Physic Engines

Jil Zerndt
FS 2025

Einführung und Grundlagen

Physics Engines Überblick

Physics Engines Physics Engines sind Softwarekomponenten, die physikalische Effekte in Computerprogrammen simulieren. Unity verwendet PhysX als Standard-Physics-Engine.
Ziele des Moduls:
- Physikalische Modellierung in Unity verstehen
- Grundprinzipien der Mechanik für realistische Simulationen anwenden
- Kopplung von Physiksimulatoren mit realistischen Parametern

Modellbildungsprozess: Wirklichkeit → Physikalisches Modell → Mathematisches Modell → Numerisches Modell → Unity-Implementation

Bezugssysteme in der Mechanik

Bezugssystem Ein Bezugssystem definiert:
- Einen Nullpunkt im Raum
- Die Richtungen der Koordinatenachsen (x, y, z)
- Eine Zeitmessung

Dadurch wird die Position eines Körpers eindeutig durch einen Ortsvektor $\vec{r}$ beschrieben.

Vektoren Ein Vektor ist eine physikalische Größe mit Betrag und Richtung.
- Darstellung: $\vec{r}$ (mit Pfeil über dem Symbol)
- Betrag: $|\vec{r}| = r$ (ohne Pfeil)
- In Koordinatendarstellung: $\vec{r} = \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix}$
- Einheitsvektor (Betrag $= 1$): $\vec{e}_r = \frac{\vec{r}}{|\vec{r}|}$

Rechenregeln für Vektoren
- Addition: $\vec{r}_1 + \vec{r}_2 = \begin{pmatrix} r_{x1} \\ r_{y1} \\ r_{z1} \end{pmatrix} + \begin{pmatrix} r_{x2} \\ r_{y2} \\ r_{z2} \end{pmatrix} = \begin{pmatrix} r_{x1} + r_{x2} \\ r_{y1} + r_{y2} \\ r_{z1} + r_{z2} \end{pmatrix}$
- Skalarprodukt (ergibt einen Skalar): $s = \vec{r}_1 \cdot \vec{r}_2 = |\vec{r}_1| \cdot |\vec{r}_2| \cdot \cos \angle(\vec{r}_1, \vec{r}_2) = r_{x1} r_{x2} + r_{y1} r_{y2} + r_{z1} r_{z2}$
- Kreuzprodukt (ergibt einen Vektor): $\vec{r}_1 \times \vec{r}_2 = \begin{pmatrix} r_{y1} r_{z2} - r_{z1} r_{y2} \\ r_{z1} r_{x2} - r_{x1} r_{z2} \\ r_{x1} r_{y2} - r_{y1} r_{x2} \end{pmatrix}$

Unity vs. Standard-Koordinatensystem **Unterschiede:**
- Standard-Physik: Rechtssystem
- Unity: Linkssystem
- Unity: positive y-Achse zeigt nach oben

Auswirkungen: Unterschiedliche Kreuzprodukt-Ergebnisse und Rotationsrichtungen.

SI-Einheiten Wichtige SI-Einheiten in der Mechanik:
- Länge in Meter (m)
- Masse in Kilogramm (kg)
- Zeit in Sekunden (s)
- Kraft in Newton (N $=$ kg $\cdot$ m/s^2)

Es ist wichtig, in Unity konsequent SI-Einheiten zu verwenden und bei allen Werten entsprechende Einheiten anzugeben.

Konventionen für Unity-Code

Deklaration von physikalischen Größen

```
// Korrekte Deklaration physikalischer Groessen in Unity
public float initialVelocity = 6.0f; // in m/s
public float mass = 1.0f; // in kg
public float springConstant = 10.0f; // in N/m
```

Abfrage von Vektoren

```
// Abfrage von Position und Geschwindigkeit
Vector3 position = rigidBody.position; // in m
Vector3 velocity = rigidBody.velocity; // in m/s
```

Berechnung von Kräften

```
// Beispiel: Federkraft berechnen
Vector3 springForce = -springConstant * (position - equilibriumPosition);
rigidBody.AddForce(springForce); // Kraft in N hinzufuegen
```

Für die Umrechnung der Geschwindigkeit von km/h in m/s teilt man durch 3,6: $v[m/s] = \frac{v[km/h]}{3,6}$
Beispiel: 72 km/h = 72 / 3,6 = 20 m/s

Modulstruktur und Bewertung

Kursaufbau
- 2 ECTS = 60h total (30h Präsenz, 30h Selbststudium)
- Moodle-Übungen online: 10% (4h)
- Unity-Projekte in Gruppen, 3 Phasen (16h total, 40%):
  - Einführungsbeispiel, Teil 1: 9. März (5%)
  - Teil 2: 9. April (15%)
  - Teil 3: 9. Mai (20%)
- Mündliche Abschlussprüfung: 50% (10h)

Modulinhalte **Physik-Themen:**
- Koordinatensysteme und Bezugssysteme
- Kinematik: Beschreibung von Bewegungen
- Dynamik: Einfluss von Kräften
- Kräfte: Reibung, Anziehung, Kollisionen
- Energie und Impuls: Erhaltungsgesetze
- Rotationsbewegungen und Anwendungen

Unity-Grundlagen

Vector3 in Unity
- `Vector3` für 3D-Positionen und -Richtungen
- Wichtige Eigenschaften: `Vector3.forward`, `Vector3.up`, `Vector3.right`
- Operationen: Skalarprodukt, Kreuzprodukt, Betrag, Normalisierung

Entwicklungsumgebung Wichtige Ressourcen für das Modul:

```
// Online verfuegbare Kursressourcen:
// - Moodle: https://moodle.zhaw.ch/course/view.php?id=17534
// - EduWiki: https://eduwiki.engineering.zhaw.ch/wiki/PE_Physik_Engines
// - GitHub: https://github.zhaw.ch/physicsenginesmodule
// - Tipler Lehrbuch: https://link.springer.com/book/10.1007/978-3-662-58281-7
// - Unity fuer Uebungen
// - MS Teams fuer Kommunikation
```

Prüfungsvorbereitung

Theorie-Komponente

- Physik-Formeln aus wöchentlicher Formelsammlung studieren
- Herleitungen und Anwendungen jeder Formel verstehen
- Problemlösung mit schrittweisem Vorgehen üben

Unity-Implementation

- Vector3-Operationen und Koordinatentransformationen beherrschen
- Rigidbody-Komponente und Physiksimulation verstehen
- Kräfte und Bewegungsgleichungen in C# implementieren üben

Projektarbeit

- Alle Projektphasen gründlich dokumentieren
- Physikalische Prinzipien hinter implementierten Features verstehen
- Code-Implementation und Physik-Konzepte erklären können

Formel-Eingabe in Word Kursanforderung: Formelsammlung mit Word-Formeleditor führen.

- Alt + Shift + * (oder Alt + Shift + =) für Gleichungen
- Tiefgestellt: Unterstrich (_)
- Hochgestellt: Zirkumflex (^)
- Formatierung mit Leertaste abschließen

Prüfungshinweis: Die mündliche Prüfung (15 Min) fokussiert auf Physik-Konzepte. Unity-spezifische Details sind nicht prüfungsrelevant, aber Projektverständnis wird erwartet.

Kinematik

Kinematik Die Kinematik beschreibt die Bewegung ohne Betrachtung der Ursachen. Eine Bewegung wird vollständig charakterisiert durch:

- Ort: $\vec{r}(t)$
- Geschwindigkeit: $\vec{v}(t) = \frac{d\vec{r}}{dt}$
- Beschleunigung: $\vec{a}(t) = \frac{d\vec{v}}{dt} = \frac{d^2\vec{r}}{dt^2}$

Zusammenhänge zwischen Ort Geschwindigkeit und Beschleunigung

- Geschwindigkeit = Ableitung des Ortes nach der Zeit: $\vec{v} = \frac{d\vec{r}}{dt}$
- Beschleunigung = Ableitung der Geschwindigkeit nach der Zeit: $\vec{a} = \frac{d\vec{v}}{dt}$
- Ort = Integral der Geschwindigkeit nach der Zeit: $\vec{r} = \int \vec{v}\, dt$
- Geschwindigkeit = Integral der Beschleunigung nach der Zeit: $\vec{v} = \int \vec{a}\, dt$

Geschwindigkeit und Beschleunigung

Mittlere vs. Momentangeschwindigkeit **Mittlere Geschwindigkeit:**

$$\bar{v}_x = \frac{\Delta r_x}{\Delta t} = \frac{r_x(t_2) - r_x(t_1)}{t_2 - t_1} \tag{1}$$

Momentangeschwindigkeit:

$$v_x(t) = \lim_{\Delta t \to 0} \frac{\Delta r_x}{\Delta t} = \frac{dr_x}{dt} \tag{2}$$

Unterschied zu Schnelligkeit:

- Geschwindigkeit: Vektorielle Größe mit Richtung
- Schnelligkeit: Betrag der Geschwindigkeit (Skalar)
- $|\vec{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$

Mittlere vs. Momentanbeschleunigung **Mittlere Beschleunigung:**

$$\bar{a}_x = \frac{\Delta v_x}{\Delta t} = \frac{v_x(t_2) - v_x(t_1)}{t_2 - t_1} \tag{3}$$

Momentanbeschleunigung:

$$a_x(t) = \lim_{\Delta t \to 0} \frac{\Delta v_x}{\Delta t} = \frac{dv_x}{dt} = \frac{d^2 r_x}{dt^2} \tag{4}$$

Differenzenquotient vs. Differentialquotient

- Der Differenzenquotient (mittlere Geschwindigkeit) ist eine Approximation über ein endliches Zeitintervall: $\frac{\Delta r_x}{\Delta t}$
- Der Differentialquotient (Momentangeschwindigkeit) ist der Grenzwert für ein infinitesimal kleines Zeitintervall: $\lim_{\Delta t \to 0} \frac{\Delta r_x}{\Delta t} = \frac{dr_x}{dt}$
- In Unity wird mit fixen Zeitschritten $\Delta t = 20$ ms gerechnet, was einer Abtastfrequenz von $f_{sample} = 50$ Hz entspricht

Momentangeschwindigkeit und -beschleunigung

Momentangeschwindigkeit Die Momentangeschwindigkeit zur Zeit t_0 ist definiert als:

$$\vec{v}(t_0) = \lim_{t_1 \to t_0} \frac{\Delta \vec{r}}{t_1 - t_0} = \frac{d\vec{r}}{dt} \tag{5}$$

Sie entspricht geometrisch der Steigung der Tangente im Punkt $(t_0, r_x(t_0))$.

Der Betrag der Geschwindigkeit wird oft als Schnelligkeit bezeichnet:

$$|\vec{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2} \tag{6}$$

Bei gleichbleibender Schnelligkeit kann sich dennoch die Richtung der Geschwindigkeit ändern, z.B. bei einer Kreisbewegung.

