FH Technikum Wien

BACHELORARBEIT

Collision Detection in der Computergrafik: Die gebräuchlichsten Bounding Volumes

ausgeführt von Stefan Denner
A-2500 Baden, Hofackergasse 8

Begutachter: Dipl.-Ing. (FH) Dietmar Schreiner

Baden, 8.12.2008

FACHHOCHSCHULE
TECHNIKUM WIEN

Ausgeführt an der FH Technikum Wien

Studiengang Informatik

Eidesstattliche Erklärung

„Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.“

_____ _____

Ort, Datum Unterschrift

Abstract

In dieser Arbeit werden vier der bekanntesten und gebräuchlichsten Bounding Volumes, die für die Collision Detection (deutsch: Kollisionserkennung) in der Computergrafik verwendet werden, vorgestellt und näher erklärt. Dazu gehören die Sphere (deutsch: Kugel), die Axis-aligned Bounding Box, die Oriented Bounding Box und die Halfspace Intersection Volumes, zu denen die Slab-based Volumes und Dicrete Orientation Polytopes gehören. Dabei wird auch definiert, was ein Bounding Volume ist und wozu diese überhaupt dient. Zusätzlich wird auch kurz erläutert, was Collision Detection bedeutet und wozu sie in der Computergrafik verwendet wird. Überdies wird in dieser Arbeit zwischen 3D und 2D keine Unterscheidung gemacht, da alle vorgestellten Strukturen und Algorithmen in beiden Dimensionen funktionieren. Es werden auch Code- und Implementierungsbeispiele zu den einzelnen Bounding Volumes vorgestellt, und zwar in der Programmiersprache C bzw. in C++. Somit sind grundlegende Programmierkenntnisse hilfreich zum Verstehen dieser Arbeit, aber nicht Voraussetzung, da die Codebeispiele nur zum Veranschaulichen von schon erklärten Algorithmen und Datenstrukturen bzw. für Entwickler als Grundstein für ihre eigenen Implementierungen dienen. Schlussendlich werden der Vollständigkeit halber auch noch weitere Möglichkeiten für Bounding Volumes kurz erläutert.

Keywords: Collision Detection, Computergrafik, Bounding Volume, Axis-aligned Bounding Box, AABB, Oriented Bounding Box, OBB, Halfspace Intersection Volume, Slab-based Volume, Discrete Orientation Polytope, k-DOP

Inhaltsverzeichnis

1. Einleitung

Die Collision Detection in der Computergrafik ist eine eigene Disziplin, in der Kollisionen zwischen zwei oder mehreren Objekten erkannt werden sollen. Das hat vor allem in Computerspielen den Sinn eine Illusion einer vermeidlich robusten Spielwelt zu erzeugen, in der es dem Spieler nicht möglich gemacht werden soll durch Wände zu laufen oder durch den Boden zu fallen. Diese Fälle müssen aber durch den Programmierer selbst bedacht und implementiert werden. Dieses Problem, obwohl es aus der Sicht eines Menschen sehr einfach und logisch erscheint, wie z.B. einen Spieler nicht durch eine Wand laufen zu lassen, ist für einen Entwickler eine spezielle Herausforderung und nicht einfach zu lösen.

Ein Problem ist, dass die für das Zeichnen auf den Bildschirm verwendete Darstellung der dreidimensionalen Daten von vermeintlich „soliden“ Objekten wie Wände nicht für das Erkennen von Kollisionen geeignet sind. Dreidimensionale Objekte in der Computergrafik werden durch Polygone, also einzelne Dreiecke, definiert, die zusammenhängen und deren Oberfläche das zu zeichnende Objekt darstellen. Es gibt Methoden um Polygone auf Kollisionen bzw. Überschneidungen zu testen, diese sind aber, wie auch bei [Ericson05] auf der Seite 75 erläutert, sehr komplex und können bei Objekten, die aus hunderten oder sogar tausenden Polygonen bestehen, zu Performance-Einbußen der Applikation führen. Dies ist vor allem bei Echtzeitanwendungen mit 30 bis 60 Bildern pro Sekunde wie z.B. Computerspielen, ein großes Problem, da für jeden Frame bzw. für jedes Einzelbild, die Routine zum Erkennen und Handhaben von Kollisionen von Objekten durchlaufen werden muss. Ist diese Routine aber zu zeitaufwendig, können die 30 bis 60 Bilder pro Sekunde nicht erreicht werden und die Grafikapplikation erzeugt keine flüssigen Bildabläufe mehr und gleicht dann mehr einer Diashow, was die Illusion von „bewegten Bildern“ zerstört

Um das oben erläuterte Phänomen zu vermeiden, wurden die Bounding Volumes (deutsch: einhüllende Volumen) zum Erkennen von Kollisionen und Überschneidungen einzelner Objekte eingeführt. Ein Bounding Volume ist eine einfache geometrische Form, wie eine Kugel oder eine Box, welche ein komplexeres Objekt, bestehend aus vielen hunderten oder sogar tausenden Polygonen, komplett umhüllt. Die Idee ist, dass diese „Hülle“ einfachere und schnellere Überlappungstests aufweisen als die Objekte die sie umschließen [Ericson05]. Überdies kann man mittels Ausschlussverfahren auch darauf schließen, dass sich zwei Objekte sicher nicht überlappen, wenn deren Bounding Volumes sich nicht überschneiden. Somit kann auf komplexere Tests verzichtet und eine Performance-Steigerung erreicht werden. Dies wird in Abbildung 1 noch einmal ersichtlich gemacht. Man erkennt, dass eine Überschneidung der Objekte A und B im Vorhinein ausgeschlossen werden kann, da sich deren BVs nicht überschneiden. Eine genauere Überprüfung kann somit ausgelassen werden, was zu einer Performance-Steigerung führt. Im Fall von C und D kann eine Überschneidung nicht ausgeschlossen werden, da die BVs sich überschneiden. Somit wird auch der genauere Test durchgeführt, der aber zu einem negativen Ergebnis führt. Im ersten Augenblick scheint es unsinnig, da jetzt zwei Tests pro Objekt ausgeführt werden müssen. Aber unterm Strich sind die meisten Objekte nicht nah genug aneinander, sodass sich deren BVs nicht überschneiden und somit wird in den meisten Fällen eine Performance-Steigerung der Applikation erreicht.

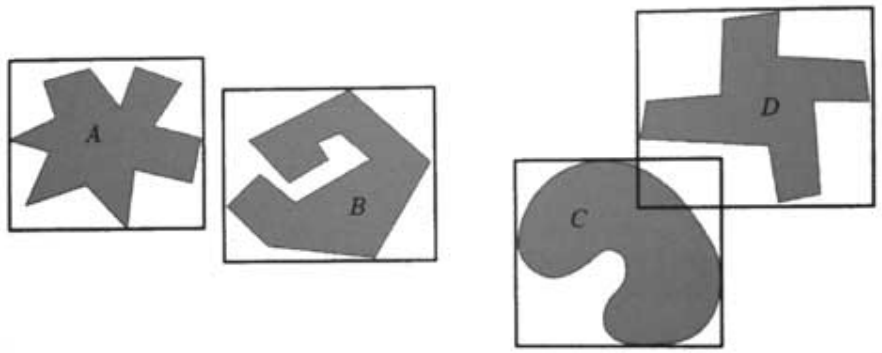

Abbildung 1: Objekte können nur dann überlappen, wenn auch deren BVs überlappen [Ericson05].

Schlussendlich kann man folgende allgemein gültige Charakteristika eines Bounding Volume zuweisen [Endres08]:

- kostengünstige Überschneidungstests
- bestmögliche Anpassung an das Objekt (eng anliegend)
- geringe Erstellungskosten
- Leicht zu rotieren und zu transformieren
- Geringer Speicherverbrauch

Damit ein BV leicht auf Kollisionen testbar ist, sollte diese eine einfache geometrische Form haben. Sie sollte aber auch so eng wie möglich an dem Objekt anliegen, damit das Ausschließen von Kollisionen schon im Vorhinein gut funktioniert. Somit ist ein Kompromiss zu finden zwischen diesen beiden Eigenschaften, wobei eine einfachere geometrische Form in der Regel ungenauer am Objekt anliegt als eine komplexere, aber dafür schnellere Tests erlaubt. Diese zwei Eigenschaften verhalten sich somit umgekehrt proportional zueinander, wie auch in Abbildung 2 zu erkennen ist. Genauso verhält es sich mit dem Speicherverbrauch des BV und der Eigenschaft, wie gut diese Kollisionen im Vorhinein ausschließen können.

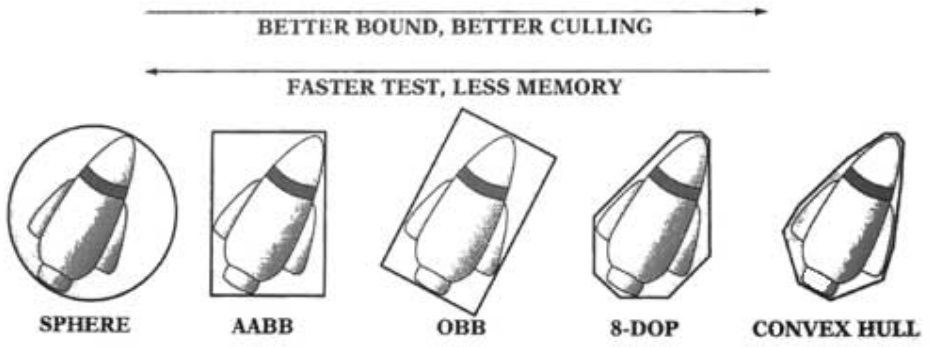

Abbildung 2: Die fünf häufigsten BVs aufgereiht nach Komplexität [Ericson05].

