HANSER

Leseprobe

Jonas Freiknecht

Spiele entwickeln mit Gamestudio

Virtuelle 3D-Welten mit Gamestudio A8 und Lite-C

ISBN (Buch): 978-3-446-43119-5

ISBN (E-Book): 978-3-446-43267-3

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-43119-5

sowie im Buchhandel.

12 Physik

Kommen wir nun zu einem Thema, das in der Spieleentwicklung erst vor einigen Jahren aufgetaucht ist. Mit *Half Life 2* kam das erste Mal eine Physik-Engine zum Einsatz, die den Namen auch verdiente und es schaffte, Physik als Spielelement und nicht nur als netten Effekt in die Szenerie einzubauen. So musste man etwa Wippen auf einer Seite beschweren, um auf die andere steigen und ein Dach erklimmen zu können. Was aber genau ist eine Physik-Engine?

12.1 Was ist eine Physik-Engine?

Kein Spiel, ausgenommen Textadventures und virtuelle Gesellschaftsspiele, kommt ohne Kollisionsabfrage aus. Wir haben mit *c_trace*, *c_move* und *c_rotate* bereits die wichtigsten Funktionen kennen gelernt, um auch in Lite-C Kollisionen zwischen zwei Objekten zu erkennen. Die Berührung zweier Objekte zu erkennen, basiert auf mehr oder weniger simpler Mathematik. Es gilt zu erkennen, ob sich Ebenen, Linien, Punkte oder polygonale Figuren überschneiden. Physik vereint den weiten Bereich der Mathematik mit der Simulation von Reibung, Bewegung, Impulsen und Schwerkraft. Durch sie haben wir die Chance, unser Spiel realistischer zu gestalten, indem wir den Spieler Gegenstände stapeln lassen, Seile und Wäscheleinen realitätsgetreu aufhängen, Brücken bei zu großer Belastung einbrechen oder einfach nur eine Schaukel auf einem Kinderspielplatz durch einen Stoß schwingen lassen. Eine Physik-Engine ist also eine Software, die in der Lage ist, physikalische Gegebenheiten in einer digitalen Welt abzubilden. Sie hilft den Entwicklern, eine realistische Umgebung zu schaffen und nimmt ihnen dabei Arbeitsaufwand ab, indem sie vorgefertigte Funktionen zur Simulation von physikalischen Prozessen bereitstellt.

Bevor Sie beginnen ein Spiel zu entwickeln, sollten Sie sich überlegen, ob Sie wirklich auf eine Physik-Engine angewiesen sind. Ein *Frogger*-Klon benötigt sicherlich keine Physik, ebenso wenig ein Kartenspiel. Diese Überlegung ist wichtig, da eine aktive Physik-Engine abhängig von der Objektanzahl der aktuellen Szene Rechenleistung der Grafikkarte bzw. des Prozessors benötigt.

Welche Art von Simulationen deckt eine Physik-Engine eigentlich ab? Das ist von Mal zu Mal unterschiedlich. Hier ist eine kurze Liste mit Features, die von den meisten Engines unterstützt werden:

- **Starre Körper** (*rigid body*): Stellen Sie beispielsweise ein Glas auf eine schräge Oberfläche, dann rutscht es langsam hinunter.
- **Weiche Körper** (soft body): Ein Umhang fällt etwa auf den Boden und faltet sich dabei zusammen.
- **Simulation von Flüssigkeiten** (*fluids*): Wasser fließt realistisch. Ein tolles Beispiel bietet das Spiel *From Dust*, in dem Sie eine Welt frei gestalten und etwa auch Wasser in leere Flussbetten fließen lassen können.
- **Ragdolls:** Körperteile einer virtuellen Figur verhalten sich korrekt und können beispielsweise nur über einen bestimmten Winkel gebogen werden. Bei einem Fall fällt jeder Körperteil auf den Boden und das Modell bleibt nicht, wie im Animations-Frame vorgesehen, starr liegen.
- **Gelenke** (*joints*): Zwei Objekte können über verschiedene Gelenke (z. B. Kugelgelenke) miteinander verbunden werden und verändern ihre Position entsprechend dem Bewegungsradius des Gelenks.

12.2 Einige Physik-Engines im Vergleich

Auch wenn Lite-C per Default die *PhysX-Engine*[1] unterstützt, möchte ich Ihnen dennoch ein paar andere Physik-Engines vorstellen. In der Community gibt es mehrere Projekte, in denen entweder die *Havok* oder auch die *Newton-Engine* in Gamestudio-Spiele eingebunden werden.

- **PhysX:** Das SDK von Nvidias-Physik-Engine ist für Konsolen und PCs kostenlos verfügbar. Die Engine hat die Besonderheit, dass die Simulation hardwareseitig durch eine so genannte *Physics Processing Unit* (PPU) unterstützt wird. Somit wird die CPU entlastet. Verwendung findet *PhysX* etwa in *Batman: Arkham City*, *Mafia 2* oder *Mirrors Edge*.
- **Havok:** *Havok* ist neben *PhysX* die andere große Physik-Engine auf dem Markt und wird unter anderem in *Alan Wake*, *BioShock* und *Half-Life 2* eingesetzt.
- **Open Dynamics Engine (ODE):** Als quelloffene C/C++-Bibliothek wird sie häufig in wissenschaftlichen Simulationen, etwa für den Bereich der Robotik eingesetzt. Jedoch findet sie auch häufig in kommerziellen Projekten, wie etwa *BloodRayne 2* oder *Call of Juarez*, Verwendung.
- **Bullet:** Eine weitere Open-Source-Engine, die etwa in *GTA IV* und *Red Dead Redemption* eingesetzt wird

Nun aber genug der Theorie, lassen Sie uns mit dem Programmieren beginnen.

[1] http://www.nvidia.de/object/physx_new_de.html

12.3 Aktivieren der Physik-Engine

Wie bereits erwähnt, ist die Physik-Engine erst einmal deaktiviert, da die Simulation sehr rechenintensiv ist. Um die Physik-Engine zu aktivieren, sind zwei Schritte vonnöten:

- Inkludieren von *ackphysx.h*.
- Aufrufen von `physX_open();`.

Nun können wir ein Level laden und einzelne Objekte in der Physik-Engine registrieren. Dadurch geben wir die Kontrolle über die Objekte aus der Hand, weil deren Position nun nicht mehr über die Positionsvektoren verändert werden kann. Wir können diese dann lediglich über Kräfte und Impulse der Physik-Engine beeinflussen.

Ganz so hilflos sind wir nicht, wenn wir eine von der Physik beeinflusste Entity umsetzen möchten. Die Funktion **pXent_setposition** ermöglicht uns das.

Wenn wir diese Objekte dann nicht mehr benötigen, heben wir die Registrierung auf und schließen am Ende unseres Spiels die Physik-Engine.

Lite-C gliedert die physikalischen Funktionen in vier Kategorien, die durch Präfixe gekennzeichnet sind.

Tabelle 12.1 Präfixe für Physikfunktionen

Präfix	Aufgabenbereich
physX_	Funktionen zum Starten und Stoppen der Physik-Engine
pX_	Funktionen, die das Verhalten der Physik-Engine beeinflussen
pXent_	Physikalische Funktionen, die lediglich ein Entity beeinflussen
pXcon_	Funktionen für Gelenke, die die Bewegung von Entities einschränken

Beginnen wir mit einem einfachen Beispiel, indem wir einen Ball durch ein Level rollen lassen. Wir steuern dabei die Bewegung alleine durch eine Kraft, die wir auf den Ball wirken lassen.

Beispiel 24

Ordner: „24_physikalischer_ball“

Schwierigkeitsgrad: Einfach

Demonstriert: Bewegung eines Balls mit der Physik-Engine

Wir beginnen damit, dass wir neben unseren zwei bekannten Includes nun noch *ackphysx.h* hinzufügen. Darauf erstellen wir uns einen globalen Pointer *entBall* und einen Vektor *vecBallRotate*, der die Drehung unserer Entity speichert.

Listing 12.1 Registrieren einer Entity in der Physik-Engine

```
// Ball wird erstellt und als Kugel in der Physik-Engine registriert.
void initBall() {
   entBall = ent_create ("ball.mdl", vector(-400, 0, 100), NULL);
   pXent_settype (entBall, PH_RIGID, PH_SPHERE);

   // Setzen der Reibung des Balles
   pXent_setfriction (entBall,50);

   // Setzen der Dämpfung des Balles
   pXent_setdamping (entBall,10,10);

   // Setzen der Elastizität
   pXent_setelasticity (entBall,50);
}
```

In der Methode *initBall* treffen wir auf einige neue Funktionen.

1. Zu Beginn erstellen wir unseren Ball über den gewohnten Weg mit *ent_create*. Dann registrieren wir *entBall* in der Physik-Engine über **pXent_settype** und teilen ihr mit, dass wir einen beweglichen starren Körper haben wollen (*PH_RIGID*), der die Hülle einer Kugel hat (*PH_SPERE*).
2. Mit **pXent_setfriction** teilen wir der Engine mit, dass unser Ball eine mittelmäßige Reibung aufweisen soll, sodass er leicht am Terrain hängen bleibt, auf dem er sich bewegt. Der Wert 0, würde keine Reibung darstellen, wie es etwa bei einer Bewegung im Weltall vorkommen würde. Der Wert 100 hätte eine sehr starke Reibung. Ein Wert über 100 würde eine unendlich starke Reibung angeben.
3. **pXent_setdamping** setzt eine Geschwindigkeitsdämpfung für unseren Ball. Der zweite Parameter sagt aus, dass unser Ball an Geschwindigkeit verlieren soll, wenn eine Kraft, etwa über **pXent_addforceglobal** auf ihn einwirkt, und der dritte Parameter, wenn, wie in unserem Falle, ein Drehmoment, etwa über **pXent_addtorqueglobal** auf ihn einwirkt.
4. Schließlich legen wir über **pXent_setelasticity** fest, wie sehr unser Ball von anderen Objekten abprallt. Bei einem Wertebereich von 0 bis 100 legen wir auch hier eine mittelmäßige Elastizität fest.

Kommen wir nun zur Methode *main*.

Listing 12.2 Starten der Physik-Engine und Anwenden von Kräften auf eine Entity

```
void main() {
   // Starten der Physik-Engine
   physX_open();
   level_load("terrain1.hmp");

   // Erstellen eines Balles
   initBall();

   while (1) {

      // Wir erstellen ein Drehmoment, das wir auf den Ball einwirken
      // lassen
      vecBallRotate.x = (key_cur-key_cul)*10*time_step;
      vecBallRotate.y = (key_cuu-key_cud)*10*time_step;
```

```
      vecBallRotate.z = 0;
      pXent_addtorqueglobal (entBall, vecBallRotate);

      // Die Kamera folgt dem Ball
      camera.x = entBall.x - 300;
      camera.y = entBall.y;
      camera.z = 500;
      camera.tilt = -60;

      // Sollte der Ball vom Level rollen, wird er neu erstellt
      if (entBall.z < -100) {
         ptr_remove(entBall);
         initBall();
      }

      wait(1);
   }
}
```

1. Wie besprochen, öffnen wir die Physik-Engine über *physX_open* und laden unser bekanntes Terrain. In einer Schleife setzen wir unseren Drehwinkel abhängig von den Pfeiltasten. Eine Rotation um die z-Achse (wie bei einem Kreisel) benötigen wir nicht, da wir mit unserem Ball nur vorwärts und seitwärts rollen wollen.
2. **pXent_addtorqueglobal** fügt nun diesen Drehwinkel zu unserem Ball hinzu. Stellen Sie sich das so vor als würden Sie einen Tennisball nicht anstupsen, sondern ihn dazu bringen, sich zu drehen. Durch seinen Reibungswiderstand wird er sich dann auch bewegen.
3. Im Folgenden sorgen wir dafür, dass die Kamera unserem Ball folgt. Mit der letzten If-Abfrage prüfen wir, ob der Ball eventuell in das Terrain hinuntergefallen ist. Wenn ja, löschen wir die Entity mit *ptr_remove* und erstellen einen neuen Ball über *initBall*.