Fläche unter dem Geschwindigkeits-Zeit-Diagramm Bei einer Bewegung mit variablem $v(t)$ berechnet sich die zurückgelegte Strecke als Fläche unter der v-t-Kurve:

$$\Delta x = \int_{t_1}^{t_2} v(t)\, dt \tag{7}$$

Integration und Differentiation

Ableitungsregeln

- Konstante Summanden: $\frac{d}{dt}(C) = 0$
- Potenzfunktionen: $\frac{d}{dt}(at^n) = a \cdot n \cdot t^{n-1}$
- Exponentialfunktion: $\frac{d}{dx}(e^x) = e^x$
- Logarithmus: $\frac{d}{dx}(\ln x) = \frac{1}{x}$
- Sinus/Kosinus: $\frac{d}{dx}(\sin x) = \cos x$, $\frac{d}{dx}(\cos x) = -\sin x$

Regeln für zusammengesetzte Funktionen

- Summenregel: $\frac{d}{dt}(f(t) + g(t)) = \frac{df}{dt} + \frac{dg}{dt}$
- Produktregel: $\frac{d}{dt}(f(t) \cdot g(t)) = \frac{df}{dt} \cdot g(t) + f(t) \cdot \frac{dg}{dt}$
- Kettenregel: $\frac{d}{dt}(f(g(t))) = \frac{df}{dg} \cdot \frac{dg}{dt}$

Berechnung von Bewegungen mit konstanter Beschleunigung

Gegebene Größen

- Anfangsposition r_0
- Anfangsgeschwindigkeit v_0
- Konstante Beschleunigung a

Schritte zur Berechnung

1. Geschwindigkeit in Abhängigkeit von der Zeit bestimmen:

$$v(t) = v_0 + at \tag{8}$$

2. Position in Abhängigkeit von der Zeit bestimmen:

$$r(t) = r_0 + v_0 t + \frac{1}{2} a t^2 \tag{9}$$

3. Alternative Formel bei bekannter Strecke (ohne Zeit):

$$v^2 = v_0^2 + 2a(r - r_0) \tag{10}$$

Bewegung in Unity implementieren

```
// Implementierung von Bewegungen mit konstanter Beschleunigung
void FixedUpdate() {
    // Aktuelle Zeit seit Start
    currentTime += Time.deltaTime;

    // Aktuelle Geschwindigkeit nach v = v0 + a*t berechnen
    float currentVelocity = initialVelocity + acceleration * currentTime;

    // Bewegung mit aktueller Geschwindigkeit
    Vector3 displacement = new Vector3(currentVelocity, 0, 0) * Time.deltaTime;
    transform.position += displacement;

    // Alternative: Direkte Berechnung der Position mit r = r0 + v0*t + 0.5*a*t^2
    // Vector3 newPosition = initialPosition + initialVelocity * currentTime +
    //                       0.5f * acceleration * currentTime * currentTime;
    // transform.position = newPosition;
}
```

Freier Fall Ein Körper fällt aus der Höhe r_0 mit Anfangsgeschwindigkeit $v_0 = 0$.

- Beschleunigung: $a(t) = -g$ (g = 9.81 m/s²)
- Geschwindigkeit: $v(t) = -gt$
- Position: $r(t) = r_0 - \frac{1}{2}gt^2$

Alternativ: Ein Körper wird mit Anfangsgeschwindigkeit v_0 nach oben geworfen:

- Maximale Höhe: $h_{max} = \frac{v_0^2}{2g}$
- Zeit bis zum höchsten Punkt: $t_{max} = \frac{v_0}{g}$
- Gesamtflugzeit: $t_{gesamt} = \frac{2v_0}{g}$

Unity-Implementation

Position und Geschwindigkeit in Unity

```
// Aktuelle Position eines GameObjects abrufen
Vector3 currentPosition = transform.position;

// Verschiebung zwischen zwei Positionen berechnen
Vector3 displacement = finalPosition - initialPosition;

// Position ueber Zeit aktualisieren
transform.position += velocity * Time.deltaTime;
```

Kinematische Bewegung implementieren

```
public class KinematicMotion : MonoBehaviour
{
    public Vector3 initialVelocity = Vector3.zero;
    public Vector3 acceleration = Vector3.zero;

    private Vector3 initialPosition;
    private float startTime;

    void Start()
    {
        initialPosition = transform.position;
        startTime = Time.time;
    }

    void Update()
    {
        float t = Time.time - startTime;

        // Neue Position mit kinematischer Gleichung berechnen
        Vector3 newPosition = initialPosition +
                              initialVelocity * t +
                              0.5f * acceleration * t * t;

        transform.position = newPosition;
    }
}
```

Spezialfälle der Bewegung

Gleichförmige Bewegung ($\vec{a} = 0$):

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

Freier Fall in Unity ($\vec{a} = -g\hat{j}$):

- $g = 9.81\,m/s^2$ (Erdbeschleunigung)
- Unity Standard-Gravitation: $-9.81\,m/s^2$ in Y-Richtung

Kinematische Probleme lösen

Schritt 1: Bekannte Variablen identifizieren

- Anfangsposition $\vec{r}_0$
- Anfangsgeschwindigkeit $\vec{v}_0$
- Beschleunigung $\vec{a}$
- Zeit t oder End-Position/Geschwindigkeit

Schritt 2: Passende Gleichung wählen

- $\vec{v}(t) = \vec{v}_0 + \vec{a}t$ wenn Zeit bekannt ist
- $\vec{r}(t) = \vec{r}_0 + \vec{v}_0 t + \frac{1}{2}\vec{a}t^2$ für Position
- $\vec{v}^2 = \vec{v}_0^2 + 2\vec{a} \cdot \Delta\vec{r}$ wenn Zeit unbekannt ist

Schritt 3: Komponentenweise lösen

- Vektoren in x-, y-, z-Komponenten aufteilen
- Jede Komponente unabhängig lösen
- Ergebnisse zum finalen Vektor kombinieren

Wurfbewegung Ein Ball wird aus Höhe $h = 10m$ mit Anfangsgeschwindigkeit $\vec{v}_0 = (5, 8, 0)\,m/s$ geworfen. Berechne Zeit bis zum Aufprall und horizontale Distanz.

Gegeben: $\vec{r}_0 = (0, 10, 0)$, $\vec{v}_0 = (5, 8, 0)$, $\vec{a} = (0, -9.81, 0)$
Y-Komponente (vertikal): $y(t) = 10 + 8t - 4.905t^2$
Ball trifft Boden bei $y(t) = 0$: $10 + 8t - 4.905t^2 = 0$ $t = 2.24s$ (mit quadratischer Formel)
X-Komponente (horizontal): $x(t) = 5t = 5 \times 2.24 = 11.2m$

Dynamik

Dynamik Die Dynamik untersucht die Ursachen von Bewegungen - die wirkenden Kräfte. Grundlage sind die Newton'schen Gesetze.

Newton'sche Axiome Isaac Newton formulierte die drei grundlegenden Gesetze der Bewegung:
1. Trägheitsgesetz: Ein Körper bleibt im Zustand der Ruhe oder der gleichförmigen geradlinigen Bewegung, solange keine Kraft auf ihn wirkt.
2. Bewegungsgesetz: Die Änderung der Bewegung ist proportional zur einwirkenden Kraft und erfolgt in Richtung der Kraft.
3. Wechselwirkungsgesetz: Übt ein Körper auf einen anderen eine Kraft aus (actio), so wirkt eine gleich große, entgegengesetzte Kraft zurück (reactio).

Ein viertes Prinzip ist das Superpositionsprinzip: Kräfte addieren sich vektoriell.

Newton'sche Gesetze

Trägheitsgesetz Das erste Newton'sche Gesetz:

$$\vec{F} = 0 \Rightarrow \vec{v} = \text{const.} \tag{11}$$

Dies bedeutet auch: $\vec{F} = 0 \Rightarrow \vec{a} = 0$
Das Gesetz gilt nur in Inertialsystemen (Bezugssysteme ohne Beschleunigung oder Rotation).

Impuls Der Impuls $\vec{p}$ eines Körpers ist das Produkt aus seiner Masse und seiner Geschwindigkeit:

$$\vec{p} = m \cdot \vec{v} \tag{12}$$

Der Impuls ist eine vektorielle Größe mit der Einheit kg · m/s.

Bewegungsgesetz Das zweite Newton'sche Gesetz in seiner allgemeinen Form:

$$\vec{F} = \frac{d\vec{p}}{dt} \tag{13}$$

Für Körper mit konstanter Masse:

$$\vec{F} = m \cdot \vec{a} \tag{14}$$

In integraler Form:

$$\vec{p} = \vec{p}_0 + \int_0^t \vec{F}(t)\, dt \tag{15}$$

wobei $\int \vec{F}\, dt$ als Kraftstoß bezeichnet wird.

Wechselwirkungsgesetz Das dritte Newton'sche Gesetz:

$$\vec{F}_{12} = -\vec{F}_{21} \tag{16}$$

Kräftepaare haben folgende Eigenschaften:
1. Gleicher Betrag, entgegengesetzte Richtung
2. Greifen an verschiedenen Körpern an
3. Haben die gleiche physikalische Ursache

Superpositionsprinzip Mehrere Kräfte addieren sich vektoriell:

$$\vec{F}_{res} = \sum_{i=1}^{n} \vec{F}_i \tag{17}$$

Für jede Komponente:

$$F_x = \sum_{i=1}^{n} F_{xi} \quad F_y = \sum_{i=1}^{n} F_{yi} \quad F_z = \sum_{i=1}^{n} F_{zi} \tag{18}$$

Wichtige Krafttypen

Grundlegende Kräfte

Gravitationskraft (nahe Erdoberfläche):

$\vec{F}_g = m\vec{g}$ mit $g = 9.81\, m/s^2$

Federkraft (Hooke'sches Gesetz):

$\vec{F}_s = -k\Delta\vec{l}$

Reibungskraft:

- Haftreibung: $f_s \leq \mu_s N$
- Gleitreibung: $f_k = \mu_k N$

Dämpfungskraft:

$\vec{F}_d = -b\vec{v}$ (geschwindigkeitsproportional)

Harmonischer Oszillator

Harmonische Schwingung Bei einer Federkraft entsteht eine harmonische Schwingung mit der Bewegungsgleichung:

$$m\frac{d^2x}{dt^2} = -kx$$

Lösung: $x(t) = A\cos(\omega t + \phi)$
wobei $\omega = \sqrt{\frac{k}{m}}$ die Kreisfrequenz ist.

