UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Autorensystem zum Erstellen echtzeitfähiger physikalisch simulierter 3D Szenen

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Dominik Ospelt

Erstgutachter: Prof. Dr. Stefan Müller
Arbeitsgruppe Computergrafik, Institut für Computervisualistik

Zweitgutachter: Dipl. Inf. Stefan Rilling
Arbeitsgruppe Computergrafik, Institut für Computervisualistik

Koblenz, im November 2009

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)

Zusammenfassung

Diese Bachelorarbeit befasst sich mit der Entwicklung eines Autorensystems zum modellieren von 3D Szenen mit physikalischer Beschreibung. Ähnlich einem herkömmlichen 3D Modellierungstool soll ein Benutzer Szenen erstellen können mit dem Unterschied, dass bei der Erstellung der Geometrie physikalische Eigenschaften direkt berechnet und eingestellt werden können. Wichtig für solche Systeme ist vor allem ihre Erweiterbarkeit und Anpassungsfähigkeit an die entsprechenden Anforderungen des Benutzers. Der Fokus liegt hierbei auf der Entwicklung einer einfachen Architektur, die leicht erweiterbar und veränderbar ist.

Abstract

This thesis deals with the development of an authoring system for modeling 3D environments with physical description. In contrast to creating scenes in other common modeling tools, one can now compute and describe physical entities of a scene additional to the usual geometry. It is very important for those authoring systems to be extendable and customizable for specific requirement of the user. The focus lies on developing simple program architecture, which is easy to extend and to modify.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1

1.2 Stand der Technik 1

1.3 Anforderungen 2

2 Grundlagen der Physiksimulation **3**

2.1 Gesetzte von Newton 3

2.2 Einwirkende Kräfte 4

2.3 Physikengines 7

3 Umsetzung **7**

3.1 verwendete Bibliotheken 7

3.1.1 OGRE - Object-Oriented Graphics Rendering Engine SDK 8

3.1.2 Nvidia PhysX SDK 8

3.1.3 Trolltech Qt SDK 8

3.2 Architektur 9

3.2.1 Datenstrukturen 9

3.2.2 Aufbau des GUI 11

3.2.3 Qt SIGNAL/SLOT Prinzip 12

3.2.4 Verwendete QWidgets 13

3.2.5 Mouse Controller 16

3.3 Daten Im-/Export 18

4 Ergebnis **19**

4.1 Workflow 19

4.2 Autorenprozesses am Beispiel (eines Abrisskrans) 21

4.3 Ausblick 24

4.4 Fazit 25

1 Einleitung

1.1 Motivation

Physikengines finden heute bereits in vielen Spielen, aber auch in Wissenschaftlichen Simulationen Verwendung. Jedoch ist das Beschreiben komplexer physikalischer Szenen im Code sehr aufwändig und nicht immer intuitiv. Abgesehen davon können Änderungen an der Physikszene dann nur noch direkt am Code vorgenommen werden. Es entsteht der Bedarf an einem Tool, mit dem Entwickler ohne große Vorkenntnisse und möglichst intuitiv eigene 3D Szenen erstellen können, um diese später in ihre Applikation zu integrieren. Der Autorenprozess sollte ähnlich verlaufen wie beim Erstellen von 3D Modellen, allerdings liegt der Fokus hier auf der physikalische Beschreibung.

Ich habe für meine Bachelorarbeit das Thema dieses Autorensystems gewählt, um einen Einblick in die Architektur komplexerer Programme zu bekommen. Das Verbinden der Physiksimulation mit einer Benutzeroberfläche zum Gestalten und Verändern der Physikszene in Echtzeit steht hierbei im Vordergrund.

1.2 Stand der Technik

Moderne Modellierungstools wie '3DStudioMax', 'Maya' und das kostenlose Tool 'Blender' unterstützen bereits Physikengines, mit denen 3D Szenen modelliert und animiert werden. Allerdings werden hier die Physikszenen nicht im Echtzeitkontext verwendet, sondern im Renderingprozess bei der Berechnung von Animationen. Es gibt zwar Möglichkeiten zur Vorschau, jedoch muss die physikalisch berechnete Animationssequenz bei Änderungen oft erst neu Berechnet werden.

Es existieren auch Echtzeitfähige Programme wie die 'Phun 2D Physics Sandbox' von Emil Ernerfeldt, ein Autorensystem, welches einfaches und spielerisches erstellen von Physikszenen im 2D möglich macht. Allerdings ist die Physikbeschreibung, die von diesem Programm erstellt wird, nur Zweidimensional und kann somit nicht direkt in 3D-Anwendungen integriert werden.

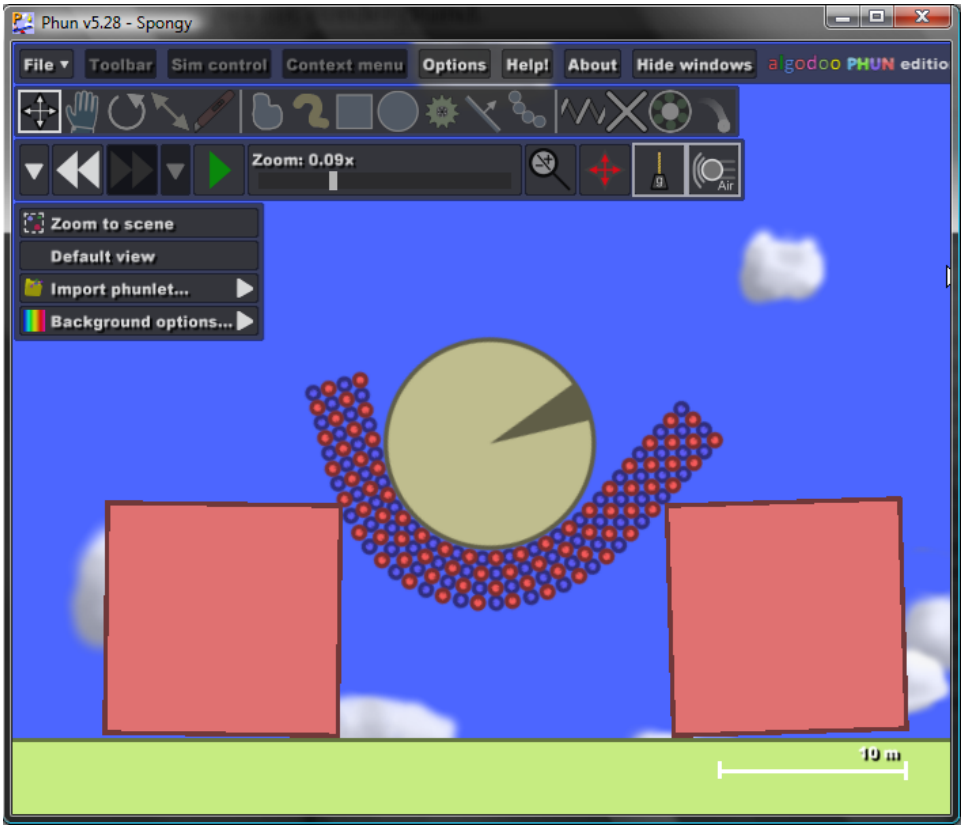

Abbildung 1: Phun 2D - Emil Ernerfeld

1.3 Anforderungen

Die Anwendung soll dem Benutzer das Erstellen von einfachen Objekten im 3D sowie deren physikalische Beschreibung möglichst einfach machen. Dazu wird ein leicht zu verstehendes Userinterface Implementiert, das schnell zu guten Ergebnissen führen soll. Mithilfe des Autorensystems soll der Benutzer in der Lage sein, einfache geometrische Grundkörper zu erstellen, diese zu kombinieren und so komplexere Objekte zu erschaffen. Zusätzlich kann der der Autor die physikalischen Eigenschaften wie die Masse des Objekts verändern. Die so erstellten Objekte sollen dann durch Verschiedene Gelenke miteinander verknüpft werden.

Der Benutzer soll die Szene jederzeit simulieren können, seine Änderungen sollen immer direkt sichtbar sein. Es soll jederzeit die Möglichkeit bestehen die aktuelle Physikszene zu speichern, bzw. eine bereits gespeicherte Szene zu laden. Nach Abschluss des Autorenprozesses soll die Möglichkeit bestehen, in ein menschenlesbares Format exportieren zu können, um die

erstellte Physikszene in anderen Programmen weiter verwenden zu können.

2 Grundlagen der Physiksimulation

2.1 Gesetzte von Newton

Die Gesetze von Newton spielen eine Wichtige Rolle bei der physikalischen Simulation. Nahezu alle weiteren Gesetzmäßigkeiten beruhen auf diese Regeln.

