Dokumentation: Kollisionsdetektion, WPF-Projekt in SSDS

Ivan Kurilin Dimitrij Pivovar

20. Januar 2025

Technische Hochschule Köln
Prof. Herr Dr. Konen

1 Einführung

1.1 Projektziel

Das Ziel dieses WPF-Projekts war es, eine Kollisionsdetektion in einer 2D-Umgebung zu implementieren. Die Objekte sollten eine realistische Reaktion auf Kollisionen mit den Bällen zeigen und dem Benutzer verschiedene Methoden zur Veranschaulichung bieten, um die Kollisionen näher zu betrachten und besser zu verstehen.

1.2 Überblick

Für die Entwicklung des Projekts stand uns eine einfachere Version in Processing zur Verfügung. In dieser wurden gleichmäßig bewegte Bälle im Raum simuliert, die nur mit den Wänden kollidieren konnten. Auf dieser Grundlage haben wir ein Projekt entwickelt, bei dem nicht nur die Kollisionen sichtbar sind, sondern dem Benutzer auch eine benutzerfreundliche Oberfläche, Funktionen zur Visualisierung und Optimierungsalgorithmen bereitgestellt werden.

2 Benutzeroberfläche

2.1 Design

Dem User wird eine benutzerfreundliche Oberfläche angeboten, die man ganz einfach durch die Menüpunkte navigieren kann.

2.2 Funktionen

- **Normalmodus**:
  Dies ist die Grundlage des Projekts. Hier kann der Benutzer sehen, wie das Programm zu Beginn war.
- **Kollisionsdetektion (Bruteforce)**:
  Eine einfache Überprüfung auf Kollisionen mittels Bruteforce mit einer Laufzeit von $O(n^2)$.

- **Impuls Demonstration**:
  In diesem Modus hat der Benutzer die Möglichkeit, per Mausklick auf die Bälle zu klicken. Das Ziel dieses Modus ist es, zu verdeutlichen, welchen Einfluss die Masse auf die Kollision mit anderen Bällen hat, die weniger Masse besitzen.

- **Tunneling Demonstration**:
  In diesem Modus wird der quantenmechanische Tunneleffekt simuliert. Dies veranschaulicht das Phänomen, bei dem Teilchen eine Barriere durchqueren können, die sie gemäß klassischer Physik nicht passieren könnten.

- **Vektorenrichtung Drawer**:
  Dieser Effekt zeichnet Pfeile, die in die Bewegungsrichtung der Bälle zeigen.

- **Effet**:
  Dieser Modus ermöglicht es, die Rotation des Balls zu betrachten.

- **Quadtree Algorithmus**:
  Der Quadtree Algorithmus verbessert die Performance der Kollisionsdetektion. Durch die hierarchische Unterteilung des Raumes in vier Teilbereiche pro Knoten können Kollisionen effizienter erkannt werden. Die Laufzeit für das Einfügen eines Elements in einen Quadtree beträgt O(log(n)).

3 Physikalische Grundlagen

3.1 Elastische Kollision

Wir bezeichnen einen Stoß dabei als elastisch, wenn die Summe der kinetischen Energien der Stoßpartner nach dem Stoß genau so groß ist wie vor dem Stoß. [1]
Das bedeutet für uns, dass keine kinetische Energie in andere Energieformen wie Wärme oder Verformungsenergie umgewandelt wird. Ein klassisches Beispiel für eine nahezu elastische Kollision ist der Zusammenstoß von Billardkugeln: Nach dem Stoß bewegen sich die Kugeln mit nahezu unveränderter kinetischer Energie weiter.
Im Gegensatz dazu steht die unelastische Kollision, bei der ein Teil der kinetischen Energie in andere Energieformen umgewandelt wird, was oft zu einer dauerhaften Verformung der Objekte führt. Ein typisches Beispiel hierfür ist ein Autounfall, bei dem die kinetische Energie in Verformungsenergie umgewandelt wird, was zu Schäden am Fahrzeug führt.