Eigenschaften des harmonischen Oszillators

Kreisfrequenz:

$\omega = \sqrt{\frac{k}{m}}$

Schwingungsdauer:

$T = \frac{2\pi}{\omega} = 2\pi\sqrt{\frac{m}{k}}$

Frequenz:

$f = \frac{1}{T} = \frac{\omega}{2\pi}$

Gesamtenergie:

$E = \frac{1}{2}kA^2$ (konstant)

Unity Implementation

Unity Force Modes Unity bietet verschiedene Modi für Kraftanwendung über `Rigidbody.AddForce()`:
- `Force`: Kontinuierliche Kraft mit Masse (Standard)
- `Acceleration`: Kontinuierliche Kraft ohne Masse
- `Impulse`: Impulsive Kraft mit Masse
- `VelocityChange`: Impulsive Kraft ohne Masse

Kräfte in Unity anwenden

```
public class ForceController : MonoBehaviour
{
    private Rigidbody rb;
    public float springConstant = 100f;
    public float damping = 10f;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    void FixedUpdate()
    {
        // Gravitation (automatisch von Unity angewandt)
        // Vector3 gravity = Physics.gravity * rb.mass;

        // Federkraft zum Ursprung
        Vector3 springForce = -springConstant * transform.position;

        // Daempfungskraft
        Vector3 dampingForce = -damping * rb.velocity;

        // Gesamtkraft anwenden
        rb.AddForce(springForce + dampingForce);
    }
}
```

Harmonischer Oszillator in Unity

```
public class HarmonicOscillator : MonoBehaviour
{
    public float springConstant = 50f;
    public float amplitude = 2f;
    public float phase = 0f;

    private float mass;
    private float omega;
    private Vector3 equilibrium;
    private float startTime;

    void Start()
    {
        mass = GetComponent<Rigidbody>().mass;
        omega = Mathf.Sqrt(springConstant / mass);
        equilibrium = transform.position;
        startTime = Time.time;
    }

    void Update()
    {
        float t = Time.time - startTime;
        float x = amplitude * Mathf.Cos(omega * t + phase);

        transform.position = equilibrium + Vector3.right * x;
    }
}
```

Drehmoment

Drehmoment Rotatorisches Äquivalent zur Kraft:

$$\vec{\tau} = \vec{r} \times \vec{F}$$

Für Rotation um feste Achse: $\tau = rF\sin\theta$

Drehmoment in Unity

```
public class TorqueExample : MonoBehaviour
{
    private Rigidbody rb;
    public float torqueStrength = 100f;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    void Update()
    {
        if (Input.GetKey(KeyCode.Q))
        {
            // Drehmoment um Y-Achse anwenden
            rb.AddTorque(Vector3.up * torqueStrength);
        }

        // Rotationsfeder zur aufrechten Position
        Vector3 restoreTorque = -torqueStrength * transform.eulerAngles.x
                                * Vector3.right;
        rb.AddTorque(restoreTorque);
    }
}
```

Problemlösungsstrategie für Kräfte

Schritt 1: System identifizieren

- Untersuchungsobjekt(e) definieren
- Koordinatensystem wählen
- Zeitintervall bestimmen

Schritt 2: Freikörperdiagramm zeichnen

- Alle auf das Objekt wirkenden Kräfte zeigen
- Kraftvektoren mit Symbolen beschriften
- Keine Kräfte einbeziehen, die das Objekt auf andere ausübt

Schritt 3: Newton'sches Bewegungsgesetz anwenden

- $\sum \vec{F} = m\vec{a}$ in Komponentenform schreiben
- $\sum F_x = ma_x$, $\sum F_y = ma_y$, $\sum F_z = ma_z$
- Unbekannte Größen lösen

Schritt 4: In Unity implementieren

- `Rigidbody.AddForce()` für jede Kraft verwenden
- Passenden Force-Modus berücksichtigen
- `FixedUpdate()` für Physikberechnungen nutzen

Klotz auf schiefer Ebene Ein Klotz der Masse $m = 2kg$ gleitet eine reibungsfreie schiefe Ebene mit Winkel $\theta = 30$ř hinunter. Berechne die Beschleunigung und implementiere in Unity.

Freikörperdiagramm: Gewichtskraft mg nach unten, Normalkraft N senkrecht zur Oberfläche.
Komponentenanalyse:

- Entlang der Ebene: $mg\sin\theta = ma$
- Senkrecht dazu: $N - mg\cos\theta = 0$

Lösung: $a = g\sin\theta = 9.81 \times \sin(30$ř$) = 4.905\,m/s^2$
Unity Implementation:

```
Vector3 inclineForce = mass * Physics.gravity.magnitude *
                       Mathf.Sin(30f * Mathf.Deg2Rad) *
                       inclineDirection;
rb.AddForce(inclineForce);
```

Kräfte

Grundlegende Wechselwirkungen

Vier fundamentale Kräfte In der Physik gibt es vier fundamentale Wechselwirkungen:
1. **Gravitation**: Anziehung zwischen Massen
2. **Elektromagnetische Kraft**: Kräfte zwischen elektrischen Ladungen
3. **Starke Kernkraft**: hält den Atomkern zusammen
4. **Schwache Kernkraft**: verantwortlich für radioaktiven Beta-Zerfall

Für die makroskopische Mechanik sind hauptsächlich Gravitation und elektromagnetische Kräfte relevant.

Kraft Eine Kraft ist ein Einfluss, der den Bewegungszustand eines Körpers ändert.

$$\vec{F} = \frac{d\vec{p}}{dt} \tag{19}$$

Einheit: Newton (N) $= 1\ \frac{\text{kg}\cdot\text{m}}{\text{s}^2}$

Gravitationskraft

Newton'sches Gravitationsgesetz Anziehungskraft zwischen zwei Massen m_1 und m_2 im Abstand R:

$$F_G = G \cdot \frac{m_1 \cdot m_2}{R^2} \tag{20}$$

Gravitationskonstante: $G = 6.67 \cdot 10^{-11}\ \frac{\text{m}^3}{\text{kg}\cdot\text{s}^2}$

Erdbeschleunigung Gewichtskraft eines Körpers der Masse m auf der Erdoberfläche:

$$F_G = m \cdot g \tag{21}$$

Erdbeschleunigung:

$$g = G \cdot \frac{M_{\text{Erde}}}{R_{\text{Erde}}^2} \approx 9.81\ \frac{\text{m}}{\text{s}^2} \tag{22}$$

Mit zunehmender Höhe h:

$$g(h) = G \cdot \frac{M_{\text{Erde}}}{(R_{\text{Erde}} + h)^2} \tag{23}$$

Federkraft

Hooke'sches Gesetz Für eine lineare Feder:

$$\vec{F} = -k \cdot \vec{x} \tag{24}$$

- k: Federkonstante (N/m)
- $\vec{x}$: Auslenkung aus der Ruhelage
- Negatives Vorzeichen: Kraft wirkt der Auslenkung entgegen

Spannenergie einer Feder Potentielle Energie einer gespannten Feder:

$$E_{\text{spann}} = \frac{1}{2} \cdot k \cdot x^2 \tag{25}$$

Harmonischer Oszillator System mit rücktreibender Kraft proportional zur Auslenkung.
Bewegungsgleichung:

$$\frac{d^2x}{dt^2} = -\frac{k}{m} \cdot x \tag{26}$$

Lösung:

$$x(t) = x_0 \cdot \cos(\omega t) \quad \text{mit } \omega = \sqrt{\frac{k}{m}} \tag{27}$$

Schwingungsdauer: $T = \frac{2\pi}{\omega}$

Reibungskräfte

Arten der Reibung **Äußere Reibung** (Kontaktflächen von Festkörpern):
- Haftreibung
- Gleitreibung
- Rollreibung, Wälzreibung, Bohrreibung, Seilreibung

Innere Reibung: zwischen benachbarten Teilchen bei Verformungen

Trockene Reibung (Coulomb-Reibung)

$$\vec{F}_R = \mu \cdot \vec{F}_N \tag{28}$$

- μ: Reibungskoeffizient (dimensionslos)
- $\vec{F}_N$: Normalkraft
- Richtung: entgegengesetzt zur Bewegungsrichtung

Unterscheidung:
- Haftreibung: $\vec{F}_{\text{Haft}} \leq \mu_{\text{Haft}} \cdot \vec{F}_N$
- Gleitreibung: $\vec{F}_{\text{Gleit}} = \mu_{\text{Gleit}} \cdot \vec{F}_N$
- Regel: $\mu_{\text{Haft}} > \mu_{\text{Gleit}}$

Viskose Reibung **Laminare Strömung (Stokes'sche Reibung für Kugel):**

$$\vec{F}_R = -6 \cdot \pi \cdot \eta \cdot r \cdot v \cdot \vec{e}_v \tag{29}$$

(η: Viskosität des Mediums)
Turbulente Strömung:

$$\vec{F}_R = -\frac{1}{2} \cdot \rho \cdot A \cdot c_w \cdot \vec{v}^2 \cdot \vec{e}_v \tag{30}$$

(ρ: Dichte, A: Stirnfläche, c_w: Widerstandsbeiwert)

Unity Implementation

Trockene Reibung in Unity

```
void FixedUpdate() {
    // Normale Kraft berechnen (bei horizontaler Flaeche)
    float normalForce = rigidbody.mass * 9.81f;

    // Reibungskraft berechnen
    float frictionForce = frictionCoefficient * normalForce;

    // Richtung der Geschwindigkeit bestimmen
    Vector3 velocityDirection = rigidbody.velocity.normalized;

    // Reibungskraft nur anwenden, wenn Objekt sich bewegt
    if (rigidbody.velocity.magnitude > 0.01f) {
        Vector3 frictionVector = -velocityDirection * frictionForce;
        rigidbody.AddForce(frictionVector);
    }
}
```

Luftwiderstand in Unity

```
void FixedUpdate() {
    // Parameter fuer Luftwiderstand
    float airDensity = 1.2f;       // kg/m^3
    float dragCoefficient = 0.5f; // dimensionslos
    float frontalArea = 1.0f;     // m^2

    // Aktuelle Geschwindigkeit
    Vector3 velocity = rigidbody.velocity;
    float velocityMagnitude = velocity.magnitude;

    // Luftwiderstandskraft berechnen
    float dragForceMagnitude = 0.5f * airDensity * frontalArea *
                               dragCoefficient * velocityMagnitude * velocityMagnitude;

    // Richtung entgegen der Bewegung
    Vector3 dragForce = -velocity.normalized * dragForceMagnitude;

    rigidbody.AddForce(dragForce);
}
```

Trägheitskräfte

Beschleunigte Bezugssysteme In beschleunigten Bezugssystemen treten Trägheitskräfte (Scheinkräfte) auf:

Translatorische Trägheitskraft:

$$\vec{F}_{\text{Trägheit}} = -m \cdot \vec{a}_{\text{System}} \tag{31}$$

Zentrifugalkraft bei Rotation:

$$\vec{F}_{\text{Zentrifugal}} = m \cdot \omega^2 \cdot r \cdot \vec{e}_r \tag{32}$$

Corioliskraft bei Bewegung in rotierendem System:

$$\vec{F}_{\text{Coriolis}} = 2 \cdot m \cdot \vec{v} \times \vec{\omega} \tag{33}$$

Trägheitskräfte im Alltag

- Im bremsenden Zug: nach vorne gedrückt fühlen
- In der Kurve: nach außen gedrückt (Zentrifugalkraft)
- Corioliskraft: Ablenkung von Winden und Meeresströmungen
  - Nordhalbkugel: Ablenkung nach rechts
  - Südhalbkugel: Ablenkung nach links

Trägheitskräfte sind keine ächten"Kräfte im Sinne von Wechselwirkungen zwischen Körpern, sondern entstehen durch die Wahl des Bezugssystems. In einem Inertialsystem existieren sie nicht.

