Physic Engines

Jil Zerndt
FS 2025

Bezugssysteme und Einführung

Einführung in Physik Engines

Physics Engines
Physics Engines sind Softwarekomponenten, die physikalische Effekte in Computerprogrammen (z.B. Spielen) simulieren. In diesem Modul geht es um:
- Physikalische Modellierung in Unity
- Verstehen physikalischer Grundprinzipien für die korrekte Anwendung in Unity
- Kopplung von Physiksimulatoren (wie PhysX) mit realistischen Parametern

Selbst wenn Unity's PhysX viele Funktionen automatisch berechnet, benötigt man ein fundiertes Verständnis der Physik, um realistische Simulationen zu erstellen.

Die Modellbildung in Unity folgt diesem Ablauf:
1. Wirklichkeit → Physikalisches Modell
2. Physikalisches Modell → Mathematisches Modell
3. Mathematisches Modell → Numerisches Modell
4. Numerisches Modell → Modellierte Werte
5. Diese können dann mit experimentellen Daten verglichen werden

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 Bezugssystem
Ein Unterschied zwischen Unity und den in der Physik üblichen Bezugssystemen:
- Physik und Mathematik verwenden üblicherweise ein Rechtssystem
- Unity verwendet ein Linkssystem
- In Unity zeigt die positive y-Achse nach oben (nicht die z-Achse)

Dies hat Auswirkungen auf die Berechnung von Kreuzprodukten und die Interpretation von Rotationen.

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

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

Introduction to Unity

Course Overview

Physics Engines Course
This course covers the fundamentals of physics simulation in Unity game engine, focusing on Newtonian mechanics and their implementation in interactive applications. The course bridges theoretical physics concepts with practical programming in Unity using C#.

Course Structure
- 2 Credits = 60 hours total (30h presence, 30h self-study)
- Moodle exercises online: 10% (4h)
- Unity projects in groups, 3 phases (16h total, 40%):
  - Introduction example, Part 1: 9 March (5%)
  - Part 2: 9 April (15%)
  - Part 3: 9 May (20%)
- Final oral exam: 50% (10h)

Physics Topics Covered

Newtonian Mechanics Overview

- Coordinate systems and reference frames
- Kinematics: describing motion
- Dynamics: influence of forces
- Forces: friction, attraction, collisions
- Energy and momentum: conservation laws
- Rotational motion applications

Unity Integration

Unity serves as the practical platform for implementing physics concepts, allowing visualization and interaction with physical simulations through C# scripting.

Development Environment

Unity Setup Requirements

Essential tools and resources for the course:

```
// Course resources available online:
// - 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 textbook: https://link.springer.com/book/10.1007/978-3-662-58281-7
// - Unity for exercises
// - MS Teams for communication
```

Vector Operations in Unity

- Vector3 structure for 3D positions and directions
- Built-in functions for common vector operations
- Static properties: Vector3.forward, Vector3.back, Vector3.up, Vector3.down, Vector3.left, Vector3.right
- Mathematical operations: dot product, cross product, magnitude, normalization

Course Assessment

Exam Preparation Strategy

Theory Component

- Study physics formulas from weekly formula collection
- Understand derivations and applications of each formula
- Practice problem-solving using step-by-step approaches

Unity Implementation

- Master Vector3 operations and coordinate transformations
- Understand Rigidbody component and physics simulation
- Practice implementing forces and motion equations in C#

Project Work

- Document all project phases thoroughly
- Understand physics principles behind implemented features
- Prepare to explain code implementation and physics concepts

Formula Entry in Word

Course requirement: Maintain a formula collection using Word's equation editor.

- Use Alt + Shift + * (or Alt + Shift + =) to insert equations
- Subscript: underscore (_)
- Superscript: caret (^)
- Complete formatting with Space

The oral exam lasts 15 minutes and covers physics concepts from the entire course. Unity game engine knowledge is not part of the examination content, but one question will relate to the semester project.

Kinematik der Translation

Grundlagen der Kinematik

Kinematik

Die Kinematik beschreibt die Bewegung eines Körpers, ohne auf deren Ursachen einzugehen. Die Bewegung eines Körpers ist vollständig beschrieben durch:

- seinen Ort (Vektor $\vec{r}$)
- seine Geschwindigkeit ($\vec{v}$)
- seine Beschleunigung ($\vec{a}$)

Diese drei Größen hängen durch Ableiten bzw. Integrieren zusammen.

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: $\vec{v} = \int \vec{a}\, dt$

Mittlere Geschwindigkeit und Beschleunigung

Mittlere Geschwindigkeit

Die mittlere Geschwindigkeit ist die Änderung des Ortes dividiert durch die dafür benötigte Zeit:

$$\bar{v}_x = \frac{\Delta r_x}{\Delta t} \tag{1}$$

Sie stellt den Durchschnittswert über das betrachtete Zeitintervall dar.

Mittlere Beschleunigung

Die mittlere Beschleunigung ist die Änderung der Geschwindigkeit dividiert durch die dafür benötigte Zeit:

$$\bar{a}_x = \frac{\Delta v_x}{\Delta t} \tag{2}$$

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:

$$\vec{v}(t_0) = \lim_{t_1 \to t_0} \frac{\Delta \vec{r}}{t_1 - t_0} = \frac{d\vec{r}}{dt} \tag{3}$$

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

Der Betrag der Geschwindigkeit wird oft als Schnelligkeit bezeichnet:

$$|\vec{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2} \tag{4}$$

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:

$$\Delta x = \int_{t_1}^{t_2} v(t)\, dt \tag{5}$$

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:

$$v(t) = v_0 + at \tag{6}$$

2. Position in Abhängigkeit von der Zeit bestimmen:

$$r(t) = r_0 + v_0 t + \frac{1}{2} a t^2 \tag{7}$$

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

$$v^2 = v_0^2 + 2a(r - r_0) \tag{8}$$

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: $t_{gesamt} = \frac{2v_0}{g}$

Kinematics

Basic Concepts

Kinematics

The branch of mechanics that describes motion without considering the forces that cause it. Focuses on position, velocity, and acceleration as functions of time.

Reference Frames

All motion is relative to a chosen coordinate system. Unity uses a left-handed coordinate system where:

- X-axis: right (positive) / left (negative)
- Y-axis: up (positive) / down (negative)
- Z-axis: forward (positive) / backward (negative)

Position and Displacement

Position Vector

Vector $\vec{r}(t)$ that describes the location of an object at time t relative to the origin of the coordinate system.

$$\vec{r}(t) = x(t)\hat{i} + y(t)\hat{j} + z(t)\hat{k}$$

Displacement

Change in position vector over a time interval:

$$\Delta\vec{r} = \vec{r}(t_2) - \vec{r}(t_1)$$

Unity Position Implementation

```
// Get current position of a GameObject
Vector3 currentPosition = transform.position;

// Calculate displacement between two positions
Vector3 displacement = finalPosition - initialPosition;

// Update position over time
transform.position += velocity * Time.deltaTime;
```

Velocity

Average Velocity

$$\vec{v}_{avg} = \frac{\Delta\vec{r}}{\Delta t} = \frac{\vec{r}(t_2) - \vec{r}(t_1)}{t_2 - t_1}$$

Instantaneous Velocity

$$\vec{v}(t) = \lim_{\Delta t \to 0} \frac{\Delta\vec{r}}{\Delta t} = \frac{d\vec{r}}{dt}$$

Speed vs. Velocity

- Speed: magnitude of velocity vector (scalar)
- Velocity: vector quantity with both magnitude and direction
- $|\vec{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$

Acceleration

Average Acceleration

$$\vec{a}_{avg} = \frac{\Delta\vec{v}}{\Delta t} = \frac{\vec{v}(t_2) - \vec{v}(t_1)}{t_2 - t_1}$$

Instantaneous Acceleration

$$\vec{a}(t) = \lim_{\Delta t \to 0} \frac{\Delta\vec{v}}{\Delta t} = \frac{d\vec{v}}{dt} = \frac{d^2\vec{r}}{dt^2}$$

Motion Equations

Kinematic Equations for Constant Acceleration

For motion with constant acceleration $\vec{a}$:

Position as function of time:

$$\vec{r}(t) = \vec{r}_0 + \vec{v}_0 t + \frac{1}{2}\vec{a}t^2$$

Velocity as function of time:

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

Velocity-position relation:

$$\vec{v}^2 = \vec{v}_0^2 + 2\vec{a} \cdot (\vec{r} - \vec{r}_0)$$

Unity Kinematic Implementation

```
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;

        // Calculate new position using kinematic equation
        Vector3 newPosition = initialPosition +
                              initialVelocity * t +
                              0.5f * acceleration * t * t;

        transform.position = newPosition;
    }
}
```

Special Cases

Uniform Motion

Motion with constant velocity ($\vec{a} = 0$):

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

Free Fall

Motion under gravity alone ($\vec{a} = -g\hat{j}$ in Unity coordinates):

- $g = 9.81\, m/s^2$ (Earth's gravitational acceleration)
- Unity default gravity: $-9.81\, m/s^2$ in Y-direction

Solving Kinematic Problems

Step 1: Identify known variables

- Initial position $\vec{r}_0$
- Initial velocity $\vec{v}_0$
- Acceleration $\vec{a}$
- Time t or final position/velocity

Step 2: Choose appropriate equation

- Use $\vec{v}(t) = \vec{v}_0 + \vec{a}t$ when time is known
- Use $\vec{r}(t) = \vec{r}_0 + \vec{v}_0 t + \frac{1}{2}\vec{a}t^2$ for position
- Use $\vec{v}^2 = \vec{v}_0^2 + 2\vec{a} \cdot \Delta\vec{r}$ when time is unknown

Step 3: Solve component-wise

- Break vectors into x, y, z components
- Solve each component independently
- Combine results into final vector

Projectile Motion Problem

A ball is thrown from height $h = 10m$ with initial velocity $\vec{v}_0 = (5, 8, 0)\, m/s$. Calculate time to hit ground and horizontal distance traveled.

Given: $\vec{r}_0 = (0, 10, 0)$, $\vec{v}_0 = (5, 8, 0)$, $\vec{a} = (0, -9.81, 0)$
Y-component (vertical): $y(t) = 10 + 8t - 4.905t^2$
When ball hits ground: $y(t) = 0$ $10 + 8t - 4.905t^2 = 0$ $t = 2.24s$ (using quadratic formula)
X-component (horizontal): $x(t) = 5t = 5 \times 2.24 = 11.2m$

Dynamik der Translation

Grundlagen der Dynamik

Dynamik

Die Dynamik beschäftigt sich mit den Ursachen von Bewegungen, also den Kräften, die auf einen Körper wirken. Sie baut auf der Kinematik auf und erweitert diese um die Betrachtung der wirkenden Kräfte.