Wie bei [Ericson05] beschrieben werden Bounding Volumes im Vorhinein errechnet und weniger während der Laufzeit neu erstellt. Falls es einmal notwendig ist das BV während der Laufzeit zu verändern, darf dieser Aufwand nicht zu viel Zeit beanspruchen. Auch wenn einmal eine Änderung von Nöten ist, z.B. wenn sich das in dem BV beinhaltete Objekt bewegt, muss auch entschieden werden, ob das BV transformiert wird oder, falls es schneller vonstatten geht, auch während der Laufzeit gleich neu erstellt wird.

Zudem muss auch berücksichtigt werden, dass BVs zusätzlich zu der normalen, am Bildschirm anzuzeigenden Geometrie gespeichert werden müssen. Dabei ist es vorteilhaft, wenn der Speicherverbrauch des BV so gering wie möglich ist. Da sich viele Eigenschaften der verschiedenen BVs gegenseitig ausschließen, ist bezüglich derer Vor- und Nachteile kein spezifisches BV die beste Wahl für jede Situation. Die beste Möglichkeit hierbei ist, verschiedene BVs für ein und dieselbe Applikation zu testen um zu evaluieren, welches BV für diese am besten geeignet wäre. Das perfekte Gleichgewicht zwischen Speicherverbrauch und Performance zu finden ist hierbei die Schwierigkeit, was die Collision Detection auch zu einer eigenen, sehr komplexen und wissenschaftlich orientierten Aufgabenstellung macht.

Im Großen und Ganzen hat die Collision Detection in den letzen Jahren, vor allem in der Computerspieleindustrie, immer mehr an Bedeutung gewonnen und wurde schon früh als eine vermeintliche Engstelle bezüglich der Performance erkannt. Deswegen wurden Methoden entwickelt, um die Tests zu beschleunigen und dabei kam es oft vor, dass verschiedene Wege im Bezug auf Implementierungen gegangen wurden. Dazu gehören auch die verschiedenen Arten von Bounding Volumes. In den folgenden Abschnitten werden die gebräuchlichsten BVs vorgestellt. Dazu gehören die Sphere (deutsch: Kugel), die Axis-aligned Bounding Box (kurz: AABB), die Oriented Bounding Box (kurz: OBB) und die Half Space Intersection Volumes, wobei da die so genannten Discrete Orientation Polytopes (kurz: k-DOP) nach den Slab-based Volumes zu den bekanntesten Vertretern gehören. Im letzten Kapitel wird zusätzlich noch eine kurze Übersicht über weitere Arten bzw. Möglichkeiten für Bounding Volumes aufgezeigt, die der Vollständigkeit dient.

2. Sphere

Eine Sphere (deutsch: Kugel) ist das einfachste BV. Eine Sphere ist in 2D ein Kreis und in 3D eine Kugel, die das Objekt umschließt. In Abbildung 2 kann eine Sphere in 2D und in Abbildung 3 eine Sphere in 3D gesehen werden. Ein Vorteil der Sphere ist, dass sie rotationsunabhängig ist [Endres08], jegliche Rotation des beinhalteten Objektes bleibt für dieses BV ohne Konsequenzen und die Sphere muss nicht transformiert, sondern maximal nur neu positioniert zu werden.

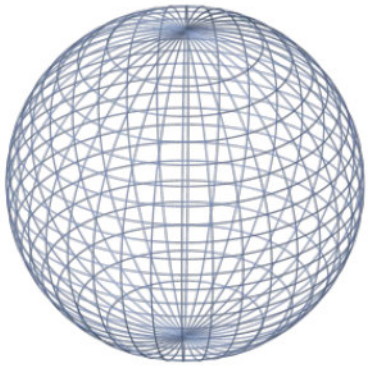

Abbildung 3: Eine Sphere in 3D, eine Kugel.

2.1. Darstellung einer Sphere

Eine Sphere ist das BV mit dem geringsten Speicherverbrauch [Ericson05], da sie nur aus einem Mittelpunkt und dem Radius der Kugel bzw. des Kreises besteht. Darüber hinaus braucht auch nur der Radius gespeichert zu werden, wenn der Mittelpunkt des zu umschließenden Objekts auch gleichzeitig als Mittelpunkt der Sphere verwendet wird.

Ein Implementierungsbeispiel für eine Bounding-Sphere kann somit wie folgt aussehen [Ericson05].

```
struct Sphere
{
     Vector c;
     float r;
}
```

Der Vektor c beschreibt hierbei den Mittelpunkt des Kreises bzw. der Kugel und der Radius r wird als Gleitkommazahl gespeichert.

2.2. Überschneidung zweier Spheres austesten

Das Testen von Überschneidungen zweier Spheres ist sehr einfach. Dabei muss nur darauf geachtet werden, ob der Abstand der beiden Mittelpunkte kleiner ist als die Summe der beiden Radien.

Die Implementierung eines Überschneidungstests zweier Spheres kann folgendermaßen aussehen [Ericson05].

```
int TestSphereSphere(Sphere a, Sphere b)
{
     //Vector d zwischen den Mittelpunkten berechnen!
     Vector d = a.c – b.c;
     //d*d (=Skalarprodukt) ist der Abstand hoch 2!
     float dist2 = Dot(d, d);

     //Überschneidung, wenn Abstand hoch zwei <= Summe der
     //Radien hoch 2 ist!
     float radiusSum = a.r + b.r;
     return dist2 <= radiusSum * radiusSum;
}
```

Die Funktion namens „Dot“ mit zwei Vektoren als Parameter berechnet das Skalarprodukt der beiden Vektoren und kann auch, wie es hier der Fall ist, die quadrierte Länge eines Vektors zurückgeben, wenn der Vektor als Parameter eins und Parameter zwei an die Funktion übergeben wird.

Eine Implementierung kann in 3D folgendermaßen aussehen:

```
float Dot(Vector a, Vector b)
{
     return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}
```

Wie beim Überschneidungstest zu erkennen ist, wird nicht der Abstand an sich zur Berechnung hergezogen, sondern der quadrierte Abstand. Das liegt daran, dass das Wurzelziehen eine sehr aufwendige Operation ist und das Quadrieren der Radien zum Schluss zum Vergleich schneller ist als das Wurzelziehen des Abstandes.

3. Axis-aligned Bounding Box (AABB)

Eine Axis-aligned Bounding Box (deutsch: an Achsen ausgerichtete Hüllbox) (kurz: AABB) ist in 3D ein Quader bzw. in 2D ein Rechteck, deren Seiten parallel zu den Achsen des globalen Koordinatensystems sind. Ein Beispiel für eine zweidimensionale AABB sind in den Abbildungen 1 und 2 auf der Seite 2 zu erkennen. Ein Beispiel für eine dreidimensionale AABB, welche ein Modell eines Menschen umschließt, ist in Abbildung 4 zu sehen.

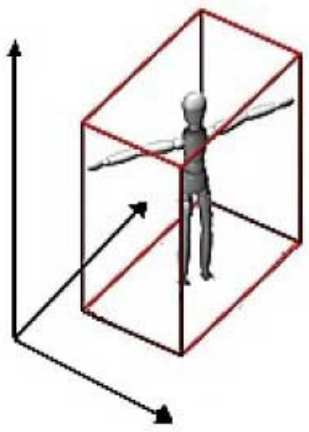

Abbildung 4: Eine Axis-aligned Bounding Box (AABB) in 3D [Endres08].

3.1. Darstellung einer AABB

Es gibt, wie auch bei [Ericson05] ganz genau beschrieben, drei gebräuchliche Möglichkeiten, wie eine AABB in 2D, aber auch in 3D, repräsentiert werden kann. Diese lauten min-max Repräsentation, min-widths Repräsentation und center-radius Repräsentation und werden in den folgenden drei Abschnitten erklärt und in Abbildung 5 ersichtlich gemacht.

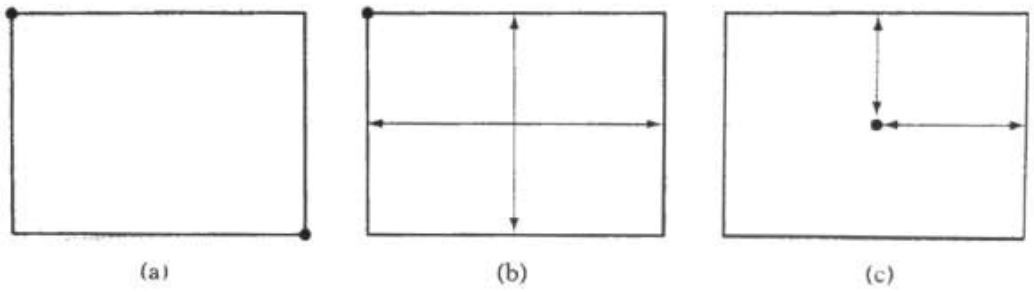

Abbildung 5: Die drei ABB-Repräsentationen: (a) min-max, (b) min-widths, und (c) center-radius [Ericson05].

3.1.1. min-max Repräsentation

Diese Repräsentation speichert nur die „Extrempunkte“ der AABB. Dabei hat eine AABB, egal ob zwei- oder dreidimensional, genau zwei. Der erste Punkt ist derjenige, der in allen Achsenrichtungen die geringsten Werte enthält, deswegen heißt dieser „min“. Der zweite Punkt enthält die höchsten Werte in allen Achsenrichtungen, deswegen auch „max“. Ein Beispiel hierfür ist in Abbildung 5(a) zu sehen.

Hier eine einfache Möglichkeit zur Implementierung dieser Datenstruktur als „struct“ in C++ [Ericson05]:

```
struct AABB
{
     Vector min;
     Vector max;
};
```

Ein „Vector“ ist hierbei eine Klasse oder eine Struktur, die einen Punkt oder einen Vektor im Koordinatensystem repräsentiert. Solch ein Vektor beinhaltet entweder zwei oder auch drei Fließkommawerte, darauf ankommend, in welcher Dimension gearbeitet wird.

