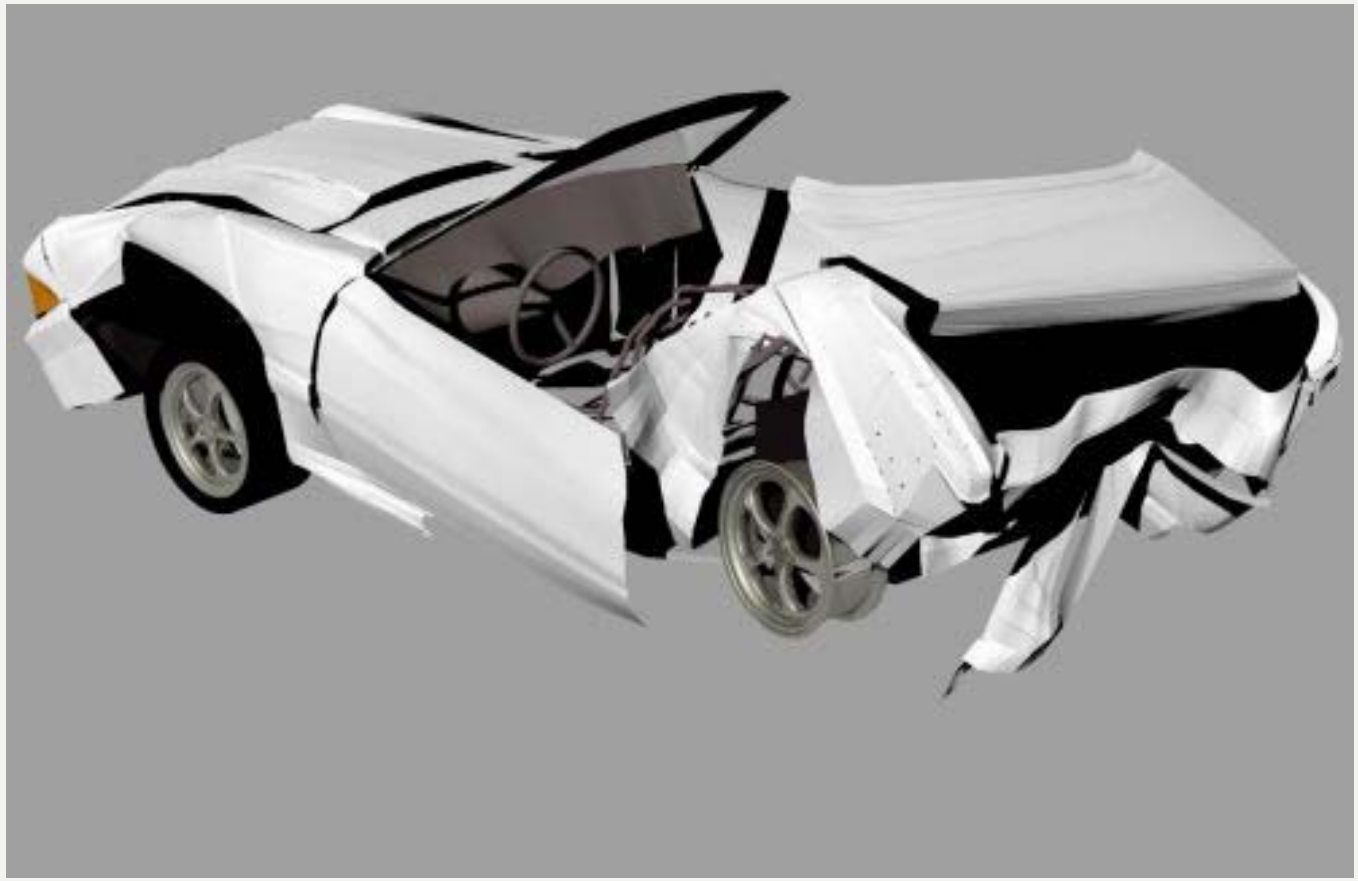
LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Prof. Andreas Butz | Dipl.Inf. Otmar Hilliges