1.Gesetz: Trägheit

Wirken keine Kräfte auf einen Körper in Ruhe, so bleibt er in Ruhe. Wenn sich ein Körper bewegt, und keine Kräfte von außen darauf einwirken, so bleibt er mit Konstanter Geschwindigkeit und Richtung in Bewegung.

2.Gesetz: Aktionsprinzip

Hat ein Körper eine gleichbleibende Masse m bezüglich der Zeit, so ist seine Beschleunigung a Proportional zur einwirkenden Kraft F und umgekehrt proportional zur Masse m des Körpers.

$F = m \cdot a$

Die Änderung der Bewegung geschieht nach der Richtung der einwirkenden Kraft. Ändert sich die Masse des Körpers über die Zeit t, so gilt:

$F = \frac{d}{dt} \cdot (mv) = ma + \frac{dm}{dt} \cdot v$

wobei v die Geschwindigkeit des Körpers ist.

3.Gesetz: Reaktionsprinzip

Wird eine Kraft von einem Körper A auf einen Körper B ausgeübt, so wirkt eine gleichgroße, entgegengesetzte Kraft von Körper B auf Körper A.

2.2 Einwirkende Kräfte

Gravitationskräfte

Betrachtet man zwei Massen m und M, so ziehen sie sich gegenseitig mit gleichstarker, aber entgegengesetzter Kraft an.

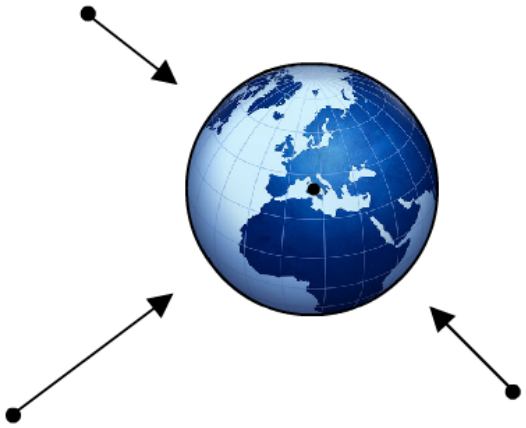

Abbildung 2: Schwerkraft

$F_{gravity} = \frac{GmM}{r^2}$

wobei r der Abstand der beiden Punktmassen ist und $G = 6.67 \cdot 10^{-}11 \frac{Nm^2}{kg^2}$ die universelle Gravitationskonstante. Für Simulationen auf der Erdoberfläche vereinfachen wir die Formel durch die Annahme einer planaren Oberfläche zu

$F = -mgU$

wobei $g = 9.81 \frac{m}{s^2}$ und U der nach oben zeigende Einheitsvektor.

Haft-/Gleitreibung

Reibungskräfte wirken, wenn ein Körper an der Oberfläche eines anderen Körpers bewegt werden soll. Die Reibungskraft wirkt entlang der Tangente der sich berührenden Flächen. Es wird angenommen, dass die Reibung unabhängig von der berührenden Fläche und, ist der Körper einmal in Bewegung, unabhängig von der Geschwindigkeit ist.

$$F = \begin{cases} -c_k \cdot \frac{v}{|v|} & \text{für} \quad v \neq 0 \\ 0 & \text{für} \quad v = 0 \end{cases}$$

c_k ist hierbei der Gleitreibungskoeffizient, er gibt das Verhältnis zwischen Reibung und normaler Kraft an

$$c_k = F_{reibung}/F_{normal}$$

Wegen der Division durch 0 bei Nullkräften wird zwischen Haft- und Gleitreibung unterschieden. Befindet sich ein Körper auf einer Oberfläche und bewegt sich nicht, so wird eine Kraft ausgeübt ihn in Bewegung zu versetzen. Ist diese Kraft nicht größer als die Haftreibung, so bewegt sich der Körper nicht. Wenn die Kraft jedoch die Haftreibung überschreitet, dann treten die Gesetzmäßigkeiten für die Gleitreibung in Kraft, und der Körper beginnt sich zu bewegen. Solange wie sich der Körper bewegt, wird die Gleitreibung angewendet.[Ebe04]

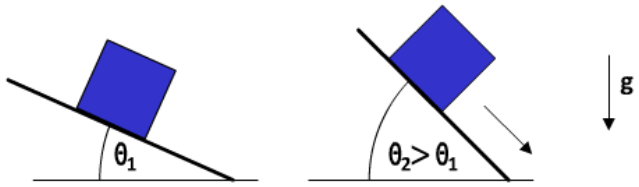

Abbildung 3: Haft- und Gleitreibung

Drehmoment

Die Kraft, die über einen Hebel auf einen Körper wirkt, wird als Drehmoment bezeichnet. Das Drehmoment verhält sich proportional zur Länge des Hebels und zur ausgeübten Kraft auf das Ende des Hebels. Das Drehmoment berechnet sich fogendermassen

$$\tau = r \times F$$

Der Punkt, auf den das Drehmoment wirkt, befindet sich hierbei im Ursprung, der Hebel liegt auf einem Vektor r, und F beschreibt die Kraft, die am Ende des Hebels wirkt. F muss nicht zwingend senkrecht zu r sein.Wenn ein weiterer Punkt s auf der Verlängerung der Kraftlinie liegen würde, so wäre das resultierende Drehmoment dasselbe.[Ebe04]

Für ein System mit p Partikeln an den Positionen r_i mit Einwirkenden Kräften $F_i, 1 \leq i \leq p$, ist das Drehmoment

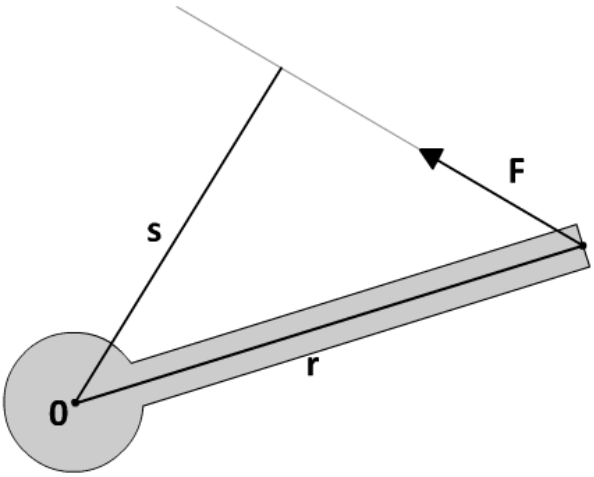

Abbildung 4: Drehmoment

$$\tau = \sum_{i=1}^{p} r_i \times F_i$$

Wegen Newtons 3. Gesetz ergibt das Drehmoment der nicht von außen wirkenden Kräfte in der Summe 0.

Trägheitsmomente

Bei unsymmetrischen Körpern reicht für die Rotation ein Trägheitsmoment nicht aus. Das kann man sich leicht vorstellen, wenn man einen kleinen Stab um eine Beliebige Achse rotieren lassen möchte. Je nach Rotationsachse müssen verschieden starke Kräfte an verschiedenen Stellen ausgeübt werden, um eine Bewegung des Körpers zu Bewirken. Der Trägheitstensor beschreibt für einen Körper seine Trägheitsmomente, oder die Kraft, die zu einer Änderung der Rotation notwendig ist, bezüglich der Rotation. Im homogenen Koordinatensystem kann der Tensor als Matrix dargestellt werden.

$$\sum_i m_i \begin{pmatrix} y_i^2 + z_i^2 & -x_i y_i & -x_i z_i \\ -y_i x_i & x_i^2 + z_i^2 & -y_i z_i \\ -z_i x_1 & -z_i y_i & x_i^2 + y_i^2 \end{pmatrix}$$

2.3 Physikengines

Eine Physikengine ist eine Programmkomponente, in der die Simulation der Umgebung durch Algorithmen berechnet werden kann. Sie kann bestimmte Effekte simulieren oder vorhersagen. Im Allgemeinen kann man zwei verschiedene Arten von Physikengines unterscheiden, die echtzeitfähigen und die präzisen Physikengines. Präzise Physikengines benötigen einen hohe Rechenleistung und Rechenzeit, um genaue Ergebnisse liefern zu können. Sie werden meistens in der Forschung oder in computeranimierten Filmen verwendet. In Echzeitanwendungen, wie z.B. den Computerspielen wird die Physikengine so vereinfacht, das Ergebnisse in angemessenen Zeitabständen berechnet werden können. Dazu werden verschidene Verfahren angewand, wie z.B. das Runge-Kutta-Näherungsverfahren zum lösen der Differentialgleichungen. Physikengines haben zwei Hauptkomponenten, ein System zur Kollisionserkennung und eine Komponente zum berechnen der dynamischen Simulation.