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.

Erstes Newton'sches Gesetz (Trägheitsgesetz)

Trägheitsgesetz

Das erste Newton'sche Gesetz lässt sich mathematisch ausdrücken als:

$$\vec{F} = 0 \Rightarrow \vec{v} = \text{const.} \tag{9}$$

Dies bedeutet auch:

$$\vec{F} = 0 \Rightarrow \vec{a} = \frac{d\vec{v}}{dt} = \frac{d}{dt}(\text{const.}) = 0 \tag{10}$$

Das Gesetz gilt nur in sogenannten Inertialsystemen: Bezugssystemen ohne Beschleunigung oder Rotation.

Trägheitsgesetz auf der Luftkissenbahn

Ein Gleiter auf einer Luftkissenbahn bewegt sich ohne Reibung mit konstanter Geschwindigkeit weiter, wenn keine Kraft auf ihn wirkt. Dies ist eine direkte Demonstration des Trägheitsgesetzes.
Das gleiche Prinzip gilt für einen Stein, der von der Spitze eines fahrenden Schiffes fällt - er behält seine horizontale Geschwindigkeit bei und landet am Fuß des Mastes, nicht dahinter oder davor.

Zweites Newton'sches Gesetz (Bewegungsgesetz)

Impuls

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

$$\vec{p} = m \cdot \vec{v} \tag{11}$$

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

Bewegungsgesetz

Das zweite Newton'sche Gesetz in seiner allgemeinen Form:

$$\vec{F} = \frac{d\vec{p}}{dt} \tag{12}$$

Für Körper mit konstanter Masse vereinfacht sich dies zu:

$$\vec{F} = m \cdot \vec{a} \tag{13}$$

In integraler Form:

$$\vec{p} = \vec{p}_0 + \int_0^t \vec{F}(t)\, dt \tag{14}$$

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

Berechnung von Kräften und Beschleunigungen

Berechnung der Beschleunigung aus Kräften

- Alle auf den Körper wirkenden Kräfte identifizieren
- Kräfte vektoriell addieren zur resultierenden Kraft $\vec{F}_{res}$
- Beschleunigung berechnen: $\vec{a} = \frac{\vec{F}_{res}}{m}$

Beispiel: Fahrstuhl

- Fahrstuhl in Ruhe: $F_{\text{Seil}} - F_G = 0 \Rightarrow F_{\text{Seil}} = m \cdot g$
- Fahrstuhl beschleunigt nach oben: $F_{\text{Seil}} - F_G = m \cdot a \Rightarrow F_{\text{Seil}} = m \cdot (g + a)$
- Fahrstuhl beschleunigt nach unten: $F_{\text{Seil}} - F_G = -m \cdot a \Rightarrow F_{\text{Seil}} = m \cdot (g - a)$

Drittes Newton'sches Gesetz (Wechselwirkungsgesetz)

Wechselwirkungsgesetz

Das dritte Newton'sche Gesetz lautet mathematisch:

$$\vec{F}_{12} = -\vec{F}_{21} \tag{15}$$

wobei $\vec{F}_{12}$ die Kraft bezeichnet, die Körper 1 auf Körper 2 ausübt, und $\vec{F}_{21}$ die Kraft, die Körper 2 auf Körper 1 ausübt.

Merkmale von Kräftepaaren

Kräftepaare des Wechselwirkungsgesetzes haben folgende Eigenschaften:

1. Gleicher Betrag, entgegengesetzte Richtung
2. Greifen an verschiedenen Körpern an
3. Haben die gleiche physikalische Ursache

Anwendungen des Wechselwirkungsgesetzes

- Gravitationskraft: Die Erde zieht einen Menschen an, aber der Mensch zieht auch die Erde an (mit der gleichen Kraft)
- Bremsen eines Autos: Das Auto stößt die Straße nach vorn, die Straße stößt das Auto nach hinten
- Zwei verbundene Boote: Wenn eine Person in einem Boot das andere Boot an einer Leine zu sich zieht, bewegen sich beide Boote aufeinander zu

Superpositionsprinzip

Superpositionsprinzip

Das Superpositionsprinzip besagt, dass sich mehrere Kräfte $\vec{F}_1, \vec{F}_2, ..., \vec{F}_n$, die auf einen Punkt wirken, vektoriell zu einer resultierenden Kraft $\vec{F}$ addieren:

$$\vec{F} = \sum_{i=1}^{n} \vec{F}_i \tag{16}$$

Für jede Komponente gilt entsprechend:

$$F_x = \sum_{i=1}^{n} F_{xi} \quad F_y = \sum_{i=1}^{n} F_{yi} \quad F_z = \sum_{i=1}^{n} F_{zi} \tag{17}$$

Kräfte "freischneiden"

Schritte zum Freischneiden

1. Zeichne den zu untersuchenden Körper isoliert
2. Zeichne alle auf den Körper wirkenden Kräfte als Vektoren ein
3. Bestimme die resultierende Kraft: $\vec{F}_{res} = \sum \vec{F}_i$
4. Berechne die Beschleunigung: $\vec{a} = \frac{\vec{F}_{res}}{m}$

Beispiel: Kiste auf schiefer Ebene

- Gewichtskraft: $\vec{F}_G = m \cdot \vec{g}$ (nach unten)
- Normalkraft: $\vec{F}_N$ (senkrecht zur Ebene)
- Reibungskraft: $\vec{F}_R = \mu \cdot \vec{F}_N$ (parallel zur Ebene, entgegen der Bewegungsrichtung)
- Hangabtriebskraft: $F_{Hang} = m \cdot g \cdot \sin \alpha$ (parallel zur Ebene nach unten)

Kräfte in Unity implementieren

```
// Anwenden einer Kraft in Unity
void FixedUpdate() {
    // Gravitationskraft berechnen
    Vector3 gravityForce = new Vector3(0, -9.81f * mass, 0);

    // Normalkraft berechnet Unity automatisch bei Kollisionen

    // Hangabtriebskraft bei schiefer Ebene mit Winkel alpha
    float alpha = 30f * Mathf.Deg2Rad; // 30 Grad in Radian
    float hangForce = mass * 9.81f * Mathf.Sin(alpha);
    Vector3 hangForceVector = new Vector3(hangForce, 0, 0);

    // Reibungskraft
    float frictionCoeff = 0.3f;
    Vector3 frictionForce = -frictionCoeff * mass * 9.81f * Mathf.Cos(alpha) *
        rigidBody.velocity.normalized;

    // Alle Kraefte anwenden
    rigidBody.AddForce(gravityForce + hangForceVector + frictionForce);
}
```

Dynamics and Forces

Newton's Laws of Motion

Newton's First Law (Law of Inertia)

An object at rest stays at rest, and an object in motion stays in motion at constant velocity, unless acted upon by a net external force.

$$\sum \vec{F} = 0 \Rightarrow \vec{v} = \text{constant}$$

Newton's Second Law

The acceleration of an object is directly proportional to the net force acting on it and inversely proportional to its mass.

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

Newton's Third Law (Action-Reaction)

For every action, there is an equal and opposite reaction.

$$\vec{F}_{AB} = -\vec{F}_{BA}$$

Common Forces

Gravitational Force

Force between two masses:

$$\vec{F}_g = -\frac{Gm_1m_2}{r^2}\hat{r}$$

Near Earth's surface:

$$\vec{F}_g = m\vec{g} = -mg\hat{j}$$

where $g = 9.81\,m/s^2$

Spring Force (Hooke's Law)

Force exerted by a spring:

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

where k is the spring constant and $\Delta\vec{l}$ is displacement from equilibrium.

Friction Force

Static friction: $f_s \leq \mu_s N$
Kinetic friction: $f_k = \mu_k N$
where μ_s, μ_k are friction coefficients and N is normal force.

Damping Force

Force opposing motion, proportional to velocity:

$$\vec{F}_d = -k_{damper}\vec{v}$$

Used in Unity for realistic motion with air resistance or viscous damping.

Unity Force Implementation

Applying Forces in Unity

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

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

    void FixedUpdate()
    {
        // Gravity (automatically applied by Unity)
        // Vector3 gravity = Physics.gravity * rb.mass;

        // Spring force to origin
        Vector3 springForce = -springConstant * transform.position;

        // Damping force
        Vector3 dampingForce = -damping * rb.velocity;

        // Apply total force
        rb.AddForce(springForce + dampingForce);
    }
}
```

Unity Force Modes

Unity provides different ways to apply forces through `Rigidbody.AddForce()`:

- `Force`: Continuous force using mass (default)
- `Acceleration`: Continuous force ignoring mass
- `Impulse`: Instantaneous force using mass
- `VelocityChange`: Instantaneous force ignoring mass

Harmonic Oscillator

Simple Harmonic Motion

Motion under spring force with equation:

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

Solution: $x(t) = A\cos(\omega t + \phi)$

where $\omega = \sqrt{\frac{k}{m}}$ is the angular frequency.

Harmonic Oscillator Properties

Angular frequency:

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

Period:

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

Frequency:

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

Total energy:

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

Harmonic Oscillator 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;
    }
}
```

Rotational Forces

Torque (Moment)

Rotational equivalent of force:

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

For rotation about fixed axis:

$$\tau = rF\sin\theta$$

Angular Spring (Torsional Spring)

Analogous to linear spring for rotational motion:

$$\tau = -D\Delta\phi$$

where D is the angular spring constant and $\Delta\phi$ is angular displacement.

