

# Was ist eine Physik-Simulation?

Manuel Kaufmann, Valentino Rugolo, Fabio Sulser

15. Oktober 2013

# Ablauf

- **Grundlagen:** Definition, grundlegende Konzepte, Beispiele mit Processing
- **Box2D und toxiclibs:** Beispiellibraries für Physik-Simulationen in Processing
- **Ammo.js:** Physik-Simulationen im Web

# Grundlagen

## Physik-Engine: eine Definition

"... Simulation physikalischer Prozesse..."

⇒ *Physikalische Modelle liegen zu Grunde*

"... Vereinfachung der Programmierung..."

⇒ *Ziel ist die Visualisierung*

"... meist ... Effizienz vor Exaktheit..."

⇒ *Oft ist die Physik dahinter zu komplex*

## Libraries für Physik-Simulationen

- Je näher an der Realität, desto aufwendiger und komplexer werden Berechnungen und Konzepte dahinter
- Wieso das Rad neu erfinden?

⇒ *Library verwenden*

- Es soll jedoch nachvollziehbar sein, WIE das Rad erfunden wurde

⇒ *Grundlagen sind wichtig, um Funktionen der Library zu verstehen*

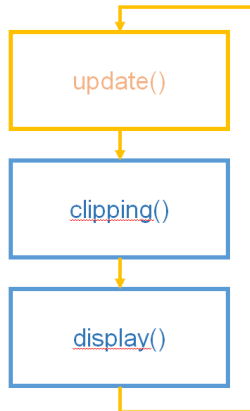
- Deshalb trifft man einfache bzw. stark einschränkende Annahmen über physikalische Welt

# Grundlagen - Bewegung

- Vektoren zur Repräsentation von Position, Geschwindigkeit und Beschleunigung
- Hier ist `update()` unsere "Engine":

```
direction.normalize();  
acceleration = direction;  
velocity.add(acceleration);  
location.add(velocity);
```

⇒ **acceleration.pde**

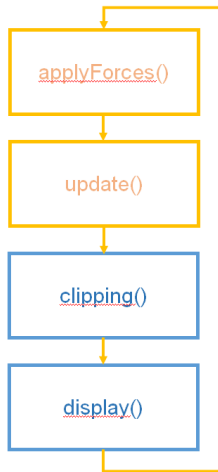


## Grundlagen - Kräfte

- Ausgangslage ist stets die entsprechende physikalische Formel
- Falls eine Kraft simuliert werden soll, wird sie vor `update()` auf die Beschleunigung gerechnet:

```
PVector f = PVector.div(frc, mss)  
acceleration.add(f);
```

⇒ **forces.pde** simuliert Gravitation und Luftwiderstand (Reibung).



# Grundlagen - Oszillationen

- Im Prinzip nichts Neues
  - Ort und Winkel
  - Geschwindigkeit wird Winkelgeschwindigkeit
  - Beschleunigung wird Winkelbeschleunigung
- Zusätzliches Wissen über trigonometrische Funktionen nötig
- Pipeline bleibt die selbe

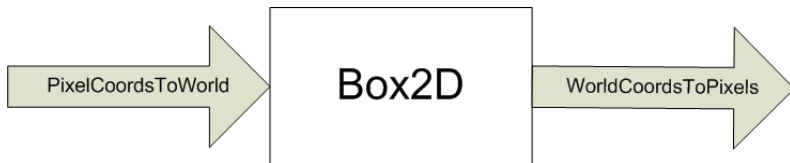
⇒ **pendulum.pde**





## Box2D - 'a true physics engine'

Die Engine weiss nichts über die Welt der Grafik und Pixel



## Box2D - Eigenschaften

- Simuliert starre Körper
  - Existiert seit 2006
  - Geschrieben in C++
  - Findet bspw. Anwendung in Angry Birds, Crayon Physics etc.
- + sehr realistisch
- aufwändige Translationen der Koordinaten
  - '2D-limitiert'