Programmierpraktikum 3D Computer Grafik

Kollisionserkennung | Physics Simulation

Kollisionen



Collisions Detection in OpenGL

- OpenGL bietet keine direkte Methode zur Erkennung von Kollisionen
- Programme müssen diese Aufgabe selbst übernehmen
- Viele unterschiedliche Ansätze
  - Ausnutzung der Kamera und des Z-Puffers
  - Numerische Methoden
  - Analytische (Auflösen von Gleichungssystemen höherer Dimension)
  - …

Agenda

- Kollisionserkennung
  - Kugel ↔ Ebene
  - Kugel ↔ Zylinder
  - Kugel ↔ Kugel
- Physik-basiertes Modellieren
  - Kollisionsreaktion
- Kollisionen von komplexen Modellen

Kollisionserkennung (Kugel ↔ Ebene)

- Prinzipielles Vorgehen:
  - Abstand des Objektmittelpunkts von der Ebene berechnen
  - Ablauf:
    - Mittelpunkt des Objekts in die Normalform der Ebene einsetzen → Ergebnis = Abstand
    - Falls der Abstand nun kleiner dem Radius ist besteht eine Kollision

Kollisionserkennung (Kugel ↔ Ebene)

Beschreibung einer Ebene:

- Punkt und Normale (Normalform)
- X_n: Normale der Ebene (Normalisiert)
- X: Punkt innerhalb der Ebene
- d: Distanz der Ebene vom Ursprung

$$X_n \cdot X = d$$

Beschreibung eines Strahls (Ray):

- Gerade (Parameterform)
- p_r: Punkt auf dem Strahl (3D-Vektor)
- r_S: Startpunkt des Strahls (3D-Vektor)
- r_D: Richtung des Strahls (3D-Vektor)

$$p_R = r_S + \lambda \cdot r_d$$

Kollisionserkennung (Kugel ↔ Ebene)

- Berechnung durch Einsetzen des Punktes in Ebenengleichung

$$p_R = r_S + \lambda \cdot r_d$$

$$X_n \cdot X = d$$

$$\Rightarrow X_n \cdot p_R = \left(X_n \cdot r_S\right) + \lambda \cdot \left(X_n \cdot r_d\right) = d$$

$$\Rightarrow \lambda = \frac{d - X_n \cdot r_S}{X_n \cdot r_d}$$

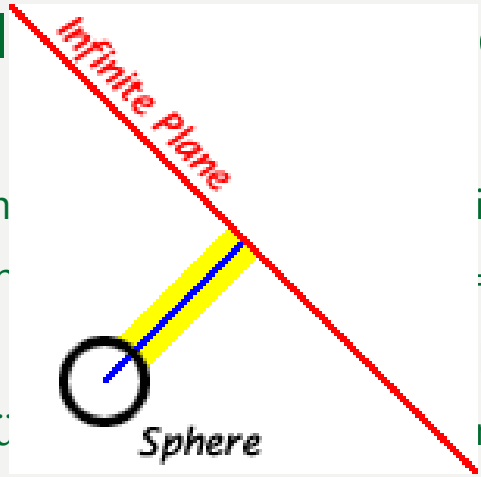
$$\Rightarrow \lambda = \frac{X_n \cdot X - X_n \cdot r_S}{X_n \cdot r_d} = \frac{X_n \cdot \left(X - r_S\right)}{X_n \cdot r_d}$$

Kollisionserkennung (Kugel ↔ Ebene)

- *lambda* gibt die Entfernung vom Startpunkt des Strahls zum Kollisionspunkt an
- Eigenschaften:
  - *lambda* > Radius: Keine Kollision
  - $X_n \cdot r_d = 0$: Vektoren stehen senkrecht aufeinander → keine Kollision
  - 0 < *lambda* < Radius: Kollision

Kollision mit Polygonen

- Bisher: Test ob Kugel(mittelpunkt) in einer Ebene liegt.
- Problem?
- Ebenen sind unendlich – Polygone nicht!
- Mehrere Mögl en ob Punkt in einem Polygon liegt:
  - Dreieck: Punkt m inden dann Winkel zwischen den Linien berechner => Punkt innerhalb)
  - Schneller: Punkt Polygonseiten einsetzen => Ergebniss >= 0 fü nerhalb