Applying Torque 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))
        {
            // Apply torque around Y-axis
            rb.AddTorque(Vector3.up * torqueStrength);
        }

        // Rotational spring to upright position
        Vector3 restoreToruqe = -torqueStrength * transform.eulerAngles.x
                                * Vector3.right;
        rb.AddTorque(restoreToruqe);
    }
}
```

Problem Solving Strategy

Force Analysis Method

Step 1: Identify the system

- Define the object(s) of interest
- Choose coordinate system
- Identify time interval of interest

Step 2: Draw free-body diagram

- Show all forces acting on the object
- Label force vectors with symbols
- Do not include forces the object exerts on other objects

Step 3: Apply Newton's Second Law

- Write $\sum \vec{F} = m\vec{a}$ in component form
- $\sum F_x = ma_x$, $\sum F_y = ma_y$, $\sum F_z = ma_z$
- Solve for unknown quantities

Step 4: Implement in Unity

- Use `Rigidbody.AddForce()` for each force
- Consider appropriate force mode
- Use `FixedUpdate()` for physics calculations

Block on Inclined Plane

A block of mass $m = 2kg$ slides down a frictionless incline of angle $\theta = 30$ř. Find acceleration and implement in Unity.

Free-body diagram: Weight mg downward, normal force N perpendicular to surface.
Component analysis:

- Along incline: $mg\sin\theta = ma$
- Perpendicular: $N - mg\cos\theta = 0$

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

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

Kräfte

Arten von Kräften

Grundlegende Wechselwirkungen

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 die Gravitation und die elektromagnetischen Kräfte relevant. Die starke und schwache Kernkraft wirken nur auf atomarer Ebene.

Kraft

Eine Kraft ist ein Einfluss, der den Bewegungszustand eines Körpers ändert. Im SI-System wird sie in Newton (N) gemessen:

$$1\,\mathrm{N} = 1\,\frac{\mathrm{kg}\cdot\mathrm{m}}{\mathrm{s}^2} \tag{18}$$

Kräfte ändern den Impuls eines Körpers gemäß der Formel:

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

Gravitationskraft

Gravitationsgesetz

Das Newton'sche Gravitationsgesetz beschreibt die Anziehungskraft zwischen zwei Massen m_1 und m_2, die im Abstand R voneinander entfernt sind:

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

Dabei ist G die Gravitationskonstante mit dem Wert:

$$G = 6.67 \cdot 10^{-11}\,\frac{\mathrm{m}^3}{\mathrm{kg}\cdot\mathrm{s}^2} \tag{21}$$

Die Kraft wirkt entlang der Verbindungslinie der beiden Massen und ist eine Anziehungskraft.

Erdbeschleunigung

Die Gewichtskraft F_G eines Körpers der Masse m auf der Erdoberfläche ist:

$$F_G = m \cdot g \tag{22}$$

Die Erdbeschleunigung g lässt sich aus dem Gravitationsgesetz herleiten:

$$g = G \cdot \frac{M_{\mathrm{Erde}}}{R_{\mathrm{Erde}}^2} \approx 9.81\,\frac{\mathrm{m}}{\mathrm{s}^2} \tag{23}$$

Mit zunehmender Höhe h nimmt die Fallbeschleunigung ab:

$$g(h) = G \cdot \frac{M_{\mathrm{Erde}}}{(R_{\mathrm{Erde}} + h)^2} \tag{24}$$

Federkraft

Federkraft (Hooke'sches Gesetz)

Für eine lineare Feder gilt das Hooke'sche Gesetz:

$$\vec{F} = -k \cdot \vec{x} \tag{25}$$

Dabei ist:
- k: die Federkonstante in N/m
- $\vec{x}$: die Auslenkung der Feder aus ihrer Ruhelage
- Das negative Vorzeichen bedeutet, dass die Kraft der Auslenkung entgegengerichtet ist

Spannenergie einer Feder

Die in einer gespannten Feder gespeicherte potentielle Energie beträgt:

$$E_{\mathrm{spann}} = \frac{1}{2} \cdot k \cdot x^2 \tag{26}$$

Diese Energie kann in kinetische Energie umgewandelt werden, wenn die Feder entspannt wird.

Harmonischer Oszillator

Ein harmonischer Oszillator ist ein System, bei dem die rücktreibende Kraft proportional zur Auslenkung ist. Beispiele sind eine Masse an einer Feder oder ein Pendel bei kleinen Auslenkungen.
Die Bewegungsgleichung eines harmonischen Oszillators lautet:

$$\frac{d^2x}{dt^2} = -\frac{k}{m} \cdot x \tag{27}$$

Die Lösung ist eine harmonische Schwingung mit der Kreisfrequenz ω:

$$x(t) = x_0 \cdot \cos(\omega t) \tag{28}$$

wobei $\omega = \sqrt{\frac{k}{m}}$ und die Schwingungsdauer $T = \frac{2\pi}{\omega}$ beträgt.

Simulation eines harmonischen Oszillators in Unity

```
// Harmonischer Oszillator
void FixedUpdate() {
    // Federkraft berechnen (F = -k*x)
    float springForce = -springConstant * rigidbody.position.x;

    // Kraft anwenden
    rigidbody.AddForce(new Vector3(springForce, 0f, 0f));

    // Zeit aktualisieren
    currentTimeStep += Time.deltaTime;
}
```

Reibungskräfte

Arten der Reibung

Man unterscheidet zwischen:
- Äußerer Reibung (Kontaktflächen von sich berührenden Festkörpern):
  - Haftreibung
  - Gleitreibung
  - Rollreibung, Wälzreibung, Bohrreibung, Seilreibung
- Innerer Reibung (zwischen benachbarten Teilchen bei Verformungen innerhalb von Festkörpern, Flüssigkeiten und Gasen)

Trockene Reibung

Für trockene Reibung (Coulomb-Reibung) gilt:

$$\vec{F}_R = \mu \cdot \vec{F}_N \tag{29}$$

Dabei ist:

- μ: Reibungskoeffizient (dimensionslos)
- $\vec{F}_N$: Normalkraft
- Die Richtung der Reibungskraft ist stets entgegengesetzt zur Bewegungsrichtung bzw. zur Zugkraft

Man unterscheidet zwischen:

- Haftreibung: $\vec{F}_{\text{Haft}} \leq \mu_{\text{Haft}} \cdot \vec{F}_N$ (gleicht äußere Kraft bis zu einem Maximalwert aus)
- Gleitreibung: $\vec{F}_{\text{Gleit}} = \mu_{\text{Gleit}} \cdot \vec{F}_N$ (konstant bei gegebener Normalkraft)
- Dabei gilt in der Regel: $\mu_{\text{Haft}} > \mu_{\text{Gleit}}$

Viskose Reibung

Für die Reibung in Flüssigkeiten und Gasen gelten je nach Strömungsregime:

- Laminare Strömung (Stokes'sche Reibung für eine Kugel):

$$\vec{F}_R = -6 \cdot \pi \cdot \eta \cdot r \cdot v \cdot \vec{e}_v \tag{30}$$

  wobei η die Viskosität des Mediums ist
- Turbulente Strömung:

$$\vec{F}_R = -\frac{1}{2} \cdot \rho \cdot A \cdot c_w \cdot \vec{v}^2 \cdot \vec{e}_v \tag{31}$$

  wobei ρ die Dichte des Mediums, A die Stirnfläche und c_w der Widerstandsbeiwert ist

Reibung in Unity implementieren

Trockene Reibung

```
// Trockene Reibung implementieren
void FixedUpdate() {
    // Normale Kraft berechnen (Gewichtskraft 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) {
        // Reibungskraft entgegen der Bewegungsrichtung
        Vector3 frictionVector = -velocityDirection * frictionForce;
        rigidbody.AddForce(frictionVector);
    }
}
```

Luftwiderstand (turbulente Reibung)

```
// Luftwiderstand implementieren
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 und Betrag
    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;

    // Kraft anwenden
    rigidbody.AddForce(dragForce);
}
```

Trägheitskräfte

Beschleunigte Bezugssysteme

In beschleunigten Bezugssystemen treten Trägheitskräfte (auch Scheinkräfte genannt) auf, um die Newton'schen Gesetze weiterhin anwenden zu können.
Beispiele:

- Translatorische Trägheitskraft bei linearer Beschleunigung:

$$\vec{F}_{\text{Trägheit}} = -m \cdot \vec{a}_{\text{System}} \tag{32}$$

- Zentrifugalkraft bei Rotation:

$$\vec{F}_{\text{Zentrifugal}} = m \cdot \omega^2 \cdot r \cdot \vec{e}_r \tag{33}$$

- Corioliskraft bei Bewegung in einem rotierenden System:

$$\vec{F}_{\text{Coriolis}} = 2 \cdot m \cdot \vec{v} \times \vec{\omega} \tag{34}$$

Trägheitskräfte im Alltag

- Im bremsenden Zug fühlt man sich nach vorne gedrückt
- In einer Kurve wird man nach außen gedrückt (Zentrifugalkraft)
- Die Corioliskraft beeinflusst Winde und Meeresströmungen auf der Erde (auf der Nordhalbkugel werden sie nach rechts, auf der Südhalbkugel nach links abgelenkt)

Trägheitskräfte sind keine ëchten"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:

$$\vec{p} = m \cdot \vec{v} \tag{35}$$

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

Impulserhaltung

Das Prinzip der Impulserhaltung besagt, dass in einem abgeschlossenen System der Gesamtimpuls konstant bleibt, wenn keine äußeren Kräfte wirken:

$$\sum \vec{p}_{\text{vorher}} = \sum \vec{p}_{\text{nachher}} \tag{36}$$

In Komponenten ausgedrückt für zwei Körper:

$$m_1 \cdot v_{1,\text{vorher}} + m_2 \cdot v_{2,\text{vorher}} = m_1 \cdot v_{1,\text{nachher}} + m_2 \cdot v_{2,\text{nachher}} \tag{37}$$

Die Impulserhaltung gilt auch bei dissipativen Vorgängen wie inelastischen Stößen, bei denen Energie verloren geht.

Kraftstoß

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

$$\vec{I} = \int_{t_1}^{t_2} \vec{F}(t)\, dt \tag{38}$$

Der Kraftstoß ist gleich der Impulsänderung:

$$\vec{I} = \Delta\vec{p} = \vec{p}_{\text{nachher}} - \vec{p}_{\text{vorher}} \tag{39}$$

Für eine konstante Kraft während der Zeitspanne Δt gilt:

$$\vec{I} = \vec{F} \cdot \Delta t \tag{40}$$

Stöße

Elastischer Stoß

Bei einem elastischen Stoß bleiben sowohl der Gesamtimpuls als auch die Gesamtenergie erhalten. Es gilt:

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

$$\frac{1}{2}m_1 \cdot v_{1,\text{vorher}}^2 + \frac{1}{2}m_2 \cdot v_{2,\text{vorher}}^2 = \frac{1}{2}m_1 \cdot v_{1,\text{nachher}}^2 + \frac{1}{2}m_2 \cdot v_{2,\text{nachher}}^2 \tag{42}$$

Inelastischer Stoß

Bei einem inelastischen Stoß bleibt nur der Gesamtimpuls erhalten, während die mechanische Energie teilweise in andere Energieformen (z.B. Wärme) umgewandelt wird.
Im Extremfall des vollständig inelastischen Stoßes "kleben"die Körper nach dem Stoß zusammen und bewegen sich mit einer gemeinsamen Geschwindigkeit v_{nachher}:

$$v_{\text{nachher}} = \frac{m_1 \cdot v_{1,\text{vorher}} + m_2 \cdot v_{2,\text{vorher}}}{m_1 + m_2} \tag{43}$$

Berechnung von Stößen in 1D

Elastischer Stoß

Gegeben: Massen m_1 und m_2, Anfangsgeschwindigkeiten v_1 und v_2
1. Schwerpunktgeschwindigkeit berechnen:

$$v_{Spt} = \frac{m_1 \cdot v_1 + m_2 \cdot v_2}{m_1 + m_2} \tag{44}$$

2. Geschwindigkeiten nach dem Stoß berechnen (Spiegelung an der Schwerpunktgeschwindigkeit):

$$u_1 = v_{Spt} - (v_1 - v_{Spt}) = 2v_{Spt} - v_1 \tag{45}$$

$$u_2 = v_{Spt} - (v_2 - v_{Spt}) = 2v_{Spt} - v_2 \tag{46}$$

Oder mit den Formeln:

$$u_1 = \frac{m_1 - m_2}{m_1 + m_2}v_1 + \frac{2m_2}{m_1 + m_2}v_2 \tag{47}$$

$$u_2 = \frac{2m_1}{m_1 + m_2}v_1 + \frac{m_2 - m_1}{m_1 + m_2}v_2 \tag{48}$$

Vollständig inelastischer Stoß

Gegeben: Massen m_1 und m_2, Anfangsgeschwindigkeiten v_1 und v_2
1. Gemeinsame Geschwindigkeit nach dem Stoß berechnen:

$$u_1 = u_2 = \frac{m_1 \cdot v_1 + m_2 \cdot v_2}{m_1 + m_2} \tag{49}$$

Elastischer Stoß mit ungleichen Massen

Wenn eine leichte Kugel auf eine ruhende schwere Kugel trifft ($m_1 < m_2$ und $v_2 = 0$):

$$u_1 = \frac{m_1 - m_2}{m_1 + m_2}v_1 \tag{50}$$

$$u_2 = \frac{2m_1}{m_1 + m_2}v_1 \tag{51}$$

Für $m_1 \ll m_2$ (z.B. Tennisball gegen Wand) gilt näherungsweise:

$$u_1 \approx -v_1 \quad \text{(Richtungsumkehr)} \tag{52}$$

$$u_2 \approx 0 \quad \text{(schwerer Körper bleibt fast in Ruhe)} \tag{53}$$

Umgekehrt, wenn eine schwere Kugel auf eine leichte, ruhende Kugel trifft ($m_1 > m_2$ und $v_2 = 0$):

$$u_1 \approx v_1 \quad \text{(fast unverändert)} \tag{54}$$

$$u_2 \approx 2v_1 \quad \text{(doppelte Geschwindigkeit des auftreffenden Körpers)} \tag{55}$$

Vergleich: Impuls- und Energieerhaltung

- Impulserhaltung gilt immer, wenn keine äußeren Kräfte wirken (Grundprinzip der Mechanik)
- Energieerhaltung gilt nur für konservative Vorgänge ohne Energieverluste
- Bei elastischen Stößen gelten beide Erhaltungssätze
- Bei inelastischen Stößen gilt nur die Impulserhaltung, die mechanische Energie nimmt ab
- Die Impulserhaltung liefert Vektorgleichungen (Richtungen wichtig)
- Die Energieerhaltung liefert eine Skalargleichung (nur Beträge wichtig)

Anwendungen der Impulserhaltung

Ballistisches Pendel

Ein ballistisches Pendel ist ein Gerät zur Messung der Geschwindigkeit von Projektilen. Es besteht aus einem Pendelkörper, in den das Projektil einschlägt und stecken bleibt (vollständig inelastischer Stoß).
Gegeben:

- Masse des Projektils m_K
- Masse des Pendels m_P
- Auslenkung des Pendels x
- Pendellänge L

Die Geschwindigkeit des Projektils lässt sich berechnen:

$$v_K = \frac{m_P + m_K}{m_K} \cdot \sqrt{2g \cdot (L - \sqrt{L^2 - x^2})} \tag{56}$$

Dabei wird ein zweistufiger Prozess betrachtet:
1. Inelastischer Stoß: Impulserhaltung, Energieverlust
2. Pendelschwingung: Energieerhaltung (kinetische zu potentieller Energie)

Raketenantrieb

Der Raketenantrieb basiert auf dem Rückstoßprinzip und der Impulserhaltung. Die Rakete stößt Treibstoff mit hoher Geschwindigkeit aus und erhält dadurch einen Impuls in die entgegengesetzte Richtung.
Die Raketengleichung (Ziolkowski-Gleichung) beschreibt die maximale Geschwindigkeit einer Rakete:

$$v_{\text{max}} = v_{\text{rel}} \cdot \ln \frac{m_0}{m_{\text{leer}}} \tag{57}$$

Dabei ist:

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

Stoßsimulation in Unity

