UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Interact - ein interaktives 2D Physikspiel

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von

Daniel Kunz
Martin Leutelt

Betreuer: Dipl.-Inform. Stefan Rilling
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im September 2008

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Motivation

In dieser Studienarbeit sollte ein Spiel entwickelt werden, das intuitiv und einfach zu bedienen ist und gerade deswegen einen gewissen "Suchtfaktor" aufweisen. Das Spielszenario besteht aus einem virtuellen Blatt Papier, auf dem frei Hand gezeichnet werden kann. Es entstehen so zweidimensionale Linien, Kreise oder Polygone, die mit physikalischen Eigenschaften aus der Starrkörperphysik, wie zum Beispiel Gravitation, versehen werden. Ziel des Spiels ist es, durch Zeichnen von Objekten, die dann der Schwerkraft ausgesetzt sind, einen Ball in der Szene so anzustoßen, dass dieser einen Stern berührt, wodurch man das nächste Level erreicht. Die Idee des Spiels bekamen wir durch ein Spiel für das Apple iPhone, das iPhysics genannt wird. Dieses wiederum basiert auf einem Spiel mit dem Namen CrayonPhysics. *interact* ist diesen beiden Spielen sehr ähnlich, allerdings existiert iPhysics ausschliesslich auf dem iPhone und CrayonPhysics ist auf Microsoft Windows beschränkt.

Die Level selbst wie auch die gesamte Grafik des Spiels sind bewusst einfach gehalten, da dies gerade die Einfachheit und damit den Spielspaß garantiert. Es ist zudem möglich selbst Levels zu erstellen. Wahlweise kann das Spiel auch mit der derzeit sehr populären Nintendo Wii Remote gesteuert werden.

Im Vordergrund stand für die Entwickler vor allem die Projektarbeit und eine möglichst umfassende Konzeption, bei der alle Teile des Projekts sinnvoll zusammengeführt werden. Es musste daher ein Framework entwickelt werden, das Grafik und Physik miteinander vereint und dabei weitestgehend modular aufgebaut ist, um einzelne Teile, wie etwa die grafische Benutzeroberfläche oder die Physikengie, leicht austauschen zu können. Aber auch andere Aspekte wie Musik und eine ansprechende und verständliche Benutzeroberfläche sollten berücksichtigt werden.

2 interact

Das Programm wurde entwickelt unter Mac OS X 1.5 und kann als ausführbare Datei unter `http://www.uni-koblenz.de/~interact/` heruntergeladen werden. Bei Fragen sind die Entwickler unter *interact [at] uni-koblenz.de* erreichbar. Die Dokumentation kann unter `http://www.uni-koblenz.de/~interact/documentation/` eingesehen werden. Der Quellcode ist erhältlich auf Anfrage bei der AG-CG der Universität Koblenz.

3 Steuerung

Es kann sowohl mit der Maus als auch mit der Nintendo Wii Remote (siehe Abschnitt 8.4) gezeichnet werden. Da die Wiimote sehr empfindlich und umpräzise und somit für filigrane Zeichnungen weniger geeignet ist, war es notwendig, deren Sensitivität zu verringern. Die Wiimote Software bietet diesen Parameter zwar an, allerdings ist dieser fehlerhaft. Daher wurde dies über die GUI realisiert, indem im Wiimote Modus immer eine bestimmte Anzahl an Werten gesammelt und nur deren Mittelwert gespeichert wird.

4 Framework

Für die Software musste ein Framework entwickelt werden, das die Grafik, die Physik und weitere interne Funktionalität miteinander verbindet. Ziel war es, das System möglichst modular zu gestalten, um etwa die Benutzeroberfläche oder die Physikengine bei Bedarf leicht austauschen zu können und ohne aufwändige Änderungen am Code vornehmen zu müssen. Das Framework ist aus folgenden Hauptteilen zusammengesetzt (siehe Abbildung 2):

4.1 Scene

Die Szene bildet den Kern von *interact*. Hier laufen bei Bedarf alle Informationen zusammen oder werden von hier an andere Frameworkelemente geliefert. Sie verwaltet auch den Container, in dem alle Objekte der Szene gespeichert sind. Dadurch ist es möglich, im Editormodus die gesamte Szene inklusive aller Parameter als Level zu speichern und bei Bedarf wieder zu laden.

4.2 Vertex

Die Vertexklasse ist eine einfache Repräsentation eines zweidimensionalen Punktes. In ihr werden dessen x- und y-Koordinaten gespeichert. Für die urpsrüngliche Implementation des ConcaveChoppers (siehe Abschnitt 7.1) wurde hier noch eine natürliche Zahl gespeichert, die die Position des Nachfolgevertex innerhalb des Vectors angibt.

4.3 SceneObject

SceneObject ist die Basisklasse für Polygone, Linien und Kreise (siehe Abschnitt 4.3.1, 4.3.2 und 4.3.3). Sie beinhaltet deren gemeinsame Eigenschaften und unter anderem eine Liste, in welcher alle Vertices eines Objekts gespeichert werden. Jedes Objekt besitzt ein Zentrum, welches nicht den

geometrischen Mittelpunkt darstellt, sondern den Punkt markiert, an dem das Objekt zu zeichnen begonnen wurde. Dieses Zentrum ist immer der Punkt $(0, 0)$, da die Vertices eines Objekts in lokalen Koordinaten gespeichert werden.

4.3.1 Polygon

Ein Polygon ist ein SceneObject, welches eine Liste mit Vertices enthält, wobei immer drei aufeinanderfolgende als ein Dreieck angenommen werden. Polygone sind in *interact* immer simple konkave oder konvexe Objekte ohne Löcher, die intern als Gruppierung von Dreiecken repräsentiert sind und von der Grafik als ein Objekt dargestellt werden. Nicht-simple Polygone, also solche, die sich selbst überschneiden, werden als geschlossener Linienzug repräsentiert. Während des Zeichnens werden Polygone und Linien als Linienzug dargestellt, das sich erst beim Beenden des Zeichenvorgangs entscheidet, um welchen Objekttyp es sich handelt. Die Kriterien, die für ein Polygon erfüllt sein müssen, sind:

1. Start- und Endpunkt des Linienzugs müssen innerhalb eines bestimmten Radius liegen UND
2. der Linienzug darf sich nicht selbst überschneiden.

4.3.2 Line

Ein Linienzug wird durch eine Feste Reihenfolge ihrer Punkte repräsentiert, wobei immer zwei aufeinanderfolgende Punkte ein Liniensegment bilden. Da die Physikengine in ihrer derzeitigen Version keine Linien darstellen kann, da eine Linie keine “Dicke” besitzt, diese aber zum Berechnen der Masse benötigt wird, werden die Liniensegmente in der Physik als schmale Kästen repräsentiert (siehe Abbildung 1).

4.3.3 Circle

Kreise werden nur durch ihr Zentrum und einen Punkt auf ihrem Umkreis definiert. Daraus lässt sich der Radius durch die euklidische Distanz

$$d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} \tag{1}$$

berechnen und der Kreis mit folgender Methode zeichnen:

```
1  const double twoPi = 2.0f * M_PI;
2  // (a higher number yields a more perfect circle)
3  // currently, CIRCLESEGMENTS is set to 100
4  const double k_increment = twoPi / CIRCLESEGMENTS;
5  double theta = 0.0;
6  double radius = 0.0;
7  double newX = 0.0;
8  double newY = 0.0;
9
10 void Circle::drawSceneObject() {
11
12   radius = getRadius();
13
14   if( radius > 0 ) {
15
16     glBegin( GL_TRIANGLE_FAN );
17
18     for ( int i = 0; i < CIRCLESEGMENTS; ++i ) {
19
20       newX = m_CircleCenterX + radius * cos( theta );
21       newY = m_CircleCenterY + radius * sin( theta );
22       glVertex2f( newX, newY );
23       theta += k_increment;
24
25     }
26
27     glEnd();
28
29   }
30
31 }
```

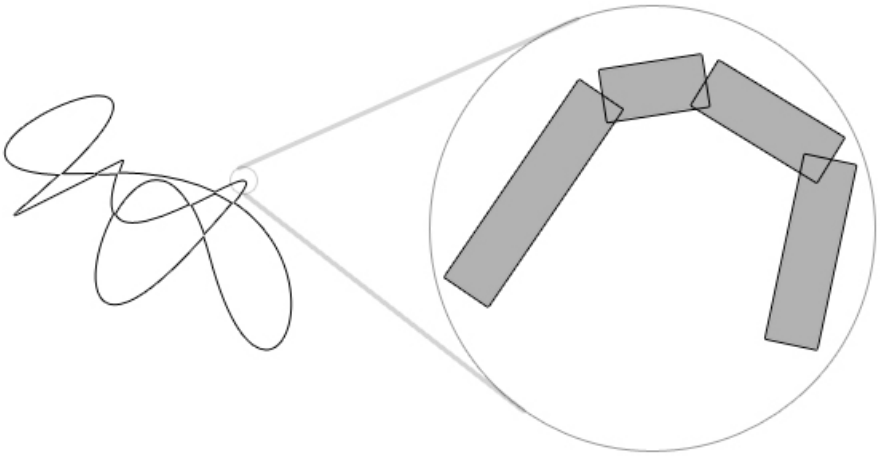

Abbildung 1: links: Linienzug; rechts: vergrößerter Ausschnitt. Der Linienzug setzt sich aus Liniensegmenten zusammen.

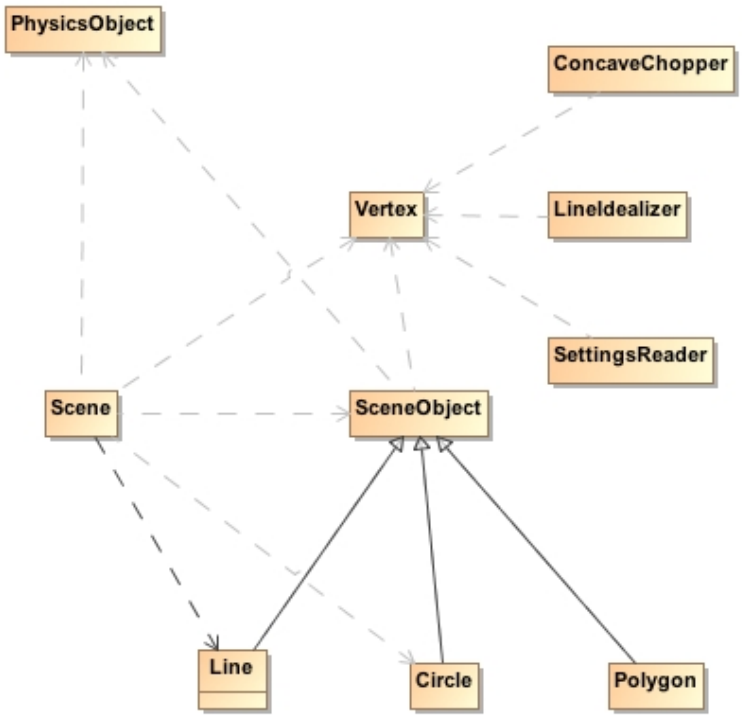

Abbildung 2: Die wichtigsten Elemente des Frameworks in einem vereinfachten Diagramm.

4.4 SettingsReader

Die Levels werden in *interact* in Dateien mit der Endung .lvl geschrieben. Die Daten liegen hierbei im Textformat vor. Die Aufgabe des SettingsReaders ist es, diese Dateien auszulesen und für eine spätere Abfrage zum Beispiel aus der Szene bereitzustellen.

5 OpenGL

Für die Realisierung der Grafik wurde OpenGL verwendet, das hier allerdings, mit Ausnahme des *Picking*, nicht näher erklärt werden soll.

5.1 Picking

Oft möchte man Objekte in der Szene löschen, nicht nur während des Spiels selbst, sondern auch bei der Levelerstellung im "Editor Modus". Das Löschen sollte durch Klicken auf das entsprechende Objekt realisiert werden. Ein ursprünglicher Ansatz hierfür war, das Objekt "durchzustreichen", d.h. es mit einer Zickzack-Linie zu übermalen. Diese Geste ist für viele Spieler recht intuitiv, wobei dies aber vor allem für die Steuerung per Touchscreen gilt. Für Zeigegeräte wie etwa Mouse oder Wiimote bot sich das Löschen per Klick auf das jeweilige Objekt an, zumal die meisten Nutzer daran gewöhnt sind, eher die Knöpfe dieser Geräte einzusetzen, statt Gesten zu benutzen. Zudem lässt sich etwa die Wiimote oft nicht präzise genug steuern, sodass sich das Löschen vor allem kleinerer Objekte teilweise schwierig gestaltet.