3.1.2. min-widths Repräsentation

Eine weitere Möglichkeit zur Repräsentation einer AABB ist die min-widths Repräsentation. Sie ist der min-max Repräsentation sehr ähnlich, nur mit dem Unterschied, dass statt dem „max“-Wert die vom „min“-Wert ausgehenden Abstände bis zu den einzelnen Rändern der AABB in allen Achsenrichtungen gespeichert werden. Hierbei sind es zwei, oder auch drei Fließkommawerte, die die Weite der AABB in allen Richtungen von „min“ aus beschreiben. Ein Beispiel ist in Abbildung 5(b) zu sehen.

Eine Möglichkeit zu Implementierung findet sich hier [Ericson05]:

```
struct AABB
{
     Vector min;
     float d[3];
};
```

In diesem Fall werden drei Fließkommazahlen gespeichert, was für die dritte Dimension gilt. Für die zweite Dimension bräuchte man nur zwei Werte. Im Grunde kann auch ein Vektor gespeichert werden anstatt dem float-Array d[3].

3.1.3. center-radius Repräsentation

Diese Repräsentation ist mit der vorherigen zu vergleichen, nur werden nicht die Abstände von „min“ aus berücksichtigt, sondern vom Punkt im Zentrum der AABB aus. Es wird somit ein Mittelpunkt gespeichert und jeweils für jede Achse die Entfernung zum Rand, sozusagen der „Radius“. In Abbildung 5(c) ist dies ersichtlich.

Hier eine Möglichkeit zur Implementierung [Ericson05]:

```
struct AABB
{
     Vector c;
     float r[3];
};
```

3.1.4. Vergleich

Laut [Ericson05] muss bei der min-max Repräsentation beim Neuberechnen der AABB, z.B. aufgrund von Transformationen des beinhaltenden Objekts, sowohl der „min“, als auch der „max“-Wert neu berechnet werden. Bei den anderen Repräsentationen ist dies wegen den relativen Entfernungswerten nicht nötig, da muss nur der „min“-Wert bzw. der Mittelpunkt neu berechnet werden. Zudem sind die Radien bzw. Abstände, da sie ja relative Werte sind, oft kleiner sind als die globalen, absoluten Werte des „max“ in der min-max Repräsentation. Zudem ist der Vorteil der center-radius Repräsentation, dass sie mit dem Theorem des „Separating Axis Tests“ (siehe Kapitel 4.2. Überschneidung zweier OBBs austesten) kompatibel ist, mit dem auch Tests mit anderen BVs möglich sind.

3.2. Überschneidung zweier AABBs austesten

Die Tests zum Überschneiden zweier AABBs sind, egal um welche Repräsentation es sich handelt, vom Sinn und vom Ziel her immer gleich, wie bei [Endres08] kurz und bündig beschrieben: *„Überschneidung (von zwei AABBs) findet statt genau dann, wenn sie sich in Richtung aller 3 Koordinatenachsen überschneiden“*.

Somit braucht nur für jede Koordinatenachse überprüft werden, ob sich die AABBs überschneiden, wobei eine Überschneidung nur dann stattfindet, wenn sie sich auf allen Achsen zugleich überschneiden. Hierzu gibt es in den nächsten drei Abschnitten für jede der drei Repräsentationen eine mögliche Implementierung aus [Ericson05].

3.2.1. Implementierungsbeispiel für min-max Repräsentation

In diesem Beispiel wird geprüft, ob sich die Seiten der AABBs, die parallel zu den jeweiligen Koordinatenachsen sind, sich nicht überschneiden. Dadurch kann schon früh eine Kollision ausgeschlossen werden. In dieser Repräsentation wird einfach nur sichergestellt, ob die jeweiligen „min“-Werte der einen AABB nicht größer sind als die dazugehörigen „max“-Werte der anderen.

```
int TestAABBAABB(AABB a, AABB b)
{
     //Teste für jede Achse, ob sie sich NICHT überschneiden!
     if(a.max[0] < b.min[0] || a.min[0] > b.max[0])
          return 0;
     if(a.max[1] < b.min[1] || a.min[1] > b.max[1])
          return 0;
     if(a.max[2] < b.min[2] || a.min[2] > b.max[2])
          return 0;

     //Alle Seiten überschneiden sich auf jeder Achse!
     return 1;
}
```

Eine visuelle Darstellung dieses Tests in 2D, also mit nur zwei Achsen anstatt von drei wie im Implementierungsbeispiel, ist in Abbildung 6 zu sehen.

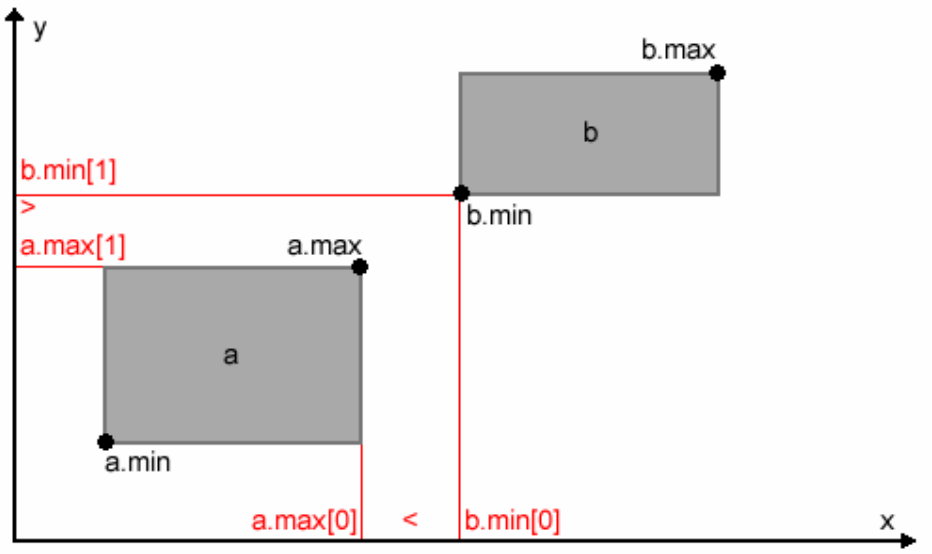

Abbildung 6: Der Überschneidungstest zweier AABBs mit min-max Repräsentation.

3.2.2. Implementierungsbeispiel für min-widths Repräsentation

Im Gegensatz zum vorherigen Test ist diese Implementierung im Bezug auf die Anzahl der zu durchführenden Operationen nicht vergleichbar. Hier wird der Abstand t zwischen den jeweiligen „min“-Werten pro Achse berechnet und überprüft, ob dieser Wert von t größer ist als das d von der einen AABB oder der negative Wert von t größer als der Wert von d von der anderen AABB ist. Ist dies der Fall, dann findet keine Überschneidung statt.

```
 int TestAABBAABB(AABB a, AABB b)
{
     float t;

     //Teste jede Achse, ob sie sich NICHT überschneiden!
     if((t = a.min[0] - b.min[0]) > b.d[0] || -t > a.d[0])
          return 0;
     if(((t = a.min[1] - b.min[1]) > b.d[1] || -t > a.d[1])
          return 0;
     if(((t = a.min[2] - b.min[2]) > b.d[2] || -t > a.d[2])
          return 0;

     //Alle Achsen Überschneiden sich! AABB überschneidet sich!
     return 1;
}
```

Eine visuelle Darstellung für die x-Achse dieses Tests in 2D ist in Abbildung 7 zu erkennen.

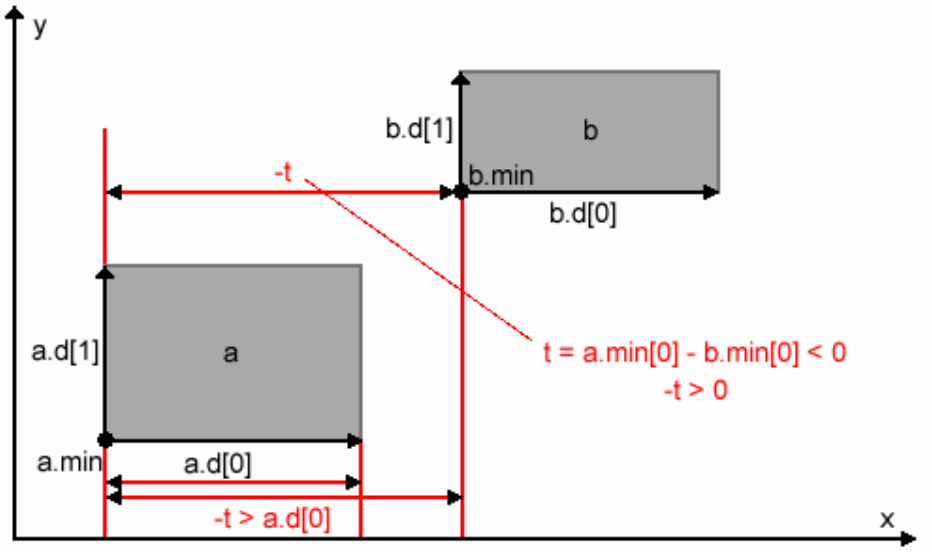

Abbildung 7: Der Überschneidungstest zweier AABBs mit min-widths Repräsentation.