Um eine Kollision festzustellen, ist dies bei einer Ballkollision relativ einfach. Dafür benötigen wir die Distanz zwischen den beiden Bällen. Wenn diese Distanz kleiner ist als die Summe der Radien der Bälle, findet eine Kollision statt.

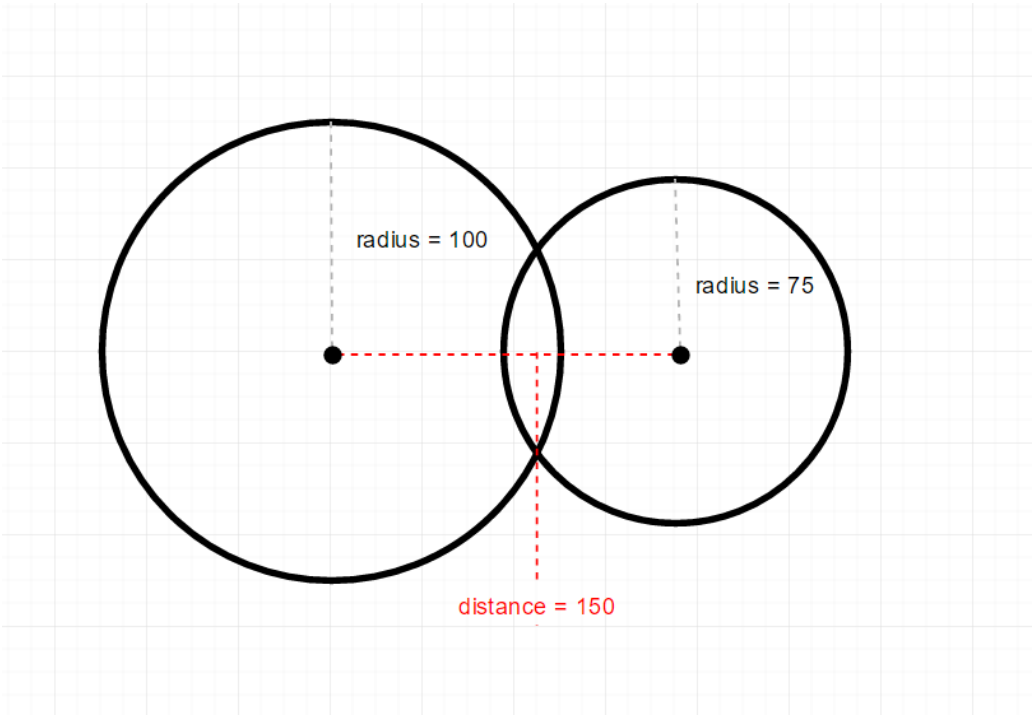

Abbildung 1: Kollision [6]

Die Formel zur Berechnung der Distanz zwischen den Punkten (x_1) und (x_2) lautet:

$$\text{Abstand} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}[7]$$

Im Code wird dies wie folgt umgesetzt:

```
float dx = (float)(b1.sx - b2.sx);
float dy = (float)(b1.sy - b2.sy);
float distance = (float)Math.sqrt(dx * dx + dy * dy);
```

3.2 Impuls und Impulserhaltungssatz

Impuls bezeichnet das Produkt aus der Masse eines Objekts und seiner Geschwindigkeit. Es handelt sich um eine Vektorgröße, die sowohl eine Richtung als auch eine Größe hat. Der Impuls eines Objekts bleibt nach einer elastischen Kollision erhalten, was bedeutet, dass die Gesamtimpulsmenge vor und nach der Kollision gleich bleibt.
Bei der Kollision zweier gleichmäßiger Objekte bleibt der Gesamtimpuls des Systems erhalten. Das bedeutet, dass die Summe der Impulse der beiden Objekte vor der Kollision gleich der Summe der Impulse nach der Kollision ist.
Wenn ein Objekt mit größerem Impuls auf ein Objekt mit kleinerem Impuls trifft, wird ein Teil des Impulses des ersten Objekts auf das zweite Objekt übertragen. Dies führt zu einer Änderung der Geschwindigkeiten beider Objekte, jedoch bleibt der Gesamtimpuls der Objekte unverändert.

