OBJEKTORIENTIERTE GEBÄUDEMODELLIERUNG

Diplomarbeit

der Philosophisch–naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Thierry Matthey

1997

Leiter der Arbeit:

Prof. Dr. H. Bieri

Institut für Informatik und
angewandte Mathematik

Zusammenfassung

Die computerunterstützte Gebäudemodellierung hat ihre Wurzeln in der Computergrafik und befasst sich mit der Modellierung von architektonischen Bauten. Der Modellierungsablauf beinhaltet die Beschaffung der Spezifikation der Gebäude, die Manipulation der Szene mit den Gebäuden und die Visualisierung der Szene. Die Gebäude selber werden je nach Aufgabenstellung innen und/oder aussen mit entsprechendem Detaillierungsgrad modelliert.

In der vorliegenden Arbeit werden drei bekannte Modellierungsansätze und deren Möglichkeiten besprochen. Weiter wird ein eigener objektorientierter Ansatz vorgestellt, der einem hierarchischen Baukastensystem ähnelt. Die Modellierung beschränkt sich hier auf die äussere Hülle der Gebäude und lässt Innenausstattung aus.

Für die Implementation wurde der objektorientierte Ansatz gewählt und mit Hilfe des Frameworks *BOOGA*[1] umgesetzt. Dabei ist versucht worden, die OO-Möglichkeiten des Frameworks voll auszuschöpfen. Die Arbeit erhebt keinen allgemeinen Anspruch auf Fotorealismus, sondern soll primär die Möglichkeiten der objektorientierten Modellierung aufzeigen.

[1] **B**erne's **O**bject–**O**riented **G**raphics **A**rchitecture.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das Erzeugen von statischen realistischen Computerbildern scheint in weiten Teilen gelöst. Die Unterschiede zwischen echten Fotografien und auf dem Computer berechneten Bildern sind kaum noch von blossem Auge erkennbar und erlauben, utopische oder schon längst vergangene Szenen, wie Städte der Antike, zu modellieren.

Nach wie vor aufwendig, schwierig und arbeitsintensiv ist die Bereitstellung der benötigten geometrischen Modelle. Sind technische Gebilde meist in recht kurzer Zeit mittels CAD-Programmen, geeigneten Bibliotheken und Texturen zu modellieren, so werden die Grenzen dieser Methode bei einem Stadtrundgang oder beim Betreten eines grösseren Gebäudekomplexes rasch offenbar. Ein Gebäude kann zwar mit einfachen geometrischen Objekten repräsentiert werden, doch werden für ansprechende Ergebnisse eine Vielzahl von geometrischen Objekten benötigt, was die Modellierung und Manipulation von grösseren Szenen umständlich, träge und monoton werden lässt.

Die computerunterstützte Gebäudemodellierung versucht nun hier mit geeigneten Ansätzen, die auf Gebäude ausgerichtet sind, die Modellierung und Manipulation zu vereinfachen. Die Wahl des Ansatzes hängt einerseits von der Aufgabenstellung, anderseits von schon vorhandenen Daten und den Spezifikationen der Gebäude ab.

1.1 Motivation und Ziele

Die vorliegende Arbeit befasst sich mit der computerunterstützten Gebäudemodellierung. Folgende Ziele habe ich verfolgt:

- Entwicklung eines eigenen Ansatzes, der die Modellierung von Gebäuden und Gebäudekomplexen vereinfacht oder teilweise automatisiert. Der Ansatz sollte die Möglichkeiten der objektorientierten Umgebung ausnutzen.
- Möglichst einfache Integration in das objektorientierte Framework *BOOGA*[1] und einen Aufbau auf der bestehenden objektorientierten Philosophie.
- Erstellen kleinerer Anwendungen, die modellierte Gebäude verarbeiten können.
- Zusammenstellen einer kleinen Bibliothek von Gebäuden, Szenen und kleinen gebäudespezifischen, geometrischen Objekten.

[1] **B**erne's **O**bject–**O**riented **G**raphics **A**rchitecture.

1.2 Aufbau der Arbeit

Die Arbeit ist in drei grössere Teile gegliedert. Im ersten Teil wird eine Einführung in die allgemeine Gebäudemodellierung mit dem Modellierungsablauf gegeben. Ausserdem werden drei wichtige Modellierungsansätze und deren Möglichkeiten erklärt.

Im zweiten Teil wird ein eigener objektorientierter Ansatz entwickelt. Dazu werden die nötigen Begriffe und Grundlagen eingeführt. Der Modellierungsansatz wird ausführlich beschprochen. Am Schluss dieses Teiles wird der Ansatz anhand eines einfachen Beispiels und einer kleinen Anwendung erklärt. Dieser Teil schliesst mit farbigen Illustrationen, die die Möglichkeiten der Gebäudemodellierung aufzeigen sollen.

Der dritte Teil befasst sich mit der Implementation des objektorientierten Modellierungsansatzes und der Integration in *BOOGA*. Dabei werden die nötigen Grundlagen, Algorithmen und Begriffe eingeführt. Die selbstentwickelten Objekte werden anhand von Illustrationen erklärt. Ausserdem wird eine genaue Sprachdefinition der Objekte für die *BOOGA*-Eingabesprache BSDL [2] gegeben.

Die Arbeit schliesst mit BSDL-Beispielen für die Erzeugung von Gebäuden und dem vollständigen Quellcode der im zweiten Teil beschrochenen Anwendung.

1.3 BOOGA (Berne's Object-Oriented Graphics Architecture)

Das Framework *BOOGA* ist objektorientiert und deckt einen grossen Teil der Computergrafik ab. Es wurde von Christoph Streit [Streit 97] und Stephan Amann im Rahmen ihrer Dissertationen am Institut für Informatik und angewandte Mathematik (IAM) der Universität Bern entwickelt. Mittlerweile wird es praktisch von allen Mitgliedern der Fachgruppe Computergeometrie (FCG) eingesetzt. Es soll die Entwicklung von Applikationen vereinfachen, indem es einerseits standardisierte, wiederverwendbare Softwarekomponenten enthält und anderseits das Entwickeln und das Implementieren neuer Komponenten unterstützt. *BOOGA* dient auch als Forschungsplattform zur Entwicklung neuer Techniken und Algorithmen. Durch die vielfältigen Anforderungen wird eine flexible Plattform gefordert, die einfach erweiterbar ist. Damit wird ein hoher Wiederverwendungsgrad erreicht. Es sollte auch möglich sein, Resultate der Entwickler in Folgearbeiten ohne Änderung wiederverwenden zu können. Das Gebiet von *BOOGA* umfasst Teilbereiche der 2D und 3D Computergrafik. Es werden die Objektmodellierung für 2D und 3D, die Bildgenerierung und die Bildanalyse unterstützt. Das Design der 2D und 3D Grafik ist so weit als möglich vereinheitlicht. Die Architektur basiert auf einem einfachen Datenflussmodell (Abb. 1.1). Die Daten sind 2D bzw. 3D Szenen, in *BOOGA* Welten genannt. Die möglichen Operationen und Übergänge der Welten werden Komponenten genannt. Es wird von vier Komponententypen ausgegangen:

[2] **B**OOGA **S**cene **D**escription **L**anguage.

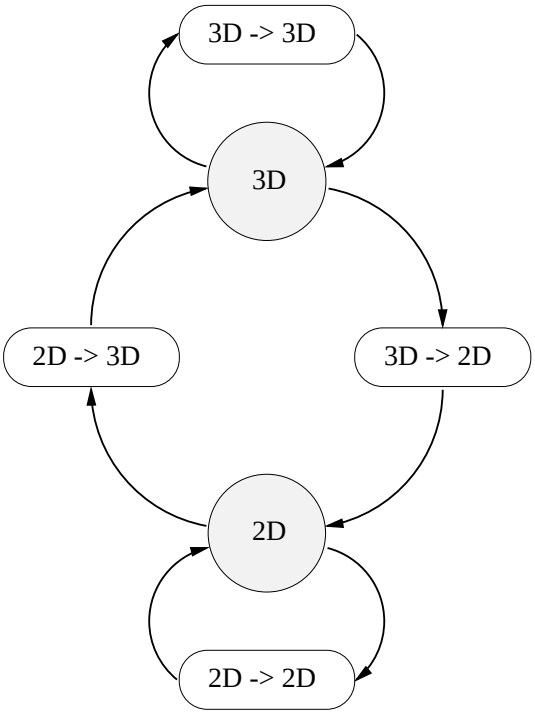

Abb. 1.1: Datenflussmodell von BOOGA.

- Arbeiten mit einer 2D Welt.
- Erzeugen einer 3D Welt anhand einer 2D Welt.
- Arbeiten mit einer 3D Welt.
- Erzeugen einer 2D Welt anhand einer 3D Welt.

Die Einfachheit des Konzeptes lässt eine breite Abdeckung der Anwendungsgebiete zu. Die Erweiterbarkeit und die komponentenorientierte Softwareentwicklung wurde durch ein objektorientiertes Design sowie durch den Gebrauch passender *design patterns* [Gam 95] erreicht.

1.4 Hilfsmittel

Die Arbeit habe ich parallel auf den Hardwareplattformen Sun Sparc (SunOS) und IBM RS/6000 (AIX) mit OpenGL-Hardwarebeschleunigern entwickelt und dabei das Programmiertool SNiFF+ 2.1 verwendet, wie alle anderen, die mit BOOGA arbeiten, was mir das Kompilieren, Editieren und Debuggen sehr stark erleichterte. SNiFF+ enthält vielfältige Tools zur Bearbeitung von Grossprojekten. Zum einem wird gleichzeitiges Entwickeln mit eigenem Workspace und eigenem Entwicklerstatus unterstützt, zum andern enthält SNiFF+ einen Hierarchie- und einen Klassenbrowser, die den Umgang mit den Klassen erleichtern, sowie Tools, die das Updaten oder das Vergleichen zweier Textdateien (Quellcode) übernehmen. Unter SNiFF+ wurde der relativ strenge C++-Compiler GNU-Compiler

2.7.2 unter SunOS und *cset* unter AIX benutzt. Unerlaubte und unkorrekte Speicherzugriffe wurden mit Hilfe von Purify Version 3.2 und 4.0 unter SunOS ausfindig gemacht. Purify ist auch bei der Suche nach nicht freigegebenem Speicher behilflich.

Als Basis und Plattform für die Entwicklung wurde BOOGA verwendet, das viele wichtige Ausgabemöglichkeiten des Renderings abdeckt und das eine Eingabesprache (BSDL) für die Modellierung von Szenen mit Objekten enthält. Dank BOOGA konnte ich von schon implementierten Applikationen und Klassen profitieren. Sie dienten mir als Beispiel für die Entwicklung von Applikationen und Klassen.

Die Dokumentation editierte ich mit Microsoft Word und portierte sie dann nach LaTeX, was mir eine sichere und einfache Verwaltung der Kapitel, der Inhaltsverzeichnisse, etc. bot. Für die eingebundenen Figuren benutzte ich XFIG, das importierbare Dateien für LaTeX erzeugt. Damit war es mir mit nicht allzu grossem Aufwand möglich, ein professionelles Layout zu erstellen.

Für die Modellierung der Altstadt Bern ist mir eine Spezialkarte[3] der OLG Bern in digitaler Form zur Verfügung gestellt worden. Mit dieser Karte und dem OCAD-Programm[4] von Hans Steinegger[5] konnte ich die Grundrisse bequem und schnell überarbeiten. Mit einem eigens geschriebenen Konvertierungsprogramm portierte ich die überarbeitete OL-Karte ins BSDL-Format. Das Erzeugen des Geländemodelles und das Platzieren der Häuser wurde mit kleineren, auf BOOGA aufbauenden Applikationen erledigt.

1.5 Danksagungen

An dieser Stelle möchte ich Herrn Professor Bieri, dem Leiter der vorliegenden Arbeit, für seine kompetente Führung und den mir gewährten Freiraum in der Entwicklungsphase danken.

Herzlicher Dank gebührt auch meinem Betreuer Bernhard Bühlmann, der mich in dieser Arbeit unterstützt hat. Er hatte sich, wann immer möglich, die Zeit genommen, mir Fragen ausführlich und nach bestem Wissen zu beantworten. Auch lehrte er mir die Tricks, die einen Informatiker am Leben erhalten.