Lösen der Kollisionserkennung kann bei großen Szenen mit vielen komplexen Objekten sehr aufwändig werden, deshalb wird die Kollisionserkennung häufig auch in zwei Phasen aufgeteilt. In der sogenannten 'Broad Phase' wird zunächst eine Kollision mit Bounding Volumes(wie zum Beispiel AABB's oder k-DOP's) berechnet und nur die Treffer dieser Phase werden in der 'Narrow Phase' berechnet. Das Erstellen der Bounding Volumes kann in der Vorverarbeitung erfolgen.

Die Simulation wird in vom Benutzer festgelegten Zeitschritten berechnet. Dabei kommen die Bewegungsgesetze von Newton zum Einsatz. Gibt es Kollisionen, ist es manchmal sinnvoll, auf die Lagrange- Bewegungsgesetze umzuschalten, weil dort die Berechnung der resultierenden Kräfte einfacher wird. Die Wahl der Zeitschritte bestimmt die Stabilität der Simulation, sie solten nicht zu groß gewählt werden.

3 Umsetzung

3.1 verwendete Bibliotheken

Im Rahmen dieser Arbeit wurden drei Bibliotheken verwendet, um die Arbeit direkt auf einem objektorientierten Level durchführen zu können. Im Folgenden sind diese Bibliotheken genauer beschrieben.

3.1.1 OGRE - Object-Oriented Graphics Rendering Engine SDK

OGRE ist eine 3D Engine, die es dem Benutzer ermöglicht hardwarebeschleunigte 3D Grafiken zu nutzen. Hierzu abstrahiert OGRE darunterliegenden Systembibliotheken, wie zum Beispiel Direct3D und OpenGL, zu Intuitiven Klassen und Weltobjekten.[Ogr] OGRE bleibt dabei Plattformunabhängig und bietet viele Funktionen zur Manipulation und Verwaltung der Objekte und Materialien. Ich habe mich entschlossen OGRE zu verwenden, weil für mein Autorenwerkzeug eine Objektorientierte Sicht auf grafische Objekte ein wesentlicher Bestandteil der Architektur ist. Im Rahmen dieser Arbeit wird OGRE zum rendern der Physikszene in OpenGL genutzt.

3.1.2 Nvidia PhysX SDK

Nvidia PhysX ist eine 3D-Physikengine, die eine große Auswahl an Funktionen zur Beschreibung und Simulation physikalischer Körper und Szenen bietet. Je nach Konfiguration der Hardware des Systems können physikalische Berechnungen auch hardwarebeschleunigt durchgeführt werden. Mein Autorensystem Beschränkt sich hierbei auf die Verwendung von einfachen geometrischen Festkörpern und Bewegungseinschränkungen zwischen diesen Kollisionsgeometrien(Constraints). Das System, auf welchem das Autorensystem entwickelt wird, nutzt die Hardwarebeschleunigung nicht.

3.1.3 Trolltech Qt SDK

Qt bietet ein plattformunabhängiges System zur Entwicklung von (grafischen) Benutzeroberflächen. Dafür stellt Qt eine große Anzahl an Klassen zur Verfügung, die bereits wichtige Fensterelemente und Funktionen beinhalten. Durch ein eigenes Callbacksystem wird der Informationsaustausch zwischen den einzelnen Klassen in Qt stark vereinfacht. Für meine Arbeit habe ich Qt gewählt, um direkt Objektorientiert mit Fenstern und grafischen Elementen arbeiten zu können. Qt bietet außerdem eine Schnittstelle, mit der sich OpenGL gestützte Darstellungsfenster leicht in die Applikation integrieren lassen.[Tro]

3.2 Architektur

Damit Endbenutzer mit dem Autorensystem die bestmöglichen Ergebnisse erzielen können, muss das Autorensystem bestimmte Eigenschaften erfüllen. Es muss leicht erweiterbar gestaltet sein, damit Entwickler die Möglichkeit haben, das System anzupassen und weiterzuentwickeln. Der Aufbau soll möglichst einfach sein. Die Benutzeroberfläche muss intuitiv und leicht zu bedienen sein. Eine klare und übersichtliche Architektur ist wichtig, um diesen Anforderungen gerecht zu werden.

3.2.1 Datenstrukturen

Zunächst ist es für das Autorensystem wichtig eine interne Repräsentation von Dynamischen Objekten und Verbindungsstücken zwischen diesen Objekten zu finden. Bei den Dynamischen Objekten beschränke ich mich in dieser Arbeit auf Festkörper. Diese Klassen müssen auch dafür sorgen, dass die Darstellung der der grafischen Objekte immer mit der aktuell simulierten Physikszene übereinstimmt. Außerdem sollten sie so gestaltet sein, das eine Erweiterung der Funktionalität später noch möglich ist.

DynamicObject

Diese Klasse repräsentiert dynamische Objekte. Durch die eine update-Methode werden Physikszene und die Grafikszene synchronisiert. Dafür besitzt die Klasse Referenzen, die jeweils einmal das physikalische und das einmal grafische Objekt repräsentieren. Somit können alle Manipulationen und Operationen direkt an dem physikalischen Objekt durchgeführt und gleichzeitig direkt visualisiert werden. Die Klasse besitzt alle Methoden zum manipulieren des Dynamischen Objektes, und delegiert sie an den entsprechenden Member weiter. So werden änderungen, die ausschließlich die Physik betreffen (z.B. Masse), auch nur an das PhysicsObject weitergegeben. Gleichzeitig werden änderungen die Grafik betreffend (z.B Markieren bei Selektion) nur an das RenderObjekt weitergegeben.

PhysicsObject/RenderObject

Diese Klassen beschreiben die grafische bzw. die physikalische Repräsentation des DynamicObjects. Beide Klassen haben besitzen eine Bestimmte Anzahl von Primitiven(Shapes), die einzelne SzeneNodes beim RenderObject und einzelne KollisionsPrimitiven beim PhysicsObject darstellen. Beide Klassen haben Methoden zum transformieren des gesamten Objekts und seiner einzelnen Primitiven. Jede der beiden Klassen verwaltet die-

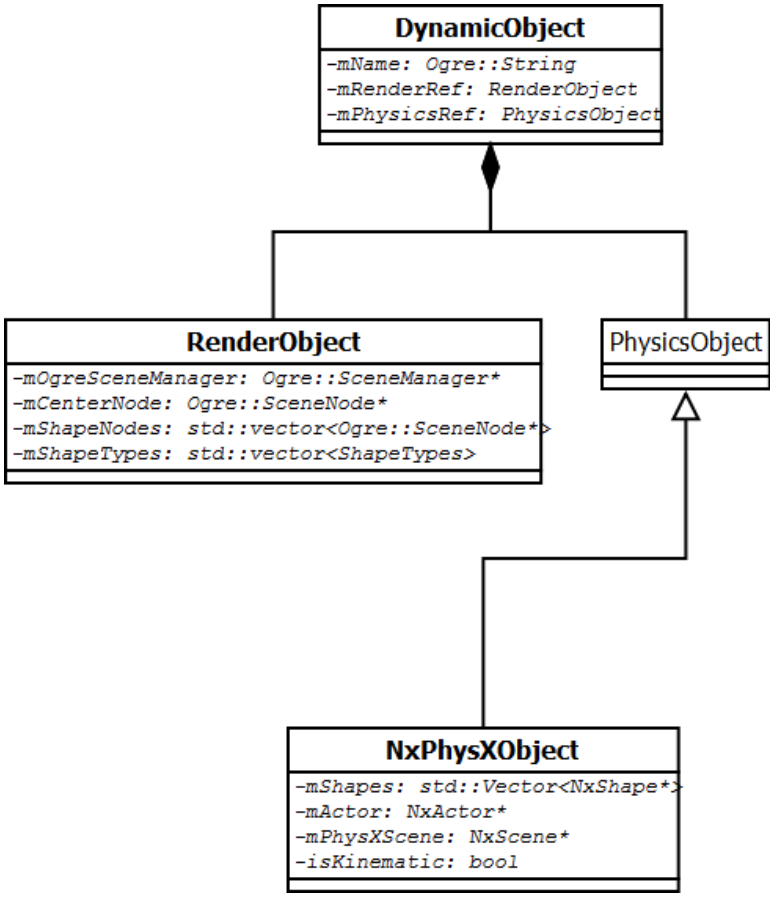

Abbildung 5: DynamicObject UML

se Shapes intern, weil je nach Physikengine unterschiedliche Architekturen für Kollisionsgeometrien vorliegen. Die Position und Größe der Einzelnen Shapes kann jederzeit verändert werden. Durch die Abkapselung des PhysicsObjects ist eine Implementierung durch verschiedene Physikengines möglich. Die Klasse PhysicsObject ist abstrakt, und wurde im Rahmen dieser Arbeit durch die Klasse NxPhysXObjekt implementiert, die eine Schnittstelle zur Nvidia PhysX Engine realisiert.

