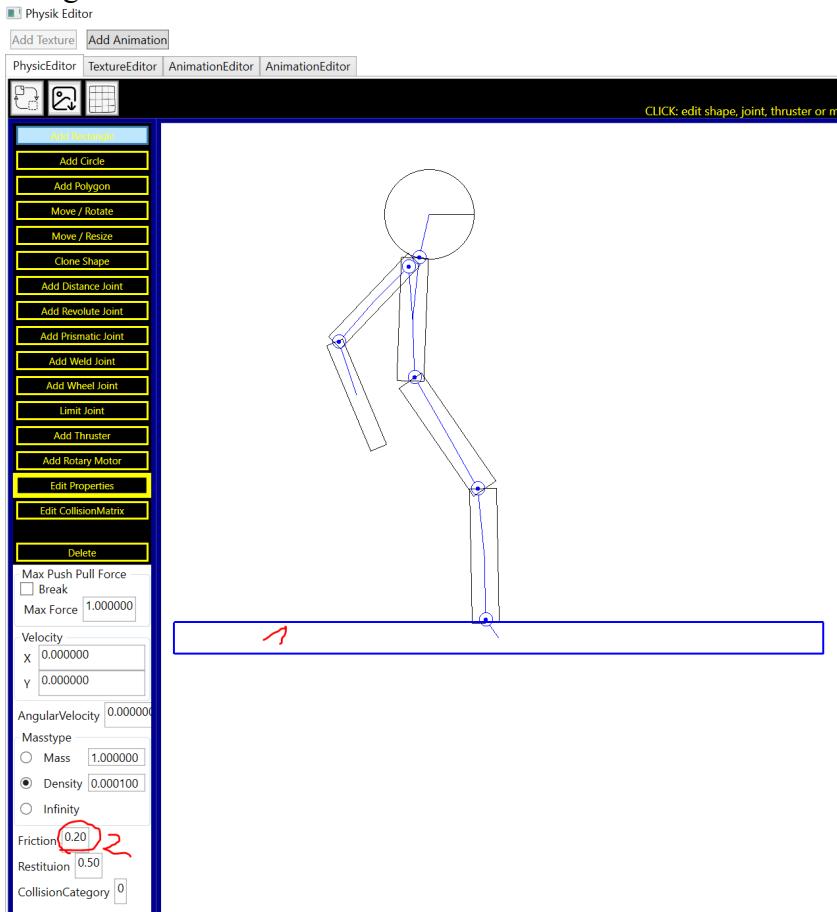
Nun testen wir das Ganze indem wir beim Simulator-Control auf den Play-Button (1) drücken. Wir sehen dann auch links (2) dass der Simulator aktiv ist:



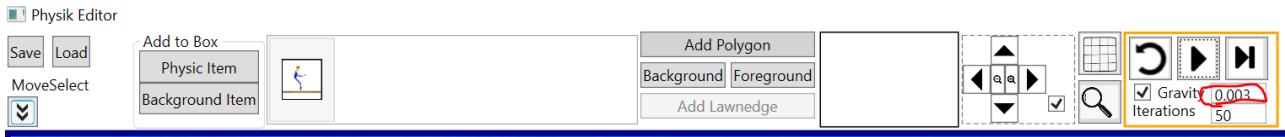
Der Fahrer fährt noch mit zu viel Reibung die Schanze runter also korrigieren wir den Reibungswert indem wir die Schanze anklicken und dann den Frictionwert auf 0 stellen.



Das gleiche können wir auch mit den Skiern machen.



Um dem ganzen mehr Dynamik zu verleihen erhöhen wir die Schwerkraft:



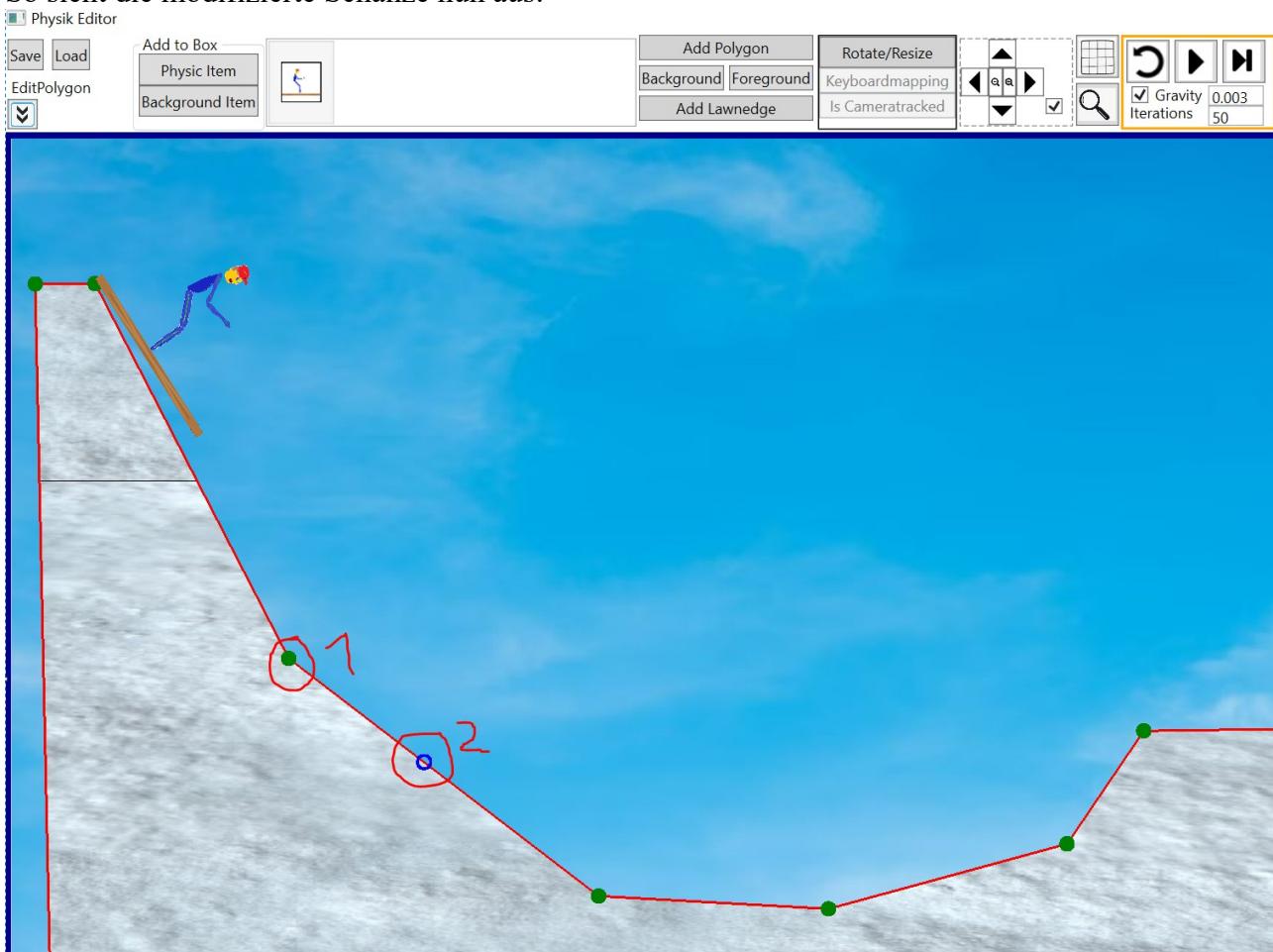
Außerdem können wir die Schanze noch steiler machen, indem wir sie zuerst markieren und dann mit gedrückter Maustaste die Eckpunkte verschieben.

Wenn du mit Links auf ein Eckpunkt klickst, dann kannst du ihn verschieben (1)

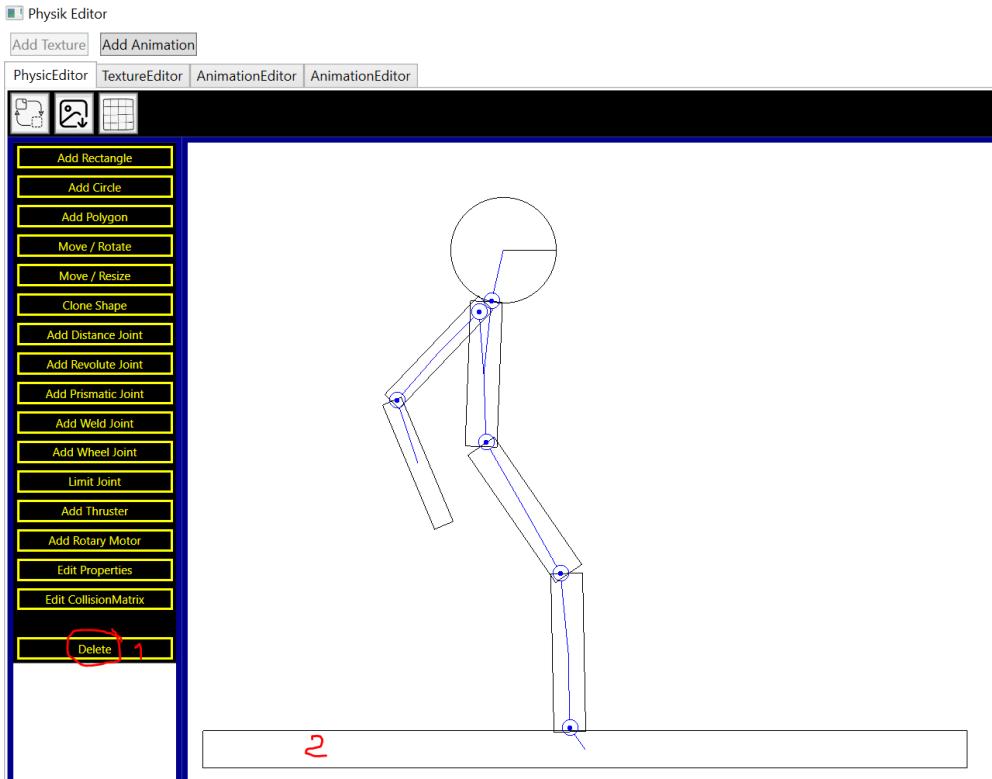
Wenn du mit Links auf eine Kante klickst, kannst du einen neuen Polygonpunkt erzeugen (2)

Wenn du mit Rechts auf ein Polygonpunkt klickst, kannst du ihn löschen.

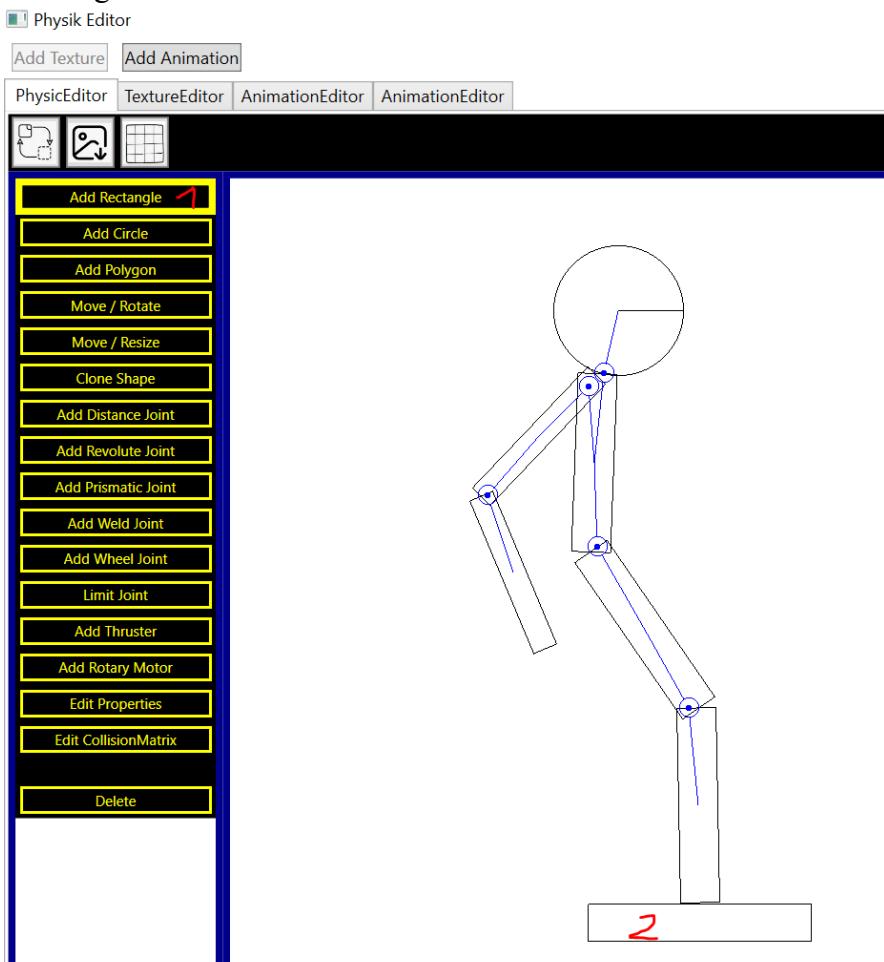
So sieht die modifizierte Schanze nun aus:



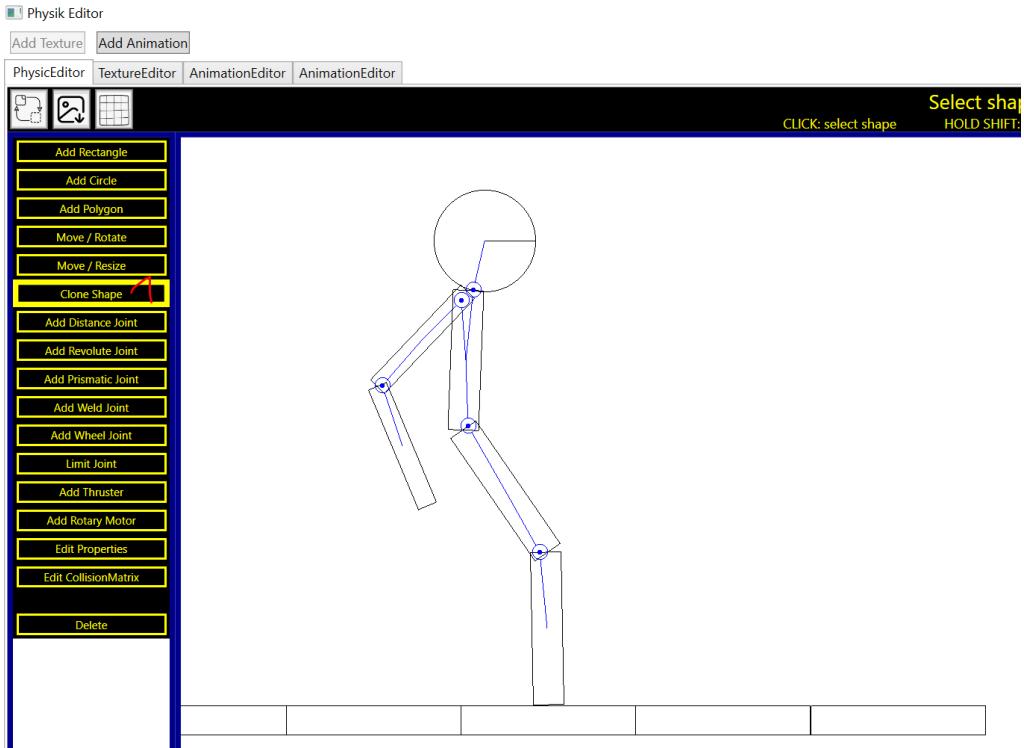
Aktuell sind die Skier noch sehr steif, da wir nur ein Rechteck genommen haben. Wenn wir wollen, dass sie federn, dann können wir sie aus lauter kleineren Rechtecken aufbauen. Dazu editieren wir per Rechtsklick in der Auswahlbox den Fahrer und entfernen seinen Ski:



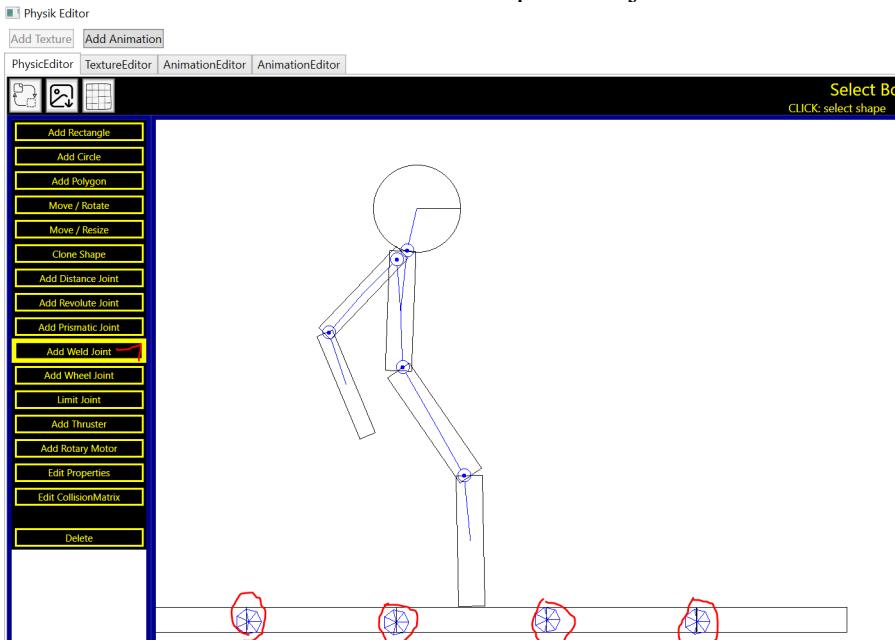
Nun fügen wir ein neues Rechteck ein aber diesmal kürzer:



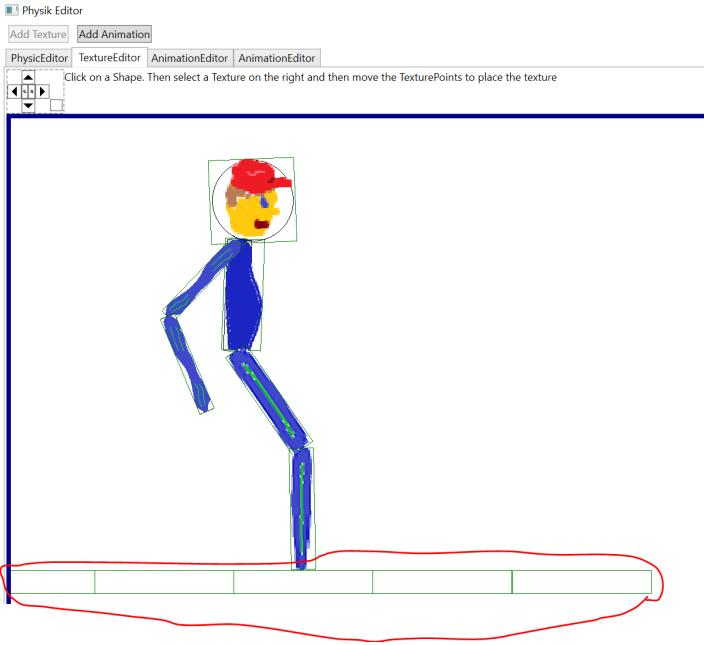
Mit der Clone-Shapefunktion und gedrückter Shift-Taste erzeugen wir nun Kopien von den kleineren Skibrett. Durch die Shift-Taste verschiebt es das neue Objekt immer nur horizontal oder vertikal.



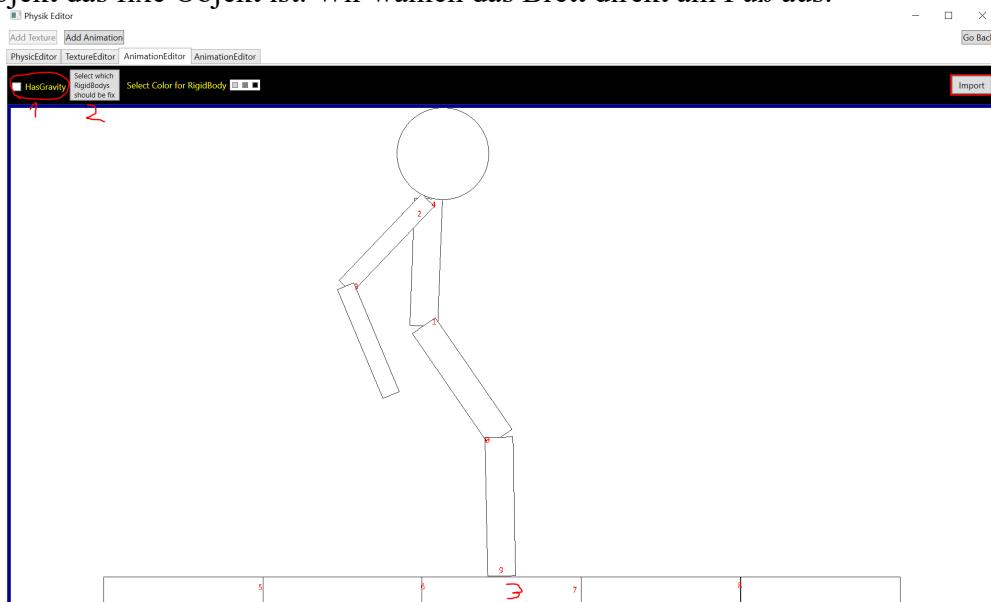
Jetzt müssen wir die Einzelbretter noch per Weldjoints verbinden.



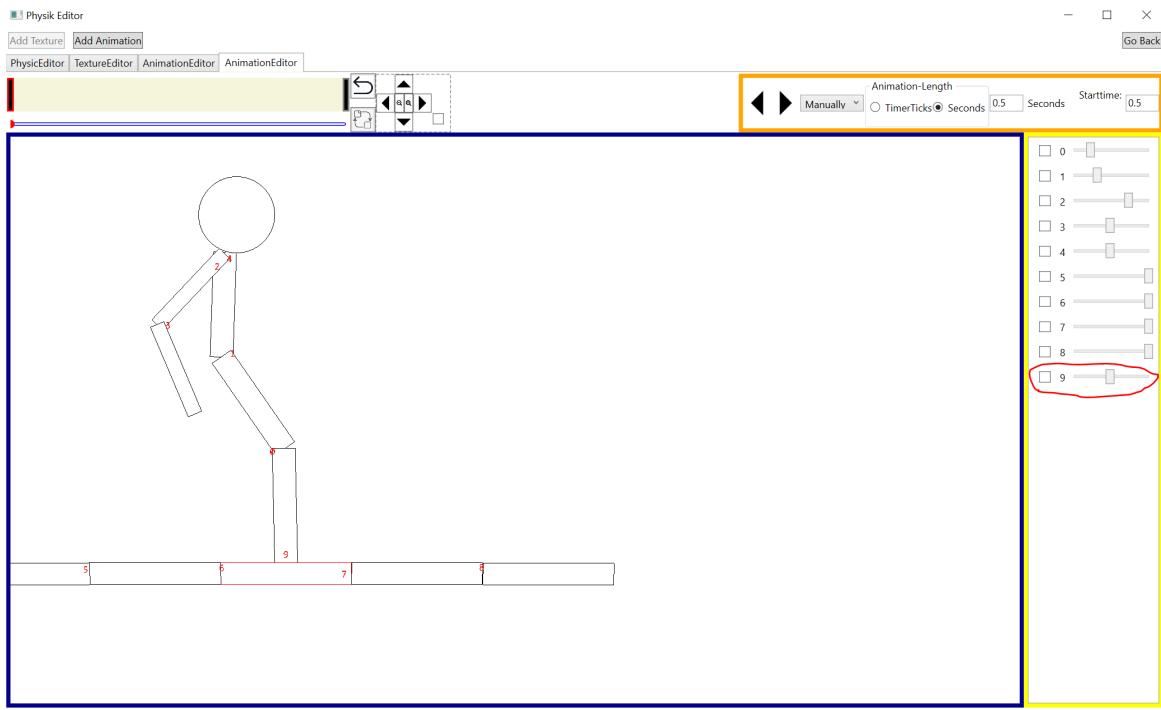
Und der Fuß muss erneut per Revolutejoint an das Brett gebunden werden.
Wenn wir nun in den Textureditor gehen sehen wir, dass wir den neuen Skibrettern noch Texturen verpassen müssen:



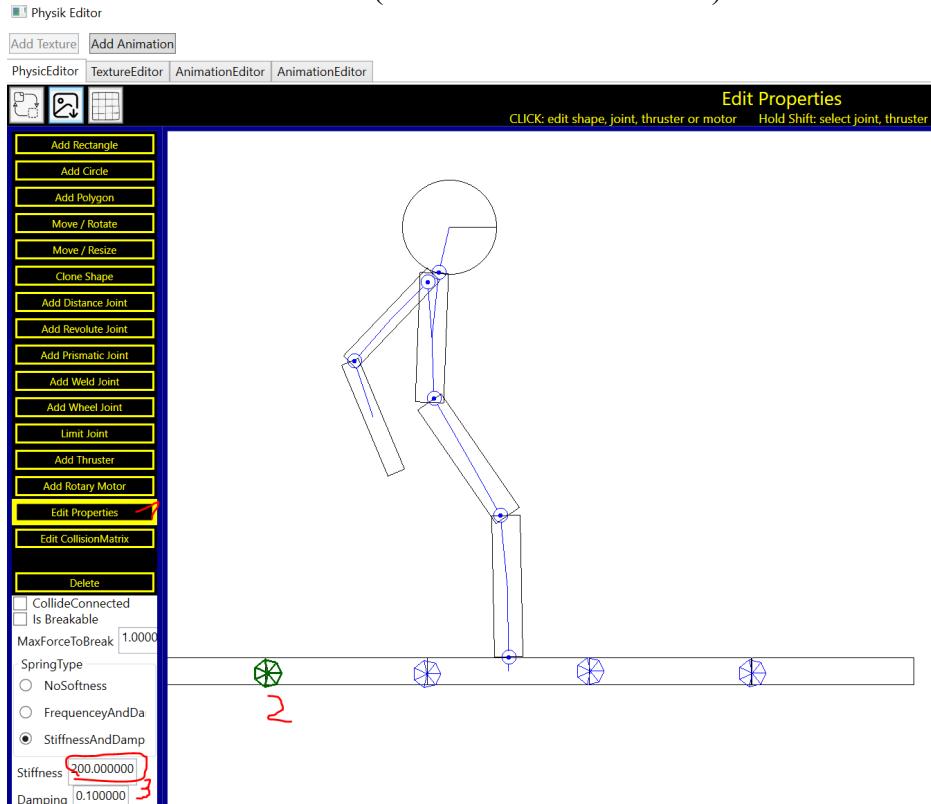
Und weil wir das Physikmodel verändert haben müssen wir beim Animations-Tab erneut festlegen, welches Objekt das fixe Objekt ist. Wir wählen das Brett direkt am Fuß aus:



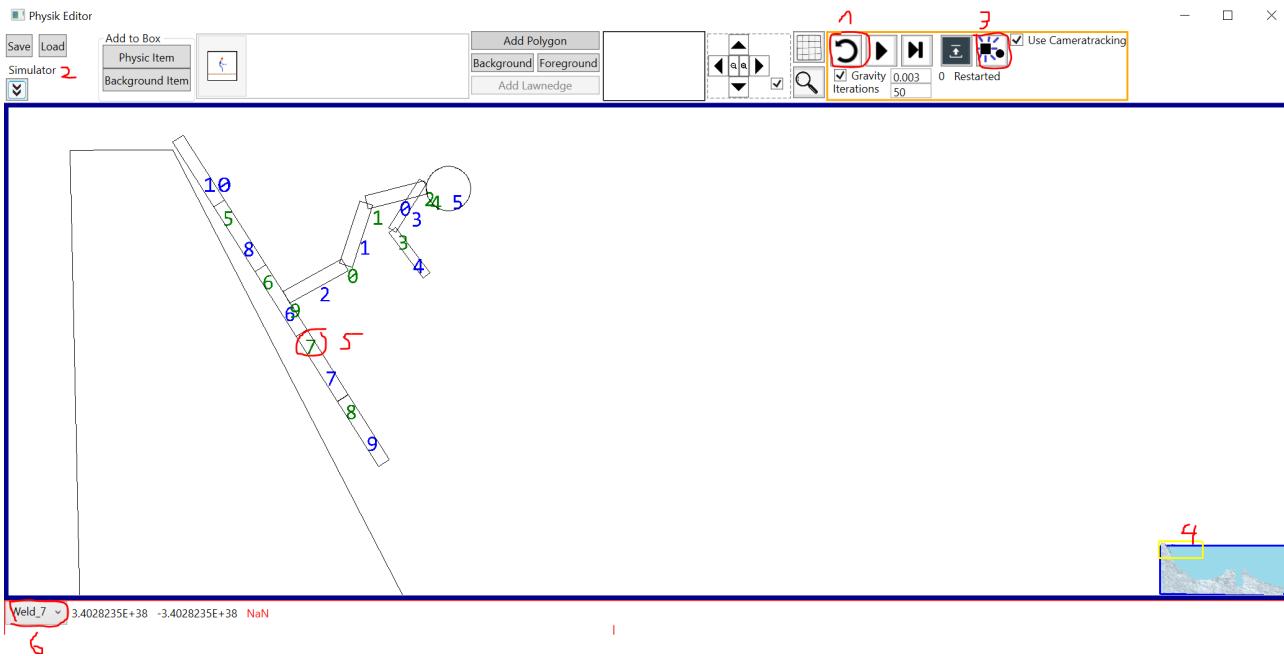
Bei der ersten Animation testen wir und sehen, dass noch alles geht da das Fußgelenk in der ersten Animation nicht mit drin war. Aber bei der zweiten Animation sehen wir, dass auf einmal die Checkbox von den Fußgelenk nicht mehr aktiviert ist. Der Grund dafür ist, weil das alte RevoluteJoint-Gelenk am Fuß gelöscht wurde und durch ein neues ersetzt wurde. Wir aktivieren Gelenk 9 und legen erneut für dieses Gelenk und Start- und Endwert fest.



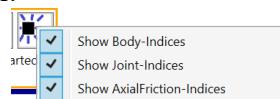
Wenn wir nun diesen Fahrer testen, dann sehen wir, dass der Ski nun elastisch ist. Man kann die Elastizität auch einstellen, indem wir den Stiffnesswert der Weldjoints ändern. Mehr Stiffness bedeutet der Ski wird härter (mehr wie ein steifes Brett).



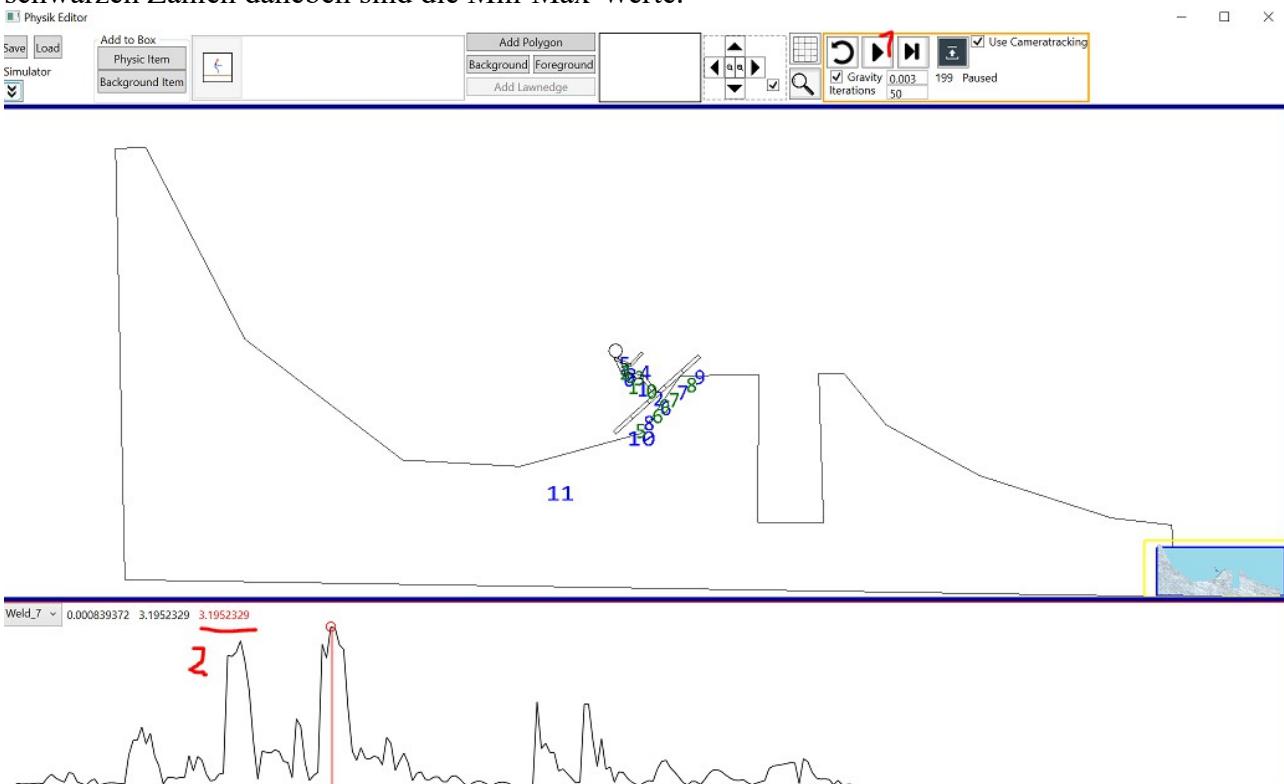
Wir wollen nun die Skiebretter noch zerbrechlich gestalten. Dazu aktivieren wir zuerst den Simulator (1) und sehen das er aktiv ist (2). Dann aktivieren wir den ForceTracker (3) und zoomen ganz nah an den Skifahrer (4) um all seine Gelenknummern (grüne Zahlen) und Rechtecke (blaue Zahlen) zu sehen. Das Weld-Joint mit der Nummer 7 soll zerbrechlich werden. Zuerst wählen wir in der Comobox (6) dieses Gelenk aus. Ab jetzt werden die Kräfte, die auf dieses Gelenk wirken im weißen unteren Bildbereich angezeigt.



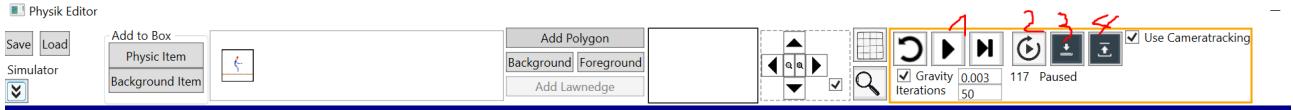
Hinweis: Man kann über Das ContextMenü vom ForceButton sagen, welche Force-Indizes im Hauptbildschirm angezeigt werden soll:



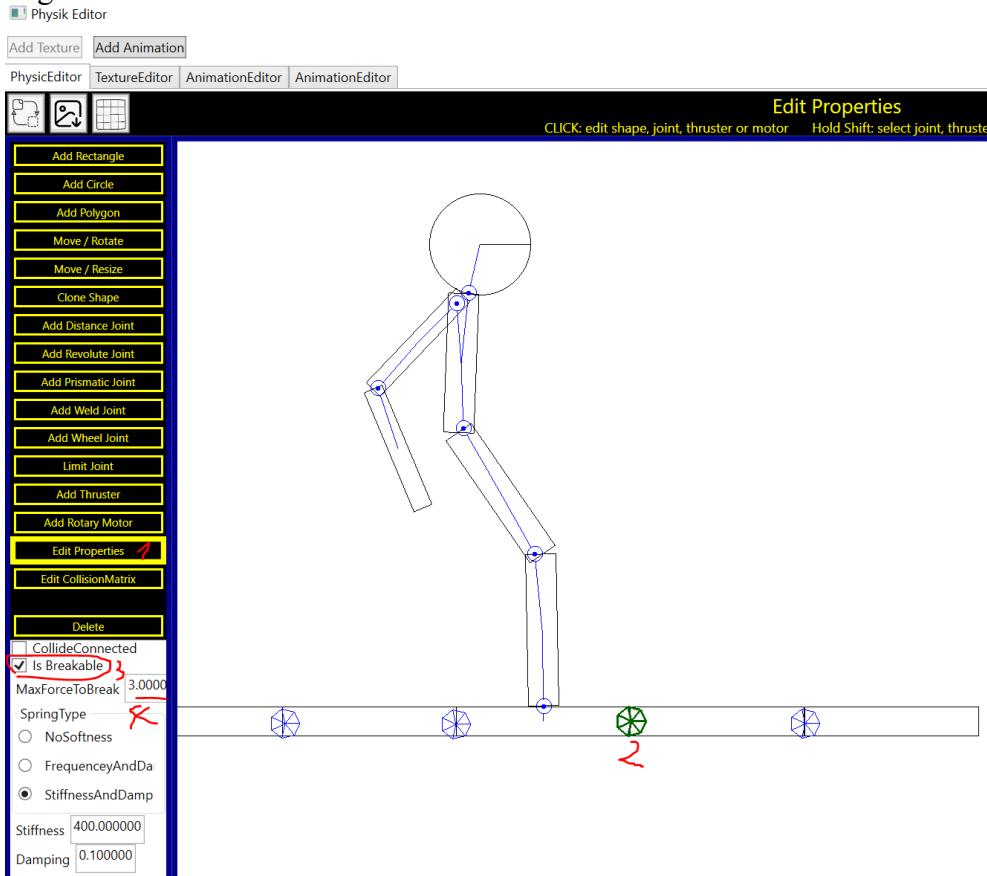
Wir starten nun die Simulation und sehen nun, welche Kräfte zu welchen Zeitpunkt auf das ausgewählte Gelenk „Weld_7“ gewirkt haben. Wir können mit der Maus eine Stelle markieren und sehen dann bei der roten Zahlen, welche Kraft zu diesen Zeitpunkt gewirkt hat. Die beiden schwarzen Zahlen daneben sind die Min-Max-Werte.



Ein Weg, um zu ermitteln, welche Kraftwerte ok sind und was nicht ist, indem man ein paar Sprünge macht und sich die Aufzeichnung speichert. Dazu muss man erst die Simulation laufen lassen (1) und mit den Pfeiltasten den Springer steuern. Dann drückt man auf Stop (dort wo der Playbutton (1)) ist und dann kann man mit (2) die Simulation wiederholen oder man speichert die aufgezeichneten Tastendrücke (3) und kann sie dann mit (4) auch wieder laden um sie dann mit (2) abspielen zu lassen.



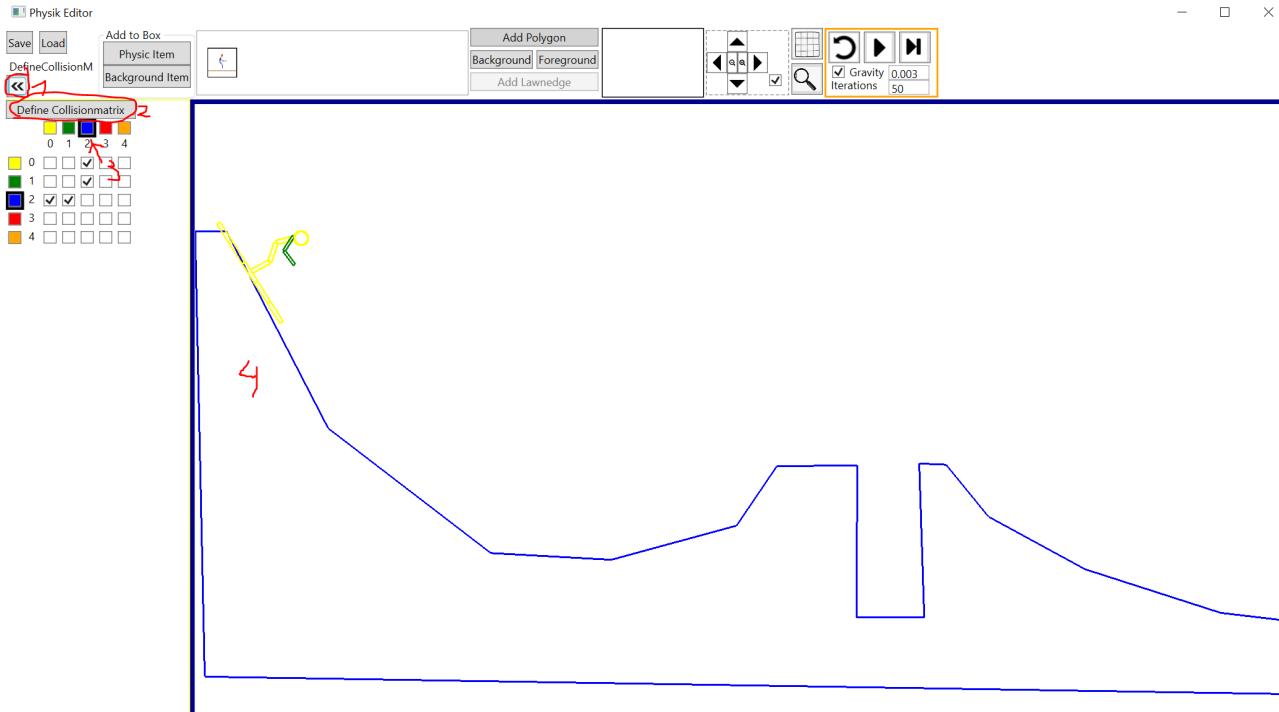
Wir haben nun durch den Forcetracker ermittelt, dass z.B. die Spitzenlast auf das Gelenk Nummer 7 3,19 war und wir wollen, dass es aber bei einer Kraft von 3 bricht. Also editieren wir den Skifahrer und stellen beim Gelenk ein, dass es „Breakable“ ist und dass der MaxForceToBreak-Wert bei 3 liegt:



Und schon bricht der Ski wenn die Kraft zu groß wird:



Damit die Arme des Fahrers nicht gegen sein Körper stoßen hatten wir ihr eine andere Kollisionskategorie gegeben. Allerdings führt das nun dazu, dass die Arme auch nicht mit der Skischranze kollidieren können. Damit der gesamte Fahrer mit der Schanze kollidieren kann müssen wir im Editor die Kollisionskategorie von der Schanze auf blau ändern und dann sagen wir es soll Gelb mit Blau und Grün mit Blau kollidieren:



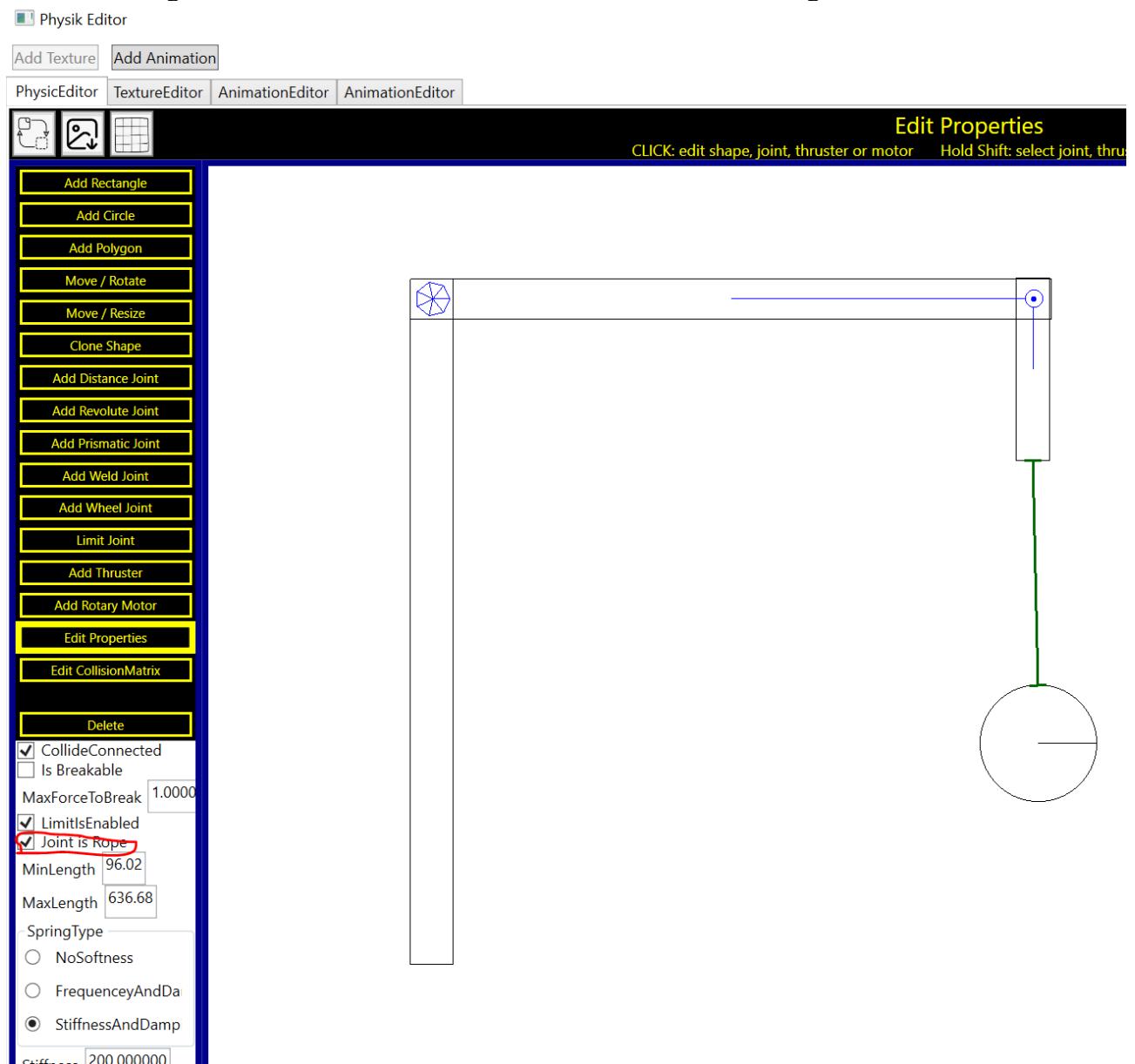
Beispiel 2: Die Abrisskugel

In diesen Beispiel soll eine Kugel simuliert werden, die an ein Seil hängt und dann gegen ein Stapel Kisten fliegt.