Durch einen Klick in die Szene erhält man die Koordinaten des Mousezeigers. Ein möglicher, aber unschöner, Weg ist es nun, für alle Objekte zu testen, ob der Punkt diese schneidet. Obwohl wir in der zweidimensionalen Szene nur vergleichsweise wenige Objekte haben, die sich normalerweise gar nicht oder nur kurzzeitig überschneiden, ist dieser Weg dennoch sehr ineffektiv. Einfacher und schneller geht es hier mit dem sogenannten "Picking", welches sich mithilfe von OpenGL über die Hardware realisieren lässt (siehe Listing 2). Klickt man im *Select-Modus* in die Szene, wird diese gerendert, jedoch nicht angezeigt. Dabei wird geprüft, welche Objekte innerhalb des vorher definierten Fensterbereichs liegen. Die Namen (Nummern) dieser Objekte werden dann in ein Array eingetragen. Definiert man den Fensterbereich entsprechend klein und sortiert man die Objekte gegebenenfalls noch nach ihren Tiefenwerten, kann man das vorderste, direkt unter dem Mauszeiger befindliche, Objekt ermitteln.

5.1.1 Objekte benennen

Um ein Objekt eindeutig identifizieren zu können, muss es einen Namen erhalten. Dazu bekommt jedes SceneObject in *interact* eine eigene ID, bestehend aus einer natürlichen Zahl. In der Zeichenmethode wird dann zuerst ein Namestack mittels *glInitNames()* in OpenGL erzeugt. Dieser Stack muss dann mit einem Wert initialisiert werden, zum Beispiel mit *glPushName(0)*. Nun muss nur noch direkt vor jedem Zeichenbefehl ein Name für das Objekt auf den Stack geladen werden: *glLoadName(irgendeineID)*. Die Namen können theoretisch auch komplexer sein. Ein Name wird, passend zur State Machine in OpenGL, solange an jedes gezeichnete Objekt vergeben, bis ein neuer Name auf den Stack gelegt wird. Auch sind hierarschische Namensgebungen möglich, sodass man beispielsweise Teile eines Objektes extra benennen kann.

5.1.2 Der Select Modus

Standardmäßig befindet OpenGL sich im "Rendermodus", das heißt alle Objekte werden direkt gezeichnet. Für das Picking muss nun in den "Selectmodus" umgeschaltet werden (siehe Listing 2, Zeile 22). Dabei "simuliert" die Hardware das Zeichnen aller Objekte nur, das heißt sie werden nicht eingefärbt beziehungsweise es werden keine Pixel in den Framebuffer geschrieben. Zusätzlich definiert man sich ein "viewing volume", das sich direkt um den Mousezeiger befindet. Die Ausmaße dieses volumes sind frei wählbar, sollten aber nicht zu groß ausfallen um Mehrfachselektion durch nebeneinander liegende Objekte zu vermeiden. Für *interact* wurde eine Größe von 5x5 Pixeln gewählt, was sich als guter Kompromiss herausgestellt hat, um sowohl Ungenauigkeiten bei der Führung des Zeigegeräts als auch der geringen Größe einiger Objekte gerecht zu werden (siehe Listing 2, Zeile 30). Die fünf Parameter des Befehls void *gluPickMatrix(GLdouble x, GLdouble y, GLdouble delX, GLdouble delY, GLint *viewport);* sind die x und y-Koordinate des Zentrums, die horizontalen und vertikalen Maße des Picking Bereiches und der Viewport. Letzterer ist bekannt aus Listing 2, Zeile 15.

Dazu wird noch ein "selectbuffer" angelegt, der aus einem integer array besteht. Die Größe des selectbuffers wurde hier mit 64 festgelegt. Dies ist völlig ausreichend, da pro gezeichnetem Objekt im Selectmodus drei + "Anzahl der Objekte bis dahin auf dem Stack" Werte abgespeichert werden. Da wir uns in *interact* nur in zweidimensionaler Ansicht befinden und eventuell kurzzeitig übereinander befindliche Objekte von der Physikengine auseinandergeschoben werden, liegen im Normalfall nur ein gezeichnetes Objekt und das Rechteck im Hintergrund, auf dem die Hintergrundtextur platziert ist, übereinander beziehungsweise im viewing volume. Beim Picking im Dreidimensionalen kann es vorkommen, dass sehr viele Objek-

te hintereinander liegen. In diesem Fall müsste die Reihenfolge der Objekte zuerst noch nach der Größe des z-Wertes sortiert oder gefiltert werden.

OpenGL benötigt, um den selectbuffer anlegen zu können, ein integer array, welches übergeben werden muss (siehe Listing 2, Zeile 6 u. 12). Nach Durchlauf des simulierten Zeichnens liefert die Funktion die Anzahl der getroffenen Objekte zurück (siehe Listing 2, Zeile 38).

Eine Beispielausgabe des Selektionbuffers nach einem Pick sieht in *interact* zum Beispiel so aus:

Listing 1: Beispielinhalt des Selectionbuffers in *interact*

```
1 Selectbuffer: 1 3221225471 3221225471 0 2 2147483647
    2147483647 0 121 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0 0
2 Objekt mit ID 121 wurde angeklickt!
```

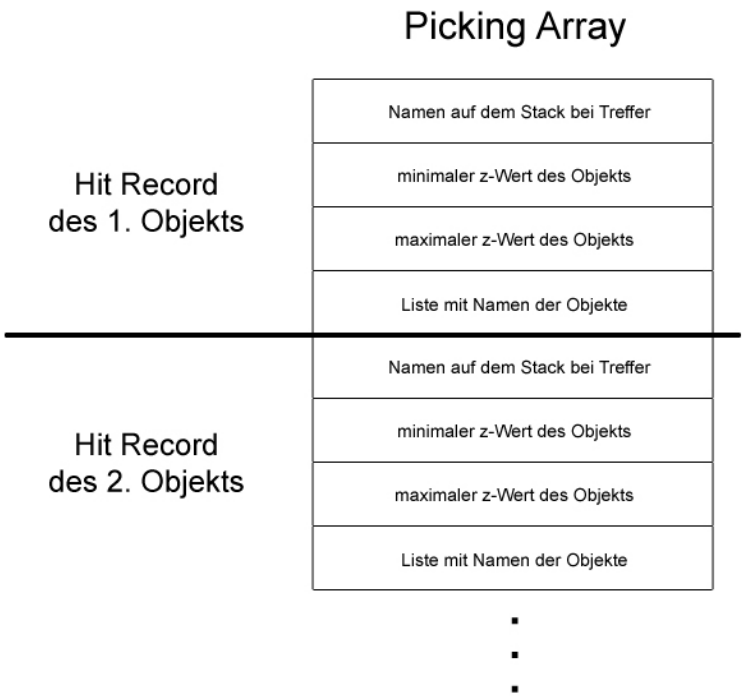

Abbildung 3: Aufbau des Picking Arrays in OpenGL

Wie in Listing 1 erkennbar, steht an der ersten Stelle des 64 Stellen großen Arrays die Anzahl der Objekte, die bis dahin auf dem Stack liegen. Danach folgen intern als unsigned integer repräsentierte z-Werte für Minimum und Maximum. Diese sind in diesem Fall gleich, da die Objekte ja immer feste z-Werte haben und ausschließlich zweidimensional sind. Dann folgt der Name des Objekts. Da es sich beim ersten Objekt um das Rechteck mit der Hintergrundgrafik handelt, welchem in der Zeichenmethode der Name *0* gegeben wurde, steht hier die 0. Damit ist der erste Hit Record zu ende. Direkt danach folgt dann das zweite Objekt. Vorn steht wieder die Anzahl der Objekte/Namen auf dem Stack, danach kommen die z-Werte und danach die Liste mit den Namen der Objekte, also 0 und 121. Da in *interact* immer nur 2 Objekte auf dem Stack liegen können, erhält man den Namen des gewünschten Objekts, indem man auf die neunte Stelle im selectionbuffer zugreift (siehe Listing 2, Zeile 43). Da jedes Objekt einer Szene eine eindeutige ID besitzt, muss dieses in einer Schleife nun nur noch gefunden und dessen grafische und physikalische Repräsentation aus der Szene gelöscht werden.

Die wesentlichen Schritte beim Picking Verfahren sind also:

1. OpenGL umschalten auf “selection mode”
2. die Szene zeichnen lassen, wobei nun nichts in den Framebuffer geschrieben wird
3. während des Zeichnens die Objekte benennen
4. immer wenn die Hardware ein Objekt im definierten Bereich um den Mousecursor zeichnen würde, liefert sie uns dessen Namen
5. OpenGL zurückschalten auf “render mode”

Listing 2: Picking, wie es in *interact* realisiert wurde

```
1  // Process the selection, which is triggered by a
2  // middle mouse click at (x, y).
3  void Picker::processSelection( int x, int y ) {
4
5    // Space for selection buffer
6    static GLuint selectBuffer[ BUFFER_LENGTH ];
7
8    // Hit counter and viewport storage
9    GLint hits, viewport[ 4 ];
10
11   // Set up selection buffer
12   glSelectBuffer( BUFFER_LENGTH, selectBuffer );
13
```

```
14    // Get the viewport
15    glGetIntegerv( GL_VIEWPORT, viewport );
16
17    // Switch to projection and save the matrix
18    glMatrixMode( GL_PROJECTION );
19    glPushMatrix();
20
21    // Change render mode
22    glRenderMode( GL_SELECT );
23
24    // Establish new clipping volume to be unit cube
25    // around mouse cursor point (x, y) and
26    // extending five pixels
27    // in the vertical and horizontal direction
28    glLoadIdentity();
29
30    gluPickMatrix ( ( GLdouble ) x, ( GLdouble ) (
          viewport[ 3 ] - y ), 5.0, 5.0, viewport );
31
32    Scene *scene = Scene::getInstance();
33
34    // Draw the scene
35    scene->draw();
36
37    // Collect the hits
38    hits = glRenderMode( GL_RENDER );
39
40    // If at least a single hit occurred, delete the
          object.
41    if( hits > 1 ) {
42
43          scene->deleteSceneObject( selectBuffer[ 8 ] );
44
45    }
46
47    // Restore the projection matrix
48    glMatrixMode( GL_PROJECTION );
49    glPopMatrix();
50
51    // Go back to modelview for normal rendering
52    glMatrixMode( GL_MODELVIEW );
53
54 }
```

6 Physikalische Grundlagen

In diesem Abschnitt sollen die physikalischen Grundlagen erklärt werden, die für *interact* gebraucht worden sind und die Teil einer jeden Physikengine sind. Im Speziellen geht es hierbei um die "Physik der starren Körper", die "Festkörperphysik" bzw. die "Rigid Body Physics" im Englischen. Zunächst werden allerdings die Grundlagen eines Teilfeldes der Starrkörperphysik, der Kinematik, beleuchtet. Sie beschreibt die Bewegung von Körpern, ohne dass physikalische Größen wie verschiedene Kräfte mit einbezogen werden. Die drei wichtigsten Komponenten, Position, Geschwindigkeit und Beschleunigung, werden hier genauer erleutert.

6.1 Kinematik

6.1.1 Position

Die Position eines Körpers im Raum kann auf verschiedene Arten dargestellt werden. Im 3-Dimensionalen kann es sich bei der Position z.B. um einen 3d-Vektor (x, y, z) handeln. Ein Punkt im 3d-Raum besitzt also 3 Freiheitsgrade bezüglich der Achsen. Zu diesen drei Freiheitsgraden kommen drei weitere, die Rotationen um die entsprechenden Achsen des Koordinatensystems, hinzu.

Im 2-Dimensionalen fällt die Tiefenkoordinate z dagegen weg und die Position eines Punktes lässt sich durch einen 2d-Vektor darstellen. Im Verlauf dieser Ausarbeitung wird nur die 2-Dimensionlität von Belang sein und auch die Rotation um x- oder y-Achse kann ignoriert werden. Jeder Punkt wird also nur noch drei Freiheitsgerade haben, zwei bezüglich der Translation und einen bezüglich der Rotation.

6.1.2 Geschwindigkeit

Wird der Positionsvektor eines Körpers nach der Zeit abgeleitet, erhält man die Geschwindigkeit des Körpers. Wird der Positionsvektor als *r* ausgedrückt, dann ergibt sich daraus folgende Formel:

$$\frac{dr}{dt} = v = \dot{r} \tag{2}$$

6.1.3 Beschleunigung

Ein weiterer wichtiger Begriff ist die Beschleunigung. Sie wird durch die zweite Ableitung des Ortes nach der Zeit beschrieben, was wiederum die erste Ableitung der Geschwindigkeit ist. Dies ergibt die Formel:

$$\frac{d^2r}{dt^2} = \ddot{r} = \frac{d\dot{r}}{dt} = \frac{dv}{dt} = \dot{v} = a \tag{3}$$

Integriert man wiederum die Beschleunigung dann erhält man also die Geschwindigkeit eines Körpers. Dies ist wichtig wenn die Position eines sich bewegenden Körpers aus der Beschleunigung heraus berechnet werden soll.