```
// Elastischen Stoss zwischen zwei Objekten simulieren
void OnCollisionEnter(Collision collision) {
    Rigidbody otherRigidbody = collision.rigidbody;

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

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

    // Normale des Stosses (Verbindungslinie der Mittelpunkte)
    Vector3 normal = (otherRigidbody.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 dem Stoss entlang der Normalen
        float u1n = ((m1 - m2) * v1n + 2 * m2 * v2n) / (m1 + m2);
        float u2n = ((m2 - m1) * v2n + 2 * m1 * v1n) / (m1 + m2);

        // Geschwindigkeitsaenderung nur entlang der Normalen
        Vector3 v1Tangential = v1 - v1n * normal;
        Vector3 v2Tangential = v2 - v2n * normal;

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

        // Geschwindigkeiten setzen
        rigidbody.velocity = newV1;
        otherRigidbody.velocity = newV2;
    }
}
```

Arbeit und Energie

Arbeit

Arbeit

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

$$W = \int_{\vec{r}_0}^{\vec{r}_1} \vec{F} \cdot d\vec{r} \tag{58}$$

Die Einheit der Arbeit ist Joule (J): $1\,\text{J} = 1\,\text{N} \cdot \text{m} = 1\,\frac{\text{kg}\cdot\text{m}^2}{\text{s}^2}$
Für den Spezialfall einer konstanten Kraft in Richtung des Weges vereinfacht sich die Formel zu:

$$W = F \cdot \Delta x \tag{59}$$

Arbeit mit Winkel zwischen Kraft und Weg

Wenn die Kraft einen Winkel α mit der Wegrichtung einschließt, gilt:

$$W = F \cdot \cos(\alpha) \cdot \Delta x \tag{60}$$

Nur die Komponente der Kraft in Richtung des Weges verrichtet Arbeit.

Graphische Darstellung der Arbeit

In einem Weg-Kraft-Diagramm entspricht die Arbeit der Fläche unter der Kraft-Kurve:

$$W = \int_{x_0}^{x_1} F_x(x)\,dx \tag{61}$$

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

Formen physikalischer Arbeit

Arten von Arbeit

Verschiedene Arten physikalischer Arbeit:
- Hubarbeit: $W = m \cdot g \cdot h$
- Beschleunigungsarbeit: $W = \frac{1}{2} \cdot m \cdot (v_1^2 - v_0^2)$
- Deformationsarbeit (Federspannung): $W = \frac{1}{2} \cdot k \cdot (x_1^2 - x_0^2)$
- Reibungsarbeit: $W = \mu \cdot F_N \cdot \Delta x$

Berechnung der Arbeit in Praxisbeispielen

Hubarbeit

- Gegeben: Masse m, Höhendifferenz h
- Berechnung: $W = m \cdot g \cdot h$
- Beispiel: Ein 20 kg schwerer Körper wird 5 m angehoben: $W = 20\ \text{kg} \cdot 9.81\,\frac{\text{m}}{\text{s}^2} \cdot 5\ \text{m} = 981\ \text{J}$

Federarbeit

- Gegeben: Federkonstante k, Auslenkungen x_0 und x_1
- Berechnung: $W = \frac{1}{2} \cdot k \cdot (x_1^2 - x_0^2)$
- Beispiel: Eine Feder mit $k = 100$ N/m wird von 0 auf 0.2 m gedehnt: $W = \frac{1}{2} \cdot 100\,\frac{\text{N}}{\text{m}} \cdot (0.2^2 - 0^2)\ \text{m}^2 = 2\ \text{J}$

Reibungsarbeit

- Gegeben: Reibungskoeffizient μ, Normalkraft F_N, Strecke Δx
- Berechnung: $W = \mu \cdot F_N \cdot \Delta x$
- Beispiel: Ein 5 kg schwerer Körper wird 10 m weit gezogen mit $\mu = 0.2$: $W = 0.2 \cdot (5\ \text{kg} \cdot 9.81\,\frac{\text{m}}{\text{s}^2}) \cdot 10\ \text{m} = 98.1\ \text{J}$

Energie

Energie

Energie ist die Fähigkeit eines Systems, Arbeit zu verrichten. Die an einem Körper verrichtete Arbeit ist gleich der Änderung seiner Energie:

$$W = \Delta E = E_{nachher} - E_{vorher} \tag{62}$$

Die Einheit der Energie ist ebenfalls Joule (J).
Während Arbeit einen Prozess beschreibt (Prozessgröße), charakterisiert Energie einen Zustand (Zustandsgröße).

Energieformen

Die wichtigsten Energieformen in der Mechanik:
- Kinetische Energie (Bewegungsenergie): $E_{kin} = \frac{1}{2} \cdot m \cdot v^2$
- Potentielle Energie im Schwerefeld: $E_{pot} = m \cdot g \cdot h$
- Spannenergie einer Feder: $E_{spann} = \frac{1}{2} \cdot k \cdot x^2$
- Rotationsenergie: $E_{rot} = \frac{1}{2} \cdot J \cdot \omega^2$

Energieerhaltung

Das Prinzip der Energieerhaltung besagt, dass in einem abgeschlossenen System ohne Einwirkung nichtkonservativer Kräfte die Gesamtenergie konstant bleibt:

$$E_{ges} = E_{kin} + E_{pot} + E_{spann} + ... = \text{const.} \tag{63}$$

Bei konservativen Kräften bleibt die mechanische Energie erhalten. Bei dissipativen Kräften (wie Reibung) wird mechanische Energie in andere Energieformen (meist Wärme) umgewandelt.

Energiewandlungen beim Pendel

Ein Pendel der Masse m und Länge l zeigt deutlich die Umwandlung zwischen potentieller und kinetischer Energie:
- Am höchsten Punkt: maximale potentielle Energie, keine kinetische Energie

$$E_{pot,max} = m \cdot g \cdot h = m \cdot g \cdot l \cdot (1 - \cos\alpha) \tag{64}$$

- Am tiefsten Punkt: maximale kinetische Energie, minimale potentielle Energie

$$E_{kin,max} = \frac{1}{2} \cdot m \cdot v_{max}^2 = m \cdot g \cdot l \cdot (1 - \cos\alpha) \tag{65}$$

- Die Gesamtenergie $E_{ges} = E_{pot} + E_{kin}$ bleibt konstant (bei Vernachlässigung der Reibung)

Anwendung der Energieerhaltung zur Problemlösung

Allgemeines Vorgehen

1. Identifiziere die relevanten Energieformen im Anfangs- und Endzustand
2. Stelle die Energieerhaltungsgleichung auf (bei konservativen Systemen)
3. Berücksichtige eventuelle Energieverluste durch Reibung
4. Löse die Gleichung nach der gesuchten Größe auf

Beispiel: Ball wird von Höhe h_1 fallen gelassen und springt auf Höhe h_2

- Anfangszustand: $E_1 = m \cdot g \cdot h_1$
- Endzustand: $E_2 = m \cdot g \cdot h_2$
- Bei teilelastischem Stoß gilt: $h_2 = e^2 \cdot h_1$, wobei e der Restitutionskoeffizient ist
- Der Energieverlust beträgt: $\Delta E = m \cdot g \cdot (h_1 - h_2) = m \cdot g \cdot h_1 \cdot (1 - e^2)$

Leistung und Wirkungsgrad

Leistung

Die Leistung P ist die pro Zeiteinheit verrichtete Arbeit:

$$P = \frac{dW}{dt} \tag{66}$$

Die Einheit der Leistung ist Watt (W): $1\,\mathrm{W} = 1\,\frac{\mathrm{J}}{\mathrm{s}} = 1\,\frac{\mathrm{kg \cdot m^2}}{\mathrm{s^3}}$
Für konstante Leistung gilt:

$$W = P \cdot t \tag{67}$$

Wirkungsgrad

Der Wirkungsgrad η ist das Verhältnis von Nutzenergie zu zugeführter Energie:

$$\eta = \frac{E_{Nutz}}{E_{zugeführt}} \tag{68}$$

Der Wirkungsgrad ist eine dimensionslose Zahl zwischen 0 und 1 (oder in Prozent ausgedrückt: 0-100%).

Leistung und Wirkungsgrad einer Pumpe

Eine Pumpe hebt Wasser aus einer Tiefe von 5 m auf eine Höhe von 15 m. Sie liefert 0.2 m³ Wasser pro Minute und benötigt eine elektrische Leistung von 700 W.

- Nützliche Leistung:

$$P_{Nutz} = \frac{\Delta E_{pot}}{\Delta t} = \frac{m \cdot g \cdot \Delta h}{\Delta t} \tag{69}$$

$$= \rho \cdot V \cdot g \cdot \Delta h / \Delta t \tag{70}$$

$$= 1000\,\frac{\mathrm{kg}}{\mathrm{m^3}} \cdot 0.2\,\frac{\mathrm{m^3}}{60\ \mathrm{s}} \cdot 9.81\,\frac{\mathrm{m}}{\mathrm{s^2}} \cdot 20\ \mathrm{m} \tag{71}$$

$$\approx 654\ \mathrm{W} \tag{72}$$

- Wirkungsgrad:

$$\eta = \frac{P_{Nutz}}{P_{el}} = \frac{654\ \mathrm{W}}{700\ \mathrm{W}} \approx 0.93 = 93\% \tag{73}$$

Energie und Leistung in Unity