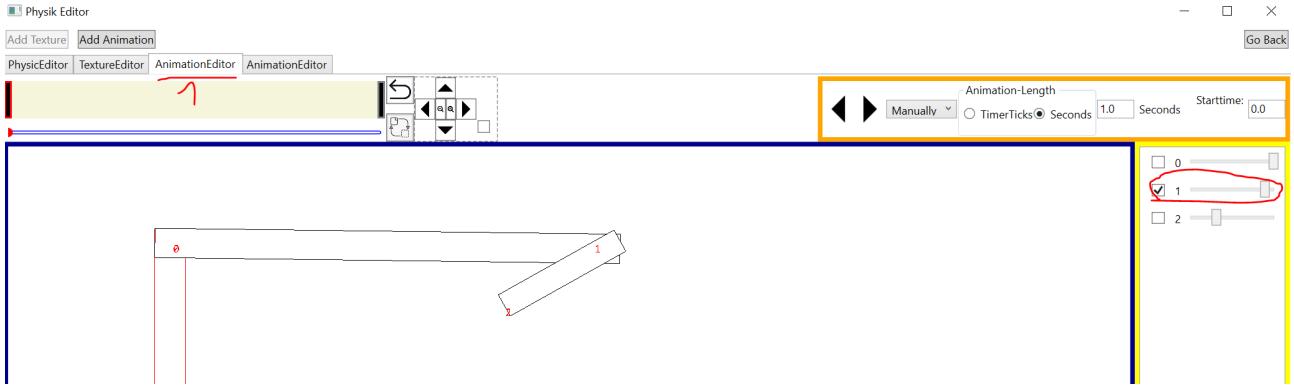
Wir erzeugen dazu zuerst die Abrisskugel indem wir auf „Physic Item“ gehen:



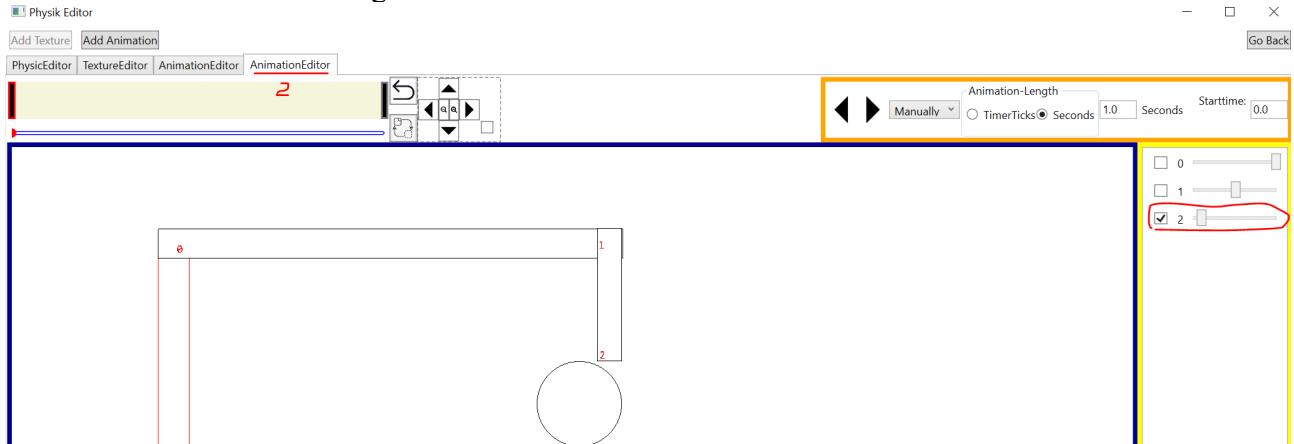
Aus Rechtecken und ein Kreis bauen wir das Abrissgerät. Die Kugel hängt per Distance-Joint am Rechteck und über die JointIsRope-Option verhält sich das Distanzjoint so wie ein Seil, dessen Länge man einstellen kann. Würde man diese Option auf False stellen, dann wäre das Distanzjoint eine Eisenstange, die auch dann Widerstand leistet, wenn es zusammen gedrückt wird.



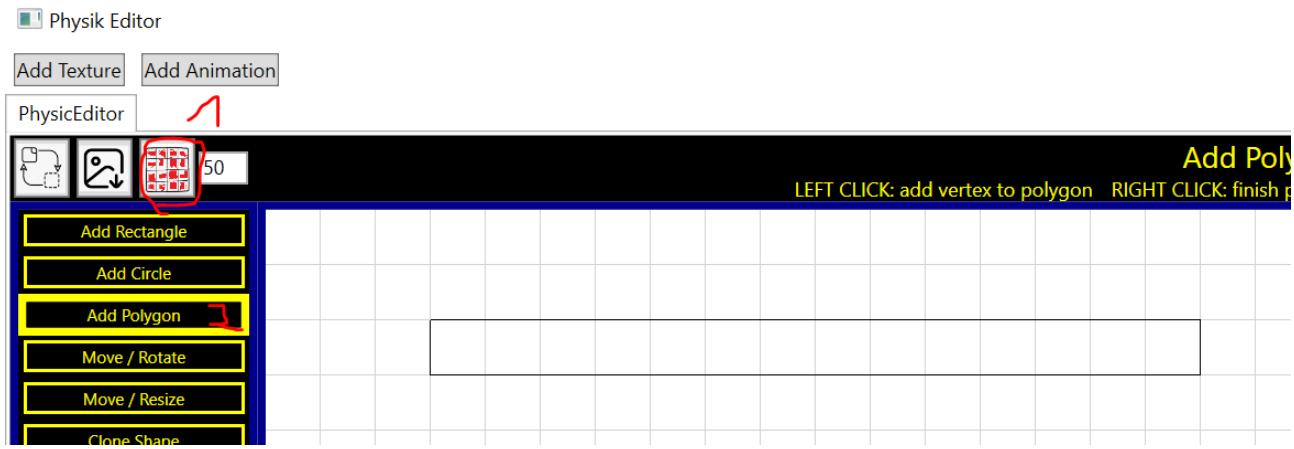
Animation 1 steuert den Arm wo die Kugel dranhängt:



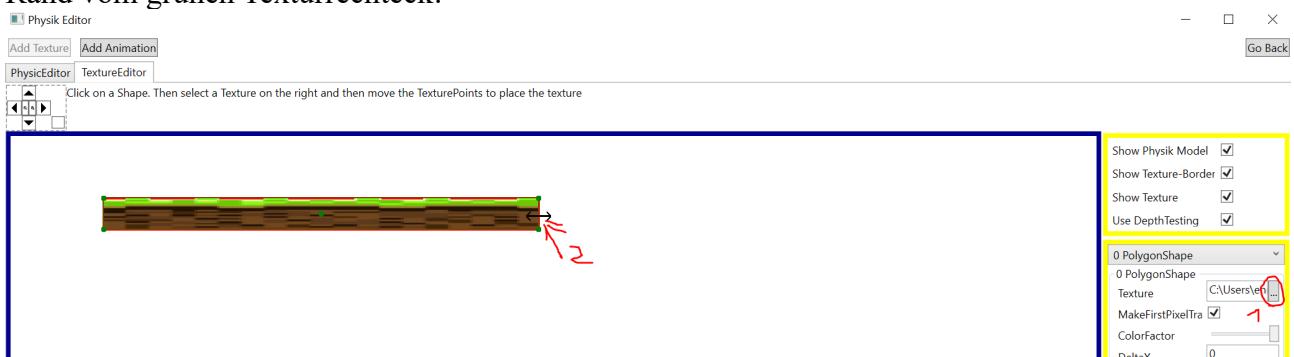
Und Animation 2 die Seillänge:



Nun erzeugen wir ein neues Physik-Objekt was unseren Fußboden darstellen soll. Wir aktivieren den Grid-Mode und erzeugen ein Polygon in der Form eines länglichen Rechtecks:

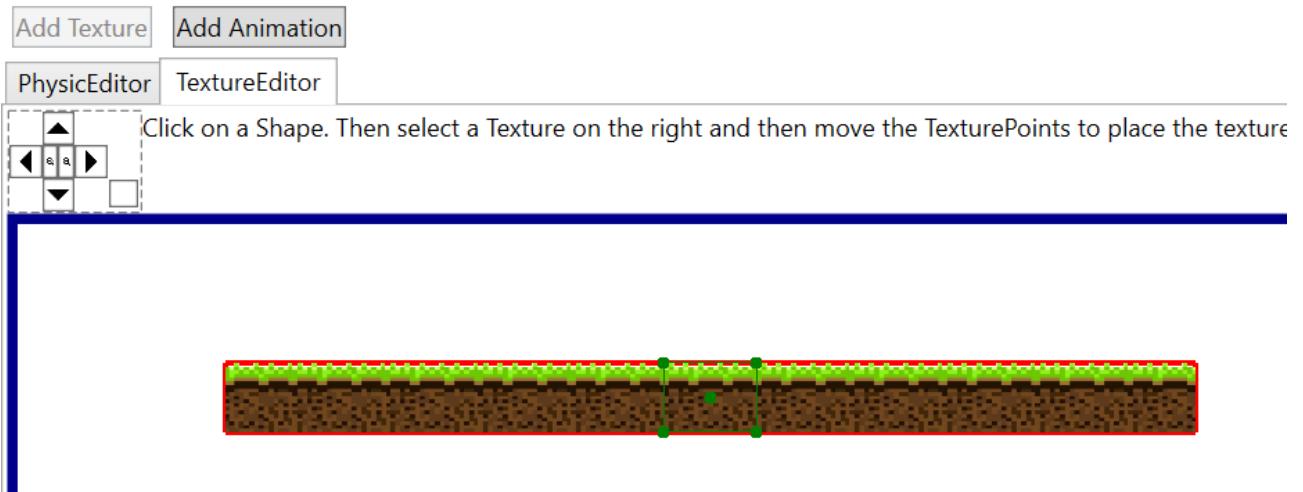


Nun wählen wir für dieses Polygon eine Grastextur aus und gehen mit der Maus an den rechten Rand vom grünen Texturechteck:

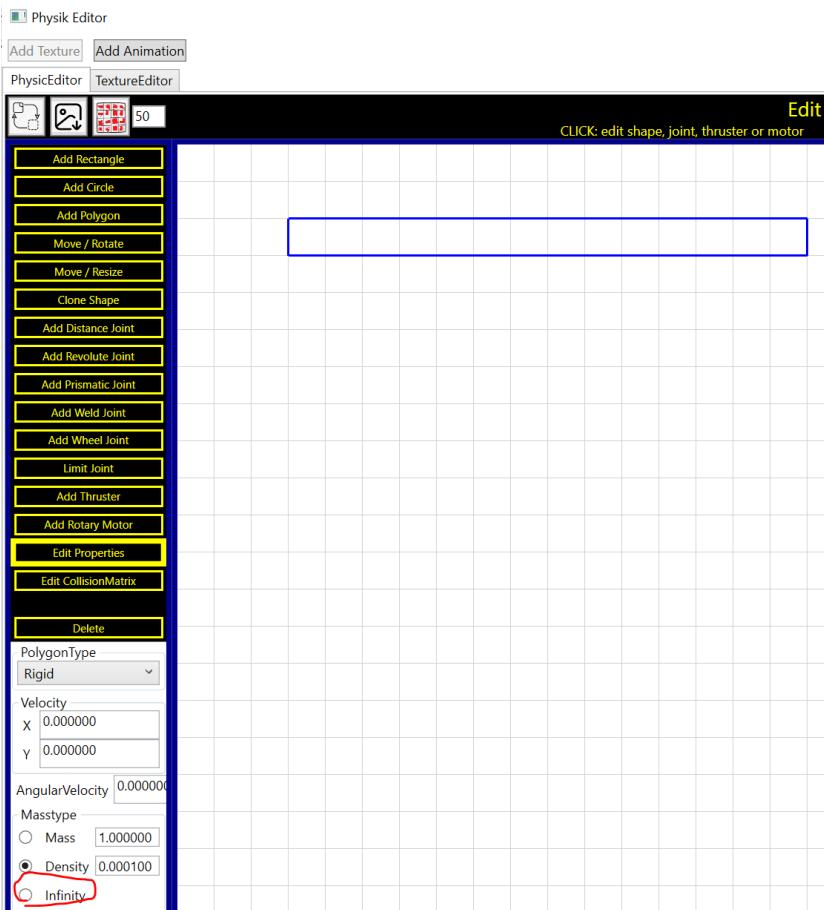


Wir machen das Rechteck nun schmäler was dazu führt, dass die Textur sich mehrmals wiederholt:

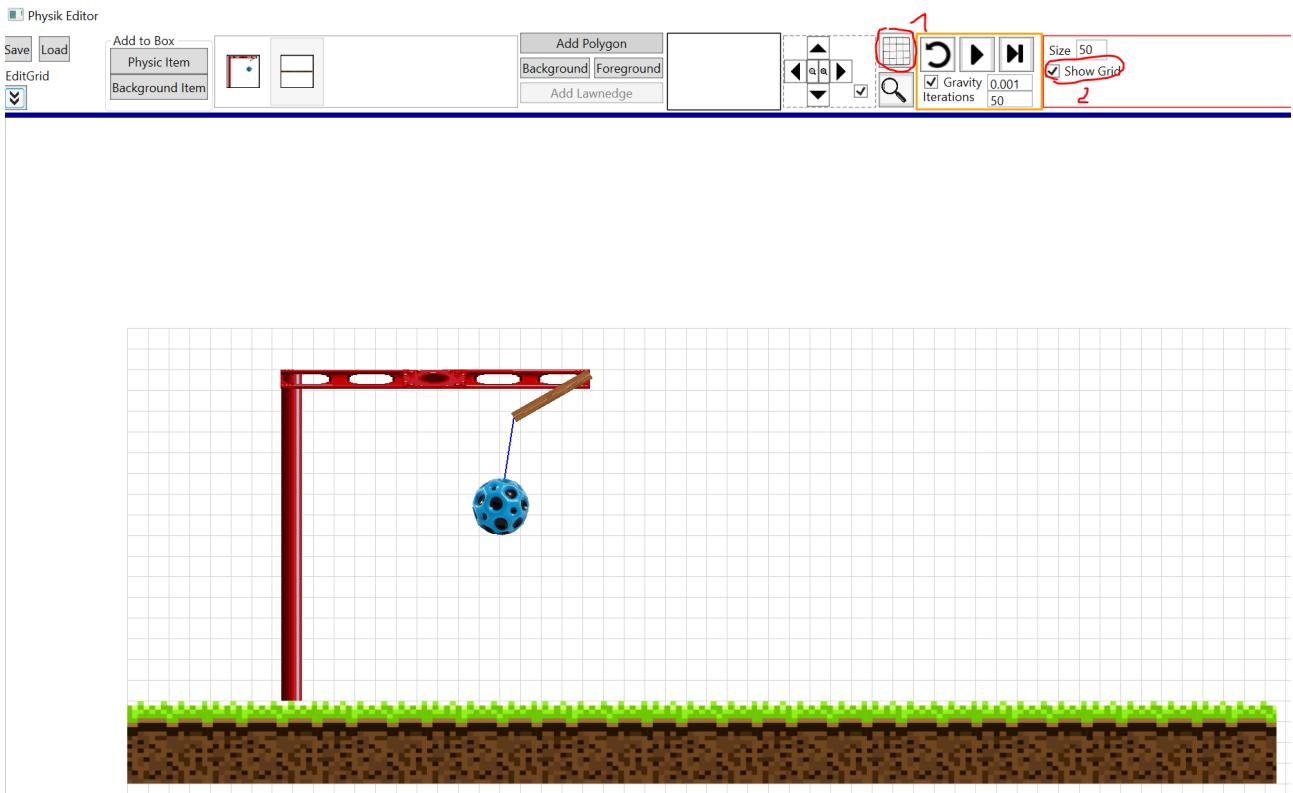
 Physik Editor



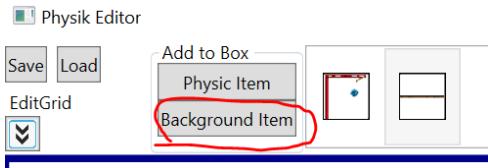
Dieser Textur-Kacheleffekt geht nur mit texturierten Physik-Polygonen weswegen wir diesmal nicht das Levelrand-Polygon genommen haben. Damit der Boden starr ist müssen wir seine Masse noch auf Unendlich stellen:



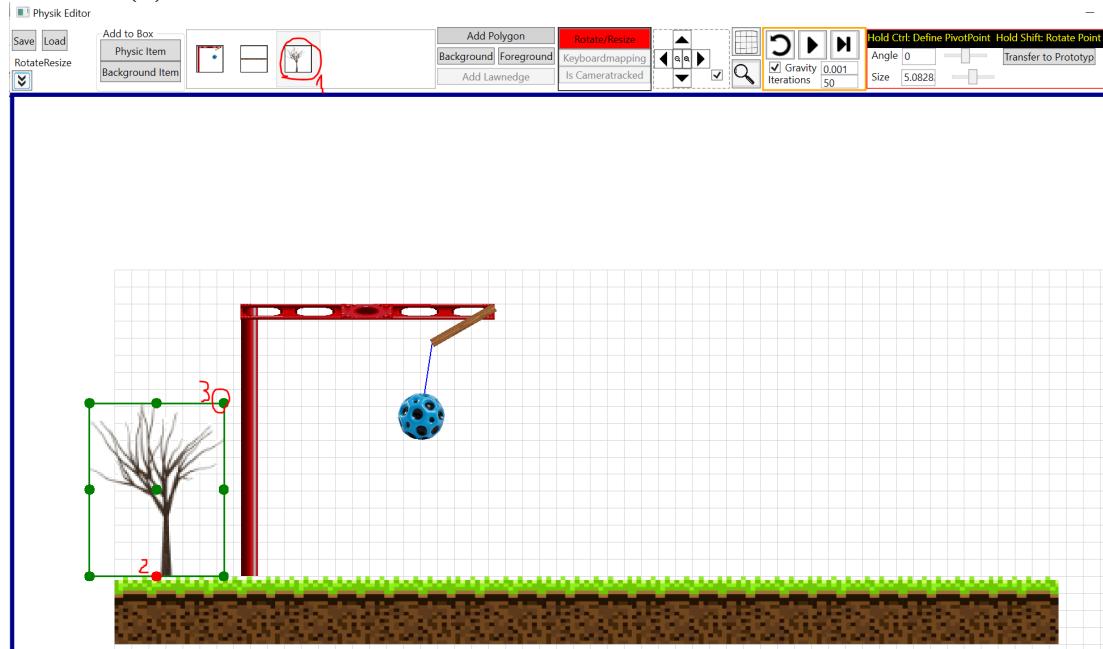
Wir aktivieren im Editor nun den Grid-Mode und platzieren unsere beiden Objekte:



Damit der Hintergrund nicht so leer ist platzieren wir ein Hintergrundobjekt:

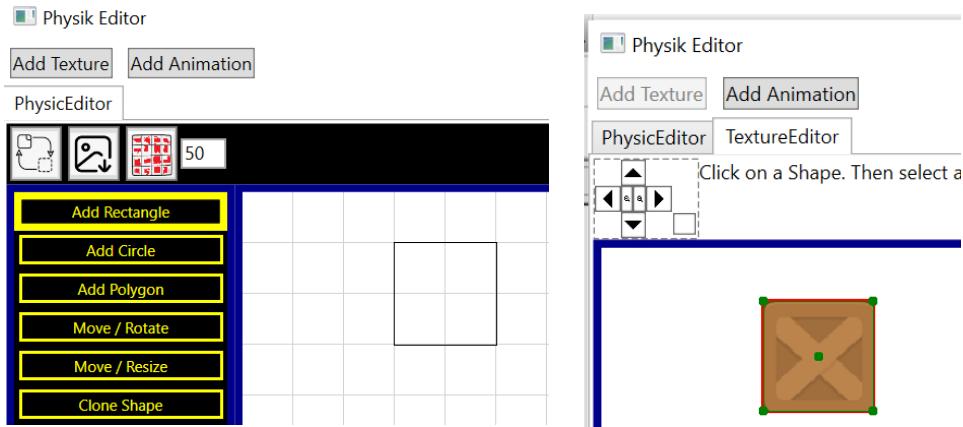


Wir ziehen das Objekt von der Box oben in den Editor und legen den unteren mittleren Punkt als Pivot-Punkt fest. So können wir den Baum exakt auf dem Boden platzieren. Nun vergrößern wir den Baum noch (3).

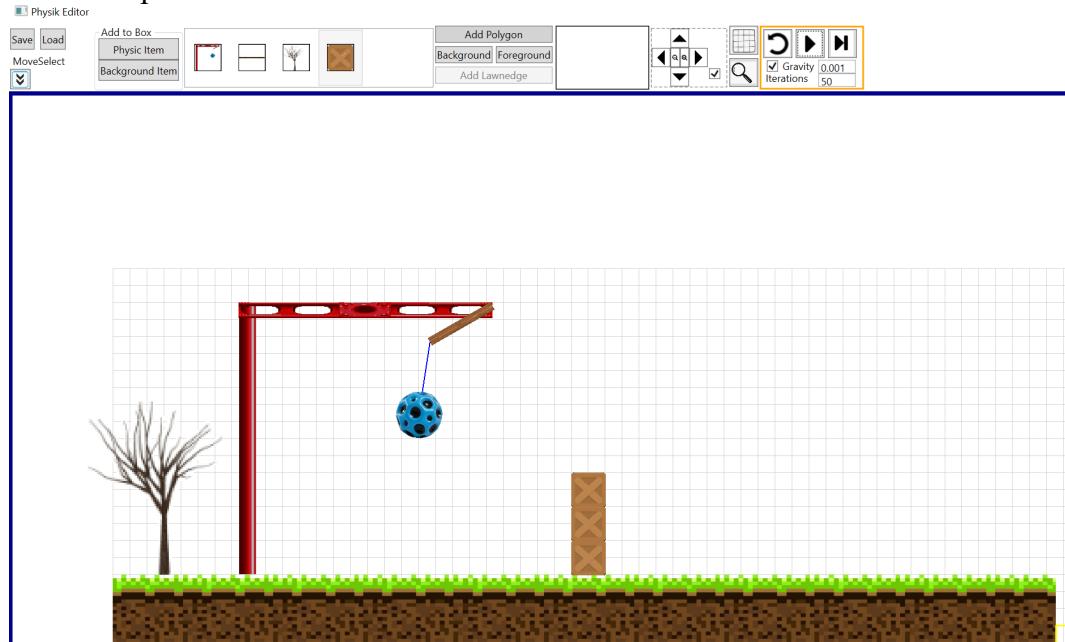


Jetzt brauchen wir noch etwas, was wir mit der Kugel umstoßen können.

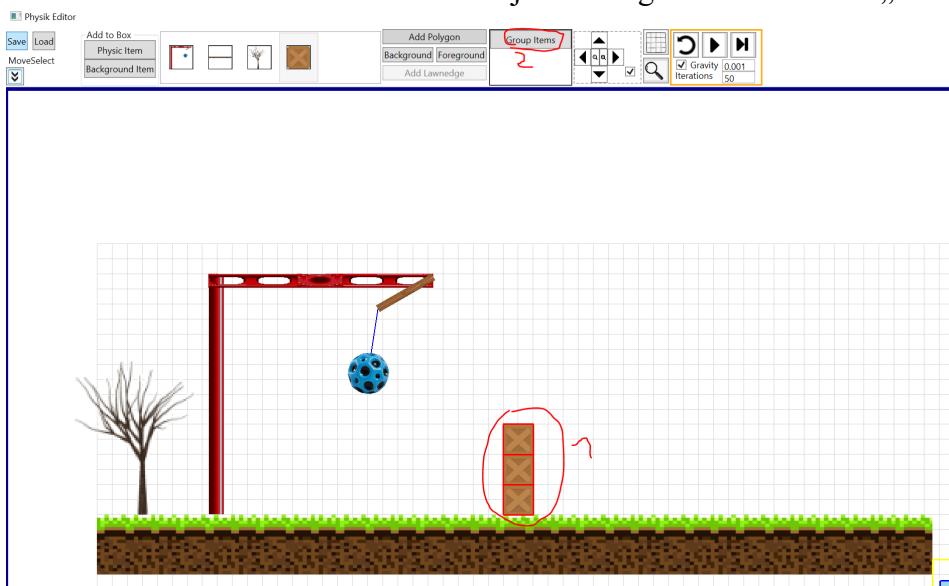
Dazu erschaffen wir ein Physik-Rechteck und geben ihm eine Textur:



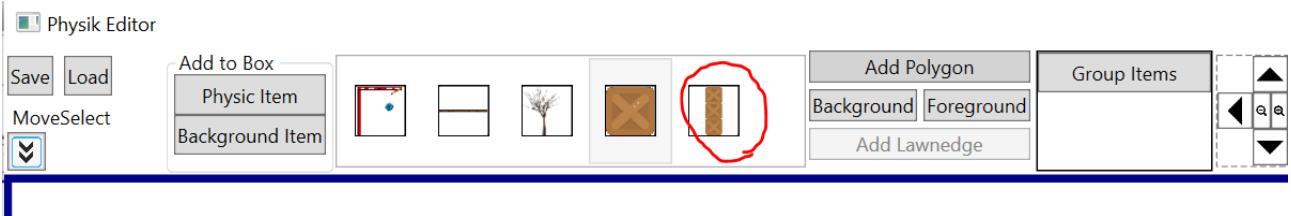
Und platzieren ein paar Kisten davon im Level:



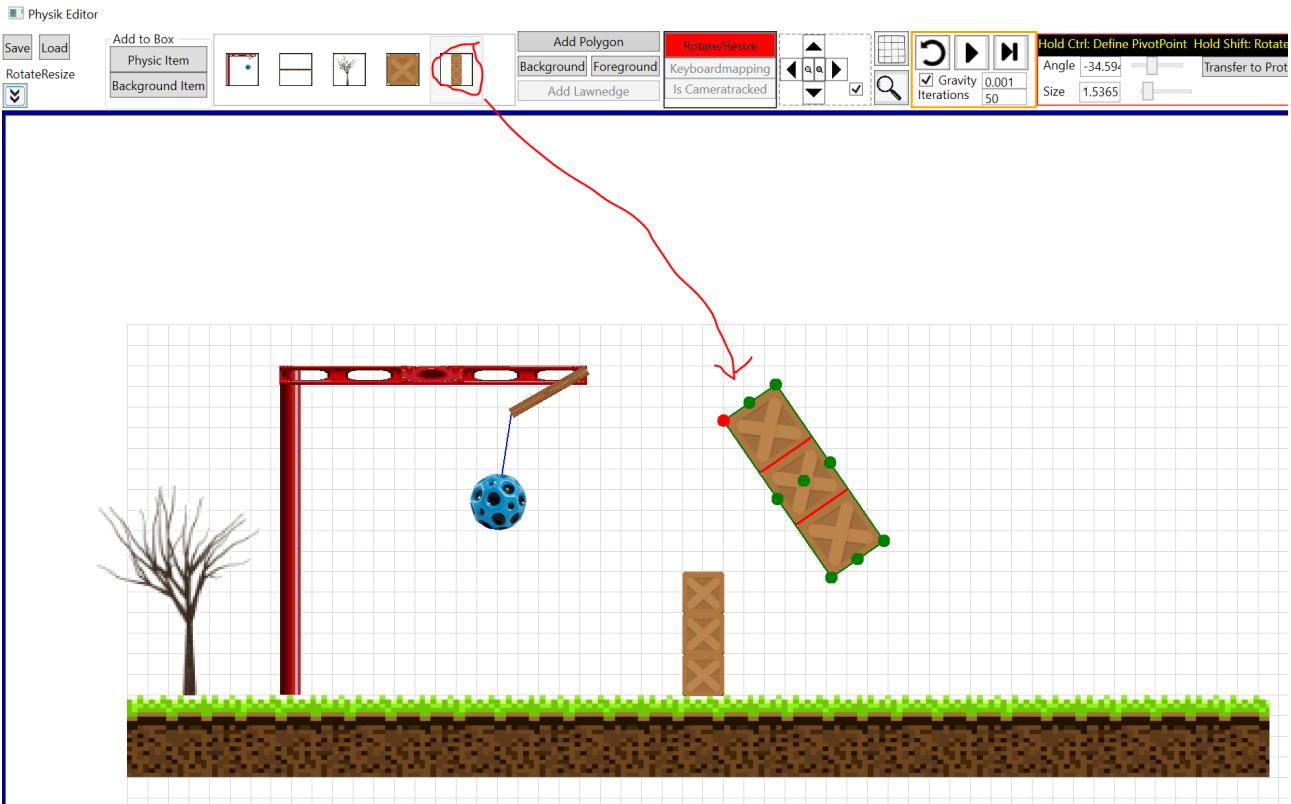
Mit gedrückter Maustaste können wir ein Selektionsrechteck aufspannen und alle Rechtecke markieren und daraus dann ein neues Objekt erzeugen indem wir auf „Group Items“ klicken:



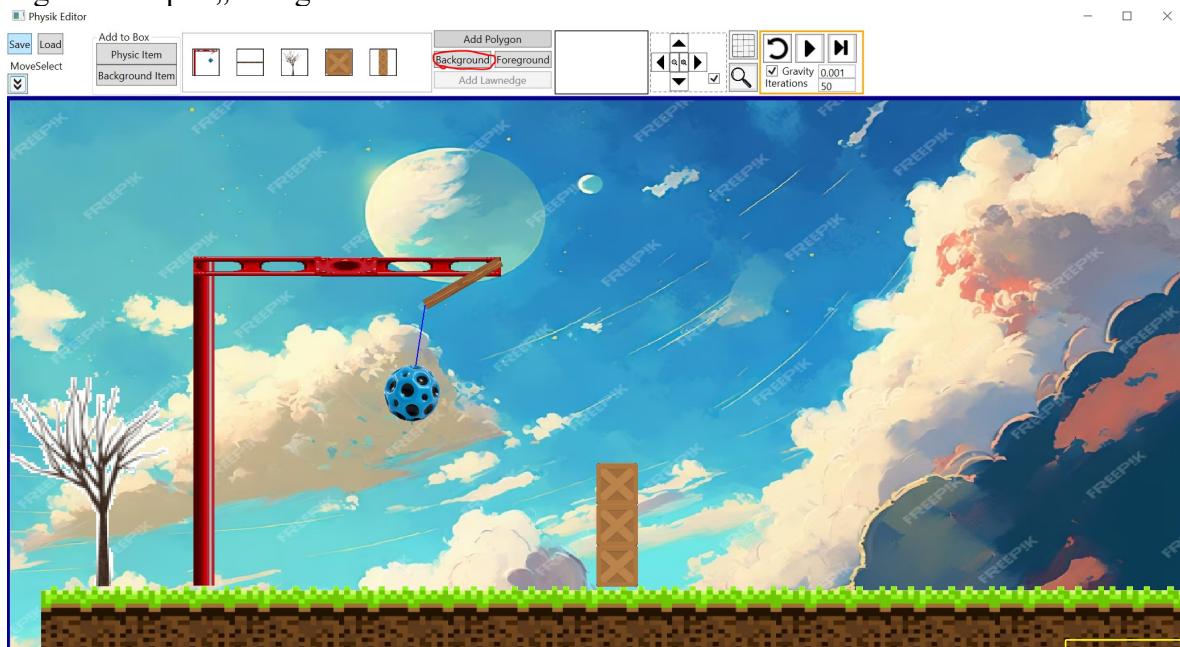
Dadurch erscheinen dann die 3 Boxen in der Auswahlbox:



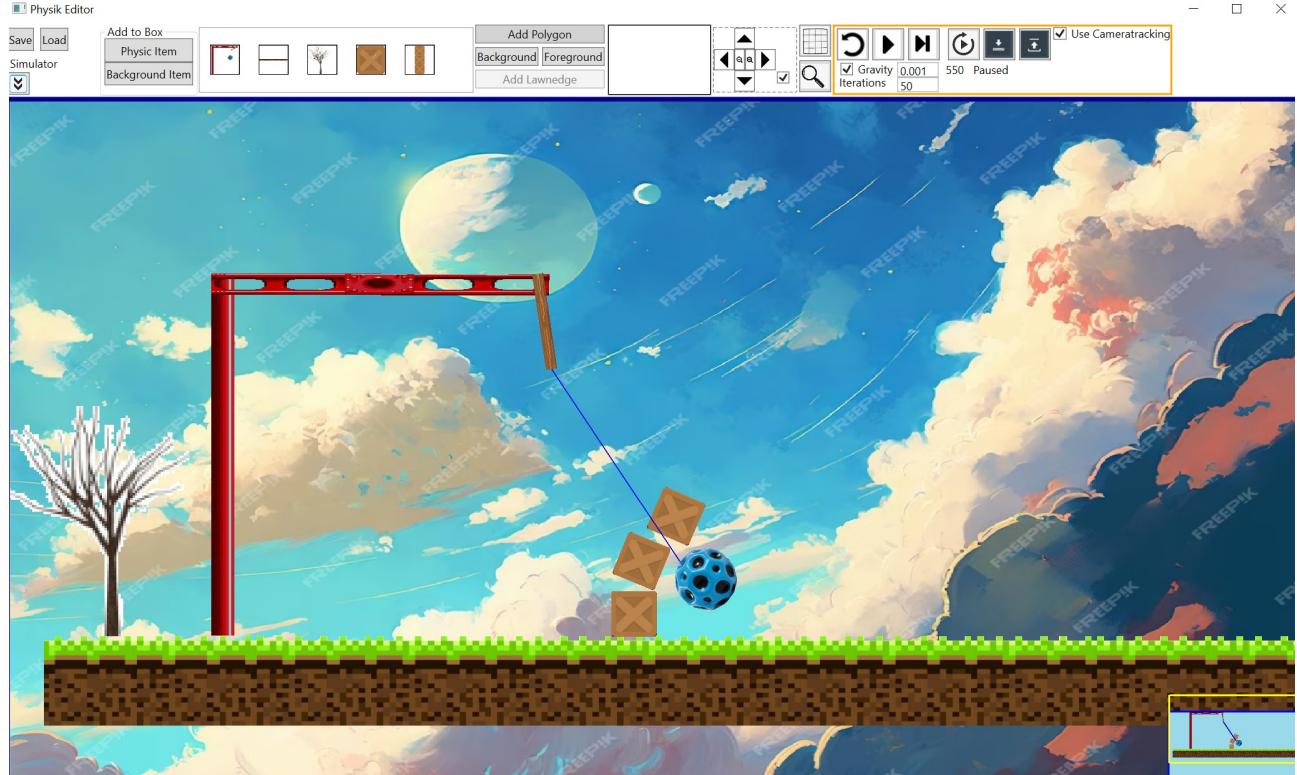
Wenn ich dieses gruppiert Objekt dann in den Editor ziehe, erscheint es wie als ob es ein einzelnes Objekt ist. Ich kann es dann auch drehen und die Größe ändern.



Wir definieren für den Arm Keyboardmapping, indem wir ihn selektieren und dann die Pfeiltasten zur Steuerung der Animation nutzen. Das Vorgehen ist wie beim Skifahrer auch. Dann noch das Hintergrundbild per „Background“ ändern:

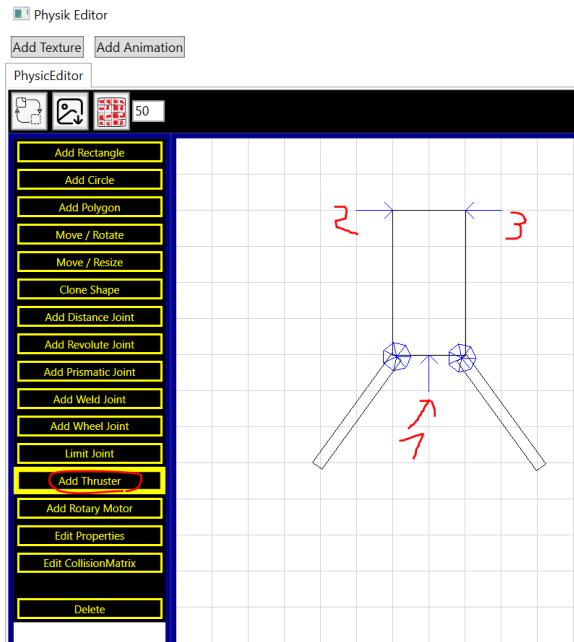


Schon haben wir die Möglichkeit die Kisten mit der Kugel umzustoßen:

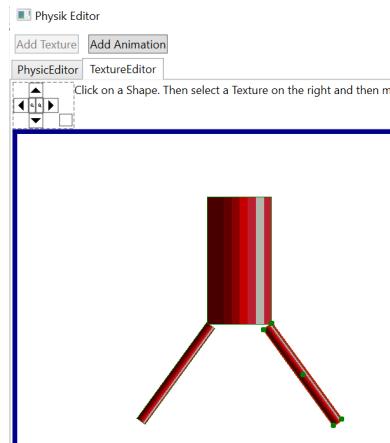


Beispiel 3: Das Raumschiff

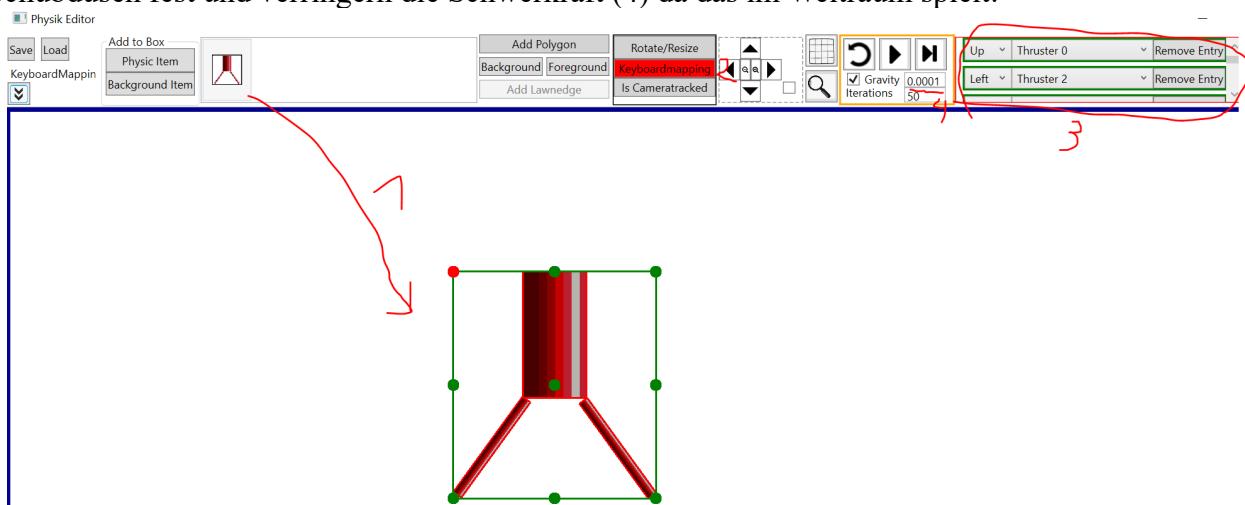
Wir erzeugen diesmal ein Raumschiff indem wir 3 Rechtecke nehmen, welche per Weldjoint verbunden sind. Per Schubdüse können wir dann das Schiff bewegen. Der blaue Schubdüsenpfeil zeigt in die Richtung, in die er dann drückt.