Nun haben wir mit weniger als 40 Zeilen Code einen Ball physikalisch korrekt rollen lassen. Aber es kommt noch besser. Lassen Sie uns nun ein kleines Minispiel programmieren, das später auch in unserem RPG Verwendung finden soll.

Wenn wir so weit sind, unser Spiel auszuliefern, müssen wir noch einige DLLs aus dem Gamestudio-Ordner in unser Spieleverzeichnis kopieren:

DLL	Ort
ackphysX.dll	GStudio8\acknex_plugins\
PhysXLoader.dll	GStudio8\
PhysXCooking.dll	GStudio8\
NxCharacter.dll	GStudio8\

Verstärken Sie die Krafteinwirkung auf den Ball, indem Sie *vecBallRotate* den Wert > 1 geben.

12.4 Dosenwerfen

Wir wissen bereits wie wir einen Ball rollen lassen können. Interessant wäre nun zu erfahren, wie wir ihn werfen, also wie wir eine Kraft an einem bestimmten Punkt des Balles anbringen. Das wollen wir in diesem Abschnitt lernen und dazu auch gleich ein kleines Spiel implementieren, das wir später wiederverwenden können.

Bild 12.1 Dosenwerfen auf Basis einer Physik-Engine

Beispiel 25

Ordner: „25_dosenwerfen“

Schwierigkeitsgrad: Schwer

Demonstriert:

- Kollision zwischen Physik-Entities
- Speicherreservierung
- Einfacher Gameplay-Ablauf
- Himmelstexturen

Öffnen wir nun die *main.c* im Ordner *25_dosenwerfen* und beginnen wir bei der Methode *main*.

Listing 12.3 Hauptmethode der Dosenwerfen-Demo

```
void main() {
   physX_open();
   level_load("level1.wmb");

   // Arrays für Fässer und Positionen werden initialisiert
```

```
    entBarrels = (ENTITY*)sys_malloc(sizeof(ENTITY*)*6);
    vecBarrelPos = (VECTOR*)sys_malloc(sizeof(VECTOR*)*6);

    // Wir setzen die Vektoren, um eine Fässerpyramide erzeugen zu können
    vecBarrelPos[0] = vector(-24, 44, 8);
    vecBarrelPos[1] = vector(0, 44, 8);
    vecBarrelPos[2] = vector(24, 44, 8);
    vecBarrelPos[3] = vector(-12, 44, 40);
    vecBarrelPos[4] = vector(12, 44, 40);
    vecBarrelPos[5] = vector(0, 44, 72);

    vec_set(camera.x, vector(0, -144, 28));

    // Platziere die Fässer
    initBarrels();

    // Mit der linken Maustaste werfen wir einen Ball
    on_mouse_left = throwBall;

    // Mit Space stellen wir die Fässer neu auf
    on_space = initBarrels;

    // Mit ESC beenden wir das Spiel
    on_esc = exitGame;

    // Mit D schalten wir das Debug-Panel an und aus
    on_d = toggleDebug;

    while (1) {

       updateBarrelDebug();
       camera.pan -= mouse_force.x * time_step*MOUSE_SENSITIVITY;
       camera.tilt = clamp(
          camera.tilt+mouse_force.y*time_step*MOUSE_SENSITIVITY,
          -30,
          30
       );

       // Wenn alle Fässer unter den Tisch gefallen sind dann
       // stelle die Fässer neu auf
       if ((v1 < 0) && (v2 < 0) &&
           (v3 < 0) && (v4 < 0) &&
           (v5 < 0) && (v6 < 0)) {
             initBarrels();
       }
       wait(1);
    }
}
```

1. Zu Anfang öffnen wir die Physik-Engine wie gehabt und laden ein Level, den ich im WED vorbereitet habe. Schauen Sie sich dieses bei Interesse gerne kurz an, indem Sie im entsprechenden Ordner *level1.wmp* öffnen.
2. Wir reservieren uns nun Speicher für 6 Entity-Pointer, die später einmal auf unsere Dosen zeigen sollen. Ebenso erzeugen wir einen Array von Vektoren, die die Positionen unserer Fässer speichern. Die Zahlen dafür weisen wir direkt im Anschluss zu. Am einfachsten ist es, diese im WED abzulesen.

3. Wir setzen die Kamera auf eine Position, von der aus wir unsere Wurfbude gut sehen können.
4. Dann weisen wir einigen Tasten auf der Tastatur und Maus Funktionen zu.
   - **Linksklick:** Ballwerfen
   - **Leertaste:** Fässer neu aufstellen
   - **Escape:** Spiel beenden und Speicher aufräumen
   - **D:** Debug-Panel anzeigen, welches wiederum die genaue Position der Fässer anzeigt
5. In der nun folgenden While-Schleife updaten wir das Debug-Panel, sorgen dafür, dass wir uns mit der Maus umschauen können und überprüfen zuletzt in der langen If-Abfrage, ob alle unsere Fässer bereits unter den Tisch gefallen sind. Wenn ja, stellen wir diese neu auf.

Kommen wir nun zum Aufstellen der Fässer.

Listing 12.4 Aufstellen der Dosen

```
// Platzieren der Fässer
void initBarrels() {
   int i;
   for (i=0; i<6; i++) {

      // Lösche die alten Fässer wenn nötig
      if(entBarrels[i] != NULL) {
         pXent_settype(entBarrels[i], 0, 0);
         ptr_remove(entBarrels[i]);
         entBarrels[i] = NULL;
      }

      // Erstelle die Fässer neu
      entBarrels[i] = ent_create("barrel.mdl", vecBarrelPos[i], NULL);

      // ... mache sie "physikalisch"
      pXent_settype(entBarrels[i], PH_RIGID, PH_BOX);

      // ... und verkleinere deren Masse, damit wir sie mit unserem
      // kleinen Ball beeinflussen können.
      var mass = pXent_getmass(entBarrels[i]);
      pXent_setmass(entBarrels[i], mass / 2);
   }
   vThrows = 0;
}
```

1. Da wir einen Array von Entities benutzen, können wir ganz komfortabel mit einer For-Schleife über unsere Fässer iterieren. Wir überprüfen zuerst, ob die alten Fässer noch existieren. Wenn ja, …
   a) … heben wir die Registrierung per *pXent_settype* aus unserer Physik-Engine auf, indem wir als zweiten Parameter 0 übergeben. Damit weiß die Engine, dass sie sich nicht mehr um die Entity kümmern muss.
   b) Folgend löschen wir das aktuelle Fass und setzen den Pointer auf *NULL*. Man kann nicht oft genug betonen, dass dieses Vorgehen wichtig ist, da wir sonst weiterhin bei der Abfrage `entBarrels[i] != NULL` einen Speicherbereich gezeigt bekommen würden, der aber gar keine Entity mehr enthält.

2. Nun können wir uns der Aufgabe der Erstellung unserer neuen Fässer widmen. Wir erzeugen, wie schon oft gesehen, mit *ent_create* ein neues Fass und registrieren dieses mit *pXent_settype* in der Physik-Engine. Wir sagen dieser mit den Parametern, dass es sich um einen sich bewegenden Körper handelt (*PH_RIGID*), der bitte als Box (*PH_BOX*) behandelt werden soll.
3. Da unser Ball, den wir gleich erstellen, leichter ist als unsere Fässer, müssen wir die physikalische Masse reduzieren. Ich hole hier mit **pXent_getmass** die aktuelle Masse unseres Fasses, teile sie durch zwei und weise sie dem Fass wieder zu. Damit gehen wir sicher, dass unser Ball nicht zu leicht ist, um unsere Fässer nicht bewegen zu können.
4. Zuletzt setzen wir die globale Variable *vThrows* auf 0. Wir möchten nämlich auch zählen, wie viele Würfe der Spieler braucht, um alle Fässer abzuräumen, um ihm dann später einen entsprechenden Preis geben zu können.

Woher weiß die Engine eigentlich, was für eine Masse eine Entity hat? Die Antwort ist: Die Physik-Engine berechnet die Masse aus der Form und der Größe der Entity.

Nun bleibt nur noch der Ballwurf zu erklären.

Listing 12.5 Werfen eines Balls

```
// Aktion für den Ball. Nach 5 Sekunden Lebenszeit wird die Registrierung des
// Balls in der Physik-Engine entfernt.
action actBall() {
   wait(-5);
   pXent_settype(me, 0, 0);
   ptr_remove(me);
}

// Funktion zum Werfen eines Balls
void throwBall() {

   // Der Ball wird erstellt
   ENTITY* entBall = ent_create(
      "ball.mdl",
      vector(camera.x, camera.y, camera.z),
      actBall);

   VECTOR vecBallForce;
   // Wir berechnen einen Vektor, der in die Blickrichtung unserer Kamera zeigt
   vec_for_angle(vecBallForce.x, camera.pan);
   // Wir verlängern den Vektor, damit er "kräftiger" wird
   vec_scale(vecBallForce.x, 24);

   // Der Ball wird in der Physik-Engine registriert
   pXent_settype (entBall, PH_RIGID, PH_SPHERE);

   // und eine Kraft in Blickrichtung der Kamera wird auf ihn angewendet.
   pXent_addforcelocal (entBall, vecBallForce, entBall.x);
   vThrows +=1;
}
```

1. Auch hier erstellen wir eine einfache Entity genau da, wo unsere Kamera sitzt (`vector(camera.x, camera.y, camera.z)`) und weisen ihr die Aktion *actBall* zu, die dafür sorgt, dass die Registrierung des Balls nach 5 Sekunden gelöscht wird. Schließlich kann es passieren, dass wir sonst viel zu viele Entities im Level hätten und unser Spiel merklich langsamer werden würde.
2. In diesem Abschnitt soll ja gezeigt werden, wie sich ein Ball durch eine Kraft bewegt. Diese Kraft berechnen wir im Vektor *vevBallForce*. Zu Beginn wollen wir erreichen, dass der Vektor genau in die Blickrichtung unserer Kamera weist. Mit der Funktion *vec_for_angle* setzen wir diese Anforderung recht schnell um. Nun haben wir also eine Kraft, die in genau eine Richtung wirkt. Allerdings ist unsere Kraft noch zu klein, als dass sie unsere Fässer bewegen könnte. Mit *vec_scale* können wir unseren Vektor skalieren, respektive unsere Kraft verstärken. Das tun wir um den Faktor 24. Hier gilt es, ein wenig mit dem Faktor zu experimentieren.
3. Nun registrieren wir unseren Ball in der Physik-Engine als sich bewegende Kugel.
4. Mit *pXent_addforcelocal* lassen wir unsere Kraft *vecBallForce* am Punkt `entBall.x`, also genau im Mittelpunkt des Balles wirken. Die Physik-Engine stößt nun den Ball in Richtung *vecBallForce*.
5. Wir erhöhen *vThrows* um 1, um einen weiteren Wurf zu zählen.

Das waren die wesentlichen Funktionen unserer Wurfbude. *exitGame* räumt lediglich die Arrays der Fässer und Vektoren auf und beendet dann das Spiel. *updateBarrelDebug* weist ein paar Dummy-Variablen die Werte der z-Position der Fässer zu, damit wir sie im Debug-Panel darstellen können.