Als nächstes danke ich allen FCG[6] Mitarbeitern, die mich mit Tips und Tricks und guten Ideen unterstützt haben.

Schliesslich seien auch all jene erwähnt, die sich für mein Projekt interessierten und dadurch manche Anregungen an mich weitergeben konnten.

Last but not least möchte ich meinem norwegischen Freundeskreis und speziell meiner Freundin danken, die es mir ermöglichten, von Norwegen aus meine Arbeit voranzutreiben.

[3] Landeskarte für Orientierungslauf; Massstab 1:5'000 und Äquidistanz 5m.
[4] OCAD ist ein Programm zum Zeichnen von Landeskarten.
[5] `http://www.ocad.com/`.
[6] Fachgruppe für Computergrafik der Universität Bern.

Teil I

Bekannte Gebäudemodellierungsansätze

Kapitel 2

Einführung

Dieses Kapitel gibt eine Einführung in die computergestützte Gebäudemodellierung und deren Motivation. Ausserdem wird der grobe Modellierungsvorgang und die Anwendungsmöglichkeiten der Gebäudemodellierung erklärt.

2.1 Motivation

Die Gebäudemodellierung wurde von Architekten mit dem Aufkommen der Computergrafik in den 50er Jahren als Visualisierungshilfe entdeckt und von Informatikern als exemplarisches Anwendungsgebiet der Computergrafik dankbar aufgenommen. Bis anhin hatte der Architekt nur zwei Möglichkeiten um sein Bauprojekt zu präsentieren: Entweder er erstellte ein Modell aus Karton und Holz, oder er zeichnete Ansichtsskizzen des Bauprojektes.

Das Zeichnen solcher Ansichtsskizzen erfordert einen geübten Zeichner, der es versteht, einen Bauplan korrekt umzusetzen. Neben der Interpretation eines Planes muss der Zeichner auch mit Perspektiven, Schatten und Beleuchtung umgehen können, um ein möglichst realitätsnahes, korrektes Bild zeichnen zu können. Ausserdem muss eine ansprechende Ansicht und eine dem Bauprojekt angepasste Umgebung gewählt werden. Der Zeichner trägt eine gewisse Verantwortung für die Zufriedenstellung des Kunden, indem er eine möglichst genaue Ansichtsskizze des fertigen Bauprojektes anfertigen sollte. Nebst Fehlern, die beim Zeichnen entstehen können, steht dem Zeichner eine grosse Wahlfreiheit zu, die den Eindruck vom Bauprojekt stark beeinflussen. Hier können natürlich Konflikte entstehen, wenn das Bauprojekt beschönigend oder überspitzt dargestellt wird. Beispielsweise interessieren sich die Nachbarn einer Überbauung für deren Schatten und möchten gerne wissen, wie stark sie davon betroffen sind. Ein korrekte Studie des Schlagschattens der Überbauung kann Klarheit schaffen. Ein gewichtiger Nachteil der Ansichtsskizze ist, dass für jede neue Ansicht eine neue Skizze angefertigt werden muss. Für den Kunden kann dies zu einem langwierigen Prozess führen, wenn er sein Bauprojekt aus möglichst allen Winkel betrachten möchte.

Die zweite Möglichkeit, das Erstellen eines Modells aus Karton und Holz, ist oft arbeitsintensiv und schwierig. Das Modell gibt aber je nach Detaillierungsgrad das Grundlegende des Bauprojektes recht gut wieder. Der Modellbauer versucht, das Modell dem Massstab entsprechend zu detaillieren. Bei zu gross gewähltem Massstab wirkt die Ausarbeitung der

einzelnen Bauten eher dürftig, und Verarbeitungsungenauigkeiten machen sich bezüglich der Grösse der Bauten bemerkbar, da solche Fehler massstabunabhängig sind. Für kleine Massstäbe steigt der Arbeits- und Materialaufwand und so auch die Kosten. Ein fertiggestelltes Modell gibt dem Kunden die Möglichkeit, sein Bauprojekt von allen Seiten zu betrachten. Der Kunde kann mit der Beleuchtung spielen oder, falls vorgesehen, Objekte im Modell verschieben. Nachteilig ist neben dem grossen Arbeitsaufwand auch, dass sich der Kunde nicht frei massstabgetreu durch das Modell bewegen kann.

In den Anfängen verstand man unter computergestützter Gebäudemodellierung einfachstes geometrisches Modellieren von Gebäuden mit einfachsten 3D-Objekten. Mit den damaligen Hard- und Softwaremöglichkeiten waren Visualisierung als Wireframe (Abb. 2.1) nicht unüblich, und die Manipulationsmöglichkeiten waren nicht gerade üppig. Hier sei

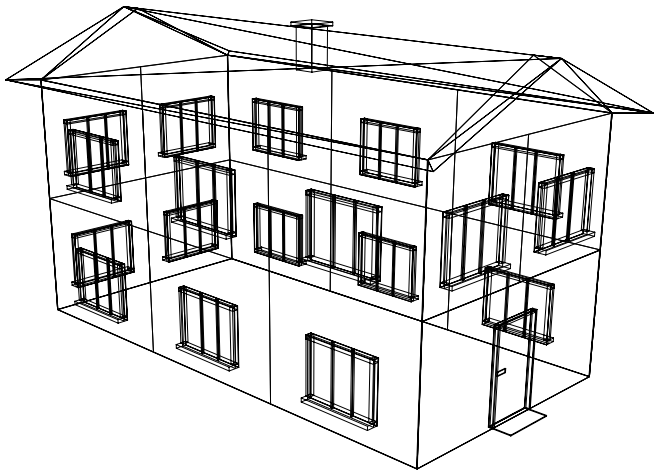

Abb. 2.1: Haus als Wireframe.

auf einen der ersten interaktiven 3D-Editoren, SKETCHPAD, von Ivan Sutherland[Suth 63] 1963 am MIT entwickelt, verwiesen, der zwar mit Constraints arbeitete, aber als Ausgabe nur Wireframes kannte. Trotzdem war er für viele Architekten eine echte Hilfe, da sie sich das Zeichnen der Ansichtsskizzen und das Bauen von Modellen ersparen konnten. Ausserdem gab es auch Architekten, die Szenen nachmodellierten, um grössere Reisen zu umgehen. Heute umfasst die Gebäudemodellierung den Vorgang von der Eingabe bis zur Ausgabe und kennt nicht nur das klassische, geometrische Modellieren. Die neuesten Ansätze bedienen sich Methoden der Bildverarbeitung, Bilderkennung, künstlichen Intelligenz und Rekonstruktion von 3D-Daten anhand von 2D-Bildern. Damit ist die Gebäudemodellierung nicht mehr ein reines Teilgebiet der Computergrafik, sondern ein Forschungsgebiet, an dem sich Informatiker verschiedener Gebiete wie auch Architekten beteiligen. Ausserdem werden die Eigenheiten von Gebäuden mit den neusten Ansätzen immer intensiver genutzt, um so die Effizienz zu steigern und schliesslich den Benutzer zu entlasten.

Seitdem es neue Ansätze gibt und die Hard- und Software solche Fortschritte gemacht haben, wird heute die Gebäudemodellierung nicht mehr ausschliesslich vom Architekten

eingesetzt. Der Raumplaner beispielsweise benutzt heute den Computer, um die Nutzung seiner Zonen zu visualisieren oder um das zeitliche Verhalten der Stadt zu dokumentieren, indem er mit einem 3D-Modell arbeitet, das die Region oder die Stadt repräsentiert. Der Archäologe wiederum bedient sich der Gebäudemodellierung, um schon längst zerstörte Bauten zu rekonstruieren. Die Spielprogramme und Simulatoren setzen immer mehr Häuser oder Gebäudekomplexe ein, um Szenen realistischer wirken zu lassen. Die hier gestellten Hauptanforderungen an die Gebäudemodellierung ist die Visualisierung in Echtzeit und die Interaktionsmöglichkeiten. Seit dem Aufkommen von *virtual reality* ist die Modellierung von Häusern, Gebäuden oder auch grösseren Gebäudekomplexen nicht mehr wegzudenken. Die modellierten Gebäude dienen als Treffpunkt von Benutzern von *virtual reality* und spielen eine ähnlichwichtige Rolle wie im realen Leben.

Die genauere Beschreibung des Modellierungsablaufes in den drei folgenden Unterkapiteln bezieht sich ohne Einschränkung der Allgemeinheit auf einzelne Gebäude.

2.2 Der Modellierungsablauf

Das geometrische Modellieren kann als Ablauf aufgefasst und mit einem verallgemeinerten Datenflussmodell (Abb. 2.2) beschrieben werden. Das Datenflussmodell wird anhand des gewählten Modellierungsansatzes entsprechend verfeinert und ausgebaut, ohne aber den grundlegenden Ablauf bestehend aus Eingabe, Szene und Ausgabe zu verändern.

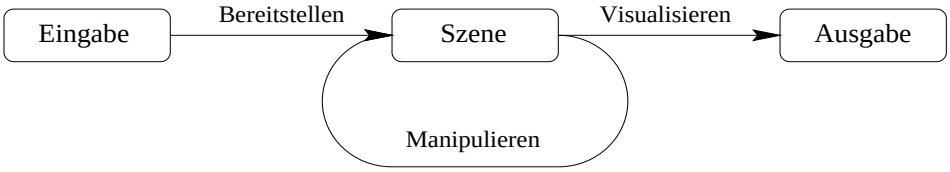

Abb. 2.2: Der Modellierungsablauf.

Die Eingabe befasst sich mit den Daten, die das Gebäude beschreiben, definieren oder spezifizieren. Die Eingabedaten liegen oft in unterschiedlichsten Formen und Formaten vor. Mit der Bereitstellung der Eingabedaten werden die Daten interpretiert und aufbereitet, so dass anschliessend die Szene repräsentiert und erzeugt werden kann. Das Aufbereiten der Daten ist nicht immer eine einfache Konversion, sondern oft ein langwieriger, iterativer Prozess. Ist die Szene modelliert, kann diese abhängig von der Repräsentation manipuliert oder verfeinert werden. Das Verfeinern der Szene ist oft ein iterativer Vorgang, deshalb auch die Schleife im Datenflussmodell. Zum Schluss muss die Szene ausgegeben werden. Die Visualisierung geschieht normalerweise mit Hilfe einer Renderingtechnik, die die Szene abbildet. Die Szene wird typischerweise als 2D-Bild ausgegeben, aber es ist auch denkbar, dass die Ausgabe von einer weiteren Applikation als Eingabe verwendet und eine andere Ausgabeform gewünscht wird.

2.3 Eingabe und Bereitstellung der Daten

Wie schon oben erwähnt kann die Eingabe auf verschiedenste Arten geschehen. Die verarbeitbaren Eingaben hängen vom gewählten Modellierungsansatz und den zur Verfügung

stehenden Hilfsmitteln ab. Einen kurzen Überblick über mögliche Eingabeformen gibt die folgende Aufzählung:

- Eingabe der Koordinaten und Platzierung von geometrischen Objekten durch den Benutzer.
- Bibliothek und CAD-Datei.
- Bauplan des Gebäudes (Abb. 2.3).
- Texturen.
- Bilderserie oder Videofilm mit unterschiedlichsten Ansichten des Gebäudes.
- Photogrammetrische Bilder[1].
- Tiefenbilder.
- Beschreibungen von Gebäuden in Textform.

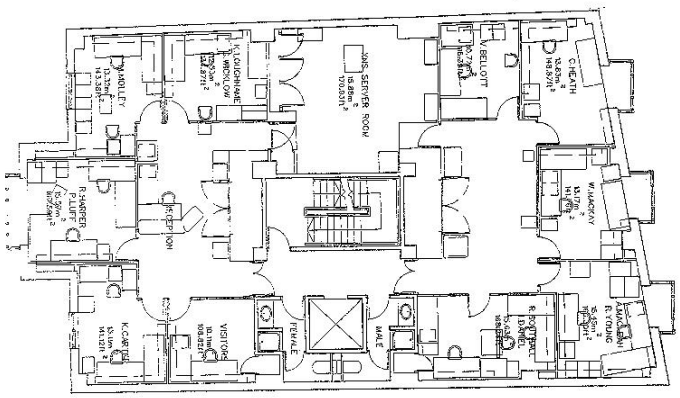

Abb. 2.3: Bauplan eines Hauses.