DynamicJoint

Hat man in einer Physikszene eine bestimmte Anzahl verschiedener Körper erzeugt, so will man diese nun in Beziehung miteinander bringen. Die Klasse DynamicJoint repräsentiert die Bewegungseinschränkungen zwischen Zwei Dynamischen Objekten. Mit ihr werden verschiedene Arten von Gelenken realisiert, die Einschränkung der Bewegung zwischen einzelnen Dynamischen Objekten zueinander realisieren. Der DynamicJoint

besteht aus einem Ankerpunkt und einer Hauptachse. Der DynamicJoint kann bis zu zwei Ziele haben. Diese Entsprechen den Objekten, für die die Bewegungseinschränkung gelten soll. Ist nur ein Objekt als Ziel angegeben, so wird das Objekt über das Gelenk relativ zur Welt in seiner Bewegung eingeschränkt.

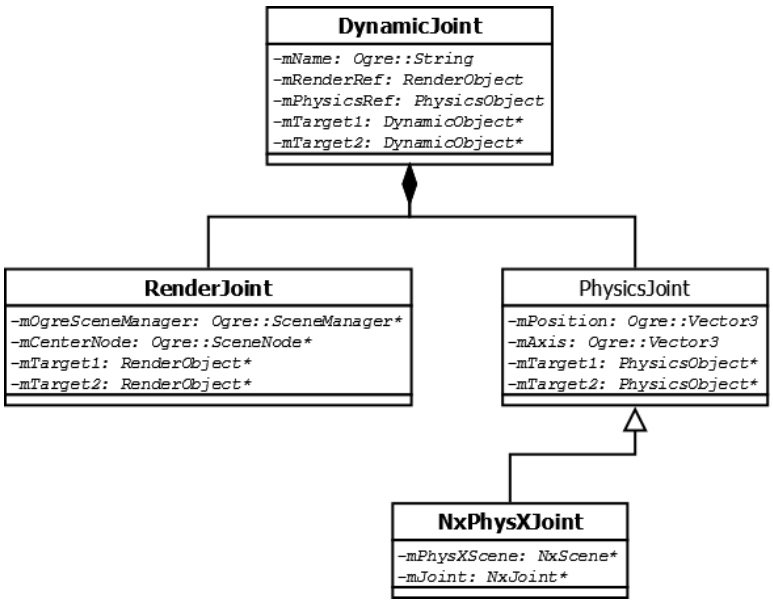

Abbildung 6: DynamicJoint UML

PhysicsJoint/RenderJoint

Auch der DynamicJoint besteht, wie schon das DynamicObject, jeweils aus einer physikalischen und einer grafischen Komponente. Grund dafür ist auch hier wieder die Abkapselung des PhysicJoints, um eine Implementierung der Physik durch verschiedene Physikengines zu ermöglichen. Die abstrakte Klasse PhysicsJoint ist im Rahmen dieser Arbeit durch die Klasse NxPhysXJoint implementiert.

3.2.2 Aufbau des GUI

Das Graphical User Interface wurde einfach gestaltet, um den Einstieg in die Arbeit mit dem Autorensystem zu vereinfachen. Das Hauptfenster hat eine Werkzeugleiste am oberen Rand. Die einzelnen Buttons sind zur besseren Übersicht gruppiert. Am rechten Rand befindet sich ein Browserbereich, der die 3D Szene in einer Baumstruktur darstellt. Darunter befindet sich ein Editorbereich, der nur bei selektierten Objekten oder Verbindungsstücken sichtbar ist. Hier können änderungen wie Position, Größe

und Ausrichtung direkt eingegeben werden. In der Mitte befindet sich das grafische Editorfenster, unterteilt in vier Verschiedene Ansichten (3D, Top, Front, Side). Hier werden Objekte erzeugt, manipuliert und miteinander Verbunden. Auf den ersten Blick erinnert der Aufbau an andere 3D Modellierungstools. Das erleichtert Benutzern mit Modelliererfahrung den Einstieg.

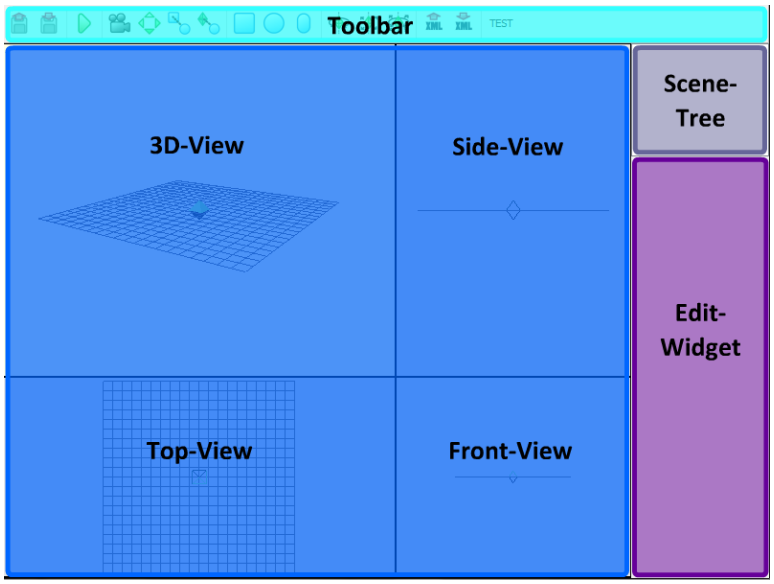

Abbildung 7: Benutzeroberfläche

3.2.3 Qt SIGNAL/SLOT Prinzip

Klassen in Qt verwenden ein eigenes Callbacksystem. Ein SIGNAL wird gesendet wenn ein bestimmtes Ereignis auftritt. Die Vordefinierten Klassen in Qt haben schon eine große Anzahl an Signalen, aber es ist jederzeit möglich diese Klassen Abzuleiten und eigene Signale hinzuzufügen.

Ein SLOT ist eine normale C++ Methode. Sie hat keinen Rückgabetyp (void), und kann als Antwort auf ein Signal aufgerufen werden. Auch hier gibt es in allen Qt Klassen schon einige vordefinierte Slots, es ist aber üblich diese Klassen abzuleiten um eine eigene Signalverarbeitung zu haben.

Der SIGNAL und SLOT Mechanismus ist Typsicher, d.h. die Signatur eines Signals muss zur Signatur des entsprechenden Slots passen. Der Compiler kann Typunstimmigkeiten erkennen, solange die Signaturen kompatibel sind. Eine Klasse, die ein Signal sendet, weiß nicht welche Slots in welchen Klassen dadurch ausgeführt werden. Der Signal/Slot Mechanismus

von Qt sorgt dafür, das alle Slots, die mit dem Signal verbunden sind, direkt nach dem Signal ausgeführt werden. Signale und Slots können eine beliebige Anzahl an Argumenten eines beliebigen Typs verwenden. Einem Slot können beliebig viele Signale zugeordnet werden, und ein Signal kann beliebig viele Slots auslösen.

Qt stellt für die Umsetzung des Callbacksystems den MetaObjectCompiler bereit. Dieser erstellt eigene .cpp Dateien in denen die Funktionspointer gespeichert werden. Diese Dateien werden automatisch generiert und der Benutzer muss sich nicht um die Verwaltung oder den Aufbau dieser Dateien kümmern. Das Signal/Slot Prinzip ist theoretisch langsamer als normale Callbacks, weil mehrere interne Funktionsaufrufe dem Aufruf der Slotfunktion vorangehen. Dies ist aber in großen Anwendungen kaum von Bedeutung, weil der Benutzer kaum etwas davon merkt. In dieser Arbeit wird das Signal/Slot Prinzip nicht für die Physik-Callbacks verwendet, sondern nur für die die Steuerung des GUI und der änderungen an dynamischen Objekten und Verbindungsstücken. Es kommt nicht vor, dass ein Signal (abgesehen vom mainLoop-Timer) jeden Frame aufgerufen wird, weshalb die leicht schlechtere Performance zu vernachlässigen ist.

Mithilfe des SIGNAL/SLOT Prinzips können die Daten in allen QWidgets konsistent gehalten werden. Wird z.B ein Dynamisches Objekt selektiert, egal ob im SzenenBaum oder im 3D Fenster, so wird ein Qt-Signal gesendet, das die entsprechende Slotfunktion bei den anderen QWidgets auslöst.