12.4.1 Exkurs: Ein realistischer Himmel

Ein kleines Schmankerl habe ich in diesem Kapitel noch für Sie. Schauen Sie sich im Level mal den Himmel an. Statt einer langweiligen blauen Leere sehen wir nun einen wirklich schönen Horizont und sogar ein paar Wolken. Das erreichen wir über einen so genannten Sky-Cube, der eigentlich mehr in Kapitel 21 über das Level-Design gehört, aber hier dennoch kurz erwähnt werden soll.

Listing 12.6 Definition eines Himmels mittels Sky-Cube

```
ENTITY* skycube = {
   type = "skyday+6.tga";
   flags2 = SKY | CUBE | SHOW;
}
```

Bild 12.2 Ein Sky-Cube im Spiel

Wenn wir bei einer Entity die Flags *SKY* und *CUBE* aktivieren, werden diese als Kubus verwendet, der einen Himmel darstellt. Dieser Himmel ist eine Bitmap mit der Endung *+6*, was aussagt, dass sie aus 6 Teilen besteht. Schauen wir uns eine solche einmal an.

Bild 12.3 Ein Sky-Cube im Rohformat

Um unsere Welt wird ein riesiger hohler Kubus gelegt, dessen Innenflächen mit den einzelnen Bauteilen der Bitmap gefüllt werden. Die Engine sorgt dann dafür, dass wir die Kanten und Übergänge nicht mehr sehen und uns der Kubus als runder Körper erscheint. Als Alternative zu *CUBE* können Sie auch das Flag *DOME* setzen, um eine kuppelartige Himmelsstruktur zu erhalten. Diese eignet sich etwa für einfache Sternenhimmel oder Wolken, wie wir später sehen werden. Es ist auch möglich, mehrere Himmel zu kombinieren, etwa wenn man Wolken auf einem Sky-Cube darstellen möchte. Dabei spielt das Schlüsselwort *layer* wieder eine Rolle. Himmel mit höherem Layer werden zuoberst dargestellt. Wie genau Wolken als *DOME* und Himmel als *CUBE* dargestellt werden, können Sie in Abschnitt 13.4 über die Wettereffekte nachlesen.

Programmtechnisch erzeugt man einen Himmel wie folgt:

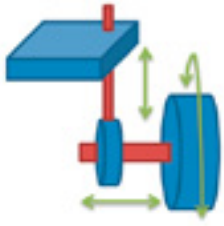
```
ent_createlayer("skyday+6.tga", SKY | CUBE | SHOW, 1);
```

12.5 Gelenke

Gelenke dienen dazu, die Bewegung von Entities einzuschränken, indem sie über gewisse Kriterien aneinander gebunden werden. Wenn Sie beispielsweise zwei Bälle miteinander verbinden oder eine Kette erstellen möchten, deren Glieder aneinanderhängen und auf äußere Einflüsse richtig reagieren, dann sind Sie an dem Punkt angelangt, an dem Sie Gelenke einsetzen sollten.

Lite-C bietet dafür sechs verschiedene Gelenke (*Joints*) an.

Tabelle 12.2 Verschiedene Gelenktypen in Lite-C

Gelenk	Beschreibung
PH_HINGE (*Drehgelenk* oder *Angelgelenk*) 	Ein Angelgelenk verbindet zwei Entities so, dass beide an eine gemeinsame Rotationsachse gebunden sind. Stellen Sie sich dazu etwa eine Tür vor, die mit dem Türrahmen über ein *PH_HINGE*-Gelenk verbunden ist. Den Winkel können wir für unser Gelenk einschränken, ebenso, wie es in der Realität bei der Tür der Fall ist. Diese kann sich ja meistens auch von −90° bis 90° bewegen.
PH_BALL (*Kugelgelenk*) 	Rotieren Sie mal Ihre Hand am Handgelenk. Wenn Sie diese Aufgabe gemeistert haben, wissen Sie auch gleich was ein Kugelgelenk ist.
PH_SLIDER (*Dreh-Schub-Gelenk*) 	Ein solches Gelenk lässt zwei Entities über eine Achse aufeinander zu oder voneinander weg rutschen. Stellen Sie sich vor, Sie hätten zwei Ringe an einem Finger und könnten diese bewegen. Beide sind an die Achse (Ihren Finger) gebunden. Ebenso wie beim *PH_SLIDER* können Sie die Ringe auch noch um Ihren Finger drehen.
PH_WHEEL (*Rad*) 	Das Rad ist eigentlich kein Gelenk, wird aber hier dennoch aufgeführt. Räder können sich, um zu beschleunigen oder um zu bremsen, um eine Achse drehen, und um zu lenken, also um nach links und rechts zu schwenken, um eine weitere Achse.

Gelenk	Beschreibung
PH_ROPE (*Seil*) 	Ein Seil verbindet zwei Entities und sorgt dafür, dass diese niemals weiter als eine angegebene Entfernung voneinander wegbewegt werden und nie näher als eine bestimmte Entfernung aufeinander zu bewegt werden.
PH_6DJOINT (*All-Around-Gelenk*)	Mit diesem Gelenk lässt sich jede beliebige Art von Gelenk abbilden.

Das übliche Vorgehen zum Erstellen von Joints ist, dass die Entities, die verknüpft werden sollen, zunächst einmal erstellt werden. Dann registrieren Sie diese in der Physik-Engine über *pXent_settype*, verbinden sie über **pXcon_add** und konfigurieren das neue Gelenk letztendlich über **pXcon_setparams1** und **pXcon_setparams2.** Wenn das Gelenk einmal nicht mehr gebraucht wird, kann es über **pXcon_remove** gelöscht werden. Weiterhin besteht für Gelenke die Möglichkeit, zu brechen. Wenn eine zu große Kraft (*Force*) oder eine zu starke Drehung (*Torque*) auf sie einwirkt, wird das Gelenk automatisch gelöst. Die Schwellenwerte für Kraft und Drehung können wir individuell festlegen. Nun folgen ein paar Beispiele zum Nachvollziehen.

Beispiel 26

Ordner: „26_gelenke“

Schwierigkeitsgrad: Schwer

Demonstriert:

- Demonstration aller vorhandenen Elemente
- Konstruktion eines einfachen Autos, das sich physikalisch korrekt verhält

Wenn Sie das Beispiel ausführen, merken Sie, dass überall im Raum Bälle verteilt sind, die über die verschiedenen Gelenke verbunden sind. Mit der Leertaste können Sie weitere Bälle auf die Konstruktionen abschießen und schauen, wie sich diese verhalten.

12.5.1 Drehgelenke mit PH_HINGE

Wir beginnen unsere Erklärung mit *PH_HINGE*. Dabei handelt es sich um die kleinen Bälle, von denen der eine festsitzt und der andere bei Beschuss um den festen Ball herum rotiert.

Listing 12.7 Ein Drehgelenk mit PH_HINGE

```
// Erstellen eines Türgelenks
void createHinge(VECTOR* _offset, char* _model, int _angleDistance) {
   ENTITY* entHinge1 = ent_create(_model, _offset, NULL);
   ENTITY* entHinge2 = ent_create(
      _model,
      vector(_offset.x+40, _offset.y, _offset.z),
```

```
      NULL);

   entHinge2.pan = entHinge1.pan+_angleDistance;

   set(entHinge1, PASSABLE);
   set(entHinge2, PASSABLE);

   // Die beiden erstellten Entities werden in Boxform in der Physik-Engine
   // registriert
   pXent_settype(entHinge1, PH_RIGID, PH_BOX);
   pXent_settype(entHinge2, PH_RIGID, PH_BOX);

   // Über ein Türgelenk wird erst die erste Entity mit der Welt verbunden...
   pXcon_add(PH_HINGE, entHinge1, NULL, 0);
   // ... und die zweite dann mit dem ersten
   pXcon_add(PH_HINGE, entHinge1, entHinge2, 0);

   // Festlegen der Bewegungsmöglichkeiten der Entities
   pXcon_setparams1(entHinge1, NULL, vector(0,0,-1), NULL);
   pXcon_setparams2(entHinge1, vector(-90, 90, 0), NULL, NULL);

   pXcon_setparams1(entHinge2, NULL, vector(0,0,-1), NULL);
   pXcon_setparams2(entHinge2, vector(-90, 90, 0), NULL, NULL);
}
```

1. Wir erstellen zwei Entities, die in einem Abstand von 40 Quants nebeneinander platziert werden.
2. Beide Entities werden passierbar gemacht, sodass wir, also die Spieler-Entity, nicht mit ihnen kollidieren.
3. Wir registrieren beide als bewegliche Körper in der Physik-Engine.
4. Nun greifen wir das erste Mal zu **pXcon_add.** Normalerweise verbindet diese Funktion zwei Entities. Wir übergeben aber beim ersten Aufruf als zweite Entity eine „NULL“. Damit sagen wir der Physik-Engine, dass wir die erste Entity *entHinge1* an die Welt binden wollen, sodass diese sich nicht vom Fleck bewegen kann. Ich habe das hier als sinnvoll erachtet, da wir ja sehen möchten, wie sich unser zweiter Ball um den ersten drehen kann.
5. Mit dem zweiten Aufruf von *pXcon_add* binden wir nun *entHinge1* an *entHinge2* und übergeben als ersten Parameter den Gelenktyp. Der letzte Parameter, hier die 0, sagt aus, dass unsere Objekte nicht miteinander kollidieren sollen. Steht der Wert auf 1, dann findet eine Kollision statt.
6. Nun müssen wir unserem Gelenk noch die passenden Parameter zuweisen. Die Funktionen *pXcon_setparam1* und *pXcon_setparam2* sind generisch einsetzbar und somit für jede Art von Gelenk zu verwenden. Jeder Parameter wird als Vektor übergeben. Manche Parameter haben einen Default-Wert. Wenn wir diesen verwenden möchten, übergeben wir für diesen Parameter *NULL*. Die Parameter für *PH_HINGE* sind in Tabelle 12.3 aufgelistet.

Über *pXcon_setparam1* und *pXcon_setparam2* können wir für ein Gelenk insgesamt sechs Parameter festlegen. Ich werde in den folgenden Tabellen nur die nötigen Parameter nennen. Die anderen müssen Sie je nach Anwendungsfall mit NULL oder mit *nullvector* belegen. Das Handbuch gibt dazu Auskunft.

Tabelle 12.3 Parameter für das Gelenk *PH_HINGE*

Parameter	Verwendungszweck
1	Position des Gelenks, an dem die beiden Entities verbunden sind. *Default*: Mittige Position zwischen beiden Entities.
2	Achsenrichtung für das Gelenk. Für eine Tür verwenden Sie etwa `vector(0,0,1)`, um das Gelenk an der z-Achse umschwenken zu lassen. *Default:* Schwenken an der z-Achse.
3	Schwellenwerte für Kraft und Drehung, um das Gelenk zu zerstören. *x* steht für Kraft, *y* für Drehung, z. B. `vector(25,25,0)`. Wenn Sie nicht möchten, dass das Gelenk bricht, übergeben Sie *NULL*.
4	Erlaubte Winkel der einzelnen Entities. *x* steht für den kleinstmöglichen Winkel um den sich die Entity um die Achse drehen darf, *y* für den größten z. B. `vector(-45,45,0)`. **Achtung:** Diese Winkel können nur einmal im Spiel gesetzt werden. Wollen Sie diese zu einem späteren Zeitpunkt ändern, müssen Sie das Gelenk neu erstellen.