6.1.4 Orientierung

Zu den zwei bereits erwähnten translatorischen Freiheitsgraden in x- und y-Richtung und z-Richtung kommt nun die Rotation um diese Achsen hinzu. Das Maß der Rotation wird im weiteren Verlauf mit Ω notiert. Ω beschreibt die Drehung des Körpers im Verhältnis zum Weltkoordinatensystem in Radiant. Ω ist positiv und wird gegen den Uhrzeigersinn angegeben.

6.1.5 Winkelgeschwindigkeit

Die Winkelgeschwindigkeit ω ist die Ableitung des Winkels nach der Zeit.

$$\frac{d\Omega}{dt} = \omega \tag{4}$$

6.1.6 Winkelbeschleunigung

Die Winkelbeschleunigung ist die zweite Ableitung des Winkels oder auch die Ableitung der Winkelgeschwinigkeit ω.

$$\frac{d^2\Omega}{dt^2} = \frac{d\omega}{dt} = \dot{\omega} = \alpha \tag{5}$$

6.2 Dynamik

Die Dynamik beschreibt die Bewegung von Körpern unter dem Anwenden von Kräften. Auf diese physikalischen Effekte wird hier weiter eingegangen.

6.2.1 Kraft

Die Beschleunigung eines Objektes wird dadurch verändert, indem eine Kraft auf den Körper angewendet wird. Nach Newton ergibt sich folgende Formel für die Kraft:

$$F = \dot{p} = \frac{dp}{dt} = \frac{d(mv)}{dt} = m\dot{v} = ma \tag{6}$$

Die Masse multipliziert mit der Geschwindigkeit wird auch *Impuls* genannt. Aus obiger Formel lässt sich auch erkennen, dass die Kraft der Ableitung der Masse multipliziert mit der Geschwindigkeit entspricht. Da die Masse

der Objekte bei nicht-relativistischen Geschwindigkeiten als konstant angenommen wird, ergibt sich die bekannte Formel:

$$F = ma \tag{7}$$

Zu beachten ist hierbei, dass diese Formeln nur für einzelne Punktmassen gelten. Ein 2-dimensionaler Körper definiert seine Masse durch die Fläche die er einnimmt. Darum nimmt man die Masse eines starren Körpers als die Summe der Massen seiner einzelnen Punkte an. Daraus ergibt sich die Formel für den Gesamtimpuls:

$$p^T = \sum_i m^i v^i \tag{8}$$

Durch die Einführung des Massezentrums mit folgender Formel

$$r^{CM} = \frac{\sum_i m^i r^i}{M} \tag{9}$$

kann die obige Formel für den Gesamtimpuls weiter vereinfacht werden zu:

$$\begin{aligned} \frac{d(Mr^{CM})}{dt} &= \sum_i \frac{d(m^i r^i)}{dt} \\ &= \sum_i m^i v^i \\ &= p^T \end{aligned} \tag{10}$$

Daraus erhält man:

$$\begin{aligned} p^T &= \frac{d(Mr^{CM})}{dt} \\ &= Mv^{CM} \end{aligned} \tag{11}$$

Es lässt sich erkennen, dass es genügt, die Geschwindigkeit des Massezentrums multipliziert mit der Gesamtmasse des Körpers zu betrachten, um den Gesamtimpuls zu erhalten. So können auch die starren Körper als Punktmassen angenommen werden, was die Berechnungen für translatorische Bewegungen innerhalb einer Physikengine wesentlich vereinfachen wird. Durch die Annahme des Massezentrums lässt sich auch die Formel für die Kraft, die auf den gesamten Körper wirkt, vereinfachen zu:

$$\begin{aligned} F^T &= \dot{p}^T \\ &= M\dot{v}^{CM} \\ &= Ma^{CM} \end{aligned} \tag{12}$$

Nun kann durch das Teilen einer Kraft durch die Masse eines Körpers die Beschleunigung eines Körpers in seinem Massezentrum berechnet werden, über die durch Integration wiederum die Geschwindigkeit und die Position des Körpers ermittelt werden kann. Bei letzterer Formel ist zu beachten, dass die Position, an der die Kraft angewendet wurde, nicht mit in Betracht gezogen wurde. Sobald aber die Rotation von starren Körpern berechnet werden soll, muss auch die Position der Anwendung der Kraft beachtet werden.

6.2.2 Berechnung des linearen Impulses aus der Winkelgeschwindigkeit

Das Problem bei einem Körper ist, dass er aus unendlich vielen Punkten besteht. Die Formeln für den linearen Impuls beziehen sich aber lediglich auf den Punkt des Massezentrums. Für kollidierende Körper, die nicht rotieren, kann der lineare Impuls recht einfach berechnet werden, für rotierende Objekte kann aber jeder Punkt des Körpers eine andere Geschwindigkeit haben. Für unendlich viele Punkte des Körpers kann deren Geschwindigkeit aber unmöglich festgestellt werden. Allerdings lässt sich der lineare Impuls aus der Winkelgeschwindigkeit errechnen. Dabei wird davon ausgegangen, dass der Körper um einen Ursprung rotiert, dabei aber nicht transliert. Mit folgender Formel kann für einen beliebigen Punkt des Körpers dessen Geschwindigkeit berechnet werden.

$$v^B = \omega r_\perp^{OB} \tag{13}$$

r^{OB} bezeichnet den Vektor vom Ursprung des Körpers zum Punkt, dessen Geschwindigkeit bestimmt werden soll. Durch eine Erweiterung der obigen Formel mit der Geschwindigkeit des Ursprungs ergibt sich folgende Formel:

$$v^B = v^O + \omega r_\perp^{OB} \tag{14}$$

Diese besagt, dass man die Geschwindigkeit eines beliebigen Punkts des Körpers durch die Addition der Geschwindigkeit des Urspungspunktes und dem Produkt des gesuchten Punktes mit der Winkelgeschwindigkeit erhält.

6.2.3 Drehimpuls

Den Drehimpuls versteht man analog zum linearen Impuls, nur mit dem Unterschied, dass der Drehimpuls von einer Stelle im Raum aus gemessen wird.

$$L^{AB} = r_\perp^{AB} \cdot p^B \tag{15}$$

A symbolisiert den Punkt, von dem aus der Drehimpuls des Punktes B gemessen wird.

6.2.4 Drehmoment

Das Drehmoment wird durch die Ableitung des Drehimpulses definiert. Es beschreibt das physikalische Maß, das bei Abnahme oder Zunahme der Drehzahl eines Objekts wirkt.

$$\begin{aligned} \tau^{AB} &= \frac{dL^{AB}}{dt} \\ &= \frac{d(r_{\perp}^{AB} \cdot p^B)}{dt} \\ &= r_{\perp}^{AB} \cdot ma^B \\ &= r_{\perp}^{AB} \cdot F^B \end{aligned} \tag{16}$$

Die Integration des Drehmoments führt wieder zum Drehimpuls.

6.2.5 Gesamtdrehimpuls

Der Gesamtdrehimpuls verhält sich analog zum Gesamtimpuls. Er ist die Summe aller Drehimpulse vom Punkt A aus gesehen.

$$\begin{aligned} L^{AT} &= \hat{A}_i \, r_{\perp}^{Ai} \cdot p^i \\ &= \hat{A}_i \, r_{\perp}^{Ai} \cdot m^i v^i \end{aligned} \tag{17}$$

6.2.6 Trägheitstensor und Trägheitsmoment

Die Formel des Gesamtdrehimpulses lässt sich durch die Einführung des Trägheitsmoments weiter vereinfachen. Durch Einsetzen von [13] erhält man:

$$\begin{aligned} L^{AT} &= \hat{A}_i \, r_{\perp}^{Ai} \cdot m^i \omega r_{\perp}^{Ai} \\ &= \omega \hat{A}_i \, m^i r_{\perp}^{Ai} \cdot r_{\perp}^{Ai} \\ &= \omega \hat{A}_i \, m^i (r_{\perp}^{Ai})^2 \\ &= \omega I^A \end{aligned} \tag{18}$$

Bei I^A handelt es sich um einen Tensor, den Trägheitstensor. Dieser stellt die Trägheitsmomente bezüglich der Rotationsachsen dar. Ist die Rotationsachse identisch mit einer der Achsen des Koodinatensystems, ergibt sich das Trägheitsmoment aus der entsprechenden Position in I^A. Wird z.B. nur um die z-Achse des Koordinatensystems gedreht, so ist I^A gleich $I_{3,3}$. I^A

wird so zu einer Konstanten, bestehend aus der Masse des Objekts multipliziert mit seiner Fläche. Nun kann die Formel für das Drehmoment zu Folgendem umgeformt werden:

$$\begin{aligned} \tau^{AT} &= \frac{dL^{AT}}{dt} \\ &= \frac{d(I^A\omega)}{dt} \\ &= I^A\dot{\omega} \\ &= I^A\alpha \end{aligned} \tag{19}$$

Nun kann man über das Drehmoment, das auf einen Körper angewendet wird, die Winkelbeschleunigung errechnen, über die wiederum durch Integration Winkelgeschwindigkeit und Orientierung im Raum herausgefunden werden kann. Damit verfügt man über die benötigten Mittel, um die Bewegung und Drehung von Körpern innerhalb einer Physikengine zu berechnen.

7 Algorithmen

7.1 ConcaveChopper

Sowohl OpenGL als auch die Physikengine Box2D in ihrer derzeitigen Version bieten eine Möglichkeit, konvexe Polygone zu verarbeiten. Konkave Objekte können von OpenGL nur indirekt und von Box2D gar nicht verarbeitet werden. Um dennoch mit diesen häufig auftretenden Objekten arbeiten zu können, werden diese in *interact* trianguliert (siehe 8.3). Der verwendete Algorithmus für die Triangulation funktioniert jedoch nur für *simple* Polygone. *Nicht-simple*, also sich selbst überschneidende, Polygone sind aufgrund der wechselnden Orientierung nicht trivial und müssen gesondert behandelt werden (siehe Abbildung 4).

Zwar bietet die *OpenGL Utility Library (GLU)* einen Tesselator an, der in der Lage ist, konkave und nicht-simple Polygone zu triangulieren. Allerdings werden triangulierte Polygone aus einem solchen "Tesselation-Object" nicht mehr zurückgeschrieben, sodass sie beispielsweise in der Physik ebenfalls verwendet werden können. Es bestand daher zunächst die Notwendigkeit, selbst einen Algorithmus zu finden und zu implementieren, der nicht-simple Polygone in simple zerlegen kann. Zwar wurde ein solches Verfahren in *interact* entwickelt und eingesetzt. Tests mit Benutzern haben jedoch gezeigt, dass in vielen Fällen, wenn sich ein selbstüberschneidender Linienzug schließt, der Benutzer die einzelnen Segmente oft gerade *nicht* ausgefüllt haben möchte, da sich so zum Beispiel Objekte konstruieren lassen, die man wiederum mit anderen Objekten füllen kann, um etwa den Schwerpunkt des äußeren Objekts zu verändern. Die Möglichkei-

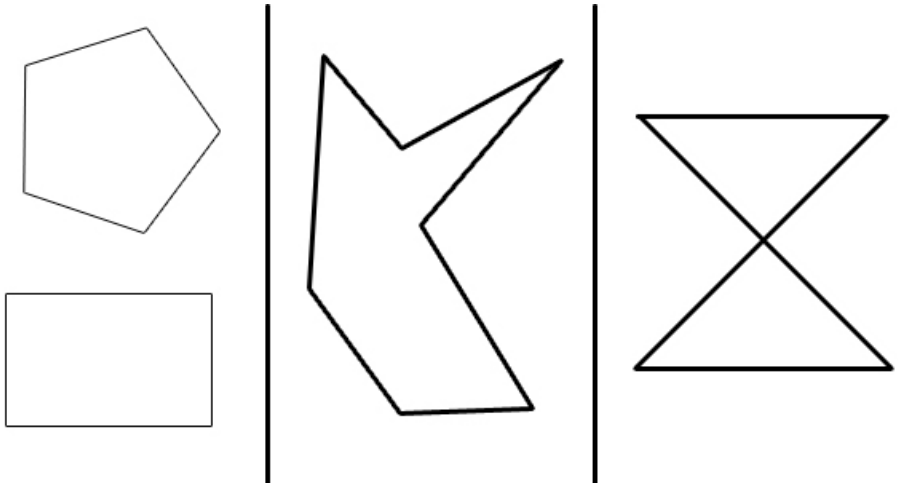

Abbildung 4: Polygontypen: simpel konvex(links); simpel konkav(Mitte); nicht-simpel / selbstüberschneidend(rechts)

ten zur Erstellung von Objekten werden dadurch nicht eingeschränkt, da nicht-simple Polygone im Zweidimensionalen auch durch simple konkave Polygone dargestellt werden können.