Die Bereitstellung der Daten beansprucht den Benutzer und den Computer unterschiedlich. Der Benutzer kann seine Szene mit vielfältigen Eingabegeräten (*input device*) modellieren. Die direkte Eingabe von geometrischen Objekten kann für den Benutzer für grössere Szenen zu einem langwierigen und monotonen Prozess werden. Ausser der Geometrie muss der Benutzer auch die Lage und die Ausrichtung der einzelnen Objekte definieren. Deshalb kennen heutige Modellierungsprogramme die hierarchische Modellierung und ausgereifte Positionierungshilfen. Zusätzlich werden Bibliotheken eingesetzt, um die Modellierung zu erleichtern. Neben geometrischen Objekten spielen Texturen bei der Eingabe ein wesentliche Rolle, da man sich so das Modellieren der kleinsten Objekte ersparen kann und gleichzeitig den Realitätsgrad steigert. Wesentlich einfacher gestaltet sich das Aufbereiten der Daten, falls die Daten schon als fertige CAD-Datei vorliegen. Die Daten müssen dann korrekt gelesen und umgesetzt werden. Die optimale Konversion zwischen

[1] Stereoskopische Aufnahmen.

den Formaten kann schwierig und aufwendig sein. Im schlimmsten Fall kann der Datenträger gar nicht mehr gelesen werden, da die Hardware nicht mehr zur Verfügung steht. Baupläne dagegen haben den Vorteil, dass diese gescannt und anschliessend vektorisiert werden können. Damit stehen alle möglichen Datenformate zur Verfügung, ohne auf aufwendige Konvertierungsprogramme ausweichen zu müssen. Ein gewichtiger Nachteil, der sowohl bei der Konversion von Daten, als auch bei der Digitalisierung von Bauplänen meistens unumgänglich ist, ist der Verlust der Struktur der Daten. Die Daten sind dann eine Ansammlung von Punkten, Linien und Flächen, die als Ganzes das Gebäude oder den Bauplan (Abb. 2.3) repräsentieren. Dieser Nachteil ist in dieser Phase tragbar, kann aber für Manipulation des Gebäudes ins Gewicht fallen.

Für die Bereitstellung der Daten für bildorientierte Eingaben, wie Bilderserien, Photogrammetrische Bilder und Tiefenbilder, werden Methoden der Bildverarbeitung, der Bilderkennung, der künstlichen Intelligenz oder der Rekonstruktion von 3D-Daten anhand von 2D-Bildern verwendet. Diese Techniken versuchen mit möglichst wenigen Benutzeranfragen aus den Bildern die relevanten Informationen zu finden, um die Szene anschliessend aufbauen zu können. So wird der Benutzer stark entlastet, dafür steigt der Rechenaufwand stark an. Die Geometrie des Gebäudes kann mit den neusten Ansätzen sehr genau bestimmt werden, auch wenn die verwendeten Algorithmen sich oftmals als aufwendig und instabil entpuppen oder unerwünschte Lösungen liefern. Hier muss der richtige Tradeoff zwischen Benutzer- und Rechenaufwand gefunden werden.

2.4 Repräsentationsmodelle und ihre Strukturen

Nach der Aufbereitung der Eingabedaten kann die Szene mit dem Gebäude verändert oder verfeinert werden. Es wird zwischen zwei grundlegenden Repräsentationsmodellen unterschieden. Der erste Ansatz ist vektororientiert und baut auf geometrischen Objekten auf, der zweite ist bildorientiert und repräsentiert die Szene mit Tiefenbildern (*depth maps*).

Der erste Ansatz ist aus der klassischen geometrischen Modellierung [Abra 91] bekannt. Die Szene wird normalerweise mit einem hierarchischen Aufbau repräsentiert und aus einzelnen Objekten aufgebaut. Die geometrischen Objekte können mit entsprechenden Editoren manipuliert werden, indem Koordinaten oder Parameter der Objekte verändert werden. Die Ansicht der Szene und die Beleuchtung, die für die Visualisierung wichtig sind, können hier eingestellt werden. Ausserdem können Farben oder Texturen zugeordnet werden, falls dies nicht schon bei der Bereitstellung der Daten geschehen ist. Die Möglichkeiten der Manipulation der Objekte hängt stark von der Struktur der Szene ab. Eine strukturlose Szene, die beispielsweise nur aus Dreiecken besteht, ist nur schlecht editierbar, da die Dreiecke im schlechtesten Fall einzeln bearbeitet werden müssen. Solche Szenen bedürfen einer Restrukturierung[2], die durch Zusammenfassen der Daten zu grösseren Objekten oder durch Ersetzen der Daten durch wenige, umfassende Objekte geschehen kann.

Der bildorientierte Ansatz benutzt Tiefenbilder. Zu jedem Bild ist die Kameraposition im WKS[3] mit Brennweite und die Tiefe der einzelnen Pixel zum Bild bekannt. Damit sind die Punkte im WKS eindeutig lokalisierbar. Die Ansicht wird durch Setzen einer virtuellen Kamera definiert und der Benutzer kann ein einfaches Beleuchtungsmodell wählen.

[2]Das Restrukturieren, Abstrahieren ist zur Zeit ein aktuelles Forschungsgebiet der Computergrafik, das durch die durch Digitalisierung entstehenden grossen Datenmengen motiviert wird.

[3]Weltkoordinatensystem.

Ausserdem können teilweise auch zusätzliche geometrische Objekte eingefügt werden, um beispielsweise einen Untergrund zu definieren. Das Verfeinern der Szene ist nur schlecht möglich, dazu müssen zusätzliche oder detailliertere Bilder und ihre Tiefenbilder eingesetzt werden.

Es gibt auch Modellierungsverfahren, die mit beiden Repräsentationsmodellen gleichzeitig arbeiten. Die Tiefenbilder dienen der teilweisen Rekonstruktion der Geometrie der Szene und mit dem groben geometrischen Modell der Szene können aus den Bildern die nötigen Texturen gewonnen werden [Debe 96]. Die Visualisierung übernimmt dann das Zusammenführen (*merge*) der beiden Modelle.

2.5 Visualisierung und Ausgabe

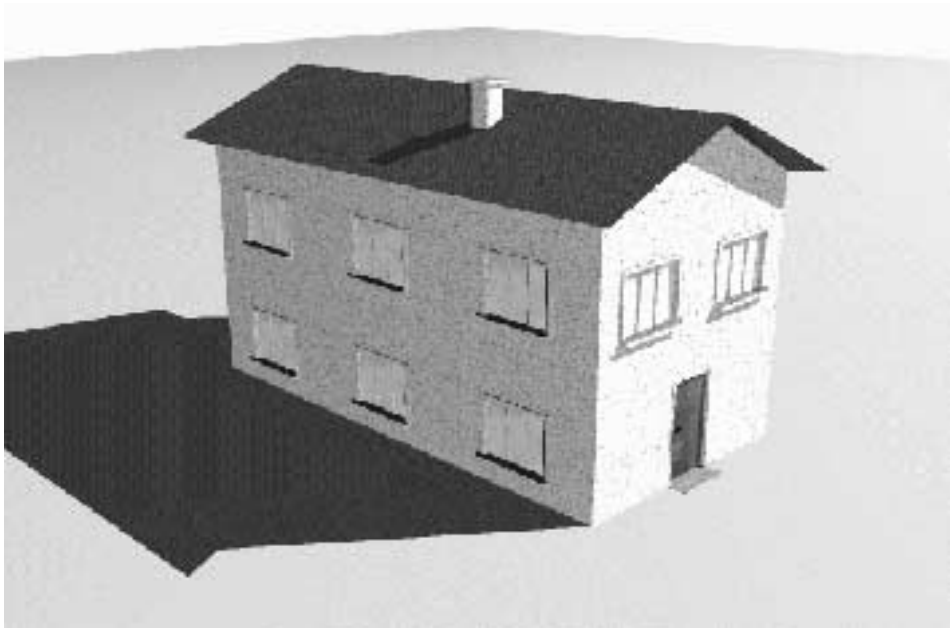

Abb. 2.4: Haus mit Schatten und Texturen.

Die Visualisierung bildet die eben erzeugte Szene ab. Übliche Visualisierungsapplikationen erzeugen ein normales 2D-Bild, aber es sind auch andere Ausgabeformen denkbar. Unter CIM [4] beispielsweise kann das modellierte Produkt automatisch erzeugt werden.

Das Abbilden der Szenen beruht auf Renderingalgorithmen, die die Repräsentation der Szene interpretieren und entsprechend abbilden können. Bei Tiefenbildern wird durch Rückprojektion der Bilder in den Bildraum (*imagewarping*) und unter Berücksichtigung einer einfachen Beleuchtung das Bild erzeugt. Geometrische Objekte können mit üblichen Techniken wie Z-Buffer, Wireframe, Radiosity, oder Raytracing abgebildet werden. Dabei werden Texturen und gewählte Beleuchtungsmodelle wie Phong [Pho 75], Gouraud Shading [Gou 71] oder Whitted [Whi 80] unterschiedlich berücksichtigt. Je nach Anwendungsgebiet wird man eine entsprechende Technik einsetzten. Wireframe wird oft für interaktives Editieren eingesetzt. Es eignet sich auch für Animationen in Echtzeit, weil der

[4] **C**omputer **I**ntegrated **M**anufacturing.

Rechenaufwand im Vergleich zu anderen Renderingtechniken klein ist. Für Präsentationen oder Filme sind aufwendigere Techniken wie Raytracing oder Radiosity unerlässlich, um eine möglichst fotorealistische Ausgabe zu erzielen. Texturen spielen hier eine wesentliche Rolle, denn sie erlauben mit kleinem Aufwand kleine Strukturen und geometrische Details wiederzugeben. Diese Techniken verursachen aber einen massiv grösseren Rechenaufwand.

Kapitel 3

Geometrisches Modellieren

Dieses Kapitel befasst sich mit dem geometrischen Modellierungsansatz, der dem klassischen geometrischen Modellieren [Abra 91, Bieri 95] aus der Computergrafik entspricht.

3.1 Der Modellierungsablauf

Die Abbildung 3.1 gibt einen Überblick über den Modellierungsablauf, der aus dem klassischen geometrischen Modellieren stammt. Das Manipulieren der Szene ist hier weggelassen worden, da es mit dem Bereitstellen der Daten zusammenfällt.

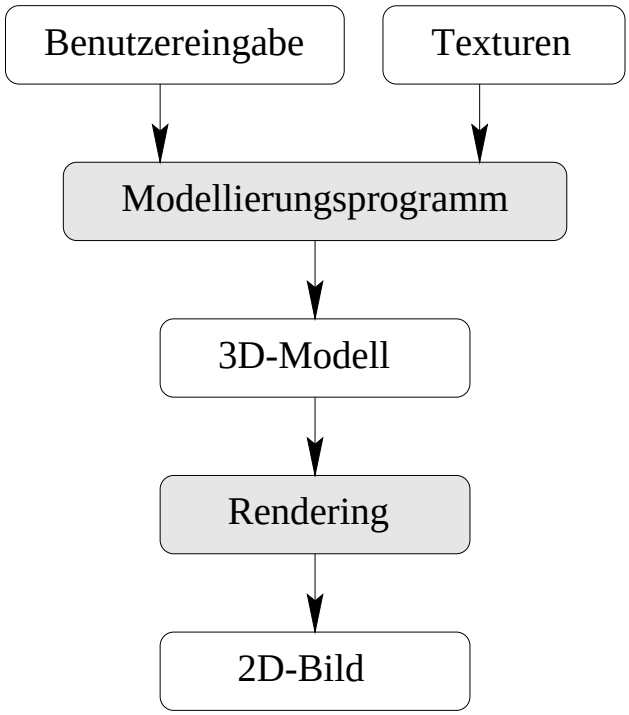

Abb. 3.1: Der Modellierungsablauf.

Die Eingabe besteht aus der Benutzereingabe und den Texturen, wobei auch das Importieren von geometrischen Daten dazugerechnet wird. Die Bereitstellung der Daten wird

von einem Modellierungsprogramm übernommen. Die Szene wird als 3D-Modell repräsentiert und mit einem Renderingprogramm als 2D-Bild visualisiert.