Die folgende Gleichung, bekannt als Impulserhaltungsgesetz stellt die Erhaltung der kinetischen Energie bei einem elastischen Stoß zwischen zwei Körpern dar.

$$m_1 v_1^2 + m_2 v_2^2 = m_1 v_1'^2 + m_2 v_2'^2$$

- m_1 und m_2 sind die Massen der Körper.

- v_1 und v_2 sind die Geschwindigkeiten der Körper vor dem Stoß.
- v_1' und v_2' sind die Geschwindigkeiten der Körper nach dem Stoß.

Da wir nun wissen, dass der Gesamtimpuls der beiden objekten nach der Kollision gleich ist, können wir den Impuls, der zwischen den beiden Objekten während der Kollision übertragen wird, berechnen. Dazu berechnen wir zunächst das Skalarprodukt, das die relative Geschwindigkeit entlang der Kollisionsachse angibt. Wir teilen dieses Skalarprodukt durch die Summe der Massen der beiden Objekte, um den Impuls zu berechnen, der auf jedes Objekt übertragen wird. Dieser Impuls wird dann auf die Geschwindigkeiten der Objekte angewendet.

$$Skalararprodukt : \vec{a} \cdot \vec{b} = a_x b_x + a_y b_y$$

Beispiel aus dem Code:

```
float nx = dx / distance;
float ny = dy / distance;
float dvx = (float)(b1.vx - b2.vx);
float dvy = (float)(b1.vy - b2.vy);
float skalarprodukt = dvx * nx + dvy * ny;

if (skalarprodukt <= 0){

float impuls = 2* skalarprodukt / (float)(b1.MASS + b2.MASS);

b1.vx -= impuls * b2.MASS * nx;
b1.vy -= impuls * b2.MASS * ny;
b2.vx += impuls * b1.MASS * nx;
b2.vy += impuls * b1.MASS * ny;
};
```

3.3 Überlappung

In der Computergrafik kann es bei Kollisionen zu einer sogenannten Überlappung (engl. Överlap") kommen, da sich die Objekte zwischen den einzelnen Frames weiterbewegen. Diese Überlappung wird durch die Kollisionserkennung festgestellt und kann durch Anpassung der Positionen der beteiligten Objekte korrigiert werden, sodass sie nicht mehr übereinander liegen.

Im Code wird dies wie folgt umgesetzt:

```
 float overlap =  (distance - (b1.Radius() + b2.Radius()));

 if (overlap < 0){
  b1.sx -= overlap * (b1.sx - b2.sx) / distance;
  b1.sy -= overlap * (b1.sy - b2.sy) / distance;
  b2.sx += overlap * (b1.sx - b2.sx) / distance;
  b2.sy += overlap * (b1.sy - b2.sy) / distance;
 }
```

3.4 Effet

Als Effet (franz. Wirkung) bezeichnet man den Drall eines Balls bzw. einer Kugel. [8] Es handelt sich um einen visuellen Effekt, mit dem man feststellen kann, dass sich ein Objekt dreht. Die Rotation des Balls ergibt sich aus seiner Geschwindigkeit und seinem Radius.

$$w = \frac{v}{r}[9]$$

Im Code wird dies wie folgt umgesetzt:

```
cBalls.ball[i].angleX += cBalls.ball[i].vx * 0.1 / cBalls.ball[i].radius;
cBalls.ball[i].angleY += cBalls.ball[i].vy * 0.1 / cBalls.ball[i].radius;
```

Letzendlich muss der Winkel des Balls kontinuierlich angepasst werden. Dies erfolgt in Processing in der fortlaufenden `draw()` Methode:

```
public class Ball
{..
 void draw() {
if (times >= 0) {
..
rotateX(angleX);
rotateY(angleY);

..
}
}..}
```

In Processing sind die Sphären einfarbig. Damit wir nun die Wirkung sehen können, benötigen wir die Klasse `PShape`, mit der wir einen beliebig geformten Körper erstellen können, und die Klasse `PImage`, die uns das Bild speichert. Mit der Methode `setTexture(image)` laden wir die Textur auf unseren Körper [10].