Der "ConcaveChopper" hatte in *interact* also ursprünglich die Aufgabe, simple konkave und konvexe Polygone zu triangulieren und nicht-simple Polygone zuerst in simple zu zerlegen und diese dann zu triangulieren. Dem ConcaveChopper wird ein Vector übergeben, der alle Punkte eines Linienzugs enthält. Immer zwei aufeinanderfolgende Punkte (auch der letzte und der erste) bilden ein Liniensegment. Nun wird jedes Liniensegment genau einmal mit allen anderen geschnitten. Findet man einen Schnittpunkt, kann man die Untersuchung abbrechen und hat ein nicht-simples Polygon gefunden. Aufgrund der oben erwähnten Vereinfachung wird der Linienzug nun einfach als geschlossene, sich überschneidende Linie dargestellt. Hat man keinen Schnittpunkt gefunden, bildet der Linienzug ein simples konkaves oder -konvexes Polygon, welches trianguliert und so abgespeichert wird.

Da die Entwicklung und Implementation eines Algorithmus zur Aufteilung nicht-simpler in simple Polygone mit Abstand am meisten Zeit benötigte und das Problem nicht trivial ist, soll er im Folgenden beschrieben werden, auch wenn er in *interact* aus den oben genannten Gründen keine Verwendung mehr findet.

7.1.1 Schnittpunkte

Schnittpunkt zweier Linien in 2D [Bou89]

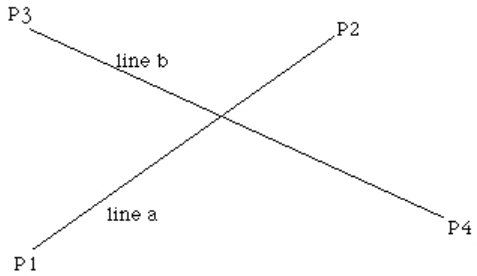

Abbildung 5: Zwei sich überschneidende Linien in 2D [Bou89]

Seine zwei Linien (siehe Abbildung 5) gegeben mit den Gleichungen:

$$\begin{aligned} P_a &= P1 + u_a(P2 - P1) \\ P_b &= P3 + u_b(P4 - P3) \end{aligned} \tag{20}$$

Setzt man $P_a = P_b$, so erhält man folgende Gleichungen mit den zwei unbekannten u_a und u_b:

$$\begin{aligned} x1 + u_a(x2 - x1) &= x3 + u_b(x4 - x3) \\ y1 + u_a(y2 - y1) &= y3 + u_b(y4 - y3) \end{aligned} \tag{21}$$

Aufgelöst nach u_a und u_b ergibt dies:

$$\begin{aligned} u_a &= \frac{(x4 - x3)(y1 - y3) - (y4 - y3)(x1 - x3)}{(y4 - y3)(x2 - x1) - (x4 - x3)(y2 - y1)} \\ u_b &= \frac{(x2 - x1)(y1 - y3) - (y2 - y1)(x1 - x3)}{(y4 - y3)(x2 - x1) - (x4 - x3)(y2 - y1)} \end{aligned} \tag{22}$$

Setzt man nun u_a oder u_b in die entsprechende Gleichung (siehe Gleichung 21) ein, erhält man x und y des Schnittpunktes zum Beispiel durch:

$$x = x1 + u_a(x2 - x1)$$
$$y = y1 + u_a(y2 - y1) \tag{23}$$

In den Gleichungen für u_a und u_b (siehe Gleichung 22) ist der Nenner jeweils der gleiche. Wenn dieser 0 ist, dann sind die beiden Linien parallel. Bei der Implementation beziehungsweise generell bei der Berechnung ist zu beachten, dass nicht durch 0 geteilt werden darf! Ist zusätzlich der Zähler 0, so sind die beiden Linien deckungsgleich. Diese Gleichungen gelten für Geraden. Wird die Schnittpunktberechnung lediglich für *Liniensegmente* benötigt, muss nur getestet werden, ob u_a und u_b zwischen 0 und 1 liegen. Liegt eines der beiden zwischen 0 und 1, so liegt der Schnittpunkt auf dem entsprechenden Liniensegment. Liegen beide dazwischen, so befindet sich der Schnittpunkt auch auf beiden Liniensegmenten.

Sweep Line Für die Umsetzung des Algorithmus zur Aufteilung von nicht-simplen Polygonen wurden alle Schnittpunkte eines solchen Polygons benötigt. Hierfür könnte ein sogenannter *Sweep Line* Algorithmus verwendet werden. Dieser hat im Normalfall bei der Suche nach dem ersten vorkommenden Schnittpunkt einen Aufwand von $O(n\ log\ n)$ und bei der Suche nach allen Schnittpunkten einen Aufwand von $O((n+k)\ log\ n)$, wobei k die Anzahl der Schnittpunkte und n die Anzahl der Linien ist. Ein bekanntes Verfahren für die Detektion aller Schnittpunkte ist der *Bentley-Ottmann Algorithmus* [BO79]. Der Test auf lediglich einen Schnittpunkt könnte effizient zum Beispiel mit dem *Shamos-Hoey Algorithmus* implementiert werden [SH76].

Da die Anzahl der Liniensegmente eines Polygons in *interact* vergleichsweise gering und der Algorithmus aufwändig in der Implementierung ist, wurde stattdessen eine einfache Methode benutzt, bei der jedes Liniensegment einmal mit jedem anderen unter Verwendung des in Abschnitt 7.1.1 beschriebenen Verfahrens geschnitten wird. Dies hat einen Aufwand von $O(n^2)$, welcher jedoch kaum ins Gewicht fällt.

7.1.2 Aufteilen nicht-simpler Polygone

Will man testen, ob ein Polygon simpel oder nicht-simpel ist, reicht ein Schnitttest, der abbricht, sobald ein Schnittpunkt gefunden wurde. Da man aber für den Dekompositionsalgorithmus alle Schnittpunkte finden muss, falls es ein nicht-simples Polygon ist, und die Zahl der Schnittpunkte in einem Linienzug in *interact* vergleichsweise gering ist, empfiehlt es sich, nur

den Algorithmus zum Finden aller Schnittpunkte anzuwenden und diesen gegebenenfalls vorzeitig abbrechen zu lassen. Die Dekomposition eines nicht-simplen Polygons könnte etwa mit folgendem Algorithmus durchgeführt werden:

1. Berechne alle Schnittpunkte des Polygons
2. Erstelle als Ersatz für jeden Schnittpunkt zwei neue Punkte V_1 und V_2, wobei V_1 in Kante e_1 und V_2 in Kante e_2 eingefügt wird. Jede Kante e_i wird dadurch aufgeteilt in die zwei Kanten $e_{i,in}$ und $e_{i,out}$, die durch den Punkt V_i miteinander verbunden sind.
3. Nun müssen die Überkreuzungen der Kanten entfernt werden. Hänge die Kanten an den Schnittpunkten um durch
   (a) Verbinden von $e_{1,in}$ und $e_{2,out}$ an V_1 und
   (b) Verbinden von $e_{2,in}$ und $e_{1,out}$ an V_2 (siehe Abbildung 6)
4. Nachdem dies für alle Schnittpunkte durchgeführt wurde, bilden alle verbundenen Kanten simple (konkave oder konvexe) Polygone, die zusammen das ursprüngliche nicht-simple Polygon darstellen.

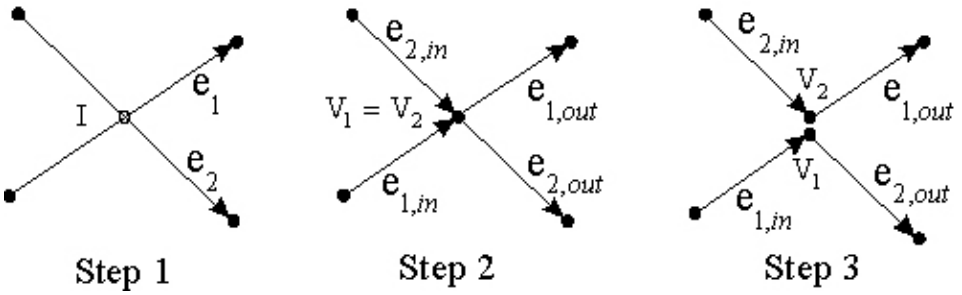

Abbildung 6: Zerlegen nichtsimpler Polygone Umhängen der Kanten in drei Schritten[1]

Die neu entstandenen simplen Polygone können noch ineinander liegen oder sich überlappen. In Box2D sollte zwar darauf geachtet werden, dass die Anzahl der unabhängigen übereinanderliegenden Polygone bei der initialen Erzeugung möglichst gering ist, da es anderenfalls zum Absturz kommen kann. Polygone jedoch, die ihrerseits wieder zu einem größeren Polygon in der Physikengine gruppiert werden, stellen kein Problem dar, da für diese keine Schnitttests beziehungsweise Kollisionsberechnungen durchgeführt werden.

[1] http://geometryalgorithms.com/Archive/algorithm_0108/algorithm_0108.htm#Decompose%20into%20Simple

Alle Vertices eines SceneObjects werden in Reihenfolge in einer Liste gespeichert. In der ursprünglichen Implementation wurde in jedem Vertex zusätzlich die *Position* seines Nachfolgevertex in dieser *Liste* gespeichert (siehe Abschnitt 4.2). Die Kanten wurden nun umgehängt, indem die entsprechenden Nachfolgepositionen vertauscht wurden. Hierbei ist zu beachten, dass sich auch zwischen dem letzten und dem ersten Punkt eine Kante befindet. Anschließend wurden die entstandenen simplen Polygone eingesammelt und dabei jeder bereits abgearbeitete Punkt in der Liste markiert um Mehrfachaufnahmen zu vermeiden.

Als nächstes musste jedes der neuen Polygone noch trianguliert werden. Die Physik erhielt lediglich die Dreiecke aller Polygone eines ursprünglich nicht-simplen Polygons. Diese Dreiecke wurden nicht als Polygone übergeben, sondern als reine Punktefolge in einer Liste. Sowohl die Physik als auch die Grafik gehen davon aus, dass immer drei aufeinanderfolgende Punkte ausschließlich zu einem Dreieck gehören. Die Information über die simplen Polygone wurde nach der Triangulation nicht mehr benötigt.

7.2 LineIdealizer

Mit der Mouse gezeichnete Linien, die sich über größere Distanzen auf dem Bildschirm erstrecken, aber auch solche, die mit einem ungenaueren Zeigegerät wie der Wiimote erstellt wurden, können oft sehr stark von der Vorstellung des Zeichnenden abweichen, da sie meist sehr "krakelig" aussehen. Des Weiteren wird in *interact* die Linienzeichnung von der Mousebewegung bestimmt. Diese wird von der grafischen Benutzeroberfläche in bestimmten Abständen erfasst und dabei die aktuelle Position in einen c++-Vector gespeichert. Da das Zeitintervall für die Erfassung der Mouseposition fest ist, hängt die Anzahl der Punkte, die zu einer Linie gehören, davon ab, wie lange und mit welcher Geschwindigkeit man zeichnet ohne abzusetzen. Je schneller man den Mousezeiger bewegt und je zeitlich kürzer man dabei zeichnet, desto weniger Punkte werden in den Vector aufgenommen. Bei langsamer und langer Zeichnung einer Linie können so leicht mehrere hundert Punkte hinzugefügt werden. Zwei aufeinander folgende Punkte bilden dabei eine Gerade. Aus Gründen, die in dieser Arbeit unter den Abschnitten 4.3.1 und 4.3.2 erläutert werden, wird für jede Linie getestet, ob sie sich selbst überschneidet, und jedes Polygon wird trianguliert. Nun erscheint es sinnvoll, die Anzahl der Punkte eines Objekts soweit wie möglich zu reduzieren. Dies hat mehrere positive Effekte sowohl für den Benutzer als auch intern für die Performance. Erstens werden die Umrisse von Linien und Polygonen wesentlich geglättet, was normalerweise der Vorstellung des Zeichners entspricht. Zweitens wird die Physikengine dadurch entlastet, da jedes Segment einer Linie in der jetzigen Version von Box2D intern als schmaler Kasten repräsentiert werden muss und die Anzahl der verarbeitbaren Physikobjekte begrenzt ist. Drittens müssen Po-

lygone so in wesentlich weniger Dreiecke zerlegt werden, was Rechenzeit beim Triangulieren und Physikobjekte spart. Zwar ist die sehr geringe Anzahl der Objekte beim Zeichnen für OpenGL kein Problem, wenn aber für viele komplexe Objekte einmalig Schnitttests und Triangulationen sowie ständig in der Physik eine große Anzahl an Physikobjekten berechnet werden muss, so verlangsamt dies das Programm merklich.