3.2 Eingabe und Bereitstellung der Daten

Die heutigen Eingabemöglichkeit (*input devices*) des Benutzers sind vielfältig und beschränken sich nicht mehr auf die Tastatur, wie in den Anfängen des Computers. Die Benutzereingabe geschieht normalerweise interaktiv, ausser der Benutzer editiere seine Szene via einer Datei, die die Szenenbeschreibung enthält. Interaktive Eingabegeräte verlangen das Zeigen und Positionieren, die beiden grundlegenden Operationen, um dem Zeichnen mit Bleistift nahe zu kommen. Der Benutzer sieht dann fortlaufend seine Szene, die sich entsprechend seinen Eingaben verändert. Neben einer Vielzahl von Eingabegeräten dominiert heute die Maus, die eine relative Bewegung ausgibt, und das Tablett (*digitizer*), das eine absolute Position ermittelt. Ausserdem ist der Lichtgriffel (*light pen*) beliebt, da man direkt auf den Bildschirm zeichnen kann. Eine Alternative dazu sind die noch etwas umständlichen und gewöhnungsbedürftigen 3D-Eingabegeräte (*3D input devices*), die virtuelles Editieren ermöglichen. Beim Datenhandschuh (*dataglove*) beispielsweise sieht der Benutzer seine virtuelle Hand in der Szene. Die virtuelle Hand bewegt sich entsprechend der Hand des Benutzer und erlaubt, sich in der Szene zu orientieren und diese zu manipulieren. Diese 3D-Eingabegräte werden im Moment noch hauptsächlich bei *virtual reality* Anwendungen eingesetzt.

Als Eingabe können auch Ausgaben von anderen Applikationen benutzt werden. Die meisten heute auf dem Mark erwerblichen Modellierungsprogramme, wie CAD-Programme, können direkt die Daten vom Scanner einlesen oder verstehen fremde Formate. Man spricht hier oft von den Importiermöglichkeiten einer Applikation, einer immer wichtiger werdenden Funktionalität. Die Importiermöglichkeiten definieren die Schnittstelle zur Aussenwelt und ermöglichen auf schon vorhandene Datenbestände zurückzugreifen.

Die Bereitstellung der Daten bei Eingabegeräten besteht darin, dass das Eingabegerät korrekt interpretiert wird und dann die gewünschte Operation ausgeführt wird. Heutige Modellierungsprogramme kennen verschiedene Modi Benutzereingaben zu interpretieren. Eine Mausbewegung beispielsweise kann in einer wählbaren Ebene in der Szene umgesetzt werden, um so Objekte entlang der gleichen Ebene verschieben zu können. Solche Funktionalitäten sind bei 2D-Eingabegeräten wichtig, um die 3D-Position in der Szene vollständig zu definieren.

Das Importieren von Daten verlangt eine Interpretation und Konversion der Daten. Die Konversion ist wie schon erwähnt nicht immer einfach und ist in den meisten Fällen nicht ohne Datenverlust möglich. Bei geometrischen Objekten kann es sein, dass ein Objekt nicht umgesetzt werden kann und man auf mehrere einfachere Objekte ausweichen muss. Die Szene verliert an Abstraktion, und ein Teil der Struktur geht verloren. Eine Freiformfläche beispielsweise, die durch Dreiecke repräsentiert wird, verliert ihre ursprüngliche Struktur. Probleme entstehen auch beim Manipulieren, denn das Objekt hat die ursprünglichen Parameter und Definitionen verloren und lässt sich nicht mehr so leicht editieren. Digitalisierte Daten, die von einem Scanner oder Framegrabber stammen können, sind oft strukturlos und speicherintensiv. Die Daten bedürfen einer Restrukturierung, um die

Handbarkeit der Daten zu steigern, und um die Datenmenge zu reduzieren. Die Restrukturierung geschieht durch bestmögliches Ersetzten der Daten durch umfassendere, abstraktere Objekte. Dieser Vorgang ist aufwendig und nicht trivial. Das Problem lässt sich etwa mit der Interpolation von Daten in der Mathematik vergleichen.

3.3 Repräsentationsmodelle und ihre Strukturen

Das Repräsentationsmodell bestimmt die Definitionsweise der geometrischen Objekte. Bei den klassischen Modellen, dem Draht-, dem Flächen- oder dem Volumenmodell, werden die Objekte mit den Informationsmitteln Punkt, Linie, Fläche und/oder Volumen definiert. Die vier Repräsentationsmodelle kennen einen hierarchischen Aufbau der Szene, der auf der Segmentierungstechnik beruht. Die Segmentierung erlaubt das Zusammenfassen von einzelnen Objekten zu einem einzigen Objekt. Für die Gebäudemodellierung sind hauptsächlich die klassischen Modelle interessant.

Drahtmodell

Das Drahtmodell (*wireframe*) repräsentiert die Objekte mit Punkten und Linien. Dieses Repräsentionsmodell ist im allgemeinen unvollständig, weil Flächen und Volumina nicht eindeutig sind. Deshalb kommt als Ausgabeform nur Wireframe in Frage. Dieses Modell war in den Anfängen der Computergrafik sehr beliebt, da es mit wenig Speicher auskommt, leicht zu implementieren und die Ausgabe einfach zu berechnen ist. Eine Szene wird aus losen, einzelnen Linien aufgebaut, doch heutige Modellierungsprogramme erzeugen die nötigen Linien um Primitivkörper, Freiformflächen oder Rotationsflächen zu repräsentieren.

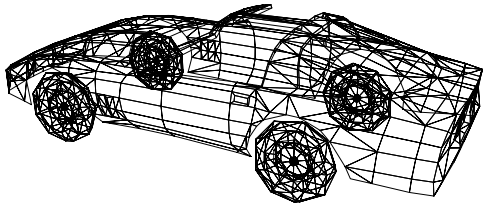

Abb. 3.2: Corvette als Wireframe.

Flächenmodell

Das Flächenmodell definiert die Objekte mit Punkten, Linien und Flächen. Dieses Modell kennt kein eigentliches Volumen. Körper müssen mit ihrem Rand, aus Flächen bestehend, modelliert werden, d.h. die Szene wird aus einzelnen Flächen aufgebaut. Die Modellierungsprogramme bieten heute nicht nur planare Polygone an, sondern können auch Kugeln, Polyeder, Zylinder, Freiformflächen, Sweepkörper[1] oder Rotationsflächen umsetzen.

[1] Körper, der mit Hilfe einer Sweepline definiert wird.

Bei der Umsetzung werden die dazugehörigen geometrischen Strukturen erzeugt. Der Modellierungsvorgang entspricht dem des Drahtmodelles, ausser dass die Objekte mit Flächen repräsentiert werden. Das Flächenmodell erlaubt eine vielfältige Modellierung von Objekten. Es können sogar Objekte repräsentiert werden, die keine globale Orientierung der Oberfläche besitzen. Als klassische Beispiele seien hier das Möbiusband und die Kleinsche Flasche (Abb. 3.3) erwähnt.

Abb. 3.3: Die Kleinsche Flasche.

Volumenmodell

Beim Volumenmodell, das eine vollständige und eindeutige Definition erlaubt, wird zwischen dem flächenorientierten (B-Rep[2]) und dem volumenorientierten Modell (CSG[3]) unterschieden. Im flächenorientierten Modell werden die Objekte, hier Körper genannt, mit Punkten, Linien, Flächen und Volumina definiert. Ein Körper definiert sich aus zueinander in Relation stehenden Flächen, die selber aus Kanten bestehen, die wiederum mit Eckpunkten definiert werden. Die Definition besteht aus der Topologie, den Relationen der Geometrie und den Koordinaten des Körper. Die Körper werden wie beim Flächenmodell durch ihren Rand repräsentiert, deshalb auch der Name *boundary representation*. Das Modellieren von Körpern ist ohne entsprechende Modellierungsprogramme aufwendig, da der Benutzer neben der Geometrie auch noch die Topologie selber definieren muss. Die Modellierungsprogramme übernehmen die Erzeugung der dazugehörigen Strukturen für primitve Körper und ermöglichen auch Objekte zu vereinigen oder miteinander zu scheiden. Viele B-Rep-Implementationen beschränken sich auf Polyeder, die nur aus planaren Polygonen bestehen, um komplexe Implementationsprobleme zu umgehen. Mit dieser Restriktion können die Körper mit den Euler-Operatoren von Baumgart [Bieri 95, FvDFH 90] modelliert werden. Diese Operatoren erlauben das einfache Konstruieren von komplexen Körpern und garantieren eine geometrisch wie auch topologisch korrekte Definition.

Die bis jetzt behandelten Repräsentationsmodelle sind zueinander auf- und abwärtskompatibel, was aber für das volumenorientierte Modell keinesfalls gilt. Es arbeitet mit einfachen Primitivkörpern wie dem Würfel, der Kugel oder dem Zylinder. Die Szene wird mit Verknüpfungen von Körpern konstruiert und wird als binärer Konstruktionsbaum (Abb. 3.4) repräsentiert. Die Blätter des Baumes sind Körper und die Knoten enthalten eine Verknüpfungsoperation wie Vereinigung, Schnitt oder Subtraktion. Da die Körper in Standardform definiert werden, können den Blättern eine Transformation bestehend

[2]Boundary Representation.
[3]**C**onstructive **S**olid **G**eometry.

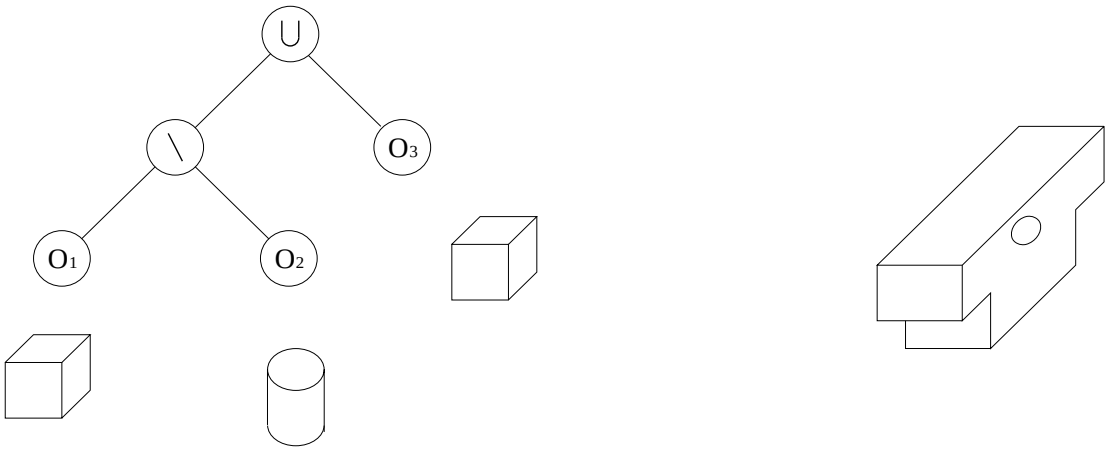

Abb. 3.4: Konstruktionsbaum und CSG-Körper.

aus Skalierung, Translation und Rotation zugeordnet werden. CSG erlaubt effizientes und einfaches Manipulieren der Objekte. Konstruktionsschritte können problemlos rückgängig gemacht werden. Die Konstruktion von Objekten ist nicht eindeutig und kann auf verschiedene Weise geschehen[4]. Oftmals benutzen Modellierungsprogramme CSG und B-Rep gleichzeitig, um Operationen im vorteilhafteren Modell ausführen zu können.

3.4 Visualisierung

Das Visualisieren besteht aus der Abbildung (*rendering*) der Szene in den Bildraum, der durch eine virtuelle Kamera definiert wird. Je nach Repräsentationsmodell können die Eigenschaften der Objekte, Beleuchtungsmodelle und Schatten umgesetzt werden. Die Ausgabeform der Szene wird vom Benutzer je nach Aufgabenstellung bestimmt.

Das Drahtmodell wird als Wireframe abgebildet, wobei man auf Schatten gänzlich verzichten muss. Beleuchtungsmodelle und Farben können nur beschränkt umgesetzt werden. Hidden-line ist mit dieser Repräsentation nicht möglich, weil sie keine Flächen und Körper kennt. Die schnelle Ausgabemöglichkeit spricht noch heute für Animationen in Echtzeit oder für interaktives Editieren und Modellieren. Ausserdem hat Wireframe den Vorteil, dass es die Geometrie der Szene vollständig wiedergibt und der Benutzer verdeckte Ecken und Kanten sehen kann (Abb. 2.1). Im Gegensatz dazu können das Flächen- und das Volumenmodell je nach Renderingtechnik Farben, Texturen, Beleuchtungsmodelle und Schatten mit unterschiedlichem Aufwand und Realitätsgrad umsetzen.