In unserem Test-Level bewegt sich eine Entity gar nicht da wir sie, wie bereits erwähnt, an der Welt *fest gemacht* haben. Somit ist das Verhalten der Winkelrestriktionen schwer zu erkennen. Achten Sie aber einmal darauf, dass, wenn Sie mit einem Ballschuss die Richtung des sich drehenden Balles ändern, der Ball im Mittelpunkt erst anfängt zu rotieren, wenn der andere Ball um 90° um ihn herumgewandert ist.

12.5.2 Kugelgelenk mit PH_BALL

In diesem Teil will ich Ihnen neben der Funktionsweise von *PH_BALL* zeigen, wie man mehrere Entities über mehrere Gelenke verknüpft. Dazu habe ich eine dynamische Funktion erstellt, mit der Sie beliebig viele Glieder an einer Art Kette aneinanderreihen können, die dann alle über ein Kugelgelenk verbunden werden.

Listing 12.8 Ein Kugelgelenk mit PH_BALL

```
// Erstellen einer Kette aus Bällen
void createBallChain(int _elements, VECTOR* _offset, char* _model, int
_modelHeight) {
   int i;

   // Wir registrieren so viel Speicher, wie wir für die gesamte Kette
   // brauchen
   entChain = (ENTITY*)sys_malloc(sizeof(ENTITY*)*_elements);

   // Wir erstellen alle Elemente der Kette und verknüpfen diese
   for (i=0; i<_elements; i++) {
      entChain[i] = ent_create(
         _model,
         vector(_offset.x, _offset.y, _offset.z-_modelHeight*i),
         NULL
      );
```

```
      set(entChain[i], PASSABLE);
      pXent_settype(entChain[i], PH_RIGID, PH_SPHERE);

      // Das erste Element der Kette wird an der Welt "fest gebunden", alle
      // anderen an dem Vorigen.
      if (i>0) {
         pXcon_add(PH_BALL, entChain[i], entChain[i-1], 1);
      } else {
         pXcon_add(PH_BALL, entChain[i], NULL, 1);
      }

      // Einschränken der Bewegungsmöglichkeiten der Gelenke
      pXcon_setparams1(entChain[i], vector(
            entChain[i].x,
            entChain[i].y,
            entChain[i].z+entChain[i].max_z), vector(0,-1,0),
         nullvector
      );
      pXcon_setparams2 (entChain[i], vector(-45,45,0), NULL, NULL);
   }
}
```

1. Wie Sie sehen, können wir als Parameter der Funktion setzen, wie viele Elemente wir in unserem Gelenk erstellen möchten. Für diese Gelenke brauchen wir einen Array von Entities, für den wir im ersten Schritt auf dem bekannten Wege Speicher reservieren.
2. In einer Schleife erstellen wir nun diese Entities, wobei die Erste am höchsten Punkt hängt und die Folgenden darunter aufgehängt werden und zwar im Abstand von `_offset.z - _modelHeight * i`.
3. Wir registrieren unsere Elemente Stück für Stück in der Physik-Engine. Welche Form Sie angeben, sei es *PH_SPHERE*, *PH_BOX* oder *PH_CAPSULE*, hängt natürlich von dem Modell ab, das Sie verwenden. Ein Sphere ist ein Kreis, eine Box ein Würfel und eine Kapsel im entfernten Sinne eine Coladose.
4. Nun verbinden wir die Elemente über das Gelenk *PH_BALL* miteinander, wobei wir immer das Aktuelle mit dem Vorigen verknüpfen. Da das erste Element (`i==0`) keinen Vorgänger hat, machen wir es an der Welt fest, sodass unsere Kette sich nicht durch das Level bewegen kann.
5. Nun setzen wir noch die Parameter über unser Kugelgelenk. Dazu folgt eine kleine Tabelle.

Tabelle 12.4 Parameter für das Gelenk PH_BALL

Parameter	Verwendungszweck
1	Position des Gelenks, an dem die Entities miteinander verbunden sind. *Default:* Mittige Position zwischen beiden Entities.

Die Erstellung eines Kugelgelenks ist, wie Sie sehen, nicht besonders anspruchsvoll. Setzen wir unsere Reise durch die Physik mit dem Drehschubgelenk fort.

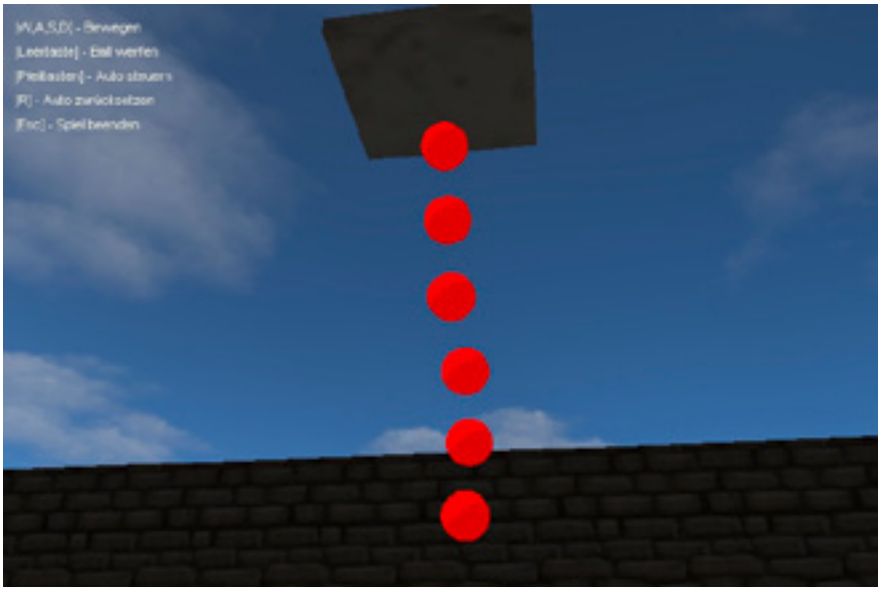


Bild 12.4 Das Kugelgelenk in der Demo

12.5.3 Drehschubgelenk mit PH_SLIDER

Mit *PH_SLIDER* können wir dafür sorgen, dass sich Entities nur aufeinander zu und voneinander weg bewegen dürfen.

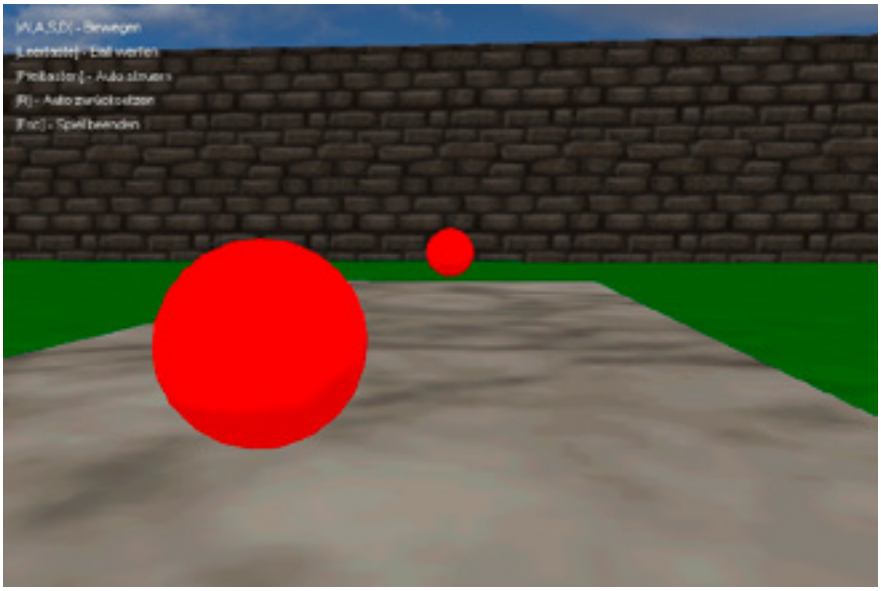


Bild 12.5 Das Drehschubgelenk in der Demo

Bitte beachten Sie, dass aufgrund der physikalischen Gegebenheiten von Masse und Krafteinwirkung auch eine kleine Bewegung in Richtung der nicht ausgewählten Achsen stattfinden kann. In der Demo werden Sie merken, dass, wenn Sie den Ball von der Seite her beschießen, er sich auch ein Stück in diese Richtung bewegt. Das Gelenk ist also nicht fest. Man müsste sich *PH_SLIDER* am besten so vorstellen, dass zwei Objekte an einer Schnur aufgefädelt sind und daran entlang rutschen können. Wenn man aber fest genug von der Seite gegen die Schnur drückt, wird auch eine kleine Bewegung in diese Richtung möglich.

Listing 12.9 Ein Drehschubgelenk mit PH_SLIDER

```
// Erstellen eines Schiebegelenks
void createSlider(VECTOR* _offset, char* _model, int _distance) {
   ENTITY* entSlider1 = ent_create(_model, _offset, NULL);
   ENTITY* entSlider2 = ent_create(
      _model,
      vector(_offset.x+_distance, _offset.y, _offset.z),
      NULL
   );

   set(entSlider1, PASSABLE);
   set(entSlider2, PASSABLE);

   pXent_settype(entSlider1, PH_RIGID, PH_BOX);
   pXent_settype(entSlider2, PH_RIGID, PH_BOX);

   pXcon_add(PH_SLIDER, entSlider1, NULL, 0);
   pXcon_add(PH_SLIDER, entSlider2, entSlider1, 0);

   pXcon_setparams1(entSlider1, NULL, vector(-1,0,0), NULL);
   pXcon_setparams2(entSlider1, vector(0, 0, 0), NULL, NULL);
   pXcon_setparams1(entSlider2, NULL, vector(-1,0,0), NULL);
   pXcon_setparams2(entSlider2, vector(-20, 140, 0), NULL, NULL);
}
```

1. Wir erstellen zwei Entities, die über eine gewisse Distanz *_distance* versetzt sind.
2. Wir machen sie passierbar und registrieren sie in der Physik-Engine als bewegliche Objekte.
3. Über ein erstes Gelenk verbinden wir ein Entity *entSlider1* mit der Welt und in einem anderen bilden wir ein Gelenk zwischen *entSlider1* und *entSlider2*.
4. Entsprechend den Parametern aus Tabelle 12.5 wird das Gelenk angepasst.

Tabelle 12.5 Parameter für das Gelenk PH_SLIDER

Parameter	Verwendungszweck
2	Achse, auf der sich die Entities bewegen dürfen. *Default:* Bewegung auf der z-Achse mit `vector(0,0,-1)`.
3	Schwellenwerte für Kraft und Drehung, um das Gelenk zu zerstören. *x* steht für Kraft, *y* für Drehung, z. B. `vector(25,25,0)`. Wenn Sie nicht möchten, dass das Gelenk bricht, übergeben Sie *NULL*.
4	Maximale Distanz zwischen den zwei Entities auf der in Parameter 2 angegebenen Achse. `vector(100,1000,0)` lässt beispielsweise eine Bewegung von insgesamt 1100 Quants auf der gewählten Achse zu.

12.5.4 Räder mit PH_WHEEL

Wie bereits erwähnt, handelt es sich bei *PH_WHEEL* nicht wirklich um ein Gelenk und wir müssen deswegen auch unsere Räder-Entities nicht in der Physik-Engine als physikalische Objekte registrieren. In diesem Beispiel möchte ich Ihnen zeigen, wie wir in Lite-C mit nur wenigen Zeilen Code ein Auto konstruieren, dessen Fahrverhalten sich wirklich sehen lassen kann.

