

13. Computer Animation

Arten von Animation

Flipbook



https://youtu.be/p3q9MM_h-M

2D Animation



<https://tenor.com/bV6ph.gif>

Zoetrope



Stop Motion



■ Artistic Control vs. Automation



Time consuming, but
flexible

Visual Effects



3D Animation



Transformationen

[EVC_Skriptum_CG](#), p.55

- Eine affine Transformation, die sowohl eine Translation als auch eine Rotation beinhaltet, kann als homogene 4×4 Matrix dargestellt werden (siehe [3. Transformationen](#)):
 - $\begin{pmatrix} R & x \\ 0 & 1 \end{pmatrix}$, mit $R \in \mathbb{R}^{3 \times 3}$ als Rotationale Komponente und Translation $x \in \mathbb{R}^3$.

Translation

[EVC_Skriptum_CG, p.55](#)

- Angenommen, wir haben keine Rotation, dann kann ein Objekt mit der initialen Position $p^{(t_0)} \in \mathbb{R}^3$ zum Zeitpunkt t_0 zu einer Zielposition $p^{(t_1)}$ zum Zeitpunkt t_1 bewegt werden mittels:

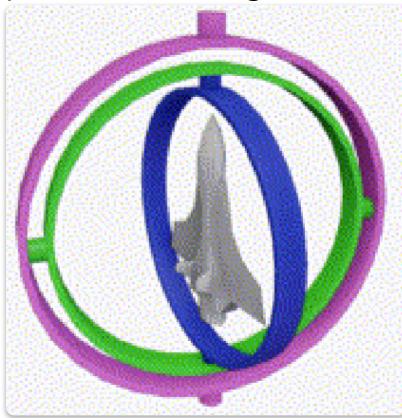
$$\mathbf{p}^{(t_1)} = \begin{bmatrix} 1 & \mathbf{x} \\ 0 & 1 \end{bmatrix} \mathbf{p}^{(t_0)}$$

- Die Position des Objekts zu einem beliebigen Zeitpunkt $t \in \mathbb{R}$, mit $t_0 < t < t_1$, kann zwischen der Start- und Endposition linear interpoliert werden:

$$p^{(t)} = p^{(t_0)} + (p^{(t_1)} - p^{(t_0)}) \frac{t - t_0}{t_1 - t_0}$$

[Rotation](#)[EVC_Skriptum_CG, p.55](#)

- Matrixdarstellungen von Rotationen** sind eine instabile Darstellung und bereiten Probleme, wie z.B. das *Gimbal Lock*. Normalerweise hat man drei Freiheitsgrade, wenn man ein Objekt im 3D-Raum rotiert.
- Verwendet man jedoch Matrizen für Rotationen, gibt es Konfigurationen, bei denen ein Objekt so gedreht wird, dass zwei Achsen parallel sind.
- Im Bild rechts zum Beispiel verklemmen sich der pinkfarbene und der grüne Kreis, und es macht keinen Unterschied mehr, ob man den inneren blauen Ring oder den äußeren pinkfarbenen Ring dreht. Somit haben wir einen Freiheitsgrad verloren.



- Eine Lösung besteht darin, **Quaternionen** zur Darstellung von Rotationen und *sphärische Interpolation* zu verwenden.
- Quaternionen sind eine Erweiterung der komplexen Zahlen und bestehen aus einer skalaren Komponente $s \in \mathbb{R}$ und einer Vektor-Komponente $\mathbf{v} \in \mathbb{R}^3$.
- Um eine beliebige Achse $\mathbf{n} \in \mathbb{R}^3$ um einen Winkel ϕ zu rotieren, kann diese Achsenwinkel-Darstellung einer Rotation in ein Quaternion q wie folgt umgewandelt werden:

- $$q = [s; \mathbf{v}] = [\cos \frac{\phi}{2}; \sin \frac{\phi}{2} \mathbf{n}]$$

(Hierbei ist n ein Einheitsvektor der Rotationsachse.)

- $q^{(t)} = \text{slerp}(q^{(t_0)}, q^{(t_1)}, t) = q^{(t_0)}(q^{(t_0)})^{-1}q^{(t_1)})^t$

(slerp steht für *spherical linear interpolation* und interpoliert zwischen zwei Quaternionen $q^{(t_0)}$ und $q^{(t_1)}$ über die Zeit t .)

- Um schließlich eine rotierte Position $P^{(t)}$ zu einem beliebigen Zeitpunkt t zu berechnen, wenden wir die interpolierte Rotation $q^{(t)}$ auf die Ausgangsposition $P^{(t_0)}$ an.
- Dazu bilden wir zuerst ein neues Quaternion, indem wir $P^{(t_0)}$ als Vektor-Komponente und Null als skalare Komponente nehmen.
- Das Ergebnis des Produkts mit der berechneten Rotation $q^{(t)}$ ist wiederum ein Quaternion mit Null als skalarer Komponente, und die Vektor-Komponente ist unsere gewünschte, rotierte Position $P^{(t)}$:

$$[0; P^{(t)}] = q^{(t)} \cdot [0; P^{(t_0)}] \cdot (q^{(t)})^{-1}$$

- Matrix representation of rotations is often unstable
 - Additional Problem: *Gimbal Lock*
- Best use **Quaternions**



$$\mathbf{q} = [s; \mathbf{v}] = [s \quad \underbrace{\begin{matrix} x & y & z \end{matrix}}_{\text{vector}}]$$

scalar

- Generalization of complex numbers
- Also written as:

$$\mathbf{q} = s + xi + yj + zk$$

with

$$i^2 = j^2 = k^2 = -1$$

$$i \cdot j \cdot k = -1$$

■ **Quaternions:** $\mathbf{q} = [s; \mathbf{v}] = [s \quad x \quad y \quad z]$



■ Operations:

■ Addition: $\mathbf{q}_1 + \mathbf{q}_2 = [(s_1 + s_2) \quad (x_1 + x_2) \quad (y_1 + y_2) \quad (z_1 + z_2)]$

■ Multiplication: $\mathbf{q}_1 \cdot \mathbf{q}_2 = \begin{bmatrix} (s_1 \cdot s_2 - x_1 \cdot x_2 - y_1 \cdot y_2 - z_1 \cdot z_2) \\ (s_1 \cdot s_2 + x_1 \cdot x_2 + y_1 \cdot y_2 - z_1 \cdot z_2) \\ (s_1 \cdot s_2 - x_1 \cdot x_2 + y_1 \cdot y_2 + z_1 \cdot z_2) \\ (s_1 \cdot s_2 + x_1 \cdot x_2 - y_1 \cdot y_2 + z_1 \cdot z_2) \end{bmatrix}$

■ Inverse of normalized Quaternion: $\mathbf{q}^{-1} = \bar{\mathbf{q}} = [s \quad -x \quad -y \quad -z]$

■ Can be computed from angle ϕ and normalized vector \mathbf{n} (*axis-angle representation*):

$$\mathbf{q} = \left[\cos \frac{\phi}{2}; \sin \frac{\phi}{2} \mathbf{n} \right]$$

■ Spherical Interpolation:

$$\mathbf{q}^{(t)} = \text{slerp}(\mathbf{q}^{(t_0)}, \mathbf{q}^{(t_1)}, t) = \mathbf{q}^{(t_0)} (\mathbf{q}^{(t_0)}{}^{-1} \mathbf{q}^{(t_1)})^t$$

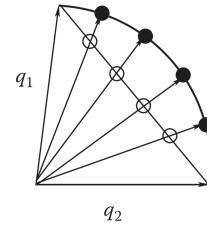


Figure taken from Shirley&Marshner:
Foundations of Computer Graphics, 5th Edition

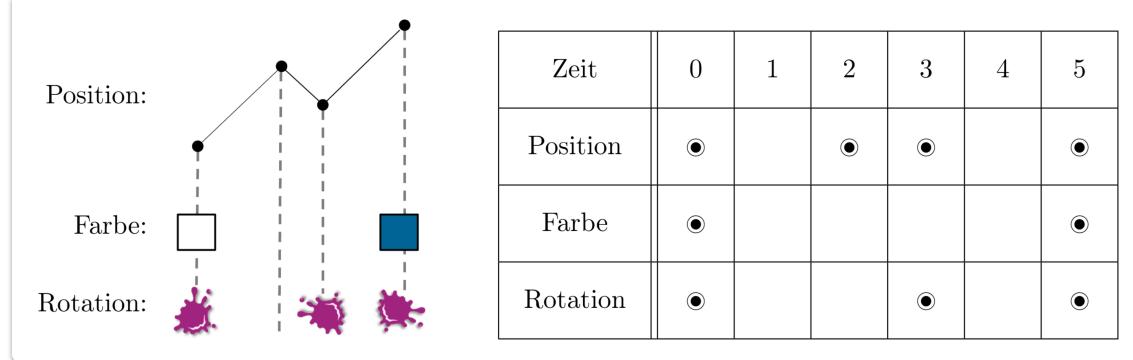
■ Apply a rotation to a point $\mathbf{p}^{(t)}$:

$$[0; \mathbf{p}^{(t)}] = \mathbf{q}^{(t)} \cdot [0; \mathbf{p}^{(t_0)}] \cdot \mathbf{q}^{(t)}{}^{-1}$$

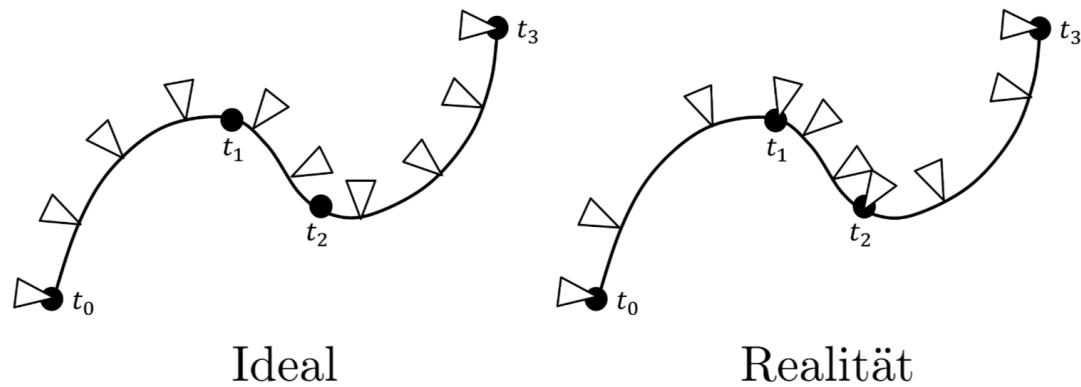
Keyframing

EVC_Skriptum_CG, p.55, p.56

- **Keyframing** ermöglicht komplexe Animationen mit einem Gleichgewicht zwischen Automatisierung und manueller Spezifikation.
- Szenenparameter werden nur zu bestimmten Zeitpunkten (**Keyframes**) festgelegt und sonst interpoliert.
- Parameteränderungen im Laufe der Zeit können in einer Tabelle kodiert werden.
- Die Zeitschritte bezeichnen wir als **Frames**.
- Jede Kombination aus einer **Zeit t** und einer Menge an **Szenenparametern f** zu diesem Zeitpunkt nennen wir einen **Keyframe k**.
- In unserem Fall besteht diese Menge von Szenenparametern f aus der Position p, Farbe c und einer Rotation, die durch ein Quaternion q dargestellt wird.
- Dies ergibt das Keyframe k: $(t_k, f^{(t_k)}) = (t_k, (p^{(t_k)}, c^{(t_k)}, q^{(t_k)}))$.
- Es kann auch nur eine **Teilmenge** der Szenenparameter angegeben werden, wie im Beispiel für Frame 2 und 3.