# Box2D und Processing

## JBox2D

- ist ein Java-Port von Box2D
- voll kompatibel mit Processing

## PBox2D

- Helfer-API, kein Processing-Wrapper
- Vereinfacht häufig gebrauchte Funktionen
- wird hier nur als Beispiel gebraucht, ist nicht zwingend notwendig

# Box2D und Processing - Erster Ansatz

## SETUP

create all objects in the world

## DRAW

1 calculate all forces

2 apply forces to objects ( $F=M*A$ )

3 calculate new positions

4 draw objects

# Box2D und Processing - Zweiter Ansatz

```
SETUP
  create all objects in the world
DRAW
  draw objects
```

# Box2D und Processing - Finaler Ansatz

## SETUP

- 1 create all objects in the world
- 2 translate pixel world to box2d world

## DRAW

- 1 ask box2d where the objects are
- 2 translate box2d world to pixel world
- 3 draw objects

## Box2D - Wichtige Elemente

- World** Verwaltet die Simulation
- Body** Ist ein „primary element“ mit einem Ort und einer Geschwindigkeit
- Shape** Verwaltet die Kollisionsgeometrie eines Objekts und verleiht einem Objekt Masse
- Fixture** Verbindet ein Shape mit einem Body und bestimmt Eigenschaften wie Dichte, Elastizität etc.
- Joint** Agiert als Verbindung zweier Bodys oder einem Body und der World.



## Box2D - Typen von Bodys

- dynamic** Ein voll simulierter Körper. Kann sich bewegen, kollidieren und reagiert auf Kräfte.
- static** Gleich wie ein dynamischer Körper, ist aber nicht fähig sich zu bewegen.
- kinematic** Kann manuell verschoben werden, kollidiert aber nur mit dynamischen Körpern.

Grundsätzlich existiert ein Body an sich physisch noch nicht. Um Masse zu erhalten, muss ein Shape mit ihm assoziiert werden.

# Box2D und Processing - Beispiel **cubes.pde**

## Vorgehen:

1. Definiere einen Body
2. Konfiguriere die Body-Definition
3. Erstelle den Body
4. Definiere ein Shape
5. Erstelle ein Fixture
6. Verbinde das Shape mit dem Body mittels dem Fixture

## Box2D und Processing - Weitere Beispiele

- **circles.pde**: Komplexere Formen als im ersten Beispiel
- **attract.pde**: Demonstration von Anziehungskraft

# toxiclibs

- Physik-Library, welche speziell für Processing entwickelt wurde
- + Keine Umstände mit dem Import
- + Das Koordinatensystem ist das selbe

## Vorweg - Welche Library verwenden?

*In meinem Projekt gibt es Kollisionen zwischen Kugeln, Pyramiden und sonstigen komisch geformten Objekten.*

⇒ **Box2D**

*In meinem Projekt gibt es viele kleine Partikel, welche sich anziehen und abstossen und manchmal an einer Feder befestigt sind.*

⇒ **toxiclibs**

# Vergleich 1

	Box2D	toxiclibs
Kollisionen	X	
3D-Physik		X
Anziehung, Abstossung		X
Federverbindungen	X	X
Andere Verbindungen*	X	
Motoren	X	
Reibung	X	

\* Scharniere, Drehgelenke, Getriebe, Flaschenzüge etc.

## Vergleich 2

Der Struktur von `toxiclibs` ist ähnlich zu demjenigen von `Box2D`.