Bei der Visualisierung wird zwischen Einzelbildern und der Möglichkeit die Szene in Echtzeit zu erkunden unterschieden. Das Erkunden der Szene ist heute mit der entsprechenden Hard- und Software sehr populär geworden und erlaubt beispielsweise dem Architekten und seinem Kunden, sein zukünftiges Haus zu betreten und die Umgebung zu erkunden. Hier müssen aber gewisse Abstriche beim Realitätsgrad in Kauf genommen werden, um einen flüssigen Ablauf aufrecht zu erhalten. Schatten und aufwendige Beleuchtungsmodelle werden ignoriert. Texturen können mit entsprechenden Grafikkarten visualisiert werden, wenn auch nicht immer optimal. Oft erreicht man mit Gouraud Shading [Gou 71]

[4]Die Nullmenge kann beispielsweise auf beliebige Art erzeugt werden.

und diffusem Licht ansprechende Ergebnisse, ohne auf Interaktivität verzichten zu müssen.

Bei der Erzeugung von Einzelbildern spielt der Rechenaufwand eine untergeordnete Rolle. Der Benutzer möchte ein möglichst realistisches Bild [Ups 89] haben und setzt deshalb aufwendige Renderingtechniken ein. Typische Vertreter sind rekursives Raytracing, inverses Raytracing, Beamtracing und Radiosity, welche Beleuchtungsmodelle und Schatten umsetzen können. Neben der gewählten Renderingtechnik hängt der Realitätsgrad einerseits von der Geometrie der Szene, anderseits von der Beleuchtung, vom Schatten,

Abb. 3.5: Lugano mit primitiven Körpern und Texturen modelliert.

von den Farben und von den Texturen ab. Es hat sich aber gezeigt, dass eine korrekte Geometrie des Gebäudes eine grosse Rolle spielt und Fehler nur schlecht mit Texturen, Beleuchtung oder Schatten aufgefangen werden können. Neben den Farben, die via eines Scanners oder Framegrabbers bestimmt werden können, spielen die Texturen für das Berechnen von realistischen Bildern eine wichtige Rolle. Texturen werden verwendet um kleinere, aufwendige Objekte oder Strukturen zu repräsentieren. So kann man sich das Modellieren von Details ersparen. Der Einsatz von Texturen wirkt sich aber nicht nur vorteilhaft aus. Die Ausgabe wird verlangsamt[5] und kann bei zu umfassenden Texturen[6], die beispielsweise eine ganze Seite eines Gebäudes repräsentieren, weder abstrakt noch fotorealistisch wirken. Es entsteht ein Bruch ("Tapeteneffekt") zwischen der realen und der virtuellen Welt, der störend wirken kann. Ausserdem enthalten umfassende Texturen meistens unerwünschte Details wie beispielsweise Fussgänger oder den Schatten eines Baumes. Texturen sollten grundsätzlich für die Repräsentation von flachen Objekten eingesetzt werden, um grössere Projektionsfehler (Abb. 3.5[7]) zu vermeiden. Sehr geeignet sind Texturen bei Strukturen von Mauern oder Dächern.

[5]Viele heutige Grafik-Hardware-Beschleuniger sind für Texturen optimiert.

[6]Texturen, die einen grösseren Bereich eines Gebäudes abdecken und Objekte mit ungleicher Tiefe enthalten.

[7]Digital Architectural Photogrammetry and CAAD [StrMas 96].

3.5 Möglichkeiten und Grenzen

Mit der heutigen Modellierungssoftware ist bequemes und einfaches Modellieren möglich. Sie unterstützt eine Vielzahl von Objekten und erzeugt die dazugehörigen geometrischen Strukturen. Objekte können miteinander zu komplexen Objekten zusammengefasst werden. Diese wiederum können beliebig weiter zusammengestzt werden. Weiter erleichtern Bibliotheken die Modellierung von grösseren Szenen, indem der Benutzer auf schon modellierte Objekte zurückgreifen kann. Die Eingabegeräte unterstützen interaktives Editieren im dreideimesionalen Raum. Ausserdem erleichtern Editierhilfen wie Grid-Optionen oder Snap-Funktionen das Positionieren von Objekten im Raum.

Die aufwendigen Renderingtechniken, die Beleuchtungsmodelle und Texturen berücksichtigen, ermöglichen mit entsprechendem Rechenaufwand fotorealistische Ausgaben. Die Berechnungszeit für Einzelbilder ist noch gerade tragbar. Für Animationen in Echtzeit und mit aufwendigen Texturen müssen neue Lösungsansätze gesucht werden.

Nachteilig ist, dass beim geometrischen Ansatz die Objekte vom Benutzer positioniert werden müssen und, dass die Szene aus einzelnen Objekten zusammengefügt wird. Das Positionierungsproblem kann durch Importieren von geometrischen Daten, die beispielsweise von einem Scanner stammen, grösstenteils gelöst werden. Die geometrischen Daten liegen aber oftmals in einer losen Struktur[8] vor. Die schwachen Abhängigkeiten (*constraints*) zwischen den einzelnen Objekte kann bei nachträglichen Änderungen eine Reihe von Folgeänderungen mit sich bringen. Bei einer nachträglichen Änderung des Grundrisses eines Hauses müssen neben den betroffenen Fronten und Dächer auch Objekte wie Türen und Fenster entsprechend geändert werden. Es wäre wünschenswert, dass beim Verändern des Grundrisses sich auch die betroffenen Objekte ändern. Der Ansatz ist aus den oben genannten Gründen für grössere, komplexere Szenen arbeitsintensiv und unflexibel. Ausserdem erreicht man schnell die Speichergrenze des Computers und interaktives Modellieren wird schwierig. Hier sind Techniken nötig, die den Speicheraufwand reduzieren und die Ausgabe beschleunigen.

Texturen lohnen sich für realistische Präsentationen, auch wenn ein erhöhter Rechenaufwand nötig ist. Dem entgegen wirkt entsprechende Grafik-Hardware, die für Texturen ausgelegt ist. Neben der Umsetzung der Texturen müssen diese erst einmal aus Bildern gewonnen werden. Eine Textur sollte von unerwünschten Details bereinigt sein. Weiter muss die Textur blickpunktunabhängig und falls nötig fortsetzbar sein. Bei grösseren Texturen sollten keine Schlagschatten sichtbar sein. Sie sollten von der Tageszeit[9] unabhängig sein. Ausserdem ist zu beachten, dass die zu repräsentierende geometrische Struktur möglichst in einer Ebene liegt, um Projektionsfehler ("Tapeteneffekt") zu vermeiden. Da die Aufbereitung von Texturen, speziell wenn sie von willkürlichen Kameraaufnahmen stammen, relativ aufwendig werden kann, verzichtet man oft auf sie. Es werden dann prozedurale oder analytische [Teu 96] Texturen eingesetzt, die ohne Bitmaps auskommen und ihre Strukturen und Muster anhand einer Prozedur bzw. einer Formel erzeugen. Diese Texturen haben den grossen Vorteil, dass sie wenig Speicher beanspruchen und nicht an die Auflösung der Bitmap gebunden sind.

Die Einstellung der Beleuchtung ist nicht immer einfach, weil für die Beurteilung die gleiche Renderingtechnik benutzt werden muss wie für die Ausgabe. Hier muss man ne-

[8] Eine Liste mit ausschliesslich primitiven Objekten.

[9] Neben der Tageszeit beeinflussen auch die Jahreszeit und das Wetter die Bildaufnahmen.

ben guten Kenntnissen des Beleuchtungsmodelles auch auf Erfahrungswerte und Previews vertrauen.

Fotografien oder Ansichtsskizzen sind als Modellierungsgrundlage eher ungeeignet, weil die Geometrie nur schlecht bestimmt werden kann, was eher gegen das Nachmodellieren von Gebäuden ohne Bauplan spricht. Hier eignet sich eher der hybride Ansatz aus Kapitel 5.

Kapitel 4

Bildorientiertes Modellieren

In diesem Kapitel wird der bildorientierte Modellierungsansatz erklärt. Dieser Ansatz benutzt *stereo correspondence* für die 3D-Rekonstruktion von 2D-Bildern und ist rasterorientiert.

4.1 Der Modellierungsablauf

Die Abbildung 4.1 gibt einen Überblick über den Modellierungsablauf. Der hier vorgestellte Modellierungsablauf verarbeitet Bilder ohne Angabe der Kameraposition.

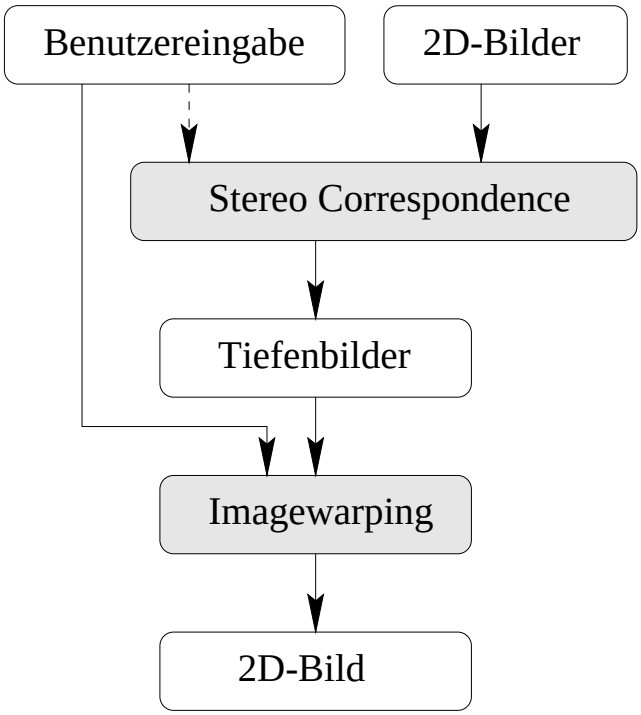

Abb. 4.1: Der Modellierungsablauf.

Die Eingabe besteht hauptsächlich aus einer Sequenz von 2D-Bildern. Die Bereitstellung der Daten wird mit Hilfe von *stereo correspondence* und anderen Methoden erledigt.

Die Szene, repräsentiert mit den Bildern und deren Tiefenbildern, wird mittels *imagewarping* als 2D-Bild visualisiert. Die Benutzereingabe beschränkt sich überwiegend auf die Einstellung der virtuellen Kamera und des Lichtes.

4.2 Eingabe der Daten

Der bildorientierte Ansatz kennt als Eingabeform grundsätzliche 2D-Rasterbilder (Abb. 4.2). Bei der Eingabe wird zwischen stereoskopischen Aufnahmen und Bildsequenzen (monoskopisch) unterschieden. Eine Bildsequenz wird normalerweise mit einem Rundgang um das Gebäude mit einer Video- oder einer Fotokamera gewonnen. Bei der Aufnahme muss beachtet werden, dass sich Einzelbilder paarweise überlappen, damit eine Rekonstruktion möglich ist. Auch wenn die Fotografie dem Video betreffend Bildqualität überlegen ist, hat das Video den Vorteil, dass die Bildsequenz das Gebäude vollständig abdeckt. Ausserdem

Abb. 4.2: Gebäude der Stadt Bern.

ist die Qualität von heute handelsüblichen Videokameras ausreichend. Denkbar sind auch zusätzliche Luftaufnahmen des Gebäudes, mit der Motivation, die Ausgabemöglichkeiten im Bereich des Daches zu verbessern. Auch wenn die Aufnahmen betreffend der Kameraeinstellungen, des Abstandes zum Gebäude und der Lichtverhältnisse frei wählbar sind, sollten sie unter etwa gleichen Bedingungen gemacht werden, um ansprechende Ergebnisse zu erzielen. Einmal in Besitz der Bildsequenz werden die Einzelbilder digitalisiert. Bei einem Video muss zusätzlich eine Wahl der Einzelbilder getroffen werden, hingegen kann jederzeit auf andere Einzelbilder zurückgegriffen werden. Bevor die Rekonstruktion durchgeführt wird, sollten die Bilder aufbereitet werden, da Aufnahmen mit einer Kamera aus physikalischen Sachzwängen immer etwas verzerrt oder unscharf werden. Um diese Verzerrungen zu korrigieren, können bekannte Methoden der Bildverarbeitung eingesetzt werden. Anschliessend werden die Bilder mit Filtern bearbeitet, um das Rauschen im Bild zu minimieren. Der Benutzer kann das Bild, falls nötig, von unerwünschten Details befreien.