- Das Anpassen einer interpolierenden Kurve durch alle Keyframe-Positionen führt zu einer flüssigeren Bewegung anstelle einer linear Positionsänderung.
- Je nach verwendeter Methode zur Konstruktion der Kurve können jedoch *plötzliche Sprünge* um die Keyframe-Positionen herum auftreten und die Geschwindigkeit, die das Objekt entlang der Kurve hat, kann *unregelmäßig* verlaufen (schlecht für eine gleichmäßige Kamerabewegung).
- Idealerweise würden wir die Zeit gleichmäßig abtasten und erwarten, dass die resultierenden Punkte entlang der Kurve ebenfalls gleichmäßig verteilt sind.
- Die meisten Kurvenberechnungsverfahren gruppieren jedoch die räumlichen Abtastpunkte nicht gleichmäßig.
- Problem:** Ungleichmäßige räumliche Verteilung der Abtastpunkte entlang der Kurve.
- Lösung für dieses Problem:** Die Zeit nicht gleichmäßig abtasten. Stattdessen bestimmen wir die *Länge der Kurve* (Bogenlänge) näherungsweise und teilen diese dann gleichmäßig ab. Dies bedeutet, dass wir die Kurve in Abschnitte gleicher Länge unterteilen.

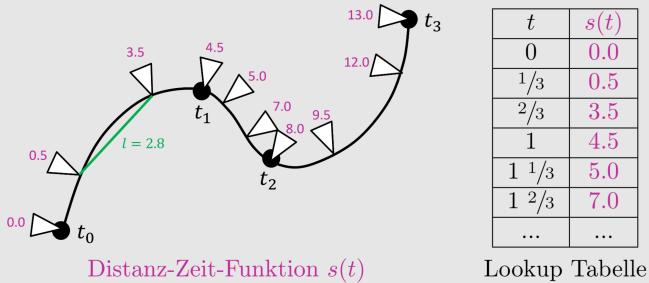


- Um dies zu erreichen, führen wir die **Distanz-Zeit-Funktion** $s(t) : \mathbb{R} \rightarrow \mathbb{R}$ ein.
- Sie gibt im Grunde an, wie weit wir entlang einer Kurve bereits gereist sind, d.h. sie ordnet jedem Zeitpunkt die bis dahin zurückgelegte Entfernung zu.
- Während $s(t)$ ursprünglich kontinuierlich ist, können wir sie diskretisieren und die Entfernung zwischen zwei Zeitpunkten durch euklidischen Distanz approximieren.
- Somit approximieren wir die Form der Kurve mittels mehrerer linearer Abschnitte:
 - $s(t_i) \approx s(t_{i-1}) + \|P_i - P_{i-1}\|$ (Die Distanz zum Zeitpunkt t_i ist die Summe der vorherigen Distanz und der euklidischen Distanz zwischen dem aktuellen und vorherigen Punkt.)

- Wir können diese approximierten Werte unserer Distanz-Zeit-Funktion s in einer *Lookup-Zeit-Tabelle* speichern.
- Wenn wir dann $s(t)$ gleichmäßig abtasten, berechnen wir die Werte t , die wir benötigen, um unsere Kurve unter Verwendung dieser Lookup-Tabelle und linearer Interpolation auszuwerten.
- Durch nicht gleichmäßiges Abtasten der Zeit haben wir somit eine *ungefähr gleichmäßige Raumabtastung* erreicht.

Beispiel:

Im Folgenden nehmen wir an, dass wir eine Methode haben, um die **genaue Distanz-Zeit-Funktion** zu berechnen, was zu den **rosa Werten** im untenstehenden Bild führt. Ein Beispiel zur Berechnung einer Approximation von $s(t)$ zur Zeit $t = 0.6$ ist rechts, wobei die **euklidische Distanz** zwischen $\mathbf{p}_{0.3}$ und $\mathbf{p}_{0.6}$ verwendet wird, von der wir ebenfalls annehmen, dass sie gegeben ist.

Approximiere $s(t)$:

$$s(t_{0.6}) \approx s(t_{0.3}) + l = 0.5 + 2.8 = 3.3$$

Indem wir unsere Approximation von 3.3 mit dem richtigen Wert von 3.5 aus der Abbildung rechts vergleichen, stellen wir fest, dass wir einen Fehler 0.2 gemacht haben.