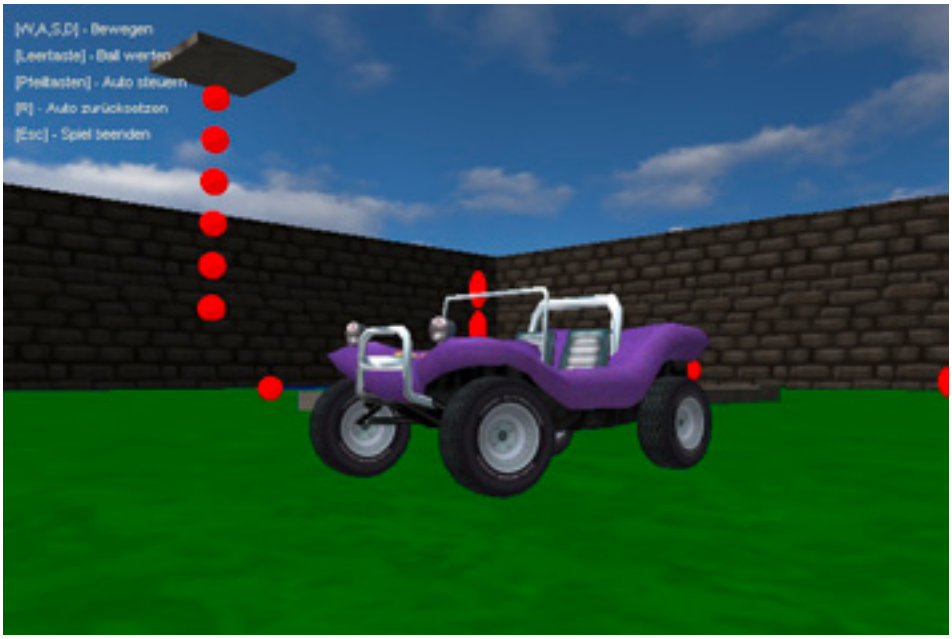


Bild 12.6 Ein sich physikalisch korrekt verhaltendes Auto

Listing 12.10 Erstellen eines Autos mit vier Rädern

```
// Erstellen eines Autos, das sich physikalisch korrekt verhält
void createCar(VECTOR* _offset, char* _modelCar, char* _modelWheel) {

   // Zuerst wird der Rumpf erstellt
   entCar = ent_create(_modelCar, _offset, NULL);

   // ... und registriert
   pXent_settype(entCar, PH_RIGID, PH_BOX);

   // Dann werden alle 4 Räder des Autos erstellt
   entFrontLeftWheel = ent_create(
      _modelWheel,
      vector(_offset.x+14, _offset.y+15, _offset.z),
      NULL
   );

   entFrontRightWheel = ent_create(
      _modelWheel,
```

```
      vector(_offset.x+14, _offset.y-15, _offset.z),
      NULL
   );

   entBackLeftWheel = ent_create(
      _modelWheel,
      vector(_offset.x-31, _offset.y+15, _offset.z),
      NULL
   );

   entBackRightWheel = ent_create(
      _modelWheel,
      vector(_offset.x-31, _offset.y-15, _offset.z),
      NULL
   );

   // Da diese über das Gelenk "PH_WHEEL" erstellt werden, das eigentlich kein
   // richtiges Gelenk ist, müssen wir die Räder auch nicht als Physik-Entities
   // registrieren!
   pXcon_add(PH_WHEEL, entFrontLeftWheel, entCar, 0);
   pXcon_add(PH_WHEEL, entFrontRightWheel, entCar, 0);
   pXcon_add(PH_WHEEL, entBackLeftWheel, entCar, 0);
   pXcon_add(PH_WHEEL, entBackRightWheel, entCar, 0);

   // Solange unser Auto existiert, bewege die Räder mit den Pfeiltasten. Da
   // diese mit dem Auto verbunden sind, wird das gesamte Auto bewegt.
   while(entCar != NULL) {
      pXcon_setwheel(
         entFrontLeftWheel,
          (key_cur - key_cul)*500*time_step,
         0,
         0
      );

      pXcon_setwheel(
         entFrontRightWheel,
          (key_cur - key_cul)*500*time_step,
         0,
         0
      );

      pXcon_setwheel(
         entBackLeftWheel,
         0,
          (key_cud - key_cuu)*100000*time_step,
         0
      );

      pXcon_setwheel(
         entBackRightWheel,
         0,
          (key_cud - key_cuu)*100000*time_step,
         0
      );
      wait(1);
   }
}
```

1. Die Karosserie eines Autos wird erzeugt und in der Physik-Engine registriert.
2. Vier Räder werden geladen und an der vorher berechneten Stelle der Karosserie platziert.
3. Mit *pXcon_add* fügen wir jedes Rad unserer Karosserie hinzu und achten darauf, dass diese nicht kollidieren können, indem wir den letzten Parameter auf 0 setzen.
4. In einer Schleife, die so lange läuft wie das Auto existiert, beeinflussen wir nun die Räder mit **pXcon_setwheel.** Dabei übergeben wir zuerst das Rad, das wir drehen wollen, dann dessen Neigungswinkel (*steerAngle*) nach links und rechts. Den benötigen wir, um unser Auto um die Kurve fahren zu lassen. Der zweite Parameter (*MotorTorque*) bestimmt die Beschleunigung, also die Drehung des Rades. Der dritte Parameter (*BreakTorque*) wird, wie der Name vermuten lässt, zum Bremsen eingesetzt. Diesen lassen wir in unserem Beispiel auf 0, da wir auch bremsen können, indem wir rückwärtsfahren.

Wo aber bleiben unsere Gelenkparameter? Das Tolle an *PH_WHEEL* ist, dass diese per Default auf den richtigen Wert gesetzt werden, sodass wir uns um nichts kümmern müssen. Wollen wir diese dennoch anpassen, können wir das natürlich wie folgt tun.

Tabelle 12.6 Parameter für PH_WHEEL

Parameter	Verwendungszweck
1	Position des Gelenks in Weltkoordinaten
2	Achse des Gelenks, das für die Lenkbewegung nach links und rechts verantwortlich ist. *Default:* Das Rad wird, wie in der Realität, entlang der z-Achse gedreht `vector(0,0,-1)`.
3	Achse des Drehgelenks, um die das Rad rotiert. *Default:* Die Achse ist relativ zum Rad-Entity entlang der x-Achse ausgerichtet.
5	Hier werden die Reibungsfaktoren für das Rad abgebildet. Der x-Wert steht für die Längsreibung des Rades. Wenn wir diese erhöhen, uns hinter das Auto stellen und einen Ball dagegen schießen, wird das Auto sich entsprechend der Reibung weniger weit nach vorne bewegen. Senken wir die Längsreibung, dann wird es beim Fahren weniger Reibung zwischen Rad und Untergrund geben, und das Auto rollt weiter. Der y-Wert bildet die Querreibung ab. Diese greift, wenn wir uns neben das Auto stellen und einen Ball dagegen schießen. Das Handbuch sagt aus, dass gerade bei Schienenfahrzeugen die Querreibung besonders hoch sein sollte, da es ja logischerweise schwer ist, etwa einen Zug seitlich von den Schienen zu drücken. *Default:* `vector(100,100,0)`
6	Dieser Parameter bestimmt die Federung des Autos. Der x-Wert gibt die Federung der Radaufhängung an. Der y-Wert bestimmt die Stoßdämpferkonstante, also wie sehr die Reifen beim Herunterfallen auf den Grund abfedern. Der z-Wert gibt den möglichen Federweg in Quants an. *Default:* `vector(1000,100,Rad-Radius)`

Da es vorkommen kann, dass sich das Auto überschlägt, habe ich in der Demo eine Reset-Funktion eingebaut, mit der wir das Auto neu erstellen können. Auch diese möchte ich kurz erklären.

Listing 12.11 Zurücksetzen des Autos

```
// Das Auto kann neu erstellt werden, falls es umgefallen ist.
void resetCar() {
   pXcon_remove(entFrontLeftWheel);
   pXcon_remove(entFrontRightWheel);
   pXcon_remove(entBackLeftWheel);
   pXcon_remove(entBackRightWheel);
   pXent_settype(entCar, 0, 0);
   ptr_remove(entCar);
   ptr_remove(entBackRightWheel);
   ptr_remove(entBackLeftWheel);
   ptr_remove(entFrontRightWheel);
   ptr_remove(entFrontLeftWheel);
   entCar = NULL;
   entBackRightWheel = NULL;
   entBackLeftWheel = NULL;
   entFrontRightWheel = NULL;
   entFrontLeftWheel = NULL;
   createCar(vector(300, 200, 29), "buggy.mdl", "buggy_wheel.mdl");
}
```

Sie sehen, dass wir, bevor wir die Entities mit *ptr_remove* löschen, alle Gelenke entfernen und die Registrierung unsere Karosserie zurücknehmen. Damit räumen wir alles ordentlich auf, bevor wir unser Auto neu erstellen. Zur Sicherheit setzen wir die Entities hier auf *NULL*, damit wir auch ja nicht aus einer anderen Funktion auf sie zugreifen können und eine == *NULL-Abfrage* immer gültig ist. Zum Schluss erstellen wir unser Auto neu.

12.5.5 Seile mit PH_ROPE

Möchten Sie zwei Entities über ein unsichtbares Band miteinander verbinden, nutzen Sie *PH_ROPE*.

Listing 12.12 Erstellen eines Seiles mit PH_ROPE

```
// Erstellen von zwei Kugeln, die über ein Seil miteinander verbunden sind
void createRope(VECTOR* _offset, char* _model, int _distance) {
   ENTITY* entEnd1 = ent_create(_model, _offset, NULL);
   ENTITY* entEnd2 = ent_create(
      _model,
      vector(_offset.x+_distance, _offset.y, _offset.z),
      NULL
   );

   set(entEnd1, PASSABLE);
   set(entEnd2, PASSABLE);

   pXent_settype(entEnd1, PH_RIGID, PH_SPHERE);
   pXent_settype(entEnd2, PH_RIGID, PH_SPHERE);

   pXcon_add(PH_ROPE, entEnd1, entEnd2, 0);

   // Da die Verbindung nicht sichtbar ist, stellen wir das Seil durch eine
```

```
    // blaue Linie dar.
    while(1) {
        draw_line3d(entEnd1.x, NULL, 100);
        draw_line3d(entEnd2.x, vector(255,0,0), 100);
        wait(1);
    }
}
```

1. Wir erstellen zwei Kugeln in einem gewissen Abstand zueinander, machen diese passierbar und registrieren Sie in der Physik-Engine.
2. Im zweiten Schritt verbinden wir sie über ein unsichtbares Seil. Achten Sie darauf, dass Sie sie nicht an der Welt festmachen, damit können Sie sie frei durch das Level bewegen.
3. Mit der Funktion *draw_line3d* wird nun eine blaue Linie zwischen beiden Entities gezeichnet, um zu zeigen, wie diese miteinander verbunden sind, wo sich also unser unsichtbares Seil befindet.
4. Die Parameter benötigen wir hier nicht, da die Default-Werte wie beim Rad bereits stimmen.