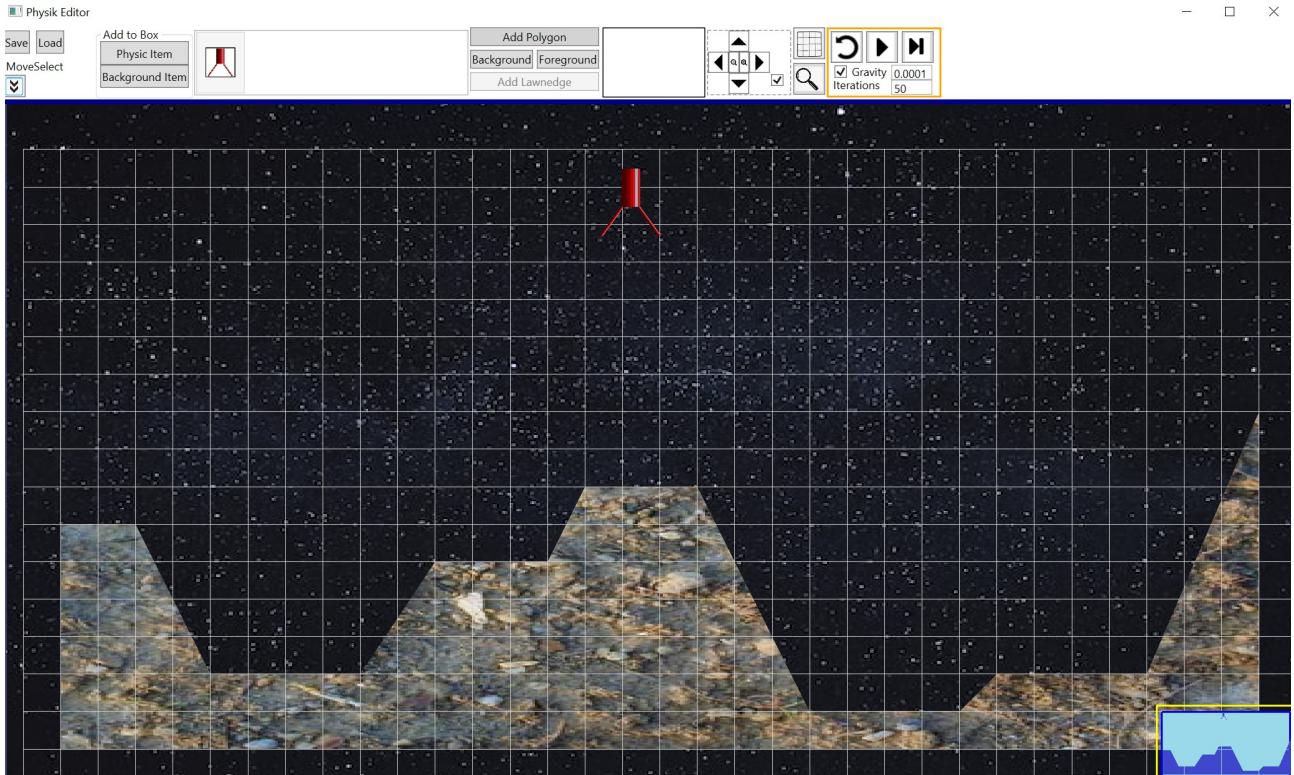
Jetzt noch eine Textur festlegen:



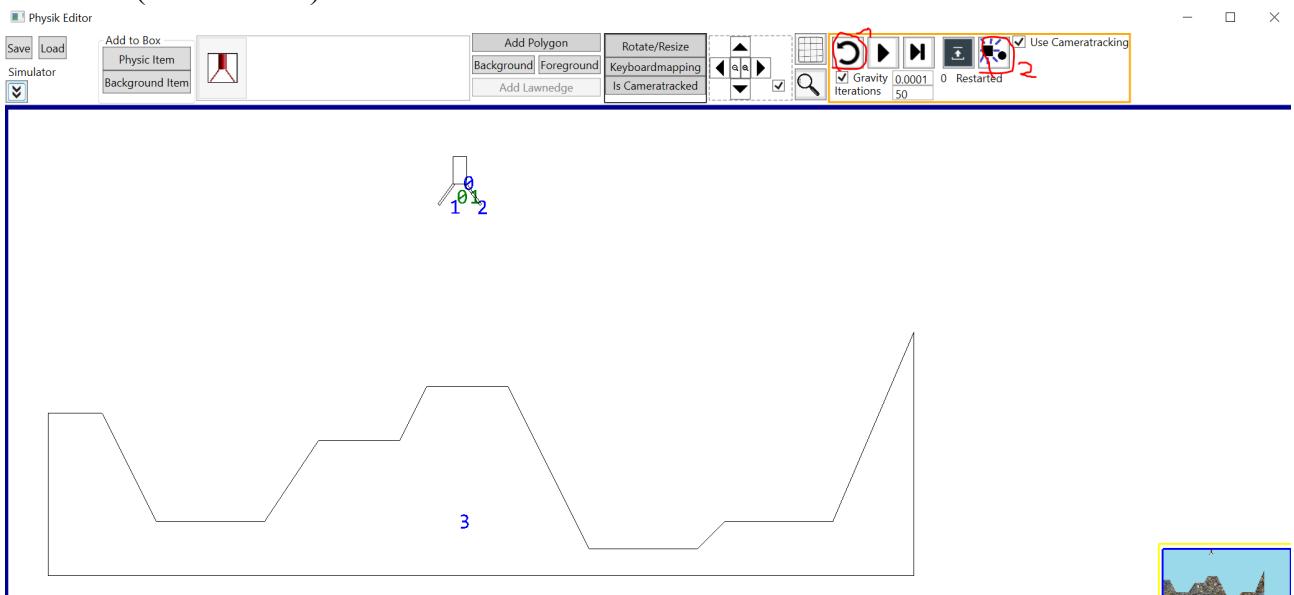
Wir platzieren das Raumschiff im Editor (1) und legen Keyboardmapping (2/3) zur Steuerung der Schubdüsen fest und verringern die Schwerkraft (4) da das im Weltraum spielt:



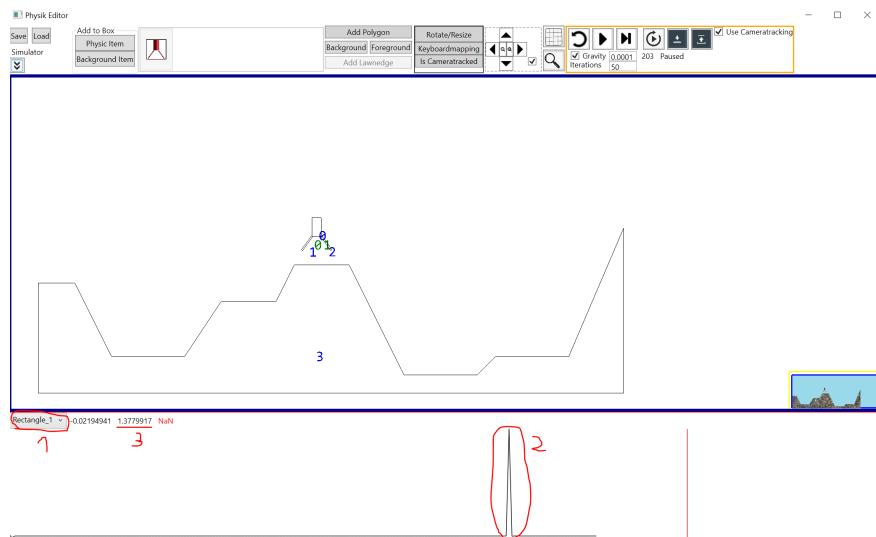
Wir zoom nun raus, aktivieren den Gridmode und erzeugen per „Add Polygon“ die Mondoberfläche:



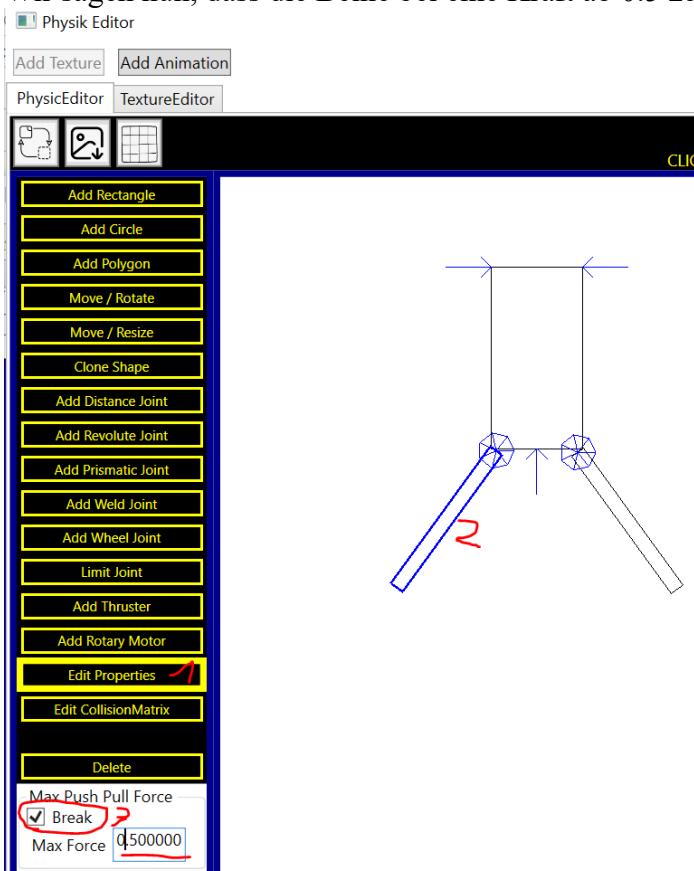
Wir aktivieren den Forcetracker (im Simulator) und wollen diesmal untersuchen, welche Kräfte auf die Beine (Blau 1 und 2) des Landers wirken:



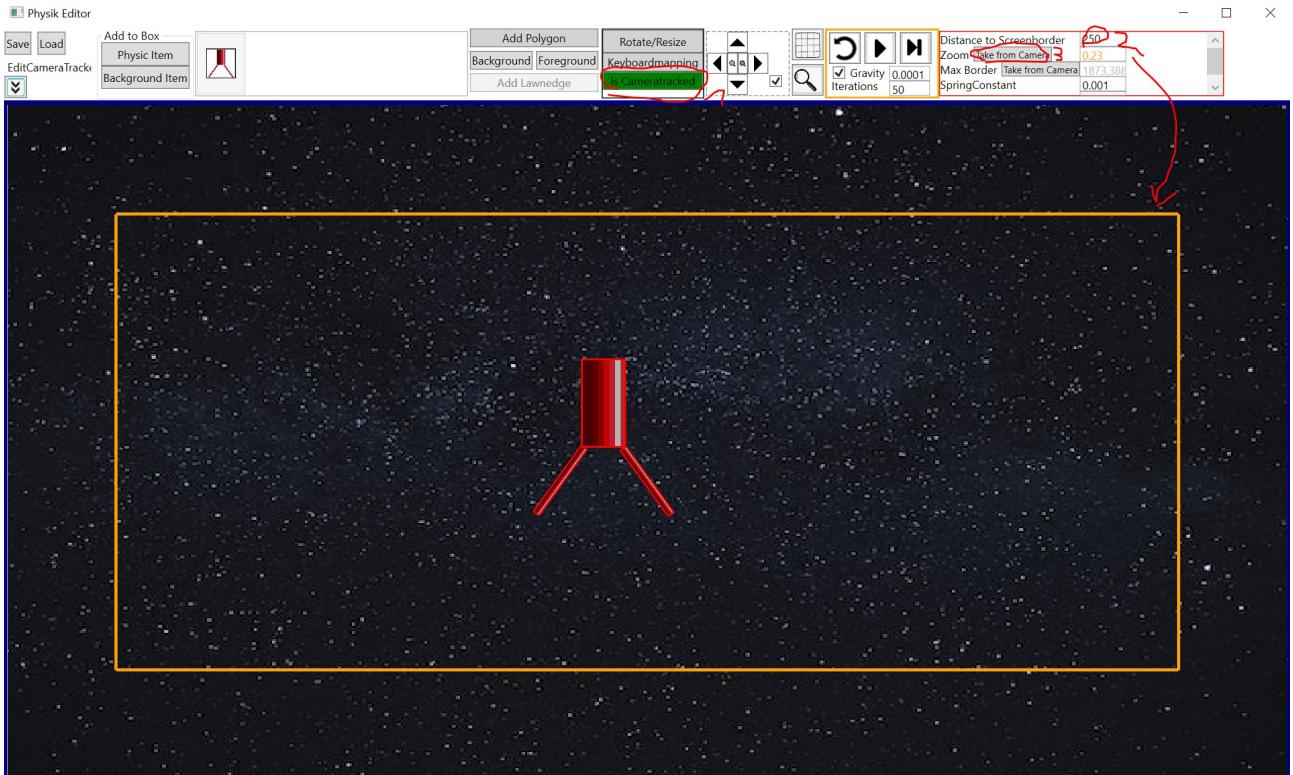
Wir lassen die Simulation laufen und sehen beim Moment des Aufschlags, dass es eine Lastspitze von 1,37 auf die Beine gibt.



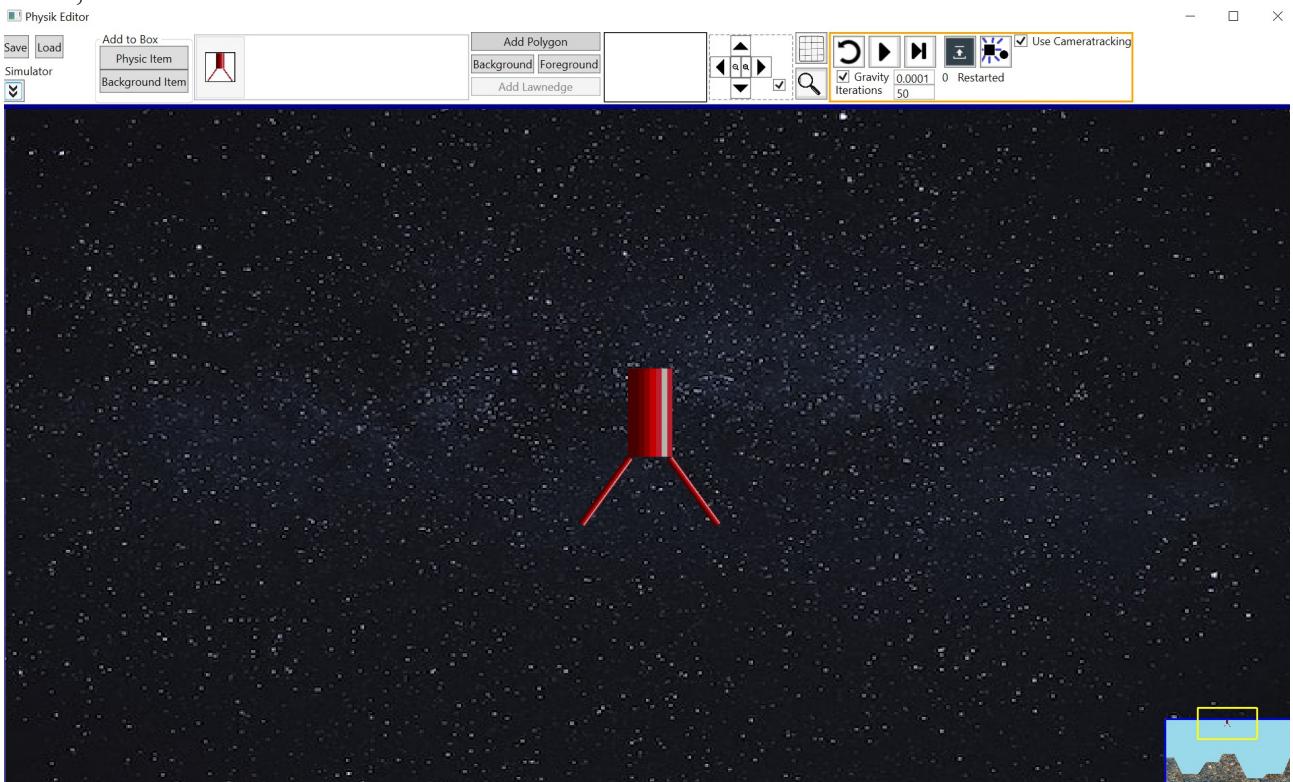
Wir sagen nun, dass die Beine bei einer Kraft ab 0.5 zerbrechen sollen:



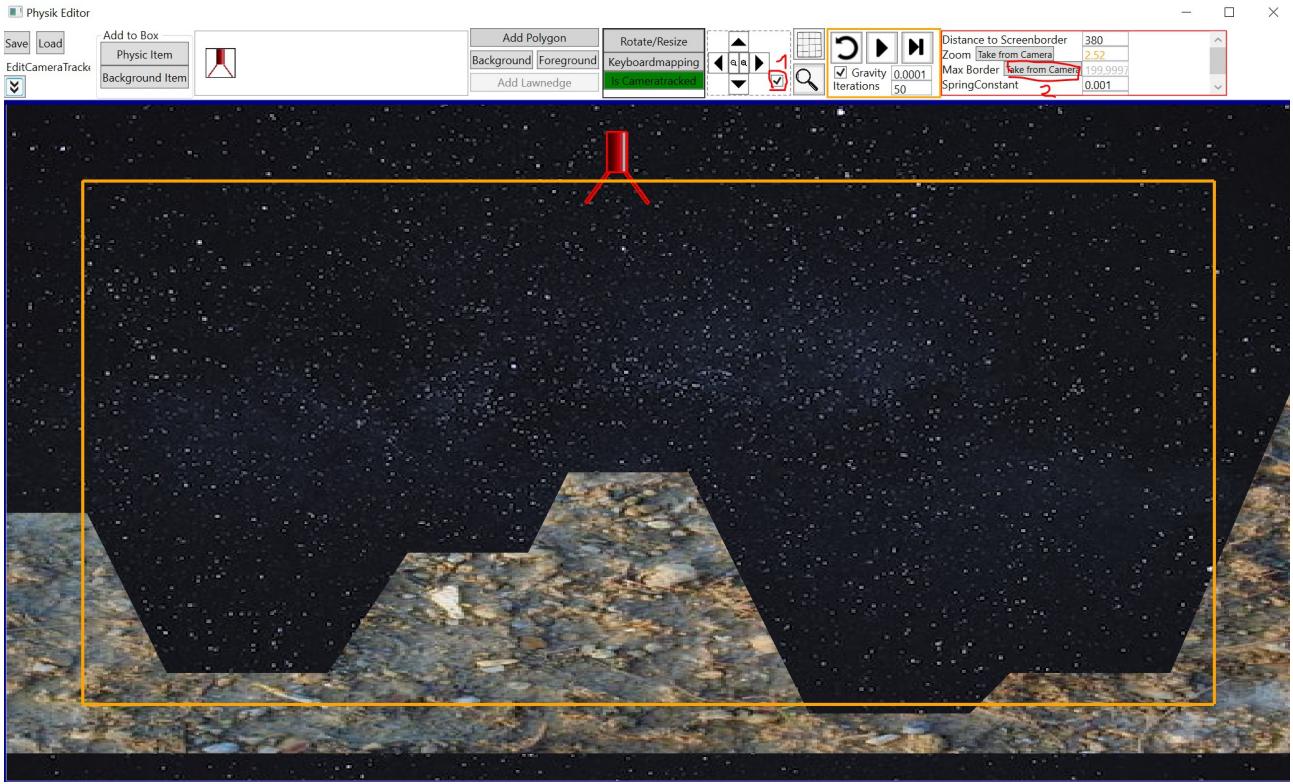
Nun wollen wir noch dass die Kamera automatisch dem Raumschiff folgt. Dazu zoomen wir mit der Kamera zuerst so, dass das Raumschiff gut zu sehen ist und klicken es dann an. Dann klicken auf auf „Is Cameratracked“ (Grün = Dieses Objekt wird von der Kamera verfolgt) und definieren dann den Bereich (2), ab dem die Kamera sich bewegt, wenn das Raumschiff dagegen kommt. Über „Take Zoom from Camera“ (3) sagen wir, dass während der Simulation die Kamera genau den Zoomfaktor haben soll, den wir gerade eingestellt haben.



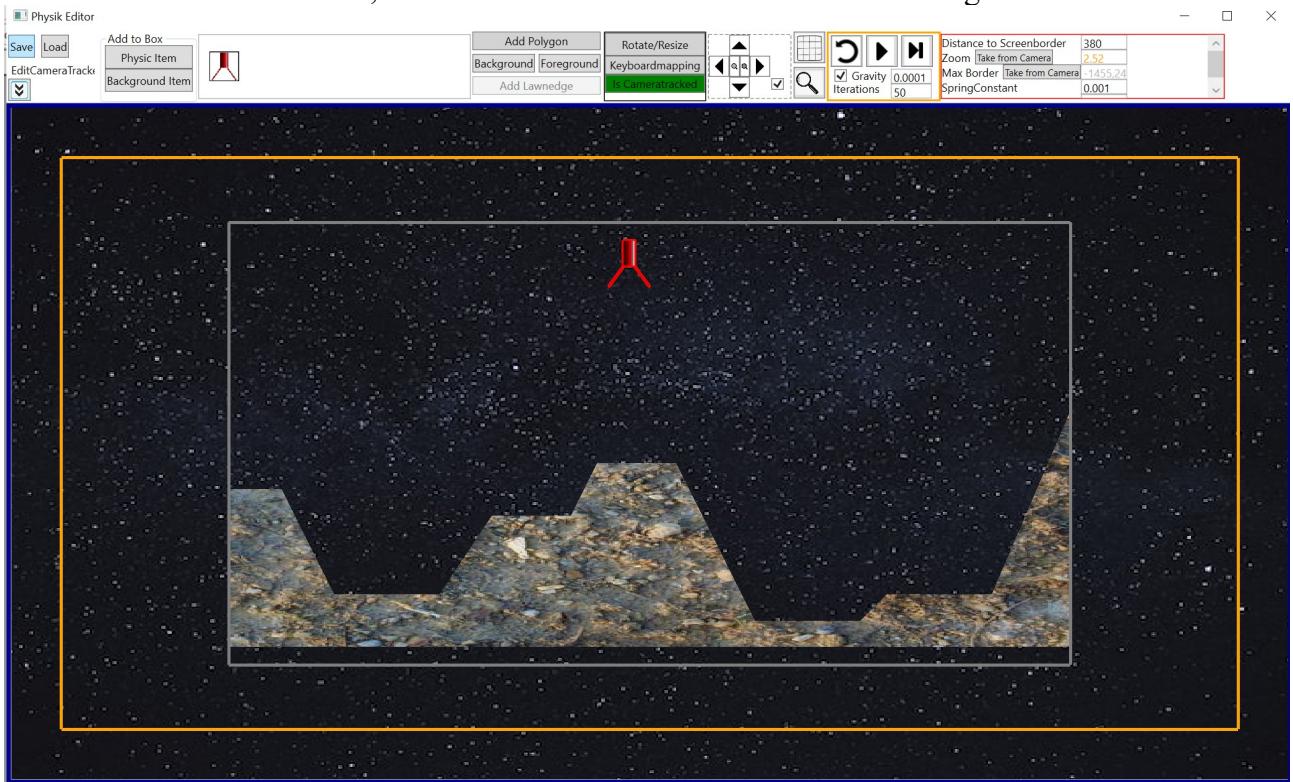
Ab jetzt folgt die Kamera im Simulator dem Raumschiff und man sieht nur noch im kleinen Fenster unten, wie weit es noch bis zum Boden ist:



Um nun zu verhindern, dass die Kamera sich außerhalb der Szene aufhält, kann man ein Rechteck definieren, innerhalb dessen die Kamera immer bleiben soll. Dazu klicken wir zuerst zweimal auf Autozoom (1) um die ganze Szene genau ins Bild zu bekommen und dann drücken auf auf „Take Max Border from Camera“ (2).

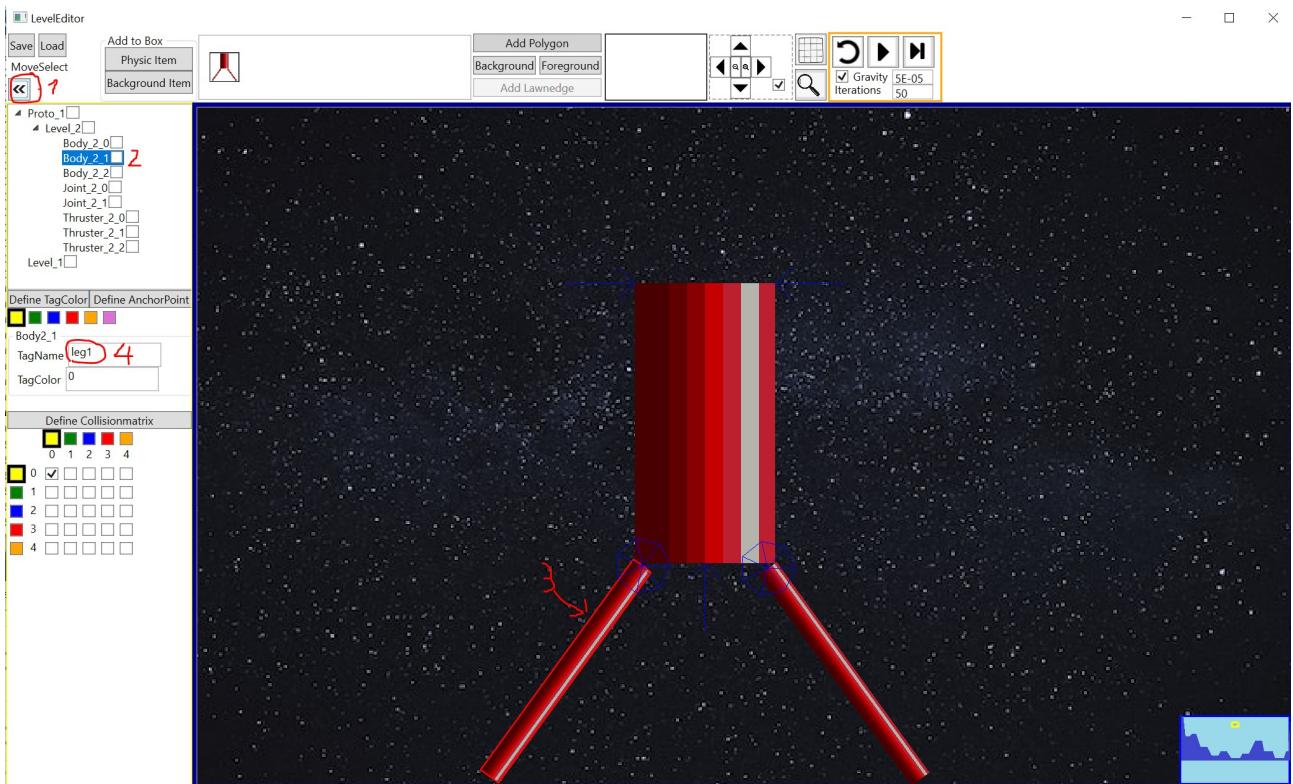


Er nimmt nun das aktuelle Sichtfenster als Bereich, wo die Kamera während der Simulation sein darf. Zoomt man weiter raus, sieht man auch das Max-Border-Rechteck in grauer Farbe:

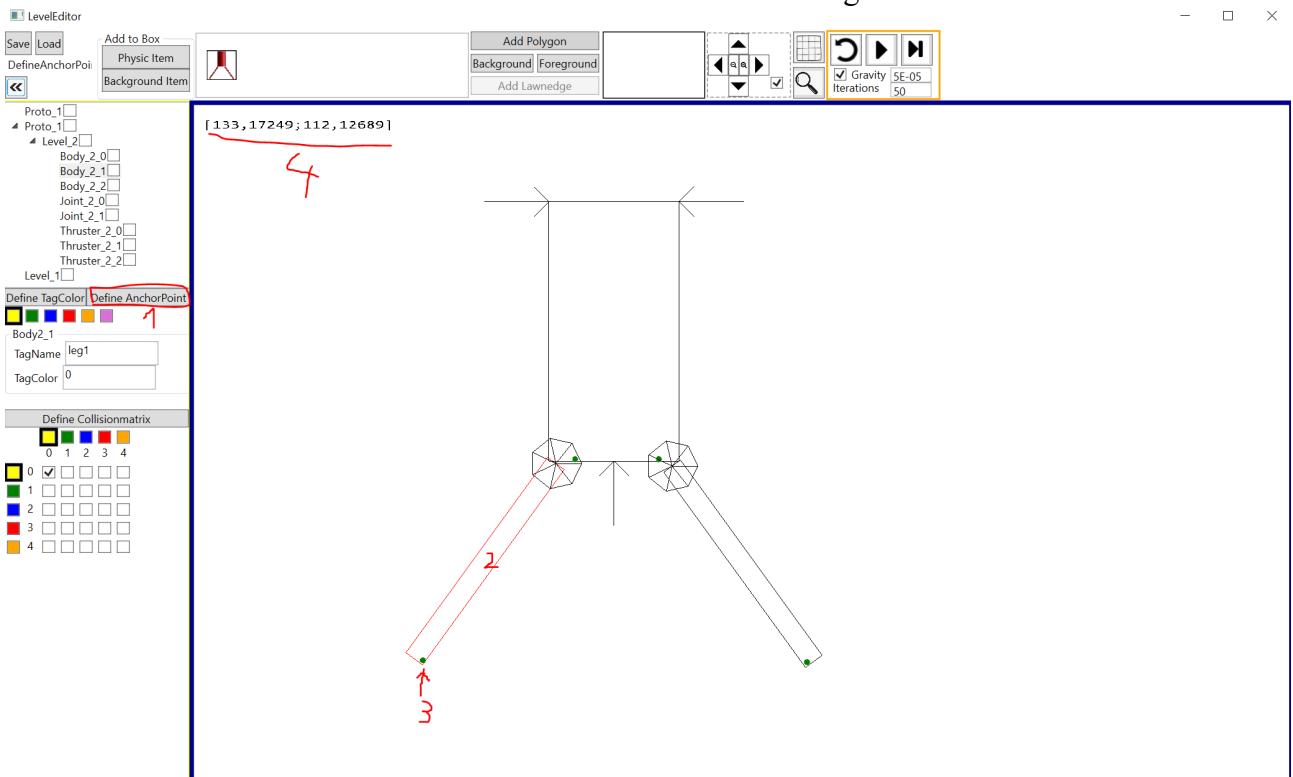


Tagging

Um das Raumschiff dann bei der Verwendung der PhysicEngine im UserCode ansprechen zu können kann man den einzelnen Bodys/Joints/Thrusters Tag-Namen geben. Dazu gehen wir in der linken Leiste (1) in den Level-Baum und selektieren dort das linke Bein, wodurch es eine rote Umrandung (3) bekommt. Wir vergeben den Tag-Name „leg1“ (4).



Neben dem Name können auch noch Ankerpunkte definiert werden. Wir gehen dazu auf „Define AnchorPoint“ (1) und selektieren im Editor den Körper, der ein Ankerpunkt erhalten soll (2) und dann klicken wir innerhalb des Körpers an die Stelle, wo der Punkt definiert werden soll. Links oben (4) sieht man seine lokalen Koordinaten. Es können mehrere Punkte pro Starrkörper definiert werden. Per Rechtsklick auf ein Punkt können diese auch wieder gelöscht werden.



Die Ankerpunkte können dann benutzt werden, um innerhalb des Spiel zu berechnen, wie viel

Abstand die Punkte bis zum Boden haben. Über den Name des Körpers und den Index des Punktes (Wir haben nur ein Punkt pro Körper. Also ist Index hier 0) können die Punkte dann angesprochen werden und über GetPosition kann dann die aktuelle Position in Weltkoordinaten ausgegeben werden.

Hier sieht man auch, wie man Gelenke per TagName anspricht. So kann dann über die IsBroken-Property geprüft werden, ob das Gelenk zerbrochen ist.

```

14     private AnchorPoint shipPoint1, shipPoint2;
15
16     public Ship(ITagDataProvider tagProvider)
17     {
18         this.weld1 = tagProvider.GetJointByTagName("weld1");
19         this.weld2 = tagProvider.GetJointByTagName("weld2");
20
21         this.shipPoint1 = new AnchorPoint(tagProvider, "leg1", 0);
22         this.shipPoint2 = new AnchorPoint(tagProvider, "leg2", 0);
23     }
24
25     4 Verweise
26     public Vector2D GetLeg1()
27     {
28         return this.shipPoint1GetPosition();
29     }
30
31     4 Verweise
32     public Vector2D GetLeg2()
33     {
34         return this.shipPoint2GetPosition();
35     }
36
37     this.IsBroken = this.weld1.IsBroken || this.weld2.IsBroken;
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
69
70
71
72
73

```

Siehe: DemoApplications/Moonlander/MoonlanderControl/Model/Ship.cs

Hier ist noch ein Beispiel, wie man Schubdüsen und Starrkörper per Tag-Name anspricht. So kann man dann die Feuerpartikel z.B. nur dann erzeugen lassen, wenn der Körper mit dem Name „ship“ noch existiert und wenn die Hauptschubdüse gerade aktiv ist.

```

26     public MainThruster(Simulator simulator)
27     {
28         this.anchorPoint1 = new AnchorPoint(simulator, "ship", 0);
29         this.anchorPoint2 = new AnchorPoint(simulator, "ship", 1);
30         this.mainThruster = simulator.GetThrusterByTagName("mainThruster");
31         this.ship = simulator.GetBodyByTagName("ship");
32
33         simulator.BodyWasDeletedHandler += Simulator_BodyWasDeletedHandler;
34     }
35
36     1 Verweis
37     private void Simulator_BodyWasDeletedHandler(RigidBodyPhysics.PhysicScene sender, RigidBodyPhysics.RigidBody.IPublicRigidBody body)
38     {
39         if (body == this.ship)
40         {
41             this.shipIsRemovedFromSimulation = true;
42         }
43     }
44
45     1 Verweis
46     public void MoveOnStep(float dt)
47     {
48         if (this.mainThruster.IsEnabled && this.shipIsRemovedFromSimulation == false)
49         {
50             this.particles.AddRange(CreateNewParticles());
51         }
52     }
53

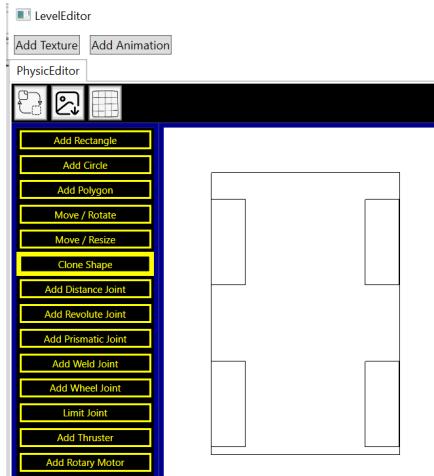
```

Siehe: DemoApplications/Moonlander/MoonlanderControl/Model/MainThruster.cs

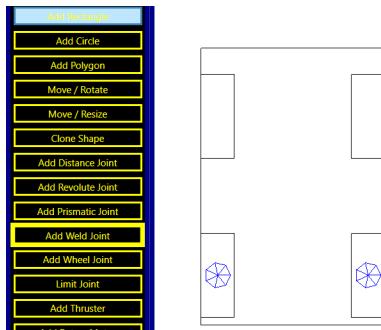
Beispiel 4: Auto was man von oben sieht

Das Physikmodel vom Auto

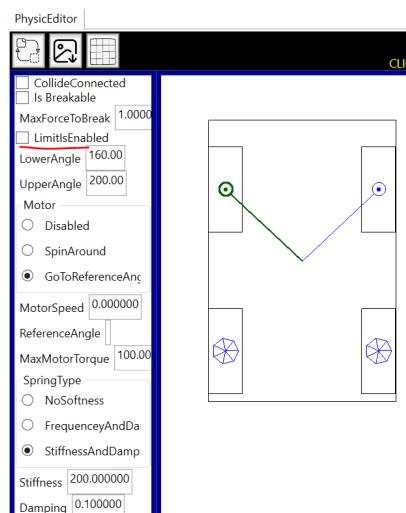
Es soll diesmal ein 2D-Autospiel erzeugt werden, wo man von oben auf das Auto drauf sieht. Dazu wird im PhysicEditor das Gehäuse vom Auto erzeugt, indem ein Rechteck erstellt wird und dann die 4 Räder über die Clone-Shape und Shift-Taste:



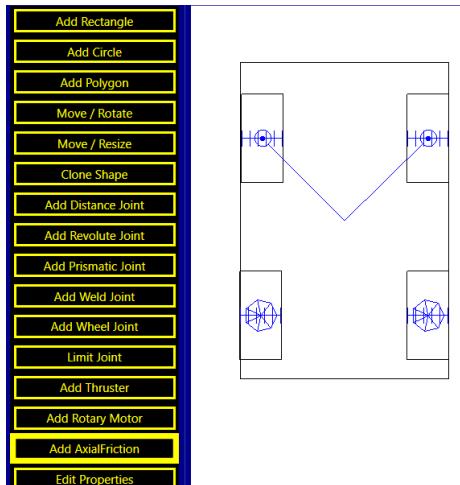
Die beiden Räder unten sind die Hinterräder. Diese werden per WeldJoint am Gehäuse befestigt, indem auf „Add Weld Joint“ geklickt wird. Dann auf das Rad, dann das Gehäuse. Dann die Shift-Taste halten und auf den grünen Punkt in der Mitte vom Rad klicken, um somit den ersten Ankerpunkt zu platzieren. Dann nochmal auf den grünen Punkt klicken, um den zweiten Ankerpunkt zu platzieren.



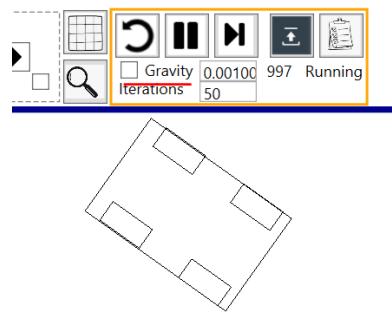
Die Vorderräder müssen lenken können. Deswegen nutzen wir hier Revolute-Joints, um auch hier die Radmitte am Gehäuse zu befestigen. Da der Ankerpunkt genau in der Radmitte ist, haben wir hier kein Hebelarm, der vom Radzentrum zum Ankerpunkt zeigt. Somit sehen wir nur den Hebelarm vom Gehäuse zur Radmitte. Wegen des fehlenden Hebelarms deaktivieren wir hier „LimitIsEnabled“.



Die Räder sollen nur nach vorne/hinten rollen können aber nicht zur Seite. Wir nutzen dafür die AxialFrictions und lassen sie pro Rad in seitliche Richtung wirken.



Wenn wir nun das Auto im Simulator testen (vorher Gravitation deaktivieren, da bei ein TopDown-Spiel diese Null ist), können wir sehen, dass es sich mit der Maus in Fahrtrichtung verschieben lässt aber nicht seitlich:

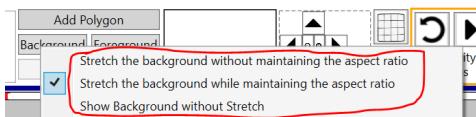


Wir weisen dem Auto nun noch eine Textur zu und erzeugen in Paint nun eine Karte, wo unser Auto drauf fahren soll. Im nächsten Abschnitt steht, wie man dann dieses so erzeugte Bild im Leveleditor benutzt.

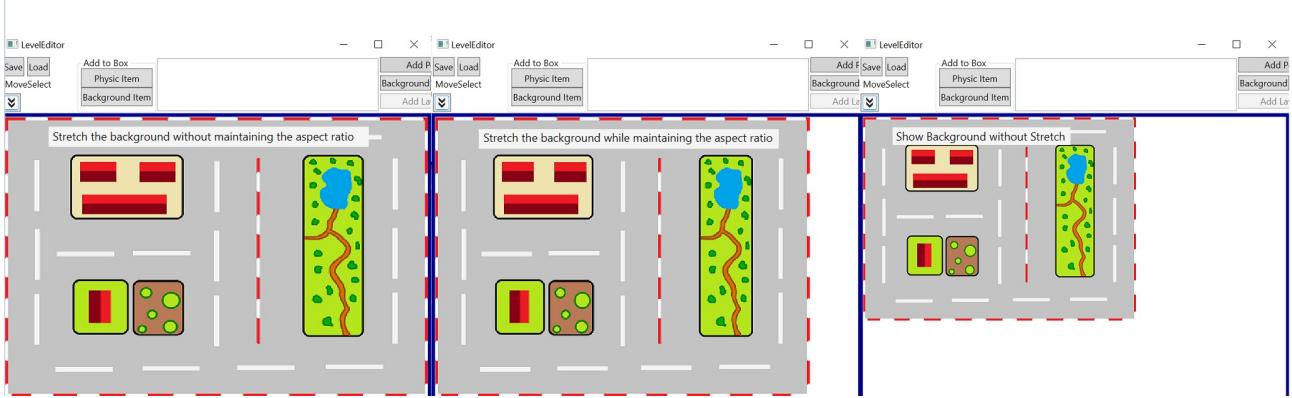
Karte von oben

Bis jetzt waren die Spiele immer so, dass man von der Seite die Objekte sieht. Die Position der Physik-Objekte hing davon ab, wie die Kamera platziert wurde. Das Hintergrundbild wurde so skaliert, dass es das gesamte Fenster einnimmt. Physik-Objekte und der Hintergrund waren zwei getrennte Dinge, die Abhängig von der Kamera-Position und der Fensterbreite sich zueinander verschieben konnten.