3.2.3. Implementierungsbeispiel für center-radius Repräsentation

Diese Implementierung kann mit dem vorhergehenden Beispiel verglichen werden, nur werden die Abstände der Mittelpunkte berechnet und mit den Summen der Abstände verglichen. Ist der Abstand der Zentren größer als die Summe der beiden Radien (Abstände), dann können sich die beiden AABBs nicht überschneiden.

```
int TestAABBAABB(AABB a, AABB b)
{
     //Teste jede Achse, ob sie sich NICHT überschneiden!
     if((a.c[0] - b.c[0]) > (a.r[0] + b.r[0]))   return 0;
     if((a.c[1] - b.c[1]) > (a.r[1] + b.r[1]))   return 0;
     if((a.c[2] - b.c[2]) > (a.r[2] + b.r[2]))   return 0;

     //Alle Achsen Überschneiden sich! AABB überschneidet sich!
     return 1;
}
```

Eine visuelle Darstellung für die x-Achse dieses Tests in 2D ist in Abbildung 8 zu sehen.

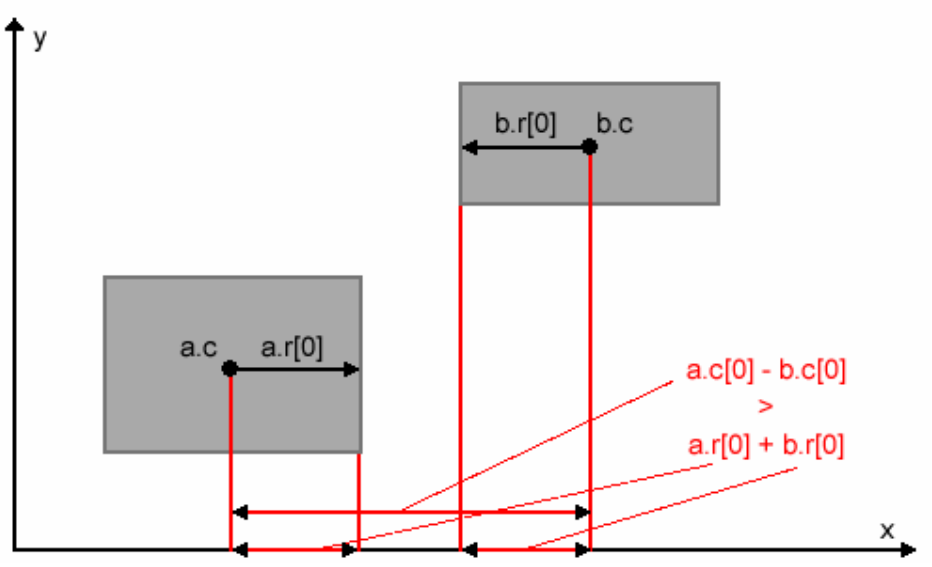

Abbildung 8: Der Überschneidungstest zweier AABBs mit center-radius Repräsentation.

4. Oriented Bounding Box (OBB)

Eine Oriented Bounding Box (deutsch: Orientierte Hüllbox) ist vergleichbar mit einer AABB, nur dass die Seiten der Box nicht an den globalen Koordinatenachsen ausgerichtet sein muss. Ein Beispiel einer OBB in 2D ist auch in Abbildung 2 und eine Repräsentation in der dritten Dimension in Abbildung 9 ersichtlich.

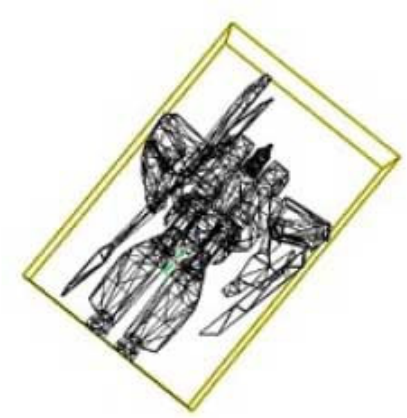

Abbildung 9: Ein Raumschiff in einer OBB [Endres08].

4.1. Darstellung einer OBB

Es gibt mehrere Möglichkeiten zur Repräsentation einer OBB. Im Grunde können folgende Möglichkeiten unterschieden werden [Endres08]:

- acht Eckknoten bzw. Eckpunkte der OBB speichern
- sechs Flächen, die die OBB einschließen
- drei parallele Flächen, sog. Slabs (siehe Kapitel 5. Halfspace Intersection Volumes)
- Zentraler Punkt + Orientierungsmatrix + Längen

Die letztere Darstellung ist im Grunde den anderen vorzuziehen, da sie im Vergleich zu den anderen Repräsentationen den eindeutig schnellsten Überschneidungstest aufweist. Außerdem ist diese Repräsentation die gebräuchlichste und wird somit in den folgenden Abschnitten genauer betrachtet.

Eine Implementierungsmöglichkeit für eine OBB mit der letzteren Darstellung kann somit wie folgt aussehen [Ericson05]:

```
struct OBB
{
     Vector c;
     Vector u[3];
     Vector e;
};
```

Der Punkt c entspricht dem Mittelpunkt der OBB. Die Orientierungsmatrix, welche 3x3 Werte besitzt, wird hier mittels der drei Vektoren gespeichert in „u“ repräsentiert, wobei diese Vektoren die lokalen Koordinatenachsen repräsentieren und parallel zu den Flächen der OBB sind. Der Vektor „e“ beinhaltet die drei Entfernungen von den Rändern zum Mittelpunkt entlang der entsprechenden lokalen Koordinatenachsen in u.

Mit der Anzahl von 15 Gleitkommazahlen in 3D und zehn in 2D benötigt die OBB im Gegensatz zu den anderen BVs mehr Speicherplatz. Es gibt Möglichkeiten, diesen Speicherverbrauch zu senken, wie z.B. die Orientierung nicht als Matrix sondern mit Winkeln oder als so genanntes „Quaternion“ zu speichern, welche dann nur drei bzw. vier Gleitkommazahlen besitzen anstatt von neun wie in der Matrix. Der Nachteil dabei ist aber, dass diese Werte für die Überschneidungstests zurückkonvertiert werden müssen in eine Matrix. Diese Operationen sind aber sehr aufwendig.

Die beste Methode zum Sparen von Speicher wäre aber, nur zwei lokale Achsen zu speichern statt drei und die dritte mittels des Kreuzprodukts der zwei anderen zu berechnen. Dies ist deshalb möglich, da die Flächen der OBB ja rechtwinkelig zueinander stehen. Dieses Errechnen des dritten Werts verbraucht nur wenig CPU-Zeit und der Speicherverbrauch kann dabei verringert werden.

4.2. Überschneidung zweier OBBs austesten

Im Gegensatz zu den vorhergehenden BVs ist der Überschneidungstest für OBBs relativ kompliziert. In der einfachsten Form wäre ein Überschneidungstest zweier OBBs so durchzuführen, dass alle Vertices der einen OBB außerhalb der Flächen, definiert durch die Seiten der anderen OBB, liegen müssen, und umgekehrt. Solch ein Test funktioniert in 2D korrekt, aber in 3D gibt es ein Problem, und zwar wird eine Überlappung erkannt, wenn eine Kante einer OBB lotrecht an einer anderen Kante der zweiten OBB steht [Ericson05]. In diesem Fall liegt kein Vertex außerhalb der Flächen und es wird eine Überlappung erkannt, obwohl keine vorliegt. Dieser einfache Test hat trotzdem seinen Nutzen, auch wenn er zusätzliche, nicht vorhandene Kollisionen erkennt. Er kann somit als Vortest für einen genaueren Test dienen.

Ein genauerer Test kann, wie bei [Gottschalk96] beschrieben, mit dem Theorem des „Separating Axis Tests“ implementiert werden. Dieses Theorem beruht darauf, dass sich zwei konvexe Objekte nicht überlappen, wenn die Summe der Radien rA und rB vom Mittelpunkt der Objekte aus, projiziert auf eine Achse L, kleiner ist als der Abstand der Mittelpunkte (T), projiziert auf dieselbe Achse ($|T \cdot L|$).

Eine Darstellung der Funktionsweise des „Separating Axis Tests“ auf zwei OBBs ist in Abbildung 10 ersichtlich.

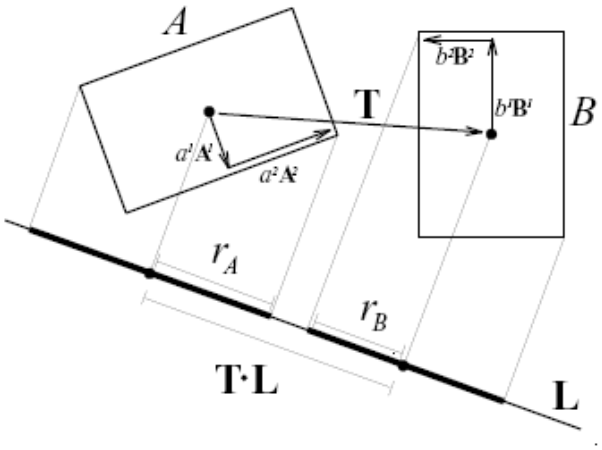

Abbildung 10: Darstellung des „Separating Axis Tests“ mit zwei OBBs [Gottschalk96].

Bei OBBs ist zu erkennen, dass zumindest 15 dieser Achsen von Nöten sind um eine Überlappung korrekt zu erkennen [Gottschalk96]. Diese Achsen sind die drei Koordinatenachsen der einen OBB (A) und die drei der anderen OBB (B) gespeichert in u (siehe vorheriges Kapitel) und die neun Achsen, die die Kreuzprodukte aus den drei Achsen von A mit jeweils den drei Achsen von B sind.

Die Nummer der Operationen in diesem Test kann reduziert werden, indem B in das Koordinatensystem von A transformiert wird. Wenn t der Translationsvektor von A nach B und r die Rotationsmatrix, um B ins Koordinatensystem von A zu bringen, ist, dann kann der Ablauf der Tests der einzelnen Achsen wie in der folgenden Tabelle zusammengefasst vonstatten gehen.