Beispiel im Code:

```
PShape our_sphere;
PImage texture = loadImage("rainbow-square.png");


Ball (int count) {
..
our_sphere = createShape(SPHERE, this.Radius());
our_sphere.setStroke(false);
our_sphere.setTexture(texture);
..
}
```

Abbildung 2: Earth Texture for Spheres

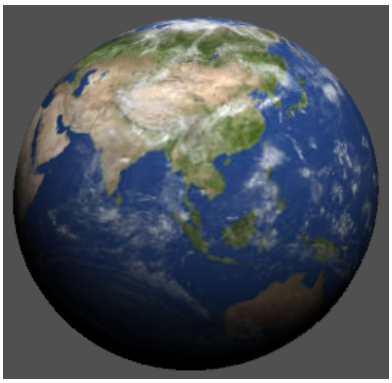

Abbildung 3: Earth Spheres

4 Algorithmen zur Kollisionsdetektion

4.1 Bruteforce

Die Brute-Force-Methode (von englisch brute force ‚rohe Gewalt') [11] ist einer der bekanntesten Suchalgorithmen in der Informatik. Sie ist bekannt für ihre einfache Implementierung, bringt jedoch auch Nachteile mit sich. Bei großen Datenmengen ist dieser Algorithmus mit seiner Laufzeit von $O(n^2)$ einer der langsamsten Algorithmen.

4.1.1 Funktion

Die Implementierung ist unkompliziert: Für jeden Ball, den wir betrachten, prüfen wir alle anderen Bälle und führen die gewünschten Operationen aus.

Beispiel im Code:

```
void bruteforce() {
for (int i = 0; i < ball.length; i++) {
for (int j = 0; j < ball.length; j++) {
...
}
```

```
}
}
```

4.2 Quad-Tree

Der Quad-Tree ist eine Baumdatenstruktur, bei der jeder Knoten genau 4 Kindknoten enthält (Erinnerung an Algorithmik Vorlesung mit Binary Trees). Der Grund warum wir diese Datenstruktur gewählt haben um darauf Kollisionsdetektion anzuwenden ist, dass Quad-Trees eine Unterteilung eines 2D Raums auf 4 Quadranten möglich macht. Rekursiv verkleinert sich bei Bedarf dadurch die Fläche zum Suchen von möglichen Kollisionen.

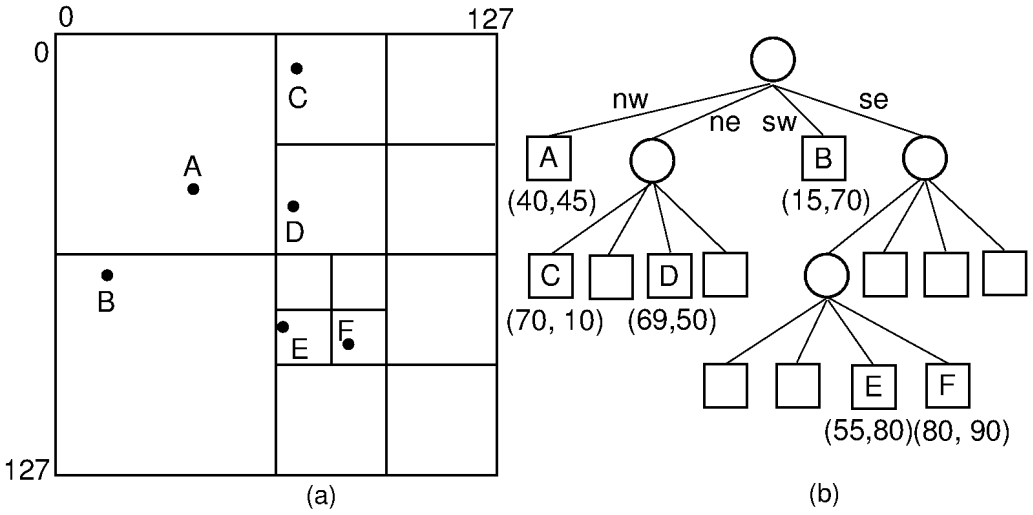

Abbildung 4: Quad-Tree [5]

Die Laufzeiten für den Quad-Tree sind für das Einfügen O(log(n)) [4]
, da der Baum bei jeder Teilung um einen konstanten Faktor kleiner wird. Da wir n Elemente einfügen ist die Gesamtlaufzeit für das Einfügen in unserem Projekt damit O(n*log(n)). Bei der Kollisionsdetektion müssen nur die Knoten durchsucht werden, die sich in der Nähe des zu prüfenden Knoten befinden. Dadurch reduziert sich die Anzahl der Abfragen im Vergleich zum Brute Force Vorgang erheblich.