Als nächstes **tasten wir $s(t)$ gleichmäßig ab, z.B. bei 0.0, 1.0, 2.0, ... und berechnen mittels linear Interpolation die Werte t , an denen wir unsere Kurve auswerten wollen**. Zum Beispiel, wenn wir die Position eines Punktes entlang der Kurve nach der Distanz $s = 6.0$ berechnen wollen, benötigen wir dafür:

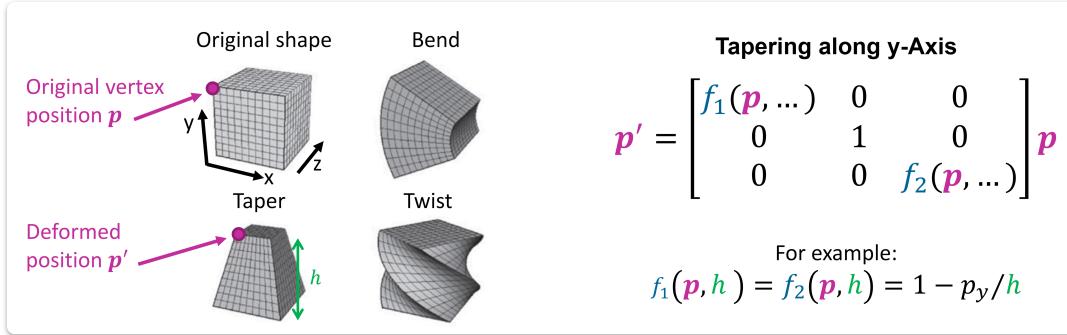
$$t = \frac{1.3 + 1.6}{2} = 2.5, \quad \text{da } 6.0 \text{ in der Mitte von } 5.0 \text{ und } 7.0 \text{ ist (siehe Tabelle).}$$

Deformationen

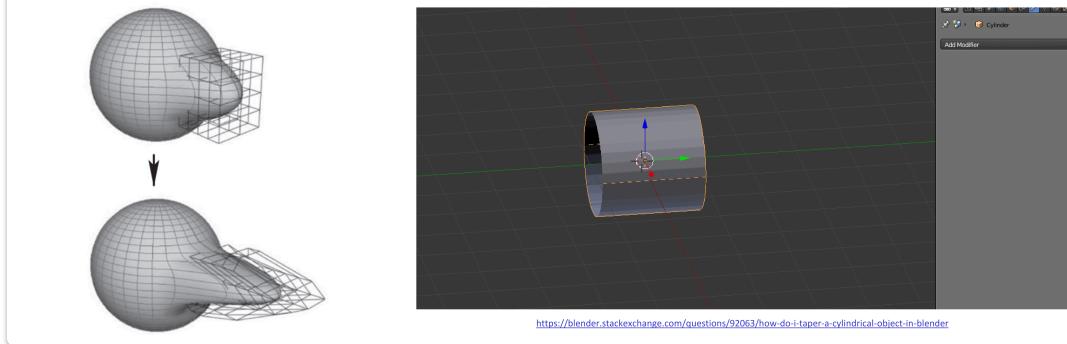
[EVC_Skriptum_CG, p.57](#)

- **Definition:** Eine Deformation kann als Funktion f definiert werden, die auf jede Vertex-Position p angewendet wird.
- Ziel ist es, die neue, deformierte Position $p' = f(p, \gamma)$ zu erhalten.
- γ sind optionale Parameter, die die Deformation steuern.
- **Einfache Deformationen:**
 - Können in Form von *nicht konstanten Matrizen* dargestellt werden.
 - Beispiele:
 - **Verjüngungs-/Taper-Operationen** (Verjüngung eines Objekts)
 - **Biege-/Bend-Operationen** (Biegen eines Objekts)
 - **Verdrehungs-/Twist-Operationen** (Verdrehen eines Objekts)
- **Komplexere Deformationen:**
 - Eine mögliche Lösung ist die Verwendung eines **Deformationskäfigs**.
 - **Vorteil:** Reduziert die Anzahl der manuell festzulegenden Vertex-Positionen erheblich.
 - **Funktionsweise:** Die Vertices (Eckpunkte) des Objekts sind an den Käfig "gebunden".

- **Effekt:** Immer wenn der Käfig bewegt wird, ändert sich auch die Position der Objekt-Vertices.



Free Form Deformation: Define f wrt. a deformation cage



Physikbasierende Simulation

[EVC_Skriptum_CG](#), p.57

Hierbei wird ein Objekt anhand physikalischer Gesetze simuliert und durch mehrere Punkte beschrieben, die über Bedingungen (engl. *Constraints*) verbunden sind.

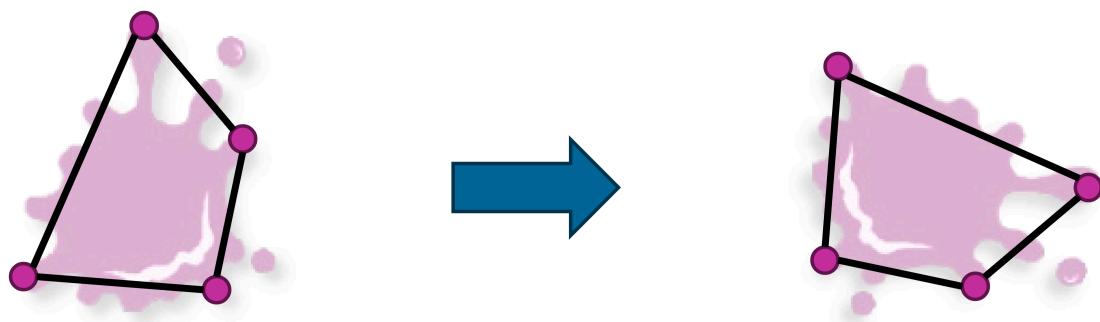
Punkt:

- **Position** $p \in \mathbb{R}^3, [m]$
- **Geschwindigkeit** $v = \frac{dp}{dt} \in \mathbb{R}^3, [m \cdot s^{-1}]$
- **Beschleunigung** $a = \frac{d^2p}{dt^2} \in \mathbb{R}^3, [m \cdot s^{-2}]$
- **Masse** $m \in \mathbb{R}, [kg]$