Impuls und Stoßgesetze

Impuls

Impuls Der Impuls $\vec{p}$ eines Körpers ist das Produkt aus seiner Masse und seiner Geschwindigkeit:

$$\vec{p} = m \cdot \vec{v} \tag{34}$$

Einheit: kg · m/s (vektorielle Größe)

Impulserhaltung In einem abgeschlossenen System ohne äußere Kräfte bleibt der Gesamtimpuls konstant:

$$\sum \vec{p}_{\text{vorher}} = \sum \vec{p}_{\text{nachher}} \tag{35}$$

Für zwei Körper:

$$m_1\vec{v}_{1,\text{vorher}} + m_2\vec{v}_{2,\text{vorher}} = m_1\vec{v}_{1,\text{nachher}} + m_2\vec{v}_{2,\text{nachher}} \tag{36}$$

Wichtig: Impulserhaltung gilt auch bei dissipativen Vorgängen (z.B. inelastische Stöße).

Kraftstoß Der Kraftstoß $\vec{I}$ ist das Zeitintegral der Kraft:

$$\vec{I} = \int_{t_1}^{t_2} \vec{F}(t)\,dt = \Delta\vec{p} \tag{37}$$

Für konstante Kraft:

$$\vec{I} = \vec{F} \cdot \Delta t = \vec{p}_{\text{nachher}} - \vec{p}_{\text{vorher}} \tag{38}$$

Stöße

Elastischer Stoß Sowohl Gesamtimpuls als auch Gesamtenergie bleiben erhalten:
Impulserhaltung:

$$m_1 v_{1,\text{vorher}} + m_2 v_{2,\text{vorher}} = m_1 v_{1,\text{nachher}} + m_2 v_{2,\text{nachher}} \tag{39}$$

Energieerhaltung:

$$\frac{1}{2}m_1 v_{1,\text{vorher}}^2 + \frac{1}{2}m_2 v_{2,\text{vorher}}^2 = \frac{1}{2}m_1 v_{1,\text{nachher}}^2 + \frac{1}{2}m_2 v_{2,\text{nachher}}^2 \tag{40}$$

Inelastischer Stoß Nur der Gesamtimpuls bleibt erhalten, mechanische Energie wird teilweise in andere Formen umgewandelt.
Vollständig inelastischer Stoß: Körper "kleben"nach dem Stoß zusammen.

$$v_{\text{nachher}} = \frac{m_1 v_{1,\text{vorher}} + m_2 v_{2,\text{vorher}}}{m_1 + m_2} \tag{41}$$

Berechnung von Stößen in 1D

Elastischer Stoß

Methode 1: Schwerpunktgeschwindigkeit

1. Schwerpunktgeschwindigkeit:

$$v_{SP} = \frac{m_1 v_1 + m_2 v_2}{m_1 + m_2} \tag{42}$$

2. Geschwindigkeiten nach dem Stoß (Spiegelung):

$$u_1 = 2v_{SP} - v_1 \tag{43}$$
$$u_2 = 2v_{SP} - v_2 \tag{44}$$

Methode 2: Direkte Formeln

$$u_1 = \frac{(m_1 - m_2)v_1 + 2m_2 v_2}{m_1 + m_2} \tag{45}$$
$$u_2 = \frac{2m_1 v_1 + (m_2 - m_1)v_2}{m_1 + m_2} \tag{46}$$

Vollständig inelastischer Stoß

$$u_1 = u_2 = \frac{m_1 v_1 + m_2 v_2}{m_1 + m_2} \tag{47}$$

Elastischer Stoß mit ungleichen Massen **Leichte Kugel trifft schwere ruhende Kugel** ($m_1 \ll m_2$, $v_2 = 0$):

$$u_1 \approx -v_1 \quad \text{(Richtungsumkehr)} \tag{48}$$
$$u_2 \approx 0 \quad \text{(schwerer Körper bleibt fast in Ruhe)} \tag{49}$$

Schwere Kugel trifft leichte ruhende Kugel ($m_1 \gg m_2$, $v_2 = 0$):

$$u_1 \approx v_1 \quad \text{(fast unverändert)} \tag{50}$$
$$u_2 \approx 2v_1 \quad \text{(doppelte Geschwindigkeit)} \tag{51}$$

Impuls- vs. Energieerhaltung

- **Impulserhaltung**: Gilt immer ohne äußere Kräfte (Grundprinzip)
- **Energieerhaltung**: Nur für konservative Vorgänge
- **Elastische Stöße**: Beide Erhaltungssätze
- **Inelastische Stöße**: Nur Impulserhaltung
- **Impuls**: Vektorgleichungen (Richtungen wichtig)
- **Energie**: Skalargleichung (nur Beträge)

Anwendungen

Ballistisches Pendel Gerät zur Messung von Projektilgeschwindigkeiten durch vollständig inelastischen Stoß.
Gegeben: Projektilmasse m_K, Pendelmasse m_P, Auslenkung x, Pendellänge L
Projektilgeschwindigkeit:

$$v_K = \frac{m_P + m_K}{m_K} \cdot \sqrt{2g \cdot (L - \sqrt{L^2 - x^2})} \tag{52}$$

Zweistufiger Prozess:

1. Inelastischer Stoß: Impulserhaltung
2. Pendelschwingung: Energieerhaltung

Raketenantrieb Basiert auf Rückstoßprinzip und Impulserhaltung.
Ziolkowski-Gleichung (maximale Geschwindigkeit):

$$v_{\text{max}} = v_{\text{rel}} \cdot \ln \frac{m_0}{m_{\text{leer}}} \tag{53}$$

- v_{rel}: Austrittsgeschwindigkeit des Treibstoffs
- m_0: Anfangsmasse (Rakete + Treibstoff)
- m_{leer}: Leermasse der Rakete

Unity Implementation

Stoßsimulation in Unity

```
void OnCollisionEnter(Collision collision) {
    Rigidbody otherRb = collision.rigidbody;
    if (otherRb == null) return;

    // Geschwindigkeiten vor dem Stoss
    Vector3 v1 = rigidbody.velocity;
    Vector3 v2 = otherRb.velocity;

    // Massen
    float m1 = rigidbody.mass;
    float m2 = otherRb.mass;

    // Normale des Stosses
    Vector3 normal = (otherRb.position - rigidbody.position).normalized;

    // Projektion der Geschwindigkeiten auf die Stossnormale
    float v1n = Vector3.Dot(v1, normal);
    float v2n = Vector3.Dot(v2, normal);

    // Nur berechnen, wenn Objekte sich annaehern
    if (v1n - v2n > 0) {
        // Geschwindigkeiten nach elastischem Stoss
        float u1n = ((m1 - m2) * v1n + 2 * m2 * v2n) / (m1 + m2);
        float u2n = ((m2 - m1) * v2n + 2 * m1 * v1n) / (m1 + m2);

        // Tangentiale Komponenten bleiben unveraendert
        Vector3 v1Tangential = v1 - v1n * normal;
        Vector3 v2Tangential = v2 - v2n * normal;

        // Neue Geschwindigkeiten
        Vector3 newV1 = v1Tangential + u1n * normal;
        Vector3 newV2 = v2Tangential + u2n * normal;

        rigidbody.velocity = newV1;
        otherRb.velocity = newV2;
    }
}
```

Wichtige Erkenntnisse

- Impulserhaltung ist universell gültig
- Kollisionsberechnung erfolgt komponentenweise
- Tangentiale Geschwindigkeiten bleiben bei Stößen unverändert
- Energieverluste nur bei inelastischen Stößen

Arbeit und Energie

Arbeit

Arbeit Die physikalische Arbeit W ist das Skalarprodukt aus Kraft und Weg:

$$W = \int_{\vec{r}_0}^{\vec{r}_1} \vec{F} \cdot d\vec{r} \tag{54}$$

Einheit: Joule (J) $= 1\,\mathrm{N \cdot m} = 1\,\frac{\mathrm{kg \cdot m^2}}{\mathrm{s^2}}$
Für konstante Kraft: $W = F \cdot \Delta x \cdot \cos(\alpha)$

Graphische Darstellung Im Kraft-Weg-Diagramm entspricht die Arbeit der Fläche unter der Kurve:

$$W = \int_{x_0}^{x_1} F_x(x)\,dx \tag{55}$$

Beispiele:
- Konstante Kraft: $W = F \cdot \Delta x$ (Rechteck)
- Linear ansteigende Kraft: $W = \frac{1}{2} F_{max} \cdot \Delta x$ (Dreieck)

Arten physikalischer Arbeit

Hubarbeit:
$W = m \cdot g \cdot h$

Beschleunigungsarbeit:
$W = \frac{1}{2} m (v_1^2 - v_0^2)$

Deformationsarbeit (Feder):
$W = \frac{1}{2} k (x_1^2 - x_0^2)$

Reibungsarbeit:
$W = \mu \cdot F_N \cdot \Delta x$

Energie

Energie Energie ist die Fähigkeit eines Systems, Arbeit zu verrichten.

$$W = \Delta E = E_{nachher} - E_{vorher} \tag{56}$$

Unterschied zu Arbeit:
- Arbeit: Prozessgröße (beschreibt einen Vorgang)
- Energie: Zustandsgröße (charakterisiert einen Zustand)

Energieformen in der Mechanik

Kinetische Energie:
$E_{kin} = \frac{1}{2} m v^2$

Potentielle Energie im Schwerefeld:
$E_{pot} = mgh$

Spannenergie einer Feder:
$E_{spann} = \frac{1}{2} k x^2$

Rotationsenergie:
$E_{rot} = \frac{1}{2} J \omega^2$

Energieerhaltung In einem abgeschlossenen System ohne nicht-konservative Kräfte bleibt die Gesamtenergie konstant:

$$E_{ges} = E_{kin} + E_{pot} + E_{spann} + E_{rot} = \text{const.} \tag{57}$$

Konservative vs. nicht-konservative Kräfte:
- **Konservativ**: Arbeit ist wegunabhängig (Gravitation, Federkraft)
- **Nicht-konservativ**: Arbeit hängt vom Weg ab (Reibung, Luftwiderstand)

Energiewandlungen beim Pendel Pendel der Masse m und Länge l zeigt Umwandlung zwischen potentieller und kinetischer Energie:
Am höchsten Punkt:

$$E_{pot,max} = mgl(1 - \cos\alpha) \tag{58}$$

Am tiefsten Punkt:

$$E_{kin,max} = \frac{1}{2} m v_{max}^2 = mgl(1 - \cos\alpha) \tag{59}$$

Gesamtenergie bleibt konstant: $E_{ges} = E_{pot} + E_{kin}$

Leistung und Wirkungsgrad

Leistung Die Leistung P ist die pro Zeiteinheit verrichtete Arbeit:

$$P = \frac{dW}{dt} = \vec{F} \cdot \vec{v} \tag{60}$$

Einheit: Watt (W) $= 1\,\frac{\mathrm{J}}{\mathrm{s}} = 1\,\frac{\mathrm{kg \cdot m^2}}{\mathrm{s^3}}$
Für Rotationsbewegung: $P = \tau\omega$

Wirkungsgrad Verhältnis von Nutzenergie zu zugeführter Energie:

$$\eta = \frac{E_{Nutz}}{E_{zugeführt}} = \frac{P_{Nutz}}{P_{zugeführt}} \tag{61}$$

Dimensionslose Zahl zwischen 0 und 1 (oder 0-100%).