Nr	L	|T·L|	rA	rB
1.	u0A	|t0|	e0A	e0B |r00| + e1B |r01| + e2B |r02|
2.	u1A	|t1|	e1A	e0B |r10| + e1B |r11| + e2B |r12|
3.	u2A	|t2|	e2A	e0B |r20| + e1B |r21| + e2B |r22|
4.	u0B	|t0r00 + t1r10 + t2r20|	e0A|r00| + e1A|r10| + e2A|r20|	e0B
5.	u1B	|t0r01 + t1r11 + t2r21|	e0A|r01| + e1A|r11| + e2A|r21|	e1B
6.	u2B	|t0r02 + t1r12 + t2r22|	e0A|r02| + e1A|r12| + e2A|r22|	e2B
7.	u0A x u0B	|t2r10 - t1r20|	e1A|r20| + e2A|r10|	e1B|r02| + e2B|r01|
8.	u0A x u1B	|t2r11 - t1r21|	e1A|r21| + e2A|r11|	e1B|r02| + e2B|r00|
9.	u0A x u2B	|t2r12 - t1r22|	e1A|r22| + e2A|r12|	e1B|r01| + e2B|r00|
10.	u1A x u0B	|t0r20 - t2r00|	e0A|r20| + e2A|r00|	e1B|r12| + e2B|r11|
11.	u1A x u1B	|t0r21 - t2r01|	e0A|r21| + e2A|r01|	e1B|r12| + e2B|r10|
12.	u1A x u2B	|t0r22 - t2r02|	e0A|r22| + e2A|r02|	e1B|r11| + e2B|r10|
13.	u2A x u0B	|t1r00 - t0r10|	e0A|r10| + e1A|r00|	e1B|r22| + e2B|r21|
14.	u2A x u1B	|t1r01 - t0r11|	e0A|r11| + e1A|r01|	e1B|r22| + e2B|r20|
15.	u2A x u2B	|t1r02 - t0r12|	e0A|r12| + e1A|r02|	e1B|r21| + e2B|r20|

Tabelle 1: Die Berechnung der 15 zu testenden Achsen für den „Separating Axis Test“ [Ericson05].

Ein Implementierungsbeispiel für diesen Test kann im Anhang eingesehen werden.

5. Halfspace Intersection Volume

Mit der Ausnahme von Bounding-Spheres können die meisten BVs so definiert werden, dass sie aus den Überschneidungen von mehreren Ebenen bestehen, die ein geschlossenes Volumen bilden. Zum Beispiel können AABBs und OBBs auch als der Inhalt von den Überschneidungen von sechs Flächen interpretiert werden, wo alle Flächen normal aufeinander stehen.

5.1. Slab-based Volume

Zuerst von Kay und Kajiya in [Kay86] eingeführt sind die Slab-based Volumes die ersten Halfspace Intersection Volumes, die als Bounding Volume in Betracht gezogen wurden. Sie basieren auf die Überschneidung von mehreren „Slabs“, wobei eine „Slab“ eine unendliche Region von Raum zwischen zwei parallelen Ebenen ist. Dieser Raum wird dann mit mehreren anderen „Slabs“ endlich gemacht und dieses endliche Volumen ist dann das Innere des BV.

Solch eine „Slab“ wird mittels eines Einheitsvektors, der der Normalvektor der beiden Ebenen ist (n), und zwei Skalaren (d_{near} und d_{far}) dargestellt, welche die Abstände der Ebenen vom Ursprung entlang dieses Normalvektors entsprechen. In Abbildung 11 ist ein Beispiel für eine „Slab“ in 2D ersichtlich.

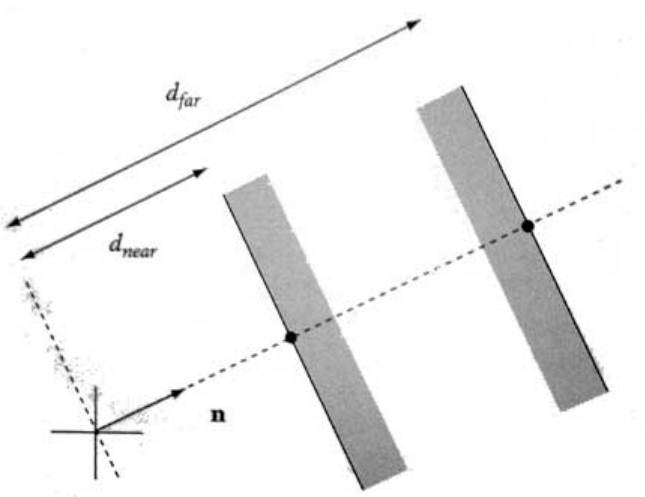

Abbildung 11: Eine Darstellung einer „Slab“ in 2D [Ericson05].

Dieses BV ist als Vorgänger der Discrete Orientation Polytopes (kurz: k-DOP) zu betrachten, welche die Slab-based Volumes ablösen. Aus diesem Grund werden die k-DOPs in den folgenden Abschnitten genauer erklärt.

5.2. Discrete Orientation Polytope (k-DOP)

Das Prinzip des Discrete Orientation Polytope (deutsch: Diskret orientierter Polyeder = Vielflächner) (kurz: k-DOP) wurde zuerst bei [Klosowski98] und [Konečný98] eingeführt, wobei sie in [Konečný98] unter dem Namen „Fixed-direction hull“ (deutsch: fixiert-ausgerichtete Hülle) verwendet werden. Es handelt sich bei beiden Autoren aber um dasselbe Prinzip und funktioniert nach demselben Prinzip wie die Slab-based Volumes. Sie bestehen aus mehreren Paaren von parallelen Flächen deren Schnittpunkte und Schnittflächen einen Innenraum, sprich ein Volumen definieren. Der einzige Unterschied zu den Slab-based Volumes ist, dass die Ausrichtungen dieser Flächen, also deren Normalen, schon im Vorhinein festgelegt sind, was dazu führt, dass die Überschneidungstests schneller vonstatten gehen können. Sie sind, wie auch bei [Ericson05] und bei [Konečný98] beschrieben, mit den Tests der AABB vergleichbar, nur dass nicht nur auf drei bzw. zwei Achsen des Koordinatensystems getestet werden muss, sondern auf k/2 Achsen, wobei das „k“ in „k-DOP“ für die Anzahl der einzelnen Flächen, die das Objekt umschließen, steht. Dabei ist zu beachten, das eine 6-DOP in 3D, deren Flächen an den Achsen des Koordinatensystems ausgerichtet sind, einer AABB entspricht. Somit ist ein k-DOP mit k=6 eine Generalisierung einer dreidimensionalen und ein 4-DOP in 2D eine Generalisierung einer zweidimensionalen AABB.

Es gibt aber noch weitere Möglichkeiten für k-DOPs [Klosowski98], vor allem im dreidimensionalen Raum. Eine weitere wäre die 14-DOP. Sie besteht aus den sechs Normalen der 6-DOP, sprich der dreidimensionalen AABB, kombiniert mit acht diagonalen Ebenen, die die einzelnen acht Ecken der AABB „abschneiden“. Eine weitere Möglichkeit, die 18-DOP, enthält die sechs Normalen der AABB, kombiniert mit den acht diagonalen der 14-DOP und mit zusätzlich vier diagonalen, die vier Kanten (entweder in x-, y- oder z-Richtung) der AABB „abschneiden“. Schlussendlich kann noch eine 26-DOP erzeugt werden, indem zu der 18-DOP noch die acht übrig gebliebenen Kanten „abgeschnitten“ werden.

Die Anzahl der Ebenen kann aber variiert werden, sodass auch 10-DOPs und 8-DOPs erzeugt werden können. Es müssen nur die Normalen der einzelnen Ebenen definiert werden, wobei diese so gewählt werden sollten, dass das BV das zu umschließende Objekt so gleichmäßig wie möglich umhüllen kann. Dafür eignen sich vor allem die Vektoren, die normal auf die Koordinatenachsen stehen, sprich (±1, 0, 0), (0, ±1, 0), und (0, 0, ±1) und auch die Vektoren, die diagonal stehen, sprich jede mögliche Kombination aus +1, -1 und 0. Eine Kombination aus allen Möglichkeiten entspricht dann einer 26-DOP in 3D.

5.2.1. Darstellung einer k-DOP

Da bei einer k-DOP die Normalen schon im Vorhinein bekannt sind, müssen nur die minimalen und maximalen Abstände der Ebenen entlang jeder Normalen gespeichert werden. Zum Beispiel würde die Speicherstruktur einer 8-DOP wo k=8 ist wie folgt aussehen [Ericson05].

```
struct 8DOP
{
     float min[4];
     float max[4];
};
```

Die vier Werte in den float-Arrays „min“ und „max“ entsprechen den minimalen und maximalen Abständen der Flächen vom Ursprung des Koordinatensystems aus. Die vier Normalen können so ausgewählt werden, dass sie in 3D in den Richtungen (±1, ±1, ±1) verweisen. In 2D können diese vier Normalen so gewählt werden, wie in Abbildung 12 zu sehen ist, sodass sie nach (0, 1), (1, 1), (1, 0), (1, -1) weisen.

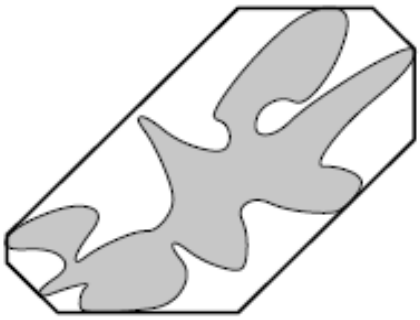

Abbildung 12: Eine zweidimensionale 8-DOP [Klosowski98].