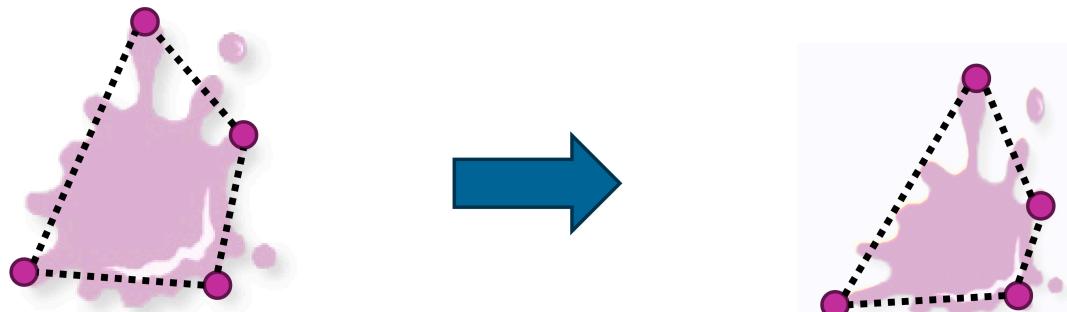
Einschränkung:

- **Starr:** Distanz/Volumen bleibt gleich, nur Transformationen (Holzwürfel, Stahlschwert) (engl. *rigid*)
- **Weich:** Distanz/Volumen variabel, Transformationen & Deformationen (Kleidung, weiches Gewebe) (engl. *soft*)

Unser Hauptinteresse besteht darin, die **Position** p zu simulieren. Dazu betrachten wir ebenfalls die Veränderung der Position über die Zeit, also die **Geschwindigkeit** eines Punktes.



- Distance stays the same
- Transformations only



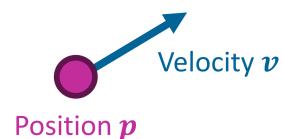
- Distance can change
- Transformations + Deformations

Bewegung bei konstanter Geschwindigkeit

Wenn die Geschwindigkeit v konstant ist (d.h. sie hängt nicht von der Zeit ab), können wir die zukünftige Position $\mathbf{p}^{(t+\Delta t)}$ eines Punktes nach einem beliebigen Zeitschritt Δt basierend auf der aktuellen Position $\mathbf{p}^{(t)}$ zur Zeit t wie folgt berechnen:

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}$$

- Represent by multiple points
- Velocity \mathbf{v} is the change of position over time



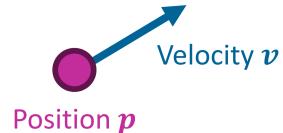
- If there is *constant* velocity, we can thus change the position via:

$$\mathbf{p}^{(t+1)} = \mathbf{p}^{(t)} + \frac{d\mathbf{p}^{(t)}}{dt} = \mathbf{p}^{(t)} + \mathbf{v}$$

Auf diese Weise erfolgt die Bewegung des Punktes mit konstanter Geschwindigkeit entlang einer geraden Linie (**Newtons erstes Gesetz**).

■ With *constant velocity*:

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}$$



[https://commons.wikimedia.org/wiki/File:Portrait_of_Sir_Isaac_Newton,_1689_\(brightened\).jpg](https://commons.wikimedia.org/wiki/File:Portrait_of_Sir_Isaac_Newton,_1689_(brightened).jpg)

Newton's 1st law

"A body *remains* at rest,
or in motion
at a constant speed
in a straight line,
unless acted upon by a **force**."

Änderung der Geschwindigkeit

Wir können die Geschwindigkeit ändern, indem wir eine Kraft $\mathbb{F} \in \mathbb{R}^3$, [$kg \cdot m \cdot s^{-2}$] wie zum Beispiel die Schwerkraft auf unseren Körper ausüben. Die Änderung der Geschwindigkeit wird als **Beschleunigung** a bezeichnet, die gemäß dem **zweiten Newtonschen Gesetz** berechnet werden kann:

$$\mathbb{F} = m \cdot a \Rightarrow a = \frac{\mathbb{F}}{m}$$

■ If there is *varying* velocity and a *constant* Force:

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}^{(t)} + \frac{1}{2}(\Delta t)^2 \mathbf{a}$$

$$\mathbf{v}^{(t+\Delta t)} = \mathbf{v}^{(t)} + \Delta t \mathbf{a}$$

$$\mathbf{a} = \frac{\mathbb{F}}{m}$$

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \int_t^{t+\Delta t} \mathbf{v}^{(t')} dt'$$

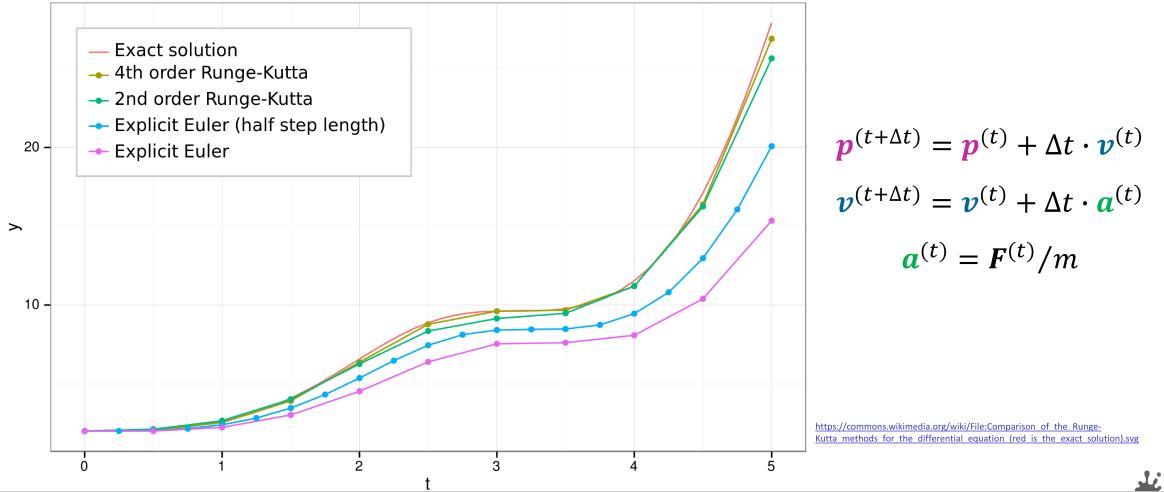
Hard to compute

⇒ use numerical integration methods!