Impuls und Drehimpuls

Linearer Impuls

$$\vec{p} = m\vec{v} \tag{62}$$

Newton'sches Gesetz: $\vec{F} = \frac{d\vec{p}}{dt}$

Kraftstoß Änderung des Impulses durch Kraft über Zeit:

$$\vec{J} = \int_{t_1}^{t_2} \vec{F}\,dt = \Delta\vec{p} = m\vec{v}_f - m\vec{v}_i \tag{63}$$

Für konstante Kraft: $\vec{J} = \vec{F}\Delta t$

Impulserhaltung Ohne äußere Kräfte bleibt der Gesamtimpuls konstant:

$$\sum \vec{p}_i = \sum \vec{p}_f \tag{64}$$

Für Zwei-Körper-System:

$$m_1\vec{v}_{1i} + m_2\vec{v}_{2i} = m_1\vec{v}_{1f} + m_2\vec{v}_{2f} \tag{65}$$

Drehimpuls **Für Punktteilchen:**

$$\vec{L} = \vec{r} \times \vec{p} = m\vec{r} \times \vec{v} \tag{66}$$

Für starren Körper:

$$\vec{L} = J\vec{\omega} \tag{67}$$

Drehimpulserhaltung Ohne äußere Drehmomente:

$$\frac{d\vec{L}}{dt} = \vec{\tau}_{ext} = 0 \Rightarrow \vec{L} = \text{const.} \tag{68}$$

Anwendungen: Eiskunstläufer, Planetenbewegung, Kreisel

Kollisionen

Kollisionstypen **Elastisch:** Impuls und kinetische Energie erhalten
Inelastisch: Nur Impuls erhalten
Vollständig inelastisch: Objekte kleben nach Kollision zusammen

Elastische Kollision (1D) Für zwei Objekte mit Massen m_1, m_2 und Anfangsgeschwindigkeiten v_{1i}, v_{2i}:

$$v_{1f} = \frac{(m_1 - m_2)v_{1i} + 2m_2 v_{2i}}{m_1 + m_2} \tag{69}$$

$$v_{2f} = \frac{(m_2 - m_1)v_{2i} + 2m_1 v_{1i}}{m_1 + m_2} \tag{70}$$

Unity Implementation

Energieberechnung in Unity

```
public class EnergyMonitor : MonoBehaviour
{
    private Rigidbody rb;
    public float springConstant = 100f;
    public Vector3 equilibriumPosition = Vector3.zero;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    void Update()
    {
        // Kinetische Energie berechnen
        float kineticEnergy = 0.5f * rb.mass * rb.velocity.sqrMagnitude;

        // Potentielle Energie (Gravitation)
        float gravitationalPE = rb.mass * Mathf.Abs(Physics.gravity.y) *
                                transform.position.y;

        // Elastische potentielle Energie
        Vector3 displacement = transform.position - equilibriumPosition;
        float elasticPE = 0.5f * springConstant * displacement.sqrMagnitude;

        // Gesamtenergie
        float totalEnergy = kineticEnergy + gravitationalPE + elasticPE;

        Debug.Log($"KE: {kineticEnergy:F2}, PE: {gravitationalPE + elasticPE:F2}, " +
                  $"Total: {totalEnergy:F2}");
    }
}
```

Kollisionsbehandlung in Unity

```
public class CollisionHandler : MonoBehaviour
{
    private Rigidbody rb;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    void OnCollisionEnter(Collision collision)
    {
        Rigidbody otherRb = collision.rigidbody;
        if (otherRb == null) return;

        // Massen und Geschwindigkeiten vor Kollision
        float m1 = rb.mass;
        float m2 = otherRb.mass;
        Vector3 v1i = rb.velocity;
        Vector3 v2i = otherRb.velocity;

        // Geschwindigkeiten nach elastischer Kollision
        Vector3 v1f = ((m1 - m2) * v1i + 2 * m2 * v2i) / (m1 + m2);
        Vector3 v2f = ((m2 - m1) * v2i + 2 * m1 * v1i) / (m1 + m2);

        // Neue Geschwindigkeiten anwenden
        rb.velocity = v1f;
        otherRb.velocity = v2f;
    }
}
```

Drehimpuls in Unity

```
// Drehimpuls eines Koerpers bezueglich eines Pivot-Punkts berechnen
Vector3 AngularMomentum(Rigidbody rb, Vector3 pivotPoint)
{
    // Vektor vom Pivot zum Schwerpunkt
    Vector3 rPQ = rb.transform.position - pivotPoint;

    // Linearer Impuls des Koerpers
    Vector3 p = rb.velocity * rb.mass;

    // Drehimpuls berechnen
    Vector3 angularMomentum = Vector3.Cross(rPQ, p);
    return angularMomentum;
}
```

Energie-Impuls-Problemlösung

Schritt 1: System und Randbedingungen identifizieren

- Systemgrenzen definieren
- Konservative und nicht-konservative Kräfte identifizieren
- Erhaltungsgesetze prüfen

Schritt 2: Passenden Erhaltungssatz wählen

- Energieerhaltung für Höhen-, Federprobleme
- Impulserhaltung für Kollisionsprobleme
- Beide für komplexe mehrstufige Probleme

Schritt 3: Gleichungen aufstellen

- Anfangs- und Endzustände beschreiben
- Erhaltungsprinzipien anwenden
- Zusätzliche Randbedingungen einbeziehen

Schritt 4: Lösen und verifizieren

- Algebraisch lösen vor Zahleneinsetzung
- Einheiten und physikalische Plausibilität prüfen
- Erhaltungsgesetze verifizieren

Kollisionsanalyse Zwei Autos: Auto A (1000 kg) mit 20 m/s, Auto B (1500 kg) mit -15 m/s. Nach Kollision bewegt sich Auto A mit 5 m/s. Berechne Auto B's Endgeschwindigkeit und Energieverlust.

Gegeben: $m_A = 1000kg$, $m_B = 1500kg$, $v_{Ai} = 20m/s$, $v_{Bi} = -15m/s$, $v_{Af} = 5m/s$
Impulserhaltung: $1000(20) + 1500(-15) = 1000(5) + 1500v_{Bf}$
$20000 - 22500 = 5000 + 1500v_{Bf}$
$v_{Bf} = -5m/s$
Energieanalyse: $KE_i = \frac{1}{2}(1000)(20^2) + \frac{1}{2}(1500)(15^2) = 368750J$
$KE_f = \frac{1}{2}(1000)(5^2) + \frac{1}{2}(1500)(5^2) = 31250J$
$\Delta E = 368750 - 31250 = 337500J$ verloren

Projektilbewegung mit Energie Ein Projektil wird mit Winkel $\theta = 45$ř und Anfangsgeschwindigkeit $v_0 = 20m/s$ abgeschossen. Maximale Höhe mit Energieerhaltung berechnen.

Energiemethode: Start: $E_i = \frac{1}{2}mv_0^2$ (Boden als Referenz)
Max. Höhe: $E_f = mgh_{max} + \frac{1}{2}mv_x^2$
Da $v_x = v_0 \cos\theta$ konstant bleibt:
$\frac{1}{2}mv_0^2 = mgh_{max} + \frac{1}{2}m(v_0\cos\theta)^2$
$h_{max} = \frac{v_0^2 \sin^2\theta}{2g} = \frac{(20)^2 \sin^2(45\text{ř})}{2(9.81)} = 10.2m$

Rotation

Kinematik der Rotation

Rotationsbewegung Eine Rotationsbewegung ist die Drehung eines Körpers um eine bestimmte Achse. Die Beschreibung erfolgt analog zur Translation mit rotatorischen Größen:

- Drehwinkel φ statt Ort $\vec{r}$
- Winkelgeschwindigkeit ω statt Geschwindigkeit $\vec{v}$
- Winkelbeschleunigung α statt Beschleunigung $\vec{a}$

Freiheitsgrade Ein starrer Körper im dreidimensionalen Raum hat sechs Freiheitsgrade:

- Drei translatorische Freiheitsgrade (Bewegung in x-, y- und z-Richtung)
- Drei rotatorische Freiheitsgrade (Drehung um die x-, y- und z-Achse)

Diese Bewegungen sind für einen freien Körper unabhängig voneinander.

Rotatorische Kinematik Analog zur Translation gelten folgende Beziehungen:

$$\omega = \frac{d\varphi}{dt} \tag{71}$$

$$\alpha = \frac{d\omega}{dt} = \frac{d^2\varphi}{dt^2} \tag{72}$$

$$\varphi = \int \omega \, dt \tag{73}$$

$$\omega = \int \alpha \, dt \tag{74}$$

Für konstante Winkelbeschleunigung:

$$\omega(t) = \omega_0 + \alpha t \tag{75}$$

$$\varphi(t) = \varphi_0 + \omega_0 t + \frac{1}{2}\alpha t^2 \tag{76}$$

$$\omega^2 = \omega_0^2 + 2\alpha(\varphi - \varphi_0) \tag{77}$$

Die Maßeinheiten sind:

- Winkel φ: Radian (rad), dimensionslos
- Winkelgeschwindigkeit ω: rad/s
- Winkelbeschleunigung α: rad/s^2

Winkel in Radian Der Winkel in Radian ist definiert als das Verhältnis von Kreisbogen s zu Radius r:

$$\varphi = \frac{s}{r} \tag{78}$$

Umrechnung zwischen Grad und Radian:

$$\varphi_{rad} = \varphi_{deg} \cdot \frac{\pi}{180} \tag{79}$$

Wichtige Werte:

- Vollwinkel: $360° = 2\pi$ rad
- Rechter Winkel: $90° = \frac{\pi}{2}$ rad

Zusammenhang zwischen linearer und Rotationsbewegung Bei einer Kreisbewegung mit Radius r gelten folgende Beziehungen:

$$s = r\varphi \quad \text{(Bogenlänge)} \tag{80}$$

$$v = r\omega \quad \text{(Bahngeschwindigkeit)} \tag{81}$$

$$a_t = r\alpha \quad \text{(Tangentialbeschleunigung)} \tag{82}$$

$$a_c = \frac{v^2}{r} = r\omega^2 \quad \text{(Zentripetalbeschleunigung)} \tag{83}$$

Dabei ist v die Geschwindigkeit eines Punktes auf dem Umfang (tangential zur Kreisbahn) und ω die Winkelgeschwindigkeit der Rotation.

Schwerpunkt und Trägheitsmoment

Schwerpunkt Der Schwerpunkt (Massenmittelpunkt) eines Körpers ist der gewichtete Mittelwert der Positionen aller Massenelemente:

$$\vec{r}_S = \frac{1}{\sum_{i=1}^{n} m_i} \cdot \sum_{i=1}^{n} m_i \cdot \vec{r}_i \tag{84}$$

Für kontinuierliche Massenverteilungen:

$$\vec{r}_S = \frac{1}{M} \iiint_K \vec{r}\,\rho(\vec{r})\, dV \tag{85}$$

wobei $\rho(\vec{r})$ die Dichteverteilung und M die Gesamtmasse ist.