Kollisionserkennung (Kugel ↔ Zylinder)

- Ähnlich zu der Technik bei der Ebene:
- Problem: Welche Ebene wird gewählt?
- Lösung:
  - Berechnung des Abstands vom Kugelmittelpunkt zur Rotationsachse des Zylinders (= Gerade)
  - Über Normalform der Gerade
  - Nun darf der Abstand nicht kleiner sein als die Summe der Radien von Kugel und Zylinder

Kollisionserkennung (Kugel ↔ Kugel)

Einfachster Fall:

- Abstand der Mittelpunkte bestimmen
- Abstand < Radius_1 + Radius_2: Kollision

Problem:

- Wie verhalten sich zwei Kugeln nach einer Kollision (z.B. Billard)?

Lösung: Kollisionsreaktion

Kollisionsreaktion

Einfallswinkel = Ausfallswinkel:

- Winkel werden zur Normalen am Kollisionspunkt gemessen

Für die Berechnung notwendig:

- Kollisionspunkt
- Normale am Kollisionspunkt *N*
- Winkel zwischen Normale und Bewegungsrichtung *I*

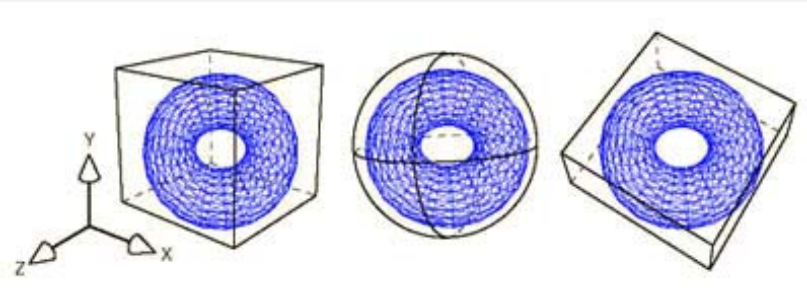
$$R = 2 \cdot \left(- I \cdot N \right) \cdot N + I$$

Motivation:

- Keine Standardobjekte (z.B. Kugeln), sondern komplexe 3D-Objekte (z.B. Asteroiden)

Verfahren:

- Hüllvolumen um den eigentlichen Körper
- Verschiedene Volumina möglich
- Verwendung von Oct-tree oder BSP-tree Verfahren üblich



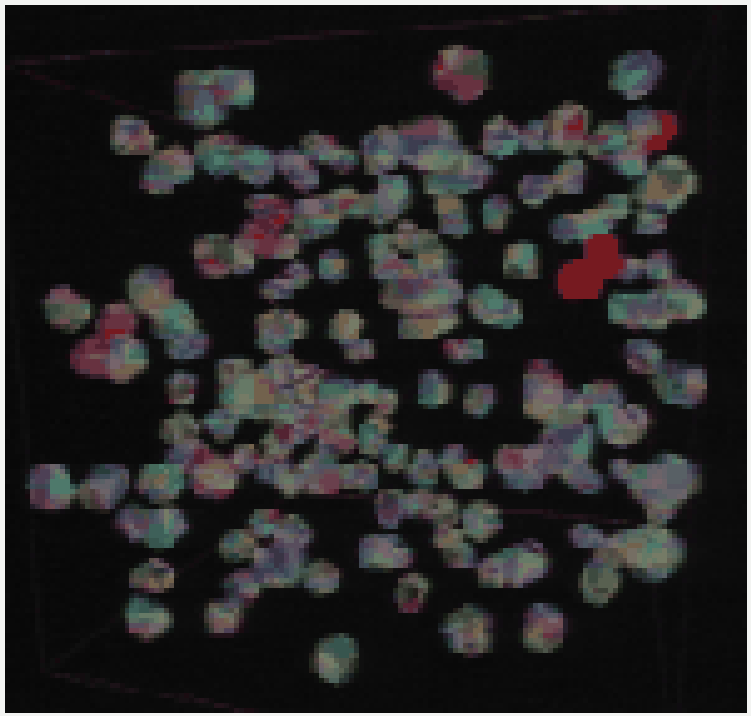
Axis-aligned Bounding Box, Bounding Sphere, Oriented Bounding Box