4.3 Probleme

- Zuerst hatten wir das Problem, dass es uns aufgrund des fehlenden Rasters in Processing schwer fiel das Detektionsverhalten nachzuvollziehen. Es wurden anfangs teilweise Elemente ignoriert und wir wussten nicht warum auf einige Elemente die Detektion angewandt wurde und auf andere nicht.

4.4 Quadtree

Ein Quadtree oder Quaternärbaum ist in der Informatik eine Baumstruktur, in der jeder innere Knoten genau vier Kindknoten hat. [12] Er wird häufig in der Computergrafik verwendet, um 2D-Räume effizient zu unterteilen. Im Vergleich zu einem Bruteforce bietet der Quadtree eine deutlich bessere Laufzeit, da die Komplexität bei Abfragen oder Einfügungen nur O(log(n)) beträgt. Obwohl der Quadtree eine bessere Leistung bietet, ist er komplexer in der Implementierung und Verwaltung.

5 Visualisierungen und Demonstrationen

5.1 Vektorenrichtungs-Darstellung

Der Effekt kann mit der Taste `v` ein- und ausgeschaltet werden. Hier werden Pfeile in Laufrichtung gezeichnet, sodass der Benutzer die Bewegungsrichtung der Bälle leicht nachvollziehen kann. Wir nehmen die Position und Velocity von jedem Element und berechnen mittels der Velocity die Richtung in der sich das Element bewegen wird. Anschließend zeichnen wir die Gerade und setzen ein Dreieck zum Pfeildarstellung an das Ende der Geraden

5.2 Tunneling-Demonstration

Wir wollten einige Probleme vorstellen, die sich bei der Kollisionsdetektion einschleichen könnten. Da wir in der Vorlesung bereits die Probleme bei der Performance durch viele Elemente bereits behandelt haben, haben wir uns an das Tunneling Problem [2] vorgenommen.
Im Prinzip ist das Tunneling Problem das Problem wenn sich ein Element zu schnell bewegt und die Framerate zu niedrig ist. Angenommen ein Element bewegt sich auf eine Wand zu, das Element bewegt sich konstante 5 Units / Frame und die Wand ist 3 Units von dem Element entfernt. Im 2. Frame wäre das Element dann 2 Units hinter der Mauer und es wurde keine Kollision erkannt, da sich die beiden Elemente in dem Frame wo die Abfrage nach der Kollision stattfindet nicht berühren.