3.2.4 Verwendete QWidgets

Um die Funktionalität von Qt auszunutzen, sind vier QWidgets implementiert. QWidgets sind einfache Fenster, die ineinander geschachtelt werden können. Wird einem QWidget ein Elter zugewiesen, wird es innerhalb des Elterfensters gerendert. Hat ein QWidget keinen Elter, so wird ein neues Fenster erzeugt. Das Autorensystem hat nur ein Hauptfenster, darin werden drei weitere QWidgets verwendet.[ea04]

- Das OgreWidget ist für die grafische Darstellung der Szene verantwortlich, sowohl die 3D Ansicht als auch die drei 2D Ansichten werden hier gerendert.
- Das SceneTreeWidget zeigt die Szene als Baumstruktur, Dynamische Objekte und Verbindungsstücke können selektiert werden.
- Das EditWidget zeigt zu jedem im OgreWidget oder im SceneTreeWidget selektierten Objekt oder Verbindungsstück jeweils spezifische

Informationen an, die direkt durch Eingabe verändert werden können. Diese Informationen können Position, Orientierung, Masse, aber auch Größe und/oder Radius des aktuell angewählten Shapes sein.

- Das OgreWidget, das SceneTreeWidget und das EditWidget sind Kinder des MainWidgets, welches die Toolbar mit allen Buttons rendert, und die Physikszene verwaltet.

MainWidgetArchitektur

Die MainWidget Klasse ist das Herz des Autorensystems. Sie erzeugt und initialisiert alle für das Programm nötigen Bestandteile und beinhaltet damit auch die WrapperKlassen für den Renderer und die PhysikEngine. Sie erzeugt auch alle weiteren permanenten Widgets, sowie die Toolbar mit allen Buttons. Über Signale und Slots werden alle wichtigen Informationen zwischen dem MainWidget und den anderen Widgets ausgetauscht. Als Zentrales Element besitzt das MainWidget zwei Listen mit allen dynamischen Objekten und den Verbindungsstücken. Diese werden hier zentral verwaltet, damit jederzeit die aktuelle Szene angezeigt, gespeichert oder geladen werden kann.

Als einziges Widget besitzt das MainWidget einen $mainLoop()$ SLOT, der durch einen internen Timer immer wieder ausgelöst wird. In dieser Hauptschleife wird zunächst die Physikszene simuliert. Dann wird für alle aktuellen dynamischen Objekte und Verbindungsstücke die grafische Darstellung angepasst. Dies geschieht durch aufrufen der $update()$ Funktion jedes einzelnen dynamischen Objektes und jedes einzelnen Verbindungsstücks.

Da das MainWidget die einzige Klasse ist, in der Informationen über die gesamte Szene gespeichert sind, müssen alle änderungen, die die Anzahl dynamischer Objekte oder Verbindungsstücke betreffen, direkt von dieser Klasse ausgeführt werden. Dazu sind alle Methoden zum hinzufügen oder löschen von Dynamischen Objekten oder Verbindungsstücken implementiert. Modifikationen an einzelnen dynamischen Objekten oder Verbindungsstücken können dagegen auch von anderen Stellen im Programm ausgeführt werden.

OgreWidgetArchitektur

Das OgreWidget ist eine Generalisierung des QGLWidgets, zum rendern von Grafiken in OpenGL. Dazu implementiert es Methoden zur Initialisierung, zum Zeichnen und zur Veränderung der Fenstergröße, die von Qt verwaltet werden. In der InitializeGL Methode wird die FensterID an Ogre übergeben, damit Ogre in das QGLWidget rendern kann. Auf diese Wei-

se lässt sich Ogre als Renderer einfach in das Qt-Framework integrieren, gleichzeitig kann man die Listener des QGLWidgets nutzen, um Mausoperationen abzufangen.

Da die Dynamischen Objekte und Verbindungsstücke in der grafischen Darstellung auf unterschiedliche Weise verändert und manipuliert werden sollen, wurde die abstrakte Klasse MouseController entworfen. Klassen, die von der Klasse MouseController erben, implementieren Methoden zum Verarbeiten der Maussignale des OgreWidgets. Zu einem Zeitpunkt existiert immer nur ein MouseController. Welcher Mauscontroller aktuell verwendet werden soll, wird per Qt-Signal vom MainWidget beim klicken auf den entsprechenden Button an das OgreWidget weitergegeben. Dann wird der MouseController vom OgreWidget abhängig vom ausgelösten Qt-Slot erzeugt und verwendet. Der MouseController wird in Kapitel 3.3 naher beschrieben.

SceneTreeWidgetArchitektur

Das SceneTreeWidget ist eine Generalisierung des QTreeWidgets, welches die Verwaltung von Objekten in einer Baumstruktur ermöglicht. Diese Baumstruktur wird durch Qt gerendert, und bietet direkt die Möglichkeit durch anklicken Objekte zu verstecken bzw. aufzuklappen. Die Übergabe der Informationen geschieht durch den Elter, also das MainWidget. Wird hier ein Objekt erzeugt, gelöscht oder verändert, werden beim SceneTreeWidget die entsprechenden Methoden aufgerufen, um ein Element einzufügen, herauszunehmen oder zu aktualisieren. Das SceneTreeWidget ist über ein Qt-Signal mit dem MainWidget verbunden, sodass beim klicken auf ein Objekt im dargestellten Szenegraph über das MainWidget ein Objekt selektiert werden kann, und weitere Informationen im EditWidget angezeigt werden können.

EditWidgetArchitektur

Das EditWidget zeigt zu selektierten dynamischen Objekten jeweils alle relevanten Größen wie Position, Orientierung und Masse an. Außerdem werden, falls nicht das gesamte Objekt im SceneTreeWidget angewählt wurde, die Größen des aktuell selektierten Kollisionsprimitiven angezeigt. Alle diese Werte sind direkt änderbar und werden direkt für das dynamische Objekt übernommen.

3.2.5 Mouse Controller

Eine einfache intuitive Steuerung ist ein wichtiger Teil des GUIs. In diesem Autorensystem soll der Benutzer 3D-Objekte einfach durch die virtuelle Welt bewegen und rotieren, außerdem einfach neue Primitiven erzeugen und diese durch Verbindungsstücke miteinander in Beziehung bringen. Die Transformation im 3D-Raum gestaltet sich immer als schwierig, da ein Standardeingabegerät wie die Maus nur in zwei Dimensionen bewegt werden kann.

Um die Verschiedenen Mausfunktionen zu realisieren, müssen die Eingaben des Benutzers über die Maus abhängig vom gewählten Bearbeitungsmodus interpretiert werden. Dies Wurde in diesem Autorensystem durch die Klasse MouseController realisiert. Auswählen kann man den Mausmodus im Hauptfenster(MainWidget). Ein Klick auf den entsprechenden Button sendet ein Qt-Signal an das Renderfenster(OgreWidget), welches dann den entsprechenden MouseController erzeugt. Der MouseController wird dann mit dem jeweiligen Listener des OgreWidgets für die entsprechenden Mauseingaben verbunden, und Implementiert ihre Funktion.

MouseController Architektur

Der MouseController ist abstrakt und muss durch spezielle Klassen implementiert werden, die den entsprechenden Mausfunktionen entsprechen. Jedoch besitzt der Mousecontroller schon bereits Implementierte Funktionen, weil diese sich unter den MouseControllern nur wenig bis gar nicht unterscheiden. Dies sind Methoden zum berechnen eines Schnittpunktes mit einem Strahl und der 3D-Szene, sowie der Schnitt eines Strahls und einer Ebene. Diese Funktionen sind Grundlage aller Mausoperationen in der Physikszene. Sie werden zur Selektion von bestimmten Dynamischen Objekten oder Verbindungsstücken verwendet.

Durch die abstrakte Klasse ist die Implementierung weiterer MouseController leicht möglich. Damit kann das Autorensystem einfach um neue Mausfunktionen erweitert werden, um eventuell speziellen Anforderungen des Benutzers gerecht zu werden. Alle grundlegenden Mausfunktionen wurden im Rahmen dieser Arbeit implementiert.