Damit man Spiele entwickeln kann, wo man von oben auf die Karte drauf schaut, ist es nötig, dass die Physik-Objekte und das Hintergrundbild miteinander verknüpft werden. Dazu hat der Button vom Backgroundimage im Leveleditor ein Contextmenü, wo ich folgende Dinge einstellen kann:



So sehen diese drei Background-Image-Modi aus:

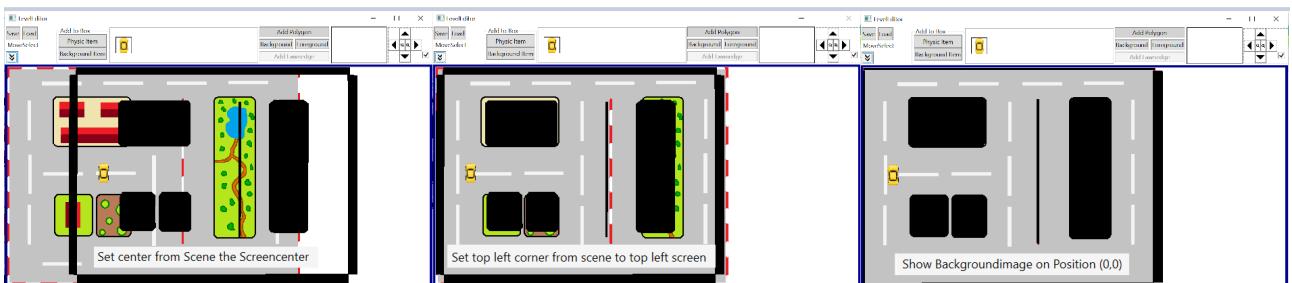


Links = Bild nimmt das ganze Fenster ein (wurde bis jetzt immer verwendet)

Mitte = Bild wird unter Beibehaltung des Seitenverhältnis skaliert

Rechts = Bild wird in Originalgröße angezeigt

Bei der Kamera kann über das ContextMenü eingestellt werden, an welcher Stelle vom Fenster die Szene (hier schwarze Objekte) angezeigt wird:

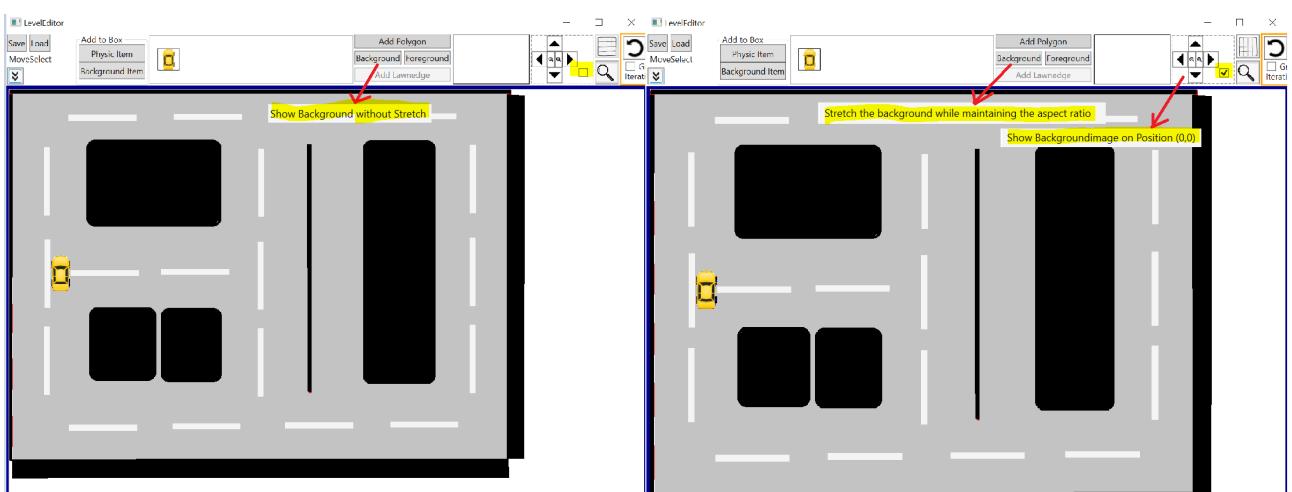


Links = Mitte der Szene wird in der Fenster-Mitte angezeigt

Mitte = Linke obere Ecke der Szene ist an der linken oberen Fenster-Ecke

Rechts = Kamera zeigt den Bereich (0,0) – (BackgroundImageWidth,Height)

Um ein Spiel zu erzeugen wo man von oben auf die Karte schaut muss die Kamera und das Hintergrundbild so eingestellt werden, dass sie miteinander „verklebt“ sind. Dazu gibt es zwei Möglichkeiten:



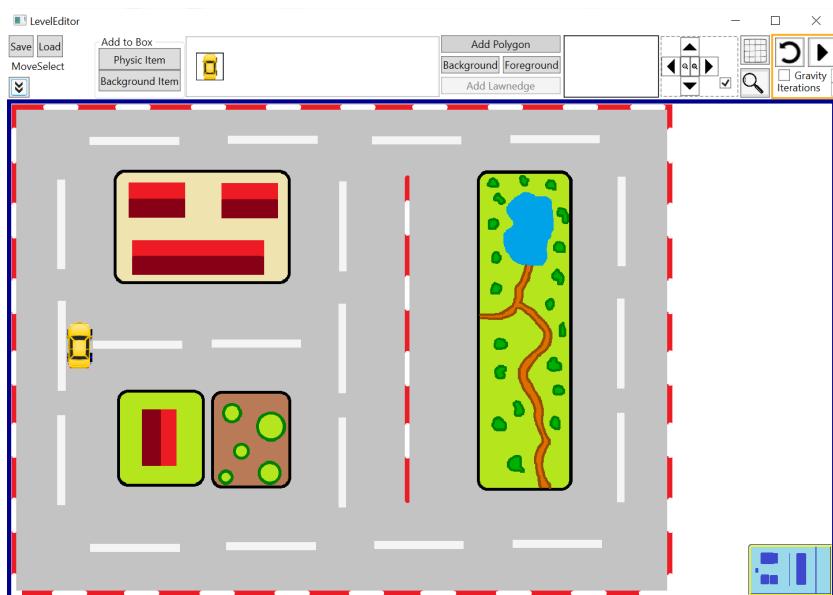
Möglichkeit 1 (linkes Bild): Das Hintergrundbild wird ohne Stretch angezeigt und der AutoZoom der Kamera wird deaktiviert. Nachteil ist hier, dass die Karte abhängig von der Größe des Bildes zu klein/zu groß für das Fenster sein kann.

Möglichkeit 2 (rechtes Bild): Das Hintergrundbild wird im StretchWithAspectRatio-Modus

angezeigt. Der AutoZoom der Kamera wird aktiviert und die Kamera wird im Modus „Show BackgroundImage“ betrieben. Hier kann ich nun die Fenstergröße ändern und die Karte passt sich genau an.

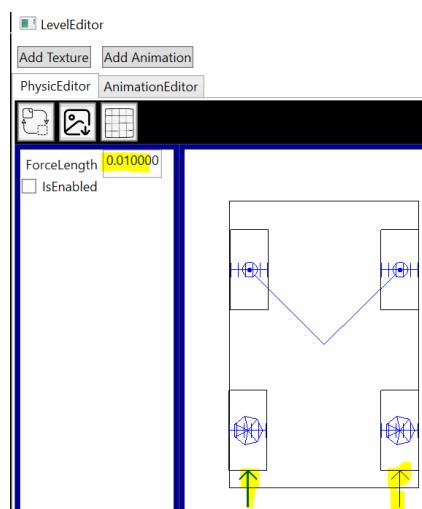
Nachdem die Kamera+Hintergrundbild so eingestellt wurden, dass sie nun genau übereinander liegen ist es nun die Aufgabe überall da, wo Wände/Häuser sind Polygone zu erzeugen um nun die Bereiche zu markieren, wo man nicht hin darf. Hier im Beispiel sind die Polygone während der Erstellung schwarz, damit man sie gut sieht. Nun kann über den Foreground-Button allen Polygonen eine transparente Farbe zugewiesen werden so dass sie dann in der Physikkollision und auf der Minimap noch da sind aber in der Haupt-Anzeige dann verschwinden.

So sieht die Karte mit transparenten Polygonen dann aus: Die Minimap zeigt die Objekte aber im Hauptbildschirm sind die Polygone weg:

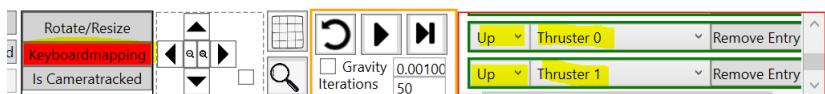


Das Auto steuern

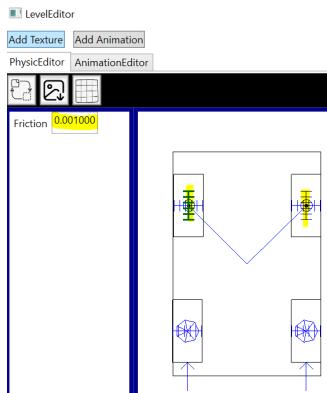
Um nach Vorne fahren zu können werden an die Hinterräder Thruster angebracht:



Wenn ich nun für diese beiden Schubdüsen ein Keyboard-Mapping anlege, dann werden sie beim drücken der Up-Taste aktiviert und beim Loslassen der Taste deaktiviert:

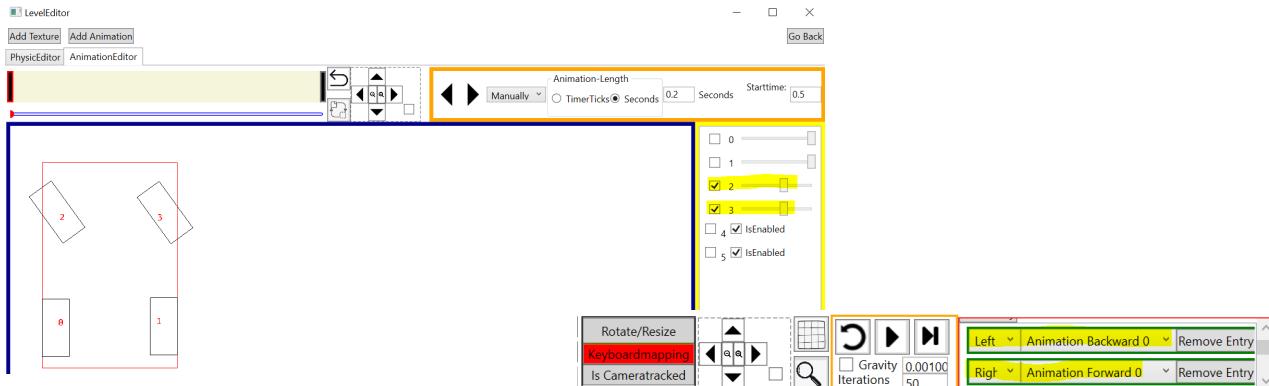


Für die Bremsen werden Axialfrictions verwendet, welche in Fahrtrichtung zeigen:



Wenn man nun auf Up drückt, fährt das Auto nach Vorne und wenn man Up loslässt, dann wird es wegen der soeben eingefügten Axialfriction immer langsamer, bis es steht. Wenn man aber während der Fahrt nur eine sehr geringe Bremswirkung will und eine starke Bremswirkung nur dann, wenn die Down-Taste gedrückt wird, dann muss man diese Logik in der GameSimulator-Klasse dann umsetzen, welche in ein späteren Abschnitt behandelt wird. Im Leveleditor hat man aktuell also nur eine schwache Bremse.

Die Lenkung kann man dann durch eine Animation der Revolute-Joints umsetzen:



Damit beide Räder aber exakt gleich viel in beide Richtungen lenken ist es nötig, dass man die Leveleditor-Textdatei dann manuell im Notepad bearbeitet. Suche dazu nach „Animation“ und dort sieht man dann im 3. und 4. Element unter Values die Revolute-Joint-Werte. 0.5 bedeutet das Rad zeigt nach vorne. 0.4 es zeigt etwas nach links und 0.6 es zeigt etwas nach rechts. Diese gelb markierten Werte wurden per Hand in eingetragen.

```
        "AnimationData": {
            "$type": "KeyFrameGlobal.AnimationOutputData, KeyFrameGlobal",
            "Frames": [
                {
                    "$type": "KeyFrameGlobal.FrameData[], KeyFrameGlobal",
                    "$values": [
                        {
                            "$type": "KeyFrameGlobal.FrameData, KeyFrameGlobal",
                            "Time": 0.0,
                            "Values": {
                                "$type": "System.Object[], System.Private.CoreLib",
                                "$values": [
                                    200.0,
                                    200.0,
                                    0.6,
                                    0.6,
                                    true,
                                    true
                                ]
                            }
                        },
                        {
                            "$type": "KeyFrameGlobal.FrameData, KeyFrameGlobal",
                            "Time": 1.0,
                            "Values": {
                                "$type": "System.Object[], System.Private.CoreLib",
                                "$values": [
                                    200.0,
                                    200.0,
                                    0.4,
                                    0.4,
                                    true,
                                    true
                                ]
                            }
                        }
                    ]
                }
            ]
        }
```

Wenn man die Lenkung über eine Animation steuert, dann kann man die Lenkung durch Loslassen der Left/Right-Tasten aber nicht auf die Null-Stellung zurück bringen. Die Vorderräder bleiben dann in der jeweiligen Auslenkung. Will man die Lenkung automatisch auf Neutral setzen dann muss man das auch in der GameSimulator-Klasse machen.

Einstellmöglichkeiten für die Kamera im Leveleditor

Wenn ich im Leveleditor etwas simuliere, dann kann ich die Kamera mit folgenden Möglichkeiten steuern:

Weg 1: Kamera ist fix – Zeige die ganze Szene

Aktiviere dazu den AutoZoom. Sollte am Level was verändert worden sein, hilft es, dass man den AutoZoom kurz deaktiviert und dann erneut aktiviert. So zeigt er dann die ganze Szene.

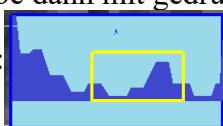


Weg 2: Zeige ein festen Bereich



Aktiviere zuerst das SmallWindow durch Klick auf die Lupe:

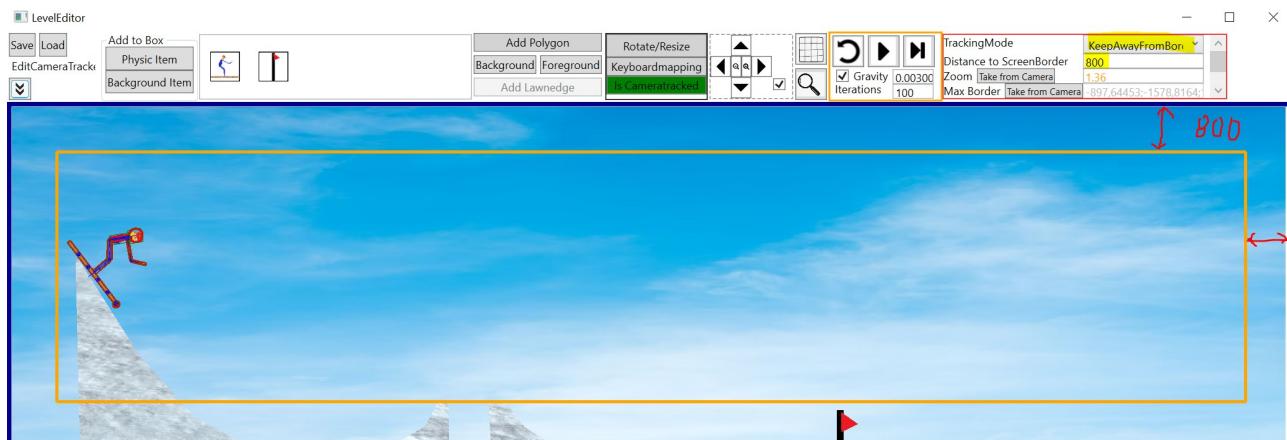
Verschiebe dann mit gedrückter Maustaste das gelbe Rechteck und verändere dessen Größe mit dem Mausrad:



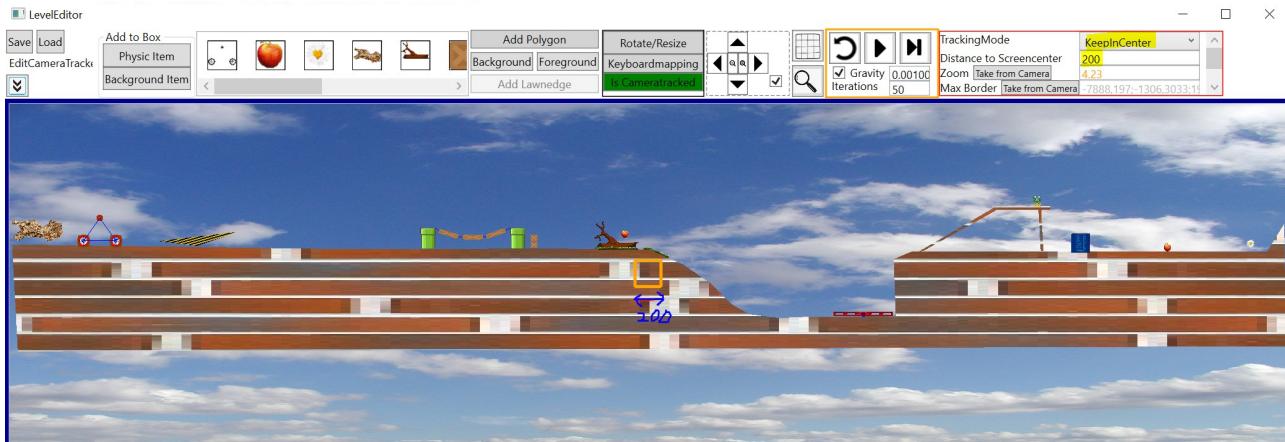
Der Leveleditor und Simulator zeigt dann nur noch den Levelbereich aus dem gelben Rechteck.

Weg 3: Die Kamera verfolgt ein Objekt was sich innerhalb vom orangenen Rechteck aufhalten darf

Klicke auf das Objekt was von der Kamera verfolgt werden soll und klicke dann auf den „Is Cameratracked“-Button so dass er grün wird (d.h. Der CameraTracker ist aktiv). Nutze dann den KeepAwayFromBorder-Modus. Beispiel: Der Skispringer muss immer im orangenen Rechteck bleiben.



Weg 4: Die Kamera verfolgt ein Objekt und zeigt es immer in der Bildschirmmitte an. Beispiel: Das Motorrad darf nie außerhalb des orangenen Rechtecks sein.



Weg 5: Die Kamera zeigt ein TopDown-Game

Man kann ein Spiel, wo man von oben auf die Karte schaut dadurch erzeugen, indem man bei der Kamera folgendes einstellt:



7.3 Leveleditor aus Programmiersicht

KeyFrameEditor

Der Editor ist so aufgebaut, dass man zuerst eine PhysicScene importieren muss. Dabei wird das PhysicScene-Objekt in ein AnimatorInputData-Objekt umgewandelt.

```
5  public class AnimatorInputData
6  {
7      5 Verweise
8      public IAnimationProperty[] Properties { get; set; } //Alle Propertys von ein Objekt, die animiert werden sollen
9      3 Verweise
10     public IAnimationObject AnimationObject { get; set; } //Um das dahinter liegende PhysikModel mit TimeStep zu bewegen
11     3 Verweise
12     public IAnimationModelDrawer AnimationModelDrawer { get; set; } //Zum Zeichnen der PhysikScene
13 }
```

Siehe: Engine/04_AnimaitonAndTexture/KeyFrameEditor/KeyFrameGlobal/AnimatorInputData.cs

Dieses Objekt hat ein Properties-Array, wo jeder Eintrag auf ein Gelenk oder Thruster von der PhysicScene verweist. Die Grundidee ist, dass man ein beliebiges Objekt animieren kann, was aus einer Menge von Properties (oder auch einfach nur Setter-Funktionen) besteht. Jeder Setter muss vom Datentyp Float oder Bool sein. Es wäre aber auch denkbar noch weitere Datentypen wie String zu unterstützen.

Über den AnimationModelDrawer kann das zu animierende Objekt (die PhysicScene) gezeichnet werden. Da eine PhysicScene nur dann sich bewegt, wenn die TimeStep-Methode aufgerufen wird, gibt es noch das AnimationObject. Der KeyFrameAnimator nutzt das AnimatorInputData-Objekt, um das für ihn unbekannte Animationsobjekt zu zeichnen und um es zu bewegen.

Seine Aufgabe ist es die Soll-Werte, welche beim der Properties-Array reingegeben werden zu definieren. Der Editor erzeugt ein AnimationOutputData-Objekt.

Dieses Objekt kann dann zur Wiedergabe der Animation genutzt werden.

```
3  public class AnimationOutputData
4  {
5      12 Verweise
6      public enum AnimationType
7      {
8          OneTime,    // Animation wird per Timer einmal abgespielt. Beispiel: Person verschießt mit Bogen ein Pfeil
9          AutoLoop,   // Animation wird per Timer unendlich oft wiederholt. Beispiel: Person die läuft
10         Manually    // Animation muss per KeyDown vor/zurück gespult werden. Beispiel: Skispringen
11     }
12
13     10 Verweise
14     public FrameData[] Frames { get; set; }
15     3 Verweise
16     public float DurationInSeconds { get; set; } //So lange ist die Animation in Sekunden
17     3 Verweise
18     public AnimationType Type { get; set; }
19     4 Verweise
20     public bool[] PropertyIsAnimated { get; set; }
21
22     0 Verweise
23     public AnimationOutputData() { } //Für den Serialisierer
24
25     1 Verweis
26     public AnimationOutputData(FrameData[] frames, float durationInSeconds, AnimationType type, bool[] propertyIsAnimated)
27     {
28         Frames = frames;
29         DurationInSeconds = durationInSeconds;
30         Type = type;
31         PropertyIsAnimated = propertyIsAnimated;
32     }
33 }
```

Siehe: Engine/04_AnimaitonAndTexture/KeyFrameEditor/KeyFrameGlobal/AnimationOutputData.cs

TextureEditor

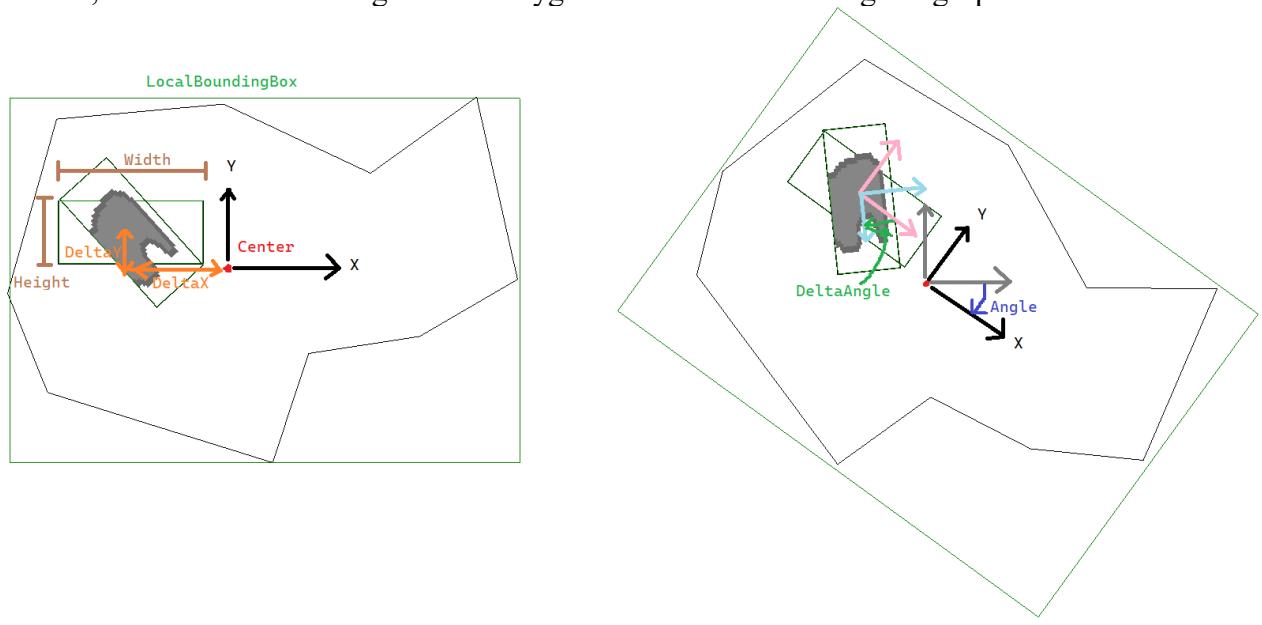
Im VisualizerGlobal-Projekt ist die VisualizerInputData-Klasse definiert, welche eine Menge von I2DAreaShape-Objekten hat.

```
6     public class VisualizerInputData
7     {
8         2 Verweise
9             public I2DAreaShape[] Shapes { get; set; }
10
11        //2D-Figur die eine Fläche einnimmt (Kein Kreis/Linie/Punkt)
12        //Diese Figur wurde im Lokalspace definiert und dann zum Punkt "Center" hin verschoben und dort dann noch um "Angle" gedreht
13        11 Verweise
14        public interface I2DAreaShape
15        {
16            27 Verweise
17                Vector2D Center { get; }
18            6 Verweise
19                float Angle { get; }
20
21            //AABB-BoundingBox von der Shape im LokalSpace (Wenn Objekt nicht gedreht wurde)
22            5 Verweise
23            RectangleF LocalBoundingBox { get; }
24        }
25    }
```

Siehe: Engine/04_AnimaitonAndTexture/TextureEditorGlobal/VisualizerInputData.cs

Jede AreaShape ist dadurch definiert, dass es ein Zentrum und eine Angle hat.

Beispiel: Hier ist ein schwarzes Polygon gegeben, was ein Center (roter Punkt) hat und es wurde um den blauen Angle im Uhrzeigersinn etwas gedreht. Die LocalBoundingBox wird dadurch definiert, dass um das noch ungedrehte Polygon eine axiale Boundingbox gespannt wird.



Im Editor kann ich nun eine Textur auf dieses Polygon ankleben. Hier wird eine graue Figur befestigt. Dazu definiere ich im ungedrehten Polygon (linkes Bild) den DeltaX, und DeltaY-Wert. Um diesen Wert wird die Textur vom Polygonzentrum aus verschoben. Dann kann diese Texture noch um DeltaAngle gedreht werden. Außerdem kann von der angebrachten Textur noch dessen Breite und Höhe (Width/Height) festgelegt werden.

Der Textur-Editor erzeugt pro AreaShape-Objekt ein TextureExportData-Objekt, was so aussieht:

```

15 public class TextureExportData
16 {
17     7 Verweise
18     public string TextureFile { get; set; }
19     5 Verweise
20     public bool MakeFirstPixelTransparent { get; set; }
21     5 Verweise
22     public Color ColorFactor { get; set; }
23     4 Verweise
24     public int DeltaX { get; set; }
25     4 Verweise
26     public int DeltaY { get; set; }
27     8 Verweise
28     public int Width { get; set; }
29     8 Verweise
30     public int Height { get; set; }
31     5 Verweise
32     public float DeltaAngle { get; set; }
33     5 Verweise
34     public float ZValue { get; set; }

```

Siehe: Engine/04_AnimationAndTexture/TextureEditorGlobal/VisualisizerOutputData.cs

Im VisualisizerGlobal-Projekt wird noch das Importer-Interface definiert:

```

3  public interface IVisualisizerImporter
4  {
5      2 Verweise
6      VisualisizerInputData Import();

```

Dieses Interface wird vom PhysicSceneImporter implementiert. Er wandelt ein PhysicScene-Objekt in ein VisualisizerInputData-Objekt um.

```

10  namespace PhysicImporter
11  {
12      2 Verweise
13      public class PhysicSceneImporter : IVisualisizerImporter
14      {
15          private PhysicScene physicScene;
16          1 Verweis
17          public PhysicSceneImporter(string physicSceneFile)
18          {
19              var rawPhysicData = Helper.FromCompactJson<PhysicSceneExportData>(File.ReadAllText(physicSceneFile));
20              this.physicScene = new PhysicScene(rawPhysicData);
21          }
22          2 Verweise
23          public VisualisizerInputData Import()
24          {
25              return new VisualisizerInputData()
26              {
27                  Shapes = physicScene.GetAllBodies().Select(Convert).ToArray()
28              };
29          }
30          1 Verweis
31          private static I2DAreaShape Convert(IPublicRigidBody body)
32          {
33              if (body is IPublicRigidRectangle)
34              {
35                  return new RigidRectangle((IPublicRigidRectangle)body);
36              }
37              if (body is IPublicRigidPolygon)
38              {
39                  return new RigidPolygon((IPublicRigidPolygon)body);
40              }
41              if (body is IPublicRigidCircle)
42              {
43                  return new RigidCircle((IPublicRigidCircle)body);
44              }
45              throw new NotImplementedException();
46          }
47      }

```

Der Editor nutzt dann nur das VisualisizerInputData-Objekt, um damit dann ein VisualisizerOutputData-Objekt zu erzeugen:

```

5  public class VisualisizerOutputData
6  {
7      7 Verweise
8      6 Verweise
9      public TextureExportData[] Textures { get; set; }

```

Über den PhysicSceneDrawer kann dann die PhysicScene gezeichnet werden:

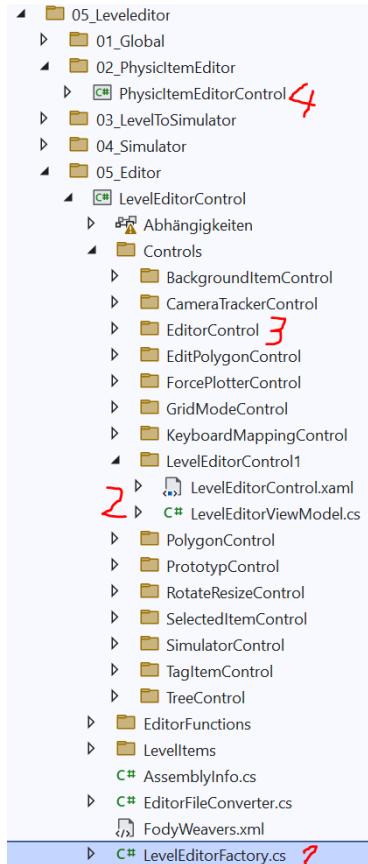
```

8  namespace PhysicImporter.Drawer
9  {
10     3 Verweise
11     public class PhysicSceneDrawer
12     {
13         private ITexturedRigidBody[] textures;
14         1 Verweis
15         public PhysicSceneDrawer(PhysicScene physicScene, VisualisizerOutputData textureData)
16         {
17             this.textures = ConvertPhysicScene(physicScene, textureData);
18         }

```

Leveleditor

Der Leveleditor ist ein UserControl (2), was aus 5 Unterprojekten besteht. Dieses UserControl (und sein zugehöriges ViewModel) wird über die LevelEditorFactory (1) erzeugt. Das LevelEditorControl(2) hat lediglich ein ContentControl, was entweder das EditorControl (3) oder das PhysicItemControl (4) anzeigt. Das PhysicItemControl nutzt dann das PhysicSceneEditorControl, TextureEditorControl und KeyFrameEditorControl zur Definition von ein PhysicItem, was aus einer PhysicScene, Texturen und Animationsdaten besteht. Im EditorControl können dann die ganzen LevelItems platziert werden.



Das LevelEditorViewModel ist zwar das VM, was am höchsten liegt aber das ist lediglich die Weiche zwischen dem EditorControl und dem PhysicItemControl. Das EditorViewModel (1) enthält alle Daten vom Leveleditor. Dazu hat es die EditorState-Variable (2) und auch die ganzen ViewModel von all den Controls. Wenn ich im Editor dann auf ein Button drücke oder mit der Maus in das Grafik-Fenster klicke, dann wird eine IEditorFunction (3) aktiviert, welche dann alle Maus- und Tastatur-Events für das Grafik-Fenster behandelt. Diese EditorFunktion bekommt als Input den EditorState und es verändert dann dort die Daten. So fügt die AddItem-Funktion zum Beispiel neue LevelItems (4) dem EditorState hinzu.

Screenshot of the Visual Studio IDE showing the file structure and code for the `EditorViewModel.cs` class.

File Structure:

```

05_Editor
  LevelEditorControl
    Abhängigkeiten
    Controls
      BackgroundItemControl
      CameraTrackerControl
      EditorControl
        DoubleDown.png
        Doubleleft.png
      EditorControl.xaml
      EditorViewModel.cs (highlighted with red box)
      FilePaths.cs
      Grid.png
      LevelEditorExportData.cs
      Magnifying_glass.png
  
```

Code View:

```

internal class EditorState : ReactiveObject
{
    public GraphicPanel2D Panel;
    public Camera2D Camera;
    public ObservableCollection<IPrototypItem> Prototyps = new ObservableCollection<IPrototypItem>();
    8 Verweise
    [Reactive] public IPrototypItem SelectedPrototyp { get; set; }
    public ObservableCollection<ILevelItem> LevelItems = new ObservableCollection<ILevelItem>();
    4
}

private EditorState editorData = new EditorState(); //Diese Variable wird der jeweils aktiven IEditorFunction-Funktion reingegeben
private IEeditorFunction function = null; //State-Designpattern für das GraphicPanel2D

public enum FunctionType { Nothing, AddItem, MoveSelect, Simulator, AddPolygon, EditPolygon, AddLawnEdge, EditLawnEdge, Key
    6
    38 Verweise
    internal interface IEeditorFunction : ITimerHandler, IGraphicPanelHandler, IDisposable
    {
        15 Verweise
        FunctionType Type { get; }
        20 Verweise
        IEeditorFunction Init(EditorState state);
        9 Verweise
        bool HasPropertyControl { get; } //Soll im MainControl innerhalb eines ContentControls ein UserControl gezeigt werden?
        8 Verweise
        System.Windows.Controls.UserControl GetPropertyControl(); //Hier können noch zusätzliche Eigenschaften angeigt werden
    11
    12
    13
}
  
```

Ein PhysicItem entspricht einer PhysicScene mit Textur- und Animationsdaten. Ich kann ein neues PhysicItem dadurch erzeugen, indem ich auf dem „Add Item“-Button vom PrototypUserControl klicke (1), was ein Event auslöst, das vom LevelEditorControl(2) behandelt wird, so dass er weiß, dass er nun das PhysicItemControl (3) anzeigen muss. Dieses Control erstellt dann ein IPrototypItem (4) was eine Draw-Methode hat. Von diesen Prototyp können nun viele PhysicLevelItems erzeugt werden. Jedes Item verweist dann auf den gleichen Prototyp aber hat eine eigene Position im Level. Das PhysicLevelItem wird dadurch gezeichnet, indem er die aktuelle ModelMatrix vom GrafikPanel so ändert (5), dass er festlegt, an welcher Stelle dann sein Prototyp-Objekt (6) gezeichnet werden soll.

Screenshot of the Visual Studio IDE showing the file structure and code for the `IPrototypItem` interface and `PhysicLevelItem` class.

File Structure:

```

05_Editor
  LevelEditorControl
    Abhängigkeiten
    Controls
      BackgroundItemControl
      CameraTrackerControl
      EditorControl
      EditPolygonControl
      ForcePlotterControl
      GridModeControl
      KeyboardMappingControl
      LevelEditorControl1 (highlighted with red box)
      PolygonControl
      PrototypControl
        C# PrototypControlExportData.cs
        C# PrototypItemViewModel.cs
        C# PrototypUserControl.xaml (highlighted with red box)
        C# PrototypViewModel.cs
        RotateResizeControl
        SelectedItemControl
        SimulatorControl
        TagItemControl
        TreeControl
    EditorFunctions
    LevelItems
  
```

Code View:

```

//Element vom PrototypControl
public interface IPrototypItem
{
    23 Verweise
    public enum Type { PhysicItem, BackgroundItem, GroupedItem }

    5 Verweise
    public Type ProtoType { get; }
    18 Verweise
    int Id { get; }
    38 Verweise
    RectangleF BoundingBox { get; }
    15 Verweise
    InitialRotatedRectangleValues InitialRecValues { get; } //Mit dem SizeFactor/Angle/Pivot werden L
    11 Verweise
    IPrototypExportData EditorExportData { get; } //Mit diesen Daten kann der Editor der dieses Item
    5 Verweise
    Bitmap GetImage(int maxWidth, int maxHeight);
    7 Verweise
    void Draw(GraphicPanel2D panel); //Zeichnet das Objekt im Bereich von X=0..BoundingBox.Width und '
    6 Verweise
    void DrawBorder(GraphicPanel2D panel, Pen borderPen);
    6 Verweise
    void DrawWithTwoColors(GraphicPanel2D panel, Color frontColor, Color backColor);
}

PhysicLevelItem.cs
public void Draw(GraphicPanel2D panel)
{
    57
    58
    59
    60
    61
    62
    63
    4 Verweise
    panel.PushMatrix();
    panel.MultTransformationMatrix(this.RotatedRectangle.GetLocalToScreenMatrix());
    prototyp.Draw(panel);
    panel.PopMatrix();
    6
}
  
```

Right pane:

```

02_PhysicItemEditor
  PhysicItemEditorControl
    Abhängigkeiten
    Model
    View
      PhysicItemControl.xaml (highlighted with red box)
  
```

Nachdem einige LevelItems platziert wurden erfolgt die Simulation durch die EditorFunction „SimulatorFunction“, welche den EditorState in ein EditorDataForSimulation (2)-Objekt umwandelt, was wiederum vom SimulatorExporter (3) in ein SimulatorInputData-Objekt (4) konvertiert wird.

```

    public void Restart()
    {
        this.simulator = new SimulatorWithForceTracking(SimulatorExporter.Convert(GetSimulatorInputData()), this.state.Panel, state.Camera, state);
    }

    public class SimulatorWithForceTracking : Simulator
    {
        private PhysicsSceneIndexDrawer indexDrawer; //Fürs ForceTracking
        private ForceTracker forceTracker; //Zeichnet die Kräfte auf, die auf die Körper und Gelenke wirken
    }

```

The screenshot shows a code editor with `SimulatorFunction.cs` open. The code defines a `Restart` method and a `SimulatorWithForceTracking` class. A tooltip for `GetSimulatorInputData` points to `LevelEditorControl.EditorFunctions.SimulatorFunction`. To the right, the Solution Explorer shows the `03_LevelToSimulator` project with its files: `LevelToSimulatorConverter`, `Abhängigkeiten`, `EditorToPhysicsSceneMapping`, `EditorDataForSimulation.cs`, `KeyboardHandlerProvider.cs`, `PhysicsSceneMerger.cs`, `SimulatorExporter.cs`, `TagDataConverter.cs`, and `Vec2Extension.cs`.

Der Grund, warum für die Simulation die Editor-Daten vorher noch konvertiert werden müssen ist, weil aus Editor-Sicht besteht ein Level aus lauter Einzel-Physicscene-Objekten. Der Simulator benötigt aber nur ein einziges Physicscene-Objekt. Die Aufgabe vom `LevelToSimulatorConverter` ist es mehrere Physicscene-Objekte zu einem großen Objekt zu mergern.

Der Simulator kann sowohl direkt aus dem Editor benutzt werden aber er wird auch von der Spielelogik benutzt. Dazu wird eine `LevelEditor`-Datei in ein `SimulatorInputData`-Objekt konvertiert und damit dann ein Simulator-Objekt erzeugt werden, dem ich von Außen auch die `Camera2D` mitgeben kann. So kann ich dann im Spiel selber festlegen, welchen Kamera-Zoom ich gerade verwenden will oder ob die Kamera gerade im Objekt-Tracking-Modus läuft oder ob ihre Position fix ist.

```

46     private void Load(string file)
47     {
48
49         var simulatorInputData = EditorFileConverter.Convert(DataFolder + file);
50
51         this.camera = new Camera2D(panel.Width, panel.Height);
52         this.simulator = new Simulator(simulatorInputData, panel, this.camera, timerIntervallInMilliseconds);

```

Erzeugung des LevelEditorViewModels im UnitTest

Das ViewModel vom `LevelEditor` erzeugt ein Uri-Objekt, welches auf eine Bilddatei aus einer Dll-Resource verweist. Das wird benötigt, damit man das Bild, was bei einen Button angezeigt wird abhängig von ein internen Zustand vom ViewModel ändern kann.

Wenn man ein `ContentControl` in der View nutzt, dann muss der `Content`-Property per Binding ein `UserControl` zugewiesen werden. Aus dem Grund erzeuge ich innerhalb des ViewModel ein `UserControl`.

Die Nutzung von Dll-Resourcen und die Erzeugung von `UserControls` innerhalb eines UnitTests wirft aber Exceptions, welche mit ein paar Tricks innerhalb der Testklasse aber verhindert werden können. Hier sind die Hinweise, um Uri- und `ContentControl`-Nutzung zu ermöglichen:

```

30     public DemoGameTests()
31     {
32         //Wenn ich die Tests hier laufen lasse, bekomme ich im LevelEditor folgende Exception:
33         //System.UriFormatException: "Invalid URI: Invalid port specified."
34         //Das ViewModel vom Editor nutzt für die Buttons eine Bilddatei, die per Uri-Objekt auf eine Assembly-Resource verweist:
35         //pack://application:,,,/LevelEditorControl;component/Controls/EditorControl/DoubleClick.png
36         //Mit folgender Anweisung verhindere ich, dass die "Invalid port specified"-Exception kommt:
37         //Schritt 1: Verhindere diese Exception: System.UriFormatException: "Invalid URI: Invalid port specified."
38         if (!UriParser.IsKnownScheme("pack")) //Diese Zeile verhindert, dass die Exception "System.InvalidOperationException : A URI scheme name 'p
39             UriParser.Register(new GenericUriParser(GenericUriParserOptions.GenericAuthority), "pack", -1);
40
41         //So geht es ab .NET 5.0
42         SetResourceAssembly(typeof(LevelEditorFactory).Assembly); //Schritt 2: Verhindere diese Exception: System.NotSupportedException: "The URI p
43         //So ging es bei .NET Framework: System.Windows.Application.ResourceAssembly = typeof(LevelEditorFactory).Assembly;
44
45         //Schritt 3: Da das ViewModel vom LevelEditor ein UserControl-Objekt erzeugt, kommt die Exception: System.InvalidOperationException: "Beim
46         //Um zu verhindern, dass bei ein XUnit-Test diese Exception kommt muss das NuGet-Paket 'Xunit.StaFact' installiert werden.
47         //Wenn man dann [StaFact] anstatt [Fact] über den Test schreibt, dann kommt die Exception nicht mehr. Quelle:
48         //http://dontcodetired.com/blog/post/Running-xUnit-Tests-on-Specific-Threads-for-WPF-and-Other-UI-Tests
49
50         //Schritt 4: Der Aufruf von SetResourceAssembly als auch die Nutzung von [StaFact] darf nicht parallel erfolgen.
51         //Deswegen muss über allen Testklassen, wo SetResourceAssembly oder [StaFact] genutzt wird mit dem Attribut
52         //|[Collection("Our Test Collection #1")] verhindert werden, dass diese Testklassen parallel laufen.
53     }

```

Siehe: [DemoApplications/UnitTests/DemoApplication.UnitTests/DemoGameTests.cs](#)

Unterprojekte

Folgende Unterprojekte gibt es jetzt bzw soll es noch geben:

RigidBodyPhysics = Position von Starrkörpern bestimmen

- PhysicSceneEditor
- PhysicSceneSimulator

PhysicControl = Sollwert von Joints festlegen

- KeyFrameAnimator
- Keyboard (Mapping von KeyDown/KeyUp-Event auf Action)
- Keyboard Record/Play
- ArmControl über Dictionary (Zielpunkt->Gelenkwinkel), das durch probieren ermittelt wurde
- KIControl (NEAT, PPO)

Visualisizer = Alle Grafik-Berechnungen, die nicht Teil der Grafikengine sind

- TextureEditor = UV-Koordinaten und Texture-Matrix definieren

Leveleditor = Platzieren, Gruppieren

- Vordergrundobjekte platzieren (Blockmode)
- Polygone erzeugen (Wie bei Elma)
- Hintergrundobjekte: Z-Abstand definiert, um wie viel schneller/langsamer es sich im Bezug zum Vordergrund beim Scrollen bewegt
- Objekte gruppieren: Über Baum sehe ich welches Objekt mit wem gruppiert ist
- Von jedem Objekt(Objektgruppe) sein Center-Punkt definieren, um es somit in das Blockraster zu bringen

Gameengine = Vereint Grafik, Physik, Sound und KI

- Klasse, welche ein Timer, PhysicEngine, GrafikPanel, SoundHandler und virtuelle Handler für Timer, KeyPress, Mouse, PhysicEvents hat
- Diese Klasse lädt eine Datei vom Leveleditor und ruft die Physic-TimeStep-Methode auf und zeichnet das Level

Zusammenfassung von Teil 7

Es wurde hier ein Leveleditor entworfen mit dem man eine Physikszenen erstellen kann und sie als Textdatei speichern kann. Im nächsten Abschnitt kommt dann noch der SpriteEditor zur Spriterzeugung und die GameSimulator-Klasse, welcher die Level-Textdatei einliest und anzeigt.