Die Technik der stereoskopischen Aufnahme stammt aus der Photogrammetrie. Eine stereoskopische Aufnahme besteht aus zwei Aufnahmen mit konstanter Brennweite und Ausrichtung der Kamera. Ausserdem liegen die Aufnahmen in der gleichen Ebene, womit

die relative Orientierung der beiden Kameras bekannt ist. Die stereoskopische Aufnahme kann entweder mit einer relativen teuren Stereokamera gemacht werden, oder mit einer konventionellen Kamera nachempfunden werden, indem man zwei Aufnahmen unter Berücksichtigung der Voraussetzungen macht. Stereoskopische Aufnahmen werden mit den gleichen Methoden wie die Bildsequenzen verarbeitet.

Als Eingabe, die hier im Modellierungsablauf nicht dargestellt ist, sind auch die dazugehörigen Tiefenbilder denkbar, die die Tiefe jedes einzelnen Pixels wiedergeben. Damit wird der Rekonstruktionsvorgang überflüssig. Eine Möglichkeit bieten 3D-Sensoren, um Tiefenbilder ohne aufwendige Berechnungen zu akquirieren. Für die Erfassung der Tiefe gibt es etliche Ansätze, die sehr genaue Resultate liefern. Das wohl älteste Verfahren, das Triangulationsverfahren, beruht auf einer dreieckförmigen Anordnung eines Projektors, eines Empfängers und des zu erfassenden Objektes und erzielt Genauigkeiten im μm-Bereich (Abb. 4.3 und 4.4[1], IAM Universität Bern). Leider ist dieses Verfahren wie auch andere auf Laborbedingungen angewiesen und kann nur kleine Objekte erfassen. Auch Radar- und Lasersysteme sind für reale Gebäude ungeeignet, weil aufwendige und kostspielige Messtechniken benötigt werden.

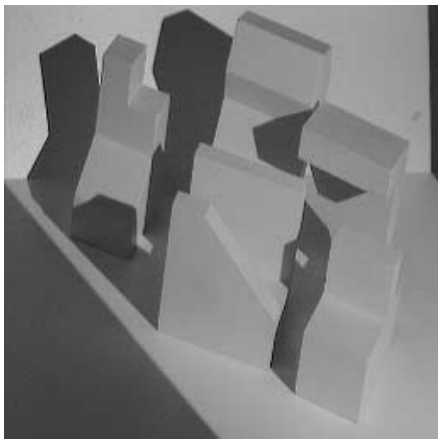

Abb. 4.3: Einfache Szene.

Abb. 4.4: Tiefenbild der Szene.

4.3 3D-Rekonstruktion von 2D-Bildern

Die Rekonstruktion besteht aus den drei folgenden Teilproblemen:

- Punkte in zwei oder mehreren Bildern identifizieren.
- Die relative Orientierung der Kameras bestimmen.
- Bestimmung der 3D-Koordinaten.

Stereo correspondence (Korrespondenzproblem) beschäftigt sich mit dem Problem, einen Punkt in zwei oder mehreren Bildern zu identifizieren. Beim Mensch wird dies mit zwei ähnlichen Bildern von der Hirnrinde übernommen und erlaubt uns das 3-dimensionale Sehen. Die Forschung hat gezeigt, dass dieses Problem mit dem Computer nicht einfach lösbar

[1] Je entfernter ein Punkt, desto heller ist er.

ist. Die meisten Ansätze erzielen sehr gute Resultate, wenn die Bilder aus ähnlicher Sicht stammen. Grosse Abstände zwischen den Kameras hat zur Folge, dass sich die Flächen und Kanten in den Bildern unterschiedlich verkürzen und damit das *matching* erschweren. Besonders das beliebte Korrelationsverfahren hat mit unterschiedlichen Verkürzungen Mühe. Bei allzu ähnlichen Bildern wird die Rekonstruktion der 3D-Koordinaten anfällig auf Ungenauigkeiten und Rauschen. Hier greift man normalerweise auf den Benutzer zurück und lässt sich einige Referenzpunkte angeben.

Für die Bestimmung der 3D-Koordinaten und der Orientierung der Kamera wird zwischen einer stereoskopischen Aufnahme und einer Bildsequenz unterschieden. Im ersten Fall kann ein lokales Koordinatensystem (Abb. 4.5 [Hild 97]) eingeführt werden, so dass sich die Kamerapositionen nur in den $x-$Koordinaten unterscheiden. Mit dem Abstand b

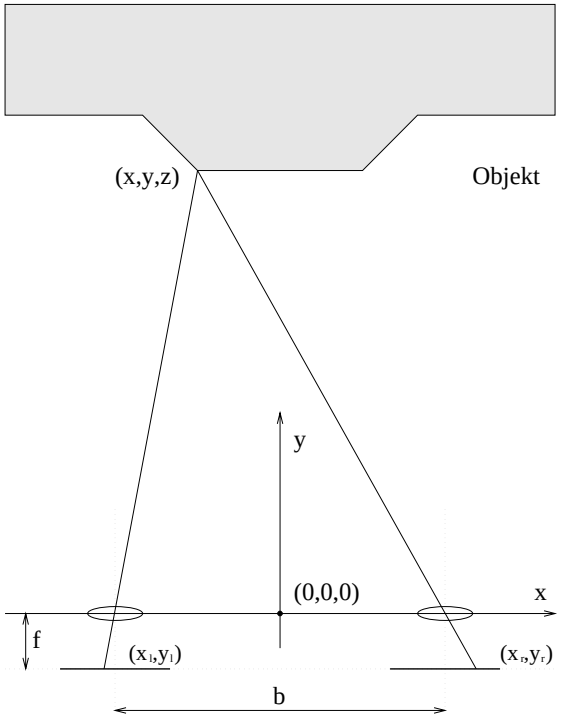

Abb. 4.5: Anordnung der Kamerasysteme.

zwischen den Kameras ist die relative Orientierung gegeben. Ausgehend von den Bildkoordinaten x_l und x_r des zu rekonstruierenden Punktes, der Brennweite f und dem Abstand b, kann der Punkt (x, y, z) im Raum bestimmt [Hild 97] werden:

$$x = \frac{b}{2}\frac{x_l + x_r}{x_l - x_r} \tag{4.1}$$

$$y = \frac{bf}{x_l - x_r} \tag{4.2}$$

$$z = \frac{b}{2}\frac{y_l + y_r}{x_l - x_r} \tag{4.3}$$

Die Genauigkeit der Erfassung des Punktes (x, y, z) hängt wesentlich von der Disparität ab, die durch die Differenz $(x_l - x_r)$ der Bildkoordinaten definiert ist. Die Disparität kommt

zweimal im Nenner vor, deshalb sollte sie möglichst gross sein. Sie wird geringer, falls sich das Objekt von der Kamera entfernt. Die Disparität wird gleichzeitig vom Abstand zwischen den Kameras und der Brennweite beeinflusst.

Bei Bildsequenzen ist die Rekonstruktion schwieriger, weil die Bilder in keiner Beziehung zueinander stehen. Diese Beziehung (relative Kameraorientierung und in der gleichen Ebene liegend) wird im stereoskopischen Fall vorausgesetzt. Hier ist man darauf angewiesen, dass es zwei Einzelbilder gibt, die aus ähnlicher Sicht aufgenommen worden sind, um korrespondierende Punkte sicherzustellen. In einem ersten Schritt müssen die Orientierungen der Kameras bestimmt werden. Dazu wird die Kamerabewegung zwischen zwei Bildern bestimmt. Diese Kamerabewegung wird als Transformation bestehend aus einer Rotation und einer Translation beschrieben. Mit Berücksichtigung der Orthogonalität der Rotationsmatrix und der Festlegung der Länge des Translationvektors kann die relative Orientierung mit 5 korrespondierenden Punkten bestimmt werden. Die Korrektheit wurde schon 1913 von Kruppa [Kru 13] in einem Satz bewiesen. Die Lösung des Problems beruht auf einem nichtlinearen Gleichungssystem, deshalb wird auf iterative Verfahren ausgewichen oder es werden zusätzliche Punkte hinzugenommen. Es gibt auch Methoden, die das Problem rekursiv mit mehreren Kameras lösen. Einmal in Besitz der relativen Orientierung können in einem zweiten Schritt die 3D-Koordinaten bestimmt werden. Die Berechnung erfolgt wie im stereoskopischen Fall, wobei noch die Kameratransformation berücksichtigt werden muss.

Hier sei noch erwähnt, dass die Rekonstruktion auch anhand des Schattens (*shape from shading*) oder der Textur möglich ist. Diese Verfahren haben im Gegensatz zu *stereo correspondence* einige strenge Voraussetzungen an die Bilder, die nur in seltenen Fällen erfüllt sind.

4.4 Visualisierung

Mit den Tiefenbildern ist nun die Tiefe jedes einzeln Pixels bekannt und mit der bekannten Kameraposition kann die Position im Raum bestimmt werden. Die Visualisierung geschieht dann durch Rückprojektion der Punkte in den neuen Bildraum. Dieses Renderingverfahren wird auch *imagewarping* genannt und man verzichtet auf Beleuchtungsmodelle, hat aber den Vorteil, dass der Aufwand pro Pixel konstant ist. Diese Eigenschaft ist für Anwendungen in Echtzeit wichtig, denn damit lässt sich die Renderingzeit voraussagen.

Aufwendigere Verfahren berücksichtigen diffuses Licht oder können zusätzliche geometrische Objekten verarbeiten, indem die Tiefenbilder in eine Raumzerlegung (*voxel*) überführt werden und mit den geometrischen Objekten zusammengeführt werden.

4.5 Möglichkeiten und Grenzen

Die Gewinnung von dreidimensionalen Koordinaten anhand von 2D-Bildern besitzt eine längere Tradition sowohl in der Photogrammetrie als auch in der Informatik. Der Grund dafür ist nicht zuletzt die erzielbare Genauigkeit, die bei bekannter relativer Orientierung sehr gross ist. Demgegenüber ist die Zuordnung von korrespondierenden Punkten in den verschiedenen Bilder nachteilig. Heutige *matching*-Verfahren arbeiten recht zuverlässig, falls die relative Orientierung bekannt ist und die Disparität nicht allzu gross ist. Proble-

matisch wird es mit grossen Disparitäten und strukturlosen Bildern, da versagen *matching*-Vefahren. Gegen eine kleine Disparität spricht aber die Bestimmung der 3D-Koordinaten, weil sich Ungenauigkeiten im Bild bei kleiner Disparität viel stärker auswirken. Die Bilder sollten deshalb vor der Rekonstruktion von Verzerrungen und Rauschen befreit werden. Ausserdem werden für kleinere Disparitäten grössere Bildsequenzen benötigt, d.h. Einzelbilder müssen sich ähnlich sein. Bei der Disparität sollte ein Mittelweg gefunden werden, so dass der Benutzer am Ende nur wenige Zuordnungen selber machen muss.

Der Hauptgrund für den Einsatz von bildorientierter Modellierung ist die schnelle Ausgabe, die unabhängig von der Komplexität der Szene auf einem konstanten Aufwand pro Pixel beruht. Die Leistung reicht oftmals für Echtzeit-Anwendungen aus. Diese Eigenschaft wird heute bei *virtual reality* Anwendungen benutzt. Die komplexe, geometrisch modellierte Szene wird mit einer aufwendigen Renderingtechnik aus verschieden Ansichten abgebildet. Dazu werden die entsprechenden Tiefenbilder bestimmt. In einem zweiten Schritt wird dann die Szene mittels *imagewarping* in Echtzeit visualisiert. Die Möglichkeit die Szene mit einer Raumzerlegung zu repräsentieren, ermöglicht vielfältige Weiterverarbeitungsmöglichkeiten und wird als gemeinsame Plattform vom geometrischen und bildorientierten Modellen eingesetzt. Die Raumzerlegung erlaubt einfaches Editieren, wie im 2D-Fall mit Rasterbildern.