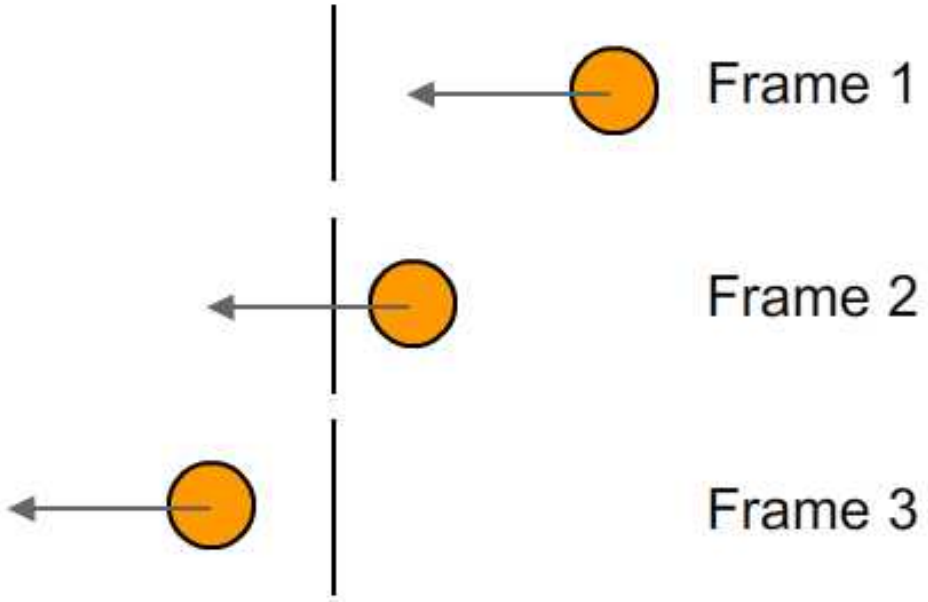

Abbildung 5: Tunneling [3]

5.3 Impuls-Demonstration

Der Modus kann mit der Taste `3` eingeschaltet werden. In diesem Modus hat der Benutzer die Möglichkeit, per Mausklick die Masse der Bälle zu steuern. Mit einer größeren Masse, wächst auch der Radius.

- Linksklick: `+10%` Masse, `+5%` Radius
- Rechtsklick: `-10%` Masse, `-5%` Radius

Durch den direkten Einfluss auf die Masse kann der Benutzer beobachten, wie sich der Impuls bei Kollisionen mit Bällen verhält, die mehr oder weniger Masse haben.

Der Radius hat ebenfalls einen Einfluss auf den Effet Effekt, da eine größere Radiusgröße zu einer langsameren Rotation führt.

5.4 Effet

Der Effet Effekt, wie schon oben beschrieben, zeigt uns die Rotation. Dieser Effekt kann im gesamten Programm mit der Taste `E` ein- und ausgeschaltet werden.

6 Fazit

Dieses Projekt sowie das Tool Processing bieten eine sehr gute Möglichkeit, Physik auf anschauliche Weise für Studierende zu demonstrieren. Durch die verschiedenen Projekte und eine spielerische Herangehensweise können vordefinierte Körper mit bekannten Formeln in relativ großen Programmen simuliert und getestet werden. Da die IDE jedoch nicht immer die beste Performance bietet, ist es besonders wichtig, auf Clean Code und die Vermeidung von Memory Leaks zu achten.

6.1 Allgemeine Probleme

- Debugger:
  Das größte Problem, auf das wir gestoßen sind, ist, dass man mit Processing keine guten Möglichkeiten hat, den Debugger zu benutzen. Bereits bei kleineren Klassen fällt es schwer, jede Instanz nachzuverfolgen. Das Problem verschärft sich, wenn man Klassen innerhalb einer Klasse erstellt, wie z.B. bei uns mit dem Quadtree, oder wenn man rekursive Methoden debuggen möchte.

- Text & Farben:
  Auch die einfache Erzeugung von Text oder das Anpassen von Farben bei Objekten hat uns an einigen Stellen viel Zeit gekostet. Wir haben festgestellt, dass durch die kontinuierliche Anzeige von Text in der draw() Methode Memory Leaks entstehen können, was dazu führt, dass das Programm abstürzen kann.
  Bei der Farbänderung mancher Objekte wirkte sich die Anpassung direkt auf den sichtbaren Bereich aus. Das machte es notwendig, separate Methoden zu entwickeln, um Objekte gezielt in den Vorder- oder Hintergrund zu verschieben und ihre Darstellung präziser zu steuern.

Literatur

[1] Leifi Physik: Zentraler elastischer Stoß

[2] gdbooks.gitbooks.io: Dynamic Intersections

[3] StackExchange: Tunneling Explained

[4] jrtechs.net Implementing a Quad Tree

[5] CS3 Data Structures and Algorithms: Darstellung eines Quadtrees

[6] Happy Coding: Collision Detection Tutorial

[7] Serlo: Abstand zweier Punkte berechnen

[8] Effet - Wikipedia

[9] Bahngeschwindigkeit und Winkelgeschwindigkeit

[10] PShape

[11] Bruteforce - Wikipedia

[12] Quadtree - Wikipedia