Teil 8: Sprites

Ziel dieses Abschnitts

Hier soll das Teilprojekt Engine/06_SpriteEditor erklärt werden.

Spriteeditor

Mit dem Spriteeditor kann man Sprite-Bilddateien erzeugen. Es gibt zwei Arten von Spritebildern die man erzeugen kann:

- Man will ein Arm oder ähnliches erzeugen, welcher an ein vorgegebenen Pivotpunkt hängt und welcher sich bewegt.
- Es soll ein Objekt/Person erzeugt werden, die sich bewegt und die wegen der Schwerkraft auf ein flachen Boden läuft.

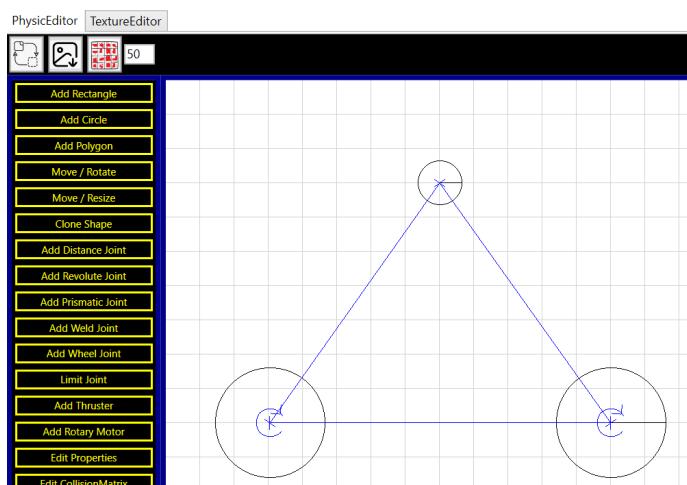
Der Spriteeditor kann über Batch-Files: `Spriteeditor.bat` gestartet werden.

Beispiel: Motorrad-Fahrer

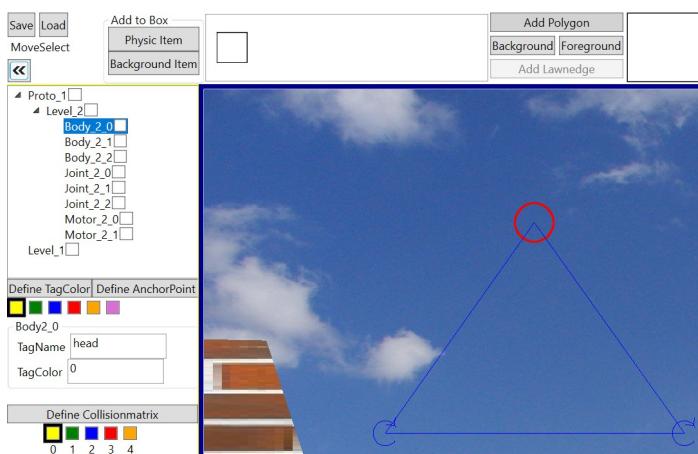
Es soll hier ein Motorrad-Fahrer erzeugt werden, dessen Physikmodel aus 3 Kreisen und 3 Federn besteht. An diesem Physikmodel hängt dann der Fahrer dran, welcher seine Arme per Sprite-Animation bewegen soll.

Schritt 1: Zuerst wird das Physikmodel im Leveleditor erzeugt

Starte unter Batch-Files den Leveleditor und erzeuge folgendes PhysicItem:



Lege für den Kopf den TagName „head“ und für die Räder „wheel1“ und „wheel2“ fest:



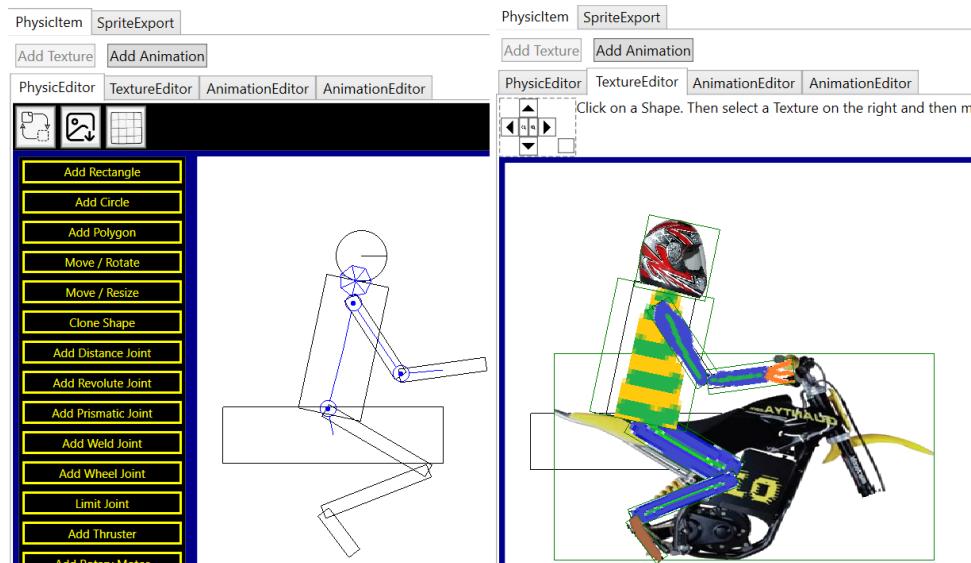
Du kannst beim DemoGame Elma im Ordner Levels/EmptyLevel.txt dir das hier erzeugte Level mit dem Motorrad auch ansehen.

Schritt 2: Im Spriteeditor den Fahrer erstellen

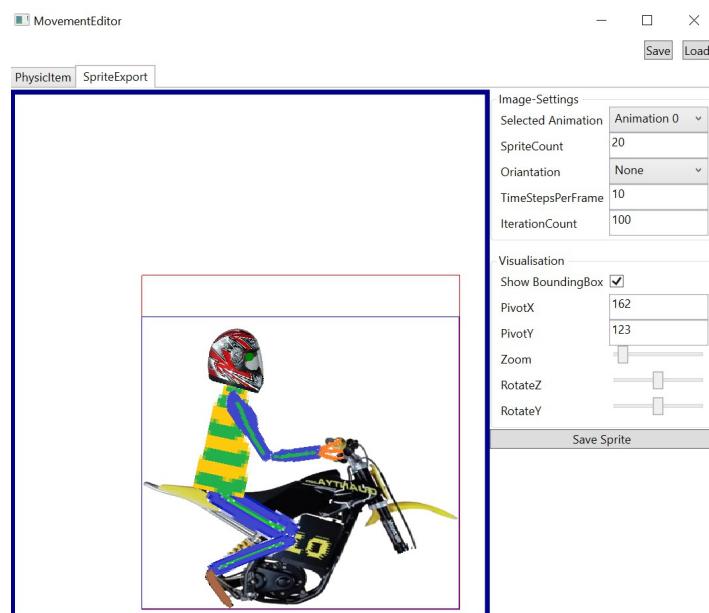
Starte nun unter Batch-Files den Spriteeditor und erstelle den im Bild gezeigten Fahrer. Bei den DemoGame Elma siehst du unter Motorbike_Sprite.txt die hier erstellte Sprite-Projektdatei.

Die Räder dürfen nicht im Spriteeditor mit auftauchen, da ihre Position sich abhängig vom Physikmodel ändern kann. Der Kopf von der Sprite-Datei soll dann an dem Kopf vom Physikmodel befestigt werden.

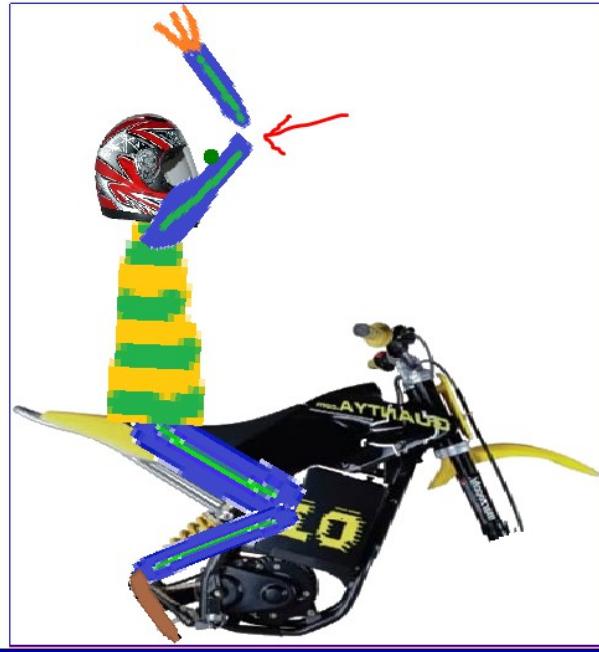
Die Erstellung des Physikmodels, die Texturierung und Animation erfolgt so, wie man es auch im Leveleditor machen würde. Wichtig: Das Gehäuse vom Motorrad muss fix sein, da es ansonsten beim animieren aus dem Bild fallen würde.



Im SpriteExport-Tab wird dann die Spritedatei erzeugt. Unter „Selected Animation“ muss das Animation-Tab ausgewählt werden, wofür man eine Spritedatei erzeugen will. Bei SpriteCount wird die Anzahl der Einzelbilder definiert. Bei Orientation wird festgelegt, an welcher Stelle die Bilddaten innerhalb von jeden Einzelbild der Spritedatei sein sollen. None heißt, es erfolgt keine Ausrichtung. Die Sprite-Datei zeigt dann das gleiche an, was auch das Animations-Tab anzeigt, wenn man die Animation abspielen lässt. Würde man hier Bottom auswählen, dann „fallen“ alle Einzelbilder auf den Fußboden.

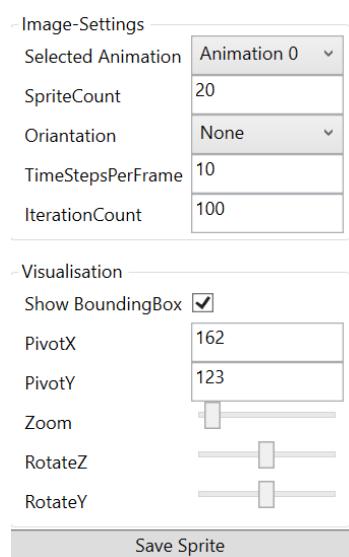


Mit TimeStepsPerFrame wird festgelegt, wie viele TimerTicks die Animation während der Animation des Physikmodels auf jeder FramePosition warten soll, bevor es zum nächsten Frame springt. Wenn diese Zahl zu gering ist, dann kann es passieren, dass die Gelenke bei schnellen Bewegungen auseinander fliegen. Hier in dem Beispiel wurde TimeStepsPerFrame auf 1 gesetzt. Dieses Auseinanderfliegen ist das gleiche, was auch passiert, wenn man den Gelenksollwert plötzlich um ein großen Wert ändert und der Arm dann nicht sofort an die richtige Position springt sondern er weg geschleudert wird und dann erst ein Frame später an der richtigen Position ist.



IterationCount ist der PGS-IterationCount-Parameter von der PhysicScene. Nur die Einstellungen unter „Image-Settings“ haben eine Auswirkung darauf, wie die Sprite-Bilddatei, die hier erzeugt wird dann aussieht.

Über die Einstellungen unter „Visualisation“ kann dann die Größe und Ausrichtung von dem Bild verändert werden. Der Nullpunkt des Bildes ist der Pivot-Punkt. Im ScreenShoot oben ist dass der grüne Punkt. Diese Punkt wird dann am Physikmodel befestigt.



Schritt 3: Sprite-Bilddatei erzeugen

Die Sprite-Datei kann nun entweder über „Save Sprite“ erzeugt werden was dann so aussieht:



Es ist aber auch möglich, dass man die Projektdatei (Textdatei) vom Spriteeditor nimmt und dann aus dem Programmcode heraus die Spritedatei erzeugt:

Parameter mit Wert 0 bedeutet hier: Animationstab mit Index 0 soll erstellt werden

```
76     ||| SpriteEditorControl
77         .SpriteDataCreator
78             .CreateFromSpriteEditorFile(dataFolder + "Motorbike_Sprite.txt", 0)
79             .Image
80             .Save(dataFolder + "Sprite1.png");
```

Siehe: DemoApplications/Elma/ElmaControl/Controls/Main/MainViewModel.cs

Der Vorteil hiervon ist, dass wenn man im Spriteeditor etwas verändert und man hat mehrere Animationen, dann muss man nicht per Hand dann all die Sprite-Dateien dann alle einzeln neu erzeugen.

Schritt 4: Die Sprite-Datei verwenden

Zur Anzeige der Spritedatei gibt es die SpriteImage-Klasse. Ihr muss im Konstruktor gesagt werden, aus wie vielen Einzelbildern die Sprite-Datei besteht und wie lange die Animation dauern soll. Außerdem ob es die Animation nur einmal abspielen soll oder zyklisch. Es muss auch angegeben werden, welcher Pixel des Sprite-Einzelbildes der Pivotpunkt sein soll. All diese Angaben sind Bestandteil von der Projektdatei des Spriteeditors. Deswegen wird diese hier neben der Bilddatei mit ausgelesen um somit dann die SpriteImage-Objekte (für jede Animation eins) zu erzeugen.

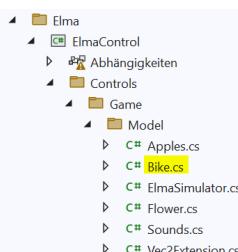
```
66     ||| var spriteData = JsonHelper.Helper.CreateFromJson<SpriteEditorExportData>(File.ReadAllText(dataFolder + "Motorbike_Sprite.txt")
67     var s = spriteData.SpriteData;
68     var a = spriteData.PhysicItemData.AnimationData;
69     this.armsUpSprite = new SpriteImage(dataFolder + "Sprite1.png",
70         s.SpriteCount, //XCount
71         1, //YCount
72         s.SpriteCount, //ImageCount
73         a[0].AnimationData.DurrationInSeconds, //DurrationInSeconds
74         true, //animateOneTime
75         s.PivotX / ScaleFactor, //pivotX
76         s.PivotY / ScaleFactor); //pivotY
```

Siehe: DemoApplications/Elma/ElmaControl/Controls/Game/Model/Bike.cs

Die SpriteImage-Klasse hat eine Draw-Methode, wo man dann angibt, an welcher Stelle der Pivotpunkt dann gezeichnet werden soll. In unserem Beispiel hier wurde der Pivotpunkt so definiert, dass er in der Mitte vom Kopf liegt. Diese Punkt wird nun an die Position des Kopfes vom Physikmodell platziert. Über RotateZAngleInDegree kann die Ausrichtung des Bildes noch definiert werden. Das ergibt sich hier aus der Ausrichtung der Räder. Man kann auch nur ein einzelnes Frame ohne Animation zeichnen, wenn der Fahrer gerade nicht seine Arme bewegt.

```
167     var w1 = this.wheel1.Center.ToGrx();
168     var w2 = this.wheel2.Center.ToGrx();
169     float angleInDegree = Vector2D.Angle360(new Vector2D(1, 0), w2 - w1);
170     sprite.RotateZAngleInDegree = angleInDegree;
171     sprite.RotateYAngleInDegree = this.yAngleInDegree;
172
173     var pivot = this.head.Center.ToGrx();
174     if (this.arm == ArmState.NoArmMovement)
175     {
176         //Zeichne nur Frame 0 von der aktiven Sprite
177         sprite.DrawSingleImage(panel, pivot, 0);
178     }
179     else
180     {
181         //Zeichne die Armbewegung (hoch oder runter)
182         sprite.Draw(panel, pivot);
183     }
```

Dieses Beispiel ist bei den DemoGames im Elma-Projekt in der Bike-Klasse zu sehen:

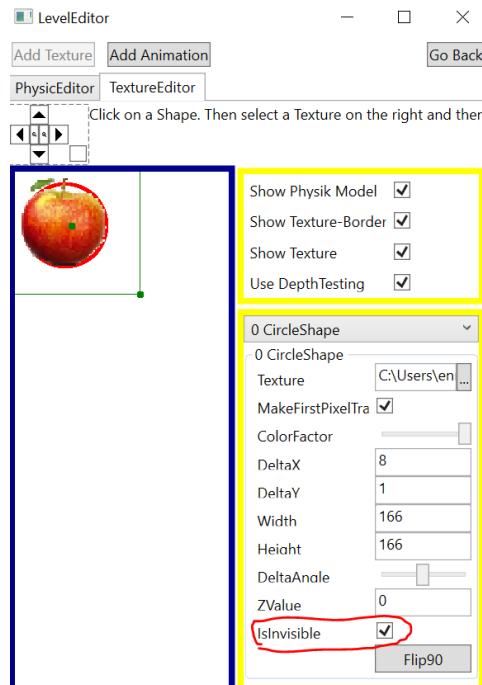


Anzeige von Sprites

Es gibt 3 Ansätze, wie man eine Sprite-Datei darstellen kann:

1 - Objekt im Editor unsichtbar machen

Man definiert im Leveleditor das Objekt so, dass man es auf „Invisible“ stellt:



Über den Tag-Name bekommt man eine Referenz auf den IPublicRigidCircle und über die SpriteImage-Klasse kann man dann die Sprite-Datei an der Stelle zeichnen, wo das Objekt gerade ist. Über den TimerTick-Handler wird die Sprite-Datei bewegt.

```
67     this.apples = simulator.GetBodiesByTagName("apple")
68     .Select(x => new Apple(simulator, (IPublicRigidCircle)x, dataFolder)).ToList();
69   }
70 
71   internal class Apple : ITimerHandler
72   {
73     private SpriteImage sprite;
74     public IPublicRigidCircle Body { get; }
75 
76     public Vector2D Center { get => this.Body.Center.ToGrx(); }
77     public float Radius { get => this.Body.Radius; }
78 
79     public Apple(Simulator.Simulator simulator, IPublicRigidCircle body, string dataFolder)
80     {
81       this.Body = body;
82       this.sprite = new SpriteImage(dataFolder + "apfel.png", 5, 1, false, 32, 32);
83 
84       this.sprite.Zoom = this.sprite.Width / this.Body.Radius;
85     }
86 
87     public void HandleTimerTick(float dt)
88     {
89       this.sprite.HandleTimerTick(dt);
90     }
91 
92     public void Draw(GraphicPanel2D panel)
93     {
94       this.sprite.Draw(panel, this.Body.Center.ToGrx());
95     }
96   }
```

Der Nachteil an diesen Ansatz ist, dass das Objekt wegen der IsInvisible-Eigenschaft im Leveleditor nicht angezeigt wird. Außerdem wird das Objekt auch nicht in der Minimap angezeigt.

2 - Anzeige von ein einzelnen Objekt als Sprite

Ich definiere ein Objekt, was das Interface IRigidBodyDrawer implementiert,

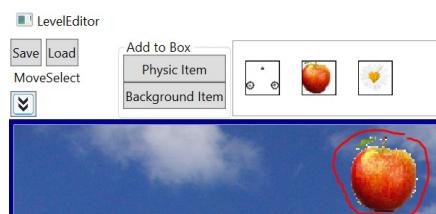
```
6  public interface IRigidBodyDrawer
7  {
8      5 Verweise
9      void Draw(GraphicPanel2D panel);
10     4 Verweise
11     void DrawBorder(GraphicPanel2D panel, Pen borderPen);
12     4 Verweise
13     void DrawWithTwoColors(GraphicPanel2D panel, Color frontColor, Color backColor);
14 }
15 }
```

und sage dem Simulator, dass er für das angegebene IPublicRigidCircle-Objekt den von mir angegebenen RigidBodyDrawer nutzen soll:

```
17 internal class Apple : ITimerHandler, IRigidBodyDrawer
18 {
19     private SpriteImage sprite;
20     10 Verweise
21     public IPublicRigidCircle Body { get; }
22
23     3 Verweise
24     public Vector2D Center { get => this.Body.Center.ToGrx(); }
25     3 Verweise
26     public float Radius { get => this.Body.Radius; }
27
28     1 Verweis
29     public Apple(Simulator.Simulator simulator, IPublicRigidCircle body, string dataFolder)
30     {
31         this.Body = body;
32         this.sprite = new SpriteImage(dataFolder + "apfel.png", 5, 1, false, 32, 32);
33
34         this.sprite.Zoom = this.sprite.Width / this.Body.Radius;
35
36         simulator.UseCustomDrawingForRigidBody(body, this);
37     }
38
39     1 Verweis
40     public void HandleTimerTick(float dt)
41     {
42         this.sprite.HandleTimerTick(dt);
43     }
44
45     3 Verweise
46     public void Draw(GraphicPanel2D panel)
47     {
48         this.sprite.Draw(panel, this.Body.Center.ToGrx());
49     }
50 }
```

Siehe: DemoApplications/Elma/ElmaControl/Controls/Game/Model/Apples.cs

Im Editor ist die IsVisible-Eigenschaft nun true. Da der Editor aber keine Sprite-Anzeige unterstützt kann man dort dann ein Einzelbild aus der Sprite-Datei anzeigen lassen und man sieht dann dieses Bild im Editor:



Der Vorteil an diesen Ansatz ist, dass man im Leveleditor das Objekt sieht und auch in der Minimap taucht das Objekt dann auf.

3 - Anzeige von mehreren Objekten als Sprite

Wenn ich ein Motorrad darstellen will, wo das Physikmodell aus drei Kreisen besteht, dann kann ich das so machen, dass ich die Sprite-Datei an den Kreis vom Kopf dran klebe und die Ausrichtung vom Spritebild ergibt sich durch die Position der Räder. Für diesen Fall nutze ich die UseCustomDrawingForRigidBody-Methode, um dem Kopf die Sprite-Datei hinzufügen und über RemoveBodyFromPhysicsSceneDrawer verhindere ich, dass die Räder vom PhysicsSceneDrawer gezeichnet werden.

```
16  internal class Bike : ITimerHandler, IRigidBodyDrawer
17
18  public Bike(Simulator.Simulator simulator, string dataFolder, GraphicPanel2D panel)
19  {
20      this.dataFolder = dataFolder;
21      this.head = (IPublicRigidCircle)simulator.GetBodyByTagName("head");
22      this.wheel1 = (IPublicRigidCircle)simulator.GetBodyByTagName("wheel1");
23      this.wheel2 = (IPublicRigidCircle)simulator.GetBodyByTagName("wheel2");
24
25      simulator.UseCustomDrawingForRigidBody(this.head, this);
26      simulator.RemoveBodyFromPhysicsSceneDrawer(this.wheel1);
27      simulator.RemoveBodyFromPhysicsSceneDrawer(this.wheel2);
28
29  }
30
31  public void Draw(GraphicPanel2D panel)
32  {
33      var pivot = this.head.Center.ToGrx();
34      var w1 = this.wheel1.Center.ToGrx();
35      var w2 = this.wheel2.Center.ToGrx();
36      float angleInDegree = Vector2D.Angle360(new Vector2D(1, 0), w2 - w1);
37
38      this.armsUpSprite.RotateZAngleInDegree = this.armsDownSprite.RotateZAngleInDegree = angleInDegree;
39      this.armsUpSprite.RotateYAngleInDegree = this.armsDownSprite.RotateYAngleInDegree = this.yAngleInDegree;
40
41      panel.DisableDepthTesting();
42      panel.ZValue2D = 0;
43      DrawSticks(panel, pivot, w1, w2);           //Zeichne die Linien von den Rädern zum Gehäuse
44      DrawSprite(panel, pivot);                  //Zeichne die Sprite-Datei
45      DrawWheels(panel, w1, w2);                 //Zeichne die Räder
46
47      panel.EnableDepthTesting();
48  }
```

Siehe: DemoApplications/Elma/ElmaControl/Controls/Game/Model/Bike.cs

Teil 9: Dynamische Objekterzeugung

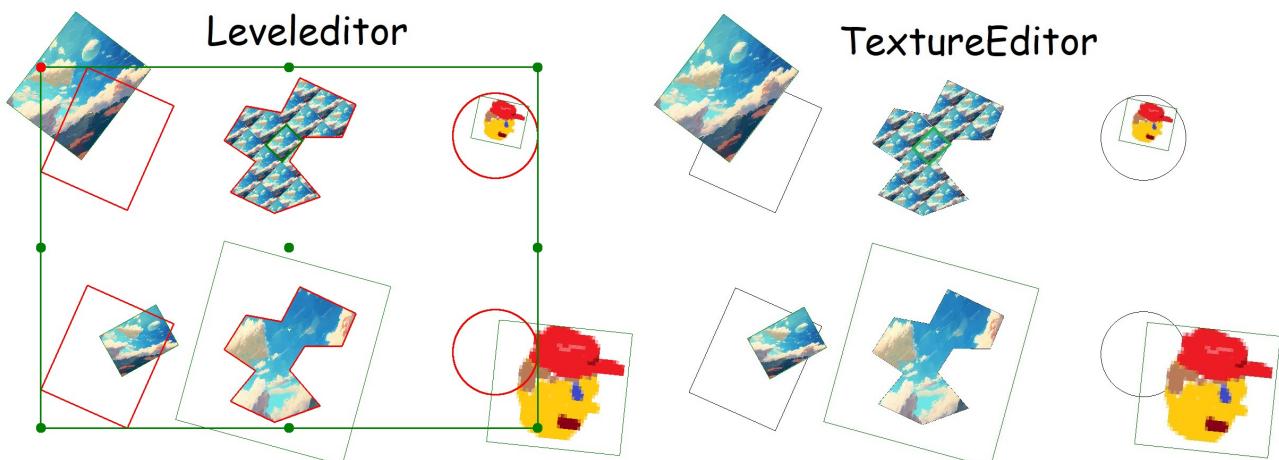
Ziel dieses Abschnitts

In den beiden vorherigen Abschnitten wurde Editoren zur Level- und Spriteerstellung erklärt. Man kann im Leveleditor eine Physiksimulation abspielen lassen aber es ist nicht möglich während der Simulation neue Objekte zu erzeugen oder vorhandene Objekte so zu zerstören, dass sie in Bruchstücke zerfällt. Hier in diesen Abschnitt soll das Teilprojekt Engine/07_DynamicObjCreation erklärt werden. Es bietet Hilfsfunktionen zur Objekterzeugen/Dupplizierung/Zerstörung von Physikobjekten. Diese Bibliothek wird dann vom GameSimulator verwendet, welcher im nächsten Abschnitt erklärt wird.

Objekte zerlegen

Es sollen Objekte per Voronoi zerlegt werden. Aus Physik-Sicht gesehen besteht ein Objekt entweder aus ein Rechteck, Polygon oder Kreis. Aus Textur-Sicht gesehen besteht ein Objekt aus ein rotierten Rechteck oder aus ein Polygon. An den Physik-Rechteck und Physik-Kreis hängt jeweils ein Textur-Rechteck. An den Physik-Polygon hängt ein Textur-Polygon.

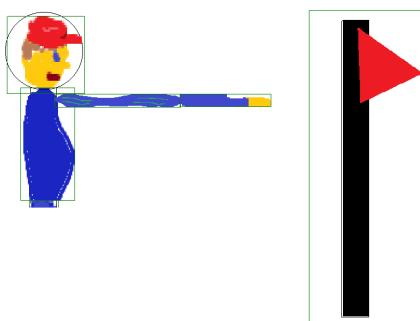
So sehen die Physik- und Texturobjekte aus:



Linkes Bild: roter Rahmen = Physikobjekt; grüner Rahmen (dünne Linie) = Texturrechteck

Rechtes Bild: schwarzer Rahmen = Physikobjekt; grüner Rahmen = Texturrechteck

Bei den Polygonen gilt also, dass die Form vom Physik- und Textur-Objekt genau gleich ist. Bei den Rechteck und Kreis kann die Textur entweder über den Physik-Rand hinaus ragen. Zum Beispiel bei der Fahne. Beim Kopf ragt die Textur teilweise über den Kreis hinaus und teilweise ist der Kreis außerhalb vom Texturrechteck.

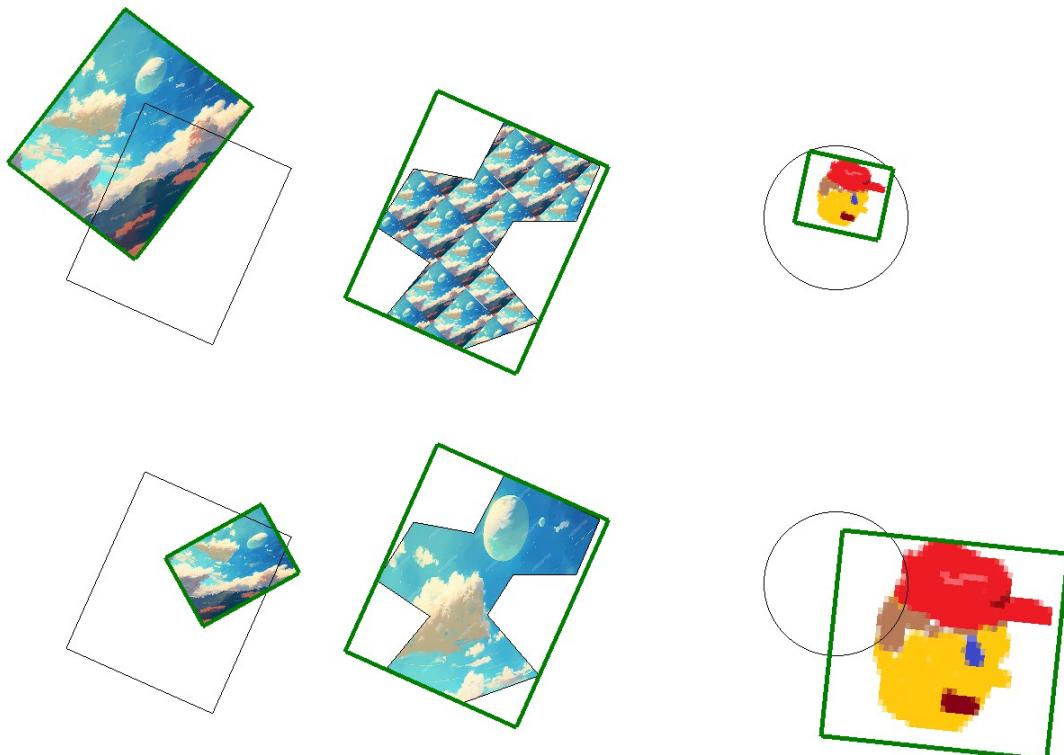


Wenn ich das Textur-Objekt zerlege und möchte, dass die Bruchstücke dann Teil der Physiksimulation sind, dann muss jedes Textur-Bruchstück auch ein eigenes Physik-Objekt haben. Damit diese Simulation realistisch aussieht, wird bei den Rechteck- und Kreis-Texturen das Physikobjekt nicht weiter beachtet da all die Physikobjekt-Bereiche, die nicht von der Textur bedeckt sind ja unsichtbar sind. Dessen Bewegung zu simulieren macht also keinen Sinn.

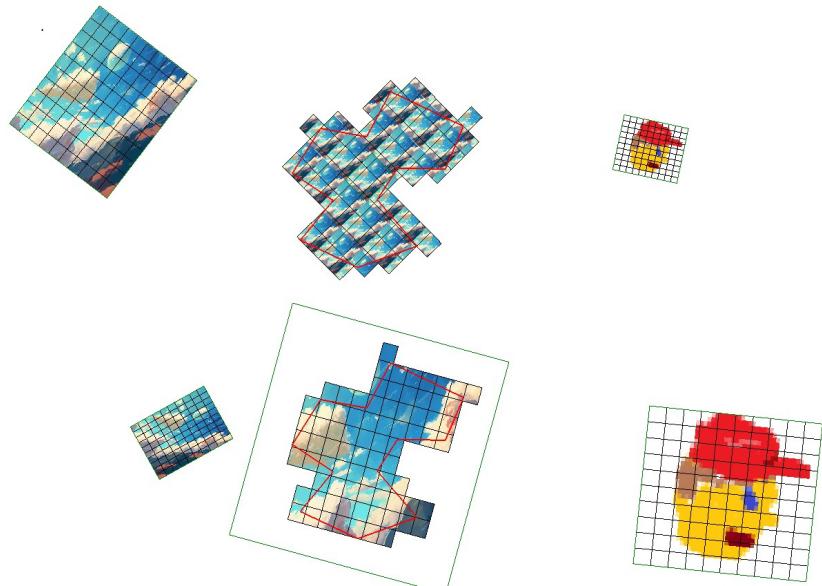
Stattdessen wird nur an der Stelle vom Textur-Rechteck ein neues Physik-Rechteck erzeugt und dieses Rechteck wird dann zerlegt. Bei den Polygonen ist das Physik- und Textur-Objekt bereits genau übereinander liegend. Deswegen kann dort das Polygon direkt zerlegt werden.

Objekte in Kästchen zerlegen

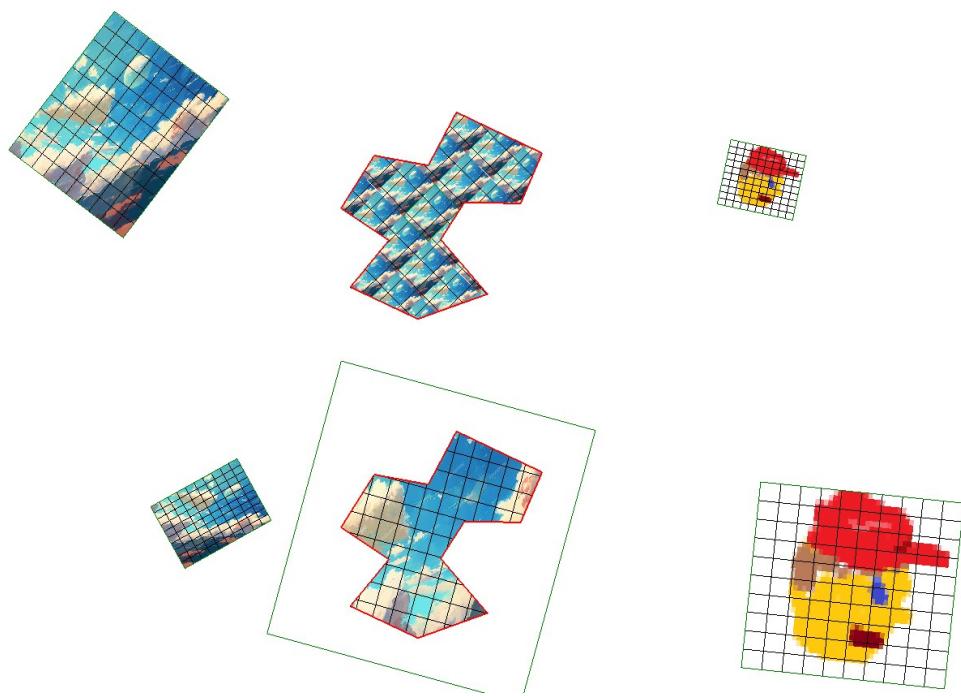
Es sind folgende 6 Objekte mit Textur gegeben, welche in Kästchen zerlegt werden sollen. Bei den Rechteck und Kreis wird einfach das grüne Textur-Rechteck genommen und zerlegt. Bei den Polygon wird eine Bounding-Box um das Polygon gespannt, was von der Ausrichtung her so ist wie das Texturrechteck. Diese rotierte Boundingbox dient zur Grundlage für die Zerlegung.



Die Textur vom Rechteck und Kreis kann ohne Probleme in gleichmäßige Boxen unterteilt werden, da sie Rechteckig sind. Bei den Polygon wird hinter das Polygon ein Kachelmuster gelegt, was von der Ausrichtung dem Texturrechteck entspricht. Es müssen nun alle Kästchen, die außerhalb vom roten Polygon sind weg geschnitten werden. In der ersten Stufe werden all die Kästchen entfernt, die komplett außerhalb vom Polygon liegen.



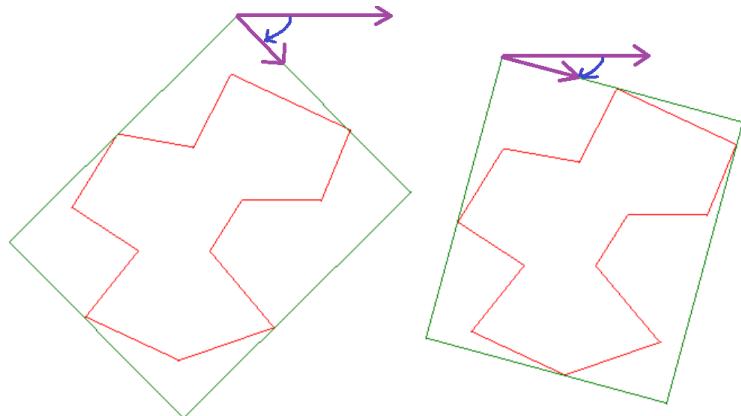
Jedes Kästchen ist ein eigenes Polygon. Es wird nun die Schnittmenge zwischen den roten Ausgangspolygon und den Kästchen-Polygonen gebildet, um somit nun Physik-Polygone zu erzeugen, die entweder wie ein Kästchen aussehen oder aber wie ein Kästchen, wo ein Teil weg geschnitten wurde.



Für die Kästchenzerlegung von Polygonen sind zwei Teilfunktionen wichtig: rotierte BoundingBox und Schnittmenge zweier Polygone. Beide Teilfunktionen werden in den nächsten Abschnitten erklärt.

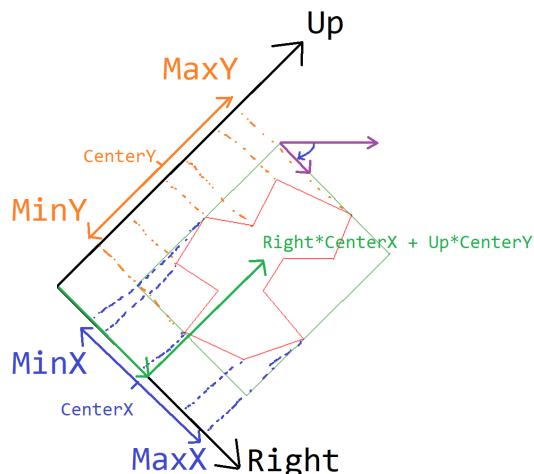
Rotierte Bounding-Box

Es ist ein Polygon (rot) gegeben und der Winkel (blau) für den X-Kanten-Vektor (lila)



Gesucht ist die grüne Bounding-Box.

Es wird dazu ein Right-Vektor (1,0) um den blauen Winkel gedreht und diese Vektor um 90 Grad gedreht um ein Up-Vektor zu erzeugen.



Nun werden alle Polygonpunkte auf die beiden Richtungsvektoren Right und Up projiziert und somit die Min-Max-Werte für die jeweilige Achse bestimmt. Die Kantenlänge der Boundingbox erhalte ich durch Max minus Min und das Zentrum durch Right*CenterX+Up*CenterY

```

39  private static void GetRotatedBoundingBoxFromPoints(Vec2D right, Vec2D up, IEnumerable<Vec2D> points, out Vec2D center, out SizeF size)
40  {
41      var rightPoints = ProjectPointsOnDirection(right, points);
42      var upPoints = ProjectPointsOnDirection(up, points);
43      GetMinMaxCenter(rightPoints, out float minX, out float maxX, out float centerX);
44      GetMinMaxCenter(upPoints, out float minY, out float maxY, out float centerY);
45
46      center = right * centerX + up * centerY;
47      size = new SizeF(maxX - minX, maxY - minY);
48  }
49
50  2 Verweise
51  private static float[] ProjectPointsOnDirection(Vec2D rayDir, IEnumerable<Vec2D> points)
52  {
53      return points.Select(x => x * rayDir).ToArray();
54  }
55  2 Verweise
56  private static void GetMinMaxCenter(float[] values, out float min, out float max, out float center)
57  {
58      min = values.Min();
59      max = values.Max();
60      center = (max + min) / 2;
61  }

```

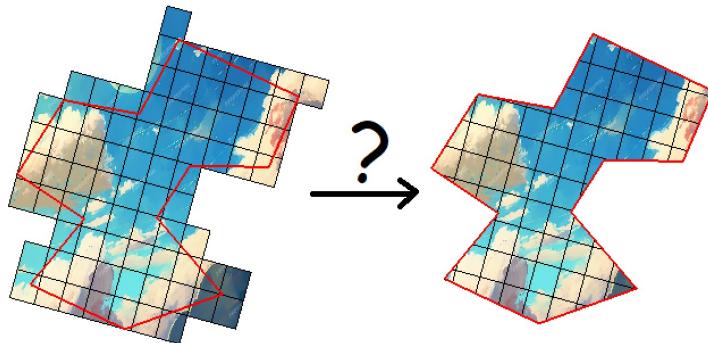
Siehe: Engine/07_DynamicObjCreation/RigidBodyDestroying/RotatedBoundingBox.cs

Diese rotierte Box wird genutzt, damit man zuerst die Kästchen im unrotierten Zustand erzeugt und dann diese unrotierten Boxen dann zum Mittelpunkt der rotierten Boundingbox verschiebt und sie dann noch dreht.