Tips und Tricks

- Nur Objekte die im Clipping Volume liegen betrachten
- Erst auf Vorhandensein einer Kollision testen
- Dann Schnittpunkt berechnen
- Kollisionserkennung vor dem Bewegen von Objekten machen
- Schnelle Objekte können sonst durch andere hindurch fliegen

Libraries

- Es gibt diverse Libraries die Kollisionserkennung implementieren
- z.B.: I Collide



Literatur

Konzept der Kollisionserkennung:

http://www.flipcode.com/articles/article_basiccollisions.shtml

3D-Kollisionserkennung (Teil 1):

http://www.scherfgen-software.net/index.php?action=tutorials&topic=collision_1

NeHe Productions: OpenGL Lesson #30:

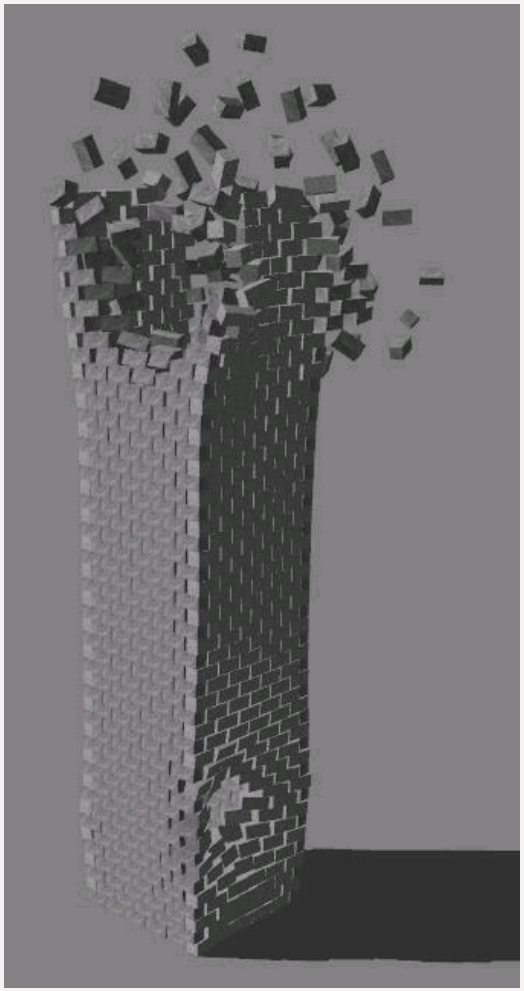
http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=30

Team Gamma Collision Detection

http://www.cs.unc.edu/~geom/collide/index.shtml

Introduction to Ageia PhysX

Generelles Prinzip



- Ageia (jetzt Nvidia) PhysX ist eine Library die es erlaubt realistisches mechanisches / dynamisches zu modellieren
- Rigid body dynamics
  - Kräfte
  - Reibung
  - Trägheit
  - Position, Beschleunigung, Geschwindigkeit
- Kollisionskontrolle (collision detection)
- Kollisionsbehandlung (collision response)

Vorraussetzungen

- Physiksimulation ist komplex
- Grundverständniss für physikalische Prinzipien wird vorraus gesetzt
- -> Verständnis für Interaktion von Konzepten ist wichtig
- Berechnung und Simulation von Phänomenen ist gekapselt

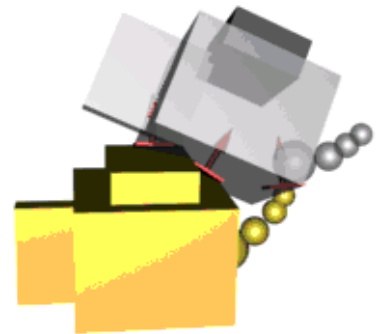
Verschiedene Repräsentationen

- Simulation läuft unabhängig von der grafischen Repräsentation
- Synchronisation zwischen Graphik und Simulation notwendig
- Simulation kann als Controller (MVC) verstanden werden
- Grafische Objekte werden nach der Simulation neu positioniert / orientiert