```
// Berechnung und Ueberwachung der Energien in Unity
void CalculateEnergies() {
    // Kinetische Energie
    float kineticEnergy = 0.5f * rigidbody.mass *
                          rigidbody.velocity.sqrMagnitude;

    // Potentielle Energie (Hoehe relativ zum Boden)
    float potentialEnergy = rigidbody.mass * 9.81f *
                            (transform.position.y - groundLevel);

    // Spannenergie (falls zutreffend)
    float springEnergy = 0.5f * springConstant *
                         (transform.position.x - equilibriumPosition)^2;

    // Gesamtenergie
    float totalEnergy = kineticEnergy + potentialEnergy + springEnergy;

    // Ueberwachen der Energieerhaltung
    if (Mathf.Abs(totalEnergy - initialTotalEnergy) > tolerance) {
        Debug.LogWarning("Energy not conserved! Check for numerical errors or
            non-conservative forces.");
    }

    // Momentane Leistung berechnen (Energieaenderung pro Zeit)
    float power = (totalEnergy - lastTotalEnergy) / Time.deltaTime;
    lastTotalEnergy = totalEnergy;
}
```

Energy and Momentum

Work and Energy

Work

Work done by a force $\vec{F}$ over displacement $\vec{s}$:

$$W = \vec{F} \cdot \vec{s} = Fs\cos\theta$$

For variable force:

$$W = \int \vec{F} \cdot d\vec{s}$$

Kinetic Energy

Energy of motion:

$$K = \frac{1}{2}mv^2$$

For rotational motion:

$$K_{rot} = \frac{1}{2}I\omega^2$$

where I is moment of inertia and ω is angular velocity.

Potential Energy

Energy stored in position or configuration:
Gravitational: $U_g = mgh$
Elastic (spring): $U_s = \frac{1}{2}kx^2$
General: $U(\vec{r}) = -\int \vec{F} \cdot d\vec{r}$

Work-Energy Theorem

The work done on an object equals its change in kinetic energy:

$$W_{net} = \Delta K = K_f - K_i$$

Conservation of Energy

Mechanical Energy Conservation

In absence of non-conservative forces:

$$E = K + U = \text{constant}$$

For conservative systems:

$$\frac{1}{2}mv_1^2 + U_1 = \frac{1}{2}mv_2^2 + U_2$$

Conservative vs. Non-Conservative Forces

- **Conservative:** Work is path-independent (gravity, spring force)
- **Non-conservative:** Work depends on path (friction, air resistance)

Energy Calculation 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()
    {
        // Calculate kinetic energy
        float kineticEnergy = 0.5f * rb.mass * rb.velocity.sqrMagnitude;

        // Calculate gravitational potential energy
        float gravitationalPE = rb.mass * Mathf.Abs(Physics.gravity.y) *
                                transform.position.y;

        // Calculate elastic potential energy
        Vector3 displacement = transform.position - equilibriumPosition;
        float elasticPE = 0.5f * springConstant * displacement.sqrMagnitude;

        // Total mechanical energy
        float totalEnergy = kineticEnergy + gravitationalPE + elasticPE;

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

Power

Power

Rate of doing work: $P = \frac{dW}{dt} = \vec{F} \cdot \vec{v}$

For rotational motion: $P = \tau\omega$

Momentum and Impulse

Linear Momentum

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

Newton's Second Law in terms of momentum: $\vec{F} = \frac{d\vec{p}}{dt}$

Impulse

Change in momentum due to force over time: $\vec{J} = \int_{t_1}^{t_2} \vec{F}\, dt = \Delta\vec{p} = m\vec{v}_f - m\vec{v}_i$

For constant force: $\vec{J} = \vec{F}\Delta t$

Conservation of Momentum

In absence of external forces, total momentum is conserved: $\sum \vec{p}_i = \sum \vec{p}_f$

For two-body system: $m_1\vec{v}_{1i} + m_2\vec{v}_{2i} = m_1\vec{v}_{1f} + m_2\vec{v}_{2f}$

Collisions

Types of Collisions

- **Elastic:** Both momentum and kinetic energy conserved
- **Inelastic:** Only momentum conserved
- **Perfectly inelastic:** Objects stick together after collision

Elastic Collision (1D)

For two objects with masses m_1, m_2 and initial velocities v_{1i}, v_{2i}:

Final velocities:

$$v_{1f} = \frac{(m_1 - m_2)v_{1i} + 2m_2v_{2i}}{m_1 + m_2}$$

$$v_{2f} = \frac{(m_2 - m_1)v_{2i} + 2m_1v_{1i}}{m_1 + m_2}$$

Collision Implementation 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;

        // Get masses and velocities before collision
        float m1 = rb.mass;
        float m2 = otherRb.mass;
        Vector3 v1i = rb.velocity;
        Vector3 v2i = otherRb.velocity;

        // Calculate velocities after elastic collision
        Vector3 v1f = ((m1 - m2) * v1i + 2 * m2 * v2i) / (m1 + m2);
        Vector3 v2f = ((m2 - m1) * v2i + 2 * m1 * v1i) / (m1 + m2);

        // Apply new velocities
        rb.velocity = v1f;
        otherRb.velocity = v2f;
    }
}
```

Angular Momentum

Angular Momentum

For point particle: $\vec{L} = \vec{r} \times \vec{p} = \vec{r} \times m\vec{v}$

For rigid body: $\vec{L} = I\vec{\omega}$

where I is moment of inertia and $\vec{\omega}$ is angular velocity.

Conservation of Angular Momentum

In absence of external torques: $\frac{d\vec{L}}{dt} = \vec{\tau}_{ext} = 0 \Rightarrow \vec{L} = \text{constant}$

Angular Momentum in Unity

```
// Calculate angular momentum of a body at point Q with respect to pivot P
Vector3 AngularMomentum(Rigidbody rb, Vector3 pivotPoint)
{
    // Vector from pivot point to center of mass
    Vector3 rPQ = rb.transform.position - pivotPoint;

    // Linear momentum of the body
    Vector3 p = rb.velocity * rb.mass;

    // Calculate and return angular momentum
    Vector3 angularMomentum = Vector3.Cross(rPQ, p);
    return angularMomentum;
}
```

Energy-Momentum Problem Solving

Step 1: Identify the system and constraints

- Define system boundaries
- Identify conservative and non-conservative forces
- Check for momentum/energy conservation conditions

Step 2: Choose appropriate conservation law

- Use energy conservation for problems involving heights, springs
- Use momentum conservation for collision problems
- Use both for complex multi-stage problems

Step 3: Set up equations

- Write initial and final energy/momentum expressions
- Apply conservation principles
- Include any additional constraints

Step 4: Solve and verify

- Solve algebraically before substituting numbers
- Check units and physical reasonableness
- Verify conservation laws are satisfied

Collision Analysis Problem

Two cars approach each other: Car A (1000 kg) at 20 m/s, Car B (1500 kg) at -15 m/s. After collision, Car A moves at 5 m/s. Find Car B's final velocity and energy lost.

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

Projectile Motion with Energy

A projectile is launched at angle $\theta = 45$ř with initial speed $v_0 = 20m/s$. Find maximum height using energy conservation.

Energy approach: At launch: $E_i = \frac{1}{2}mv_0^2$ (taking ground as reference)
At max height: $E_f = mgh_{max} + \frac{1}{2}mv_x^2$
Since $v_x = v_0 \cos\theta$ remains constant:
$\frac{1}{2}mv_0^2 = mgh_{max} + \frac{1}{2}m(v_0\cos\theta)^2$
$\frac{1}{2}v_0^2 = gh_{max} + \frac{1}{2}v_0^2\cos^2\theta$
$h_{max} = \frac{v_0^2 \sin^2\theta}{2g} = \frac{(20)^2\sin^2(45\text{ř})}{2(9.81)} = 10.2m$

Rotation

Kinematik der Rotation

Rotationsbewegung

Eine Rotationsbewegung ist die Drehung eines Körpers um eine bestimmte Achse. Die Beschreibung erfolgt analog zur Translation, jedoch werden andere Größen verwendet:

- 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.