5.2.2. Überschneidung zweier k-DOPs austesten

Der Überschneidungstest zweier k-DOPs ist mit dem zwischen AABBs vergleichbar. Zwischen k-DOPs muss auf k/2 Achsen, die den Normalen entsprechen, getestet werden. Im Unterschied muss bei AABBs durch die drei globalen Koordinatenachsen getestet werden (siehe Kapitel 3.2. Überschneidung zweier AABBs austesten). Der Hintergrund und die Logik der beiden Tests sind aber dieselben.

Schlussendlich kann eine Implementierung folgendermaßen aussehen [Ericson05]:

```
int TestKDOPKDOP(KDOP &a, KDOP &b, int k)
{
     //Intervalle überlappen sich nicht?
     for(int i = 0; i < k; i++)
          if(a.min[i] > b.max[i] || a.max[i] < b.min[i])
               return 0;

     //Alle Intervalle überlappen sich!
     return 1;
}
```

6. Weitere Möglichkeiten für Bounding Volumes

Zusätzlich zu den vorgestellten BVs, wurden noch einige andere Möglichkeiten in Betracht gezogen. Dazu gehören auch Kegel und Zylinder, wie sie auch bei [Held97] für verschiedenste Überschneidungstests verwendet werden.

Eine weitere, häufig verwendete Form für ein Bounding Volume sind die so genannten Swept Sphere Volumes, wie sie auch bei [Larsen99] Verwendung finden. Das Prinzip dieser BVs ist so zu verstehen, dass sie aus einer primitiven Form, wie einem einfachen Punkt, einer Linie oder einem Rechteck, mit einem Abstand nach außen, bestehen. Die Variante mit dem Punkt, auch „Point Swept Sphere“ oder kurz PSS genannt, entspricht dabei einer Kugel. Die Variante mit der Linie, auch „Line Swept Sphere“ oder LSS genannt, ist ein Zylinder mit abgerundeten Enden und das Rechteck, auch „Rectangle Swept Sphere“ oder kurz RSS genannt, ist ein Rechteck mit abgerundeten Seiten. Beispiele für alle drei Möglichkeiten sind in Abbildung 13 zu erkennen.

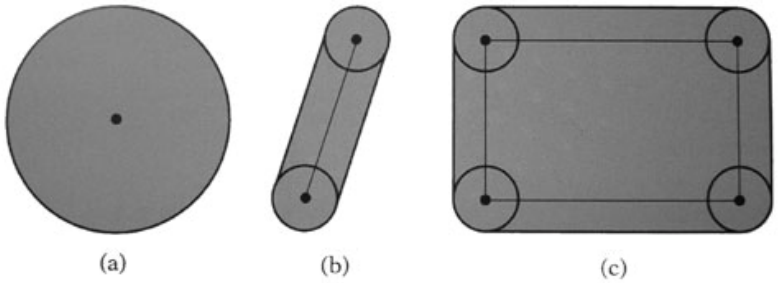

Abbildung 13: (a) Eine Point Swept Sphere, (b) Eine Line Swept Sphere und (c) eine Rectangle Swept Sphere [Ericson05].

Eine weitere Form für ein BV ist die bei [Krishnan98] vorgestellte „Spherical Shell“ (deutsch: Kugelschale). Eine „Spherical Shell“ ist eine Überschneidung des Volumens zweier Kugeln mit demselben Mittelpunkt und verschiedenen Radien mit einem Kegel, dessen Spitze im Zentrum der beider Kugeln liegt. Ein visuelles Beispiel für eine „Spherical Shell“ ist in Abbildung 14 ersichtlich.

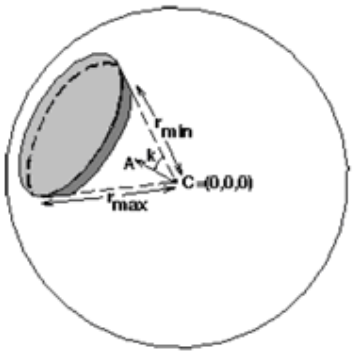

Abbildung 14: Eine „Spherical Shell“ in 3D [Krishnan98].

Eine weitere Möglichkeit für ein BV sind die in [Guibas03] vorgestellten „Zonotopes“, speziell die zwei- und dreidimensionalen Varianten namens „Zonogon“ und „Zonohedron“. Ein „Zonotope“ ist im Allgemeinen ein Polyeder, sprich ein Vielflächner so wie die k-DOPs, der aus der „Minkowskisumme“ von Geradenabschnitten besteht. Ein Beispiel für eine „Zonotope“ ist in Abbildung 15 zu sehen.

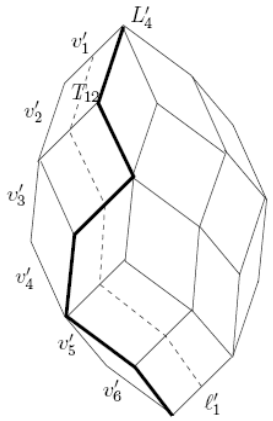

Abbildung 15: Ein Beispiel für ein „Zonotope“ in 3D, ein „Zonohedron“ [Guibas03].

Das Berechnen der „Minkowskisumme“ kann folgendermaßen erklärt werden [Ericson05]: Wenn A und B zwei Objekte mit Punkten und a und b die Positionsvektoren von Paaren von Punkten von A und B sind, dann ist die „Minkowskisumme“, A⊕B, wie in der folgenden Formel (1) definiert.

$$A \oplus B = \{ a + b : a \in A, b \in B \} \qquad (1)$$

Die Minkowskisumme kann visuell auch als eine Region von A angesehen werden, die nach jeden Punkt in B translatiert wurde und umgekehrt. Ein visuelles Beispiel für eine „Minkowskisumme“ zweier Objekte ist in Abbildung 16 zu sehen.

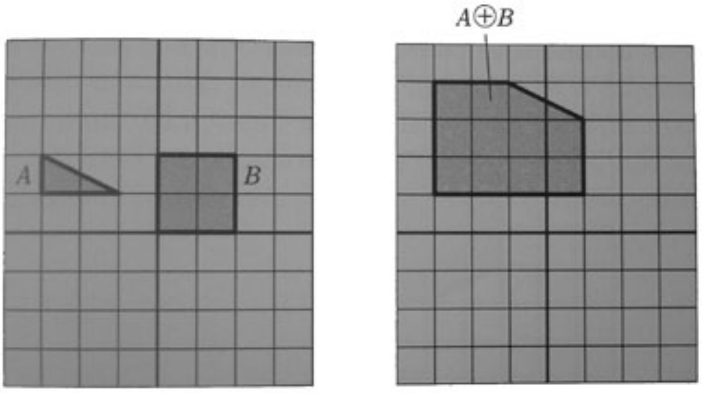

Abbildung 16: Die „Minkowskisumme“ eines Dreiecks A und eines Quadrats B [Ericson05].

Zuletzt gibt es noch die „Convex Hull“ (deutsch: konvexe Hülle), das BV welches im Vergleich zu den anderen Möglichkeiten am engsten an einem Objekt anliegt. Es umschließt ein Objekt mit einzelnen Flächen, so wie die Halfspace Interesection Volumes, wobei bei der „Convex Hull“ die Flächen und deren Schnittpunkte so eng wie möglich anliegen. Bei Objekten, die selbst nur aus konvexen Polygonen bestehen, umschließt die „Convex Hull“ jene ohne Freiräume zu hinterlassen. Als Analogie kann eine „Convex Hull“ auch als ein Gummiband angesehen werden, welches das Objekt so eng wie möglich umschließt. Diese Variante eines BV ist eher für sehr genaue Tests zu gebrauchen, denn die Überlappungstests sind sehr komplex. Ein Beispiel für eine konvexe Hülle in 2D ist in Abbildung 17 zu erkennen.

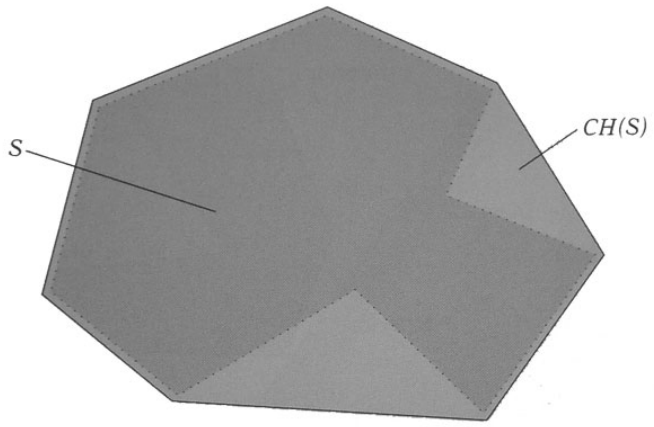

Abbildung 17: Eine konvexe Hülle CH(S) umhüllt ein nicht konvexes Objekt S [Ericson05].

7. Conclusion

Das Verwenden von Bounding Volumes ist für die Collision Detection in der Computergrafik sehr wichtig und hilft die Performance zu steigern, indem Algorithmen mit weniger Zeitaufwand verwendet und die Anzahl der zu durchführenden Operationen verringert werden. Durch das Ausschließen von Kollisionen bzw. Überlappungen der Bounding Volumes kann schon im Vorhinein, ohne komplexe Überlappungstests der Objekte selbst, festgestellt werden, ob die Objekte kollidieren.

Ein Bounding Volume ist hierbei eine möglichst einfache, geometrische Form, die schnelle mathematische Tests zur Überprüfung von Überschneidungen mit sich selbst und anderen Objekten, wie andere Bounding Volumes oder Primitive, aufweist. In dieser Arbeit wurden die einzelnen Bounding Volumes analysiert und für die gebräuchlichsten BVs ein Überschneidungstest mit anderen BVs desselben Typs in der Form von Quellcode dargelegt. Überschneidungstests mit anderen geometrischen Objekten, wie Linien, Dreiecke, etc. wurden nicht gezeigt. Das Laufzeitverhalten, also die möglichen Änderungen des BV während der Laufzeit aufgrund von Transformationen oder Rotationen des beinhaltenden Objekts, wurden nicht bearbeitet.