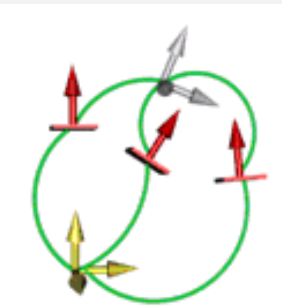
Bodies, Actors and Tensors

- Grafische Repräsentation (Polygone, Meshes etc)

- Approximation für die Kollisionserkennung



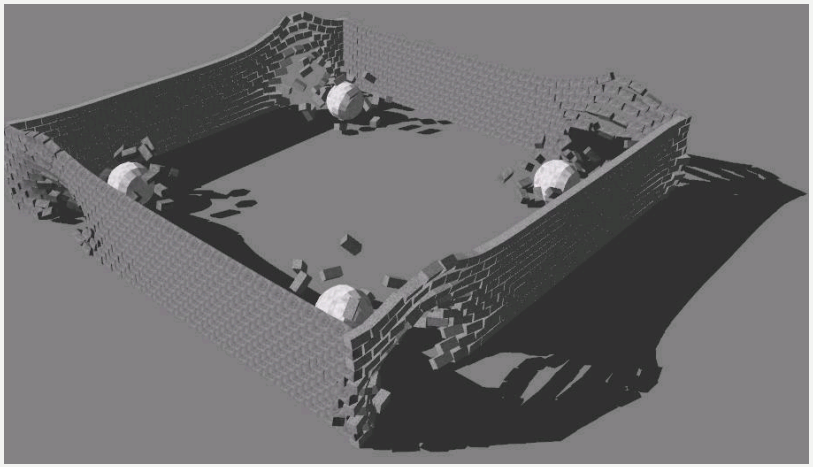
- Trägheitstensoren für die Simulation

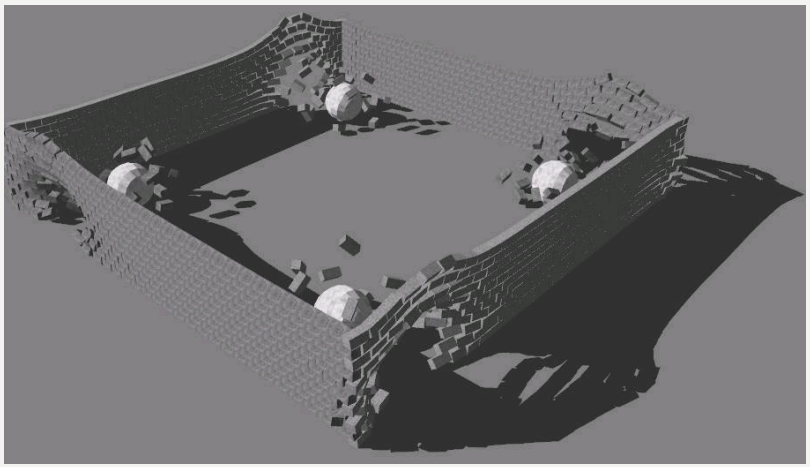


Actors, Shape, Bodies

- Elemente innerhalb der Simulation heißen „actors“
- Actors können mit einem Volumen (shape) für die Kollisionskontrolle assoziiert sein
- Können mit einer grafischen Repräsentation verknüpft sein

Verhalten von Objekten





Ständig vorhandene Faktoren

- Gravitation
- (Luft)wiederstand
- Reibung (statisch und dynamisch)

Variable Faktoren

- Kollisionen
- Von außen einwirkende Kräfte

Kräfte

- Kräfte (force) sind die natürliche Währung
- Kraft = Masse * Beschleunigung
- Beschleunigung = Geschwindigkeitsänderung pro Zeiteinheit
- Kräfte können am Schwerpunkt angebracht werden oder an spezieller Position z.B. Triebwerk eines Raumschiffes