- World (== VerletPhysics)
- Body (== VerletParticle)
- Shape
- Fixture
- Joint (== VerletSpring)

## toxiclib - Einfaches Beispiel

- **toxiclib.pde**



## toxiclib - ein mächtiges Tool

- **softbody.pde**: Vernetzung mehrerer Partikel zu einem Netz
- **complex.pde**: Komplexe, bewegliche Form
- **additive\_waves.pde**: Sich überlagernde Wellen

# Ammo.js - Physik-Simulationen im Web

# Hintergrund

- Enscripted Physik-Library C++ (Bullet Physics)
- Keine JavaScript-Optimierung
- ca. 23'000 Zeilen aber schlecht dokumentiert
- 3D-Umgebung
- Findet Anwendung in verschiedenen Bereichen wie Games (GTA, Toy Story), Filmen (Hangover, Sherlock Holmes) oder sonstigen Tools (Blender, Cinema 4D)

# Library

- Vordefinierte Formen
- Benutzerdefinierte Formen
- collision detection
- Fahrzeug-System

## Eigenschaften

- Ammo speichert keine Objekte, bei jedem Rendern werden die aktuellen Objektkoordinaten berechnet.
- Neue Objekte zu erstellen ist sehr aufwändig, dafür sind extrem detaillierte Gebilde möglich.
- Keine Kollisionsevents
- Gespeicherte Kontakte werden ausgelesen. Diese enthalten Objekte und Position.

# Three.js

- JavaScript-Bibliothek zum Erstellen und Darstellen von 3D-Objekten
- Veröffentlichung 2010, Anfänge bereits 2000
- Material, Kameraperspektive, Licht, Objekte
- HTML5, WebGL, SVG (skalierbare Vektorgrafiken)

# Szene erstellen

## HTML

```
<canvas id="viewport" width="800" height="600"></canvas>
```

## JavaScript

```
renderer = new THREE.CanvasRenderer({canvas : viewport});  
scene = new THREE.Scene;  
camera = new THREE.PerspectiveCamera(34, 1, 1, 1000);  
camera.position.set(-10, 30, -200);  
camera.lookAt(scene.position);  
scene.add(camera);  
  
var ambientLight = new THREE.AmbientLight(0x555555);  
scene.add(ambientLight);  
  
var directionalLight = new THREE.DirectionalLight(0xffffff);  
directionalLight.position.set(-5, 5, -1.5).normalize();  
scene.add(directionalLight);
```

## Three.js - Objekte

- sind schwer zu erstellen
- Positionierung an Position oder relativ zu einem Objekt?
- BSP relativ: Scharnier zwischen Objekten



## Three.js - Kugelobjekt erstellen

```
ball\_geometry = new THREE.SphereGeometry( 3 ),
ball\_material = new THREE.MeshLambertMaterial({
    color: 0x0000ff, overdraw: true });

ball = new THREE.Mesh( ball\_geometry, ball\_material );
ball = new THREE.Mesh( ball\_geometry );
scene.add( ramp_1 );

ball.position.y = 50;
ball.position.x = Math.random() * 40 - 20;
20 ball.useQuaternion = true;
scene.add( ball );
```

## Three.js - Quaderobjekt erstellen

```
var ramp_geometry= new THREE.CubeGeometry( 50, 2, 10 ),
material_red = new THREE.MeshLambertMaterial({
    color: 0xdd0000, overdraw: true }),
scene.add( ramp_1 );

ramp_1.position.x = -20;
ramp_1.position.y = 25;
ramp_1.rotation.z = -Math.PI / 28;
```

# Ammo.js - Objekte erstellen

```
var collisionConf = new Ammo.btDefaultCollisionConfiguration;
world = new Ammo.btDiscreteDynamicsWorld(
    // Disponent f r Kollisionshandling
    new Ammo.btCollisionDispatcher(collisionConf),
    // Ueberpr fung von berlagerungen
    new Ammo.btDbvtBroadphase,
    // Constraintl ser
    new Ammo.btSequentialImpulseConstraintSolver,
    // Kollisionskonfiguration
    collisionConf
);
world.setGravity(new Ammo.btVector3(0, -9.81, 0));
```