Schwerpunktsatz Der Schwerpunktsatz besagt, dass eine an einem beliebigen Punkt eines starren Körpers angreifende Kraft

- eine Translation des Schwerpunktes bewirkt, so als ob die Kraft direkt am Schwerpunkt angreifen würde, und
- ein Drehmoment um den Schwerpunkt erzeugt, wenn die Kraftwirkungslinie nicht durch den Schwerpunkt verläuft.

Dies erlaubt die Zerlegung der Bewegung in eine Translation des Schwerpunktes und eine Rotation um den Schwerpunkt.

Trägheitsmoment Das Trägheitsmoment I ist ein Maß für den Widerstand eines Körpers gegenüber Rotationsbeschleunigungen:

$$I = \sum_{i=1}^{n} m_i \cdot r_i^2 \tag{86}$$

Für kontinuierliche Massenverteilungen:

$$I = \iiint_K r^2\, \rho(\vec{r})\, dV = \int r^2\, dm \tag{87}$$

Dabei ist r der senkrechte Abstand des Massenelements von der Drehachse.

Trägheitsmomente wichtiger Körper

Punktmasse:
$I = mr^2$

Dünner Stab (Länge L, Achse durch Mitte):
$I = \frac{1}{12}mL^2$

Dünner Stab (Achse am Ende):
$I = \frac{1}{3}mL^2$

Vollzylinder (Radius R, Symmetrieachse):
$I = \frac{1}{2}mR^2$

Hohlzylinder (Symmetrieachse):
$I = mR^2$

Vollkugel (Radius R, Achse durch Mittelpunkt):
$I = \frac{2}{5}mR^2$

Steiner'scher Satz (Parallelachsentheorem) Für eine Achse parallel zur Schwerpunktachse im Abstand d:

$$I = I_S + md^2 \tag{88}$$

Dabei ist:
- I_S das Trägheitsmoment bezüglich einer Achse durch den Schwerpunkt
- m die Gesamtmasse des Körpers
- d der Abstand zwischen den parallelen Achsen

Dynamik der Rotation

Drehmoment Das Drehmoment $\vec{\tau}$ ist die Ursache einer Rotationsbeschleunigung:

$$\vec{\tau} = \vec{r} \times \vec{F} \tag{89}$$

Der Betrag des Drehmoments ist:

$$\tau = rF \sin\phi = Fd \tag{90}$$

wobei ϕ der Winkel zwischen Hebelarm und Kraft ist und d der senkrechte Abstand der Kraftwirkungslinie von der Drehachse.

Newton'sches Gesetz für Rotationen Analog zum zweiten Newton'schen Gesetz für Translation:

$$\sum \vec{\tau} = I\vec{\alpha} \tag{91}$$

Diese Gleichung verknüpft das resultierende Drehmoment mit der Winkelbeschleunigung.

Drehimpuls Der Drehimpuls $\vec{L}$ ist das rotatorische Analogon zum linearen Impuls:

$$\vec{L} = I\vec{\omega} \tag{92}$$

Für ein Punktteilchen mit Impuls $\vec{p}$ im Abstand $\vec{r}$ von der Drehachse:

$$\vec{L} = \vec{r} \times \vec{p} = m\vec{r} \times \vec{v} \tag{93}$$

Das Drehmoment ändert den Drehimpuls gemäß:

$$\vec{\tau} = \frac{d\vec{L}}{dt} \tag{94}$$

Drehimpulserhaltung In Abwesenheit äußerer Drehmomente bleibt der Drehimpuls konstant:

$$\frac{d\vec{L}}{dt} = \vec{\tau}_{ext} = 0 \Rightarrow \vec{L} = \text{const.} \tag{95}$$

Anwendungen:
- Eiskunstläufer, der die Arme anzieht, um schneller zu rotieren
- Stabilität von Fahrrädern aufgrund der rotierenden Räder
- Präzession eines Kreisels
- Planetenbewegung

Rotationsenergie Die kinetische Energie der Rotation:

$$E_{rot} = \frac{1}{2}I\omega^2 \tag{96}$$

Für kombinierte Translation und Rotation:

$$E_{ges} = \frac{1}{2}mv_{cm}^2 + \frac{1}{2}I_{cm}\omega^2 \tag{97}$$

Rollbedingung ohne Schlupf: $v_{cm} = R\omega$
Dann: $E_{ges} = \frac{1}{2}mv^2(1 + \frac{I}{mR^2})$

Arbeit und Leistung bei Rotation **Rotationsarbeit:**

$$W = \int \tau \, d\varphi \tag{98}$$

Für konstantes Drehmoment: $W = \tau\Delta\varphi$
Rotationsleistung:

$$P = \tau\omega \tag{99}$$

Arbeitssatz für Rotation:

$$W_{net} = \Delta E_{rot} = \frac{1}{2}I\omega_f^2 - \frac{1}{2}I\omega_i^2 \tag{100}$$

Orientierung im Raum und Quaternionen

Euler-Winkel Euler-Winkel beschreiben eine Rotation im dreidimensionalen Raum durch drei Winkel:
- Pitch (φ): Drehung um die x-Achse
- Yaw (θ): Drehung um die y-Achse
- Roll (ψ): Drehung um die z-Achse

Wichtig: Die Reihenfolge der Drehungen ist entscheidend!
- Intrinsische Drehungen: Achsen drehen sich mit dem Objekt
- Extrinsische Drehungen: Achsen bleiben fixiert

Quaternionen in Unity Unity verwendet Quaternionen zur Darstellung von Rotationen wegen ihrer Vorteile:
- Keine "Gimbal-LockProbleme
- Effizientere Interpolation zwischen Rotationen
- Numerisch stabiler

Quaternionen haben die Form $q = w + xi + yj + zk$, wobei $i^2 = j^2 = k^2 = ijk = -1$.
Rotation um Achse $\vec{n}$ mit Winkel α:

$$q = \cos\left(\frac{\alpha}{2}\right) + \sin\left(\frac{\alpha}{2}\right)(n_x i + n_y j + n_z k) \tag{101}$$

Standfestigkeit und Gleichgewicht

Standfestigkeit Ein Körper ist standfest, wenn die Wirkungslinie der Gewichtskraft durch die Unterstützungsfläche verläuft.

Gleichgewichtsarten:

- **Stabil**: Rückkehr zur Ausgangslage (Energie-Minimum)
- **Labil**: Entfernung von Ausgangslage (Energie-Maximum)
- **Indifferent**: Verharren in neuer Position (konstante Energie)

Kippkriterium Ein Körper kippt, wenn die Wirkungslinie der Gewichtskraft außerhalb der Unterstützungsfläche verläuft.
Für einen Quader (Breite b, Höhe h) auf geneigter Ebene (Winkel α):

$$\tan \alpha > \frac{b}{2h} \tag{102}$$

Unity Implementation

Rotationsbewegung in Unity

```
public class RotationalMotion : MonoBehaviour
{
    public float angularVelocity = 90f; // Grad pro Sekunde
    public float angularAcceleration = 10f; // Grad pro Sekunde^2

    private float currentAngularVel;

    void Start()
    {
        currentAngularVel = angularVelocity;
    }

    void Update()
    {
        // Winkelgeschwindigkeit aktualisieren
        currentAngularVel += angularAcceleration * Time.deltaTime;

        // Rotation anwenden
        transform.Rotate(Vector3.up, currentAngularVel * Time.deltaTime);

        // Alternative: Winkelgeschwindigkeit direkt am Rigidbody setzen
        // GetComponent<Rigidbody>().angularVelocity =
        //     Vector3.up * currentAngularVel * Mathf.Deg2Rad;
    }
}
```

Drehmoment berechnen und anwenden

```
public class TorqueCalculation : MonoBehaviour
{
    public Transform forceApplication; // Kraftangriffspunkt
    public Vector3 forceVector = Vector3.forward;
    public float torqueStrength = 100f;

    private Rigidbody rb;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    void Update()
    {
        // Hebelarm vom Rotationszentrum zum Kraftangriffspunkt
        Vector3 leverArm = forceApplication.position - transform.position;

        // Drehmoment als Kreuzprodukt berechnen
        Vector3 torque = Vector3.Cross(leverArm, forceVector);

        // Drehmoment anwenden
        rb.AddTorque(torque);

        // Direkte Drehmomentanwendung
        if (Input.GetKey(KeyCode.Q))
        {
            rb.AddTorque(Vector3.up * torqueStrength);
        }

        // Debug-Visualisierung
        Debug.DrawRay(transform.position, leverArm, Color.red);
        Debug.DrawRay(forceApplication.position, forceVector, Color.blue);
        Debug.DrawRay(transform.position, torque, Color.green);
    }
}
```

Quaternionen in Unity verwenden

```
public class QuaternionExample : MonoBehaviour
{
    void Start()
    {
        // Quaternion aus Euler-Winkeln erzeugen
        Quaternion rotation = Quaternion.Euler(30f, 45f, 0f);

        // Quaternion aus Achse und Winkel erzeugen
        Vector3 axis = Vector3.up;
        float angle = 45f;
        Quaternion fromAxisAngle = Quaternion.AngleAxis(angle, axis);

        // Rotationen kombinieren (zuerst rotA, dann rotB)
        Quaternion combinedRotation = rotation * fromAxisAngle;

        // Quaternion auf Vektor anwenden
        Vector3 rotatedVector = rotation * Vector3.forward;

        // Objekt rotieren
        transform.rotation = rotation;

        // Sphaerische Interpolation zwischen Rotationen
        Quaternion interpolated = Quaternion.Slerp(
            Quaternion.identity, rotation, 0.5f);
    }
}
```

Rotationsenergie-Monitor

```
public class RotationalEnergyMonitor : MonoBehaviour
{
    private Rigidbody rb;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    void Update()
    {
        // Rotationsenergie berechnen
        float angularSpeed = rb.angularVelocity.magnitude;

        // Traegheitsmoment approximieren (fuer Kugel)
        float radius = GetComponent<SphereCollider>().radius;
        float momentOfInertia = 0.4f * rb.mass * radius * radius;

        float rotationalKE = 0.5f * momentOfInertia * angularSpeed * angularSpeed;

        // Translationsenergie
        float translationalKE = 0.5f * rb.mass * rb.velocity.sqrMagnitude;

        // Gesamtenergie
        float totalKE = rotationalKE + translationalKE;

        Debug.Log($"Rot KE: {rotationalKE:F2}, Trans KE: {translationalKE:F2}, " +
                  $"Total: {totalKE:F2}");
    }
}
```

Rotationsprobleme lösen

Schritt 1: Rotationsachse identifizieren

- Feste Achse (Türscharnier) oder bewegliche Achse (rollender Ball)
- Koordinatensystem mit Achse senkrecht zur Bewegungsebene wählen

Schritt 2: Trägheitsmoment berechnen

- Standardformeln für einfache Formen verwenden
- Steiner'schen Satz anwenden wenn nötig
- Für zusammengesetzte Objekte einzelne Momente summieren

Schritt 3: Kräfte und Drehmomente analysieren

- Drehmoment berechnen: $\tau = rF\sin\phi$
- $\sum \tau = I\alpha$ für Dynamik anwenden
- Randbedingungen berücksichtigen (Rollen, feste Achse, etc.)