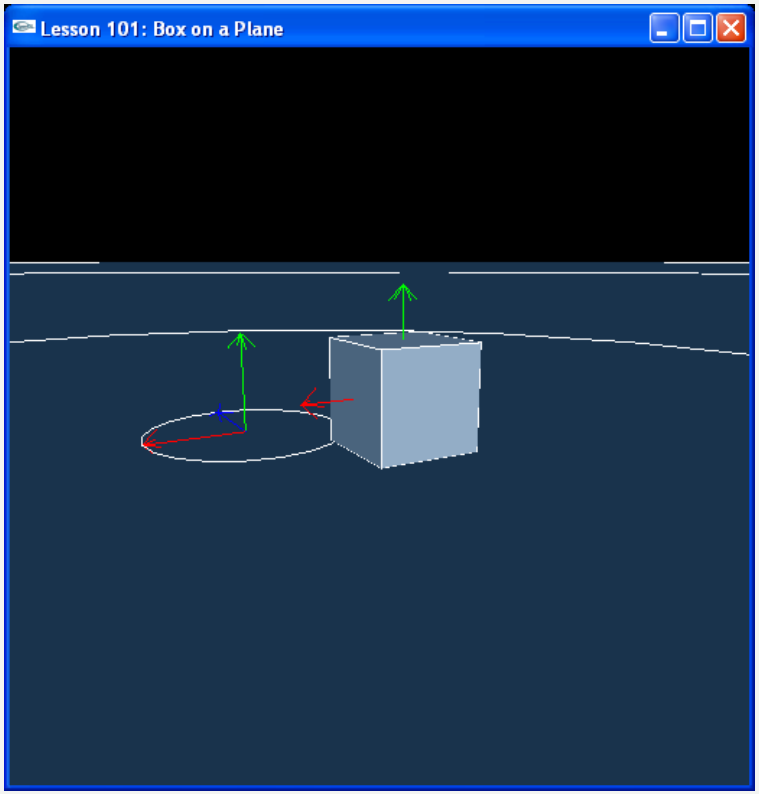
```
void addForceAtPos(const NxVec3 & force, const NxVec3 & pos, NxForceMode);
void addForceAtLocalPos(const NxVec3 & force, const NxrVec3 & pos, NxForceMode);
void addLocalForceAtPos(const NxVec3 & force, const NxVec3 & pos, NxForceMode);
void addLocalForceAtLocalPos(const NxrVec3& force, const NxVec3& pos, NxForceMode);

void addForce(const NxVec3 &, NxForceMode);
void addLocalForce(const NxVec3 &, NxForceMode);
void addTorque(const NxVec3 &, NxForceMode);
void addLocalTorque(const NxVec3 &, NxForceMode);
```

Vorgehen zum erstellen einer Szene

1. Initialisieren der Physik Engine
2. Erstellen von statischen actors (Boden, Wände)
3. Erstellen von dynamischen actors
    1. Erzeugen eines actors mit actor description (Masse, Material)
    2. Erzeugen eines Volumens für die Kollisionskontrolle
    3. Erzeugen einer grafischen Repräsentation
4. Simulationsschritt ablaufen lassen
5. Grafische Repräsentation updaten
6. Inputs verarbeiten (Kräfte berechnen)
7. Schritt 4…

Box on a plane



- SDK Initialisierung
- Szene initialisieren
- Actore(n) erzeugen
- Simulation laufen lassen
- Kräfte

```
int main(int argc, char** argv)
{
    InitGlut(argc, argv);
    InitNx();
    glutMainLoop();
    ReleaseNx();
    return 0;
}
```

PhysX SDK Initialisieren

```
#include "Lesson101.h"
// Physics SDK globals
NxPhysicsSDK*   gPhysicsSDK = NULL;
NxScene* gScene = NULL;
NxVec3  gDefaultGravity(0,-9.8,0);
void InitNx()
{
    // Create the physics SDK
    gPhysicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION);
    if (!gPhysicsSDK)  return;
    // Create the scene
    NxSceneDesc sceneDesc;
    sceneDesc.gravity = gDefaultGravity;
    gScene = gPhysicsSDK->createScene(sceneDesc);
}
```