- MouseObjectCreate zum erstellen von dynamischen Objekten mit einem einfachen Kollisionskörper(Quader, Kugel, Kapsel)
- MouseJointCreate zum erstellen eines Gelenks(ein, zwei oder drei Freiheitsgrade in der Rotation)

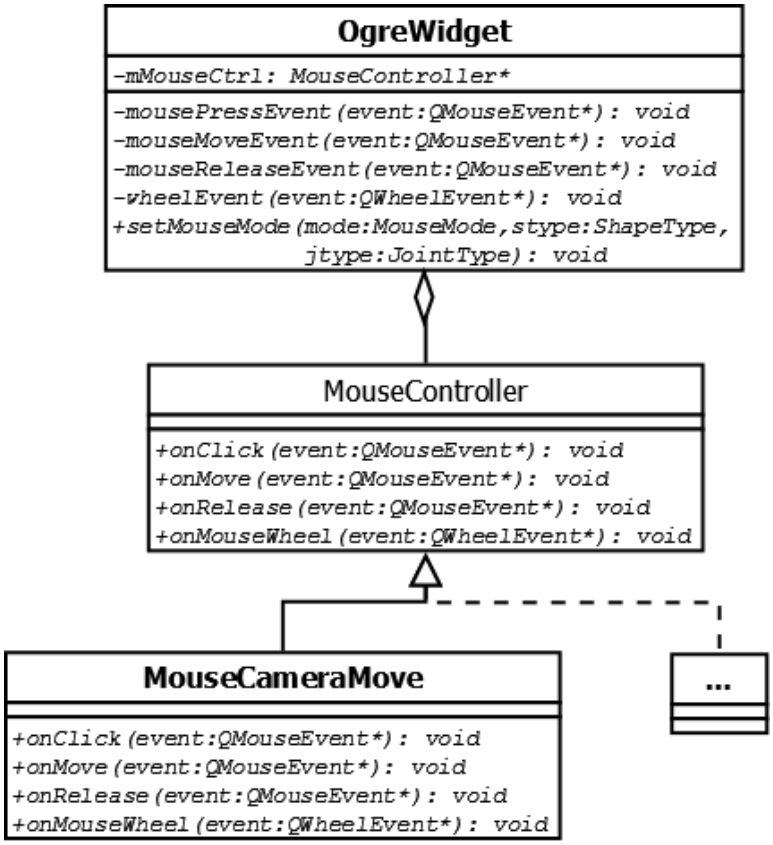

Abbildung 8: MouseController UML

- MouseObjectMove zum Bewegen und Drehen von Objekten und Verbindungsstücken in der Physikszene
- MouseCameraMove zum Bewegen und drehen der Kamera im aktuell gewählten Viewport
- MouseCombineObjects zum Verbinden von einfachen dynamischen Objekten zu komplexeren Objekten(Zusammenfügen der einzelnen Kollisionsprimitiven)
- MouseJointLink zum Verbinden eines Gelenks mit Objekten, wobei maximal zwei Objekte mit einem Verbindungsstück verbunden sein können. Werden mehr als zwei Objekte selektiert, so wird jeweils die am längsten bestehende Verbindung wieder aufgelöst.

3.3 Daten Im-/Export

Für das Autorensystem ist es unbedingt erforderlich, das man nach dem Autorenprozess die Daten für andere Programme verfügbar machen kann. Deshalb wurde in dieser Arbeit ein Export in das XML-Format (Extensible Markup Language) realisiert. Die Baumstruktur von XML eignet sich hervorragend für die Physikszene, weil innerhalb der Applikation jedes Dynamische Objekt aus Unterobjekten wie zum Beispiel den Shapes besteht.

In dem XML Dokument sollen nur die Informationen beschreiben werden, die für die vollständige Beschreibung der Physikszene ausreichend sind. Informationen wie Kameraposition, Materialfarben oder Modellgeometrie kann vernachlässigt werden. Modellgeometrie wird erst dann wichtig, wenn sich die einzelnen Kollisionsgeometrien nicht mehr durch wenige Parameter beschreiben lassen, was im Rahmen dieser Arbeit nicht der Fall ist. Dies erleichtert die Lesbarkeit des Dokuments sowie den Import. Eine Erweiterung des Formats um entsprechenden Parameter zur Darstellung dieser Geometrie ist allerdings denkbar.

Die Dynamischen Objekte werden in der XML-Struktur unter Kollisionsgeometrien zusammengefasst. Jedes Physikobjekt hat einen Namen bzw. eine ID, eine globale Position, eine globale Orientierung und eine bestimmte Anzahl von Kollisionsshapes. Jedes Shape-Tag enthält wiederum Typ, lokale Position und Orientierung, sowie die Ausdehnungsparameter abhängig von Typ. Die Gelenke werden in einem Constraints-Tag beschrieben. Jedes Gelenk enthält seine Position, seine Hauptachse, den Gelenktyp und die Namen der Objekte, die mit dem Gelenk verbunden sind.

```
<PhysicsScene>
    <CollisionGeometry>
        <PhysicsObject id="Object 0" mass="149.999996" kinematic="false">
            <Position x="4.0" y="1.0" z="-3.0" type="global" />
            <Orientation x="0.0" y="0.0" z="0.0" w="1.0" type="global" />
            <Shape type="Sphere" radius="1.0">
                <Position x="0.0" y="0.0" z="0.0" type="local" />
                <Orientation x="0.0" y="0.0" z="0.0" w="1.0" type="local" />
            </Shape>
        </PhysicsObject>
    </CollisionGeometry>
    <Constraints>
        <PhysicsJoint id="Joint 0" type="Spherical">
            <Position x="-4" y="2" z="-7" type="global" />
            <Axis x="0" y="0" z="1" type="global" />
            <Link linkid="Object 0" />
        </PhysicsJoint>
    </Constraints>
</PhysicsScene>
```

Abbildung 9: XML-Einfaches Beispiel

4 Ergebnis

Das Ziel dieser Bachelorarbeit ist die Umsetzung des in den vorangehenden Kapiteln beschriebenen Autorensystems. Die in Kapitel 3 beschriebene Architektur wurde mit Microsoft Visual Studio 2005 in C++ realisiert. Die ebenfalls in Kapitel 3 beschriebenen Bibliotheken wurden integriert. Ergebnis ist eine Anwendung, die 3D-Grafik, Physiksimulation und eine einfache Bedienoberfläche kombiniert. Dabei bleibt die Anwendung leicht erweiterbar, sowohl was das einführen neuer Bearbeitungsmodi betrifft, als auch die Umstellung auf andere bzw. Einführung weiterer Physikengines zur Berechnung der Simulation.

4.1 Workflow

Beim Erstellen von physikalischen Szenen kann der Benutzer Schrittweise vorgehen. Die vier Hauptschritte des Autorenprozesses sind im Folgenden genauer beschrieben.

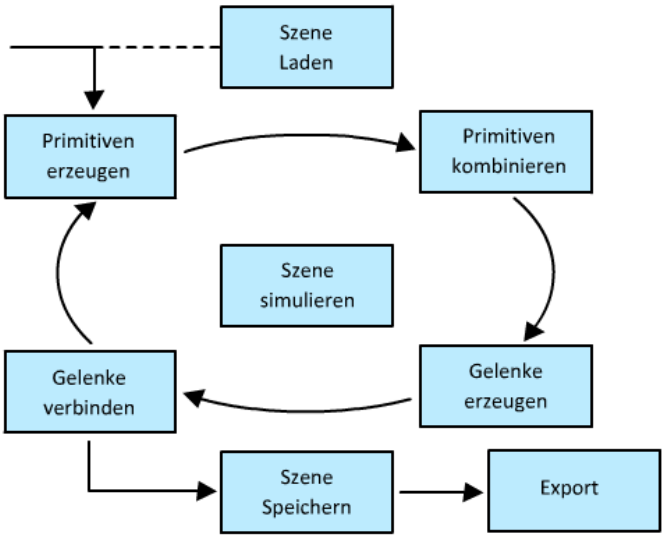

Abbildung 10: Workflow

Erzeugen einfacher Primitiven

Zunächst muss die Szene durch verschiedene Grundobjekte modelliert werden. Hierfür erzeugt man mit den Entsprechenden Buttons Kugeln, Quader und Kapseln und setzt diese an die gewünschte Position. Für die exakte GrÖße, Position und Orientierung kann zusätzlich das EditWidget ver-

wendet werden. Dieses wird sichtbar, sobald man ein Objekt mit der Maus anklickt oder es im SceneTreeWidget auswählt.

Abbildung 11: Basisprimitiven

Kombinieren der Primitiven

Nachdem ein Modell oder eine Szene grob nachmodelliert wurde, werden die Objekte, die sich später vÖllig starr zueinander verhalten sollen, miteinander kombiniert. Mit dem Parent-Tool werden zunächst das Kind und dann der Elter ausgewählt. Alle Kollisionsshapes des Kindes werden dem Elter hinzugefügt. Das Kindobjekt wird danach gelÖscht. Der Ursprung des Elternobjektes bleibt erhalten und wird nicht verschoben. Dies ist für die Physiksimulation nicht so relevant, aber wenn der Benutzer das Objekt später rotiert, geschieht dies um den Ursprung des Elters.

Abbildung 12: Objekte Zusamenfügen

Erzeugen von Verbindungsstücken

Zu diesem Zeitpunkt kann das Modell der Szene bereits vollständig modelliert sein. Allerdings haben die Objekte im Moment noch keine Beziehung zueinander. Simuliert man die Szene, wird lediglich eine eventuelle Kollision berechnet, außerdem die Wirkung der Schwerkraft. Im Nächsten Schritt werden nun die Verbindungsstücke(Joints) erzeugt, die später einzelne Objekte zusammenhalten sollen. Dies wird analog wie das erstellen der Primitiven über die entsprechenden Buttons in der Toolbar getan.