$$\mathbf{a}^{(t)} = \mathbb{F}^{(t)}/m$$

Wenn sowohl eine sich ändernde Geschwindigkeit als auch eine sich ändernde Kraft annehmen, beinhaltet die Berechnung von $\mathbb{P}^{(t+\Delta t)}$ mehrere Integrale.

■ Numerical integration: **Explicit Euler**



- Large timesteps: Unstable
Small timesteps: more computations
- Better explicit integration scheme:
Runge-Kutta

- More accurate, even with larger timesteps
- But also involves more computations again
- Often used in offline settings
or when higher accuracy is required

Numerische Integration

Explicit Euler:

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}^{(t)}$$

$$\mathbf{v}^{(t+\Delta t)} = \mathbf{v}^{(t)} + \Delta t \cdot \mathbf{a}^{(t)}$$

$$\mathbf{a}^{(t)} = \frac{\mathbf{F}^{(t)}}{m}$$

unstabil bei großen Zeitschritten
(hohe Abweichung/Oszillationen),
einfache Berechnung,
real-time

Implicit Euler:

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}^{(t+\Delta t)}$$

$$\mathbf{v}^{(t+\Delta t)} = \mathbf{v}^{(t)} + \Delta t \cdot \mathbf{a}^{(t+\Delta t)}$$

$$\mathbf{a}^{(t+\Delta t)} = \frac{\mathbf{F}^{(t+\Delta t)}}{m}$$

bedingungslos stabil,
aufwändige Berechnung
(Lösung eines Gleichungssystems),
offline

Semi-Implicit Euler:

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}^{(t+\Delta t)}$$

$$\mathbf{v}^{(t+\Delta t)} = \mathbf{v}^{(t)} + \Delta t \cdot \mathbf{a}^{(t)}$$

$$\mathbf{a}^{(t)} = \frac{\mathbf{F}^{(t)}}{m}$$

energiebewahrend,
einfache Berechnung,
real-time

```

1 //Expliziter Euler
2 Vec3 F = berechneKraft();
3 Vec3 beschleunigung = F * objekt.inverseMasse;
4 objekt.position += objekt.geschwindigkeit * deltaZeit;
5 objekt.geschwindigkeit += beschleunigung * deltaZeit;

1 //Semi-Impliziter Euler
2 Vec3 F = berechneKraft();
3 Vec3 beschleunigung = F * objekt.inverseMasse;
4 objekt.geschwindigkeit += beschleunigung * deltaZeit;
5 objekt.position += objekt.geschwindigkeit * deltaZeit;

```

■ Numerical integration: **Explicit Euler**

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}^{(t)}$$

$$\mathbf{v}^{(t+\Delta t)} = \mathbf{v}^{(t)} + \Delta t \cdot \mathbf{a}^{(t)}$$

$$\mathbf{a}^{(t)} = \mathbf{F}^{(t)} / m$$

```

Vec3 F = computeForce();
Vec3 acceleration = F * object.inverseMass;
object.position += object.velocity * deltaTime;
object.velocity += acceleration * deltaTime;

```

■ Numerical integration: **Semi-Implicit Euler**

$$\mathbf{p}^{(t+\Delta t)} = \mathbf{p}^{(t)} + \Delta t \cdot \mathbf{v}^{(t+\Delta t)}$$

$$\mathbf{v}^{(t+\Delta t)} = \mathbf{v}^{(t)} + \Delta t \cdot \mathbf{a}^{(t)}$$

$$\mathbf{a}^{(t)} = \mathbf{F}^{(t)} / m$$

```

Vec3 F = computeForce();
Vec3 acceleration = F * object.inverseMass;
object.velocity += acceleration * deltaTime;
object.position += object.velocity * deltaTime;

```

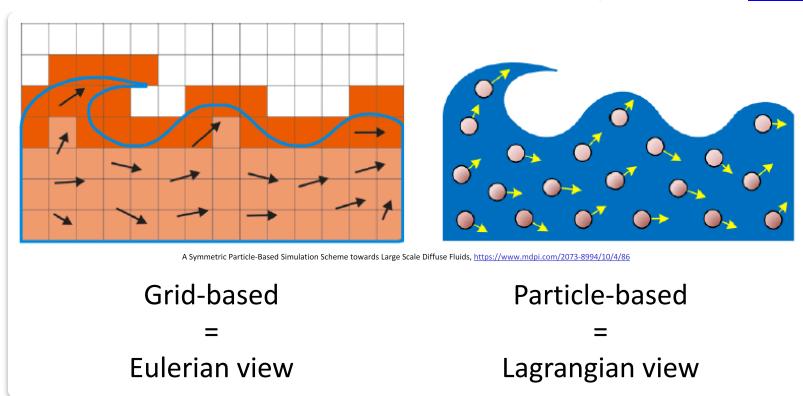
*Simple & more stable
Often used in practice*

Differentialgleichungen

[EVC_Skriptum_CG](#), p.58

Bisher haben wir uns sogenannte **partikelbasierte Simulationen** angesehen, bei denen eine Form durch mehrere Punkte approximiert wird. Dies wird auch als **Lagrange-Sichtweise** der Simulation bezeichnet.