Tabelle 12.7 Parameter für PH_ROPE

Parameter	Verwendungszweck
1	Ankerpunkt am ersten Entity in lokalen Koordinaten. Hier bestimmen wir, wo unser Seil am ersten Entity fest gemacht wird. *Default:* Nullpunkt.
2	Wie Parameter 1 nur für die zweite Entity
3	Schwellenwerte für Kraft und Drehung, um das Gelenk zu zerstören. *x* steht für Kraft, *y* für Drehung, z. B. `vector(25, 25, 0)`. Wenn Sie nicht möchten, dass das Gelenk bricht, übergeben Sie *NULL*.
6	Vektor für die Bewegungseinschränkung der Entities durch das Seil. Der x-Wert bestimmt die Federkonstante, wobei 0 ein wenig elastisches Seil darstellt. Der y-Wert bestimmt die Minimaldistanz, die zwischen den Entities zulässig ist. Der z-Wert dagegen setzt die Maximaldistanz.

12.5.6 Ein beliebig konfigurierbares Gelenk mit PH_6DJOINT

Kommen wir nun zum Allrounder der Gelenke, dem *PH_6DJOINT*. Dieses ist wohl am besten beschreibbar durch den Begriff *flexibel*, wir können damit also jedes Gelenk abbilden, das wir möchten. Warum gibt es dann noch die anderen? Nun, Sie werden sehen, dass die Verwendung von *PH_6DJOINT* nicht ganz einfach ist. Schauen wir uns ein Beispiel an, in dem wir eine Art Baumstamm einer Palme erstellen, den wir mit Bällen zum Federn bringen können. Weiterhin zeige ich, dass dieses Gelenk ebenfalls *motorisiert* werden kann, wie das *PH_WHEEL*, sich also drehen kann, wenn wir es so einrichten.

Listing 12.13 Ein dynamisches Gelenk mit PH_6DJOINT

```
// Wir erstellen ein 6D-Joint, mit dem sich alle Arten von Gelenken abbilden
// lassen.
void create6DJoint(VECTOR* _offset, char* _model) {

   // Wir erstellen einen Array für die Bewegungsmöglichkeiten des Gelenks
   var vMotion[6] = { 0,0,0,0,0,NX_D6JOINT_MOTION_FREE };

   // Dieser Array dient der Eigenrotation des Gelenks
   var vDrive[6] = { 0,0,0,0,0,NX_D6JOINT_DRIVE_VELOCITY };

   ENTITY* entPrevious = NULL;
   ENTITY* entTemp = NULL;
   int i;

   // Wir erstellen drei Entities, die wir miteinander verbinden
   for(i=0; i<3; i++) {
      entTemp = ent_create(
         _model,
         vector(_offset.x, _offset.y, _offset.z+i*34),
         NULL
      );

      entTemp.scale_z = 2;
      pXent_settype(entTemp, PH_RIGID, PH_SPHERE);
      pXcon_add(PH_6DJOINT, entTemp, entPrevious, 0);
      pXcon_set6djoint(entTemp, vMotion, vDrive);
      pXcon_setmotor(entTemp, vector(20,0,0), 0);
      entPrevious = entTemp;
   }
}
```

1. Zu Beginn erstellen wir zwei Arrays:
   a) **vMotion:** Die ersten drei Parameter (0–2) schränken die Bewegung in Richtung x, y und z relativ zum Achsengelenk ein. Der Default-Wert *NX_D6JOINT_MOTION_LOCKED* verbietet eine Bewegung in Richtung der Achse, *NX_D6JOINT_MOTION_LIMITED* beschränkt die Bewegung auf die über *pXent_setparams2* angegebenen Limits und *NX_D6JOINT_MOTION_FREE* erlaubt eine freie Bewegung der Achse. Die letzten drei Parameter (3–5) schränken die Winkelfreiheitsgrade ein.
   b) **vDrive:** Die ersten drei Parameter (0–2) definieren lineare Achsenmotoren. Die Parameter 3–5 definieren Drehmotoren um die drei Rotationsachsen, die übrigens in der Physik *swing*, *lerp* und *twist* heißen. Möchten Sie keinen Motor auf der Achse oder dem Winkel positionieren, übergeben Sie eine 0. Möchten Sie einen Motor setzen, der sich auf ein Ziel zubewegt, dann nutzen Sie *NX_D6JOINT_DRIVE_POSITION*, und für einen Motor, der sich lediglich mit einer Zielgeschwindigkeit bewegt, wird *NX_D6JOINT_DRIVE_VELOCITY* eingesetzt.
2. Wir erstellen nun drei langgezogene Bälle, die wir übereinander anordnen. Mit *scale_x*, *scale_y* und *scale_z* können wir die Größe der Entities zur Laufzeit ändern. Wenn Sie einen Blick in die *actBall* werfen, sehen Sie, dass wir auch hier, anstatt den Ball direkt zu entfernen, selbigen vorher schrumpfen lassen. Das sorgt für einen netten visuellen Effekt.

3. In der Schleife registrieren wir nun jede Entity in der Physik-Engine und hängen sie in der Zeile darauf per *PH_6DJOINT* an die vorige Entity.
4. Nun weisen wir ihr über **pXcon_set6djoint** unsere vordefinierten Parameter in Form der beiden Arrays zu …
5. … und lassen sie mit **pXcon_setmotor** mit einer Geschwindigkeit von 20 Einheiten pro Frame um die im zweiten Array definierten Achse rotieren.
6. Schließlich setzen wir noch unsere letzte Entity auf die Aktuelle, damit wir die nächste Entity an die davor anbauen können.

Ich möchte Sie hiermit einmal mehr ausdrücklich dazu ermutigen, so viel wie möglich mit den Werten der Joints zu spielen, um herauszufinden, wie sie am besten zu benutzen sind. Physik ist und bleibt eine Frickelei, bei der man ein bisschen experimentieren muss bis alles so läuft, wie man es gerne hätte.

Wir spielen hier mit den *scale_x..z*-Eigenschaften von Entities und setzen diese einzeln. Wie jedoch viele andere Werte, die x-, y- und z-Komponenten besitzen, können wir diese per *vec_set* verändern. Um also eine Entity um den Wert 2 in alle Richtungen zu vergrößern, lässt sich auch `vec_set(my.scale_x,vector(my.scale_x*2,my.scale_y*2,My.scale_z*2));` verwenden.

12.6 Physikalische Spielersteuerung

Bisher konnte unser Spieler nicht mit den Physik-Objekten, die wir erstellt haben, kollidieren, da er selbst nicht Teil des physikalischen Geschehens war. Das wollen wir nun ändern. In der folgenden Demo laufen wir durch eine kleine Szene, in der wir Physik-Objekte anstoßen und diese sogar durch Drücken und Halten der linken Maustaste hin und her tragen können.

Beispiel 27

Ordner: „27_physikalische_bewegung“

Schwierigkeitsgrad: Medium

Demonstriert:

- Player physikalisch bewegen
- Tragen und Absetzen von Entities

Wir beginnen bei der Aktion unseres Spielers.

Listing 12.14 Aktion für eine physikalische Spielersteuerung

```
// Spieler kann springen
void playerJump() {

   // Brich alle die Funktionen ab, die noch laufen
   proc_kill(5);
   player.JUMP_HEIGHT = 100;
}

// Aktion für einen Spieler, der physikalische Objekte beeinflussen kann
action actPlayerPhysX() {
   player = my;
   c_setminmax(my);
   set(me, INVISIBLE);

   // Registriere den Spieler in der Physik-Engine
   pXent_settype(my,PH_CHAR,PH_CAPSULE);
   VECTOR vecPlayerMoveSpeed;
   ANGLE angPlayerRotation;
   wait(1);

   // Solange der Spieler existiert...
   while(player)
   {
      vecPlayerMoveSpeed.x = (key_w - key_s) *
          (WALK_SPEED+key_shiftl*RUN_SPEED) * time_step;
      vecPlayerMoveSpeed.y = (key_a - key_d) * WALK_SPEED * time_step;

      // Suche nach Hindernissen des Spielers
      vTraceDown = c_trace(
         player.x,
         vector(player.x, player.y, player.z-500),
         IGNORE_ME | IGNORE_PASSENTS | IGNORE_PASSABLE | USE_BOX
      );

      // Ist der Spieler noch dabei zu springen, lass ihn weiter springen
      if (player.JUMP_HEIGHT > 0) {
         vecPlayerMoveSpeed.z = time_step*20;
         player.JUMP_HEIGHT -=1;
         on_space = NULL;
      } else {
         // Springt der Spieler gerade nicht, und ist er noch in der Luft...
         if (vTraceDown > 20) {
            // ... lass ihn fallen
            vecPlayerMoveSpeed.z = -time_step*20;
            on_space = NULL;
         } else {
            // Ist der Spieler auf dem Boden, lass ihn nicht mehr
            // fallen...
            vecPlayerMoveSpeed.z = 0;
            // ... und erlaube ihm wieder zu springen
            on_space = playerJump;
         }
      }
      angPlayerRotation.pan -= mouse_force.x * time_step*MOUSE_SENSITIVITY;

      // Setze die Position des Spielers (pXent_move für ein Physik-Objekt)
```

```
        pXent_move(me, vecPlayerMoveSpeed, nullvector);
        // Rotiere den Spieler
        pXent_rotate(me, angPlayerRotation, nullvector);

        moveCamera();
        wait(1);
    }
}
```

1. Wir registrieren den Spieler über *pXent_settype* als *PH_CHAR* und sagen der Physik-Engine damit, dass unser Entity nicht von Kräften beeinflusst werden darf, aber dennoch Kollisionen erkennt und auch auslöst. Wir können dieses Entity nur noch über die zwei Funktionen **pXent_move** und **pXent_rotate** bewegen, wie wir später sehen werden.
2. In einer While-Schleife, die so lange läuft wie *player* ungleich *NULL* ist, steuern wir nun unsere Entity mit den WASD-Tasten. Die Bewegungsrichtungen speichern wir im Vektor *vecPlayerMoveSpeed* zwischen.
3. Mit *c_trace* schauen wir, ob wir festen Boden unter den Füßen haben oder ob wir in der Luft hängen.
4. Nun kommen wir zu einer neuen Funktion und zwar zum Springen. *JUMP_HEIGHT* ist per Define auf *skill1* gemappt und wird beim Aufruf der Funktion *playerJump* auf 100 gesetzt. Ist *JUMP_HEIGHT* größer als 0, dann bewegen wir den Spieler in z-Richtung nach oben und verringern *JUMP_HEIGHT*, um nicht zu lange zu springen. Mit `on_space = NULL;` verbieten wir dem Spieler, mit der Leertaste zu springen, da wir ja gerade mitten in einem Sprung sind.
5. Springen wir nicht (*JUMP_HEIGHT* gleich 0), dann lassen wir den Spieler fallen, sofern er noch höher als 20 Quant über dem Erdboden oder der nächsten Entity ist. Solange wir fallen, darf der Spieler auch nicht springen.
6. Befindet er sich auf dem Erdboden, dann erlauben wir dem Spieler wieder zu springen, indem wir der *Leertaste* die Funktion *playerJump* zuweisen.
7. Nach der If-Abfrage berechnen wir nun die Drehung des Spielers und speichern sie im Winkel *vecPlayerRotation* zwischen.
8. Nun führen wir mit *pXent_move* und *pXent_rotate* unsere Spielerbewegung durch. Beachten Sie, dass wir uns hier relativ zur derzeitigen Position des Spielers drehen und deswegen den ersten Parameter der beiden Funktionen setzen. Würden wir eine absolute Position angeben wollen, würden wir den zweiten Parameter nutzen und den Ersten auf *nullvector* setzen.
9. Zuletzt bewegen wir wie immer die Kamera.

Die Funktion **proc_kill** in *playerJump* sorgt dafür, dass alle anderen Aufrufe der Funktion *playerJump*, die derzeit noch aktiv sind, beendet werden. Damit gehen wir sicher, dass wir es nicht doch irgendwie schaffen, mehrmals zu springen.

