2D Spiele programmieren in Java

Teil 5: 2D Kollisionen

Dr. Katja Wegner
Dr. Ursula Rost

Kollisionen

- Erkennen und darauf reagieren
- Spiellogik erweitern
- 3D Kollisionen werden nicht behandelt
  - Sehr komplex
  - Viele Publikationen über schnelle und korrekte Algorithmen vorhanden

Kollisionsarten (1/2)

- A priori
  - Ein Algorithmus bestimmt die Bewegungen der Objekte und berechnet die Kollisionen bevor diese eintreten
- A posteriori
  - Man überprüft zu einem bestimmten Zeitpunkt für alle Objekte, ob Kollisionen vorliegen und veranlasst die nötigen Updates

Kollisionsarten (2/2)

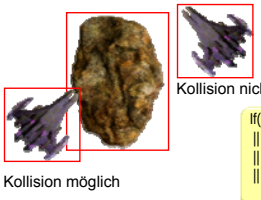
- A priori
  - Pros:
    - Höhere Genauigkeit
    - Mehr Stabilität
  - Cons:
    - Hohe Komplexität der Algorithmen
- A posteriori
  - Pros:
    - Einfacher (z.B. Reibung, verformbare Körper, Zeit müssen nicht berücksichtigt werden)
  - Cons:
    - Bestimmt vielleicht Kollisionen, die physisch nicht korrekt sind

2D Kollisionen bestimmen

- Ein Algorithmus für eine pixel-genaue Kollisionsbestimmung in 3 Schritten

1. Schritt

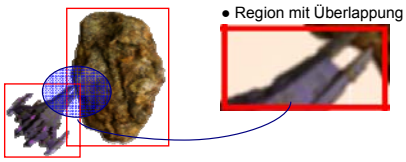
- Jedes Objekt wird als eine rechteckige Box betrachtet → Überlappen diese Boxen?




```
If( box1.x > box2.x + box2.width
 || box1.x + box1.width < box2.x
 || box1.y > box2.y + box2.height
 || box1.y + box1.height < box2.y )
    // Kollision nicht möglich
```

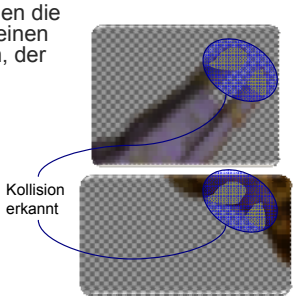
2. Schritt

- Region bestimmen, in der die Überlappung liegt



3. Schritt

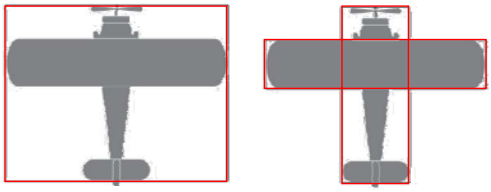
- Die Pixel bestimmen die in beiden Bildern einen Alpha-Wert haben, der nicht 0 ist



Frage: Was sind die Vor- und Nachteile dieser Methode?

Vereinfachter Algorithmus

- Begrenzungsrechteck für 1. Schritt entsprechend der Form optimieren

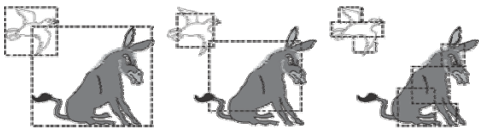


Begrenzungsrechteck 2

Rechtecke nähern Umrisse an und reduzieren so den Rechenaufwand.

Sollten so gewählt Werden, dass eine hohe Genauigkeit gegeben ist.

Wenn nötig können verschiedene Algorithmen (mit steigender Komplexität) angewendet werden.

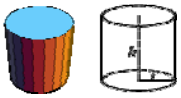


Andere Umrisse (1/2)

- **Kreis** (Zylinder in 3D), beschreibbar durch Mittelpunkt und Radius, umgibt das Objekt. Schneller Test auf Kollisionen:

```
If( (circle1.x – circle2.x)²
    + (circle1.y – circle2.y)²
    > circle1.raidus² + circle2.radius² )
      // Kollision nicht möglich
```

Zylinder: gut für bestimmte 3D Objekte, z.B. (einfache) Umrisse für aufrecht stehende Objekte

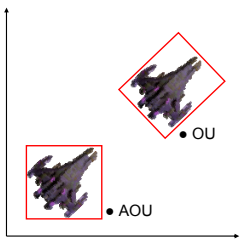


Andere Umrisse (2/3)

An Achsen-orientierter Umriss (AOU): ausgerichtet an den Achsen des Koordinatensystems.

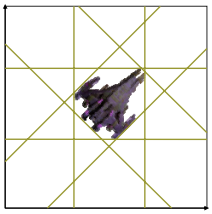
Orientierter Umriss (OU): richtet sich am Objekt aus.

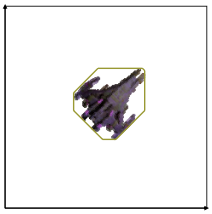
AOUs sind viel einfacher auf Überschneidungen zu testen als OUs, aber müssen immer wieder neu berechnet werden, wenn das Objekt gedreht wurde.



Andere Umrisse (2/3)

- Polygone (in 3D Polyeder)

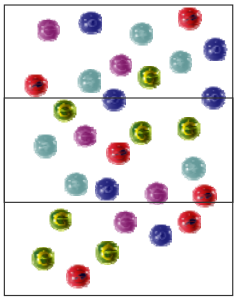




Suchraum verkleinern

- Für n Objekte müssen $n^2/2$ Vergleiche durchgeführt werden
  → **für große n nicht praktikabel**
- Problem:
  - Wie kann Anzahl der vergleiche reduziert werden?
- Lösung:
  - Zerlege Problem in Regionen und löse jede Region einzeln, VORSICHT bei Objekten, die sich zwischen Regionen bewegen

100 Bälle, 1 Region
= $100^2/2$ Vergleiche = 5000
100 Bälle, 10 Regionen (gleichverteilt)
= $10 * 10^2/2$ = 500



Übung 5.1

- Die Spielfigur Muka soll eine Startpunktzahl bekommen (z.B. 20) und bei jeder Kollision mit einem Gegenspieler wird ein Punkt abgezogen. Implementieren Sie die entsprechenden Methoden. Ist die Punkzahl 0, wird das Spiel beendet.

Übung 5.2

- Fügen Sie dem Spiel eine FoodEntity hinzu. Wenn Muka mit diesem Bonus kollidiert, wird sein Punktestand um eins erhöht und das Entity gelöscht.