Schritt 4: Erhaltungsgesetze anwenden

- Energieerhaltung für Objekte auf schiefen Ebenen
- Drehimpulserhaltung für isolierte Systeme
- Kombinierte linear-rotatorische Analyse für komplexe Bewegungen

Kugel rollt schiefe Ebene hinunter

Eine Vollkugel (Radius R, Masse m) rollt ohne Schlupf eine schiefe Ebene mit Winkel θ hinunter. Berechne die Beschleunigung des Schwerpunkts.

Gegeben: $I = \frac{2}{5}mR^2$ für Vollkugel, Rollbedingung $a = R\alpha$
Kräfteanalyse:

- Entlang der Ebene: $mg\sin\theta - f = ma$
- Drehmoment um Schwerpunkt: $fR = I\alpha = \frac{2}{5}mR^2 \cdot \frac{a}{R}$

Aus Drehmomentgleichung: $f = \frac{2}{5}ma$
Einsetzen in Kraftgleichung: $mg\sin\theta - \frac{2}{5}ma = ma$
$mg\sin\theta = ma(1 + \frac{2}{5}) = \frac{7}{5}ma$
$a = \frac{5g\sin\theta}{7} = 0.714 \times g\sin\theta$
Ergebnis: Die Beschleunigung ist kleiner als beim Rutschen ($g\sin\theta$), da Energie in Rotation fließt.

Türsteuerung mit Drehmoment

Unity-Script für realistische Türsimulation mit Drehmoment und Dämpfung:

```
public class DoorController : MonoBehaviour
{
    public float maxTorque = 50f;
    public float dampingCoefficient = 2f;
    public float maxAngle = 90f;

    private Rigidbody rb;
    private float initialAngle;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
        initialAngle = transform.eulerAngles.y;
    }

    void FixedUpdate()
    {
        float currentAngle = transform.eulerAngles.y - initialAngle;
        if (currentAngle > 180f) currentAngle -= 360f;

        // Tuer oeffnen
        if (Input.GetKey(KeyCode.Space) && currentAngle < maxAngle)
        {
            rb.AddTorque(Vector3.up * maxTorque);
        }

        // Daempfung anwenden
        Vector3 dampingTorque = -dampingCoefficient * rb.angularVelocity;
        rb.AddTorque(dampingTorque);

        // Rotationsbegrenzung
        if (Mathf.Abs(currentAngle) > maxAngle)
        {
            rb.angularVelocity = Vector3.zero;
            Vector3 angles = transform.eulerAngles;
            angles.y = initialAngle + Mathf.Sign(currentAngle) * maxAngle;
            transform.eulerAngles = angles;
        }
    }
}
```

Maximaler Überhang gestapelter Objekte Ein Turm aus n identischen Bausteinen der Länge L kann maximal überhängen um:

$$\text{Überhang}_{\text{max}} = \frac{L}{2} \cdot \sum_{i=1}^{n} \frac{1}{i} = \frac{L}{2}(1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}) \tag{103}$$

Dies basiert auf der harmonischen Reihe und zeigt, dass theoretisch unbegrenzte Überhänge möglich sind (divergente Reihe).

Erweiterte Themen und Unity-Implementierung

Unity Physics System

Unity Physics Pipeline Unity's Physiksimulation läuft in diskreten Schritten:

- `FixedUpdate()`: Wird in festen Intervallen aufgerufen (Standard 50Hz)
- Physikberechnungen zwischen FixedUpdate-Aufrufen
- `Update()`: Wird einmal pro Frame aufgerufen (variable Rate)
- **Regel**: FixedUpdate für physikbezogenen Code verwenden

Zeitsteuerung:

- `Time.fixedDeltaTime`: Zeitschritt für Physik (Standard: 0.02s = 50Hz)
- `Time.deltaTime`: Zeit seit letztem Frame (variabel)
- `Time.timeScale`: Globaler Zeitfaktor (für Slow-Motion etc.)

Rigidbody-Komponenten-Eigenschaften **Grundlegende Eigenschaften:**

- **Mass**: Objektmasse in kg
- **Drag**: Linearer Dämpfungskoeffizient (Luftwiderstand)
- **Angular Drag**: Rotationsdämpfungskoeffizient
- **Use Gravity**: Reagiert auf globale Gravitation
- **Is Kinematic**: Physik-gesteuert vs. Skript-gesteuert

Erweiterte Eigenschaften:

- **Interpolate**: Glättung für visuelle Darstellung
- **Collision Detection**: Continuous, Discrete, etc.
- **Constraints**: Einschränkung von Position/Rotation
- **Center of Mass**: Schwerpunkt-Position

Unity Physics-Konfiguration

```
public class PhysicsConfiguration : MonoBehaviour
{
    void Start()
    {
        // Globale Physik-Einstellungen
        Physics.gravity = new Vector3(0, -9.81f, 0);
        Time.fixedDeltaTime = 0.02f; // 50 Hz Physik-Update

        // Solver-Einstellungen fuer bessere Stabilitaet
        Physics.defaultSolverIterations = 6;
        Physics.defaultSolverVelocityIterations = 1;

        // Rigidbody-Konfiguration
        Rigidbody rb = GetComponent<Rigidbody>();
        rb.mass = 1.0f;
        rb.drag = 0.1f;                    // Lineare Daempfung
        rb.angularDrag = 0.05f;            // Winkel-Daempfung
        rb.useGravity = true;
        rb.isKinematic = false;

        // Kollisionserkennung-Modus
        rb.collisionDetectionMode = CollisionDetectionMode.Continuous;

        // Interpolation fuer glatte Bewegung
        rb.interpolation = RigidbodyInterpolation.Interpolate;

        // Schwerpunkt manuell setzen (falls noetig)
        rb.centerOfMass = new Vector3(0, -0.5f, 0);
    }
}
```

Kraftanwendungsmethoden

Unity Kraftanwendung **Kraftmethoden:**

- **AddForce()**: Kontinuierliche Kraftanwendung
- **AddForceAtPosition()**: Kraft an spezifischem Weltpunkt
- **AddTorque()**: Rotationskraft-Anwendung
- **AddExplosionForce()**: Radiale Kraft von Explosionszentrum
- **AddRelativeForce()**: Kraft im lokalen Koordinatensystem

Force-Modi:

- **Force**: Kontinuierliche Kraft (berücksichtigt Masse und Zeit)
- **Acceleration**: Beschleunigung (ignoriert Masse)
- **Impulse**: Impulsive Kraft (berücksichtigt Masse)
- **VelocityChange**: Geschwindigkeitsänderung (ignoriert Masse)

Umfassender Kraft-Controller

```
public class AdvancedForceController : MonoBehaviour
{
    private Rigidbody rb;

    [Header("Kraftparameter")]
    public float thrustForce = 100f;
    public float torqueStrength = 50f;
    public float explosionForce = 500f;
    public float explosionRadius = 10f;
    public float maxVelocity = 20f;

    [Header("Steuerung")]
    public KeyCode thrustKey = KeyCode.W;
    public KeyCode leftKey = KeyCode.A;
    public KeyCode rightKey = KeyCode.D;
    public KeyCode explosionKey = KeyCode.Space;

    void Start()
    {
        rb = GetComponent<Rigidbody>();

        // Sicherstellen, dass Rigidbody korrekt konfiguriert ist
        rb.useGravity = true;
        rb.drag = 0.1f;
        rb.angularDrag = 0.1f;
    }

    void FixedUpdate()
    {
        HandleMovement();
        HandleRotation();
        HandleSpecialForces();

        // Geschwindigkeitsbegrenzung
        LimitVelocity();
    }

    void HandleMovement()
    {
        // Schubkraft in Vorwaertsrichtung
        if (Input.GetKey(thrustKey))
        {
            Vector3 thrust = transform.forward * thrustForce;
            rb.AddForce(thrust, ForceMode.Force);

            // Visuelle/Audio-Effekte
            CreateThrustEffects();
        }

        // Seitenschub
        if (Input.GetKey(leftKey))
        {
            rb.AddForce(-transform.right * thrustForce * 0.5f, ForceMode.Force);
        }
        if (Input.GetKey(rightKey))
        {
            rb.AddForce(transform.right * thrustForce * 0.5f, ForceMode.Force);
        }
    }

    void HandleRotation()
    {
        // Gier-Steuerung (Yaw)
        if (Input.GetKey(KeyCode.Q))
        {
            rb.AddTorque(-transform.up * torqueStrength, ForceMode.Force);
        }
        if (Input.GetKey(KeyCode.E))
        {
            rb.AddTorque(transform.up * torqueStrength, ForceMode.Force);
        }
```

Kollisionserkennung und -reaktion

Unity Kollisions-Events **Kollisions-Events (für Collider):**

- **OnCollisionEnter()**: Beim Start der Kollision
- **OnCollisionStay()**: Während anhaltender Kollision
- **OnCollisionExit()**: Beim Ende der Kollision

Trigger-Events (für Trigger):

- **OnTriggerEnter()**: Objekt betritt Trigger-Zone
- **OnTriggerStay()**: Objekt bleibt in Trigger-Zone
- **OnTriggerExit()**: Objekt verlässt Trigger-Zone

Kollisions-Informationen:

- **Collision.contacts**: Kontaktpunkte
- **Collision.impulse**: Impuls der Kollision
- **Collision.relativeVelocity**: Relative Geschwindigkeit
- **ContactPoint.normal**: Oberflächennormale

Erweiterte Kollisionsbehandlung

```
public class CollisionAnalyzer : MonoBehaviour
{
    [Header("Kollisions-Einstellungen")]
    public float restitution = 0.8f;           // Restitutionskoeffizient
    public float minVelocityForSound = 2f;     // Min. Geschw. fuer Sound
    public float minVelocityForSparks = 5f;    // Min. Geschw. fuer Funken
    public AudioClip collisionSound;
    public ParticleSystem sparkEffect;

    [Header("Material-Eigenschaften")]
    public float metalDensity = 7800f;         // kg/m^3
    public float woodDensity = 600f;
    public float plasticDensity = 1200f;

    private Rigidbody rb;
    private AudioSource audioSource;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
        audioSource = GetComponent<AudioSource>();
    }

    void OnCollisionEnter(Collision collision)
    {
        // Kollisionsdetails berechnen
        Vector3 relativeVelocity = rb.velocity -
                                   (collision.rigidbody?.velocity ?? Vector3.zero);
        float collisionSpeed = relativeVelocity.magnitude;
        float collisionEnergy = 0.5f * rb.mass * collisionSpeed * collisionSpeed;

        // Sound basierend auf Kollisionsintensitaet
        if (collisionSpeed > minVelocityForSound && audioSource && collisionSound)
        {
            float volume = Mathf.Clamp01(collisionSpeed / 10f);
            float pitch = 0.8f + collisionSpeed / 20f;
            audioSource.pitch = pitch;
            audioSource.PlayOneShot(collisionSound, volume);
        }

        // Partikeleffekte fuer starke Kollisionen
        if (collisionSpeed > minVelocityForSparks && sparkEffect)
        {
            Vector3 contactPoint = collision.contacts[0].point;
            CreateSparkEffect(contactPoint, collision.contacts[0].normal);
        }

        // Materialspezifische Reaktionen
        HandleMaterialInteraction(collision, collisionSpeed);

        // Benutzerdefinierte Kollisionsreaktion
        ApplyCustomCollisionResponse(collision, relativeVelocity);

        // Kollisionsdaten fuer Analyse protokollieren
        LogCollisionData(collision, collisionSpeed, collisionEnergy);
    }

    void HandleMaterialInteraction(Collision collision, float speed)
    {
        // Materialeigenschaften basierend auf Tags
        string otherTag = collision.gameObject.tag;

        switch (otherTag)
        {
            case "Metal":
                // Metallische Kollision - hoher Restitutionskoeffizient
                restitution = 0.9f;
                break;
            case "Wood":
                // Holz - mittlerer Restitutionskoeffizient
```