Ein anderer Ansatz besteht darin, den Raum mit einem oft gleichmäßigen Gitter zu unterteilen, was dann als **Euler-Sichtweise** bezeichnet wird.



Letztendlich formulieren wir **Differentialgleichungen**, d.h. Gleichungen, die eine Funktion und ihre Ableitungen enthalten.

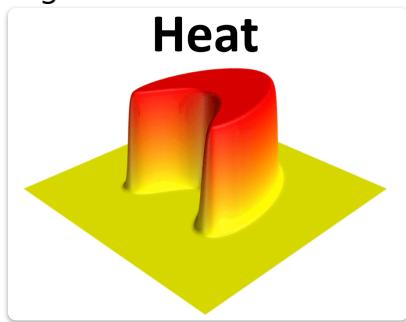
Arten von Differentialgleichungen

Gewöhnliche Differentialgleichungen (ODEs)

- Enthalten nur vollständige Ableitungen (geschrieben als $\frac{dp}{dx}$, $p'(x)$ oder manchmal \dot{p}).
- **Beispiel:** Zweites Newtonsches Gesetz $F(t, \mathbf{p}, \mathbf{v}) = m \frac{d^2 \mathbf{p}^{(t)}}{dt^2}$.

Partielle Differentialgleichungen (PDEs)

- Komplexer als ODEs.
- Beschreiben mehrdimensionale Funktionen mit mehreren Parametern und deren partiellen Ableitungen (z.B. $\frac{\partial p}{\partial x}$ oder $\partial_x p$) bezüglich nur eines der Parameter.
- **Beispiel:** Die 2D-Wärmeleitungsgleichung (Bild 2), die beschreibt, wie sich Wärme entlang einer 2D-Oberfläche ausbreitet.
- **Weitere Anwendungen:** Akustische Wellen oder Fluidsimulationen (wobei die zugrunde liegende PDE die Navier-Stokes-Gleichung ist).



Numerische Berechnung

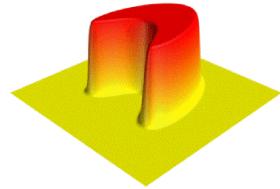
Eine Technik zur numerischen Berechnung von Ableitungen sind sogenannte **Finite-Differenzen-Verfahren** (Bild 3).

Die unbekannten Werte können wiederum mithilfe von **expliziten** oder **impliziten Verfahren** berechnet werden.

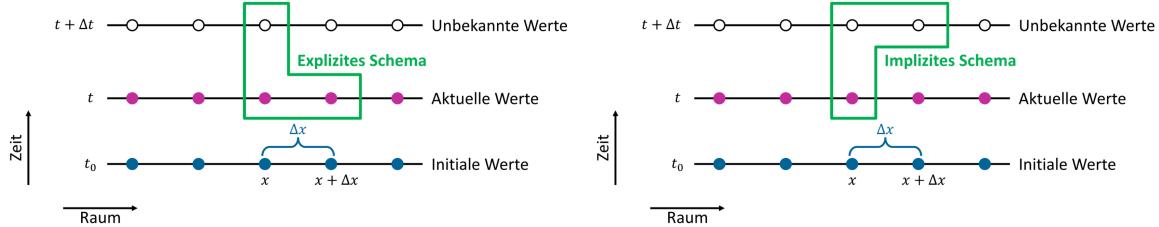
$$\frac{\partial h}{\partial t} = \alpha \left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right), \quad \text{with } h : \Omega \subset \mathbb{R}^2 \rightarrow \mathbb{R}$$

$$\frac{\partial u(x, t)}{\partial x} \approx \frac{u(x + \Delta x, t) - u(x, t)}{\Delta x}$$

(2)



(3)



Charakter Animation

EVC_Skriptum_CG, p.58

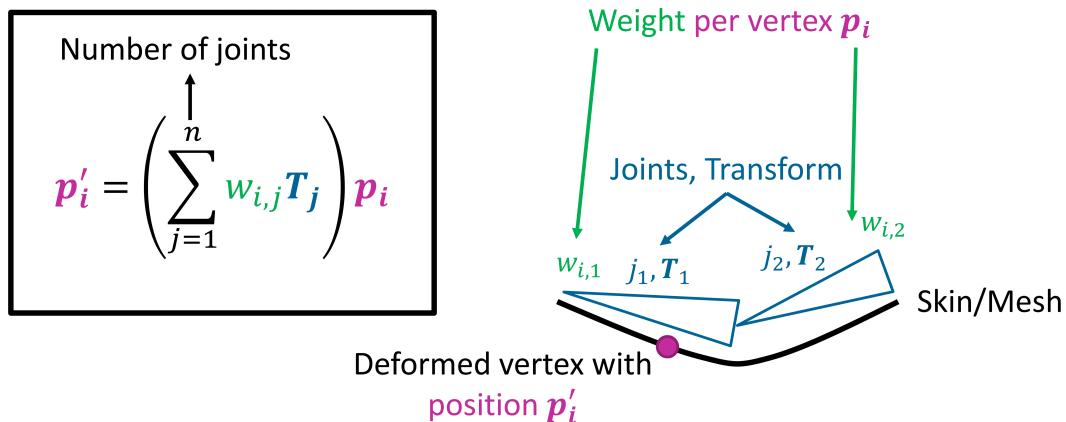
Linear Blend Skinning (LBS) ist eine einfache Methode zur Berechnung aktualisierter Vertex-Positionen basierend auf der Deformation eines zugrundeliegenden Skeletts.