Schnittmenge zwischen zwei Polygonen

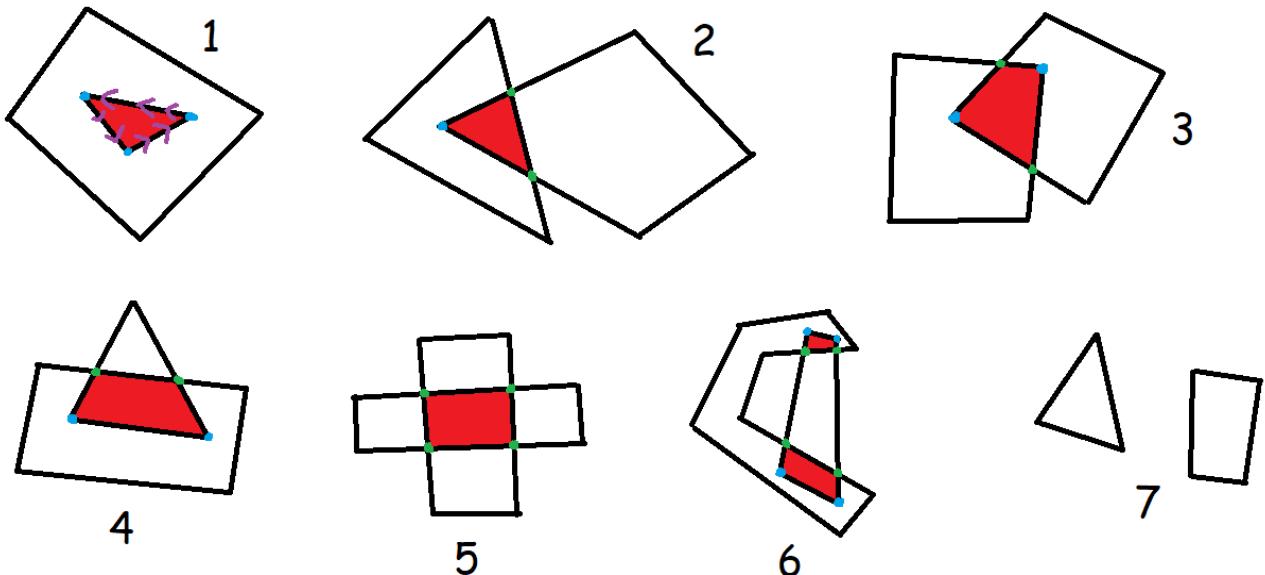
Gegeben ist das rote Polygon und lauter rotierte Kästchen, die entweder komplett oder Teilweise im

Polygon liegen. Gesucht ist die Schnittmenge zwischen den Kästchen und dem Polygon.



Eine Möglichkeit um die Schnittmenge zu bilden wäre, dass man über alle Pixel iteriert und dann die `Polygon.PointIsInside`-Methode nutzt um zu prüfen, ob der jeweilige Pixel sowohl im roten Polygon liegt als auch in einer der schwarzen Kästchen. Da die Physikengine aber die Starrkörper in Form von Rechtecken, Kreisen oder Polygonen benötigt, wird eine Funktion benötigt, welche die Schnittmenge nicht als Menge von Pixeln sondern als Polygon zurück gibt.

Hier sind ein paar Beispiele, wo jeweils zwei Polygone gegeben sind die sich teilweise überschneiden. Die Schnittmenge ist rot markiert. Gesucht sind die roten Polygone.



Um diese Polygone zu finden wird zuerst geprüft, ob die Eckpunkte der schwarzen Polygone jeweils im anderen Polygon liegen. Wenn ja, werden sie blau markiert. Dann wird für jede Kante des einen schwarzen Polygons geprüft, ob es ein Schnittpunkt mit einer Kante vom anderen Polygon gibt. Wenn ja, wird dieser Kantenschnittpunkt grün markiert.

Gibt es keine blauen und grünen Punkte (Beispiel 7) dann gibt es keine Schnittmenge und es wird null zurück gegeben. Liegen alle Punkte des einen Polygons innerhalb des anderen Polygons und gibt es keine Schnittpunkte mit den Kanten (Beispiel 1), dann wird das innere Polygon zurück gegeben.

Wenn es aber Kantenschnittpunkte (grüne Punkte) gibt, dann gibt es folgende Ansätze:

Ansatz 1:

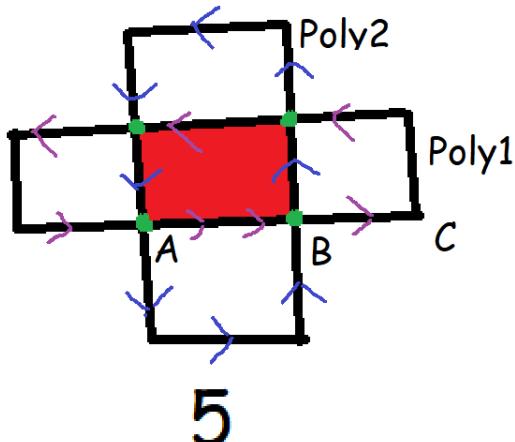
Eine Idee, wie man das rote Polygon finden kann ist, wenn ich auf ein beliebigen blauen Punkt starte und dann laufe ich auf der Polygonkante bis zum nächsten blauen oder grünen Punkt entlang. Komme ich bei einem blauen Polygon-Eckpunkt raus, nehme ich einfach den nächsten Polygonpunkt von dem Polygon, auf dem ich gerade laufe. Komme ich bei einem grünen Punkt raus, dann wechsle ich das Polygon und laufe auf der Kante vom anderen Polygon weiter. Die Polygone sind beide

CCW angegeben. Jeder Punkt hat ein Flag, wo gesagt wird, ob er bereits besucht wird. Komme ich beim durchwandern bei einem Punkt an, wo ich schon mal war, dann wird die Suche abgebrochen und alle besuchten Punkte werden zurück gegeben.

Für die Beispiele 1 bis 4 funktioniert dieser erste Ansatz. Bei Beispiel 5 gibt es aber kein blauen Punkt, auf dem ich starten kann. Hier muss ich auf einen grünen Punkt starten.

Ansatz 2:

Die Beispiele 2 bis 6 sind alle Beispiele, wo es Kantenschnittpunkte gibt. D.h. Es gibt auf jeden Fall einen grünen Punkt. Der zweite Ansatz ist, dass man also nicht auf einen blauen sondern auf einen grünen Punkt startet.



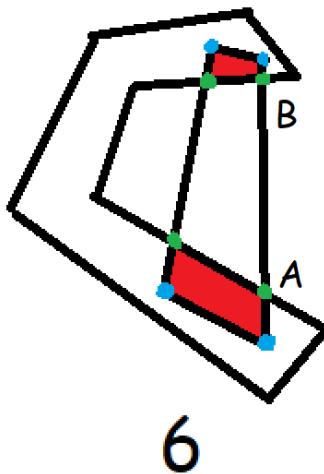
Wenn ich auf Poly1 beim Punkt A starte und dann in CCW-Richtung (lila Pfeile) laufe, dann komme ich beim Punkt B an. Dort gehe ich dann zu Poly2 und laufe auf Poly2 entlang der blauen Pfeile bis zum nächsten grünen Punkt. Dort erfolgt wieder der Polygonwechsel. Das mache ich, bis ich bei Punkt A rauskomme. Dort stelle ich dann fest, dass der Punkt schon besucht wurde und ich breche den Durchlauf ab und gebe die 4 besuchten Punkte als Polygon zurück.

Wenn ich aber zufälligerweise auf Punkt B starte und bei Poly1 in CCW-Richtung laufe, dann komme ich bei Punkt C raus. Dieser Punkt liegt aber außerhalb von Poly2 und ist kein Randpunkt von der roten Fläche. Wenn ich also feststelle, dass der zweite Punkt nach dem Startpunkt kein grün oder blau markierter Punkt ist, dann muss ich zum grünen Startpunkt zurück und dort muss ich zuerst einen Polygon-Wechsel machen und dann erst loslaufen. D.h. ich starte bei B und laufe diesmal nicht nach C sondern auf Poly2 zum oberen rechten grünen Punkt.

Dieser zweite Ansatz funktioniert nicht nur für Beispiel 5 sondern auch für die Beispiele 2 bis 5, da dort überall jeweils ein grüner Kanten-Schnittpunkt vorkommt auf dem ich starten kann. Bei Beispiel 6 hat aber Ansatz 2 ein Problem. Er würde z.B. bei Punkt A starten und dann nur die untere rote Fläche als Polygon zurück geben.

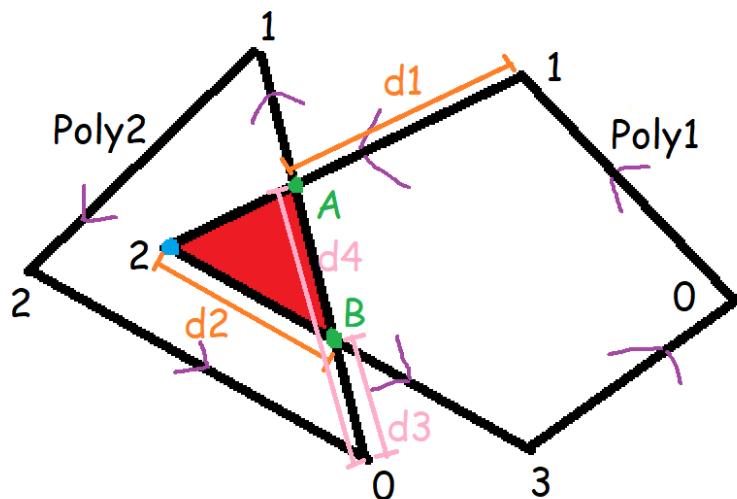
Ansatz 3:

Um auch solche Fälle abzudecken, wo es mehrere rote Teilflächen gibt, wird dieser Schleifendurchlauf mehrmals gemacht. Man startet immer jeweils bei einem grünen Punkt, wo man noch nicht war. Das macht man so lange, bis alle grünen Punkte besucht wurden. D.h. zuerst starte ich bei A und ermittle die 4 Polygonpunkte vom unteren roten Bereich. Der nächste grüne Punkt, wo ich noch nicht war, ist Punkt B. Dort ermittle ich wieder alle 4 Polygonpunkte vom oberen roten Bereich. Danach gibt es keine weiteren nicht besuchten grünen Punkte und die beiden gefundenen roten Polygone werden zurück gegeben.



Wenn ich auf ein Polygon-Eckpunkt stehe und wissen will, was der nächste Eckpunkt ist, dann kann ich das leicht ermitteln, da alle Eckpunkte eines Polygons in ein Array gespeichert sind. Wenn ich aber auf einen blauen Punkt stehe und ich will wissen, was der nächste grüne Kantenschnittpunkt ist, dann muss ich für jede Kante die Information speichern, ob sie grüne Punkte enthält und welchen Abstand sie zum Kanten-Startpunkt haben. Ein grüner Punkt liegt aber auf zwei Kanten. Deswegen muss der Abstand zu beiden Kanten-Startpunkten angegeben werden.

Beispiel: Poly1 hat zwei grüne Schnittpunkte mit Poly2.



Beim grünen Punkt A wird gespeichert, dass sie auf der Kante mit den Punktindizes 1-2 auf Poly1 liegt und dass sie auf dieser Kante den Abstand d1 hat. Außerdem liegt Punkt A auch auf der Kante 0-1 von Poly2 und hat dort den Abstand d4.

Punkt B liegt bei Poly1 auf der Kante 2-3 mit Abstand d2 und bei Poly2 auf der Kante 0-1 mit Abstand d3. Die Klasse BorderPoint speichert die Information dann für unser Beispiel auf folgende Weise:

```

5 //Schnittpunkt zwischen den Kanten Poly1[Poly1Edge]..Poly1[Poly1Edge+1] und Poly2[Poly2Edge] .. Poly2[Poly2Edge+1]
12 Verweise
6 internal class BorderPoint
{
    3 Verweise
8     public int Poly1Edge { get; } //index aus poly1
2 Verweise
9     public int Poly2Edge { get; } //index aus poly1
6 Verweise
10    public float T1 { get; } //Abstand zwischen this.Position und Poly1[Poly1Edge]
5 Verweise
11    public float T2 { get; } //Abstand zwischen this.Position und Poly2[Poly2Edge]
3 Verweise
12    public Vec2D Position { get; }
13    public bool Visited = false; //Wurde dieser BorderPoint schon genutzt, um damit ein Polygon zu bilden?
14
15    1 Verweis
16    public BorderPoint(int poly1Edge, int poly2Edge, float t1, float t2, Vec2D position)
17    {
18        Poly1Edge = poly1Edge;
19        Poly2Edge = poly2Edge;
20        T1 = t1;
21        T2 = t2;
22        Position = position;
23    }
}

```

Siehe: Engine/07_DynamicObjCreation/PolygonIntersection/BorderPoint.cs

```
var A = new BorderPoint(){Poly1Edge=1, Poly2Edge=0, T1=d1, T2=d4}
```

```
var B = new BorderPoint(){Poly1Edge=2, Poly2Edge=0, T1=d2, T2=d3}
```

Wenn ich nun auf ein grünen oder blauen Punkt von einer Polygonkante bin und ich will wissen, was der nächste grüne Punkt ist, dann muss ich für jede Polygonkante ein PolyEdge-Objekt anlegen und dieses Objekt kann mir dann für ein gegebenen Abstandswert t1/t2 sagen, was der nächste grüne BorderPoint ist.

```

3 //Polygonkante von Poly[StartPoint] .. Poly[StartPoint+1]
4 5 Verweise
5 internal class PolyEdge
6 {
7     private int startPointIndex;
8     private List<BorderPoint> borderPoints = new List<BorderPoint>();
9
10    public BorderPoint TryToGetNextT1Point(float t1)
11    {
12        foreach (var p in borderPoints)
13        {
14            if (p.T1 > t1)
15            {
16                return p;
17            }
18        }
19
20        return null;
21    }
22
23    1 Verweis
24    public BorderPoint TryToGetNextT2Point(float t2)
25    {
26        foreach (var p in borderPoints)
27        {
28            if (p.T2 > t2)
29            {
30                return p;
31            }
32        }
33
34        return null;
35    }
}

```

Siehe: Engine/07_DynamicObjCreation/PolygonIntersection/PolyEdge.cs

Die Information, auf welchen Polygon und auf welcher Kante und mit welchen Abstand zum Kantenstartpunkt ich stehe wird in der RunningPoint-Klasse gespeichert.

```

4 class RunningPoint
5 {
6     public int PolyIndex; //0 = Poly1; 1 = Poly2
7     public int EdgeIndex; //0..PolygonPoints.Length
8     public float T; //Abstand zum Punkt Polygon[EdgeIndex]
9

```

Siehe: Engine/07_DynamicObjCreation/PolygonIntersection/RunningPoint.cs

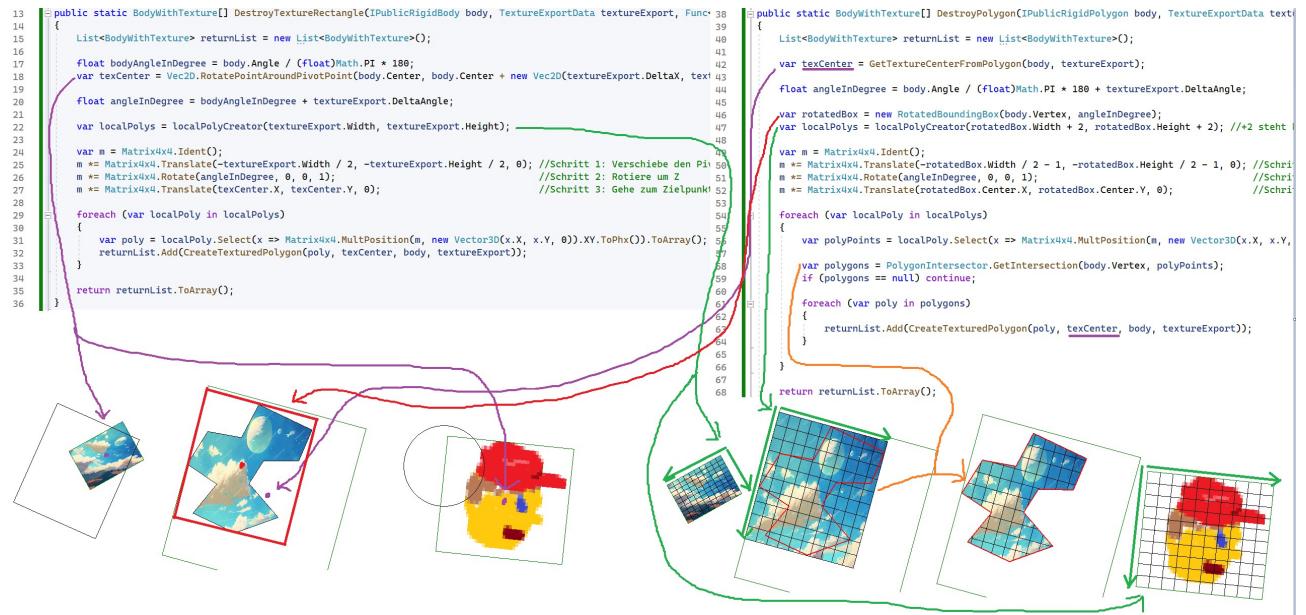
Dieser Punkt läuft nun auf den Polygonkanten entlang und springt dabei zum blauen Polygoneckpunkt oder zum grünen Polygonkantenschnittpunkt. Für Details siehe: Engine/07_DynamicObjCreation/PolygonIntersection/PolygonIntersector.cs

Objekte in Kästchen oder per Voronoi zerlegen

Nachdem nun gezeigt wurde, wie man eine rotierte Boundingbox und eine Schnittmenge zwischen zwei Polygonen bildet, soll nun gezeigt werden, wie Objekte in Kästchen oder per Voronoi zerlegt werden können. Die Destroy-Helper-Klasse hat zwei Funktionen für die Objektzerlegung. Die DestroyTextureRectangle-Funktion wird für Kreise und Rechtecke genutzt und die DestroyPolygon-Funktion für Polygone. Die beiden Funktionen sind vom Ablauf sehr ähnlich. Sie erzeugen in ein gedrehten Rechteck eine Menge von Boxen oder Voronoi-Polygonen. Beim Polygon erfolgt noch zusätzlich ein Clipping mit dem Polygonrand.

Die Unterschiede sind folgende: Beim Kreis und Rechteck kann die Textur (grünes Rechteck) völlig beliebig zum physikalischen Objekt platzieren werden. Wenn ich die Bewegung der Bruchstücke simulieren will, dann werde ich nur die Stücke sehen, welche eine Textur haben. Das Physikobjekt gilt als unsichtbar. Also wird beim Kreis und Rechteck nur das grüne Textur-Rechteck als Input genommen, um es dann zu zerlegen.

Beim Polygon gilt immer: Da wo das Physikpolygon ist, da wird auch das Textur-Polygon angezeigt. Deswegen wird hier die rotierte Boundingbox vom Polygon als Input genommen.



Siehe: Engine/07_DynamicObjCreation/RigidBodyDestroying/DestroyHelper.cs

Linke Seite (Kreis/Rechteck):

Bei Zeile 18 wird die Mitte (lila Punkt) vom grünen Texturrechteck bestimmt.

Zeile 22: Innerhalb von diesen grünen Rechteck werden die Kästchen erzeugt indem zuerst im lokalen Raum von (0,0) bis (width,height) die Daten erzeugt werden und dann mit der Matrix m in das grüne Rechteck verschoben werden (siehe Bild unten Rechts)

Rechte Seite (Polygon):

Zeile 42: Es wird hier zwar die Mitte (lila Punkt) vom Texturrechteck bestimmt aber diese Information wird nur benötigt, um die Texturdaten für die erzeugten Kästchen zu bestimmen.

Zeile 46: Es wird die rotierte Boundingbox (rot) vom Polygon bestimmt und genau in dieser roten Box werden die Kästchen unter Nutzung der Matrix m erzeugt.

Zeile 58: Da die erzeugten Kästchen nur innerhalb des Polygons sein dürfen wird noch die Schnittmenge zwischen den Kästchen und dem Polygon bestimmt.

Anmerkung zu beiden Varianten

Beide Funktionen bekommen als Input eine localPolyCreator-Funktion. Diese Funktion legt dann fest, ob die Bruchstücke aus Kästchen oder aus Voronoi-Polygonen bestehen.

Verwendung des Objekt-Destroyer

Die GameSimulator-Klasse hat eine DestroyRigidBody-Methode, wo ich dann verschiedene Parameter übergeben kann:

- Zerstörung per Box/Voronoi
- Anfangswert für die Geschwindigkeit der Bruchstücke

```
88     this.ship = simulator.GetBodyByName("ship");
89
90     //Bsp 1: Zelege den Körper in Kästchen mit Defaultparametern
91     this.simulator.DestroyRigidBody(this.ship, IRigidDestroyerParameter.DestroyMethod.Boxes);
92
93     //Bsp 2: Unterteile das Objekt in 3x3 gleichgroße Rechtecke
94     this.simulator.DestroyRigidBody(this.ship, new DestroyWithBoxesParameter() { BoxCount = 3 });
95
96     //Bsp 3: Lass die Bruchstücke vom Zentrum wegschleudern
97     this.simulator.DestroyRigidBody(this.ship, new DestroyWithBoxesParameter()
98     {
99         //body = this.ship; poly = Bruchstück/Kästchen was erzeugt wurde
100        TransformFunc = (body, poly) =>
101        {
102            poly.Velocity += body.Center - poly.Center;
103            return poly;
104        }
105    });
127     this.simulator.DestroyRigidBody(this.ship, new DestroyWithVoronoiParameter()
128     {
129         CellCount = 10,
130         TransformFunc = (body, polyObj) =>
131         {
132             polyObj.Velocity += GetRandomVec(rand, 0.02f);
133             polyObj.AngularVelocity += (float)rand.NextDouble() * 0.1f;
134         }
135     });
136 }
```

Siehe: DemoApplications/Moonlander/MoonlanderControl/Model/ShipExploder.cs

Objekte duplizieren

Wenn man Gegner dynamisch während des Spiels erzeugen will oder man will die eigene Spielfigur zu ein anderen Ort beamen, dann ist es nötig, dass man zur Simulationszeit ein LevelItem duplizieren und löschen kann. Damit das geht, muss bekannt sein, welche Objekte alle zu einen LevelItem gehören.

So sieht der Ablauf aus, damit Objekt-Duplizierung möglich ist:

Der LevelEditor (EditorViewModel) hat ein EditorState-Objekt, was alle Objekte zur Design-Zeit speichert.

```
77     private EditorState editorData = new EditorState(); //Diese Variable wird der jeweils aktiven IEditorFunction-Funktion reingegeben
```

Aus dem EditorState-Objekt wird in der SimulatorFunction ein EditorDataForSimulation-Objekt erzeugt.

```

72     private EditorDataForSimulation GetSimulatorInputData()
73     {
74         var items = state.LevelItems.Where(x => x is IPhysicMergerItem).Cast<IPhysicMergerItem>().ToList();
75
76         //PhysicItems aus dem GroupedItemsLevelItem
77         items.AddRange(state.LevelItems
78             .Where(x => x is IPhysicSceneContainer)
79             .Cast<IPhysicSceneContainer>()
80             .SelectMany(x => x.GetPhysicMergerItems())
81         );
82
83
84         var backgroundItems = state.LevelItems
85             .Where(x => x is IBackgroundItemProvider)
86             .Cast<IBackgroundItemProvider>()
87             .SelectMany(x => x.GetBackgroundItems())
88             .ToList();
89
90         backgroundItems.AddRange(state.LevelItems
91             .Where(x => x is IBackgroundItem)
92             .Cast<IBackgroundItem>());
93
94         return new EditorDataForSimulation()
95     {
96         Items = items,
97         KeyboardMappings = state.KeyboardMappings.ToArray(),
98         CollisionMatrix = state.CollisionMatrixViewModel.CollideMatrix,
99         TimerIntervalInMilliseconds = state.TimerIntervalInMilliseconds,
100        BackgroundImage = state.PolygonImages.BackgroundImage,
101        ForegroundImage = state.PolygonImages.ForegroundImage,
102        HasGravity = state.HasGravity,
103        Gravity = state.Gravity,
104        IterationCount = state.SimulatorIterationCount,
105        CameraTrackedLevelItemId = state.CameraTrackedItem != null ? state.CameraTrackedItem.Id : -1,
106        Panel = state.Panel,
107        Camera = state.Camera,
108        CameraTrackerData = state.CameraTrackerData,
109        BackgroundItems = backgroundItems.Select(x => x.GetSimulatorExportData()).ToArray(),
110        TagData = state.GetAllTagsForSimulator()
111    };
112 }

```

Siehe: Engine/05_Leveleditor/05_Editor/LevelEditorControl/EditorFunctions/SimulatorFunction.cs

Dieses Objekt wird im PhysicItemConverter in ein PhysikLevelItemExportData-Array umgewandelt. Es enthält all die LevelItems, welche Teil der Physiksimulation sein sollen. Dabei werden aber nur die ExportDaten gespeichert. Siehe Bild:

```

304     internal static PhysikLevelItemExportData[] Convert(List<IPhysicMergerItem> items, string backgroundImage, string foregroundIm-
305     {
306         List<PhysikLevelItemExportData> returnList = new List<PhysikLevelItemExportData>();
307
308         var physicData = GetPhysicData(items);
309         var textureData = GetTextureData(items, physicData, backgroundImage, foregroundImage);
310         var animationData = GetAnimationData(items);
311         var keyboardData = GetKeyboardData(items, keyboardMappings);
312         var tagData = GetTagData(items, tags);
313
314         for (int i=0;i<items.Count;i++)
315         {
316             returnList.Add(new PhysikLevelItemExportData()
317             {
318                 LevelItemId = items[i].LevelItemId,
319                 PhysicSceneData = physicData[i],
320                 TextureData = textureData[i],
321                 AnimationData = animationData[i],
322                 KeyboardMappings = keyboardData[i],
323                 TagdataEntries = tagData[i]
324             });
325         }
326
327     }
328 }

```

Siehe: Engine/05_Leveleditor/03_LevelToSimulator/PhysicItemConverter.cs

Dieses PhysikLevelItemExportData-Array wird zusammen mit den BackgroundItems in ein SimulatorInputData-Objekt konvertiert. Siehe Bild:

```

6  public static class SimulatorExporter
7  {
8      2 Verweise
9      public static SimulatorInputData Convert(EditorDataForSimulation data)
10     {
11         var physicLevelItems = PhysicItemConverter.Convert(data.Items, data.BackgroundImage, data.ForegroundImage, data.KeyboardMa|
12         return new SimulatorInputData()
13     }
14     PhysicLevelItems = physicLevelItems,
15     CollisionMatrix = data.CollisionMatrix,
16     HasGravity = data.HasGravity,
17     Gravity = data.Gravity,
18     IterationCount = data.IterationCount,
19     CameraTrackedLevelItemId = data.CameraTrackedLevelItemId,
20     CameraTrackerData = data.CameraTrackerData,
21     BackgroundImage = data.BackgroundImage,
22     BackgroundItems = data.BackgroundItems,
23     };
24 }

```

Siehe: Engine/05_Leveleditor/03_LevelToSimulator/SimulatorExporter.cs

Der EditorFileConverter wandelt eine LevelEditor-Datei in ein SimulatorInputData-Objekt um:

```

10    2 Verweise
11    public static class EditorFileConverter
12    {
13        2 Verweise
14        public static SimulatorInputData Convert(string editorFileName)
15        {
16            var data = JsonHelper.Helper.CreateFromJson<LevelEditorExportData>(FileNameReplacer.LoadEditorFile(editorFileName));
17
18            var panel = new GraphicPanel2D() { Width = 100, Height = 100, Mode = Mode2D.CPU };
19            var editor = new LevelEditorViewModel(new EditorInputData() { Panel = panel });
20            editor.LoadFromExportObject(data);
21            return editor.GetSimulatorExport();
22        }
23    }

```

Siehe: Engine/05_Leveleditor/05_Editor/LevelEditorControl/EditorFileConverter.cs

Er wird vom GameSimulator im Konstruktor benutzt:

```

26    public GameSimulator(string levelFile, Size panelSize, float timerIntercallInMilliseconds)
27        :base(EditorFileConverter.Convert(levelFile), panelSize, timerIntercallInMilliseconds)
28    {
29        this.tagStorage = CreateTagStorage(this.levelItems, inputData.PsicLevelItems);
30    }

```

Siehe: Engine/08_GameHelper/GameHelper/Simulation/GameSimulator.cs

welcher den Simulator als Basisklasse hat, welcher den ExportToRuntimeConverter benutzt, um die Export-Daten von den PhysicLevelItems in eine PhysikScene umwandelt.

```

48    public Simulator(SimulatorInputData data, Size panelSize, Camera2D camera, float timerIntercallInMilliseconds)
49    {
50        this.inputData = data;
51
52        ExportToRuntimeConverter.Convert(data.PsicLevelItems, data.CollisionMatrix, out this.physicScene, out RuntimeLevelItem[] startLevelItems);
53        this.levelItems.AddRange(startLevelItems);

```

Siehe: Engine/05_Leveleditor/04_Simulator/Simulator.cs

Dabei haben die RuntimeLevelItems die Aufgabe all die IPublic-Objekte zusammen zu fassen, welche zum gleichen LevelItem gehören.

```

7    internal static class ExportToRuntimeConverter
8    {
9        1 Verweis
10       internal static void Convert(PhysicLevelItemExportData[] exportItems, bool[,] collisionMatrix, out PhysicScene physicScene, out RuntimeLevelItem[] runtimeItems)
11       {
12           List<RuntimeLevelItem> returnList = new List<RuntimeLevelItem>();
13
14           physicScene = new PhysicScene(new RigidBodyPhysics.ExportData.PsicSceneExportData()
15           {
16               CollisionMatrix = collisionMatrix,
17           });
18
18           foreach (var item in exportItems)
19           {
20               var physicData = physicScene.AddPhysicScene(item.PsicSceneData);
21
22               returnList.Add(new RuntimeLevelItem(item.LevelItemId, physicData));
23           }
24
25           runtimeItems = returnList.ToArray();
26       }
27   }

```

Siehe: Engine/05_Leveleditor/04_Simulator/Simulator/ExportToRuntimeConverter.cs

Es gibt also ein PhysicScene-Objekt, was eine Menge von IPublic-Objekten hat. Die RuntimLevelItem-Objekte verweisen dann jeweils auf eine Teilmenge von dieser IPublic-Objekteliste.

Die Idee ist nun, das jedem IPublic-Nutzer die IPublic-Objekte zusammen mit sein ExportDaten rein gegeben werden und dass über das IPublic/RuntimLevelItem-Objekt hat man ein eindeutigen Schlüssel, worüber man an die ExportDaten erneut ran kommt oder wie man das Objekt aus dem IPublic-Verwender entfernt.

Konkret sieht das beim PhysicSceneDrawer so aus: Im Konstruktor wird die PhysicScene und ein VisualizerOutputData-Objekt rein gegeben. Die PhysicScene enthält N IPublicRigidBody-Objekte und das VisualisizerOutputData-Objekt enthält auch N TextureExportData-Objekte.

```

14     public PhysicSceneDrawer(PhysicScene physicScene, VisualizerOutputData textureData)
15     {
16         textures = ConvertPhysicScene(physicScene, textureData).ToList();
17         distanceJoints = physicScene.GetAllJoints().Where(x => x is IPublicDistanceJoint).Cast<IPublicDistanceJoint>().ToList();
18
19         physicScene.BodyWasDeletedHandler += RigidBodyWasDeleted;
20         physicScene.JointWasDeletedHandler += JointWasDeleted;
21     }

```

Siehe: Engine/04_AnimationAndTexture/TextureEditor/PhysicSceneDrawing/PhysicSceneDrawer.cs

Jedes IPublicRigidBody-TextureExportData-Paar wird genutzt um ein ITextureRigidBody-Objekt zu erzeugen:

```

48     private static ITexturedRigidBody[] ConvertPhysicScene(PhysicScene physicScene, VisualizerOutputData textureData)
49     {
50         var bodys = physicScene.GetAllBodys().ToList();
51         if (bodys.Count != textureData.Textures.Length) throw new ArgumentException("Body.Count != Textures.Count");
52
53         List<ITexturedRigidBody> textures = new List<ITexturedRigidBody>();
54         for (int i = 0; i < bodys.Count; i++)
55         {
56             textures.Add(Convert(bodys[i], textureData.Textures[i]));
57         }
58
59         return textures.ToArray();
60     }

```

Siehe: Engine/04_AnimationAndTexture/TextureEditor/PhysicSceneDrawing/PhysicSceneDrawer.cs

Oder man kann auch im Nachhinein dann einzelne Körper hinzufügen:

```

139     public void AddBody(IPublicRigidBody body, TextureExportData texturData)
140     {
141         this.textures.Add(Convert(body, texturData));
142     }

```

Über das IPublicRigidBody-Objekt kann ich dann die ExportDaten abfragen:

```

151     public TextureExportData GetTextureDataFromBody(IPublicRigidBody body)
152     {
153         var tex = this.textures.First(x => x.AssociatedBody == body);
154         return tex.TextureExportData;
155     }

```

Wenn in der PhysicScene jemand ein RigidBody löscht, dann löst dass das BodyWasDeleted-Event aus, was vom PhysicSceneDrawer benutzt wird, damit er bei sich das ITextureRigidBody-Objekt löscht.

```

19     |     physicScene.BodyWasDeletedHandler += RigidBodyWasDeleted;

23     |     private void RigidBodyWasDeleted(PhysicScene physicScene, IPublicRigidBody body)
24     {
25         var deleteList = textures.Where(x => x.AssociatedBody == body).ToList();
26         foreach (var del in deleteList)
27         {
28             textures.Remove(del);
29         }
30     }

```

Das IPublicRigidBody-Objekt ist also der Schlüssel, um beim PhysicSceneDrawer Objekte hinzuzufügen, vorhandene Daten abzufragen oder ein Objekt zu löschen.

Bei den ganzen anderen IPublic-Objekt-Verwendern hat man nicht nur ein einzelnen Körper oder

Gelenk sondern eine ganze Liste von Objekten. All diese Klassen nutzen deswegen als Schlüssel ein PhysicScenePublicData-Objekt, welches erzeugt wird, wenn man ein PhysicSceneExportData-Objekt der PhysicScene hinzufügt:

```

113     public PhysicScenePublicData AddPhysicScene(PhysicSceneExportData data)
114     {
115         var scene = new PhysicScene(data);
116         foreach (var body in scene.bodies)
117         {
118             AddRigidBody(body);
119         }
120         this.joints.AddRange(scene.joints);
121         this.thrusters.AddRange(scene.thrusters);
122         this.rotaryMotors.AddRange(scene.rotaryMotors);
123         SetVelocityFromFixBodiesToZero();
124
125         return new PhysicScenePublicData(scene.GetAllBodys(), scene.GetAllJoints(), scene.GetAllThrusters(), scene.GetAllRotaryMotors());
126     }

```

Der LevelItemAnimator und LevelItemKeyboardHandler nutzt das PhysicScenePublicData-Objekt als Schlüssel und sie bekommen jeweils die ExportDaten als Input um somit bei sich intern Objekte anzulegen, welche mehrere RigidBodys/Joints verwenden:

```

65     this.animator = new LevelItemAnimator(timerIntercallInMilliseconds);
66     this.keyHandler = new KeyboardControl.LevelItemKeyboardHandler();
67     for (int i=0;i<startLevelItems.Length;i++)
68     {
69         Animator[] animators = null;
70
71         var animationData = data.PhysicLevelItems[i].AnimationData;
72         if (animationData != null)
73         {
74             this.animator.AddLevelItem(startLevelItems[i], data.PhysicLevelItems[i].AnimationData);
75             animators = this.animator.GetAnimationRuntimeDataFromLevelItem(startLevelItems[i]);
76         }
77
78         this.keyHandler.AddLevelItem(startLevelItems[i], animators, data.PhysicLevelItems[i].KeyboardMappings);
79     }

```

Siehe: Engine/05_Leveleditor/04_Simulator/Simulator/Animation/LevelItemAnimator.cs

Auch diese Klassen können über diesen Schlüssel dann ihre ExportDaten nach außen geben oder man kann damit dann auch ihre internen Objekte löschen:

```

29     public AnimationOutputData[] GetAnimationExportDataFromLevelItem(PhysicScenePublicData physicObjects)
30     {
31         return this.levelItems
32             .First(x => x.PhysicObjects == physicObjects).AnimationData;
33     }
34
35     1 Verweis
36     public void RemoveLevelItem(PhysicScenePublicData physicObjects)
37     {
38         var del = this.levelItems.First(x => x.PhysicObjects == physicObjects);
39         this.levelItems.Remove(del);
40     }
41
42     public KeyboardMappingEntry[] GetExportDataFromLevelItem(PhysicScenePublicData physicObjects)
43     {
44         return this.levelItems.First(x => x.PhysicObjects == physicObjects).KeyboardData;
45     }
46
47     1 Verweis
48     public void RemoveLevelItem(PhysicScenePublicData physicObjects)
49     {
50         var del = this.levelItems.First(x => x.PhysicObjects == physicObjects);
51         this.levelItems.Remove(del);
52     }

```

Siehe: Engine/05_Leveleditor/04_Simulator/Simulator/

Das gleiche gilt auch für den SimulatorTagDataStorage. Der Schlüssel ist wieder ein PhysicScenePublicData-Objekt und es gibt wieder die Add,GetExport,Remove-Funktionen:

```

16     public void AddLevelItem(PhysicScenePublicData levelItem, PhysicSceneTagdataEntry[] bodyTagdataEntries)...
44
45     1 Verweis
46     public PhysicSceneTagdataEntry[] GetExportDataFromLevelItem(int levelItemId)
47     {
48         return this.entries.Where(x => x.LevelItemId == levelItemId).ToArray();
49     }
50
51     1 Verweis
52     public void RemoveLevelItem(int levelItemId)
53     {
54         var delList = this.entries.Where(x => x.LevelItemId == levelItemId).ToList();
55         foreach (var item in delList)
56         {
57             this.entries.Remove(item);
58         }
59     }

```

Siehe: Engine/08_GameHelper/GameHelper/Simulation/RigidBodyTagging/SimulatorTagStorage.cs

Somit ist also die Grundidee für der Verwaltung von LevelItems im Simulator:

Der Leveleditor/Konverter stellt ExportData-Objekte bereit.

Diese Export-Objekte werden der PhysicScene gegeben worauf hin sie IPublic/PhysicScenePublicData-Objekte erzeugt.