Es können viele Varianten von geometrischen Figuren als BV fungieren. Dabei ist aber auch auf den Kontext zu achten, in denen die einzelnen BVs eingesetzt werden sollen, denn nicht jede Applikation und Situation benötigt dieselben Anforderungen auf die einzelnen Teilaspekte der Möglichkeiten für BV. Dazu gehören der Speicherverbrauch, der bei der Sphere am geringsten und bei der OBB am höchsten ist, aber auch die Tatsache, wie eng ein Objekt mit einem gewissen BV umhüllt werden kann. Diese zwei Aspekte verhalten sich umgekehrt proportional, sodass ein BV mit besserer Passgenauigkeit auch gleichzeitig mehr Speicherplatz benötigt. Somit ist und bleibt die Relation zwischen den einzelnen Eigenschaften immer Ansichtssache und ist von der jeweiligen Situation bzw. Applikation abhängig.

Die Auswahl des BV ist eine Sache des Entwicklers und ist einerseits Geschmackssache und andererseits abhängig von den Umständen, in denen die Applikation laufen soll. Es muss bedacht werden, welche Überlappungstests am häufigsten durchgeführt werden müssen, denn es kann sein, dass bei einer bestimmten Applikation auch Überschneidungen mit Geraden und BVs oft vorkommen. Hierbei muss analysiert werden, welches BV den schnellsten Durchschnittstest mit Geraden besitzt. Aber nicht nur mit Geraden und Geradenabschnitten muss oft getestet werden, sondern auch mit anderen geometrischen und mathematischen Objekten wie Ebenen, Dreiecke, etc. Somit ist eine genaue Planung im Vorhinein unerlässlich, um gewisse BV ausschließen zu können bzw. um eine Auswahl von möglichen Kandidaten zu erlangen.

Was in dieser Arbeit nicht behandelt wurde, aber in der Regel angewendet wird, ist das Aufbauen von Bounding Volume Hierarchies, wie es auch in [Gottschalk96] mit OBBs, in [Klosowki98] mit k-DOPs und in [Larsen99] mit Swept Sphere Volumes der Fall ist. Eine Bounding Volume Hierarchy ist hierbei, wie der Name schon vermuten lässt, eine hierarchische Speicherstruktur zum Speichern mehrerer BVs derselben Art, oder auch seltener von verschiedener Art, für ein zu umhüllendes Objekt.

Ein visuelles Beispiel einer Bounding Volume Hierarchy, in diesem Fall mit OBBs, ist in Abbildung 18 zu sehen, wobei da von einem so genannten OBB-Tree gesprochen wird.

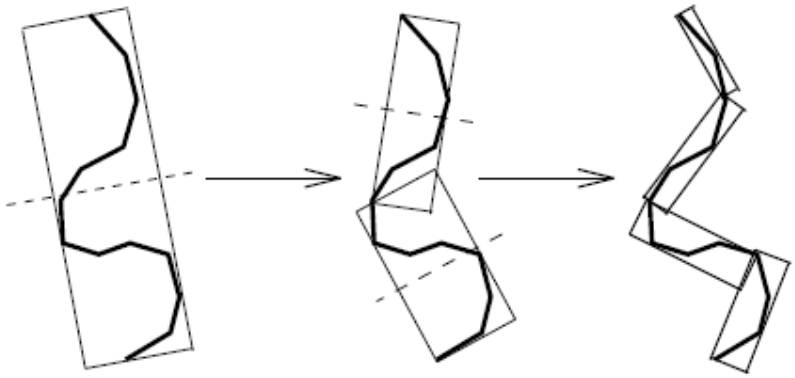

Abbildung 18: Der schrittweise Aufbau eines OBB-Tree für ein Objekt [Gottschalk96].

Durch diese Art des hierarchischen Aufbaus von mehreren BVs für ein Objekt können viel enger anliegende Ergebnisse erlangt werden. Durch die Entscheidung der Tiefe des Baumes bzw. der Hierarchie hat der Programmierer die Möglichkeit die Genauigkeit der Umhüllung zu bestimmen, denn umso größer der Baum bzw. je tiefer die Hierarchie ist, desto genauer kann das Objekt umschlossen werden. Ein anschauliches Beispiel hierzu ist in Abbildung 19 zu sehen. Hier wird ein Modell eines Flugzeugs mit k-DOPs umhüllt. Wie zu erkennen ist, steigt die Genauigkeit der Hülle mit der Anzahl der k-DOPs. Andererseits steigt durch die Anzahl der BVs auch die Anzahl der zu durchführenden Tests, was zu Performanceeinbußen führen kann. Aber im Großen und Ganzen hat der Entwickler, da er die Tiefe der Hierarchie bestimmen kann, nun um einiges mehr an Kontrolle über die Performance als zuvor.

Schlussendlich kann der Entwickler frei entscheiden, was für BVs er verwenden möchte. Dabei kann er sich nicht nur überlegen, welches BV für seine Applikation am besten geeignet ist, sondern auch, wie er seine Speicherstrukturen verwalten möchte, sprich, ob er pro zu umhüllendes Objekt ein BV verwendet oder gleich mehrere, die dann in einer Hierarchie bzw. in einer Baumstruktur gespeichert sind.

Überdies gibt es schon viele fertige, vor allem über das Internet erhältliche, Implementierungen von Collision Detection Frameworks. Beispiele für solche Frameworks wären [I-Collide], das konvexe Polyeder verwendet oder [V-Collide], das OBB-Trees und polygonale Überschneidungstest verwendet, welche wiederum aus dem Framework [RAPID] stammen. All diese Frameworks basieren großenteils auf die in dieser Arbeit vorgestellten Varianten. Somit kann mit dem Wissen dieser Arbeit verstanden werden, wie die einzelnen Frameworks grob funktionieren bzw. es kann auch entschieden werden, welches Framework am besten zu den Anforderungen der eigenen Applikation passt.

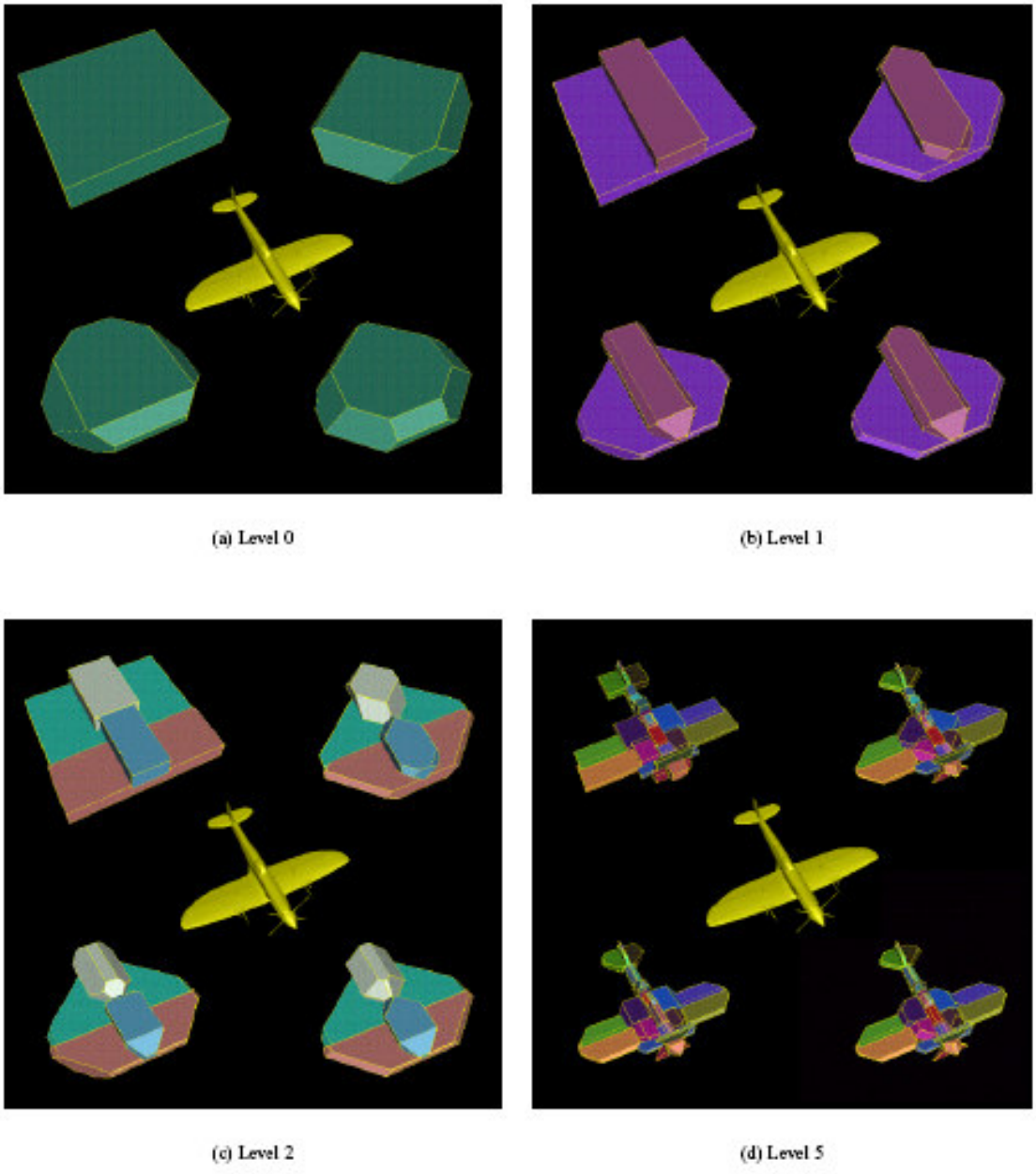

Abbildung 19: Verschiedene Tiefen von Hierarchien mit 6-, 14-, 18- und 26-DOPs [Klosowski98].