Zusammenhänge der rotatorischen Größen

Analog zur Translation gelten folgende Beziehungen:

$$\omega = \frac{d\varphi}{dt} \tag{74}$$

$$\alpha = \frac{d\omega}{dt} \tag{75}$$

$$\varphi = \int \omega \, dt \tag{76}$$

$$\omega = \int \alpha \, dt \tag{77}$$

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: $360\check{r} = 2\pi$ rad
- Rechter Winkel: $90\check{r} = \frac{\pi}{2}$ rad

Zusammenhang zwischen Bahngeschwindigkeit und Winkelgeschwindigkeit

Bei einer Kreisbewegung mit Radius r gilt für die Bahngeschwindigkeit v:

$$v = \omega \cdot r \tag{80}$$

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

Schwerpunkt und Trägheitsmoment

Schwerpunkt

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

$$\vec{r}_S = \frac{1}{\sum_{i=1}^{n} m_i} \cdot \sum_{i=1}^{n} m_i \cdot \vec{r}_i \tag{81}$$

Für kontinuierliche Massenverteilungen:

$$\vec{r}_S = \frac{1}{M} \iiint_K \vec{r} \, \rho(\vec{r}) \, dV \tag{82}$$

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 J ist ein Maß für den Widerstand eines Körpers gegenüber Rotationsbeschleunigungen. Es ist definiert als:

$$J = \sum_{i=1}^{n} m_i \cdot r_i^2 \tag{83}$$

Für kontinuierliche Massenverteilungen:

$$J = \iiint_K r^2 \, \rho(\vec{r}) \, dV \tag{84}$$

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

Trägheitsmomente einfacher Körper

Einige wichtige Trägheitsmomente:

- Punktmasse m im Abstand r von der Drehachse: $J = m \cdot r^2$
- Dünner Stab der Länge L und Masse m (Drehachse durch Mitte, senkrecht zum Stab): $J = \frac{1}{12} \cdot m \cdot L^2$
- Dünner Stab (Drehachse am Ende): $J = \frac{1}{3} \cdot m \cdot L^2$
- Vollzylinder der Masse m und Radius R (Drehachse = Symmetrieachse): $J = \frac{1}{2} \cdot m \cdot R^2$
- Hohlzylinder (Drehachse = Symmetrieachse): $J = m \cdot R^2$
- Kugel der Masse m und Radius R (Drehachse durch Mittelpunkt): $J = \frac{2}{5} \cdot m \cdot R^2$

Steiner'scher Satz

Der Steiner'sche Satz beschreibt, wie sich das Trägheitsmoment ändert, wenn die Drehachse parallel verschoben wird:

$$J = J_S + m \cdot d^2 \tag{85}$$

Dabei ist:

- J_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 $\vec{M}$ ist die Ursache einer Rotationsbeschleunigung. Es ist definiert als das Kreuzprodukt aus Hebelarm $\vec{r}$ und Kraft $\vec{F}$:

$$\vec{M} = \vec{r} \times \vec{F} \tag{86}$$

Der Betrag des Drehmoments ist:

$$M = r \cdot F \cdot \sin(\alpha) \tag{87}$$

wobei α der Winkel zwischen Hebelarm und Kraft ist.
Alternativ:

$$M = F \cdot d \tag{88}$$

wobei d der senkrechte Abstand der Kraftwirkungslinie von der Drehachse ist.

Zweites Newton'sches Gesetz für Rotationen

Analog zum zweiten Newton'schen Gesetz für translatorische Bewegungen gilt für Rotationen:

$$\vec{M} = J \cdot \vec{\alpha} \tag{89}$$

Diese Gleichung verknüpft das Drehmoment mit der resultierenden Winkelbeschleunigung.

Drehimpuls

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

$$\vec{L} = J \cdot \vec{\omega} \tag{90}$$

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

$$\vec{L} = \vec{r} \times \vec{p} \tag{91}$$

Das Drehmoment ändert den Drehimpuls gemäß:

$$\vec{M} = \frac{d\vec{L}}{dt} \tag{92}$$

Drehimpulserhaltung

In Abwesenheit äußerer Drehmomente bleibt der Drehimpuls eines Systems konstant:

$$\vec{L} = \text{const.} \tag{93}$$

Die Drehimpulserhaltung ist neben der Impuls- und Energieerhaltung ein fundamentales Erhaltungsgesetz der Physik.
Beispiele:

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

Rotationsenergie

Die kinetische Energie der Rotation (Rotationsenergie) ist gegeben durch:

$$E_{rot} = \frac{1}{2} \cdot J \cdot \omega^2 \tag{94}$$

Die Gesamtenergie eines Körpers, der sowohl translatorische als auch rotatorische Bewegung erfährt, ist:

$$E_{ges} = E_{trans} + E_{rot} = \frac{1}{2} \cdot m \cdot v^2 + \frac{1}{2} \cdot J \cdot \omega^2 \tag{95}$$

Berechnung rotatorischer Größen

Berechnung des Schwerpunkts zusammengesetzter Körper

1. Teile den Körper in einfache Formen mit bekannten Schwerpunkten
2. Berechne die Gesamtmasse: $M = \sum m_i$
3. Berechne den Schwerpunkt: $\vec{r}_S = \frac{1}{M} \sum m_i \cdot \vec{r}_i$

Berechnung des Drehmoments

1. Identifiziere alle wirkenden Kräfte $\vec{F}_i$
2. Bestimme ihre Hebelarme $\vec{r}_i$ bezüglich der Drehachse
3. Berechne die einzelnen Drehmomente: $\vec{M}_i = \vec{r}_i \times \vec{F}_i$
4. Summiere alle Drehmomente: $\vec{M}_{ges} = \sum \vec{M}_i$

Winkelbeschleunigung aus Drehmoment

1. Berechne das resultierende Drehmoment $\vec{M}_{ges}$
2. Bestimme das Trägheitsmoment J bezüglich der Drehachse
3. Berechne die Winkelbeschleunigung: $\vec{\alpha} = \frac{\vec{M}_{ges}}{J}$

Drehmoment in Unity

```
// Drehmoment auf einen Koerper aufbringen
void ApplyTorque() {
    // Angriffspunkt relativ zum Massenmittelpunkt des Koerpers
    Vector3 relativePos = transform.TransformPoint(localForcePoint) -
                          rigidbody.worldCenterOfMass;

    // Kraft berechnen (z.B. in diesem Fall eine konstante Kraft)
    Vector3 force = new Vector3(0, 0, forceAmount);

    // Drehmoment als Kreuzprodukt aus Hebelarm und Kraft
    Vector3 torque = Vector3.Cross(relativePos, force);

    // Kraft und Drehmoment auf den Koerper anwenden
    rigidbody.AddForceAtPosition(force, transform.TransformPoint(localForcePoint));

    // Alternative: Drehmoment direkt anwenden
    // rigidbody.AddTorque(torque);
}
```

Rotation eines Würfels über eine Kante

Ein Würfel der Kantenlänge a und Masse m wird über seine Kante gestoßen.

- Trägheitsmoment bezüglich der Kante: $J = \frac{m}{6}(2a^2) = \frac{m \cdot a^2}{3}$
- Wenn der Schwerpunkt über die Kante hinausragt, wirkt ein Drehmoment aufgrund der Gewichtskraft: $M = m \cdot g \cdot \frac{a}{2} \cdot \sin\varphi$ wobei φ der Winkel zwischen der Vertikalen und der Linie vom Drehpunkt zum Schwerpunkt ist.
- Die resultierende Winkelbeschleunigung ist: $\alpha = \frac{M}{J} = \frac{3 \cdot g \cdot \sin\varphi}{2a}$
- Beim horizontalen Stoßen mit einer Kraft F entlang einer Linie, die um h vom Schwerpunkt versetzt ist, entsteht ein Drehmoment: $M = F \cdot h$
- Dies führt zu einer kombinierten Translation und Rotation des Würfels.

Orientierung im Raum und Quaternionen

Euler-Winkel

Euler-Winkel sind ein Satz von drei Winkeln, die eine Rotation im dreidimensionalen Raum vollständig beschreiben:

- Pitch (φ): Drehung um die x-Achse
- Yaw (θ): Drehung um die y-Achse
- Roll (ψ): Drehung um die z-Achse

Bei der Anwendung von Euler-Winkeln ist die Reihenfolge der Drehungen entscheidend. Außerdem muss unterschieden werden zwischen:

- Intrinsischen Drehungen: Die Achsen drehen sich mit dem Objekt
- Extrinsischen Drehungen: Die Achsen bleiben fixiert

Quaternionen in Unity

Unity verwendet intern Quaternionen zur Darstellung von Rotationen, da diese mehrere Vorteile gegenüber Euler-Winkeln bieten:

- Keine "Gimbal-LockProbleme (Verlust eines Freiheitsgrades bei bestimmten Orientierungen)
- Effizientere Interpolation zwischen Rotationen
- Numerisch stabiler

Quaternionen haben die Form $q = a + bi + cj + dk$, wobei $i^2 = j^2 = k^2 = ijk = -1$.
Eine Rotation um eine Achse $\vec{n}$ mit dem Winkel α wird als Quaternion dargestellt:

$$q = \cos\left(\frac{\alpha}{2}\right) + \sin\left(\frac{\alpha}{2}\right)(n_x i + n_y j + n_z k) \tag{96}$$

Arbeiten mit Quaternionen in Unity

Erzeugung einer Quaternion

```
// Eine Quaternion aus Euler-Winkeln erzeugen
Quaternion rotation = Quaternion.Euler(30f, 45f, 0f); // Drehung: 30 Grad um x, 45
    Grad um y

// Eine Quaternion aus einer Achse und einem Winkel erzeugen
Vector3 axis = new Vector3(0, 1, 0); // y-Achse
float angle = 45f; // 45 Grad
Quaternion fromAxisAngle = Quaternion.AngleAxis(angle, axis);
```

Anwenden einer Rotation

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

// Quaternion auf einen Vektor anwenden, um ihn zu rotieren
Vector3 rotatedVector = myQuaternion * originalVector;

// Objekt rotieren
transform.rotation = myQuaternion;

// Inkrementelle Rotation
transform.rotation = myQuaternion * transform.rotation;
```

Interpolation zwischen Rotationen