Diese Objekte fassen all die Einzelobjekte von ein LevelItem zusammen und sind der Schlüssel für den PhysicSceneDrawer, LevelItemAnimator, LevelItemKeyboardHandler und SimulatorTagStorage (IPublic-Nutzer)

Alle IPublic-Nutzer haben jeweils eine Add,GetExport,Remove-Funktion.

```

17     public class Simulator
18     {
19         protected SimulatorInputData inputData;
20         protected PhysicScene physicScene;
21         protected List<RuntimeLevelItem> levelItems = new List<RuntimeLevelItem>(); //Verweist auf IPublic-Objekte aus der PhysicScene
22
23         protected PhysicSceneDrawer sceneDrawer;
24         private SmallWindow smallWindow;
25
26         protected LevelItemAnimator animator;
27         protected KeyboardControl.LevelItemKeyboardHandler keyHandler;
28
29     public class GameSimulator : Simulator, ITagDataProvider
30     {
31         private SimulatorTagStorage tagStorage;
32     }
33
34

```

Will ich nun von ein LevelItem alle ExportDaten haben, dann ermittle ich zuerst über die LevelItemId das Schlüsselobjekte und rufe damit dann von jeden IPublic-Nutzer seine ExportDaten ab. Ich kann dann eine Kopie von den ExportDaten erzeugen und diese Kopie auch bearbeiten (andere LevelItem-Position/Skalierung/Rotation) und dieses Kopie füge ich dann beim GameSimulator wieder ein, indem zuerst ein neues Schlüsselobjekt erzeugt wird und dann werden alle IPublic-Nutzer mit Daten befüllt.

```

288     public PhysikLevelItemExportData GetExportDataFromLevelItem(int levelItemId)
289     {
290         var item = this.levelItems.First(x => x.LevelItemId == levelItemId);
291
292         return new PhysikLevelItemExportData()
293         {
294             LevelItemId = levelItemId,
295             PhysicSceneData = this.physicScene.GetExportData(item),
296             TextureData = new VisualizerOutputData(item.Bodies.Select(x => new TextureExportData(this.sceneDrawer.GetTextureDataFrom
297             AnimationData = this.animator.GetAnimationExportDataFromLevelItem(item).Select(x => new AnimationOutputData(x)).ToArray(),
298             KeyboardMappings = this.keyHandler.GetExportDataFromLevelItem(item).Select(x => new KeyboardMappingEntry(x)).ToArray(), ///
299             TagdataEntries = this.tagStorage.GetExportDataFromLevelItem(levelItemId).Select(x => new PhysicSceneTagdataEntry(x)).ToA
300         };
301     }
302
303     3 Verweise | 0 1/1 bestanden
304     public int AddLevelItem(PhysikLevelItemExportData data)
305     {
306         int newId = this.levelItems.Any() ? this.levelItems.Max(x => x.LevelItemId) + 1 : 1;
307
308         var physicObjects = new RuntimeLevelItem(newId, physicScene.AddPhysicScene(data.PhysicSceneData));
309
310         for (int i=0;i<physicObjects.Bodies.Length;i++)
311         {
312             this.sceneDrawer.AddBody(physicObjects.Bodies[i], data.TextureData.Textures[i]);
313         }
314
315         Animator[] animators = null;
316         if (data.AnimationData != null)
317         {
318             this.animator.AddLevelItem(physicObjects, data.AnimationData);
319             animators = this.animator.GetAnimationRuntimeDataFromLevelItem(physicObjects);
320         }
321
322         this.keyHandler.AddLevelItem(physicObjects, animators, data.KeyboardMappings);
323
324         this.tagStorage.AddLevelItem(physicObjects, data.TagdataEntries.Select(x => new PhysicSceneTagdataEntry(x, newId)).ToArray());
325
326         levelItems.Add(physicObjects);
327
328         return newId;
328     }

```

Siehe: Engine/08_GameHelper/GameHelper/Simulation/GamSimulator.cs

So kann nun ein Objekt dupliziert und gelöscht werden. Zuerst wird im LevelEditor ein LevelItem erzeugt, was per Tag den Name „Original“ bekommt. Ich ermittle von diesen Objekt die LevelItemId und lass mir die ExportDaten dann geben und lösche es aus dem Simulator.

```

58     var body = simulator.GetBodiesByTagName("Original").First();
59     int levelItemId = simulator.GetTagDataFromBody(body).LevelItemId;
60     var exportData = simulator.GetExportDataFromLevelItem(levelItemId); //Kopie nach außen geben
61     var box = PhysicSceneExportDataHelper.GetBoundingBoxFromScene(exportData.PhysicSceneData);
62
63     simulator.RemoveLevelItem(levelItemId); //Original löschen

```

Siehe: Engine/07_DynamicObjCreation/DynamicObjCreation.UnitTests/InsertLevelItemTest.cs

Diese Kopie bearbeite ich dann mit MoveToPivotPoint und füge es wieder ein. Da es ja möglich ist, dass ich das Objekt mehrmals mit gleichen TagName einfüge, ist es nötig, dass ich bei den GetBody/Joint/..ByTagName zusätzlich noch die LevelItemId angebe, um auf die Körper und Gelenke vom jeweiligen LevelItem zuzugreifen.

```

78     var copyData = new LevelEditorGlobal.PhysikLevelItemExportData(exportData); //Kopie vom Original-Export erstellen
79     var pivotPoint = new Vector2D(box.Width + i * box.Width * 2, box.Height);
80     LevelItemExportHelper.MoveToPivotPoint(copyData, pivotPoint, allOriantations[i], 1, 0); //Kopie bearbeiten
81     int newId = simulator.AddLevelItem(copyData);
82     pivotPoints.Add(pivotPoint);
83
84     //Hiermit teste ich, dass ich bei eingefügten Objekten auf die Tagdaten zugreifen kann
85     tagPoints.Add(simulator.GetBodyByTagName(newId, "body1").Center.ToGrx());
86     tagPoints.Add(simulator.GetJointByTagName(newId, "joint1").Anchor1.ToGrx());
87     tagPoints.Add(simulator.GetThrusterByTagName(newId, "thruster1").Anchor.ToGrx());
88     tagPoints.Add(simulator.GetMotorByTagName(newId, "motor1").Body.Center.ToGrx());

```

Teil 10: Example-Games

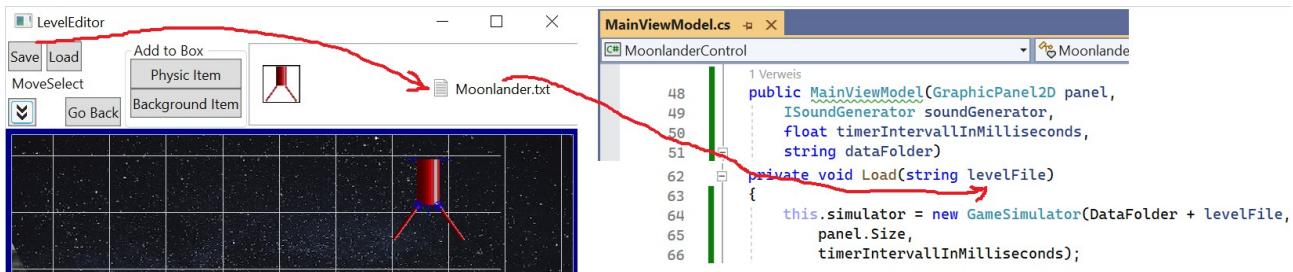
Ziel dieses Abschnitts

Hier soll anhand von ein paar kleinen Spielen gezeigt werden, wie die Physicengine zu verwenden ist.

Game 1 – Moonlander

Verwendung der GameSimulator-Klasse

Mit dem Leveleditor wurde die Moonlander.txt-Datei erzeugt und diese Datei wird im Konstruktor vom GameSimulator rein gegeben.



Siehe: Data/GameData/Moonlander/Moonlander.txt

Siehe: DemoApplications/Moonlander/MoonlanderControl/Controls/MainViewModel.cs

Der GameSimulator legt intern bei sich kein GraphicPanel2D und auch kein Timer an. Stattdessen hat der Simulator eine Draw- und MoveOnStep-Methode, um damit auf das Timer-Event zu reagieren und die Objekte zu zeichnen/bewegen. Der Grund für diese Designentscheidung ist, da man in der Anwendung nur ein GraphicPanel2D- und Timer-Objekt haben soll und ich dem Verwender der GameSimulator die Möglichkeit geben will, dass er diese Objekte auch an anderer Stelle verwendet.

Deswegen muss das MainViewModel von den jeweiligen Spiel das ITimerHandler- und IGraphicPanelHandler-Interface implementieren um die Events vom GraphicPanel2D- und Timer-Objekt zu behandeln.



Das MainWindowViewModel nutzt die MoonlanderControlFactory, um damit das Moonlander-MainControl und MainViewModel zu erzeugen. Diese Klasse ist für die Erstellung des GraphicPanel2D und des Timers zuständig und nutzt dann die Timer- und GraphicPanel2D-Events, um diese vom MainViewModel des aktiven Spiels verarbeiten zu lassen.

```

MainWindowViewModel.cs  x
[PhysicEngine] [PhysicEngine.MainWindowViewModel] [Panel_MouseMove(object? sender, MouseEventArgs e)]
140    {
141        try
142        {
143            if (this.ContentUserControl?.DataContext is ITimerHandler)
144                (this.ContentUserControl.DataContext as ITimerHandler).HandleTimerTick((float)timer.Interval);
145            }catch (Exception ex)
146            {
147                this.timer.Stop();
148                MessageBox.Show(ex.ToString());
149            }
150        }
151    }
152}
1 Verweis
153    private void Panel_SizeChanged(object? sender, EventArgs e)
154    {
155        if (this.ContentUserControl?.DataContext is ISizeChangeable)
156            (this.ContentUserControl.DataContext as ISizeChangeable).HandleSizeChanged(panel.Width, panel.Height);
157    }
1 Verweis
158    private void Panel_MouseClick(object? sender, System.Windows.Forms.MouseEventArgs e)
159    {
160        if (this.ContentUserControl?.DataContext is IGraphicPanelHandler)
161            (this.ContentUserControl.DataContext as IGraphicPanelHandler).HandleMouseClick(e);
162    }

```

Siehe: PhysicEngine/MainWindowViewModel.cs

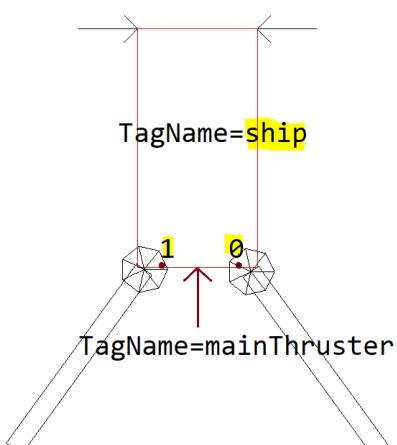
Schubdüse mit Feuerpartikeln

Der Hauptteil vom Raumschiff ist ein Rechteck wo unten die Hauptdüse dran hängt. Immer dann, wenn diese Düse aktiv ist, dann sollen Feuerpartikel erzeugt werden. Dazu werden im Leveleditor im Rechteck (rot mit TagName ship) zwei Ankerpunkte definiert, welche die untere Kante vom Raumschiff darstellen.

Zeile 34-36: Im Konstruktor von der MainThruster-Klasse lege ich die beiden AnchorPoint-Objekte an und auch ein IPublicThruster-Objekt, welches mir Auskunft darüber gibt, ob die Schubdüse gerade aktiv ist oder nicht.

Zeile 72-74: In der MoveOnStep-Methode frage ich, ob die Schubdüse gerade aktiv ist. Wenn ja, erzeuge ich neue Feuerpartikel und füge sie dem ParticleHandler hinzu.

Zeile 84-85: Ich ermittle die aktuelle Position der Ankerpunkte und erzeuge dann auf der Linie zwischen den Punkten neue Feuerpartikel.



```

MainThruster.cs  x
[MoonlanderControl] [MainThruster] [MoonlanderControl.Mode]
3 Verweise
class MainThruster
{
    private AnchorPoint anchorPoint1, anchorPoint2;
    private IPublicThruster mainThruster;
    ...
    public MainThruster(GameSimulator simulator, Sounds sounds)
    {
        this.sounds = sounds;

        this.anchorPoint1 = new AnchorPoint(simulator, "ship", 0);
        this.anchorPoint2 = new AnchorPoint(simulator, "ship", 1);
        this.mainThruster = simulator.GetThrusterByTagName("mainThruster");
        this.ship = simulator.GetBodyByTagName("ship");

        simulator.BodyWasDeletedHandler += Simulator_BodyWasDeletedHandler;
        this.mainThruster.IsEnabledChanged += MainThruster_IsEnabledChanged;
    }

    public void MoveOnStep(float dt)
    {
        if (this.mainThruster.IsEnabled && this.shipIsRemovedFromSimulation == false)
        {
            this.particleHandler.AddParticles(CreateNewParticles());
        }

        this.particleHandler.HandleTimerTick(dt);
    }

    private List<Particle> CreateNewParticles()
    {
        List<Particle> particles = new List<Particle>();

        var p1 = this.anchorPoint1.GetPosition();
        var p2 = this.anchorPoint2.GetPosition();

        var direction = (p2 - p1).Spin90().Normalize();
    }
}

```

Die Erzeugung von neuen Feuerpartikel erfolgt in jeden Time-Step. Deswegen arbeite ich dort mit dem Polling-Ansatz, um die IsEnabled-Property des IPublicThruster-Objektes abzufragen.

Die Sound-Wiedergabe verlangt aber, dass man einen Sound per Start-Befehl loslaufen lässt. Dann läuft der Sound so lange, bis Stop aufgerufen wird.

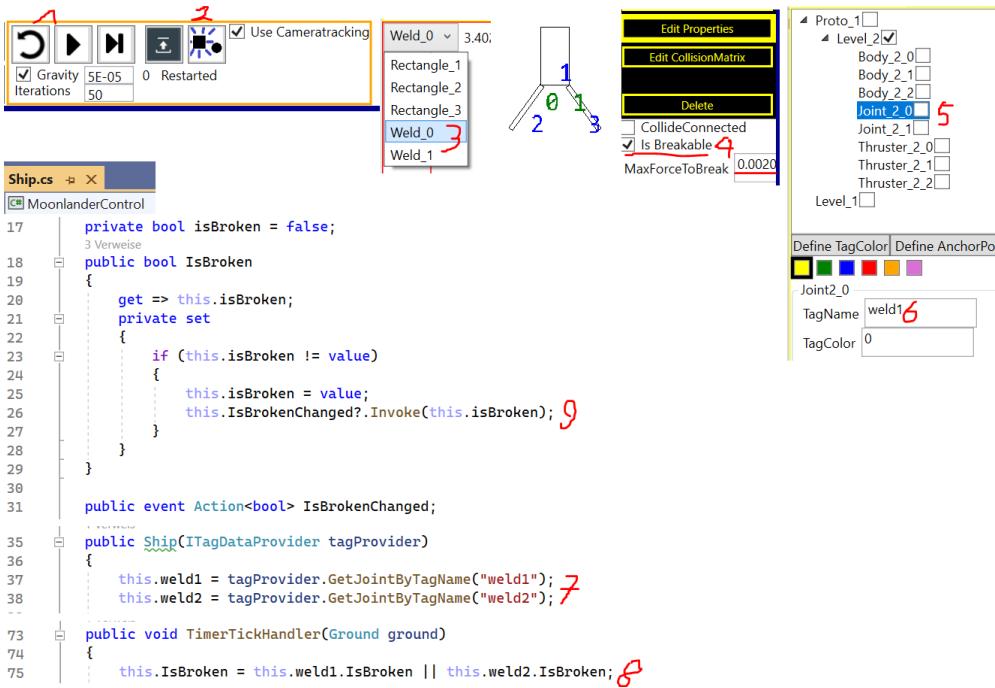
Deswegen wird hier der Eventbasierte Ansatz genommen, um die IsEnabled-Property zu überwachen.

```
40 } this.mainThruster.IsEnabledChanged += MainThruster_IsEnabledChanged;
41
42
43 1 Verweis
44 private void MainThruster_IsEnabledChanged(bool isEnabled)
45 {
46     if (isEnabled)
47         this.sounds.StartMainThrusterSound();
48     else
49         this.sounds.StopMainThrusterSound();
```

Siehe: DemoApplications/Moonlander/MoonlanderControl/Model/MainThruster.cs

Raumschiff zerstören, wenn es zu hart aufsetzt

Das Raumschiff hat zwei Beine, die über ein Weld-Joint am Schiffskörper befestigt sind. Wenn die Kräfte auf diese Welt-Joints zu groß werden, dann sollen sie abbrechen und das ganze Schiff soll explodieren. Um diese Aufgabe zu realisieren, muss zuerst der Forcetracker im Leveleditor aktiviert werden, indem beim Simulator Restart (1) gedrückt wird und dann der Forcetracker (2) aktiviert wird. Dann simuliert man das Aufsetzen des Raumschiffs und schaut sich dann die Kräfte an, welche auf die Weld-Joints gewirkt haben (3). Dann setzt man die IsBreakable-Property (4) und verwendet die Kraft aus dem Forcetracker, welche angibt, ab wann das Gelenk bricht. Dann selektiert man im LevelItem-Tree die Weld-Joints (5) und vergibt ein TagName (6), welchen man in der Ship-Klasse dann verwendet (7), um Zugriff auf das IPublicJoint-Objekt zu bekommen. Im TimerTick-Handler wird dann die IsBroken-Property gepollt (8) und wenn ein Gelenk von beiden kaputt ist, feuert das Ship-Objekt das IsBrokenChanged-Event (9).



Der Eventhandler vom IsBrokenChanged-Event ist die Ship_IsBrokenChanged-Methode (10), welche den ShipExploder nutzt, um mit dessen StartTimer-Methode (11) das Rechteck vom Raumschiff nach 3 Sekunden zu zerstören. Dazu nutzt der ShipExploder die TickCounterStopwatch, wo man ihr eine Action (12) als Inputparameter gibt, welche dann nach Ablauf der Zeit aufgerufen wird. In dieser Action wird dann zufällig eine Zerstörungsmethode gewählt (13).



Wenn das Schiff in Partikel zerlegt werden soll, dann wird zuerst der Schiffskörper (Rechteck) aus der Simulation entfernt (14) und an der Position von diesen Rechteck werden dann 100 Partikel erzeugt (15) oder es wird die DestroyRigidBody-Methode vom Simulator genutzt, um das Rechteck zu zerlegen (16).

```

105     private void DestroyWithParticles(Random rand)
106     {
107         this.simulator.RemoveRigidBody(this.ship); 14
108         this.particleHandler.AddParticles(ParticleHandler.CreateParticles((IPublicRigidRectangle)this.ship, 100, 1, Color.Red, Color.Yellow, rand));
109     }
110
111     1 Verweis
112     private void DestroyWithBoxes(Random rand)
113     {
114         this.simulator.DestroyRigidBody(this.ship, new DestroyWithBoxesParameter()
115         {
116             BoxCount = 5,
117             TransformFunc = (body, polyObj) =>
118             {
119                 polyObj.Velocity += GetRandomVec(rand, 0.02f);
120                 polyObj.AngularVelocity += (float)rand.NextDouble() * 0.1f;
121                 return polyObj;
122             };
123         });
124     }

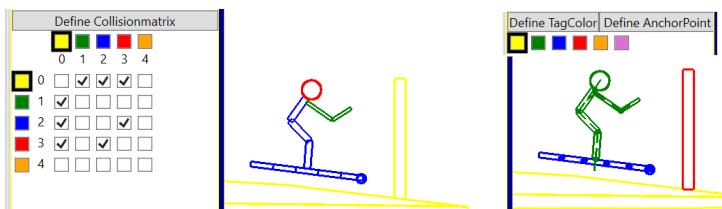
```

Game 2 – Skijumper

Prüfen, dass der Skispringer verletzungsfrei die Fahne erreicht hat

Mit der Collisionmatrix (linkes Bild) lege ich fest, welcher Körper mit welchen kollidiert. Das wird benötigt, damit die Arme am Körper vorbei schwingen können (grün kollidiert nicht mit blau und rot). Damit der Fahrer beim Hinhocken nicht mit sein Oberkörper am Knie oder Ski hängen bleibt kollidiert blau nicht mit blau. Der Kopf soll aber schon mit dem Skibratt/Körper kollidieren so dass er eine eigene Collisionkategorie bekommt.

All diese Collisionmatrix-Einstellungen haben aber nichts mit der Spiellogik zu tun, welche auswertet, was passiert, wenn ein bestimmter Körper mit ein anderen Körper kollidiert. Für diese Aufgabe ist die TagColor (rechts im Bild) gedacht.



Im CollisionOccured-Event vom Simulator wird für jeden Kollisionspunkt von den beiden Körpern, die gerade kollidieren die TagColor (Zeile 46/47) ermittelt und abhängig davon passiert dann eine bestimmte Spiellogik.

```

Skiman.cs
SkiljumperControl
22     public Skiman(GameSimulator simulator, Sounds sounds)
35     {
36         simulator.CollisionOccurred += Simulator_CollisionOccurred;
37     }
38
39     private void Simulator_CollisionOccurred(PhysicsScene sender, PhysicsBodyCollision[] collisions)
40     {
41         bool manIsTouchingTheGround = false;
42         bool callFlagIsTouched = false;
43         bool boardIsTouchingTheGround = false;
44
45         foreach (var collision in collisions)
46         {
47             byte color1 = this.tagDataProvider.GetTagDataFromBody(collision.Body1).Color;
48             byte color2 = this.tagDataProvider.GetTagDataFromBody(collision.Body2).Color;
49
50             //Tag-Color-Werte:
51             //0 = Boden
52             //1 = Skifahrer ohne Brett
53             //2 = Skibratt
54             //3 = Flagge
55
56             //Skifahrer berührt Fahne
57             if ((color1 == 1 || color1 == 2) && color2 == 3)
58             {
59                 callFlagIsTouched = true;
60             }
61
62             //Skifahrer berührt mit sein Körper den Boden
63             if (color1 == 1 && color2 == 0)
64             {
65                 manIsTouchingTheGround = true;
66             }
67
68             //Skibratt berührt den Boden
69             if (color1 == 2 && color2 == 0)
70             {
71                 boardIsTouchingTheGround = true;
72             }
73         }
74     }

```

Diese Art der Kollisionsauswertung funktioniert nur, wenn Objekte nicht dynamisch während der Simulation erzeugt werden sondern alle Objekte so direkt aus dem LevelEditor kommen. Nur so ändert sich die Reihenfolge der Körper in der Simulation nicht und nur so weiß man, dass z.B. bei ein Skibrett-Boden-Kollisionspunkt Body1 immer das Skibrett ist und Body2 der Boden. So muss ich also auf Zeile 68 nicht auch noch die Bedingung color1==0 && color2==2 abprüfen.

Wenn man bei der Kollision von dynamisch (während der Simulationszeit) erzeugten Objekten eine Spiellogik hinterlegen will, so sollte man sich den Kollisionshandler vom Astroid-Spiel ansehen.

Game 3 – Elma

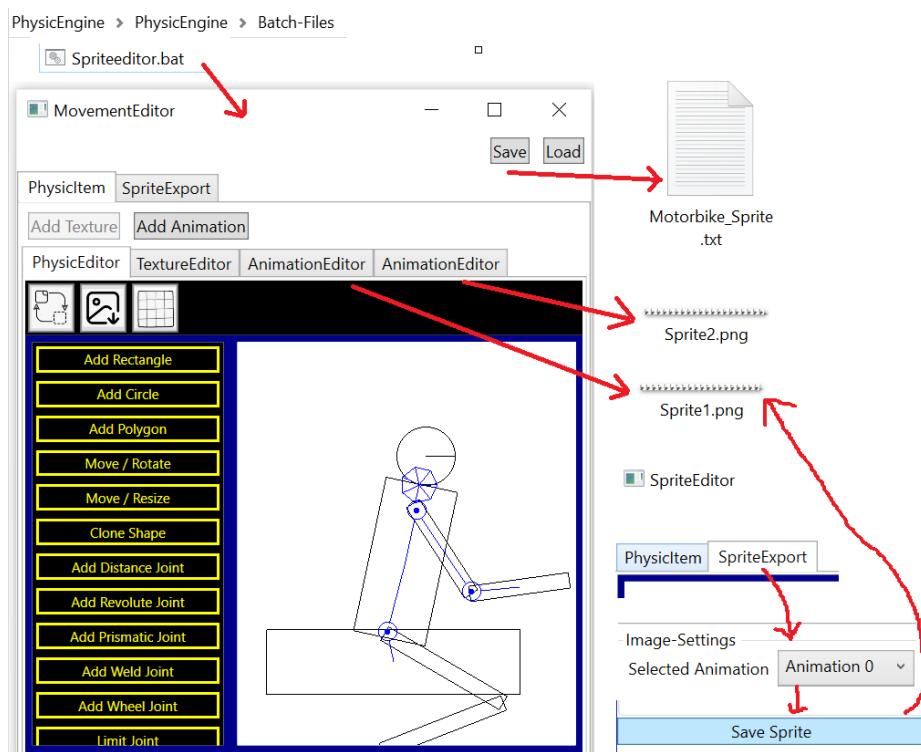
Sprites über externen Editor erzeugen

Das Motorrad bei Elma besteht aus einer Spritedatei. Es gibt eine Sprite-Animation wenn er die Arme hochhebt und eine, wenn er die Arme nach unten bewegt. Die erste Möglichkeit, wie man diese beiden Spritedateien erzeugen ist man nutzt den Spriteeditor unter dem Batch-Files-Ordner. Mit diesen Editor erzeugt man dann drei Dateien:

Motorbike_Sprite.txt = Enthält die Information, aus wie viel Einzelbildern die beiden Spriteanimationen jeweils bestehen und mit welcher Geschwindigkeit sie laufen sollen

Sprite1.png = Animation, wo der Fahrer die Arme hochhebt

Sprite2.png = Animation, wo der Fahrer die Arme runter nimmt



Siehe: Data/GameData/Elma/Motorbike_Sprite.txt

Sprite aus dem Spiel heraus erzeugen

Die zweite Möglichkeit, wie die drei Dateien (Motorbike_Sprite.txt, Sprite1.png, Sprite2.png) erzeugt werden können ist indem man den Spriteeditor ins Spiel einbindet. Das Wpf-UserControl des Spriteeditors kann über die SpriteEditorFactory erzeugt werden. Das Viewmodel von diesen Control hat eine WriteToFile-Methode, womit die Motorbike_Sprite.txt-Datei erzeugt werden

kann und mit der CreateFromMovementEditorFile-Funktion werden dann die Sprite1.png(Sprite2.png)-Dateien erzeugt.

```
16 internal class MainViewModel : ReactiveObject, ITimerHandler, IGraphicPanelHandler, IPhysicSimulated
17 {
18     private SubControlFactory factory;
19
20     private UserControl mainSelect, levelSelect, singleLevel, gameControl, levelEditor, spriteEditor;
21
22     this.spriteFile = data.DataFolder + "Motorbike_Sprite.txt";
23
24     this.spriteEditor = this.factory.CreateControl(ControlType.SpriteEditor);
25
26     case ControlType.SpriteEditor:
27     {
28         return new MovementEditorFactory().CreateEditorControl(new EditorInputData(data) { ShowSaveLoadButtons = false});
29     }
30
31     //GoBack-Handler vom Level- und SpriteEditor
32     data.IsFinished = (sender) =>
33     {
34         if (this.SelectedControl == ControlType.SpriteEditor)
35         {
36             //Aktualisiere die Sprite-Image-Dateien wenn vorher der SpriteEditor genutzt wurde
37             (this.spriteEditor.DataContext as IToTextWriteable).WriteToFile(this.spriteFile);
38             UpdateSpriteImages();
39         }
40
41     private void UpdateSpriteImages()
42     {
43         var spriteData1 = MovementEditorControl.SpriteDataCreator.CreateFromMovementEditorFile(this.spriteFile, 0);
44         var spriteData2 = MovementEditorControl.SpriteDataCreator.CreateFromMovementEditorFile(this.spriteFile, 1);
45
46         BitmapHelper.BitmapHelp.ScaleImageDownWithoutColorInterpolation(spriteData1.Image, Bike.ScaleFactor).Save(this.dataFolder + "Sprite1.png");
47         BitmapHelper.BitmapHelp.ScaleImageDownWithoutColorInterpolation(spriteData2.Image, Bike.ScaleFactor).Save(this.dataFolder + "Sprite2.png");
48     }
```

Siehe: DemoApplications/Elma/ElmaControl/Controls/Main/MainViewModel.cs

Sprite anzeigen

Zuerst müssen die beiden Sprite-png-Dateien eingelesen werden (Zeile 66-68). Dann muss dem Simulator gesagt werden, dass er für das Motorrad nicht die Standard-Draw-Methode nutzen soll sondern man übergibt mit der UseCustomDrawingForRigidBody-Methode ein IRigidBodyDrawer-Objekt, was eine Draw-Methode hat (Zeile 81). Diese zeichnet dann die Sprite-Animation. Das kann entweder ein einzelnes Bild aus der Animation sein (DrawSingleImage Zeile 369) oder die Animation, wo der Fahrer die Arme bewegt (Zeile 374).

```

Bike.cs ✘ ×
ElmaControl

18 internal class Bike : ITimerHandler, IRigidBodyDrawer, IDrawable
66     var spriteData = JsonHelper.Helper.CreateFromJson<MovementEditorExportData>(File.ReadAllText(dataFolder + "Motorbike_Sprite.txt"));
67     this.armsUpSprite = new SpriteImage(dataFolder + "Sprite1.png", spriteData.SpriteCount, 1, spriteData.SpriteCount,
68                                         spriteData.SpriteCount);
69     this.armsDownSprite = new SpriteImage(dataFolder + "Sprite2.png", spriteData.SpriteCount, 1, spriteData.SpriteCount);
70
71     simulator.UseCustomDrawingForRigidBody(this.head, this);
72     simulator.RemoveBodyFromPhysicsSceneDrawer(this.wheel1);
73     simulator.RemoveBodyFromPhysicsSceneDrawer(this.wheel2);
74
75     public void Draw(GraphicPanel2D panel)
76     {
77         if (this.bikeIsRemovedFromSimulation) return;
78
79         var pivot = this.head.Center.ToGrx();
80         var w1 = this.wheel1.Center.ToGrx();
81         var w2 = this.wheel2.Center.ToGrx();
82         float angleInDegree = Vector2D.Angle360(new Vector2D(1, 0), w2 - w1);
83
84         this.armsUpSprite.RotateZAngleInDegree = this.armsDownSprite.RotateZAngleInDegree = angleInDegree;
85         this.armsUpSprite.RotateYAngleInDegree = this.armsDownSprite.RotateYAngleInDegree = this.yAngleInDegree;
86
87         panel.DisableDepthTesting();
88         panel.ZValue2D = 0;
89         DrawSticks(panel, pivot, w1, w2);           //Zeichne die Linien von den Rädern zum Gehäuse
90         DrawSprite(panel, pivot);                  //Zeichne die Sprite-Datei
91         DrawWheels(panel, w1, w2);                //Zeichne die Räder
92
93         panel.EnableDepthTesting();
94     }
95
96     private void DrawSprite(GraphicPanel2D panel, Vector2D pivot)
97     {
98         //Ermittle, welche Sprite-Datei gezeichnet werden soll
99         var sprite = GetActiveSprite();
100
101         if (this.arm == ArmState.NoArmMovement)
102         {
103             //Zeichne nur Frame 0 von der aktiven Sprite
104             sprite.DrawSingleImage(panel, pivot, 0);
105         }
106         else
107         {
108             //Zeichne die Armbewegung (hoch oder runter)
109             sprite.Draw(panel, pivot);
110         }
111     }
112 }

```

Objekt rein grafisch zerlegen

Wenn der Motorradfahrer mit sein Kopf gegen eine Wand stößt zerspringt das Motorrad in Einzelteile. Diese Teile sind dann nicht mehr Teil der Physiksimulation. Für diese Zerlegung wird der VoronoiExploder benutzt, welcher als Input ein IDrawable benötigt. Dieser Exploder erstellt ein Bitmap von dem Objekt, was zerlegt werden soll, und zerlegt dann dieses Bitmap in Einzelteile, welche über die HandleTimerTick-Methode vom Exploder bewegt werden und über Draw gezeichnet werden.

```

GameViewModel.cs ✘ ×
ElmaControl

66     this.bike = new Bike(this.simulator, this.sounds, DataFolder, this.panel);
67     this.bike.HeadIsTouchingTheGround += Bike_HeadIsTouchingTheGround;
68
69     private void Bike_HeadIsTouchingTheGround()
70     {
71         this.sounds.PlayWallCrash();
72         this.exploder = new VoronoiExploder(this.bike, this.panel);
73         this.bike.RemoveBikeFromSimulation();
74         this.tickCounter.StartTimer(() => { BikeIsBrokenHandler.Invoke(this.lastLoadedLevel, this.keyboardFile, this.state == MainState.Replay); SaveKeyboardData(); });
75     }
76
77     private void Refresh()
78     {
79         this.simulator.Draw(this.panel);
80         if (this.exploder != null)
81         {
82             this.exploder.Draw(this.panel);
83         }
84
85         //Wenn das Motorrad kaputt ist, dann bweegt der Exploder noch die Splitter
86         if (this.exploder != null)
87         {
88             this.exploder.HandleTimerTick(dt);
89         }
90     }

```

Nutzung des Leveleditors innerhalb des Spiels

Der Leveleditor ist ein WPF-Usercontrol was über ein ContentControl angezeigt wird. Dieses wird über die LevelEditorFactory erstellt und über SetSimulatorBuildMethod übergibt man dem Editor eine Builder-Methode, welche ein eigenen Simulator erstellt, welcher die Spiel- und Anzeigelogik enthält.

```

58     case ControlType.LevelEditor: SubControlFactory.cs
59     {
60         return new LevelEditorFactory().CreateEditorControl(new EditorInputData(data)
61         {
62             DataFolder = data.DataFolder + "Levels\\",
63             ShowGoBackButton = true
64         });
65     }
66
67     internal class MainViewModel : ReactiveObject, ITimerHandler, IGraphicPanelHandler, IPHysicsSimulated
68     {
69         private SubControlFactory factory;
70
71         private UserControl mainSelect, levelSelect, singleLevel, gameControl, levelEditor, spriteEditor;
72
73         [Reactive] public System.Windows.Controls.UserControl ContentUserControl { get; set; }
74
75         this.levelEditor = this.factory.CreateControl(ControlType.LevelEditor);
76
77         //Übergebe Builder-Methode für ein ILeveleditorUsedSimulator-Objekt
78         this.LevelEditor.DataContext as ISimulatorUser).SetSimulatorBuildMethod(
79             (data, panelSize, camera, timerIntercallInMilliseconds) =>
80             (this.gameControl.DataContext as GameViewModel)
81                 .CreateSimulator(data, panelSize, camera, timerIntercallInMilliseconds)
82         );
83
84         case ControlType.LevelEditor:
85         {
86             this.ContentUserControl = this.levelEditor; break;
87             (this.levelEditor.DataContext as IToTextWriteable).LoadFromTextFile(this.emptyLevelFile);
88         }
89     }

```

Schritt 1: Leveleditor-Usercontrol erstellen

Schritt 2: Level laden und UserControl anzeigen

Siehe: DemoApplications/Elma/ElmaControl/Controls/Main/MainViewModel.cs

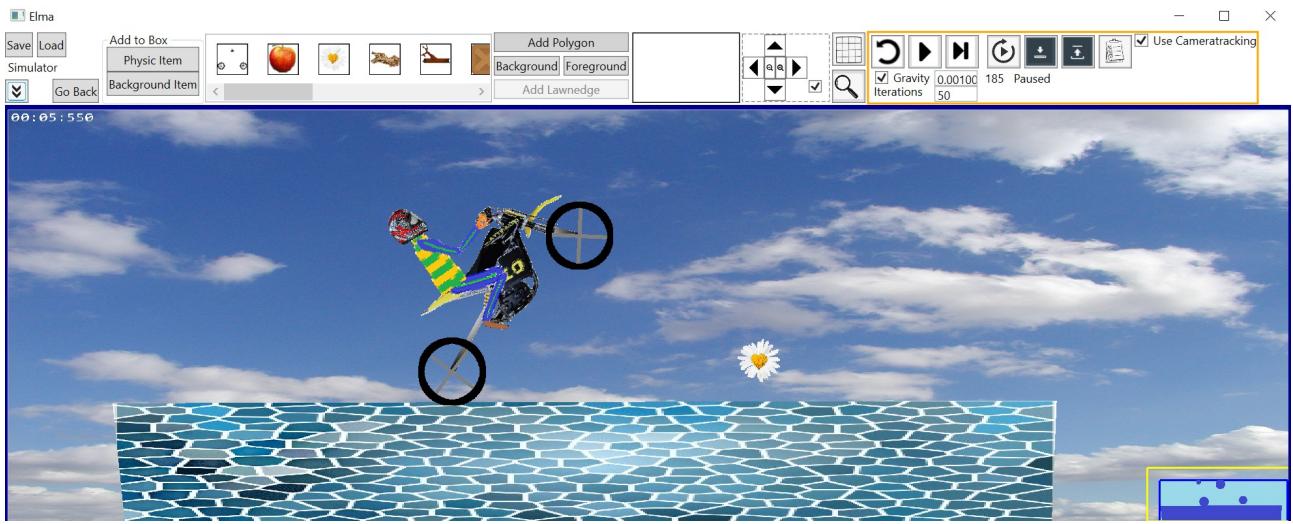
Der Simulator, welcher dem Leveleditor übergeben wird leitet vom GameSimulator ab. Der Leveleditor nutzt zum Erstellen des Simulator-Objektes den Konstruktor, welcher ein SimulatorInputData-Objekt bekommt. Der ElmaSimulator benötigt zur Soundausgabe noch zusätzlich ein Sounds-Objekt. Aus dem Grund gibt es noch eine Init-Methode, welche dann nach Erstellen des ElmaSimulators die Spielobjekte anlegt. Diese Überschreiben dann auch das Standardverhalten zur Anzeige vom Motorrad und der Äpfel, indem sie die UseCustomDrawingForRigidBody-Methode nutzen. Der ElmaSimulator erweitert den Simulator vom Leveleditor um eigene Grafik- und Spielelogik dadurch, dass er zuerst über (1) den Standardkonstruktor erstellt wird, dann über die Init-Funktion (2) werden eigene Zeichenfunktionen übergeben. Durch das Überschreiben der Draw-Methode (3) wird noch ein ElapsedTime-Text angezeigt und durch die überschriebene MoveOneStep-Methode (4) feuert der Simulator ein Event, wenn das Motorrad die Blume oder Wand berührt. Dieses Event wird vom Leveleditor aber nicht ausgewertet.

```

GameViewModel.cs
53     public ILeveleditorUsedSimulator CreateSimulator(SimulatorInputData data, Size panelSize, Camera2D camera, float timerIntercallInMilliseconds)
54     {
55         var sim = new ElmaSimulator(data, panelSize, camera, timerIntercallInMilliseconds); 1
56         sim.Init(this.sounds, this.DataFolder, this.panel); 2
57         return sim;
58     }
59
60
ElmaSimulator.cs
14     internal class ElmaSimulator : GameSimulator
15     {
16         //Wird vom Leveleditor genutzt
17         1 Verweis
18         public ElmaSimulator(SimulatorInputData data, Size panelSize, Camera2D camera, float timerIntercallInMilliseconds) 1
19         :base(data, panelSize, camera, timerIntercallInMilliseconds)
20         {
21         }
22
23         //Wird vom GameViewModel genutzt
24         1 Verweis
25         public ElmaSimulator(string levelFile, float timerIntercallInMilliseconds, Sounds sounds, string dataFolder, GraphicPanel2D panel)
26         :base(levelFile, panel.Size, timerIntercallInMilliseconds)
27         {
28             Init(sounds, dataFolder, panel);
29         }
30
31         public override void Draw(GraphicPanel2D panel) 3
32         {
33         }
34         public override void MoveOneStep(float dt) 4
35         {
36         }

```

Dank der Verwendung des ElmaSimulator zeigt der Leveleditor dann Sprites an und das Motorrad lässt sich laut der Spiellogik steuern:

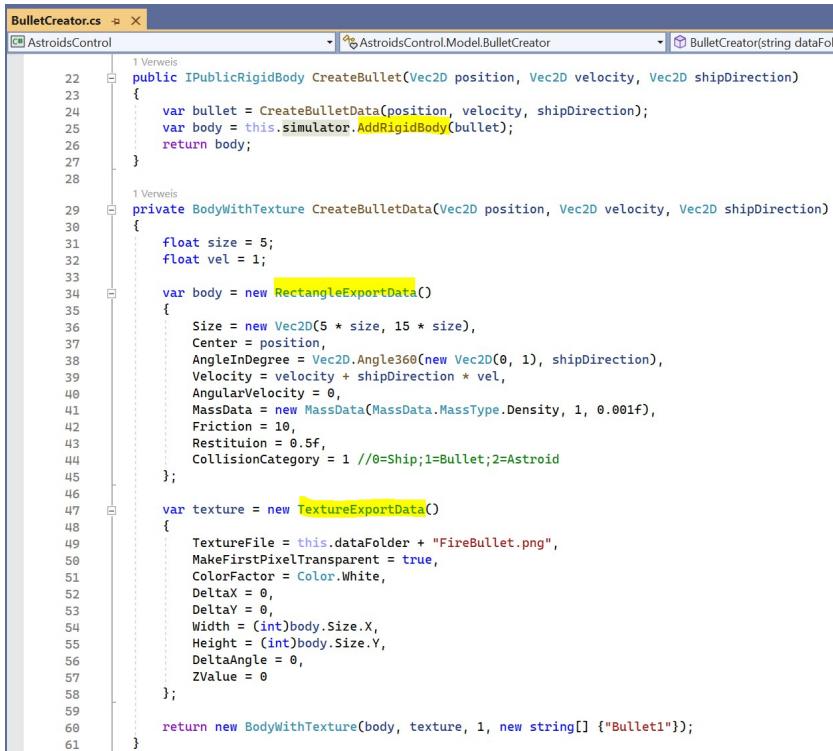


So kann man den Leveleditor im Spiel zur Levelerstellung nutzen und innerhalb des Editors kann das Spiel dann auch gleich getestet werden.

Game 4 – Astroids

Rechteck während der Simulation erstellen

Über AddRigidBody kann man ein einzelnes Rechteck/Kreis/Polygon während der Simulationszeit erstellen. In diesen Fall ist dass der Feuerball, den das Raumschiff verschießt. Er wird in Blickrichtung des Raumschiffes gefeuert.



```
BulletCreator.cs
AstroidsControl
AstroidsControl.Model.BulletCreator
BulletCreator(string dataFolder)
1 Verweis
public IPublicRigidBody CreateBullet(Vec2D position, Vec2D velocity, Vec2D shipDirection)
{
    var bullet = CreateBulletData(position, velocity, shipDirection);
    var body = this.simulator.AddRigidBody(bullet);
    return body;
}

1 Verweis
private BodyWithTexture CreateBulletData(Vec2D position, Vec2D velocity, Vec2D shipDirection)
{
    float size = 5;
    float vel = 1;

    var body = new RectangleExportData()
    {
        Size = new Vec2D(5 * size, 15 * size),
        Center = position,
        AngleInDegree = Vec2D.Angle360(new Vec2D(0, 1), shipDirection),
        Velocity = velocity + shipDirection * vel,
        AngularVelocity = 0,
        MassData = new MassData(MassData.MassType.Density, 1, 0.001f),
        Friction = 10,
        Restitution = 0.5f,
        CollisionCategory = 1 //0=Ship;1=Bullet;2=Astroid
    };

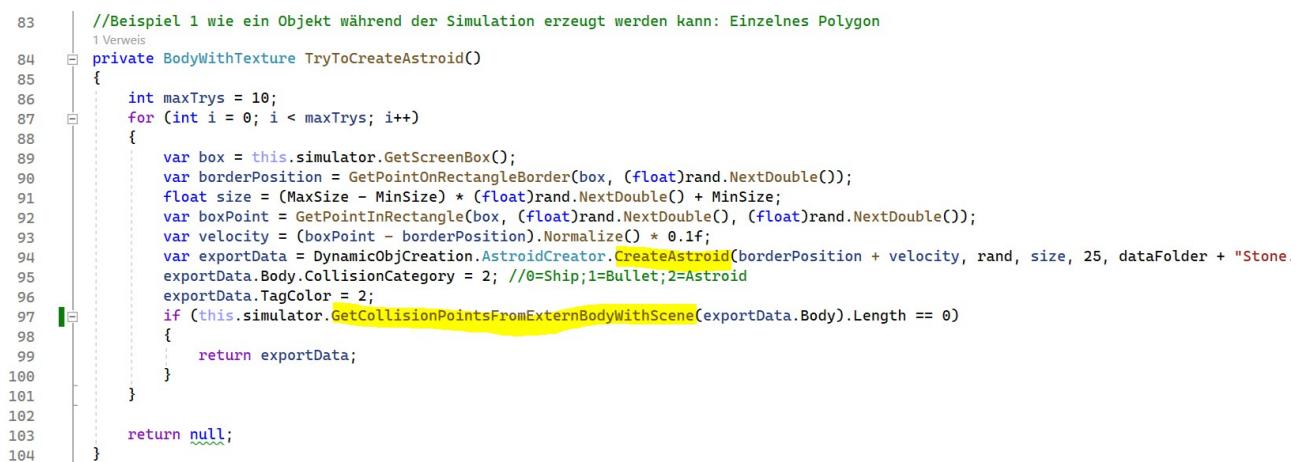
    var texture = new TextureExportData()
    {
        TextureFile = this.dataFolder + "FireBullet.png",
        MakeFirstPixelTransparent = true,
        ColorFactor = Color.White,
        DeltaX = 0,
        DeltaY = 0,
        Width = (int)body.Size.X,
        Height = (int)body.Size.Y,
        DeltaAngle = 0,
        ZValue = 0
    };

    return new BodyWithTexture(body, texture, 1, new string[] {"Bullet1"});
}
```

Siehe: DemoApplications/AstroidsControl/Model/BulletCreator.cs

Polygon dynamisch erzeugen und prüfen ob es mit der Szene kollidiert

Wenn ich ein Polygon während der Simulation erzeugen will und vor dem Einfügen im Simulator will ich aber prüfen, ob es mit anderen Objekten kollidiert, dann geht das über die GetCollisionPointsFromExternBodyWithScene-Funktion.



```
//Beispiel 1 wie ein Objekt während der Simulation erzeugt werden kann: Einzelnes Polygon
1 Verweis
private BodyWithTexture TryToCreateAstroid()
{
    int maxTries = 10;
    for (int i = 0; i < maxTries; i++)
    {
        var box = this.simulator.GetScreenBox();
        var borderPosition = GetPointOnRectangleBorder(box, (float)rand.NextDouble());
        float size = (MaxSize - MinSize) * (float)rand.NextDouble() + MinSize;
        var boxPoint = GetPointInRectangle(box, (float)rand.NextDouble(), (float)rand.NextDouble());
        var velocity = (boxPoint - borderPosition).Normalize() * 0.1f;
        var exportData = DynamicObjCreation.AstroidCreator.CreateAstroid(borderPosition + velocity, rand, size, 25, dataFolder + "Stone");
        exportData.Body.CollisionCategory = 2; //0=Ship;1=Bullet;2=Astroid
        exportData.TagColor = 2;
        if (this.simulator.GetCollisionPointsFromExternBodyWithScene(exportData.Body).Length == 0)
        {
            return exportData;
        }
    }

    return null;
}
```

Siehe: DemoApplications/AstroidsControl/Model/AstroidCreator.cs

Komplexes Objekt dynamisch erzeugen

Wenn ich ein Objekt während der Simulation erzeugen will, was aus mehreren

Körpern/Gelenken/Schubdüsen besteht, dann erzeuge ich zuerst dieses Objekt im Leveleditor und vergabe dafür ein Prototyp-TagName, über welchen ich das Objekt zuerst per GetExportDataFromLevelItem aus dem Level extrahiere und dann mit RemoveLevelItem entferne. Dieser Schritt wird einmal beim Simulationsstart gemacht (Zeile 40,41).