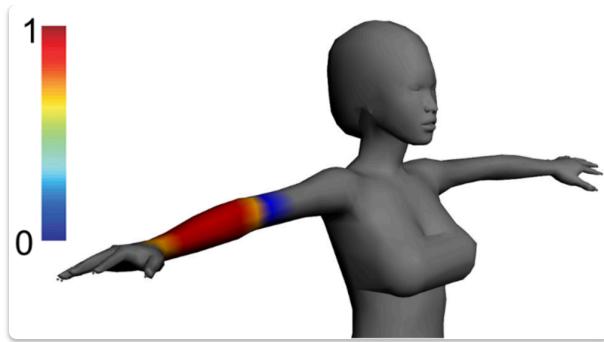
Funktionsweise von LBS

1. **Gelenk-Transformation:** Jedes Gelenk j wird eine Transformation \mathbb{T}_j zugewiesen (die auch von einer Hierarchie von Gelenken abhängen kann und zum Beispiel als Matrix $\mathbb{T}_j \in \mathbb{R}^{4 \times 4}$ dargestellt werden kann).
2. **Einfluss auf Vertices:** Jedes Gelenk j hat einen individuellen Einfluss auf jeden Vertex i , der durch das Gewicht $w_{i,j}$ erfasst wird.
3. **Deformation:** Um einen Vertex i von seiner ursprünglichen Position \mathbb{p}_i zu einer deformierten Position \mathbb{p}'_i zu bewegen, summiert man über die gewichteten Beiträge aller Gelenke (n ist die Anzahl der Gelenke):

$$\mathbb{p}'_i = \left(\sum_{j=1}^n w_{i,j} \mathbb{T}_j \right) \mathbb{p}_i$$

■ Linear Blend Skinning:

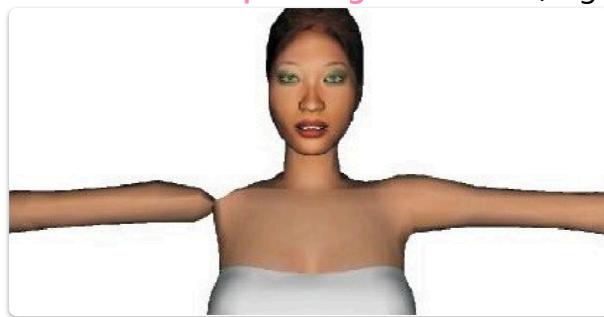




Probleme von LBS

LBS hat jedoch mehrere Probleme, wie zum Beispiel:

- Hoher manueller Aufwand.
- Das „Bonbonverpackungs“-Artefakt (engl. *candy wrapper effect*)



Alternativen zu LBS

Alternativ können **Ragdolls** basierend auf physikalischer Simulation oder **Motion-Capture-Techniken** verwendet werden:

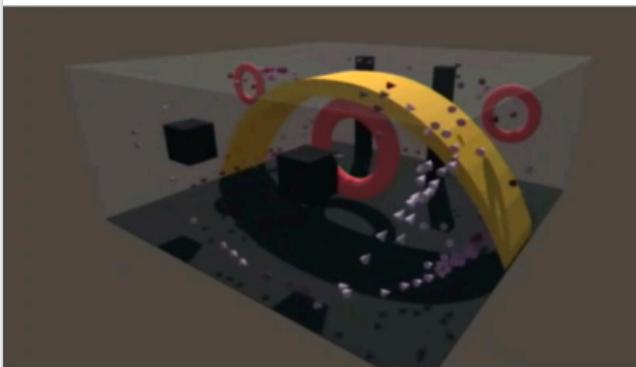
- **Motion-Capture:** Die Bewegungen realer Menschen werden direkt über Kameras und Marker auf den Schauspielern erfasst und dann auf einen virtuellen Charakter übertragen.

Prozedurale Techniken

[EVC_Skriptum_CG, p.58](#)

Ähnlich wie physikbasierte Simulationen können **prozedurale Techniken** verwendet werden, um automatisch Animationen zu erstellen.

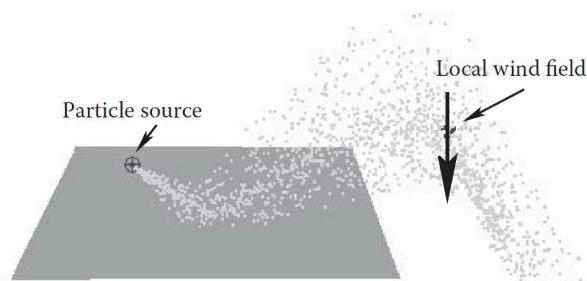
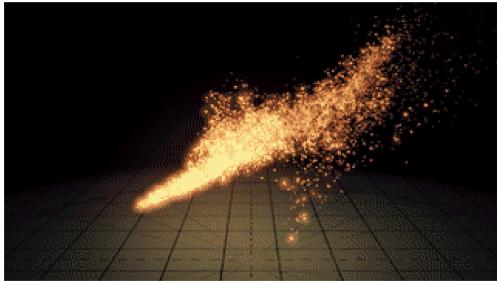
Swarm/Flock/Boids



Game of Live (Cellular automata)



Particle System + Wind Field



Merkmale

- Folgen einer bestimmten Reihe von Regeln, kombiniert mit Zufallsprinzipien.
- Ziel: plausible Animationen zu erstellen, jedoch nicht mit dem Ziel der physikalischen Korrektheit.

Beispiele

- **Game of Life:** Ein bekanntes Beispiel für einen zellulären Automaten.
- **Partikelsysteme:** Für Phänomene wie Regen, Schnee, Feuer.
- **Schwarmverhalten:** Erzeugung von plausiblem Schwarmverhalten (z.B. Vogelschwärme oder Fischschwärme).