```
// Sphaerische Interpolation zwischen zwei Rotationen
Quaternion interpolated = Quaternion.Slerp(startRotation, endRotation, t);
// t ist ein Wert zwischen 0 und 1
```

Standfestigkeit und physikalisches Gleichgewicht

Standfestigkeit

Ein Körper ist standfest, wenn die Wirkungslinie der Gewichtskraft durch die Unterstützungsfläche verläuft.
Man unterscheidet drei Arten von Gleichgewicht:

- Stabiles Gleichgewicht: Nach einer kleinen Auslenkung kehrt der Körper in seine Ausgangslage zurück (potentielle Energie im Minimum)
- Labiles Gleichgewicht: Nach einer kleinen Auslenkung entfernt sich der Körper weiter von seiner Ausgangslage (potentielle Energie im Maximum)
- Indifferentes Gleichgewicht: Nach einer Auslenkung verharrt der Körper in der neuen Position (potentielle Energie konstant)

Kippkriterium

Ein Körper kippt, wenn die Wirkungslinie der Gewichtskraft außerhalb der Unterstützungsfläche verläuft.
Für einen Quader mit Grundfläche $b \times l$ und Höhe h auf einer um den Winkel α geneigten Ebene gilt als Kippkriterium:

$$\tan\alpha > \frac{b}{2h} \tag{97}$$

Dabei ist b die Breite des Quaders in Richtung der Neigung.

Standfestigkeit eines Turms

Ein Turm aus mehreren Bausteinen ist standfest, wenn der Schwerpunkt jedes Teilstücks (bestehend aus allen Steinen oberhalb eines bestimmten Punktes) über der Unterstützungsfläche liegt.
Bei einem Überhang von Bausteinen muss der Schwerpunkt der überhängenden Steine durch das Gegengewicht der nicht überhängenden Steine ausgeglichen werden, damit der gemeinsame Schwerpunkt noch über der Unterstützungsfläche bleibt.
Der maximale Überhang für n identische Bausteine, die jeweils um einen bestimmten Abstand verschoben sind, wird durch die harmonische Reihe beschrieben:

$$\text{Überhang}_{\text{max}} = \frac{L}{2} \cdot \sum_{i=1}^{n} \frac{1}{i} \tag{98}$$

wobei L die Länge eines Bausteins ist.

In Unity ist die korrekte Modellierung von Standfestigkeit und Gleichgewicht wichtig für:

- Realistische Darstellung von stehenden oder gestapelten Objekten
- Simulierte Reaktionen auf Stöße und Kräfte
- Korrekte Kippsimulationen und Fallbewegungen

Um zuverlässige Ergebnisse zu erhalten, müssen sowohl die Kollisionsgeometrie als auch die Massenverteilung (insbesondere die Position des Schwerpunkts) korrekt definiert werden.

Rotational Motion

Rotational Kinematics

Angular Quantities

- **Angular position:** θ (radians)
- **Angular velocity:** $\omega = \frac{d\theta}{dt}$ (rad/s)
- **Angular acceleration:** $\alpha = \frac{d\omega}{dt} = \frac{d^2\theta}{dt^2}$ (rad/s²)

Relationship to Linear Motion

For circular motion with radius r:

- Arc length: $s = r\theta$
- Linear velocity: $v = r\omega$
- Tangential acceleration: $a_t = r\alpha$
- Centripetal acceleration: $a_c = \frac{v^2}{r} = r\omega^2$

Rotational Kinematic Equations

For constant angular acceleration α:

Angular velocity:

$$\omega(t) = \omega_0 + \alpha t$$

Angular position:

$$\theta(t) = \theta_0 + \omega_0 t + \frac{1}{2}\alpha t^2$$

Velocity-position relation:

$$\omega^2 = \omega_0^2 + 2\alpha(\theta - \theta_0)$$

Rotational Motion in Unity

```
public class RotationalMotion : MonoBehaviour
{
    public float angularVelocity = 90f; // degrees per second
    public float angularAcceleration = 10f; // degrees per second squared

    private float currentAngularVel;

    void Start()
    {
        currentAngularVel = angularVelocity;
    }

    void Update()
    {
        // Update angular velocity
        currentAngularVel += angularAcceleration * Time.deltaTime;

        // Apply rotation
        transform.Rotate(Vector3.up, currentAngularVel * Time.deltaTime);

        // Alternative: Set angular velocity directly on Rigidbody
        // GetComponent<Rigidbody>().angularVelocity =
        //     Vector3.up * currentAngularVel * Mathf.Deg2Rad;
    }
}
```

Moment of Inertia

Moment of Inertia

Measure of an object's resistance to rotational acceleration:

$$I = \sum_i m_i r_i^2 = \int r^2 \, dm$$

where r_i is the perpendicular distance from the rotation axis.

Common Moments of Inertia

Point mass:

$$I = mr^2$$

Solid cylinder (about center axis):

$$I = \frac{1}{2}mr^2$$

Solid sphere (about center):

$$I = \frac{2}{5}mr^2$$

Thin rod (about center):

$$I = \frac{1}{12}ml^2$$

Thin rod (about end):

$$I = \frac{1}{3}ml^2$$

Parallel Axis Theorem

For axis parallel to center-of-mass axis:

$$I = I_{cm} + md^2$$

where d is the distance between axes.

Rotational Dynamics

Newton's Second Law for Rotation

$$\sum \tau = I\alpha$$

where τ is torque, I is moment of inertia, and α is angular acceleration.

Torque

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

Magnitude: $\tau = rF\sin\phi$
where ϕ is angle between $\vec{r}$ and $\vec{F}$.

Calculating Torque in Unity

```
public class TorqueCalculation : MonoBehaviour
{
    public Transform forceApplication; // Point where force is applied
    public Vector3 forceVector = Vector3.forward;

    void Update()
    {
        // Calculate position vector from rotation center to force application
        Vector3 leverArm = forceApplication.position - transform.position;

        // Calculate torque using cross product
        Vector3 torque = Vector3.Cross(leverArm, forceVector);

        // Apply torque to rigidbody
        GetComponent<Rigidbody>().AddTorque(torque);

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

Rotational Energy

Rotational Kinetic Energy

$$K_{rot} = \frac{1}{2} I \omega^2$$

For rolling motion (combined translation and rotation):

$$K_{total} = \frac{1}{2} m v_{cm}^2 + \frac{1}{2} I_{cm} \omega^2$$

Rolling Without Slipping

Condition: $v_{cm} = R\omega$
where R is radius and v_{cm} is center-of-mass velocity.
Total kinetic energy: $K = \frac{1}{2} m v^2 (1 + \frac{I}{mR^2})$

Work and Power in Rotation

Rotational Work

$$W = \int \tau \, d\theta$$

For constant torque: $W = \tau \Delta\theta$

Rotational Power

$$P = \tau \omega$$

Analogous to $P = Fv$ for linear motion.

Work-Energy Theorem for Rotation

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

Energy Analysis for Rotating Object

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

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

    void Update()
    {
        // Calculate rotational kinetic energy
        // Unity stores angular velocity in rad/s
        float angularSpeed = rb.angularVelocity.magnitude;

        // Approximate moment of inertia (for sphere)
        float momentOfInertia = 0.4f * rb.mass *
                                Mathf.Pow(GetComponent<SphereCollider>().radius, 2);

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

        // Translational kinetic energy
        float translationalKE = 0.5f * rb.mass * rb.velocity.sqrMagnitude;

        // Total kinetic energy
        float totalKE = rotationalKE + translationalKE;

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

Angular Momentum

Angular Momentum for Rigid Body

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

For point particle:

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

Conservation of Angular Momentum

When net external torque is zero:

$$\frac{d\vec{L}}{dt} = \vec{\tau}_{ext} = 0 \Rightarrow \vec{L} = \text{constant}$$

Applications: spinning ice skater, planetary motion, gyroscopes

Rotational Problem Solving

Step 1: Identify rotation axis

- Fixed axis (door hinge) or moving axis (rolling ball)
- Choose coordinate system with axis perpendicular to motion plane

Step 2: Calculate moment of inertia

- Use standard formulas for common shapes
- Apply parallel axis theorem if needed
- For composite objects, sum individual moments

Step 3: Analyze forces and torques

- Calculate torque: $\tau = rF\sin\phi$
- Apply $\sum \tau = I\alpha$ for dynamics
- Consider constraints (rolling, fixed axis, etc.)

Step 4: Apply conservation laws

- Energy conservation for rolling down inclines
- Angular momentum conservation for isolated systems
- Combined linear and angular analysis for complex motion

Rolling Sphere Down Incline

A solid sphere (radius R, mass m) rolls without slipping down an incline of angle θ. Find the acceleration of the center of mass.

Given: $I = \frac{2}{5}mR^2$ for solid sphere, rolling condition $a = R\alpha$

Force analysis:

- Down incline: $mg\sin\theta - f = ma$
- Torque about center: $fR = I\alpha = \frac{2}{5}mR^2 \cdot \frac{a}{R}$

From torque equation: $f = \frac{2}{5}ma$

Substituting into force equation: $mg\sin\theta - \frac{2}{5}ma = ma$

$mg\sin\theta = ma(1 + \frac{2}{5}) = \frac{7}{5}ma$

$a = \frac{5g\sin\theta}{7} = 0.714 \times g\sin\theta$

Note: This is less than $g\sin\theta$ for sliding because some energy goes into rotation.

Torque Application in Unity

Create a Unity script to simulate a door opening with applied torque, including realistic damping.

```
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;

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

        // Apply damping torque
        Vector3 dampingTorque = -dampingCoefficient * rb.angularVelocity;
        rb.AddTorque(dampingTorque);

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

Advanced Topics and Unity Implementation

Unity Physics System

Unity Physics Pipeline

Unity's physics simulation runs in discrete steps:

- `FixedUpdate()`: Called at fixed intervals (default 50Hz)
- Physics calculations performed between FixedUpdate calls
- `Update()`: Called once per frame (variable rate)
- Use FixedUpdate for physics-related code

Rigidbody Component Properties

- **Mass:** Object's mass in kg
- **Drag:** Linear damping coefficient
- **Angular Drag:** Rotational damping coefficient
- **Use Gravity:** Whether object responds to global gravity
- **Is Kinematic:** Physics-controlled vs. script-controlled

Unity Physics Settings

```
public class PhysicsConfiguration : MonoBehaviour
{
    void Start()
    {
        // Global physics settings
        Physics.gravity = new Vector3(0, -9.81f, 0);
        Time.fixedDeltaTime = 0.02f; // 50 Hz physics update

        // Rigidbody configuration
        Rigidbody rb = GetComponent<Rigidbody>();
        rb.mass = 1.0f;
        rb.drag = 0.1f;          // Linear damping
        rb.angularDrag = 0.05f;  // Angular damping
        rb.useGravity = true;
        rb.isKinematic = false;