Ein Lösungsansatz wäre nun, nur jeden x-ten Punkt aus dem vector zu behalten oder immer eine gewisse Anzahl Punkte zu mitteln und nur diesen Wert abzuspeichern. Das Ergebnisobjekt entspricht aber in den wenigsten Fällen dem, welches der Benutzer gezeichnet hat. Es wurde daher ein Algorithmus entwickelt, der gezackte Linien oder leichte Bögen in sehr kleinen Bereichen innerhalb der Linie glättet und dabei markante Ecken und Konturen beziehungsweise Extrempunkte erhält.

Hierzu bildet man zunächst mit dem ersten und einem weiteren Punkt in einem bestimmten Abstand z (gemeint ist hier der Abstand der Indizes im Vector beziehungsweise im Linienzug!) eine Gerade. Der erste Punkt wird automatisch der Ergebnismenge hinzugefügt. Nun werden alle Punkte dazwischen daraufhin überprüft, ob sie von dieser Gerade weiter entfernt sind als eine bestimmte räumliche Distanz w. Falls einer dieser Punkte weiter entfernt ist als w, wird dieser zur Punktemenge hinzugefügt und als neuer Startpunkt angenommen. Sind keine weiteren Punkte mehr auf der ersten Gerade, wird deren Endpunkt als Startpunkt angenommen, dieser zur Ergebnismenge hinzugefügt und es wird mit dem nächsten Punkt in Abstand z die nächste Gerade gebildet.

Durch den Abstand z ist garantiert, dass mindestens jeder der Punkte mit diesem Abstand zur Ergebnismenge gehört. Der räumliche Abstand w garantiert zudem, dass jeder Extrempunkt zur Ergebnismenge gehört, nur gering abweichende Punkte jedoch ignoriert werden. Dies glättet den Linienzug merklich, ohne jedoch dessen Form zu verändern.

8 Externe Bibliotheken

Dieses Kapitel beschreibt, welche externen Bibliotheken verwendet wurden um komplexe, aber bereits existierende, Funktionalitäten einzubinden. Wichtig war hierbei, dass die entsprechenden Komponenten frei erhältlich sind und ohne größeren Zeitaufwand verwendet werden können.

8.1 Sound

Da OpenAL unter Mac OS X 10.5 nicht mehr unterstützt wird, wurde die Soundbibliothek FMOD Ex[2] v. 4.14 verwendet. Diese ist für das Abspielen der Hintergrundmusik und sonstiger Geräusche zuständig.

[2] http://www.fmod.org

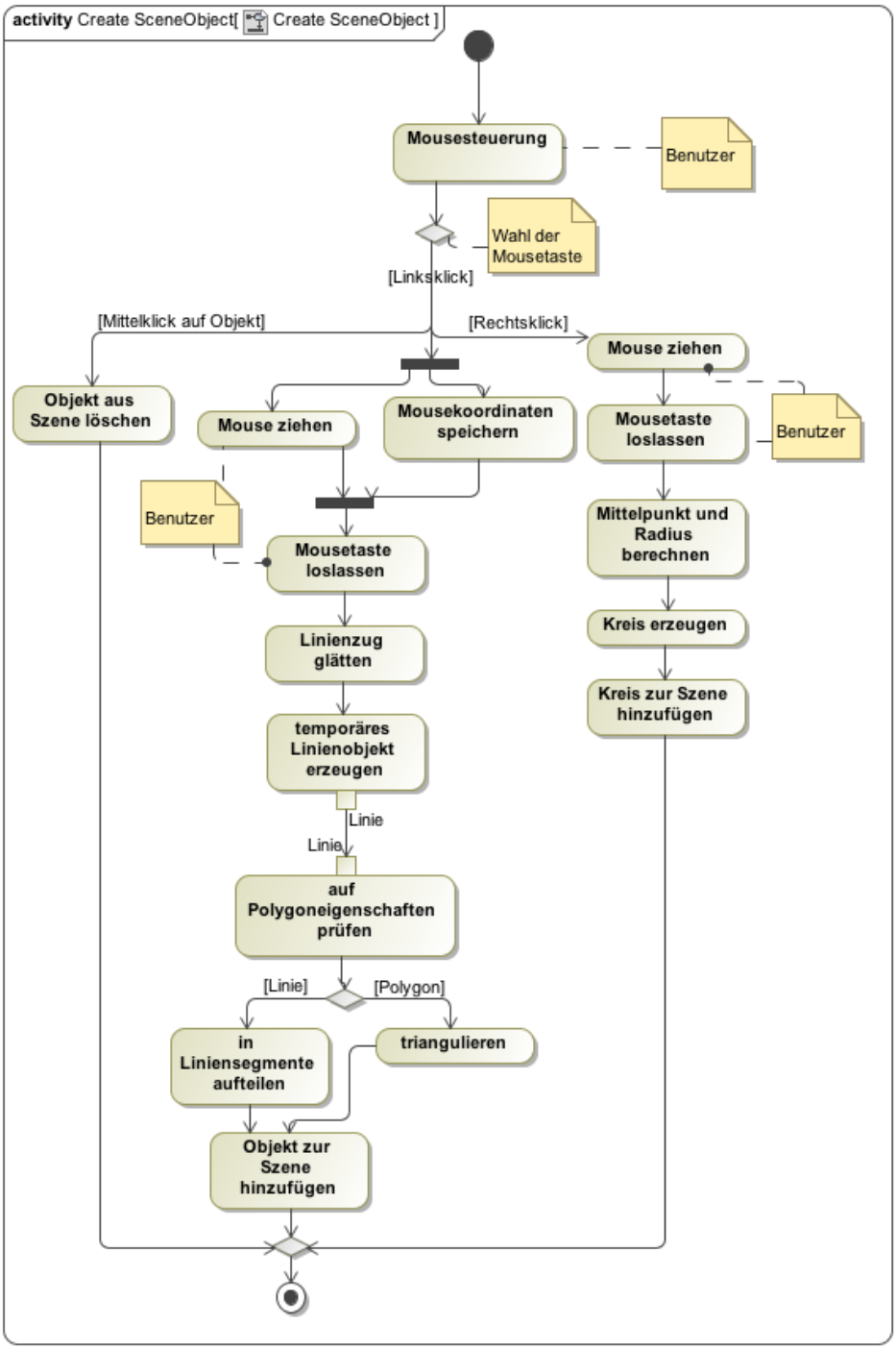

Abbildung 7: Das Aktivitätsdiagramm zeigt den Entstehungsprozess eines Szenenobjekts.

8.2 Texture Loader

Hintergrundbilder verleihen dem Spiel mehr Atmosphäre und bieten zudem die Möglichkeit mit ihnen das Level mitzugestalten. Da es in *interact* möglich ist, unsichtbare statische Objekte beim Editieren eines Levels zu erzeugen, können beispielsweise auf dem Hintergrundbild dargestellten Objekten unsichtbare Physikobjekte zugewiesen werden, was die aufwändige Texturierung erspart und dem Spieler mehr Gestaltungsspielraum gibt, als wenn er nur einfache, einfarbige Objekte verwenden könnte. Als Texturformat wurde png gewählt. Jedes Level kann seine eigene Hintergrundgrafik laden, die dann auf ein bildschirmfüllendes Rechteck im Hintergrund projiziert wird. Zum Laden der png-Dateien wird eine bereits implementierte Version eines "png-Loaders" verwendet [Van07]. Dieser liefert eine eine von OpenGL benötigte Textur ID, die dann nur noch mit wenigen Befehlen an OpenGL übergeben werden muss.

8.3 Triangulation

Mit *GL_POLYGON* steht OpenGL ein Mittel zur Verfügung konvexe Polygone darzustellen. Allerdings entstehen diese in *interact* durch das Freihandzeichnen nur in wenigen Fällen. Oft entstehen konkave Polygone, die OpenGL zwar zeichnet, die aber selten der ursprünglich gezeichneten Kontur entsprechen. Teilweise werden konkave Bereiche korrekt dargestellt, andere wiederum werden einfach überzeichnet und es werden so zum Beispiel Lücken geschlossen, sodass das Ergebnis etwas entfremdet aussieht und vor allem auch die interne Physikrepräsentation nicht mehr damit übereinstimmt. Dadurch können andere gezeichnete Objekte durch dieses hindurchfallen oder es kann zu Überlappungen kommen. Auch die Physik kann nicht mit konkaven Polygonen arbeiten und es hat sich zudem gezeigt, dass Polygone mit sehr vielen Eckpunkten zu Problemen in der Physikengine führen können.

Es war daher nötig die Polygone zu triangulieren. Dies muss für jedes Objekt nur einmal ausgeführt werden, da die entstehenden Dreiecke direkt abgespeichert werden. Vor der Triangulation eines Polygons wird dessen zunächst als Linie dargestellte Kontur noch mittels des LineIdealizers optimiert, wodurch sich die Anzahl der Eckpunkte, und damit auch der entstehenden Dreiecke, erheblich reduziert (siehe 7.2). Für die Physikengine ist es recht einfach, alle angegebenen Dreiecke fest miteinander zu verbinden. Somit können nun auch konkave Polygone sowohl in OpenGL als auch in der Physik korrekt dargestellt werden. Sowohl OpenGL als auch die Physikengine gehen davon aus, dass die Vertices der Polygone immer gegen den Uhrzeigersinn (ccw) angegeben werden. Da es in *interact* möglich ist anders herum zu zeichnen, muss in diesem Fall die Reihenfolge der Punkte entsprechend vertauscht werden. Dies geschieht ebenfalls während der

Triangulation.

Leider entsteht durch die Triangulation ein anderes Problem: Da Objekte, die sich überlappen, in der Physik auseinandergeschoben werden, bis sich lediglich ihre Konturen berühren, kann es vorkommen, dass sich manche Objekte miteinander "verhaken" oder eingeklemmt werden. Zeichnet man beispielsweise einen kleinen Kreis innerhalb eines Polygons, so kann es vorkommen, dass dieser nicht komplett aus dem Polygon herausgeschoben wird, sondern zwischen zwei Dreiecken im Polygon festsitzt. Die Physikengine schiebt ein Objekt entlang des kürzesten Weges aus einem anderen heraus. Liegt nun ein Kreis zwischen zwei Dreiecken, wird er von diesen jeweils in die andere Richtung gedrückt, wodurch er festklemmt. Es kann auch vorkommen, dass Objekte dann anfangen zu "zittern", was sehr störend im Spiel sein kann.

Ein anderes Problem im Zusammenhang mit Polygonen sind die sogenannten "nicht-simplen" Polygone, also solche, die nicht nur konkav sind, sondern sich zudem selbst überschneiden (siehe dazu 7.1).

Für die Triangulation wurde eine bereits implementierte Version eines nach Angaben des Entwicklers effizienten Triangulierers verwendet[Rat00].

8.4 Darwiin Remote

Der Nintendo Wii Remote Controller (auch *Wiimote* genannt) ist mit einer Infrarotkamera und einem Beschleunigungssensor ausgestattet. Bewegungssignale, die diese Sensoren empfangen, werden kabellos via *Bluetooth* gesendet. Um diese Signale auch an einem bluetoothfähigen Rechner empfangen und verarbeiten zu können, wurde die für *Mac OS X* verfügbare Software *Darwiin Remote* verwendet, die eigenständig gestartet wird um die Wiimote anzubinden [Hir08].

9 GUI

9.1 GLUT vs. Qt vs. wxWidgets

Zu Beginn der Studienarbeit wurde *interact* noch mit *GLUT* entwickelt. Als jedoch klar wurde, dass unter anderem zum Editieren von Levels Eingabe- und Einstellungsmöglichkeiten benötigt würden, musste auf eine leistungsfähigere grafische Benutzeroberfläche umgestellt werden. In der engeren Auswahl standen *QT* und *wxWidgets*, die beide nahezu die gleichen Möglichkeiten bieten. Die Wahl fiel auf *QT*, da dessen Dokumentation den Ruf hat recht ausführlich zu sein.

Die Umstellung auf *QT* war mit zusätzlicher Arbeit verbunden, da es erstens komplexer aufgebaut ist als *GLUT* und zweitens die bisherige Implementation noch zu sehr auf *GLUT* beruhte und die GUI somit nicht

leicht austauschbar war. Eine Implementierung, die von Anfang an auf *QT* basiert, und modular aufgebaut ist, hätte viel Zeit erspart.