Abbildung 13: verschiedene Gelenktypen

Verbinden der einzelnen Objekte mit den Gelenken

Sind die Gelenke platziert, muss man sie jetzt noch den entsprechenden Objekten zuordnen. Dazu wählt man wieder das entsprechende Werkzeug aus der Toolbar aus, in diesem Fall das Gelenk-verbinden Tool. Das Werkzeug funktioniert wie schon das Parent-Tool muss man wieder zuerst das

Gelenk auswählen, welches später Objekte verbinden soll. Dann selektiert man Objekte in der Szene, die Zuordnung erfolgt dann automatisch. Allerdings kann ein Gelenk immer nur maximal zwei Objekte miteinander verbinden. Wählt der Benutzer ein drittes Objekt aus, so wird die erste Verbindung wieder aufgelÖst. Wird hingegen überhaupt nur ein Objekt angewählt, so wird das Gelenk als Anker in der Welt verstanden.

Abbildung 14: Gelenke verbinden

4.2 Autorenprozesses am Beispiel (eines Abrisskrans)

Um die Fähigkeiten und den Einsatzbereich des Autorensystems zu demonstrieren, habe ich als Beispiel einen Abrisskran modelliert. Physikalisch gesehen besteht er aus der Kombination einiger Kollisionsprimitiven. Er benötigt verschiedene Verbindungsstücke für die Achsen der Räder, die Rotationsachse des Krans sowie einige Kettenglieder für die Abrissbirne.

Erstellen des Fahrwerks

Zunächst Werden sechs Kugeln erstellt, die später die Räder repräsentieren sollen. Darauf wird das Chassis als Box gesetzt. Der Radius der Kugeln wird nach der Selektion im EditWidget angepasst, damit alle Räder exakt die gleiche Größe haben. Die Genaue Höhe und Position kann bei Bedarf auch im EditWidget angepasst werden. Neben diesen Geometrischen Größen wird auch die Masse für alle Räder gleich eingestellt.

Erstellen der Fahrerkabine und des Kran-Arms

Anschließend wird aus drei Boxen verschiedener Größe ein Gegengewicht, ein Führerhaus und der Kran-Arm modelliert und positioniert. Diese drei Objekte verden mit dem Objekt-Verbinden-Tool zu einem Kollisionsobjekt verschmolzen, da sie ihre Position zueinander nicht verändern. In diesem Beispiel ist der Arm fest und kann nicht nach oben oder unten bewegt werden.

Erstellen der Kette und der Abrissbirne

Zum Schluss werden für die Kette repräsentativ drei Kapseln erzeugt, sowie eine Kugel, die später die Abrissbirne darstellen soll. Jetzt ist das Modell vollständig modelliert. Allerdings sind noch keinerlei Verbindungen

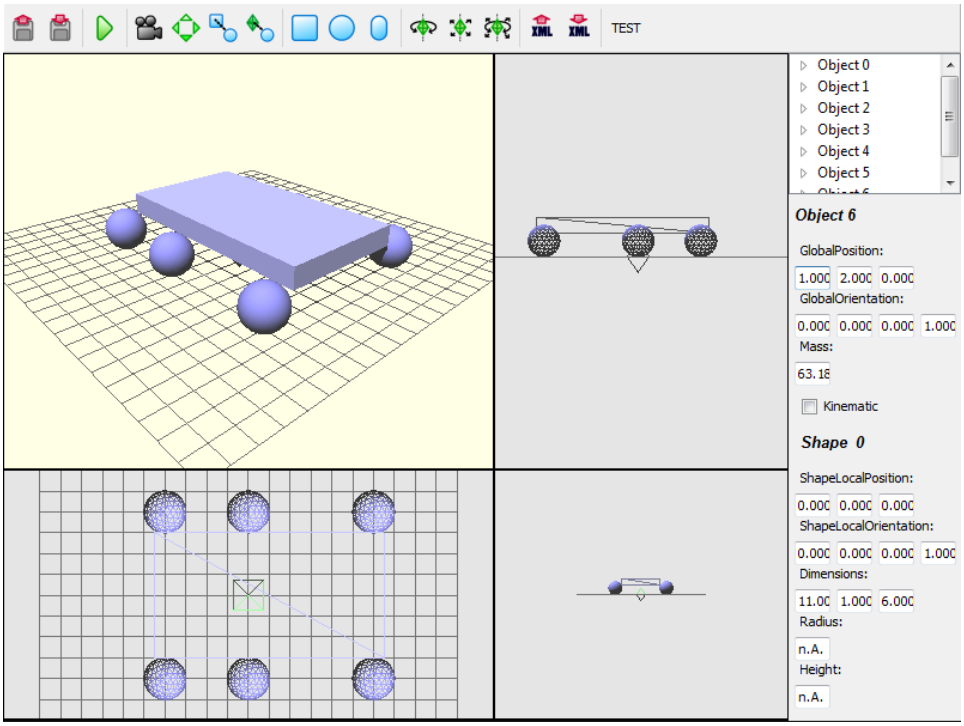

Abbildung 15: Fahrwerk des Krans

zwischen den einzelnen Teilen definiert. Das Gegengewicht, der Kran-Arm und das Führerhaus sollen ihre Position zueinander nie verändern, und können somit zu einem Objekt zusammengefasst werden. Dies wird mit dem Parent-Tool gemacht. Als Elter wird immer das Gegengewicht gewählt.

Erzeugen der Gelenke

Im nächsten Schritt werden an allen Gelenkpunkten des Krans die entsprechenden Gelenke erzeugt und deren Achsen festgelegt. Es werden einfache Achsen für die Räder und die Rotationsachse des oberen Krans benötigt, sowie Kugelgelenke für die einzelnen Kettenglieder.

Verlinken der Gelenke mit den Objekten

Abschließend wird jedes Gelenk mit zwei Objekten verbunden. In diesem Fall heißt das, jedes Rad wird mit dem Chassis verbunden, der obere Krankomplex wird mit dem Chassis verbunden, Die Kettenglieder werden untereinander verbunden und die Endstücke der Kette jeweils mit dem oberen Krankomplex oder der Abrissbirne.