        // Collision detection mode
        rb.collisionDetectionMode = CollisionDetectionMode.Continuous;
    }
}
```

Force Application Methods

Unity Force Application

- **AddForce():** Continuous force application
- **AddForceAtPosition():** Force at specific world point
- **AddTorque():** Rotational force application
- **AddExplosionForce():** Radial force from explosion center

Comprehensive Force Controller

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

    [Header("Force Parameters")]
    public float thrustForce = 100f;
    public float torqueStrength = 50f;
    public float explosionForce = 500f;
    public float explosionRadius = 10f;

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

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

    void HandleMovement()
    {
        // Thrust force in forward direction
        if (Input.GetKey(KeyCode.W))
        {
            Vector3 thrust = transform.forward * thrustForce;
            rb.AddForce(thrust, ForceMode.Force);
        }

        // Side thrust
        if (Input.GetKey(KeyCode.A))
        {
            rb.AddForce(-transform.right * thrustForce * 0.5f);
        }
        if (Input.GetKey(KeyCode.D))
        {
            rb.AddForce(transform.right * thrustForce * 0.5f);
        }
    }

    void HandleRotation()
    {
        // Yaw control
        if (Input.GetKey(KeyCode.Q))
        {
            rb.AddTorque(-transform.up * torqueStrength);
        }
        if (Input.GetKey(KeyCode.E))
        {
            rb.AddTorque(transform.up * torqueStrength);
        }
    }

    void HandleSpecialForces()
    {
        // Explosion force
        if (Input.GetKeyDown(KeyCode.Space))
        {
            Vector3 explosionCenter = transform.position - Vector3.up * 2f;
            rb.AddExplosionForce(explosionForce, explosionCenter, explosionRadius);
        }

        // Force at specific position (creates both force and torque)
        if (Input.GetKey(KeyCode.F))
        {
            Vector3 forcePosition = transform.position + transform.right;
            Vector3 force = transform.up * thrustForce;
```

Collision Detection and Response

Unity Collision Events

- **OnCollisionEnter():** When collision starts
- **OnCollisionStay():** While collision continues
- **OnCollisionExit():** When collision ends
- **OnTriggerEnter():** For trigger colliders (no physics response)

Advanced Collision Handler

```
public class CollisionAnalyzer : MonoBehaviour
{
    [Header("Collision Settings")]
    public float restitution = 0.8f; // Coefficient of restitution
    public float minVelocityForSound = 2f;
    public AudioClip collisionSound;

    private Rigidbody rb;
    private AudioSource audioSource;

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

    void OnCollisionEnter(Collision collision)
    {
        // Calculate collision details
        Vector3 relativeVelocity = rb.velocity -
                                   (collision.rigidbody?.velocity ?? Vector3.zero);
        float collisionSpeed = relativeVelocity.magnitude;

        // Play sound based on collision intensity
        if (collisionSpeed > minVelocityForSound && audioSource && collisionSound)
        {
            float volume = Mathf.Clamp01(collisionSpeed / 10f);
            audioSource.PlayOneShot(collisionSound, volume);
        }

        // Custom collision response
        if (collision.gameObject.CompareTag("Bouncy"))
        {
            ApplyBouncyCollision(collision, relativeVelocity);
        }

        // Log collision data for analysis
        LogCollisionData(collision, collisionSpeed);
    }

    void ApplyBouncyCollision(Collision collision, Vector3 relativeVelocity)
    {
        // Get collision normal
        Vector3 normal = collision.contacts[0].normal;

        // Calculate reflection velocity
        Vector3 reflectedVelocity = Vector3.Reflect(relativeVelocity, normal);

        // Apply restitution
        rb.velocity = reflectedVelocity * restitution;
    }

    void LogCollisionData(Collision collision, float speed)
    {
        Debug.Log($"Collision with {collision.gameObject.name}: " +
                  $"Speed: {speed:F2} m/s, " +
                  $"Contact points: {collision.contactCount}");
    }
}
```

Custom Physics Implementation

Numerical Integration Methods

Unity uses implicit Euler integration, but custom implementations may use:

- **Explicit Euler:** Simple but potentially unstable
- **Verlet Integration:** Better energy conservation
- **Runge-Kutta:** Higher accuracy for complex systems

Custom Physics Integrator

```
public class CustomPhysicsObject : MonoBehaviour
{
    [Header("Physics Properties")]
    public float mass = 1f;
    public Vector3 velocity = Vector3.zero;
    public Vector3 acceleration = Vector3.zero;
    public bool useGravity = true;

    [Header("Integration Settings")]
    public enum IntegrationMethod { Euler, Verlet, RungeKutta }
    public IntegrationMethod method = IntegrationMethod.Verlet;

    private Vector3 previousPosition;
    private Vector3 currentPosition;

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

    void FixedUpdate()
    {
        // Calculate forces
        Vector3 totalForce = CalculateForces();
        acceleration = totalForce / mass;

        // Integrate based on selected method
        switch (method)
        {
            case IntegrationMethod.Euler:
                EulerIntegration();
                break;
            case IntegrationMethod.Verlet:
                VerletIntegration();
                break;
            case IntegrationMethod.RungeKutta:
                RungeKuttaIntegration();
                break;
        }

        // Update transform
        transform.position = currentPosition;
    }

    Vector3 CalculateForces()
    {
        Vector3 totalForce = Vector3.zero;

        // Gravity
        if (useGravity)
        {
            totalForce += mass * Physics.gravity;
        }

        // Spring force to origin (example)
        totalForce += -50f * currentPosition;

        // Damping
        totalForce += -2f * velocity;

        return totalForce;
    }

    void EulerIntegration()
    {
        float dt = Time.fixedDeltaTime;

        velocity += acceleration * dt;
```

Performance Optimization

Physics Performance Tips

- Use appropriate collision detection modes
- Minimize active Rigidbodies
- Use object pooling for projectiles
- Optimize collision meshes
- Adjust Fixed Timestep appropriately

Physics Object Pool

```
public class PhysicsObjectPool : MonoBehaviour
{
    [Header("Pool Settings")]
    public GameObject prefab;
    public int poolSize = 100;

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

    void Start()
    {
        // Pre-instantiate pool objects
        for (int i = 0; i < poolSize; i++)
        {
            GameObject obj = Instantiate(prefab);
            obj.SetActive(false);
            pool.Enqueue(obj);
        }
    }

    public GameObject GetObject(Vector3 position, Vector3 velocity)
    {
        GameObject obj;

        if (pool.Count > 0)
        {
            obj = pool.Dequeue();
        }
        else
        {
            // Pool exhausted, create new object
            obj = Instantiate(prefab);
        }

        // Initialize object
        obj.transform.position = position;
        obj.SetActive(true);

        Rigidbody rb = obj.GetComponent<Rigidbody>();
        rb.velocity = velocity;
        rb.angularVelocity = Vector3.zero;

        activeObjects.Add(obj);
        return obj;
    }

    public void ReturnObject(GameObject obj)
    {
        if (activeObjects.Contains(obj))
        {
            activeObjects.Remove(obj);
            obj.SetActive(false);

            // Reset physics state
            Rigidbody rb = obj.GetComponent<Rigidbody>();
            rb.velocity = Vector3.zero;
            rb.angularVelocity = Vector3.zero;

            pool.Enqueue(obj);
        }
    }

    void Update()
    {
        // Auto-return objects that fall below world
        for (int i = activeObjects.Count - 1; i >= 0; i--)
        {
            if (activeObjects[i].transform.position.y < -50f)
            {
```

Exam Preparation

Unity Physics Exam Strategy

Core Physics Concepts

- Master Newton's laws and their applications
- Understand energy and momentum conservation
- Practice rotational dynamics calculations
- Know key formulas and their derivations

Unity Implementation Knowledge

- Rigidbody component properties and methods
- Force application techniques and force modes
- Collision detection and response
- Vector operations and coordinate systems

Problem-Solving Approach

- Identify physical principles involved
- Set up coordinate system and free-body diagrams
- Apply conservation laws where appropriate
- Translate to Unity implementation concepts

Common Exam Topics

- Projectile motion with Unity vectors
- Collision analysis and momentum conservation
- Rotational motion and torque calculations
- Energy conservation in mechanical systems

Comprehensive Physics Problem

A ball (mass 1 kg, radius 0.1 m) rolls down a 30° incline, then undergoes an elastic collision with a stationary ball of equal mass. Calculate the final velocities and implement the simulation in Unity.

Part 1: Rolling down incline

For solid sphere: $I = \frac{2}{5}mr^2$, $a = \frac{5g\sin\theta}{7}$

$a = \frac{5\times 9.81\times \sin(30\check{r})}{7} = 3.51\,m/s^2$

After rolling distance $d = 2m$: $v = \sqrt{2ad} = \sqrt{2\times 3.51\times 2} = 3.74\,m/s$

Part 2: Elastic collision

For equal masses in 1D elastic collision: $v_1' = 0$, $v_2' = v_1 = 3.74\,m/s$

Unity Implementation:

```
public class RollingBallSimulation : MonoBehaviour
{
    public float inclineAngle = 30f;
    public float ballMass = 1f;
    public float ballRadius = 0.1f;

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

        Debug.Log($"Rolling acceleration: {rollingAccel:F2} m/s^2");

        // Set up physics simulation
        SetupIncline();
        SetupBalls();
    }

    void SetupIncline()
    {
        // Create inclined plane
        GameObject incline = GameObject.CreatePrimitive(PrimitiveType.Cube);
        incline.transform.localScale = new Vector3(5, 0.1f, 1);
        incline.transform.rotation = Quaternion.Euler(0, 0, -inclineAngle);

        // Add physics material for rolling
        PhysicMaterial rollMaterial = new PhysicMaterial("Rolling");
        rollMaterial.staticFriction = 1f;
        rollMaterial.dynamicFriction = 1f;
        rollMaterial.frictionCombine = PhysicMaterialCombine.Maximum;

        incline.GetComponent<Collider>().material = rollMaterial;
    }

    void SetupBalls()
    {
        // Create rolling ball
        GameObject ball1 = GameObject.CreatePrimitive(PrimitiveType.Sphere);
        ball1.transform.position = new Vector3(-2, 2, 0);
        ball1.transform.localScale = Vector3.one * ballRadius * 2;

        Rigidbody rb1 = ball1.AddComponent<Rigidbody>();
        rb1.mass = ballMass;

        // Create stationary target ball
        GameObject ball2 = GameObject.CreatePrimitive(PrimitiveType.Sphere);
        ball2.transform.position = new Vector3(2, 0, 0);
        ball2.transform.localScale = Vector3.one * ballRadius * 2;

        Rigidbody rb2 = ball2.AddComponent<Rigidbody>();
        rb2.mass = ballMass;

        // Add collision handler for elastic collision
        ball1.AddComponent<ElasticCollisionHandler>();
        ball2.AddComponent<ElasticCollisionHandler>();
```

> Remember that Unity's physics system handles many complexities automatically, but understanding the underlying physics principles is crucial for creating realistic and predictable behavior. The exam will focus on your understanding of these fundamental concepts rather than Unity-specific implementation details.