9.2 LevelEditor

Die einzelnen Levels sind teilweise in sehr kurzer Zeit zu lösen. Daher war es notwendig einen Leveleditor zur Verfügung zu stellen. Mit diesem können sowohl neue Level erstellt als auch bereits bestehende Level bearbeitet werden. Zunächst kann (bei einem neuen Level) ein Hintergrundbild ausgewählt werden. Die Physik ist während des Editierens ausgeschaltet. Es können sowohl statische sichtbare und unsichtbare, wie auch dynamische sichtbare Objekte gezeichnet werden. Statische Objekte sind dabei unbewegliche Hindernisse, die für alle anderen Objekte undurchlässig sind (Abb. 8).

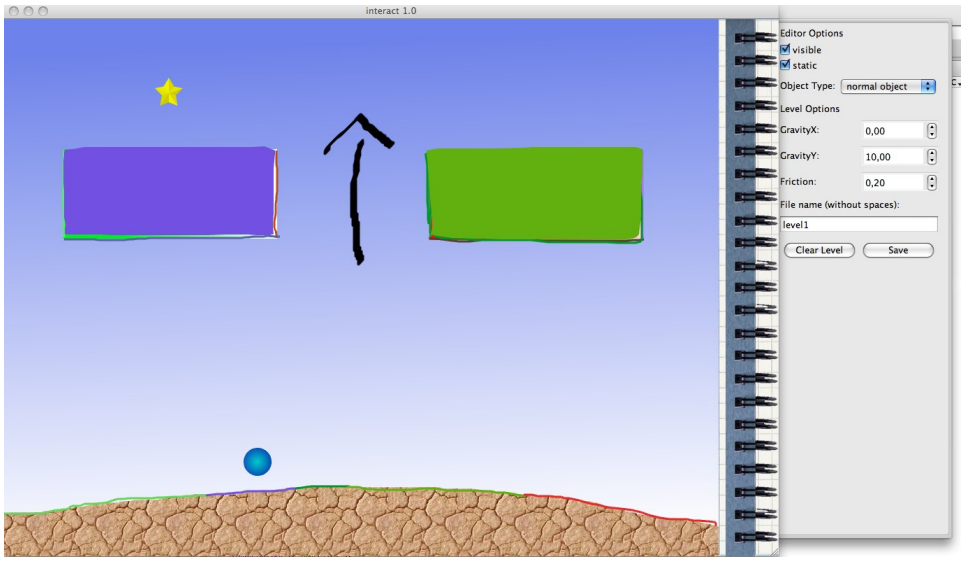

Abbildung 8: In *interact* können Levels erstellt oder editiert werden.

10 Grundsätzliche Komponenten einer Physikengine

Durch die vorhergehenden Formeln kann nun berechnet werden, wie sich einzelne Objekte unter welchen Kräften wie verhalten. Eine Physikengine benötigt aber zusätzlich die Information darüber, wo sich Objekte befinden und ob diese Objekte mit einander interagieren. Die Qualität dieser Informationen ist ausschlaggebend für das realistische Verhalten der Objekte. Wird eine Kollision von Objekten nicht korrekt oder nur ungenau bestimmt, kann dies zu unerwartetem Verhalten führen. Der Kontrollmechanismus für die Kollisionsdetektion muss theoretisch über die gesamte

Zeitspanne der Simulation kontinuierlich laufen, um konsistente Daten zu garantieren. Je mehr Objekte sich in der Simulation befinden, desto mehr Kollisionen können entstehen. Es wäre aber sehr ineffizient, wenn innerhalb der Physikengine jedes Objekt mit jedem verglichen werden müsste, um herauszufinden, ob diese Objekte kollidieren. Der Aufwand dieser Vergleiche wäre $O(n^2)$. Es sollten also nur Objekte betrachtet werden, die überhaupt für eine Kollision in Betracht kommen, alle anderen sollen nicht betrachtet werden. Dafür wird ein Kontrollmechanismus benötigt, der im Zusammenhang mit einer Physikengine "Broadphase" genannt wird.

10.1 Broadphase

Die Broadphase dient dazu, die möglicherweise kollidierenden Objekte zu filtern und nur diese aufgreifen, die vor einer Kollision stehen. Dazu liegen die Objekte in der Broadphase in einer vereinfachten Art und Weise vor, um nicht wertvolle Rechenzeit zu verschwenden und somit die Echzeitfähigkeit der Physikengine zu gefährden. Diese vereinfachte Form wird "Axis Aligned Bounding Box" genannt. Sie ist ein Rechteck, das anhand der Achsen des Koordinatensystem um das Objekt gelegt wird. In der Broadphase werden dann nur noch die AABBs der Objekte gegeneinander geprüft.

Anhand dieser AABBs kann nun ein Algorithmus angewendet werden, über den herausgefunden wird, welche AABBs sich überschneiden. Als ein bekannter Algorithmus wäre hier "Sweep and Prune" anzuführen. Dieser Algorithmus sortiert die Start- und Endpunkte der Bounding Boxes der Objekte entlang der Achsen des Koordinantensystems. Überschneiden sich diese Start- und Endpunkte entlang beider Achsen, werden die entsprechenden Objekte einer näheren Prüfung in der "Narrow Phase" unterzogen. Ein einfaches Beispiel zeigt die Wichtigkeit der Broadphase. Man stelle sich ein Pyramide aus viereckigen Kästen vor. Zwar kollidieren immer die Objekte mit den jeweiligen Nachbarn, allerdings kollidiert z.B. der oberste Kasten nicht mit Objekten der untersten Reihe. Die Broadphase verhindert also, dass alle Kästen der Pyramide gegeneinander geprüft werden.

10.2 Narrow Phase

In der "Narrow Phase" werden die Objekte behandelt, deren AABBs sich in der Broadphase überschnitten haben. Für jedes dieser Paare müss nun überprüft werden, ob sich die Formen ebenfalls überschneiden. Der möglicherweise entstehende Kontakpunkt muss nun bestimmt werden. Folgende Abbildung beschreibt die Situation grafisch.

Ein bekannter Algorithmus dazu ist der GJK-Algorithmus [GJK97]. Dieser Algorithmus kann die Distanz und den Beührungspunkt zweier sich

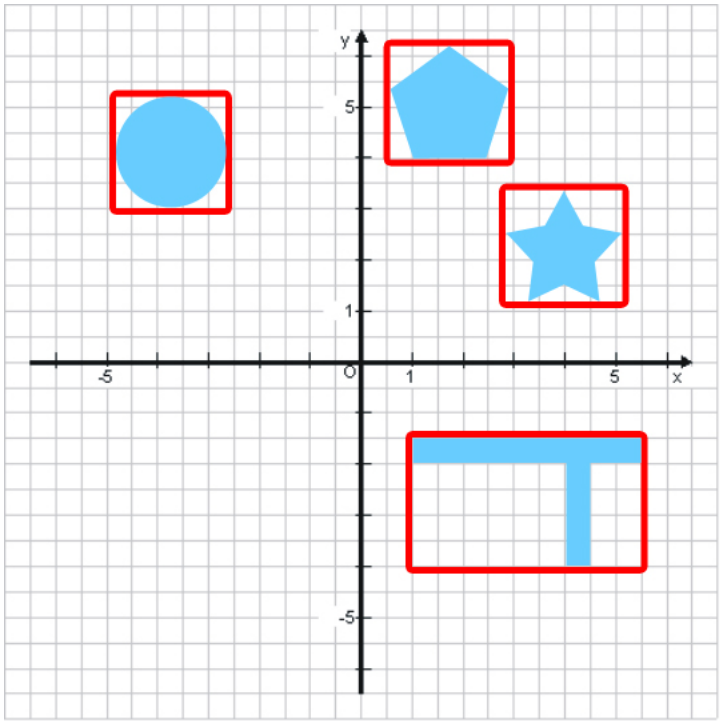

Abbildung 9: Verschiedene Geometrische Objekte (blau) und ihre zugehörigen AABBs (rot)

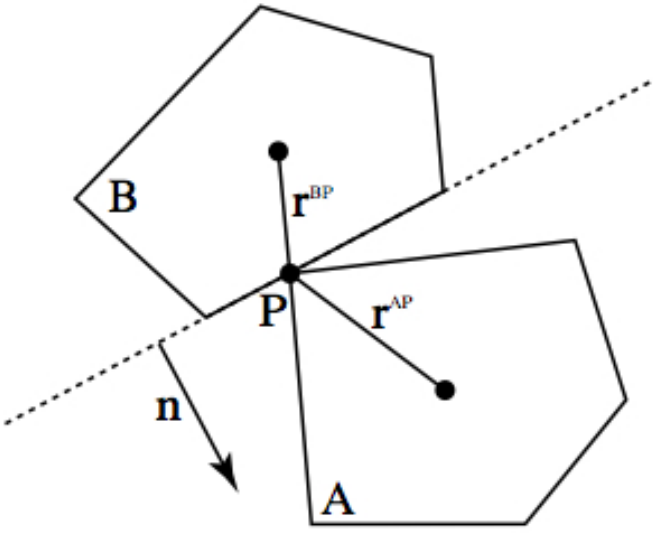

Abbildung 10: Zwei Objekte kollidieren

aufeinander zubewegenden, konvexen, geometrischen Objekte bestimmen. Typischerweise ist diese Phase die rechenaufwendigste Phase der Engine, da die geometrischen Eigenschaften der Objekte auch die Komplexität des Kollisionstests bestimmt. Deshalb ist es wichtig, dass die Broadphase alle nicht-kollidierenden Objekte bereits herausgefiltert hat, bevor die auf jeden Fall kollidierenden Objekte an die Narrow Phase übergeben werden.

10.3 Island Creation

Nachdem die Kontaktpunkte der kollidierenden Objekte herausgefunden worden sind, werden Gruppen, auch Inseln genannt, aus diesen Kollisionsobjekten gebildet.

10.4 Island Processing

Die entstandenen Inseln werden nun ausgewertet. Anhand der Kräfte und Drehmomente werden die entstehenden Beschleunigungen errechnet. Durch Integration werden die Daten für die neue Position und Orientierung bezogen. Diese können nun auf die geometrischen Objekte angewendet werden und die Berechnung erfolgt von neuem.

An dieser Stelle ist es wichtig anzumerken, dass die Korrektheit der Kollisionen abhängig von der zeitlichen Auflösung ist, auf die die Physikengine eingestellt worden ist. Werden nicht genug Überprüfungen pro Sekunde gemacht, kann es passieren dass sich Objekte bereits überlappen, dies aber zu spät festgestellt worden ist. Die Physikengine muss also räumlich und zeitlich so aufgelöst sein, dass im besten Falle kein Durchdringen von Objekten möglich ist. Allerdings spielt die Geschwindigkeit, mit der sich die Objekte bewegen, hierbei auch eine ausschlaggebende Rolle.

11 Auswahl der Physikengine

Um den Zeitrahmen der Studienarbeit nicht zu überschreiten, wurde die Entscheidung getroffen, eine bereits existierende Physikengine an das Spiel anzubinden. Eine eigene Implementierung wäre in der Zeit nicht möglich gewesen. Die richtige Wahl der Engine war somit für *interact* wichtig. Es stand vorab fest, dass *interact* ein 2-dimensionales Spiel sein soll, also würde keine Physikengine benötigt werden, die im 3-Dimensionalen rechnet. Zwar wäre es durchaus möglich gewesen eine 3d-Physikengine zu verwenden, doch erschien dies über die Voraussetzungen hinauszugehen. Auch musste die Engine nicht zwangsweise in der Lage sein, die Physik von nicht-starren Objekten zu berechnen, da in *interact* nur starre Objekte miteinerander interagieren würden. Weitere Anforderung war, dass die Engine frei verfügbar sein muss. Bekannte kommerzielle Engines wie z.B. Ha-

vok[3]schieden somit aus. Die Engine sollte im besten Falle über eine C/C++-API verfügen und gut dokumentiert sein. Die Auswahl wurde somit auf die zwei augenscheinlich passendsten Engines reduziert: Chipmunk[4]und Box2D. Box2D wurde bereits in iPhysics und CrayonPhysics verwendet und schien somit ihre Tauglichkeit bereits unter Beweis gestellt zu haben. Sie ist gut dokumentiert, wird kontinuierlich verbessert und besitzt eine große Community, die weitere Funktionen bereitstellt und bei eventuellen Problemen helfen kann. Chipmunk basiert auf Box2D, hat aber an manchen Stellen andere Wege eingeschlagen. So wurde z.B. für die Broadphase ein Spatial Hashing Algorithmus verwendet. Letztenendes fiel aber die Wahl auf Box2D, da sich aus dem Quelltext unter anderem ein Testbed kompilieren lässt, mit dessen Hilfe sehr leicht Ideen und Implementationen ausprobiert werden können und Beispiele und Lösungshilfen zu den gängigsten Problemen gefunden werden können.