Benutzerdefinierte Physik-Implementierung

Numerische Integrationsmethoden Unity verwendet implizite Euler-Integration, aber benutzerdefinierte Implementierungen können verwenden:

- **Expliziter Euler**: Einfach, aber potentiell instabil
- **Verlet-Integration**: Bessere Energieerhaltung
- **Runge-Kutta**: Höhere Genauigkeit für komplexe Systeme
- **Leapfrog**: Gut für Orbitalprobleme

Vergleich der Methoden:

- **Stabilität**: Verlet > Runge-Kutta > Euler
- **Genauigkeit**: Runge-Kutta > Verlet > Euler
- **Performance**: Euler > Verlet > Runge-Kutta
- **Energieerhaltung**: Verlet > Runge-Kutta > Euler

Benutzerdefinierter Physik-Integrator

```
public class CustomPhysicsObject : MonoBehaviour
{
    [Header("Physik-Eigenschaften")]
    public float mass = 1f;
    public Vector3 velocity = Vector3.zero;
    public Vector3 acceleration = Vector3.zero;
    public bool useGravity = true;
    public float damping = 0.1f;

    [Header("Integrations-Einstellungen")]
    public enum IntegrationMethod { Euler, Verlet, RungeKutta, Leapfrog }
    public IntegrationMethod method = IntegrationMethod.Verlet;
    public bool showTrajectory = true;
    public int trajectoryPoints = 50;

    [Header("Kraefte")]
    public Vector3 constantForce = Vector3.zero;
    public float springConstant = 50f;
    public Vector3 springCenter = Vector3.zero;

    private Vector3 previousPosition;
    private Vector3 currentPosition;
    private List<Vector3> trajectory = new List<Vector3>();
    private LineRenderer trajectoryRenderer;

    void Start()
    {
        currentPosition = transform.position;
        previousPosition = currentPosition;

        SetupTrajectoryRenderer();

        // Initiale Geschwindigkeit setzen
        if (velocity == Vector3.zero)
        {
            velocity = new Vector3(5f, 0f, 0f); // Standardgeschwindigkeit
        }
    }

    void SetupTrajectoryRenderer()
    {
        if (showTrajectory)
        {
            trajectoryRenderer = gameObject.AddComponent<LineRenderer>();
            trajectoryRenderer.material = new
                Material(Shader.Find("Sprites/Default"));
            trajectoryRenderer.color = Color.yellow;
            trajectoryRenderer.startWidth = 0.05f;
            trajectoryRenderer.endWidth = 0.05f;
            trajectoryRenderer.positionCount = trajectoryPoints;
        }
    }

    void FixedUpdate()
    {
        // Kraefte berechnen
        Vector3 totalForce = CalculateForces();
        acceleration = totalForce / mass;

        // Integration basierend auf gewaehlter Methode
        switch (method)
        {
            case IntegrationMethod.Euler:
                EulerIntegration();
                break;
            case IntegrationMethod.Verlet:
                VerletIntegration();
                break;
            case IntegrationMethod.RungeKutta:
                RungeKuttaIntegration();
```

Performance-Optimierung

Physik-Performance-Tipps **Kollisionserkennung optimieren:**

- Geeignete Kollisionserkennungs-Modi verwenden
- Compound Collider für komplexe Geometrien
- Mesh Collider nur wenn nötig (sehr teuer)
- Layer-basierte Kollisionsmatrix definieren

Rigidbody-Management:

- Anzahl aktiver Rigidbodies minimieren
- Ruhende Objekte als Kinematic setzen
- Object Pooling für Projektile verwenden
- Unnötige Constraints vermeiden

Optimierungen:

- Fixed Timestep angemessen anpassen
- Solver-Iterationen reduzieren wenn möglich
- Continuous Collision Detection sparsam einsetzen
- Physics Raycasts in Threads auslagern

Physik-Objekt-Pool

```
public class PhysicsObjectPool : MonoBehaviour
{
    [Header("Pool-Einstellungen")]
    public GameObject prefab;
    public int poolSize = 100;
    public int maxActiveObjects = 50;
    public float autoReturnTime = 10f;

    [Header("Performance")]
    public bool enableStatistics = true;
    public float cleanupInterval = 5f;

    private Queue<GameObject> pool = new Queue<GameObject>();
    private List<PooledObject> activeObjects = new List<PooledObject>();
    private Transform poolParent;

    private class PooledObject
    {
        public GameObject gameObject;
        public float spawnTime;
        public Rigidbody rigidbody;

        public PooledObject(GameObject obj)
        {
            gameObject = obj;
            spawnTime = Time.time;
            rigidbody = obj.GetComponent<Rigidbody>();
        }
    }

    void Start()
    {
        // Pool-Container erstellen
        poolParent = new GameObject("ObjectPool").transform;
        poolParent.SetParent(transform);

        // Pool vorab fuellen
        for (int i = 0; i < poolSize; i++)
        {
            GameObject obj = Instantiate(prefab, poolParent);
            obj.SetActive(false);
            pool.Enqueue(obj);
        }

        // Regelmaessige Bereinigung starten
        InvokeRepeating(nameof(CleanupObjects), cleanupInterval, cleanupInterval);
    }

    public GameObject GetObject(Vector3 position, Vector3 velocity, Vector3
         angularVelocity = default)
    {
        // Begrenzung aktiver Objekte
        if (activeObjects.Count >= maxActiveObjects)
        {
            // Aeltestes Objekt zurueckgeben
            ReturnOldestObject();
        }

        GameObject obj;

        if (pool.Count > 0)
        {
            obj = pool.Dequeue();
        }
        else
        {
            // Pool erschoepft, neues Objekt erstellen
            obj = Instantiate(prefab, poolParent);
            Debug.LogWarning("Pool exhausted, creating new object");
        }
```

Prüfungsvorbereitung

Unity Physics Prüfungsstrategie

Kernphysik-Konzepte beherrschen

- Newton'sche Gesetze und ihre Anwendungen
- Energie- und Impulserhaltung verstehen
- Rotationsdynamik-Berechnungen üben
- Wichtige Formeln und ihre Herleitungen kennen
- Vektormathematik und Koordinatensysteme

Unity-Implementierungs-Wissen

- Rigidbody-Komponenten-Eigenschaften und -Methoden
- Kraftanwendungstechniken und Force-Modi
- Kollisionserkennung und -reaktion
- Koordinatensystem-Transformationen
- Physics Material und Reibungsmodelle

Problemlösungsansatz

- Beteiligte physikalische Prinzipien identifizieren
- Koordinatensystem und Freikörperdiagramme aufstellen
- Erhaltungsgesetze anwenden wo angemessen
- In Unity-Implementierungskonzepte übersetzen
- Code-Snippets für häufige Szenarien memorieren

Häufige Prüfungsthemen

- Projektilbewegung mit Unity-Vektoren
- Kollisionsanalyse und Impulserhaltung
- Rotationsbewegung und Drehmoment-Berechnungen
- Energieerhaltung in mechanischen Systemen
- Custom Physics Implementation
- Performance-Optimierung in Unity Physics

Umfassendes Physik-Problem Eine Kugel (Masse 1 kg, Radius 0.1 m) rollt eine 30°-Schräge hinunter, kollidiert dann elastisch mit einer ruhenden Kugel gleicher Masse. Berechne die Endgeschwindigkeiten und implementiere die Simulation in Unity.

Teil 1: Rollen auf der Schräge
Für Vollkugel: $I = \frac{2}{5}mr^2$, Beschleunigung: $a = \frac{5g\sin\theta}{7}$
$a = \frac{5\times 9.81\times\sin(30\check{r})}{7} = 3.51\,m/s^2$
Nach Rollstrecke $d = 2m$: $v = \sqrt{2ad} = \sqrt{2\times 3.51\times 2} = 3.74\,m/s$
Teil 2: Elastische Kollision
Für gleiche Massen in 1D elastischer Kollision: $v_1' = 0$, $v_2' = v_1 = 3.74\,m/s$
Unity-Implementierung:

```
public class RollingBallSimulation : MonoBehaviour
{
    [Header("Physik-Parameter")]
    public float inclineAngle = 30f;
    public float ballMass = 1f;
    public float ballRadius = 0.1f;
    public float rollDistance = 2f;

    [Header("Simulation")]
    public bool autoStart = true;
    public float simulationSpeed = 1f;

    private GameObject ball1, ball2;
    private Rigidbody rb1, rb2;

    void Start()
    {
        if (autoStart)
        {
            StartSimulation();
        }
    }

    public void StartSimulation()
    {
        // Theoretische Berechnungen
        CalculateTheoreticalValues();

        // Physik-Simulation aufsetzen
        SetupSimulation();
    }

    void CalculateTheoreticalValues()
    {
        float angleRad = inclineAngle * Mathf.Deg2Rad;
        float rollingAccel = (5f * Physics.gravity.magnitude *
                              Mathf.Sin(angleRad)) / 7f;

        float finalVelocity = Mathf.Sqrt(2f * rollingAccel * rollDistance);

        Debug.Log($"Theoretische Werte:");
        Debug.Log($"Rollbeschleunigung: {rollingAccel:F2} m/s^2");
        Debug.Log($"Endgeschwindigkeit vor Kollision: {finalVelocity:F2} m/s");
        Debug.Log($"Nach elastischer Kollision: Ball1 = 0 m/s, Ball2 =
            {finalVelocity:F2} m/s");
    }

    void SetupSimulation()
    {
        // Schraege erstellen
        SetupIncline();

        // Kugeln erstellen
        SetupBalls();

        // Physik-Materialien konfigurieren
        SetupPhysicsMaterials();
```

Wichtige Erkenntnisse **Theoretische Grundlagen:**

- Physikalische Gesetze gelten universell
- Erhaltungsgesetze sind zentral für Problemlösungen
- Numerische Integration beeinflusst Genauigkeit
- Reibung und Dämpfung sind realitätsnah wichtig

Unity-spezifische Aspekte:

- Fixed Timestep ist entscheidend für Stabilität
- Physics Materials beeinflussen Verhalten stark
- Performance-Optimierung ist für komplexe Szenen notwendig
- Debugging-Tools helfen bei der Problemanalyse