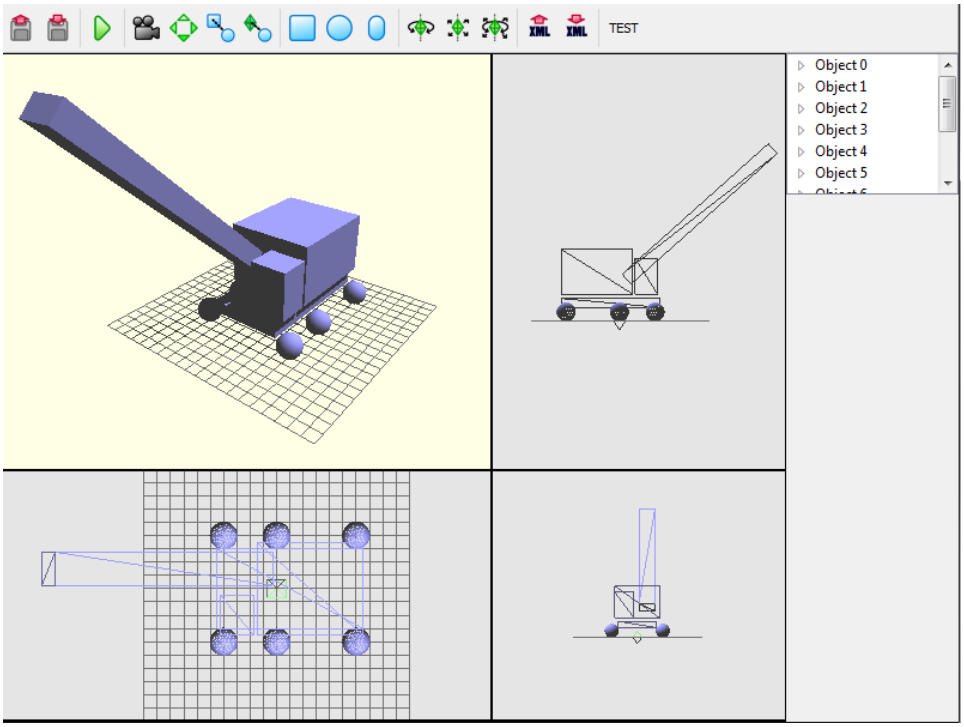

Abbildung 16: Oberer Komplex des Krans

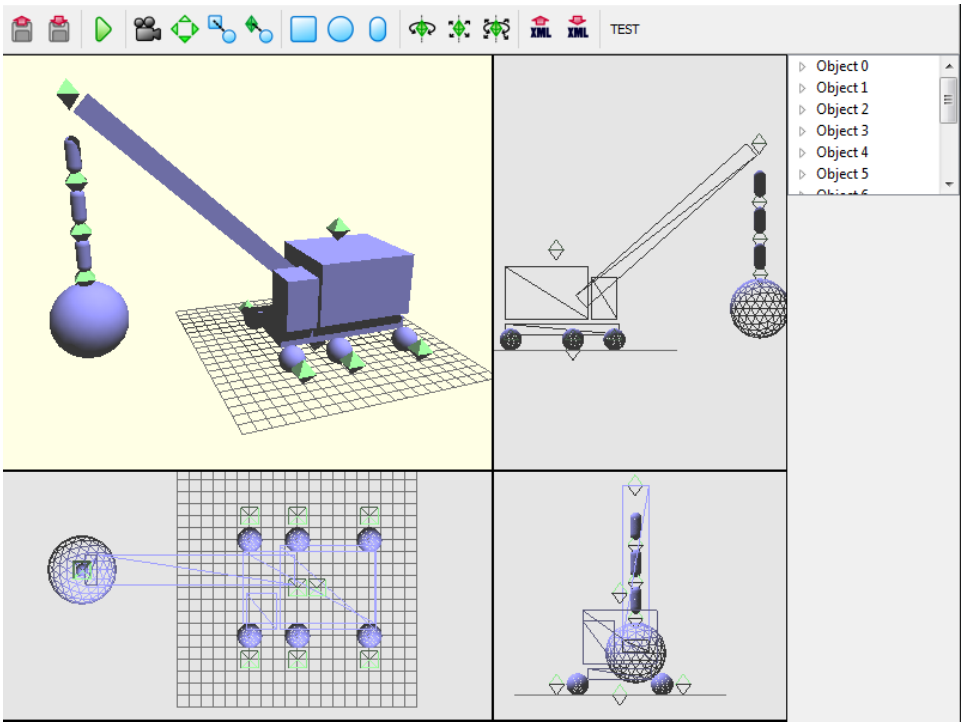

Abbildung 17: Kran mit Gelenken

Damit ist der Abrisskran fertiggestellt. Man kann die Szene im nachsten Schritt speichern, oder aber durch drücken auf den Play-Button die Simulation starten. Eine weitere möglichkeit wäre der Export in das XML-Format.

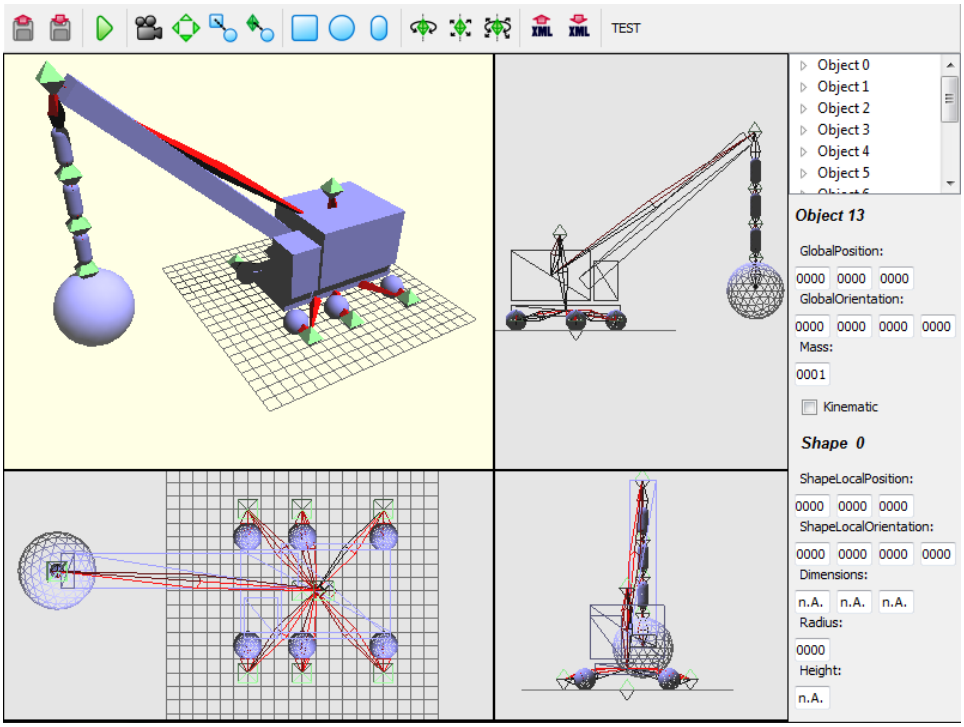

Abbildung 18: Endversion des Abrisskrans

4.3 Ausblick

Das Autorensystem bietet zu diesem Zeitpunkt alle grundlegenden Funktionen zum modellieren von physikalischen Szenen. Je nach Erwartungen oder Einsatzart wird es eventuell nötig sein, das Autorensystem durch hinzufügen von Funktionalität zu verändern bzw. zu erweitern, um es noch besser nutzen zu können. Das in dieser Arbeit vorgestellte Autorensystem wurde so konzipiert, das eine Erweiterung des Funktionsumfangs leicht zu realisieren ist.

Steuerbarkeit

Es könnten weitere MouseController hinzugefügt werden, um dem Benutzer das Modellieren noch einfacher zu machen. Beispiele dafür wären Controller zum skalieren oder duplizieren von Objekten, aber auch das Erzeugen weiterer Primitiven, wie zum Beispiel Tonnen, Pyramiden, oder gene-

rell konvexen Festkörpern.

Physikalische Größen

Bis jetzt lassen sich die Ausdehnung eines Objektes und dessen Masse beschreiben. Eine Erweiterung hierzu wäre das Hinzufügen eines Materialeditors, mit dem sich die Oberflächenbeschaffenheit der einzelnen dynamischen Objekte verändern lässt. Damit könnte der Benutzer Gleit- und Haftreibung direkt beeinflussen. Weite Möglichkeiten wären das Manuelle Editieren des Massezentrums und des Trägheitstensors. Im Rahmen dieser Arbeit werden diese Entitäten automatisch Berechnet, weil das Eintragen konkreter Werte direkt in die Tensormatrix sicherlich nicht intuitiv ist.

Constraints

In diesem Autorensystem wurden drei wichtige Arten von Gelenken eingeführt, allerdings gibt es noch mehr Möglichkeiten die Objektbewegungen gegeneinander einzuschränken. Das hinzufügen weiterer Gelenke ist durch den einfachen Aufbau der PhysicsJoint Klasse einfach zu bewältigen. Beispiele hierfür wären Verschiebungen der Objekte zueinander, wie zum Beispiel bei einer hydraulischen Pumpe.

Ein weiterer Aspekt wäre die Einführung von Motoren und Einschränkungen der Gelenkbewegung. Bei Bestimmten Modellen will man zum Beispiel einen Türstopper nicht modellieren, sondern diesen durch eine Einschränkung der Türrotation realisieren. Motoren sind dann sinnvoll, wenn man z.B. Maschinen nachbauen möchte, die eine Antriebsachse haben. Vor allem dann, wen man die Funktionalität innerhalb des Autorensystems testen möchte, nicht erst nach dem Ex- bzw. Import.

Federn

Federn werden in diesem Autorensystem in der aktuellen Version nicht behandelt. Es ist jedoch denkbar und möglich, dass diese in das bestehende System integriert werden, um dem Benutzer noch mehr Möglichkeiten zu bieten.

4.4 Fazit

Abschließend lässt sich sagen, dass es noch viele Möglichkeiten gibt das bestehende System zu erweitern und auszubauen. Das in dieser Arbeit entwickelte Autorensystem liefert die Grundlage eines Physikmodellierungstools, welches von Benutzern in Zukunft zum grafischen Erstellen von Physikmodellen und Physikszenen genutzt werden soll. Eine Weiterentwicklung durch Benutzer und Anpassung auf spezielle Probleme ist hierbei erwünscht und kann leicht vorgenommen werden. Die Erweiterbarkeit und Transparenz des Codes ist dafür sehr wichtig. Derzeit wird das Tool vom laufenden Projektpraktikum der Universität Koblenz verwendet, um Kollisionsmodelle für ein Rennspiel anzufertigen. Ich hoffe das dieses Autorenwerkzeug weiterhin in mehreren Projekten verwendung findet und durch Weiterentwicklung komplexer und mächtiger wird.

Literatur

[ea04] Matthias Kalle Dalheimer; Jesper Pedersen et al. *Practical Qt*. dpunkt.Verlag GmbH, 2004.

[Ebe04] David H. Eberly. *Game Physics*. Elsevier Inc., 2004.

[Ogr] OgreDevelopers. Ogre 1.6 api reference. http://www.ogre3d.org/docs/api/html.

[Tro] Trolltech. Qt reference documentation. http://doc.trolltech.com/4.5/index.html.