12 Wichtige Schritte zum Einsatz von Box2D

Hier werden die wichtigsten Schritte beim Einsatz einer Physikengine am Beispiel von Box2D vorgestellt.

12.1 Erzeugen einer Welt

Listing 3: Erzeugen einer Physikwelt

```
1 b2AABB worldAABB;
2 worldAABB.lowerBound.Set(-100.0f, -100.0f);
3 worldAABB.upperBound.Set(100.0f, 100.0f);
4
5 b2Vec2 gravity(0.0f, -10.0f);
6
7 bool doSleep = true;
8
9 b2World *world = new b2World( worldAABB, gravity,
    doSleep );
```

Durch diese Zeile wird ein Zeiger auf eine Physikwelt erzeugt. In ihr wird sich die Simulation der Objekte abspielen. An den Konstruktor von **b2World** wird eine Variable vom Typ **b2AABB**, eine Variable vom Typ **b2Vec2** und eine Variable vom Typ **boolean** übergeben. *worldAABB* beschreibt die Grenzen der Welt mithilfe einer AABB, die Welt ist also eine

[4]Havok Physics Engine, http://www.havok.com, zugegriffen am 06.07.2008

[4]Chipmunk Physics Engine, http://wiki.slembcke.net/main/published/Chipmunk, zugegriffen am 06.07.2008

rechteckige Box. Die Grenzen der Welt werden fest bestimmt. Falls Objekte diese Grenzen überschreiten, kann ihr physikalisches Verhalten nicht mehr korrekt bestimmt werden. Die Objekte werden sozusagen "eingefroren". Über den Parameter *gravity* wird die Richtung des Gravitationsvektors verschrieben. Für erdähnliche Schwerkraft würde hier z.B. $(0.0, -9.81)$ eingesetzt werden. So werden alle Objekte in Richtung der negativen y-Achse fallen. Theoretisch können hier aber beliebige Werte eingesetzt werden. *doSleep* legt fest, ob sich nicht bewegende Objekte weiterhin berechnet werden sollen oder nicht. Dies wird vor allem aus Gründen der Performancegemacht. Nachdem die Welt erzeugt worden ist können nun Objekte in die Simulation hinzugefügt werden.

12.2 Erzeugen von Objekten

In diesem Abschnitt wird die Erzeugung von Objekten in der Welt erläutert. Zunächst wird ein Rechteck erzeugt.

Listing 4: Erzeugen eines Objekts

```
1  b2PolygonDef properties;
2  properties.SetAsBox( 50.0f, 10.0f );
3  properties.friction = 0.3f;
4  properties.density = 1.0f;
5
6  b2BodyDef bodyDefinition;
7  bodyDefinition.position.Set( 0.0f, -10.0f );
8
9  b2Body *box = world->CreateBody( &bodyDefinition );
10 box->CreateShape( &properties );
11 box->SetMassFromShapes();
```

Zunächst werden die Eigenschaften des Rechtecks festgelegt. Dazu wird eine Variable vom Typ **b2PolygonDef** benötigt. Für dieses Objekt können nun verschiedene Eigenschaften gesetzt werden. In diesem Fall wird festgelegt, dass es sich bei dem Objekt um eine Box handeln soll, deren Ausmaße 100 Einheiten in die x-Richtung und 20 Einheiten in die y-Richtung sein sollen. Die Methode **SetAsBox()** rechnet ihre Parameter auf den Ursprung des Objekts in lokalen Koordinaten, wobei der Ursprung des Objekts sein Mittelpunkt ist. Die Box erstreckt sich also 50 Einheiten in die positive und 50 Einheiten in die negative x-Richtung, 10 Einheiten in die positive und 10 Einheiten in die negative y-Richtung. Diese Einheiten sind in Box2D immer als MKS-Einheiten zu verstehen, also Meter, Kilogramm und Sekunden. Der Reibungskoeffizient der Box wird auf 0.3 festgelegt. Die Dichte wird auf 1.0 gesetzt. Nun werden die Eigenschaften des Körpers

festgelegt. Der Körper soll an der Stelle $(0.0f, 10.0f)$ der Welt erzeugt werden. Jetzt kann der eigentliche Körper der Box erstellt werden. Durch die Methode **CreateBody()** wird ein Zeiger auf den erzeugten Körper zurückgeliefert und wiederum dem Zeiger **box* zugewiesen. Auf **box* wird nun die Methode **CreateShape()** aufgerufen, die dem Körper die Form und die Eigenschaften verleiht, die vorher durch *properties* festgelegt worden sind. Zuletzt wird für *box* die Methode **SetMassFromShapes()** aufgerufen, mit der automatisiert die Masse des Körpers anhand der an ihn angehängten Formen ausgerechnet und festgelegt werden kann.

12.2.1 Erzeugen einer Linie

Linien lassen sich nicht ohne weiteres erzeugen. Die Linie muss mit Rechtecken angenähert werden. Folgender Algorithmus erzeugt aus einer Liste von Punkten eine Linie aus aneinandergehängten Rechtecken.

Listing 5: Erzeugen einer Linie aus Rechtecken

```
1  std::vector<b2PolygonDef> PhysicsObject::
       buildLineFromVector( std::vector< Vertex* >*
       vertices, float density, float friction ){
2
3    std::vector<b2PolygonDef> lineDefs;
4
5    float lineThickness = 0.15f;
6
7    b2PolygonDef lineDef;
8    lineDef.vertexCount = 4;
9    lineDef.density = density;
10   lineDef.friction = friction;
11
12   for( int i = 0; i < ( int )vertices->size() - 1; i++
         ){
13
14     float32 dx = ( vertices->at( i + 1 )->getX() /
         SCALINGFACTOR ) - ( vertices->at( i )->getX() /
          SCALINGFACTOR );
15     float32 dy = ( vertices->at( i + 1 )->getY() /
         SCALINGFACTOR ) - ( vertices->at( i )->getY() /
          SCALINGFACTOR );
16
17     float32 m = dy / dx;
18     float32 arctan = atan( m );
19     float32 lineSegmentLength = sqrt( dx * dx + dy *
         dy);
```

```
20      float32 lineSegmentLengthHalf = lineSegmentLength
            / 2;
21
22      b2Vec2 lastPtoNextP( dx / 2, dy / 2 );
23      b2Vec2 verticesAti( vertices->at( i )->getX() /
            SCALINGFACTOR, vertices->at( i )->getY() /
            SCALINGFACTOR );
24      lastPtoNextP = lastPtoNextP + verticesAti;
25
26      lineDef.SetAsBox( lineSegmentLengthHalf,
            lineThickness, lastPtoNextP, arctan );
27
28      lineDefs.push_back( lineDef );
29
30    }
31
32    return lineDefs;
33
34  }
```

Diese Methode erzeugt aus einem Vektor mit Zeigern auf Punkte eine Linie mit einer Dichte *density* und einem Reibungskoeffizienten *friction*. Rückgabewert ist wiederum ein Vektor vom Typ **b2PolygonDef**, also einer Liste mit Polygondefinitionen für jedes einzelne Rechteck der angenäherten Linie. In der Methode wird also zunächst ein Vektor vom Typ **b2PolygonDef** erzeugt. Nun wird die Höhe der Rechtecke festgelegt, die letztenendes die Dicke der zu erzeugenden Linie bestimmen wird. Dann muss ein einzelnes Objekt vom Typ **b2PolygonDef** angelegt werden, in das die Eigenschaften jedes einzelnen Liniensegmentes gespeichert werden. Nun können in einer Schleife über die Länge des Vektors mit den Eingabepunkten die einzelnen Liniensegmente erzeugt werden. Dazu wird die Steigung und der Winkel jedes aufeinanderfolgenden Punktepaares benötigt. Die Variable *SCALINGFACTOR* wird an dieser Stelle benötigt, da die Werte der Eingabepunkte in Pixelkoordinaten vorliegen und dadurch die Werte für die Koordinaten in der Engine zu groß werden. Nachdem die Steigung m ausgerechnet worden ist, kann der Winkel über den Arcus Tangens bestimmt werden. Ebenfalls benötigt wird die halbe Länge des Liniensegments. Da zum Anlegen des Rechtecks der Mittelpunkt des Liniensegments benötigt wird, muss dieser ebenfalls berechnet werden und auf den jeweiligen Eingabepunkt addiert werden. Nun kann diese Liniensegementdefinition mit der bereits vorgestellten Methode **SetAsBox()** mit den entsprechenden Parametern angelegt werden. Diese Liniensegmentdefinition wird nun in den Vektor gespeichert, den die Methode als Rückgabe-

wert liefert. Anhand dieses Vektors kann nun ein Körper erzeugt werden, zu dem die Definitionen hinzugefügt werden.

12.3 Anfragen an die Engine

Listing 6: Abfragen der Position eines Objekts

```
1 b2Vec2 position = body->GetPosition();
```

Über die Methode **GetPosition()** kann die aktuelle Position des Körpers abgefragt werden. Sie liefert ein Objekt vom Typ **b2Vec2** zurück. Mit *position.x* und *position.y* kann auf die jeweiligen x- und y-Komponenten des Vektors zugegriffen werden.

Listing 7: Abfrage der Rotation eines Objekts

```
1 float32 angle = body->GetAngle();
```

Über die Methode **GetRotation()** kann die aktuelle Rotation des Körpers abgefragt werden. Sie liefert ein Objekt vom Typ **float32** zurück. Die Angabe des Winkels erfolgt in Radiant.

12.4 Rückmeldungen aus der Engine

Möchte der Benutzer Rückmeldungen aus der Engine erhalten, z.B. ob Kontaktpunkte hinzugefügt worden sind oder ob Objekte gelöscht werden sollen, dann müssen sog. Listener für die physikalische Welt registriert werden. Diese Listener sind nach dem Observer Pattern implementiert, das eins der Kernkomponenten einer Physikengine darstellt. Bevor die Verwendung dieser Listener dargelegt wird, wird zunächst die Funktionsweise des Observer Patterns näher dargestellt.

12.4.1 Das Observer Pattern

Bei dem Observer Pattern handelt es sich um ein softwaretechnisches Konzept. Es findet meistens dann Verwendung, wenn es in Software darum geht, anhand von Nachrichten zwischen einzelnen Komponenten der Software bestimmte Aktionen ausgelöst werden sollen. Diese Nachrichten können unter anderem dazu da sein den Zustand eines Objektes zu aktualisieren. Ein Anwendungsbeispiel wäre eine grafische Benutzungsoberfläche mit Buttons, die durch den Benutzer angeklickt werden können. Wird ein Button betätigt wird intern eine Nachricht an die Komponente der Software gesendet, die so programmiert ist, dass sie diese Art von Nachrichten erhalten kann. Diese Komponente aktualisiert dann z.B. den Eintrag eines

Textfeldes, zeichnet grafische Komponenten neu oder ähnliches. Das Observer Pattern bietet sich deswegen auch für eine Physikengine an. Hier muss, wie in interact, ein Zugriff auf die entstehenden Kontaktpunkte der kollidierenden Körper innerhalb der Engine möglich sein, um auf bestimmte Situationen zu reagieren. In *interact* muss auf die Kollision von Ball und Stern mit einem Levelwechsel reagiert werden. Deswegen muss in *interact* ein Kontaktpunkt-Listener aktiviert werden, um Nachrichten zu erhalten, wenn Kontaktpunkte hinzugefügt worden sind. Sind die zu den Kontaktpunkten gehörenden Körper Ball und Stern, so wird die Funktion für den Levelwechsel aufgerufen. Auch das Löschen von Objekten kann durch einen solchen Listener realisiert werden. Wird ein bestimmter Teil eines Objektes entfernt kann eine Nachricht versendet werden um automatisiert auch den Rest des Körpers zu löschen. Dies ist in Box2D durch den Destruction-Listener möglich.