Kapitel 5

Hybrides Modellieren

Dieses Kapitel beschäftigt sich mit dem hybriden Modellieren [Debe 96]. Der Ansatz wurde von P.E. Debevec, C.J. Taylor und J. Malik entwickelt und vereint das geometrische und das bildorientierte Modellieren. Die Abbildungen stammen aus dieser Arbeit.

5.1 Der Modellierungsablauf

Die Abbildung 5.1 gibt einen Überblick über den Modellierungsablauf.

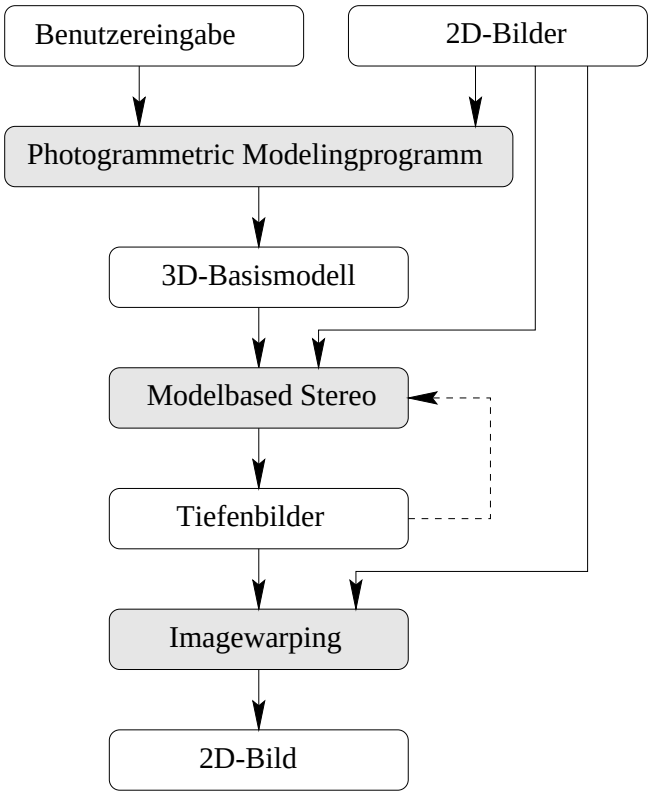

Abb. 5.1: Der Modellierungsablauf.

Die Eingabe besteht aus Benutzereingaben und 2D-Bildern. Die Bereitstellung der

Daten geschieht in zwei Schritten. In einem ersten Schritt wird ein grobes 3D-Modell erzeugt. In einem zweiten Schritt werden, geometrische Details extrahiert und die genauen Tiefenbilder bestimmt. *Imagewarping* visualisiert die Szene als 2D-Bild.

5.2 Eingabe der Daten

Der hybride Ansatz kennt wie der bildorientierte Ansatz Bildsequenzen als Eingabeform, verzichtet aber auf stereoskopische Aufnahmen. Die Bildsequenz kann mit einer handelsüblichen Video- oder einer Fotokamera akquiriert werden. Im Gegensatz zum bildorientierten Ansatz kann auf ähnliche Ansichten der Szene verzichtet werden, und die Rekonstruktion ist anhand einer kleinen Menge von Einzelbildern möglich.

5.3 Photogrammetrisches Modellieren (*Photogrammetric Modelling*)

Das photogrammetrische Modellieren beschäftigt sich mit der Gewinnung von 3D-Daten. In Abschnitt 4.3 wurde eine Methode vorgestellt, die einzelne Punkte im Raum rekonstruieren kann. Die hier vorgestellte Methode versucht die Gewinnung von einfachen Primitivobjekten und verfolgt das Ziel, ein grobes geometrisches Basismodell (Abb. 5.3)

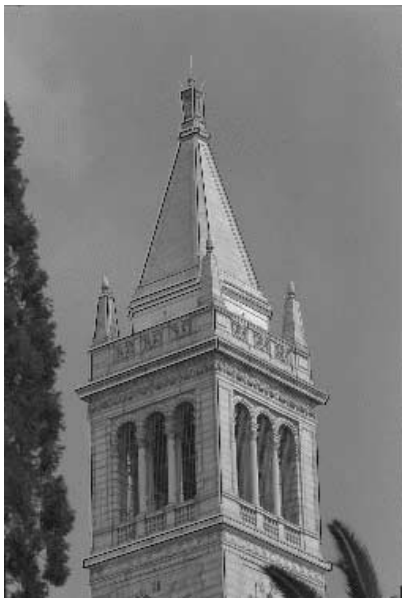

Abb. 5.2: Eingabebild.

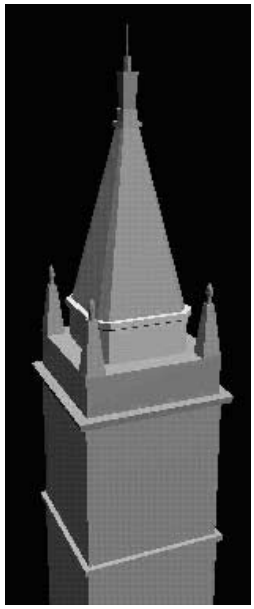

Abb. 5.3: Modell.

Abb. 5.4: Projektion.

Campanile, der Glockenturm von Berkeley.

des Gebäudes zu rekonstruieren. Die Gewinnung des Basismodells ist ein iterativer Vorgang. Die hier berücksichtigten Primitivobjekte sind Quader, Prismen und rechteckige Pyramiden. Die Objekte werden hier Blöcke genannt. Die Grösse eines Blockes wird mit der Höhe, Breite und Länge seiner assoziierten *boundingbox* repräsentiert (Abb. 5.5). Die

Lage und Ausrichtung wird mit den Parametern der allgemeinen Rotation und der Translation definiert.

Der Benutzer zeichnet die Kanten in den Bildern (Abb. 5.2) ein und ordnet die korrespondierenden Kanten einem entsprechenden Primitivobjekt im Modell zu. Mit der Zuordnung und der Berücksichtigung der geometrischen Struktur des Primitivobjektes kann die relative Grösse und Lage bestimmt werden. Durch Rückprojektion des Basismodells in die Bilder (Abb. 5.4) und einer einfachen Flächendarstellung der Szene kann der Benutzer die Genauigkeit der Geometrie jederzeit überprüfen. Der Benutzer kann die einzelnen Parameter der Blöcke teilweise oder vollständig bestimmen. So lassen sich beispielsweise die Höhe eines Blockes festlegen oder gleich grosse Blöcke zusammenfassen. Ausserdem kann der Benutzer Relationen zwischen den Blöcken einführen. Die Relationen ermöglichen einen hierarchischen Aufbau der Szene. Diese zusätzlichen Informationen (*constraints*) des Benutzers reduzieren den Freiheitsgrad der einzelnen Blöcke und vereinfacht die Rekonstruktion des Basismodells.

5.4 Repräsentationsmodell und seine Strukturen

Die Repräsentation des Basismodells ist flächenorientiert und hierarchisch. Das Basismodell wird mit einfachen parametrisierbaren Polyedern (Blöcken) modelliert und wird als Baumstruktur repräsentiert. Die Knoten enthalten die Blöcke und mit den Kanten werden die räumlichen Relationen zwischen den Blöcken definiert. Jeder Block besitzt eine minimale Anzahl von Parametern, um Grösse und Form zu bestimmen. Die Geometrie (Form) des Blockes wird relativ zur assoziierten *boundingbox* festgelegt. Die Grösse des Blockes bestimmt die *boundingbox*, bestimmt durch Länge, Breite und Höhe (Abb. 5.5).

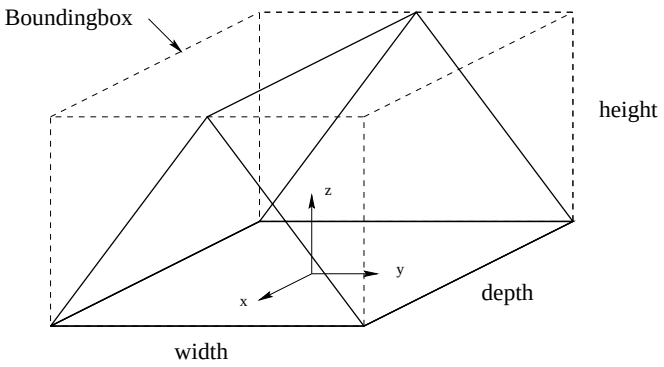

Abb. 5.5: Prisma mit *boundingbox*.

Die Relation (Anordnung) $g_i(X)$ eines Blockes zu seinem Vater wird mit einer Transformation bestehend aus einer Rotation R_i und einer Translation t_i definiert. Die Relation wird mit sechs Parametern festgelegt, wobei bei architektonischen Szenen die Relationen oftmals einfacher Natur sind und sich mit einigen wenigen Parametern vollständig bestimmen lassen. Die Rotation beispielsweise lässt sich auf eine Achse beschränken oder als konstant definieren. Die Translation kann auf gleiche Art wie die Rotation eingeschränkt werden. Ausserdem ist es möglich die Parameter mit Parametern des Vaters zu bestimmen, um relative Abhängigkeiten zwischen Vater und Sohn zu definieren. Beispielsweise kann die Grösse und die Lage des Daches abhängig zur Grösse des Stockwerkes festgelegt werden.

Das Dach ist dann vollständig bestimmt. Diese vom Benutzer definierten Abhängigkeiten und Beschränkungen haben zur Folge, dass der Freiheitsgrad kleiner wird.

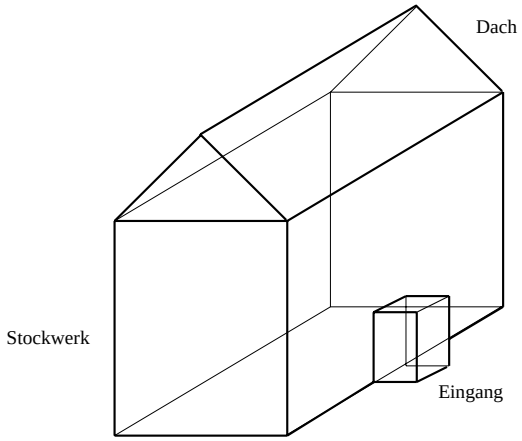

Abb. 5.6: Das geometrische Basismodell.

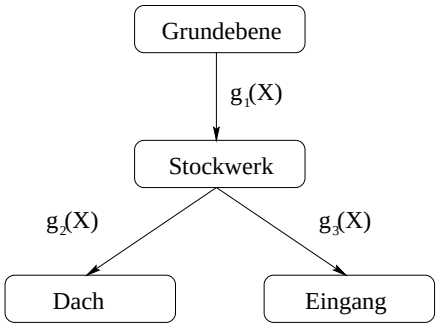

Abb. 5.7: Modellhierarchie.

Jeder Parameter der einzelnen Blöcke wird in einer Liste als symbolische Variable eingetragen (Ab. 5.8). Die Abhängigkeiten und Beschränkungen (*constraints*) werden mit Referenzen auf die Variablen in Form eines einfachen Termes definiert. Damit kann der Benutzer die meisten Symmetrien oder Beziehungen zwischen einzelnen Blöcken eines Gebäudes beschreiben. Beispielsweise wurde für den Glockenturm (Abb. 5.2) die vier kleinen Türme in der Form als identisch definiert, indem die Parameter der Form von drei Türmen auf den noch Verbleibenden referenzieren.

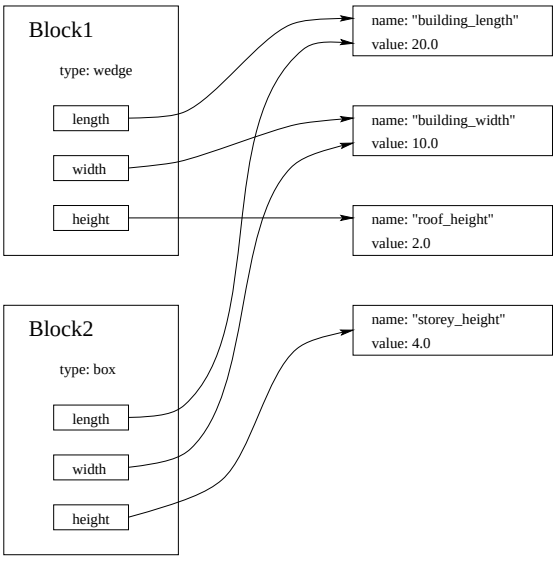

Abb. 5.8: Referenzen auf die Symbolliste.