Von diesen extrahierten Objekt erzeuge ich eine Kopie und lege dann seine Position/Geschwindigkeit fest (Zeile 113,115,116). Ich kann prüfen, ob es Kollisionspunkte mit der Szene gibt (Zeile 117).

Über AddLevelItem füge ich das neue Objekt ein und bekomme mit GetBodyByTagName Zugriff auf seine IPublic-Objekte (Zeile 72,73).

```

26     private PhysikLevelItemExportData satellitExport;
27
28     //Export-Daten vom Satelliten sich holen und dann den Satallit entfernen
29     int satellitId = this.simulator.GetTagDataFromBody(this.simulator.GetBodiesByTagName("satellit").First()).LevelItemId;
30     this.satellitExport = this.simulator.GetExportDataFromLevelItem(satellitId);
31     this.simulator.RemoveLevelItem(satellitId);
32
33     //Beispiel 2 zur Objekterzeugung: Komplexes Objekt, was per Kopie von ein im LevelEditor erstellten LevelItem erzeugt wird
34     //1 Verweis
35
36     //Beispiel 2 zur Objekterzeugung: Komplexes Objekt, was per Kopie von ein im LevelEditor erstellten LevelItem erzeugt wird
37     //1 Verweis
38
39     private PhysikLevelItemExportData TryToCreateSatellit()
40     {
41
42         var box = this.simulator.GetScreenBox();
43         var borderPosition = GetPointOnRectangleBorder(box, (float)rand.NextDouble());
44         var boxPoint = GetPointInRectangle(box, (float)rand.NextDouble(), (float)rand.NextDouble());
45         var velocity = (boxPoint - borderPosition).Normalize() * 0.1f;
46         var copyData = new PhysikLevelItemExportData(this.satellitExport); //Kopie vom Original-Export erstellen
47         float angleInDegree = Vec2D.Angle360(new Vec2D(-1, 0), velocity);
48         LevelItemExportHelper.MoveToPivotPoint(copyData, (borderPosition + velocity).ToGrx(), LevelItemExportHelper.PivotOrientation.Center, 1,
49         LevelItemExportHelper.SetVelocityFromAllBodies(copyData, velocity));
50         if (this.simulator.GetCollisionPointsFromExternLevelItemWithScene(copyData).Length > 0) return null;
51         return copyData;
52     }
53
54     var satellitData = TryToCreateSatellit();
55     if (satellitData != null)
56     {
57         int newId = simulator.AddLevelItem(satellitData);
58         var mainBody = simulator.GetBodyByTagName(newId, "satellitMain");
59         satellitInsideTester.AddBody(mainBody);
60     }
61
62 }
```

Siehe: DemoApplications/AstroidsControl/Model/AstroidCreator.cs

Kollision zwischen dynamisch erzeugten Objekten ermitteln

Wenn man Objekte per AddRigidBody/AddLevelItem eingefügt hat, dann hängt die Reihenfolge, ob nun Objekt A mit Objekt B kollidiert, oder B mit A davon ab, in welcher Reihenfolge die Objekte eingefügt wurden. Mit dem TagOrderedCollisionOccured-Event werden die beiden Objekte des Kollisionspunktes nach der TagColor sortiert. Damit ist dann die Abfrage auf Zeile 55 leichter, da ich nicht noch die Reihenfolge so wie hier beachten muss:

```
if ((c.Color1 == 1 && c.Color2 == 2) || (c.Color1 == 2 && c.Color2 == 1))
```

sondern es reicht:

```
if (c.Color1 == 1 && c.Color2 == 2)
```

```

AstroidDestroyer.cs
AstroidsControl
AstroidsControl.Model.AstroidDestroyer
32     simulator.TagOrderedCollisionOccured += Simulator_CollisionOccured;
33
34     private void Simulator_CollisionOccured(GameSimulator sender, TagColorOrderedCollisionEvent[] collisions)
35     {
36         foreach (var c in collisions)
37         {
38             //0=Ship;1=Bullet;2=Astroid;3=Satellit
39
40             //Schuss berührt Astroid
41             if (c.Color1 == 1 && c.Color2 == 2)
42             {
43                 ...
44             }
45         }
46     }

```

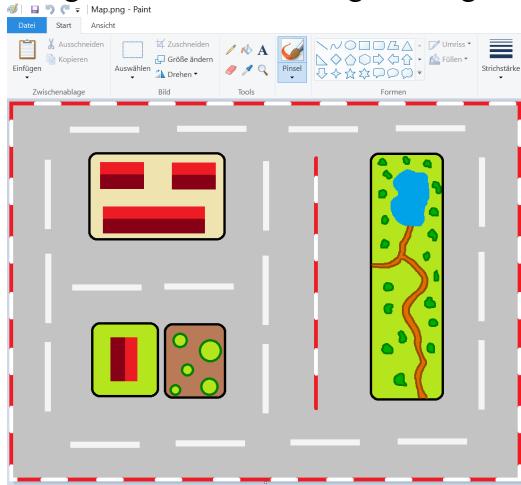
Game 5 – CarDrifter

Bei diesen Spiel wird gezeigt, wie man ein Spiel erzeugt, wo man von oben auf die Karte schaut und wo das Hintergrundbild mit den Physikobjekten verknüpft ist (TopDown-Game). Außerdem kann man das Spiel auch im Leveleditor einbinden.

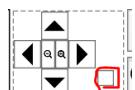
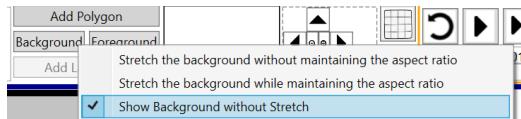
Erstellung eines TopDown-Games

Das sind die Schritte, um ein TopDown-Game zu erzeugen, bei dem das Hintergrundbild mit dem Physikmodell verknüpft ist.

Schritt 1: Im Paint die Karte erzeugen, welche im Hintergrund angezeigt wird:

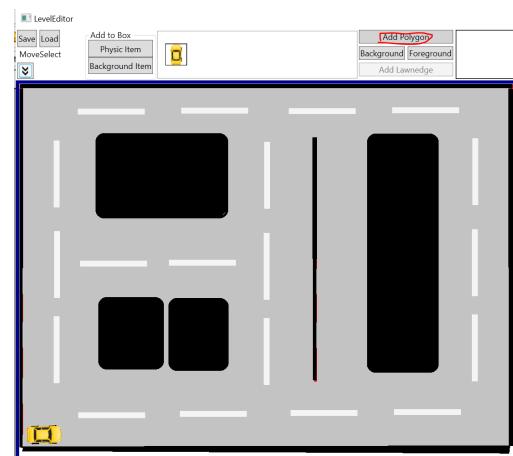


Schritt 2: Zeige mit „Background“ das soeben erzeugte Bild im Leveleditor an. Nutze den „Without Stretch“-Modus dazu:

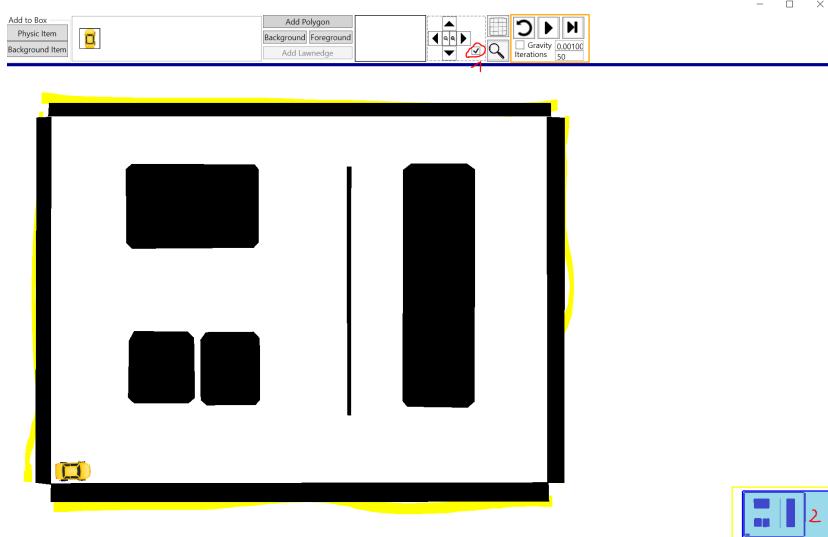


Schritt 3: Nutze als Foreground `black.png` und deaktiviere bei der Kamera den AutoZoom

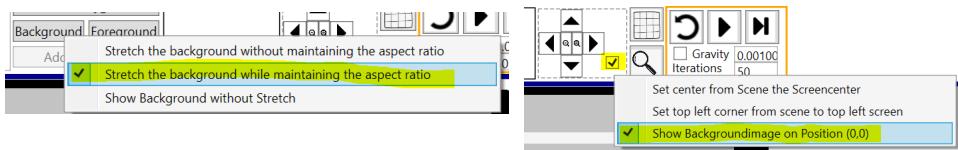
Schritt 4: Nutze „Add Polygon“ und erstelle nun die Wände von der Karte. Diese werden schwarz angezeigt:



Schritt 5: Mache die Außenwände dicker, damit das Auto dort wegen des Tunnel-Effekts nicht durchfahren kann. Aktiviere dazu den AutoZoom (1) und zoome die Kamera etwas raus (2) und verschiebe sie so, dass alle schwarzen Polygone gut sichtbar sind. Editiere dann die 4 Außenwände, indem die Randpunkte etwas verschoben werden so dass die Wand dicker wird:



Schritt 6: Aktiviere den Stretch-Modus:



Schritt 7: Nutze als Foreground die Transparente Farbe: Transparent.png

Ab jetzt sind die unsichtbaren Wände mit dem Hintergrundbild verknüpft und die Karte wird immer so groß angezeigt, wie das Fenster Platz bietet.

Innerhalb vom Leveleditor das Spiel anzeigen

Wenn ich ein Spiel erzeugen will, wo der Leveleditor Teil des Spiels sein soll oder wenn man schauen will, welche Kräfte auf die Spielfigur wirken, dann ist es möglich, dass man dem Leveleditor von außen eine Simulator-BUILDER-Methode rein gibt, welcher ein Simulator mit eigener Spiellogik erzeugt. Der Simulator der von außen rein gegeben wird muss dazu das Interface `ILeveleditorUsedSimulator` implementieren.

So wird ein Leveleditor erzeugt wo der Simulator Spiellogik enthält. Zuerst wird der Leveleditor über die `LevelEditorFactory` (Zeile 26) erzeugt und dann wird über `SetSimulatorBuildMethod` ein Simulator-BUILDER reingegeben (Zeile 28), welcher ein `ILeveleditorUsedSimulator`-Objekt erzeugt (Zeile 51). So ein Objekt stelle ich dadurch bereit, indem ich von der `GameSimulator`-Klasse ableite und dort erfolgt dann über `override` der `Draw/MoveOneStep`-Methoden die Anpassung des Simulators dann so, dass er die Spiellogik implementiert.

```

21  public class CarDrifterWithLevelEditorControlFactory : IEditorFactory
22  {
23      public System.Windows.Controls.UserControl CreateEditorControl(EditorInputData data)
24      {
25          var vm = new MainViewModel(data.Panel, data.SoundGenerator, data.TimerTickRateInMs, data.DataFolder); //Game-ViewModel
26          var levelEditorControl = new LevelEditorFactory().CreateEditorControl(data);
27          (LevelEditorControl.DataContext as ITextWriterable).LoadFromFile(data.DataFolder + "CarDrifter.txt"); //Level einladen
28          (LevelEditorControl.DataContext as ISimulatorUser).SetSimulatorBuildMethod(
29              (data, panelSize, camera, timerIntercallInMilliseconds) => (vm).CreateSimulator(data, panelSize, camera, timerIntercallInMilliseconds)
30          );
31
32          return levelEditorControl;
33      }
34  }
35
36  public ILeveleditorUsedSimulator CreateSimulator(SimulatorInputData data, Size panelSize, Camera2D camera, float timerIntercallInMilliseconds)
37  {
38      var sim = new CarDrifterSimulator(data, panelSize, camera, timerIntercallInMilliseconds);
39      sim.Init(this.sounds, this.DataFolder, this.panel);
40      return sim;
41  }
42
43  internal class CarDrifterSimulator : GameSimulator

```

Bei Elma wurde der Leveleditor erzeugt, während das Spiel lief. Hier wird ein Leveleditor mit Spiellogik durch die `EditorControlFactory` direkt bei Programstart erzeugt.

Game 6 – Bridge Builder

Bei diesen Spiel wird über ein eigenen Editor das Level-Polygon und die Endpunkte der Brückenstangen definiert. Der Zug wurde im Leveleditor erstellt. Hier wird gezeigt, wie man das Level-Polygon, die Brücke und den Zug zu ein gemeinsamen Level vereint. Außerdem wird hier auch demonstriert, wie man ein WPF-Button/WPF-Label auf dem GraphicPanel anbringt.

Leveleditor-Objekt in Nutzergenerierter Szene platzieren

Der Zug wurde im Leveleditor erstellt und wird auf Zeile 51 eingeladen. Es wird bei Zeile 55 die Boundingbox von den Zug ermittelt und dann bei Zeile 77 der Zug vom GameSimulator entfernt. Die Brücke und das Levelelement wurde in der GetBridgeAndGround-Funktion erstellt und über AddLevelItem dem Simulator hinzugefügt. Über TranslateScene wird das Export-Objekt von den Zug so manipuliert, dass er genau auf dem Boden aufsetzt. Über AddLevelItem (Zeile 110) wird der Zug dann der Simulation später hinzugefügt. Somit enthält der GameSimulator dann die Brücke, den Boden und den Zug. Das ist ähnlich wie beim Satelliten beim Asteroids-Game.

```
EditorToSimulatorConverter.cs
50     string levelFile = dataFolder + "\\Train.txt";
51     var simulator = new GameSimulator(EditorFileConverter.Convert(levelFile), panelSize, input.Camera, timerIntervalInMilliseconds);
52
53     int trainId = simulator.GetTagDataFromBody(simulator.GetBodiesByTagName("Train").First()).LevelItemId;
54     var trainExport = simulator.GetExportDataFromLevelItem(trainId);
55     var trainBox = PhysicSceneExportDataHelper.GetBoundingBoxFromScene(trainExport.PhysicSceneData);
56     float groundHeight = GetGroundPolygonPoints(input.LevelExport).Where(x => x.X < trainBox.Max.X).Min(x => x.Y);
57     float trainBottom = trainBox.Max.Y;
58     simulator.RemoveLevelItem(trainId); //Entferne den Zug mit falscher Y-Koordinate
59
60     //Setze den Zug genau auf dem Boden auf
61     PhysicSceneExportDataHelper.TranslateScene(trainExport.PhysicSceneData, Matrix4x4.Translate(0, groundHeight - trainBottom, 0));
62     trainExportData = trainExport;
63
64     int bridgeAndGroundId = simulator.AddLevelItem(GetBridgeAndGround(input, dataFolder, out ExportObject[] exportObjects));
65
66     private static PhysikLevelItemExportData GetBridgeAndGround(SimulatorInput input, string dataFolder, out ExportObject[] exportObjects)
67     {
68
69         PhysikLevelItemExportData result = null;
70
71         if (bridgeAndGroundId != -1)
72         {
73             result = new PhysikLevelItemExportData();
74             result.LevelItemId = bridgeAndGroundId;
75             result.LevelExport = input.LevelExport;
76             result.Bodies = input.Bodies;
77             result.Train = trainExport;
78             result.Box = trainBox;
79             result.BridgeAndGround = true;
80             result.BridgeAndGroundId = bridgeAndGroundId;
81             result.BridgeAndGroundX = trainBox.Max.X;
82             result.BridgeAndGroundY = trainBox.Max.Y;
83
84             exportObjects = new ExportObject[1];
85             exportObjects[0] = result;
86         }
87
88         return result;
89     }
90
91     public void SimulateBridgeFunction()
92     {
93         simulator.AddLevelItem(trainExportData);
94     }
95
96 }
```

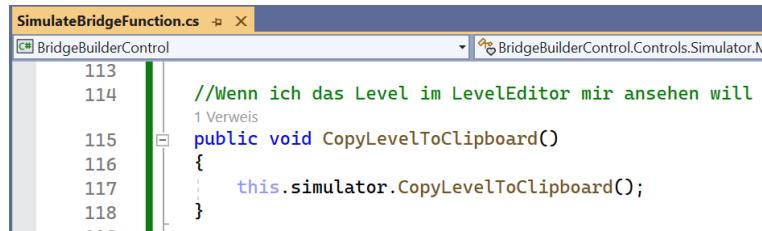
Ein WPF-Button auf dem GraphicPanel platzieren

Wenn man bei ein Grid ein WPF-Button über das GraphicPanel platzieren würde, dann würde man den Button nicht sehen, da das GraphicPanel ein WinForm-Control ist, was über ein WindowsFormsHost in die WPF-Welt eingebunden wurde. WPF erlaubt es nicht, dass man über ein WinForm-Element einfach darüber zeichnet. Es gibt aber die WPF-Popup-Klasse, welche ein Fenster ohne Rahmen ist. So ein Popup darf über ein WinForm-Element platziert werden. Allerdings hat die Standard-Popup-Klasse keine Möglichkeit, dass man dieses Fenster an ein Child-Element dran klebt. Deswegen habe ich die AirspacePopup-Klasse erstellt, welche als Kindelement den Button bekommt. Über HorizontalAlignment/VerticalAlignment/Margin wird das Popup-Fenster dann innerhalb des GraphicPanels platziert.

```
1 <UserControl x:Class="BridgeBuilderControl.Controls.BridgeEditor.BridgeEditorControl"
2   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
5   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
6   xmlns:local="clr-namespace:BridgeBuilderControl.Controls.BridgeEditor"
7   xmlns:wpfControls="clr-namespace:WpfControls.Controls;assembly=WpfControls"
8   xmlns:i="http://schemas.microsoft.com/xaml/behaviors"
9   mc:Ignorable="d"
10  d:DesignHeight="450" d:DesignWidth="800">
11
12  <Grid>
13
14    <wpfControls:AirspacePopup PlacementTarget="{Binding ElementName=graphicControlBorder}"
15      FollowPlacementTarget="True"
16      AllowOutsideScreenPlacement="True"
17      ParentWindow="{Binding RelativeSource={RelativeSource AncestorType={x:Type Window}}}"
18      IsOpen="True"
19      AllowsTransparency="True"
20      Placement="RelativePoint"
21      CoversEntireTarget="False"
22      HorizontalAlignment="Left" VerticalAlignment="Top" Margin="10 5 0 0">
23
24      <Button Content="Exit" Command="{Binding GoBackClick}" FontSize="30" FontFamily="Adobe Caslon Pro" Style="{StaticResource MainTextBlockStyle}"/>
25    </wpfControls:AirspacePopup>
26
27    <Border BorderThickness="5" BorderBrush="DarkBlue" x:Name="graphicControlBorder"/>
28
29  </Grid>
```

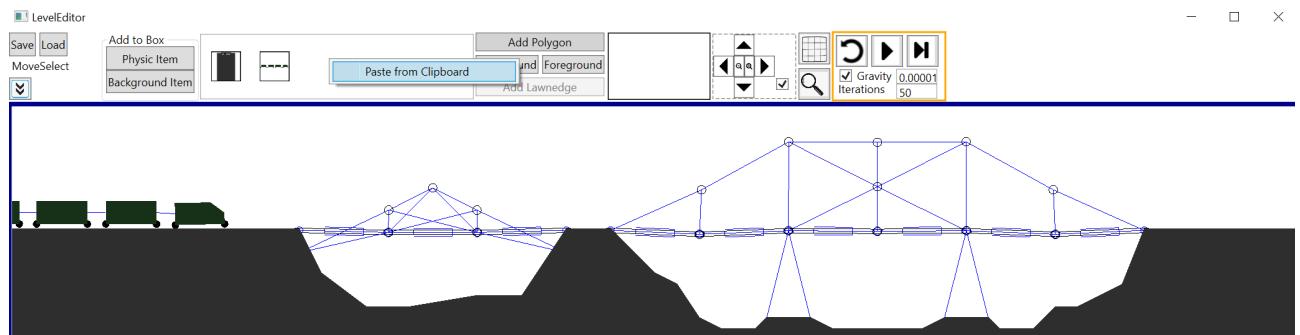
Dynamisch erzeugtes Level im Leveleditor ansehen

Das Level bei Bridgebuilder ist ein Mix aus dynamisch zur Laufzeit erzeugten Physikobjekten (Brücke, Boden) und aus Leveleditor-Objekten (Zug). Wenn man das gesamte Level im Leveleditor ansehen / untersuchen will, dann gibt es in der GameSimulator-Klasse die CopyLevelToClipboard-Funktion.



```
SimulateBridgeFunction.cs  X
BridgeBuilderControl
BridgeBuilderControl.Controls.Simulator.N
113
114     //Wenn ich das Level im LevelEditor mir ansehen will
115     1 Verweis
116     public void CopyLevelToClipboard()
117     {
118         this.simulator.CopyLevelToClipboard();
119     }
120
```

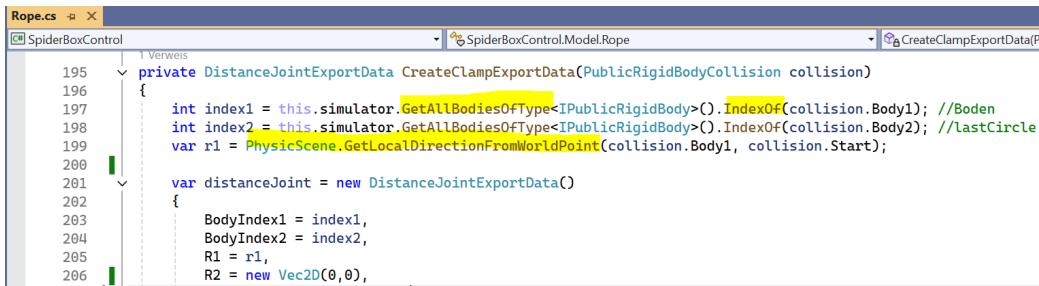
Es kopiert alle Levelitems in die Zwischenablage. Starte ich dann den Leveleditor und nutze die Paste from Clipboard-Funktion, dann werden beide Levelitems (Brücke+Boden; Zug) in die Prototypbox eingefügt und ich kann dann per Drag & Drop beide Items in den Leveleditor reinziehen und das ganze dann auch simulieren.



Game 7 – SpiderBox

Joints dynamisch erzeugen

Wenn man ein Joint erzeugen will, nimmt man normalerweise den PhysicSceneEditor. Wenn man aber zwei LevelItems per Joint verbinden will, dann kann man das über die AddJoint-Methode aus dem GameSimulator machen. Wenn das Endstück vom Seil gegen den Boden kommt, dann soll es am Boden kleben bleiben. Dazu wird ein Distanzjoint genommen. Über GetAllBodies wird der BodyIndex ermittelt und über GetLocalDirectionFromWorldPoint der Hebelarm.



```
Rope.cs  SpiderBoxControl  CreateClampExportData(P)
1 Verweis
195 private DistanceJointExportData CreateClampExportData(IPublicRigidBodyCollision collision)
196 {
197     int index1 = this.simulator.GetAllBodiesOfType<IPublicRigidBody>().IndexOf(collision.Body1); //Boden
198     int index2 = this.simulator.GetAllBodiesOfType<IPublicRigidBody>().IndexOf(collision.Body2); //lastCircle
199     var r1 = PhysicScene.GetLocalDirectionFromWorldPoint(collision.Body1, collision.Start);
200
201     var distanceJoint = new DistanceJointExportData()
202     {
203         BodyIndex1 = index1,
204         BodyIndex2 = index2,
205         R1 = r1,
206         R2 = new Vec2D(0,0),
207     };
208 }
```

Über AddJoint wird das DistanceJoint dann der Simulation hinzugefügt.

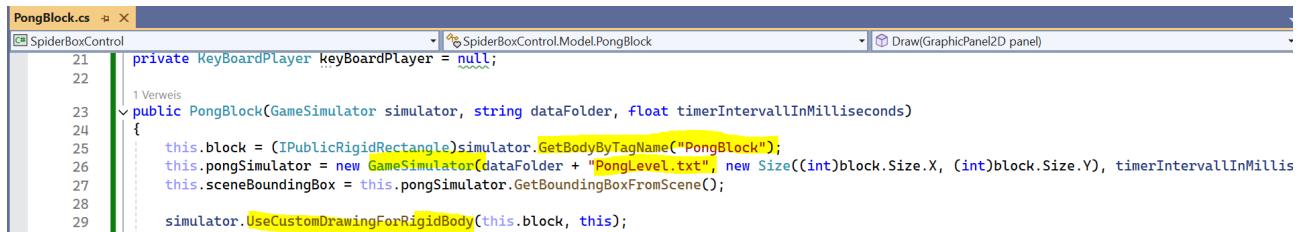
```
108 private void Simulator_CollisionOccurred(RigidbodyPhysics.PhysicScene sender, RigidbodyPhysics.CollisionDetection.PublicRigidBodyCollision[] col
109 {
110     if (this.clampLast != null) return;
111
112     foreach (var collision in collisions)
113     {
114         if (collision.Body2 == this.lastCircle)
115         {
116             this.clampLast = (IPublicDistanceJoint)sender.AddJoint(CreateClampExportData(collision));
117             this.clampLast.LengthPosition = this.circleRadius * 3;
118         }
119     }
120 }
```

Über RemoveJoint kann es wieder entfernt werden.

```
242 public void RemoveFromSimulation()
243 {
244     this.simulator.RemoveJoint(this.clampFirst);
245
246     if (this.clampLast != null)
247         this.simulator.RemoveJoint(this.clampLast);
248 }
```

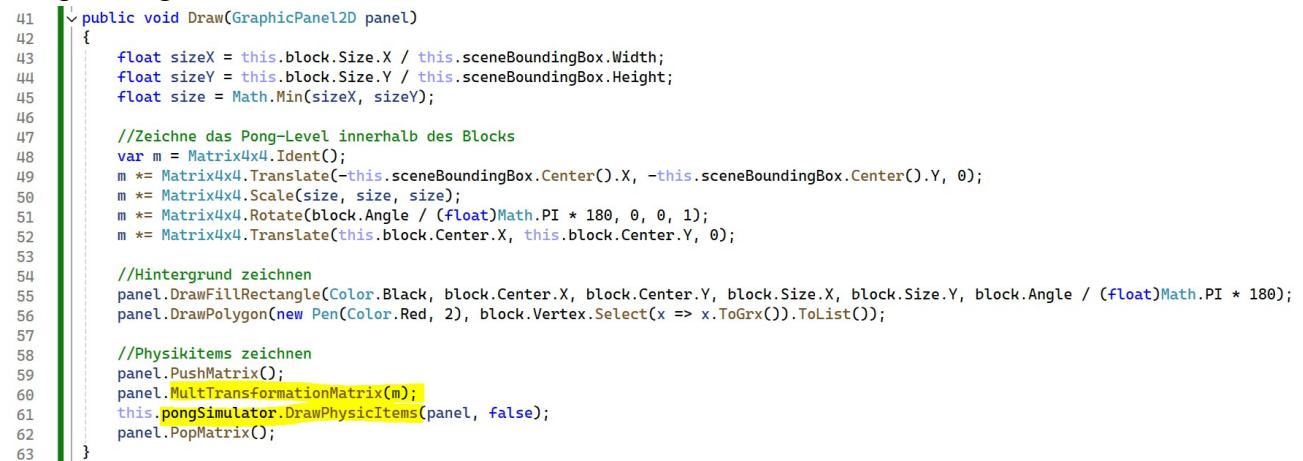
Physiksimulation anstelle von Sprites

Wenn man eine Figur animieren will, welche aus mehreren Teil-Animationen besteht und man möchte diese Animation ähnlich wie eine Sprite auf ein anderes Objekt drauf kleben, dann kann man dazu die GameSimulator-Klasse nehmen. So könnte man dann den Bogenschütze von Castle Attack animieren oder man zeigt ein Pong-Spiel innerhalb einer Box an, welche Teil einer Physiksimulation ist. Die Bewegung vom PongBlock wird über die Physikengine gesteuert. Zur Anzeige wird aber ein CustomDrawer genommen:



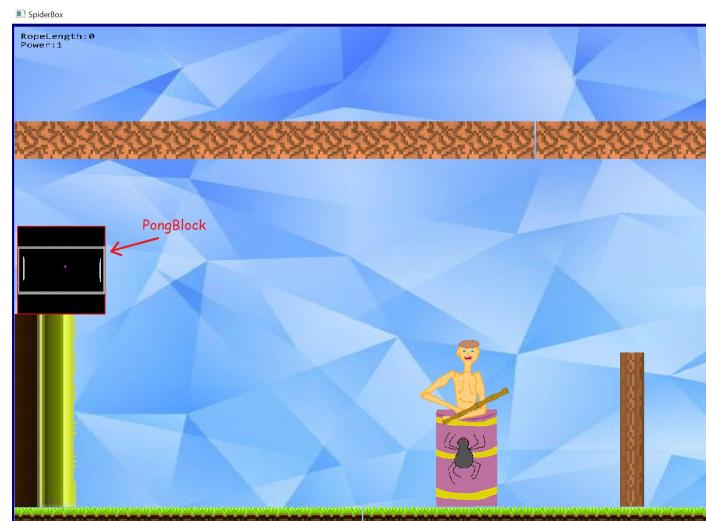
```
21 private KeyboardPlayer keyBoardPlayer = null;
22
23     1 Verweis
24     public PongBlock(GameSimulator simulator, string dataFolder, float timerIntervallInMilliseconds)
25     {
26         this.block = (IPublicRigidRectangle)simulator.GetBodyByTagName("PongBlock");
27         this.pongSimulator = new GameSimulator(dataFolder + "PongLevel.txt", new Size((int)block.Size.X, (int)block.Size.Y), timerIntervallInMillis);
28         this.sceneBoundingBox = this.pongSimulator.GetBoundingBoxFromScene();
29
30         simulator.UseCustomDrawingForRigidBody(this.block, this);
31     }
32 }
```

Dieser CustomDrawer nutzt eine GameSimulator-Klasse zur Visualisierung. Über MultTransformationMatrix wird dafür gesorgt, dass die Pong-Animation dann genau innerhalb vom PongBlock gezeichnet wird.



```
41     public void Draw(GraphicPanel2D panel)
42     {
43         float sizeX = this.block.Size.X / this.sceneBoundingBox.Width;
44         float sizeY = this.block.Size.Y / this.sceneBoundingBox.Height;
45         float size = Math.Min(sizeX, sizeY);
46
47         //Zeichne das Pong-Level innerhalb des Blocks
48         var m = Matrix4x4.Ident();
49         m *= Matrix4x4.Translate(-this.sceneBoundingBox.Center().X, -this.sceneBoundingBox.Center().Y, 0);
50         m *= Matrix4x4.Scale(size, size, size);
51         m *= Matrix4x4.Rotate(block.Angle / (float)Math.PI * 180, 0, 0, 1);
52         m *= Matrix4x4.Translate(this.block.Center.X, this.block.Center.Y, 0);
53
54         //Hintergrund zeichnen
55         panel.DrawFillRectangle(Color.Black, block.Center.X, block.Center.Y, block.Size.X, block.Size.Y, block.Angle / (float)Math.PI * 180);
56         panel.DrawPolygon(new Pen(Color.Red, 2), block.Vertex.Select(x => x.ToGrx()).ToList());
57
58         //Physikitems zeichnen
59         panel.PushMatrix();
60         panel.MultTransformationMatrix(m);
61         this.pongSimulator.DrawPhysicItems(panel, false);
62         panel.PopMatrix();
63     }
64 }
```

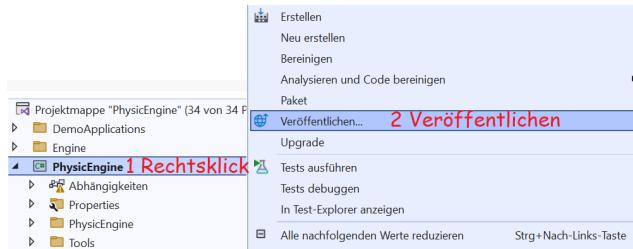
Der schwarze PongBlock wird mit ein GameSimulator bewegt. Innerhalb des Blocks wird ein Pong-Spiel angezeigt, was wiederum eine Physiksimulation ist, welche ähnlich wie eine Sprite auf dem PongBlock drauf klebt. Der Vorteil ist, dass man die Animation viel komplexer als eine einfache Sprite sein kann.



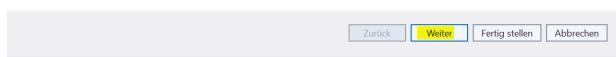
Einzelne Exedatei erstellen

Es gibt beim PhysicEngine-Projekt nur ein Projekt, was eine WPF-Exe erstellt. Im bin/Debug-Ordner vom Hauptprojekt landen dann alle Dlls und die PhysicEngine.exe. Wenn ich alle Dlls mit in die Exe reinpacken will, dann geht das laut diesen Schritten:

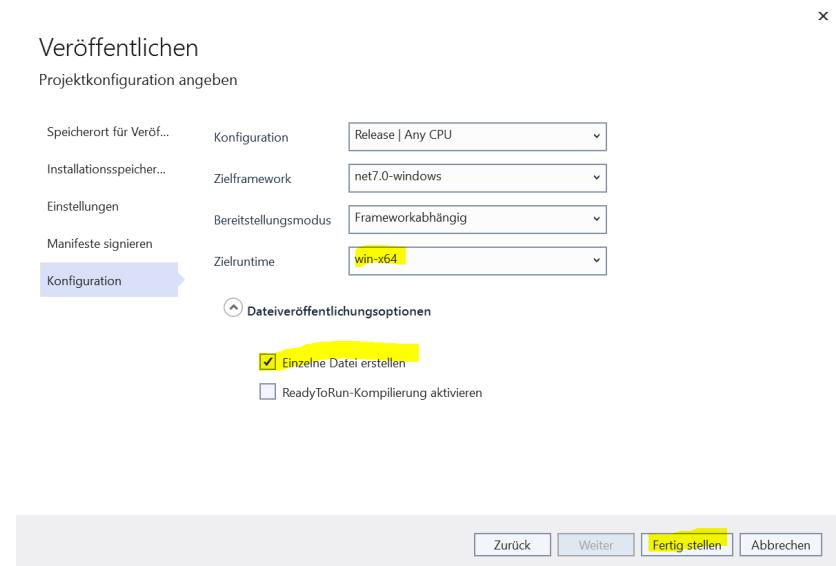
Rechtsklick auf das Hauptprojekt und dann auf Veröffentlichen gehen:



In den ersten 4 Fenstern jeweils alles so lassen wie es ist und auf weiter klicken:



Im Konfiguration-Fenster als Zielruntime „win-x64“ und „Einzelne Datei erstellen“:



Dann auf Veröffentlichen klicken:



Dann die PhysicEngine.exe aus

RigidBodyPhysics\PhysicEngine\Source\PhysicEngine\bin\Release\net7.0-windows\win-x64\app.publish nach PhysicGames\PhysicGames\Exe kopieren und alle Ordner von RigidBodyPhysics\PhysicEngine\Data\GameData nach PhysicGames\PhysicGames\Data

Teil 11: Zusammenfassung

Um die Bewegung von Starrkörpern zu simulieren muss man in einer TimeStep-Methode einfach nur folgende Dinge tun:

```

48     public void TimeStep(float dt)
49     {
50         foreach (var body in this.bodies)
51             body.Force += new Vec2D(0, Gravity);
52
53         var collisionPoints = GetCollisionPoints(this.bodies);
54
55         var constraints = CreateConstraints(bodies, collisionPoints);
56
57         foreach (var body in this.bodies)
58         {
59             body.Velocity.X += body.InverseMass * body.Force.X * dt;
60             body.Velocity.Y += body.InverseMass * body.Force.Y * dt;
61             body.AngularVelocity += body.InverseInertia * body.Torque * dt;
62
63             for (int i = 0; i < this.IterationCount; i++)
64             {
65                 foreach (var c in constraints)
66                 {
67                     Vec2D relativeVelocity = GetRelativeVelocity(c.B1, c.R1, c.B2, c.R2);
68                     float velocityInForceDirection = relativeVelocity * c.ForceDirection;
69                     float impulse = c.ImpulseMass * (c.Bias - velocityInForceDirection);
70
71                     float oldSum = c.AccumulatedImpulse;
72                     c.AccumulatedImpulse = Clamp(oldSum + impulse, c.MinImpulse, c.MaxImpulse);
73                     impulse = c.AccumulatedImpulse - oldSum;
74
75                     var impulseVec = impulse * c.ForceDirection;
76
77                     c.B1.Velocity -= impulseVec * c.B1.InverseMass;
78                     c.B1.AngularVelocity -= Vec2D.ZValueFromCross(c.R1, impulseVec) * c.B1.InverseInertia;
79
80                     c.B2.Velocity += impulseVec * c.B2.InverseMass;
81                     c.B2.AngularVelocity += Vec2D.ZValueFromCross(c.R2, impulseVec) * c.B2.InverseInertia;
82
83                 }
84             }
85
86             foreach (var body in this.bodies)
87             {
88                 body.Center += body.Velocity * dt;
89                 body.Angle += body.AngularVelocity * dt;
90                 body.Force = new Vec2D(0, 0);
91             }
92         }
93     }

```

```

class Body
{
    public Vec2D Force;
    public float Torque;
    public Vec2D Center;
    public float Angle;
    public Vec2D Velocity;
    public float AngularVelocity;
    public float InverseMass;
    public float InverseInertia;
}

class Constraint
{
    public Body B1, B2;
    public float AccumulatedImpulse = 0;
    public float ImpulseMass;
    public float Bias;
    public float MinImpulse;
    public float MaxImpulse;
    public Vec2D ForceDirection;
    public Vec2D R1, R2;
}

```

Auf die Körper wirkt die Schwerkraft und die Abstoßkraft bei den Kollisionspunkten, welche verhindert, dass die Körper sich überlappen. Die Abstoßkraft wird dadurch ermittelt, indem man sich überlegt, welcher Geschwindigkeit die Kontaktpunkte der Körper nach dem Zusammenstoß zueinander haben sollen. Auch weiß man, dass diese Kraft für eine Dauer von dt Sekunden wirken soll. Somit könnte man ein Gleichungssystem aufstellen, was alle Kräfte, die auf alle Körper wirken enthält, und dieses per Projected Gauss Seidel lösen. Da das Gleichungssystem aber viele Nullen enthält, was viel Rechenaufwand bedeutet, verwendet man das Sequentielle Impulsverfahren, was von der Idee das gleiche tut, aber effektiver ist.

Wenn wir jetzt auf das Beispiel aus dem Intro zurück kommen, dann sehen wir, dass diese Joint-Klasse eine Distanzconstraint ist, welche eine 2D-Linear-Constraint ohne Soft ist.

<https://github.com/erincatto/box2d-lite/blob/master/src/Joint.cpp>

```

44     // deltaV = deltaV0 + K * impulse
45     // invM = [(1/m1 + 1/m2) * eye(2) - skew(r1) * invI1 * skew(r1) - skew(r2) * invI2 * skew(r2)]
46     //      = [1/m1+1/m2      0] + invI1 * [r1.y*r1.y - r1.x*r1.y] + invI2 * [r1.y*r1.y - r1.x*r1.y]
47     //      [      0       1/m1+1/m2]           [-r1.x*r1.y r1.x*r1.x]           [-r1.x*r1.y r1.x*r1.x]
48     if (World::positionCorrection)
49     {
50         bias = -biasFactor * inv_dt * dp;
51     }
52     if (World::warmStarting)
53     {
54         // Apply accumulated impulse.
55         body1->velocity -= body1->invMass * P;
56         body1->angularVelocity -= body1->invI * Cross(r1, P);
57
58         body2->velocity += body2->invMass * P;
59         body2->angularVelocity += body2->invI * Cross(r2, P);
60     }

```

Sie nutzt die gleiche K-Matrix wie die Point2Point-Constraint, weil ja beim Ableiten der PositionConstraint der Soll-Distanzwert zu Null wird. Per PositionCorrection werden die Kontaktpunkte um dp verschoben und Warmstart wendet den Impuls vom letzten TimeStep an. Das ist also alles kein Hexenwerk, wenn man einmal solche Begriffe wie effektive Masse, VelocityConstraint, PositionCorrection, Warmstart und Skew-Matrix verstanden hat.