## Ammo.js - Kugelobjekt erstellen

```
shape = new Ammo.btSphereShape( 3 );
shape.calculateLocalInertia( mass, localInertia );

transform = new Ammo.btTransform;
transform.setIdentity();
transform.setOrigin(new Ammo.btVector3(
    ball.position.x, ball.position.y, 0 ));
transform.setRotation(new Ammo.btQuaternion( 0, 0, 0 ));

motionState = new Ammo.btDefaultMotionState( transform );
rbInfo = new Ammo.btRigidBodyConstructionInfo(
    mass, motionState, shape, localInertia );

// Reibung und Abstossung
rbInfo.set_m_friction( .3 );
rbInfo.set_m_restitution( .0 );
```

## Ammo.js - Quaderbjekt erstellen

```
shape = new Ammo.btBoxShape(new Ammo.btVector3( 25, 1, 5 ));
shape.calculateLocalInertia( 0, localInertia );

transform = new Ammo.btTransform;
transform.setIdentity();
transform.setOrigin(new Ammo.btVector3( -20, 25, 0 ));
transform.setRotation(new Ammo.btQuaternion(
    0, 0, -Math.PI / 28 ));

motionState = new Ammo.btDefaultMotionState( transform );

// arguments are mass, motion state, shape and inertia
rbInfo = new Ammo.btRigidBodyConstructionInfo(
    0, motionState, shape, localInertia);
```

# Ammo.js - Objekte hinzufügen

```
body = new Ammo.btRigidBody(rbInfo);  
body.mesh = ball;  
world.addRigidBody(body);  
balls.push(body);  
  
body.setCollisionFlags(body.getCollisionFlags() | 8);
```

# Animation der Objekte

```
world.stepSimulation(  
    1 / 30, // speed of simulation  
    10,     // max substeps  
    1 / 60  // size of each substep  
);  
  
for ( i = 0; i < balls.length; i++ ) {  
    transform = balls[i].getCenterOfMassTransform();  
  
    origin = transform.getOrigin();  
    rotation = transform.getRotation();  
  
    balls[i].mesh.position.set(  
        origin.x(), origin.y(), origin.z() );  
    balls[i].mesh.quaternion.set(  
        rotation.x(), rotation.y(), rotation.z(), rotation.w() );  
}  
  
renderer.render(scene, camera);
```

# Objekte verknüpfen

```
var hinge1 = new Ammo.btHingeConstraint(  
    box2,  
    new Ammo.btVector3(0,1,0),  
    new Ammo.btVector3(0,0,1),  
    false  
);  
  
var hinge2 = new Ammo.btHingeConstraint(box1,  
    box2,  
    new Ammo.btVector3(0,1,0),  
    new Ammo.btVector3(0,-1,0),  
    new Ammo.btVector3(0,0,1),  
    new Ammo.btVector3(0,0,1),  
    false  
);  
  
world.addConstraint(hinge1, true);  
world.addConstraint(hinge2, true);
```



# Positionierung der Kamera

```
camera = new THREE.PerspectiveCamera(35, 2.5, 1, 1000);  
camera.position.set(60, 40, 100);  
camera.lookAt(scene.position);  
  
scene.add(camera);
```

## Schatten hinzufügen

```
light = new THREE.DirectionalLight( 0xFFFFFF );
light.position.set( 40, 40, 25 );
light.target.position.copy( scene.position );
light.castShadow = true;
light.shadowCameraLeft = -25;
light.shadowCameraTop = -25;
light.shadowCameraRight = 25;
light.shadowCameraBottom = 25;
light.shadowBias = -.0001

scene.add( light );
```

## Links mit Beispielen

- <http://schteppe.github.io/ammo.js-demos/>
- <http://kripken.github.io/ammo.js/examples/new/ammo.html>
- <http://creativejs.com/2013/08/take-a-drive-around-a-nuclear-wasteland/>
- [http://alteredqualia.com/xg/examples/animation<sub>p</sub>hysics<sub>l</sub>evel.html](http://alteredqualia.com/xg/examples/animation_physics_level.html)