Default Material

```
void InitNx()
{
    ...
    // Create the default material
    NxMaterial* defaultMaterial =
    gScene->getMaterialFromIndex(0);
    defaultMaterial->setRestitution(0.5);
    defaultMaterial->setStaticFriction(0.5);
    defaultMaterial->setDynamicFriction(0.5);
    ...
}
```

Boden erstellen

```
void InitNx()
{
...
    groundPlane = CreateGroundPlane();
...
}
NxActor* CreateGroundPlane()
{
    // Create a plane with default descriptor
    NxPlaneShapeDesc planeDesc;
    NxActorDesc actorDesc;
    actorDesc.shapes.pushBack(&planeDesc);
    return gScene->createActor(actorDesc);
}
```

Box erstellen

```
void InitNx()
{
...
    box = CreateBox();
...
}

NxActor* CreateBox()
{
    // Set the box starting height to 3.5m so box starts off falling onto the ground
    NxReal boxStartHeight = 3.5;
    // Add a single-shape actor to the scene
    NxActorDesc actorDesc;
    NxBodyDesc bodyDesc;
    // The actor has one shape, a box, 1m on a side
    NxBoxShapeDesc boxDesc;
    boxDesc.dimensions.set(0.5,0.5,0.5);
    actorDesc.shapes.pushBack(&boxDesc);
    actorDesc.body = &bodyDesc;
    actorDesc.density = 10;
    actorDesc.globalPose.t = NxVec3(0,boxStartHeight,0);
    return gScene->createActor(actorDesc);
}
```

Simulation Starten

```
void InitNx()
{
...
    // Start the first frame of the simulation
    if (gScene)  StartPhysics();
}
...
void StartPhysics()
{
    // Update the time step
    gDeltaTime = UpdateTime();
    // Start collision and dynamics for delta time since the last frame
    gScene->simulate(gDeltaTime);
    gScene->flushStream();
}
```

Simulations-Ergebnisse abholen

```
void RenderCallback()
{
...
    if (gScene && !bPause)
    {
        GetPhysicsResults();
        ProcessInputs();
        StartPhysics();
    }
...
}
...
void GetPhysicsResults()
{
    // Get results from gScene->simulate(deltaTime)
    while (!gScene->fetchResults(NX_RIGID_BODY_FINISHED, false));
}
```

Input verarbeiten

```
// Force globals
NxVec3 gForceVec(0,0,0);
NxReal gForceStrength = 20000;
…
void ProcessForceKeys()
{
    // Process force keys
    for (int i = 0; i < MAX_KEYS; i++)
    {
        if (!gKeys[i])  { continue; }
        switch (i)
        {
            …
            // Force controls
            …
            case 'j': {gForceVec = ApplyForceToActor(box, NxVec3(1,0,0), gForceStrength); break; }
            …
        }
    }
}
```

Kraft anwenden

```
NxVec3 ApplyForceToActor(NxActor* actor, const NxVec3&
    forceDir, const NxReal forceStrength)
{
    NxVec3 forceVec = forceStrength*forceDir*gDeltaTime;
    actor->addForce(forceVec);
    return forceVec;
}
```

Actors zeichnen

```
void RenderActors()
{
    // Render all the actors in the scene
    NxU32 nbActors = gScene->getNbActors();
    NxActor** actors = gScene->getActors();
    while (nbActors--)
    {
        NxActor* actor = *actors++;
        DrawActor(actor);
}
void RenderCallback()
{
      …
      RenderActors();
      …
}
```

*Kompletter Code ist in DrawActor.cpp

Shutdown

```
void ReleaseNx()
{
    if (gScene)
    {
        GetPhysicsResults();  // Make sure to fetchResults()
    before shutting down
        gPhysicsSDK->releaseScene(*gScene);
    }
    if (gPhysicsSDK)  gPhysicsSDK->release();
}
```