Literaturverzeichnis

[Endres08] Endres, Jürgen (2008). Bounding Volumes [online], Erlangen, Universität Erlangen. Verfügbar bei: http://www10.informatik.uni-erlangen.de/de/Teaching/Courses/SS2008/RBD/bounding_volumes.pdf
[Zugang am 21.Dezember 2008].

[Ericson05] Ericson, Christer (2005). Real-Time Collision Detection, Elsevier.
ISBN-13: 978-1-55860-732-3, 1.Ed.

[Gottschalk96] Gottschalk, Stefan. Ming Lin. Dinesh Manocha (1996). OBBTree: A Hierarchical Structure for Rapid Interference Detection, Computer Graphics (SIGGGRAPH 1996 Proceedings), 171-180.

[Guibas03] Guibas, Leonidas. An Nguyen. Li Zhang (2003). Zonotopes as Bounding Volumes, Proceedings of 14th Annual ACM-SIAM Symposium on Discrete Algorithms, 803-812.

[Held97] Held, Martin (1997). ERIT: A Collection of Efficient and Reliable Intersection Tests, Journal of Graphics Tools, 2(4), 25-44.

[I-Collide] I-Collide [online], Department of Computer Science, University of North Carolina, Chapel Hill. Verfügbar bei: http://www.cs.unc.edu/~geom/I_COLLIDE/
[Zugang am 10.Januar 2009].

[Kay86] Kay, Timothy. James Kajiya (1986). Ray Tracing Complex Scenes, Computer Graphics (SIGGRAPH 1986 Proceedings), 20(4), 269-278.

[Klosowski98] Klosowski, James. Martin Held. Joseph Mitchell. Henry Sowizral. Karel Zikan (1998). Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs, IEEE Transactions on Visualization and Computer Graphics, 4(1), 21-36.

[Konečný98] Konečný, Petr (1998). Bounding Volumes in Computer Graphics, Master's Thesis, Fakultät für Informatik, Masaryk Universität, Brno, Tschechische Republik.

[Krishnan98] Krishnan, Shankar. Amol Pattekar. Ming Lin. Dinesh Manocha (1998). Spherical Shell: A Higher-order Bounding Volume for fast Promixity Queries, Robotics: The Algorithmic Perspective: WAFR 1998, 177-190.

[Larsen99] Larsen, Eric. Stefan Gottschalk. Ming Lin. Dinesh Manocha (1999). Fast Promixity Queries with Swept Sphere Volumes, Technical Report TR99-018, Department of Computer Science, University of North Carolina, Chapel Hill.

[RAPID] RAPID – Robust and Accurate Polygon Interference Detection [online], Department of Computer Science, University of North Carolina, Chapel Hill.
Verfügbar bei: http://www.cs.unc.edu/~geom/OBB/OBBT.html [Zugang am 10.Januar 2009].

[V-Collide] V-Collide – Collision Detection for Arbitrary Polygonal Objects [online], Department of Computer Science, University of North Carolina, Chapel Hill.
Verfügbar bei: http://www.cs.unc.edu/~geom/V_COLLIDE/ [Zugang am 10.Januar 2009].

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AABB	Axis-aligned Bounding Box (deutsch: an Achsen ausgerichtete Hüllbox).
BV	Bounding Volume (deutsch: einhüllendes Volumen).
k-DOP	Discrete Orientation Polytope (deutsch: diskret orientierter Polyeder) (Polyeder = Vielflächner). Das „k“ steht für die Anzahl der Flächen.
LSS	Line Swept Sphere. Eine Möglichkeit für ein Swept Sphere Volume.
OBB	Oriented Bounding Box (deutsch: orientierte Hüllbox).
PSS	Point Swept Sphere. Eine Möglichkeit für ein Swept Sphere Volume.
RSS	Rectangle Swept Sphere. Eine Möglichkeit für ein Swept Sphere Volume.

Anhang

Der folgende Quellcode ist eine mögliche Implementierung für einen Überschneidungstest zweier OBBs nach dem Theorem des „Separating Axis Tests“ und nach der in der Tabelle 1 auf Seite 14 aufgezeigten Reihenfolge der Operationen. Die Funktion „Dot“ berechnet das Skalarprodukt zweier Vektoren (für Details zu dieser Funktion siehe Kapitel 2.2. Überschneidung zweier Spheres austesten).

```
int TestOBBOBB(OBB &a, OBB &b)
{
     float ra, rb;       //Radien projiziert auf die Achse L!
     Matrix33 R, AbsR;   //Matrizen für die Rotation!

     //Erstelle Rotationsmatrix R!
     for(int i = 0; i < 3; i++)
          for(int j = 0; j < 3; j++)
               R[i][j] = Dot(a.u[i], b.u[i]);

     //Erstelle Translationsvektor t!
     Vector t = b.c - a.c;
     //Translationsvektor in A's Koordinatensystem!
     t = Vector(Dot(t,a.u[0]), Dot(t,a.u[1]), Dot(t,a.u[2]));

     //Um Fehler zu vermeiden, eine kleine Konstante EPSILON
     //zu AbsR hinzufügen, um Kreuzprodukt = 0 zu verhindern!
     for(int i = 0; i < 3; i++)
          for(int j = 0; j < 3; j++)
               AbsR[i][j] = R[i][j] + EPSILON;

     //Teste L=A0, L=A1, L=A2 (Tabelle: Zeilen 1 bis 3)!
     for(int i = 0; i < 3; i++)
     {
          ra = a.e[i];
          rb = b.e[0] * AbsR[i][0] + b.e[1] * AbsR[i][1] +
               b.e[2] * AbsR[i][2];
          if(t[i] > ra + rb)  return 0;
     }

     //Teste L=B0, L=B1, L=BA2 (Tabelle 1: Zeilen 4 bis 6)!
     for(int i = 0; i < 3; i++)
     {
          ra = a.e[0] * AbsR[0][i] + a.e[1] * AbsR[1][i] +
               a.e[2] * AbsR[2][i];
          rb = b.e[i];
          if(t[0] * R[0][i] + t[1] * R[1][i] + t[2] *
          R[2][i] > ra + rb)  return 0;
     }
```

```
//Teste Achse L = A0 x B0 (Tabelle 1: Zeile 7)!
ra = a.e[1] * AbsR[2][0] + a.e[2] * AbsR[1][0];
rb = b.e[1] * AbsR[0][2] + b.e[2] * AbsR[0][1];
if(t[2] * R[1][0] - t[1] * R[2][0] > ra + rb)
     return 0;

//Teste Achse L = A0 x B1 (Tabelle 1: Zeile 8)!
ra = a.e[1] * AbsR[2][1] + a.e[2] * AbsR[1][1];
rb = b.e[0] * AbsR[0][2] + b.e[2] * AbsR[0][0];
if(t[2] * R[1][1] - t[1] * R[2][1] > ra + rb)
     return 0;

//Teste Achse L = A0 x B2 (Tabelle 1: Zeile 9)!
ra = a.e[1] * AbsR[2][2] + a.e[2] * AbsR[1][2];
rb = b.e[0] * AbsR[0][1] + b.e[1] * AbsR[0][0];
if(t[2] * R[1][2] - t[1] * R[2][2] > ra + rb)
     return 0;

//Teste Achse L = A1 x B0 (Tabelle 1: Zeile 10)!
ra = a.e[0] * AbsR[2][0] + a.e[2] * AbsR[0][0];
rb = b.e[1] * AbsR[1][2] + b.e[2] * AbsR[1][1];
if(t[0] * R[2][0] - t[2] * R[0][0] > ra + rb)
     return 0;

//Teste Achse L = A1 x B1 (Tabelle 1: Zeile 11)!
ra = a.e[0] * AbsR[2][1] + a.e[2] * AbsR[0][1];
rb = b.e[0] * AbsR[1][2] + b.e[2] * AbsR[1][0];
if(t[0] * R[2][1] - t[2] * R[0][1] > ra + rb)
     return 0;

//Teste Achse L = A1 x B2 (Tabelle 1: Zeile 12)!
ra = a.e[0] * AbsR[2][2] + a.e[2] * AbsR[0][2];
rb = b.e[0] * AbsR[1][1] + b.e[1] * AbsR[1][0];
if(t[0] * R[2][2] - t[2] * R[0][2] > ra + rb)
     return 0;

//Teste Achse L = A2 x B0 (Tabelle 1: Zeile 13)!
ra = a.e[0] * AbsR[1][0] + a.e[1] * AbsR[0][0];
rb = b.e[1] * AbsR[2][2] + b.e[2] * AbsR[2][1];
if(t[1] * R[0][0] - t[0] * R[1][0] > ra + rb)
     return 0;

//Teste Achse L = A2 x B1 (Tabelle 1: Zeile 14)!
ra = a.e[0] * AbsR[1][1] + a.e[1] * AbsR[0][1];
rb = b.e[0] * AbsR[2][2] + b.e[2] * AbsR[2][0];
if(t[1] * R[0][1] - t[0] * R[1][1] > ra + rb)
     return 0;
```

```
        //Teste Achse L = A2 x B2 (Tabelle 1: Zeile 15)!
        ra = a.e[0] * AbsR[1][2] + a.e[1] * AbsR[0][2];
        rb = b.e[0] * AbsR[2][1] + b.e[1] * AbsR[2][0];
        if(t[1] * R[0][2] - t[0] * R[1][2] > ra + rb)
             return 0;

        //Die OBBs müssen überschneiden, da keine Separating Axis
        //gefunden wurde!
        return 1;
}
```