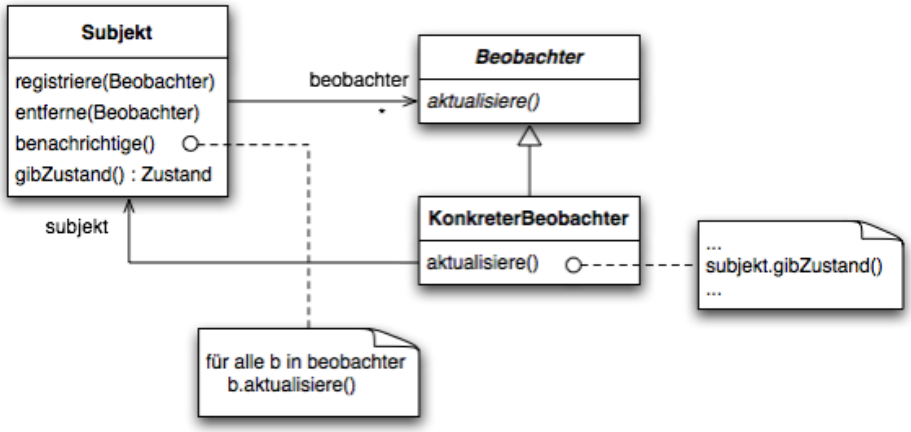

Abbildung 11: UML-Diagram des Observer Pattern [5]

Aus der Klasse **b2ContactListener** kann der Benutzer eine Klasse **ContactListener** ableiten. Für diese können dann je nach Bedarf die Methoden **Add()**, **Persist()** und **Remove()** implementiert werden. In diesen ist die Funktionalität enthalten, die bei der Reaktion auf das Hinzufügen, Bestehenbleiben und Entfernen von Kontaktpunkten ausgeführt werden soll. Nun muss dieser Listener noch an die Simulation angebunden werden.

Listing 8: Der ContactListener

```
1 world->SetContactListener( m_contactListener );
```

[5] Beobachter Pattern, http://de.wikipedia.org/wiki/Beobachter_(Entwurfsmuster), zugegriffen am 06.07.2008

Die Methode **SetContactListener(m_contactListener)** registriert hier den in *m_contactListener* enthaltenen Zeiger auf einen *ContactListener* in der Welt.

12.5 Simulationsablauf

Listing 9: Ein simples Beispielprogramm

```
1  int main(int argc, char** argv){
2
3          b2AABB worldAABB;
4          worldAABB.lowerBound.Set( -100.0f, -100.0f );
5          worldAABB.upperBound.Set( 100.0f, 100.0f );
6          b2Vec2 gravity( 0.0f, -10.0f );
7          bool doSleep = true;
8
9          b2World world( worldAABB, gravity, doSleep );
10
11         b2BodyDef groundBodyDef;
12         groundBodyDef.position.Set( 0.0f, -10.0f );
13         b2Body* groundBody = world.CreateBody( &
               groundBodyDef );
14
15         b2PolygonDef groundShapeDef;
16         groundShapeDef.SetAsBox( 50.0f, 10.0f );
17         groundBody->CreateShape( &groundShapeDef );
18
19         b2BodyDef bodyDef;
20         bodyDef.position.Set( 0.0f, 4.0f );
21         b2Body* body = world.CreateBody( &bodyDef );
22
23         b2PolygonDef shapeDef;
24         shapeDef.SetAsBox( 1.0f, 1.0f );
25
26         shapeDef.density = 1.0f;
27         shapeDef.friction = 0.3f;
28
29         body->CreateShape( &shapeDef );
30         body->SetMassFromShapes();
31
32         float32 timeStep = 1.0f / 60.0f;
33         int32 iterations = 10;
34
35         for (int32 i = 0; i < 60; ++i){
```

```
36
37                  world.Step( timeStep, iterations );
38
39                  b2Vec2 position = body->GetPosition();
40                  float32 angle = body->GetAngle();
41
42                  printf( "%4.2f %4.2f %4.2f\n",
                        position.x, position.y, angle );
43          }
44
45          return 0;
46  }
```

In diesem Beispiel wird eine Welt erzeugt und ein rechteckiger Körper als "Boden" hinzugefügt. Über diesem Körper wird ein kleinerer rechteckiger Körper erzeugt. Über *timeStep* wird die Anzahl der Schritte der Engine pro Sekunde festgelegt. *iterations* legt die Iterationen fest, die pro diskretem Zeitschritt bei der Berechnung gemacht werden sollen. Nun wird in der Schleife die Methode *Step(timeStep, iterations)* aufgerufen, die die eigentliche Berechnung durchführt. Nach jedem Schritt in der Engine werden Position und Winkel ausgegeben. Diese Daten können nun genutzt werden, um, wie in interact, Objekte neu zu zeichnen.

13 Limitationen von Box2D

13.1 Objektgröße

Box2D ist auf Meter-, Kilogramm und Sekundeneinheiten eingstellt. Im Bereich von 0.1 bis 10 Einheiten garantiert die Engine laut dem Verfasser eine gute Performance. Bei Körpern mit einer Größe von mehr als 10 Einheiten, also mehr als 10m, und Körpern mit einer Größe kleiner als 0.1, also kleiner als 10cm, werden die Berechnung teilweise ungenau und die numerischen Verfahren zur Lösung der Gleichungen instabil.

13.2 Objektanzahl

Theoretisch wird die Anzahl der physikalischen Objekte nur von der Rechen- und Speicherkapazität des Rechners beschränkt. Box2D hat allerdings eine Obergrenze für die Anzahl der Kontaktpunkte und Objekte in der Broadphase festgelegt. Diese Grenzen lassen sich aber durch den Benutzer anpassen. Die maximale Anzahl an Objekten in der Broadphase beträgt standardmäßig 512.

13.3 Rechengenauigkeit

Die Rechengenauigkeit der Engine wird durch die intern verwendeten Variablen begrenzt. Diese Variablen sind entweder 32bit-Fließkommazahlen oder 32bit-Integer. Die Genauigkeit dieser Werte ist für *interact* völlig ausreichend, wissenschaftliche Anwendungen bedürfen wahrscheinlich einer höheren Genauigkeit. Box2D besitzt als Projekt, dass hauptsächlich von einer Person betreut wird, keinen Anspruch, dieser wissenschaftlichen Genauigkeit Rechnung zu tragen. Für Spiele ist die Präzision hoch genug.

13.4 Räumliche und zeitliche Auflösung

In einem vorigen Abschnitt wurde bereits die Methode **Step()** erwähnt. Durch sie wird die zeitliche Auflösung beeinflusst. Standardmäßig werden als Parameter $60Hz$ und 10 Iterationen verwendet. Wird eine höhere zeitliche Aufläsung benötigt kann die Anzahl der *timeStep* und *iterations* erhöht werden, was allerdings zu einer schlechteren Performance führen kann. Die minimale Größe eines Objekts in der Engine beträgt $0.5mm$. Objekte die kleiner als dieser Wert sind können nicht mehr korrekt berechnet werden. Es muss notfalls überprüft werden, ob diese Grenze überschritten wird und entsprechende Objekte vom Einsetzen in die Simulation ausgeschlossen oder sichergestellt werden, dass keine Objekte erzeugt werden, die diese Grenze verletzen.

14 Fazit

Noch vor 10 Jahren war es kaum vorstellbar, die Bewegung von Objekten, Explosionen und andere physikalische Effekte realitätsgenau in einem Spiel darzustellen. Wenn sich beispielsweise zwei ineinander verzahnte Zahnräder gedreht haben konnte dies zwar durch die Grafik dargestellt werden und sah möglicherweise auch realistisch aus. Allerdings war eine weitere Interaktion mit diesen Gegenständen undenkbar. Dass der virtuelle Held eine Metallstange zwischen diese Zahnräder steckt, die dadurch verkeilen und somit ein Spielfortschritt möglich wird, war ein Ding der Unmöglichkeit. Die Rechengeschwindigkeit der damals handelsüblichen Computer war nicht ausreichend um physikalische Eigenschaften dieser Zahnräder wie Kraft, Drehmoment oder Reibung in das Spielgeschehen einzubinden. Heute ist der Einsatz von Physikengines in Spielen ein Industriestandard. Kaum eine Spiel, egal aus welchem Genre es auch kommen mag, kommt noch ohne die realistische Darstellung von physikalischem Verhalten aus. Mittlerweile gibt es Spiele die auf gerade diesem realistischen Verhalten aufbauen. Das Potential ist mit Sicherheit noch lange nicht erschöft. Durch geschickten Einsatz von solchen Komponenten kann bereits Kindern physikalisches Verhalten von Gegenständen auf spielerische

Art und Weise nahegelegt werden. Zusätzlich stellen solche Spiele oft einen Bereich dar, der sehr unterhaltsam ist.

Oft entäuschen Spiele, weil sie sich nicht verhalten, wie der Benutzer es antizipiert hat. Für *interact* war es gerade dieser Gesichtspunkt Spiele wie iPhysics oder CrayonPhysics als geistiges Vorbild zu nehmen und dieses Konzept nachzuahmen und noch unterhaltsamer und interaktiver zu gestalten. Dabei war es wichtig, dass *interact* ein vollständiges Spiel sein sollte, dass vielleicht noch durch weitere Features erweitert werden könnte, aber an sich ein abgeschlossenes Projekt darstellt. Ausser der grafischen und physikalischen Komponente des Spiels waren also auch GUI und Sound von Belang. Ohne diese Komponenten wäre *interact* nicht zu dem Programm geworden, das es jetzt ist und hätte nicht das Spielgefühl geboten, das es jetzt bietet. Die Studienarbeit hat uns mit Sicherheit in unseren Programmierkenntnissen gefestigt, aber auch neue Erkenntnisse gebracht. Wir haben gelernt, welche Probleme beim Verbinden von verschiedenen Softwarekomponeten entstehen können und wie es möglich ist, diese Probleme zu lösen oder zu umgehen. Die Einarbeitung in die Funktionsweise einer Physikengine hat gezeigt wie komplex ein solches System werden kann, aber auch, dass man mit vergleichsweise wenigen, nicht zu komplizierten, mathematischen Gleichungen physikalische Funktionsweisen simulieren kann.

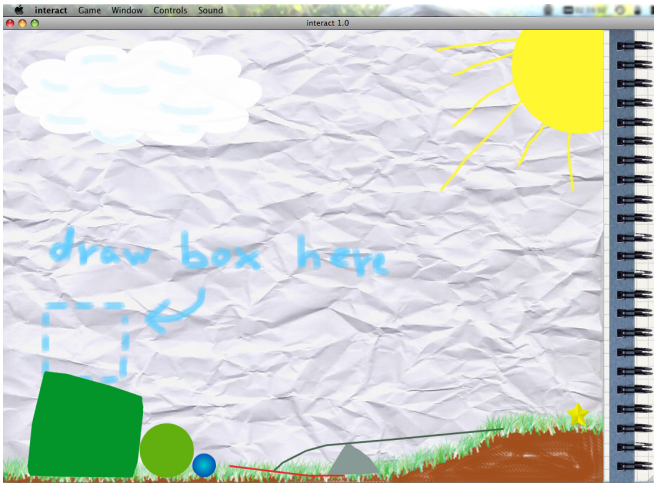

Abbildung 12: Szenenbild von *interact*

Abbildung 13: Szenenbild von *interact*. Leveldesign: Stefan Rilling

Literatur

[BO79] Jon Bentley and Thomas Ottmann. *Algorithms for Reporting and Counting Geometric Intersections*, pages 643–647. IEEE Trans. Computers C-28, 1979.

[Bou89] Paul Bourke. Intersection point of two lines(2 dimensions). Website, 1989. http://local.wasp.uwa.edu.au/~pbourke/geometry/lineline2d/.

[Cat06a] Erin Catto. Box2d forum. Website, 2006. http://www.box2d.org/forum.

[Cat06b] Erin Catto. Box2d manual. Website, 2006. http://www.box2d.org/manual.

[Cat06c] Erin Catto. Box2d physics engine. Website, 2006. http://www.box2d.org.

[GJK97] The Gilbert-Johnson-Keerthi distance algorithm: a fast version for incremental motions. Website, 1997. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=614298.

[Hec96] Chris Hecker. Rigid body physics. Website, 1996. http://chrishecker.com/Rigid_Body_Dynamics.

[Hir08] Hiroaki. DarwiinRemote version 0.7. Website, 2006-2008. http://sourceforge.net/projects/darwiin-remote/.

[Rat00] John W. Ratcliff. Efficient Polygon Triangulation. Website, 2000. http://www.flipcode.com/archives/Efficient_Polygon_Triangulation.shtml.

[SH76] Michael Shamos and Dan Hoey. Geometric Intersection Problems. In *Geometric Intersection Problems*, pages 208–215. 17-th Ann. Conf. Found. Comp. Sci., 1976.

[Van07] Lode Vandevenne. picoPNG version 20071229. Website, 2005-2007. http://members.gamedev.net/lode/projects/LodePNG/.

[YFPR07] Thomas Y. Yeh, Petros Faloutsos, Sanjay J. Patel, and Glenn Reinman. Parallax: an architecture for real-time physics. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 232–243, New York, NY, USA, 2007. ACM.