Die Spielerbewegung ist also gar nicht so verschieden von der bisherigen, nur dass wir statt *ent_move* die Funktion *pXent_move* nutzen. Schauen wir uns nun noch den Code zum Tragen und Fallenlassen der Kisten an.

Listing 12.15 Tragen von Entities

```
// Funktion, um Physik-Objekte zu tragen.
void grabObjects() {

   ENTITY* entTemp = NULL;

   while(1) {
      // Solange wir die linke Maustaste gedrückt haben, wollen wir Objekte
      // tragen
      if (mouse_left) {

         // Wenn wir bereits ein Objekt halten, aktualisieren wir dessen
         // Position
         if (entTemp != NULL) {
            vec_set(entTemp.x, vector(
                  100*cos(camera.pan)+player.x,
                  100*sin(camera.pan)+player.y,
                  player.z+20)
            );

            vec_set(entTemp.pan, camera.pan);

         // Wenn wir noch kein Objekt halten...
         } else {

            // ... aber sich unter unserem Fadenkreuz eines befindet,
            // dann nehmen wir es uns und lösen es zum Tragen von
            // der Physik-Engine.
            if (mouse_ent != NULL) {
               entTemp = mouse_ent;
               set(entTemp, TRANSLUCENT);
               pXent_settype(entTemp, 0, 0);
            }
         }

      // Wenn wir die linke Maustaste nicht (mehr) drücken und ein Objekt
      // in den Händen halten, dann lassen wir es fallen und machen es
      // wieder zu einem Physik-Objekt.
      } else {
         if (entTemp != NULL) {
            reset(entTemp, TRANSLUCENT);
            pXent_settype(entTemp,PH_RIGID, PH_BOX);
            entTemp = NULL;
         }
      }

      wait(1);
   }
}
```

1. In einer While-Schleife prüfen wir, ob die linke Maustaste gedrückt ist. Wenn ja, schauen wir nach, ob wir bereits ein Objekt tragen und setzen dessen neue Position, falls dem so ist. Tragen wir noch kein Objekt, setzen wir den Pointer *entTemp* auf die Entity **mouse_ent,** welche immer das aktuelle Objekt enthält, auf das unser Mauszeiger gerade zeigt. Unser Mauszeiger sitzt genau unter dem Fadenkreuz in der Mitte des Bildschirms. Diese Position setzen wir in *initGui*.

2. Man beachte, dass wenn wir ein neues Objekt greifen, wir dieses aus der Physik-Engine herausnehmen, damit wir es durch das Level tragen können, ohne andere Entities zu beeinflussen. Außerdem machen wir es über `set(entTemp, TRANSLUCENT);` durchsichtig, damit wir noch sehen, wo wir hinlaufen, wenn wir den großen Block tragen.
3. Das *else* gegen Ende wird aufgerufen, wenn wir die linke Maustaste loslassen. Haben wir ein Objekt getragen (`entTemp != NULL`), dann nehmen wir ihm die Transparenz und registrieren es wieder in der Physik-Engine. Damit wir ein neues Objekt greifen können, setzen wir *entTemp* wieder auf *NULL*.

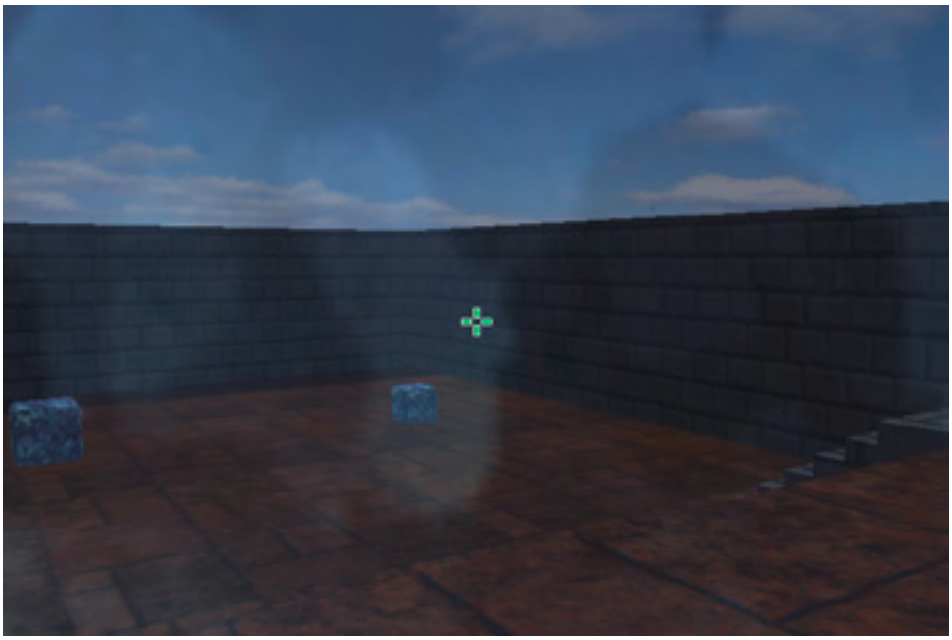


Bild 12.7 Ein Block wird vom Spieler durch das Level getragen.

Mit dieser einfachen Funktion können wir später ein paar tolle Rätsel bauen, in denen man beispielsweise eine Wippe beschweren oder sich selber eine Treppe bauen muss.

12.7 Ragdolls

Ragdolls lassen sich im funktionalen Sinne wahrscheinlich am ehesten als *Marionetten* erklären. Sie dienen dazu, einen unbelebten Körper mit samt seinen Körperteilen realistisch fallen zu lassen. Dabei wird vermieden, dass unschöne Effekte entstehen, etwa dass eine Entity schief auf einem Terrain liegen bleibt oder in eine Wand hineinrutscht. Rufen Sie sich das Seil aus dem Beispiel in Abschnitt 12.5.5 ins Gedächtnis. Dort wurden auch einzelne Glieder über Gelenke miteinander verknüpft.

Bild 12.8 Eine Ragdoll wird von einem Ball getroffen

Ich habe lange drüber nachgedacht, wie komplex ich diesen Abschnitt gestalten soll und habe mich für den einfachen Weg entschieden, der sich auf das Wesentliche konzentriert. Wir werden einen Körper konstruieren, diesen über Gelenke miteinander verbinden und ihn einer Kollision mit einem Ball aussetzen. Lassen Sie uns schnell schauen, wie man so etwas bewerkstelligt.

Beispiel 28

Ordner: „28_ragdolls“

Schwierigkeitsgrad: Schwer

Demonstriert: Verknüpfen von Entities zu einem Körper

Zuerst benötigen wir eine Funktion, die uns einen Ball werfen lässt. Wir wollen diesmal die Kraft des Wurfes bestimmen, indem wir die Leertaste mal länger mal kürzer gedrückt halten.

Listing 12.16 Werfen eines Balles

```
// Kraft, mit der wir einen Ball werfen
int nForce = 0;

// Panel, das die Kraft des zu werfenden Balles anzeigt
PANEL* panForce = {
   pos_x = 0;
   size_x = 0;
   size_y = 10;
   flags = LIGHT | SHOW;
```

```
   red = 255;
   green = 0;
   blue = 0;
}

// Aktion eines geworfenen Balles
action actBall() {
   wait(-5);
   pXent_settype(me, 0, 0);
   while(my.scale_x > 0) {
      my.scale_x -=0.1;
      my.scale_y -=0.1;
      my.scale_z -=0.1;
      wait(7);
   }
   ptr_remove(me);
}


// Funktion, um einen Ball zu werfen
void throwBall(int _force) {
   ENTITY* entBall = ent_create(
      "ball.mdl",
      vector(camera.x, camera.y, camera.z),
      actBall
   );

   VECTOR vecBallForce;

   // Die Kraft auf den Ball zeigt in Blickrichtung der Kamera
   vec_for_angle(vecBallForce.x, camera.pan);

   // ... und wird entsprechend der Aufladedauer verstärkt
   vec_scale(vecBallForce.x, _force);

   pXent_settype (entBall, PH_RIGID, PH_SPHERE);
   pXent_addforcelocal (entBall, vecBallForce, entBall.x);
}

// Aufladen des Balles
void ballCannon() {
   int nWasLoading = 0;
   while(1) {

      if (key_space && (nForce < 100)) {
         nForce +=1;
         nWasLoading = 1;
      } else {
         if (nWasLoading) {
            throwBall(nForce);
            nForce = 0;
            nWasLoading = 0;
         }
      }
      panForce.size_x = (screen_size.x / 100)*nForce;
      wait(8);
   }
}
```

1. Zu Beginn erstellen wir ein Panel, das rot illuminiert sein soll und die Breite 0 besitzt.
2. Schauen wir nun in die Methode *ballCannon*, sehen wir, dass, wenn die Leertaste gedrückt wurde, eine Variable namens *nForce* hochgezählt wird. Beim Loslassen der Leertaste wird die Funktion *trowBall* aufgerufen, die dann den Ballwurf initiiert.
3. Am Ende von *ballCannon* ist zu sehen, dass die Breite des Panels abhängig von *nForce* ist, sodass das Panel breiter wird je länger wir die Leertaste gedrückt halten.
4. *throwBall* ist nun wieder einfach zu verstehen. Es wird ein Ball erzeugt, der in der Physik-Engine registriert wird. Nun berechnen wir einen Vektor, der in die Richtung der Kamera zeigt. Diesen skalieren wir entsprechend der Kraft *nForce* und lassen ihn auf den Ball einwirken, sodass dieser in die Blickrichtung unserer Kamera fliegt.
5. Die dem Ball zugewiesene Aktion *actBall* löscht diesen nach 5 Sekunden.

Kommen wir nun zum Teil, in dem wir unsere Ragdoll erzeugen, die mit den gerade erzeugten Bällen beworfen werden kann. Beachten Sie den Aufruf von *createRagdoll* in der *main*. Dort erzeugen wir an der angegebenen Stelle eine Ragdoll mit der ID 2.

Listing 12.17 Erzeugen einer minimal komplexen Ragdoll

```
// Eine neue Ragdoll wird erzeugt
ragdoll_t* createRagdoll(VECTOR* _offset, int _group) {

   // Reservieren des Speichers für die Ragdoll
   ragdoll_t* ragdoll = sys_malloc(sizeof(ragdoll_t));

   VECTOR *vecTemp  = nullvector;
   VECTOR *vecTemp2 = nullvector;
   VECTOR *vecTemp3 = nullvector;

   ragdoll.offset = _offset;

   // Erstellung der Körperteile
   ragdoll.entTorso = ent_create("Torso.mdl", _offset, NULL);
   ragdoll.entHead = ent_create("Head.mdl", nullvector, NULL);
   ragdoll.entLeftArm = ent_create("LeftArm.mdl", nullvector, NULL);
   ragdoll.entRightArm = ent_create("RightArm.mdl", nullvector, NULL);

   ragdoll.entLeftLeg = ent_create("Leg.mdl", nullvector, NULL);
   ragdoll.entRightLeg = ent_create("Leg.mdl", nullvector, NULL);

   ragdoll.entLeftFoot = ent_create("Foot.mdl", nullvector, NULL);
   ragdoll.entRightFoot = ent_create("Foot.mdl", nullvector, NULL);

   // Körperteile werden an den Vertices des Torsos positioniert
   vec_for_vertex(vecTemp, ragdoll.entTorso, 27);
   vec_set(ragdoll.entHead.x, vecTemp.x);

   vec_for_vertex(vecTemp, ragdoll.entTorso, 41);
   vec_set(ragdoll.entRightArm.x, vecTemp.x);

   vec_for_vertex(vecTemp, ragdoll.entTorso, 55);
   vec_set(ragdoll.entLeftArm.x, vecTemp.x);

   vec_for_vertex(vecTemp, ragdoll.entTorso, 69);
   vec_set(ragdoll.entLeftLeg.x, vecTemp.x);
```

```
vec_for_vertex(vecTemp, ragdoll.entTorso, 83);
vec_set(ragdoll.entRightLeg.x, vecTemp.x);

vec_for_vertex(vecTemp, ragdoll.entRightLeg, 26);
vec_set(ragdoll.entRightFoot.x, vecTemp.x);

vec_for_vertex(vecTemp, ragdoll.entLeftLeg, 26);
vec_set(ragdoll.entLeftFoot.x, vecTemp.x);


// Registrierung der Körperteile in der Physik-Engine
// und gruppieren aller Teile
makeBodyPartPhysical(ragdoll.entTorso, _group);
makeBodyPartPhysical(ragdoll.entHead, _group);
makeBodyPartPhysical(ragdoll.entLeftArm, _group);
makeBodyPartPhysical(ragdoll.entRightArm, _group);
makeBodyPartPhysical(ragdoll.entLeftLeg, _group);
makeBodyPartPhysical(ragdoll.entRightLeg, _group);
makeBodyPartPhysical(ragdoll.entRightFoot, _group);
makeBodyPartPhysical(ragdoll.entLeftFoot, _group);

// Gelenke erstellen
// Nackengelenk
vec_for_vertex(vecTemp, ragdoll.entHead, 62);
vec_for_vertex(vecTemp2, ragdoll.entTorso, 27);
vecTemp3 = vecMiddle(vecTemp, vecTemp2);
connectViaHinge(
   ragdoll.entHead,
   ragdoll.entTorso,
   vecTemp3,
   vector(1,1,1),
   vector(-90,90,0)
);

// Linker Arm
vec_for_vertex(vecTemp, ragdoll.entLeftArm, 25);
vec_for_vertex(vecTemp2, ragdoll.entTorso, 55);
vecTemp3 = vecMiddle(vecTemp, vecTemp2);
connectViaHinge(
   ragdoll.entLeftArm,
   ragdoll.entTorso,
   vecTemp3,
   vector(1,1,1),
   vector(-90,90,0)
);

// Rechter Arm
vec_for_vertex(vecTemp, ragdoll.entRightArm, 26);
vec_for_vertex(vecTemp2, ragdoll.entTorso, 41);
vecTemp3 = vecMiddle(vecTemp, vecTemp2);
connectViaHinge(
   ragdoll.entRightArm,
   ragdoll.entTorso,
   vecTemp3,
   vector(1,1,1),
   vector(-90,90,0)
);
```

```
   // Linkes Bein
   vec_for_vertex(vecTemp, ragdoll.entLeftLeg, 25);
   vec_for_vertex(vecTemp2, ragdoll.entTorso, 69);
   vecTemp3 = vecMiddle(vecTemp, vecTemp2);
   connectViaHinge(
      ragdoll.entLeftLeg,
      ragdoll.entTorso,
      vecTemp3,
      vector(1,1,0),
      vector(-60,60,0)
   );

   // Rechtes Bein
   vec_for_vertex(vecTemp, ragdoll.entRightLeg, 25);
   vec_for_vertex(vecTemp2, ragdoll.entTorso, 83);
   vecTemp3 = vecMiddle(vecTemp, vecTemp2);
   connectViaHinge(
      ragdoll.entRightLeg,
      ragdoll.entTorso,
      vecTemp3,
      vector(1,1,0),
      vector(-40,40,0)
   );

   // Linker Fuß
   vec_for_vertex(vecTemp, ragdoll.entLeftFoot, 25);
   vec_for_vertex(vecTemp2, ragdoll.entLeftLeg, 26);
   vecTemp3 = vecMiddle(vecTemp, vecTemp2);
   connectViaHinge(
      ragdoll.entLeftFoot,
      ragdoll.entLeftLeg,
      vecTemp3,
      vector(1,0,0),
      vector(-20,20,0)
   );

   // Rechter Fuß
   vec_for_vertex(vecTemp, ragdoll.entRightFoot, 25);
   vec_for_vertex(vecTemp2, ragdoll.entRightLeg, 26);
   vecTemp3 = vecMiddle(vecTemp, vecTemp2);
   connectViaHinge(
      ragdoll.entRightFoot,
      ragdoll.entRightLeg,
      vecTemp3,
      vector(1,0,0),
      vector(-20,20,0)
   );

   return ragdoll;
}
```

Schnell ist in *createRagdoll* zu sehen, dass selbst ein leichtes Beispiel wie das unsere schon eine Menge Schreibarbeit erfordert.

1. Ein *Struct* mit dem Namen *ragdoll_t* beinhaltet alle Körperteile, aus denen die Ragdoll besteht sowie ihre Position in *offset*.

2. In *createRagdoll* reservieren wir zu Beginn Speicher für ein solches Struct und beginnen gleich darauf alle Körperteile zu erzeugen.
3. Darauf folgend werden diese Körperteile richtig positioniert, sodass beispielsweise am oberen Ende des Torsos der Kopf angebracht wird und so weiter. Dazu nutzen wir die Funktion **vec_for_vertex,** die die Position des jeweiligen Vertex ausliest, etwa die der Spitze des Halses, und dort den Kopf platziert. Die Nummer der Vertices eines Modells können sie im MED auslesen, indem Sie die einzelnen Vertices, die als kleine gelbe Punkte dargestellt werden, selektieren und einen Blick in die Statusleiste unten links werfen (Bild 12.9).
4. Sind alle Körperteile an der richtigen Stelle, so werden diese über die Hilfsfunktion *makeBodyPartPhysical* in der Physik-Engine registriert und einer Gruppe zugewiesen, sodass wir eine Kollision untereinander vermeiden. Dazu dient die ID, die der Create-Methode mitgegeben wird.
5. Nun kommt der Teil, an dem wir die Gliedmaßen miteinander verknüpfen. Dazu berechnen wir jedes Mal für beide Körperteile, die wir verbinden möchten, den Mittelpunkt der nahesten Vertices und setzen diesen als Mittelpunkt eines Scharniergelenks. Dazu dient die Funktion *vecMiddle*.
6. Haben wir alle Körperteile verbunden, sind wir fertig!

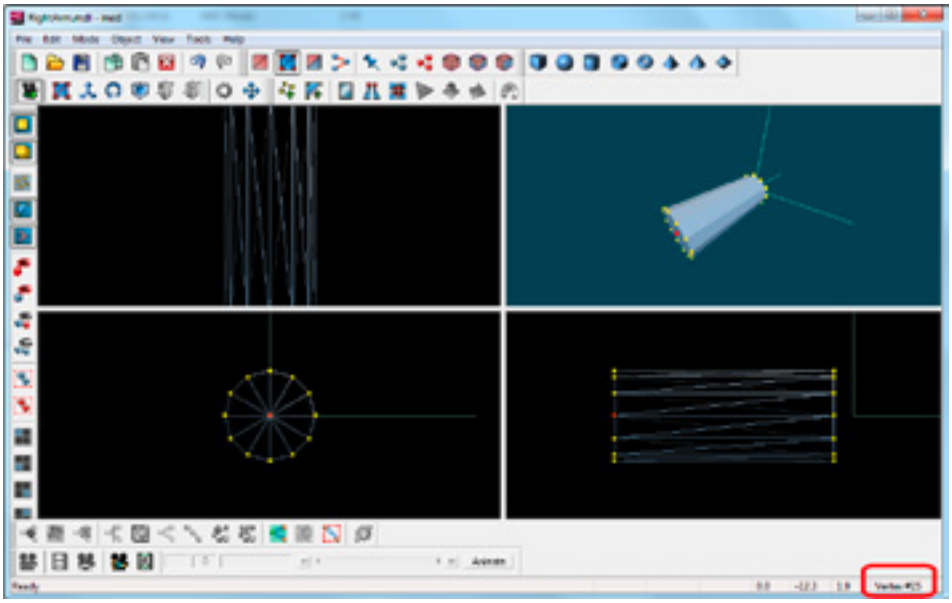


Bild 12.9 Auslesen einer Vertexnummer im MED

Bedenken Sie: Je mehr Körperteile Sie einer *Ragdoll* zuweisen, desto mehr Gelenke müssen Sie erstellen. Wenn Sie nun das Spiel ausführen, können Sie sehen, dass die Marionette, die wir erstellt haben, realistisch fortgeschleudert wird, wenn der Ball mit ihr kollidiert.

Als logische Konsequenz der bisher implementierten Funktionen würde nun eine weitere folgen, die die Körperteile der Ragdoll mit den Bones eines normalen Modells überblendet. Dazu finden Sie einige gute Beispiele im Gamestudio-Monatsmagazin[1] (AUM 92) oder in der Community. Da dieses Beispiel dann aber mehrere Seiten Umfang in Anspruch nehmen würde, habe ich mich entschieden, es nicht in die Demo aufzunehmen.

12.8 PhysX Community Edition

Hier möchte ich Sie noch auf ein Physik-Plug-in aufmerksam machen, das ein Mitglied der Gamestudio-Community, Christian Behrenberg[2], auch bekannt als *HeelX*, entwickelt hat und das mittlerweile von einigen weiteren Leuten stetig verbessert wird. Es basiert auf der quelloffenen Implementierung des offiziellen PhysiX-Plug-ins von A8.

Heruntergeladen werden kann das Plug-in bei *SourceForge* unter dem Namen *a8physx*[3].

Weiterhin gibt es schon einige interessante Ansätze zur Darstellung von Flüssigkeiten (*Fluids*), Stoffen (*Cloth*) und weichen Körpern (*Soft bodies*), die von Robert Judycki, auch bekannt als *Rojart*, implementiert wurden.

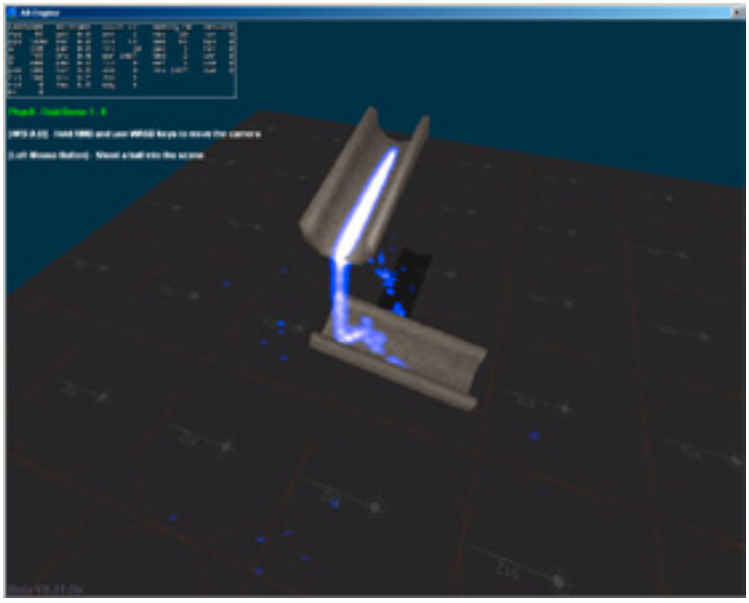


Bild 12.10 Demonstration von Flüssigkeiten, von Robert Judycki, *gamefactor.eu*

[1] http://www.coniserver.net/coni_users/web_users/pirvu/aum/aumonline_e
[2] http://www.christian-behrenberg.de
[3] sourceforge.net/projects/a8physx