Sind einmal die Blöcke und Relationen festgelegt, kann ein Punkte $P(X)$ eines Blockes im WKS wie folgt berechnet werden:

$$P_{WKS}(X) = g_1(X) \dots g_n(X) P(X) \tag{5.1}$$

Auch wenn die Blöcke relativ einfach sind, können die meisten Gebäude ansprechend nachmodelliert werden, weil Gebäude ähnliche Symmetrien aufweisen und oftmals aus Elementen aufgebaut sind. Die Blöcke sind abstrakt, weil auf die Modellierung von einzelnen Punkte und Kanten verzichtet wird. Die *constraints* erlauben eine dem Gebäude ähnliche Strukturierung, Aufbau des Modells.

5.5 Die Rekonstruktion

Der hier verwendete Rekonstruktionsalgorithmus beruht auf der Minimierung einer Fehlerfunktion $\mathcal{O}$, die als Summe $\mathcal{O} = \sum Err_i$ aller Disparitäten Err_i zwischen der projizierten Kante des Basismodells und der eingezeichneten Kante definiert ist. Disparität ist über eine Abstandfunktion $h(s)$ (Abb. 5.9) definiert. Die Projektion der Kante in die Bildebene ist durch die Kante im WKS und der Orientierung der Kamera bestimmt. Neben den freien Parametern des Modells werden mit jeder Kamera dem Gleichungssystem 6 neue freie Parameter hinzugefügt. Für die Minimierung von $\mathcal{O}$ ist die Methode von Newton-Raphson eingesetzt worden, die auf den Gradienten und die Hessesche Matrix angewiesen ist. Wegen der einfachen Definition der Fehlerfunktion Err_i, werden die Ableitungen symbolisch bestimmt, um eine maximale Genauigkeit zu erreichen.

Problematisch ist, dass die Fehlerfunktion hinsichtlich des Basismodells und der Kameraparameter nichtlinear ist und deshalb lokale Minima besitzen kann. Falls die Kameraparameter zufällig gewählt werden, kann nicht garantiert werden, dass der Algorithmus gegen das absolute Minimum konvergiert. Um dieses Problem zu umgehen, müssen Startparameter gefunden werden, die eine Konvergenz garantieren. In einem ersten Schritt werden die Rotationen der Kameras geschätzt, indem eine Fehlerfunktion minimalisiert wird. Die Schätzung der Rotationen ist normalerweise unabhängig von den Parametern des Basismodells und der Lage der Kameras, weil Gebäude viele verschiedene (horizontale und vertikale) Kanten besitzen. In einem zweiten Schritt werden die Translationen der Kameras und die Parameter des Basismodells geschätzt. Sind die Startparameter bestimmt, wird mit der nichtlinearen Minimierung weitergefahren. Die Praxis hat gezeigt, dass mit vernünftigen Startparametern etwa nach 10 Iterationen die projizierten Kanten und die eingezeichneten Kanten sich nicht mehr als um ein halbes Pixel unterscheiden.

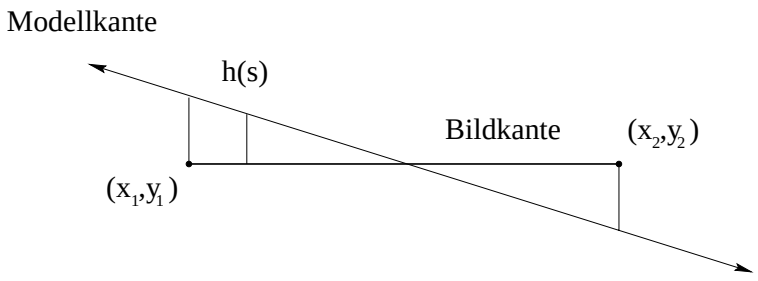

Abb. 5.9: Fehlerfunktion.

Mit dem Basismodell ist es nun möglich zusätzliche geometrische Details aus den Bildern zu extrahieren (*modelbased stereopsis*). Die Rekonstruktion beruht auf dem Prinzip von *stereo correspondence*, wobei hier vom Basismodell Gebrauch gemacht wird. Das klas-

sische *stereo correspondence*-Verfahren versucht durch Korrelation den korrespondierenden Punkt in den beiden Bildern zu finden. Das Verfahren jedoch versagt sobald sich die Ansichten allzu gross unterscheiden und sich Kanten und Flächen unterschiedlich verkürzen. Die unterschiedlichen Verkürzungen können nun mit Hilfe des Basismodells durch Rück-

Abb. 5.10: *Key image.*

Abb. 5.11: *Warped offset image.*

Abb. 5.12: *Offset image.*

projektion reduziert werden. Bei der Rückprojektion wird das *offset image* (Abb. 5.12) in das *key image* (Abb. 5.10) abgebildet, hier *warped offest image* (Abb. 5.11) genannt. *Stereo correspondence* wird auf die Bilder *key image* und *warped offest image* angesetzt und berechnet das Disparitätsbild (*disparity map*). Neben der Reduktion der unterschiedlichen Verkürzungen bringt *warped offest image* folgende angenehme Eigenschaften mit sich:

- Ein Punkt mit Disparität Null liegt auf dem Basismodell
- Das Disparitätsbild ist einfach in ein Tiefenbild zu konvertieren.
- Die Bestimmung des Tiefenbildes ist weniger sensibel auf Rauschen, weil die Ansichten sich wesentlich unterscheiden.
- Die Tiefenbilder können neben dem Extrahieren von zusätzlichen Details, auch direkt für die Visualisierung eingesetzt werden.

5.6 Visualisierung

Die Visualisierung der Szene besteht im Wesentlichen aus dem Texturieren der Szene. Die Texturen stammen aus der Bildsequenz und werden mit *view-dependent texture-mapping* umgesetzt. Die Bilder werden abhängig vom der aktuellen Betrachterposition (*virtual view*) auf die Szene projiziert. Diese Technik erzeugt realistische Bilder, falls das Modell die Geometrie möglichst genau wiedergibt und die Aufnahmen unter ähnlichen Bedingungen gemacht wurden. Die traditionellen Techniken ermöglichen das Texturieren mit einem einzelnen Bild.

Das Texturieren mit einem Bild entspricht einer einfachen Rückprojektion (*imagewarping*) des Bildes auf die Szene. Bei nicht-konvexen Objekten kann es möglich sein, dass es aus der aktuellen Kameraposition sichtbare Gebiete gibt, die aus der Sicht des Bildes verdeckt sind.

Normalerweise wiedergibt eine Aufnahme nur einen kleinen Teil des Gebäudes. Deshalb müssen für die Visualisierung der Szene mehrere Bilder für die Texturierung berücksichtigt werden. Falls das Pixel nur von einem Bild koloriert wird, kann es ohne weiteres mit herkömmlichen Renderingtechniken umgesetzt werden. Im Falle von mehreren Bildern,

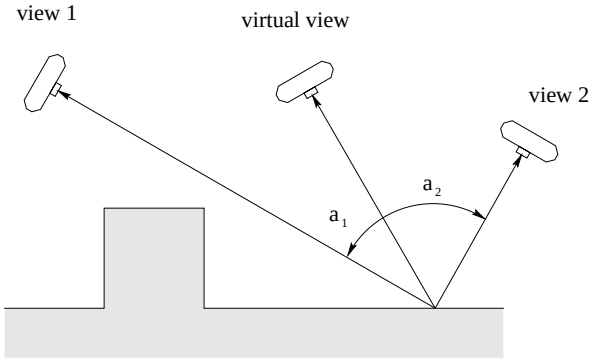

Abb. 5.13: Gewichtungsfunktion.

muss eine Auswahl zwischen den Bildern getroffen werden, bevor mit dem Rendering fortgesetzt werden kann. Setzt man voraus, dass die Aufnahmen unter gleichen Lichtverhältnissen entstanden sind und sie keine Reflexionen aufweisen, kann einfach dasjenige Bild ausgewählt werden, das der aktuellen Kameraposition am nächsten kommt.

Reflexionen und unmodellierte geometrische Details können im Überlappungsbereich unerwünschte Übergänge erzeugen. Deshalb wird eine Gewichtungsfunktion (Abb. 5.13) eingeführt, die die einzelnen Bilder abhängig von der aktuellen Kameraposition gewichtet. Weiter wird noch verlangt, dass das Gewicht eines einzelnen Bildes von der Mitte zum Rand stetig abnimmt und ausserhalb des Bildes Null ist. Ungewollte Details werden mit einer unbenutzten Farbe ausmaskiert und von der Gewichtungungsfunktion mit Null gewichtet. In der Praxis wird die Gewichtungsfunktion nur einmal pro Fläche ausgewertet, weil sich die Gewichte nur geringfügig verändern und das Resultat kaum beeinflusst wird. Auch wenn die Wahl der optimalen Gewichtungsfunktion nicht einfach ist und glatte Übergänge nicht immer garantiert werden können, werden die meisten Fehler abgefangen.

5.7 Möglichkeiten und Grenzen

Der hybride Ansatz wurde von den Entwicklern in FAÇADE umgesetzt und zeigt einen neuen Weg, um architektonische Szenen anhand von einigen wenigen Fotografien zu rekonstruieren. Die erste Komponente besteht aus einem einfachen photogrammetrischen Modellierungssystem, welches das Rekonstruieren der grundlegenden Geometrie der Szene erlaubt. Mit einer zweiten Komponente werden geometrische Details extrahiert. Für die Visualisierung werden die Bilder als Texturen des Modells wiederverwendet.

Das arbeitsintensive Positionieren und Ausrichten der einzelnen Objekte entfällt mehrheitlich, dafür muss der Benutzer Informationen über die Symmetrien und Eigenheiten der Szene in Form von Relationen und durch teilweise Definition der Parameter dem System mitteilen. Der Benutzer wird nicht jeglicher Eingaben entbunden und übernimmt das was der Computer nicht leisten kann: Die Interpretation der Szene.

Als vorteilhaft bei der Rekonstruktion des Basismodells hat sich gezeigt, dass die Objekte vollständig oder nur teilweise vom Benutzer definiert werden können. So kann die Anzahl noch freien Parameter klein gehalten werden. Der Glockenturm von Berkeley (Abb. 5.2) beispielsweise hat mit parametrisierbaren Objekten 33 Freiheitsgrade und ohne Parametrisierung deren 240. Falls mit losen Kanten gearbeitet wird, sind 2896 freie Parameter

zu bestimmen.

Auch wenn relativ einfache geometrische Objekte eingesetzt werden, ist eine recht genaue Modellierung möglich, denn die meisten Gebäude werden auch aus Elementen aufgebaut, die der *boundingbox* nahekommen. Zur Zeit gibt es schon Objekte, die das Modellieren von Bögen, Kuppeln und allgemeinen Rotationsflächen ermöglichen.

Die Aufbereitung der Texturen ist bis heute noch ein schwacher Punkt. Die Bilder verlangen bei der Aufnahme ähnliche Licht-, Schatten- und Wetterverhältnisse. Daneben ist die Auswahl der Bilder, die beim Texturieren (*view-dependent texture-mapping*) berücksichtigt werden sollen, schlecht gelöst. Die Bilder können auch unerwünschte Details enthalten, die die Visualisierung negativ beeinflussen.

Teil II

Ein objektorientierter Ansatz

Kapitel 6

Objektorientiertes Modellieren

Das folgende Kapitel behandelt das objektorientierte Modellieren. Der Ansatz baut auf der geometrischen Modellierung auf und bedient sich objektorientierter Techniken.

6.1 Motivation

Die im ersten Teil dieser Arbeit vorgestellten Modellierungsansätze basieren hauptsächlich auf einfachen geometrischen Objekten, wobei diese mittels Strukturierungsmöglichkeiten, wie Aggregaten, zu komplexeren Objekten kombiniert werden können. Diese komplexen Objekte können wiederum weiter zusammengesetzt werden, was die Modellierung von grösseren Szenen ermöglicht. Das Zusammensetzen, das auch das korrekte Positionieren beinhaltet, und die Definition der einzelnen geometrischen Objekte übernimmt der Benutzer, falls dies nicht vom gewählten Ansatz unterstützt wird. Dies entspricht einer *bottom-up*-Modellierung, die hier nur in beschränktem Masse die Eigenheiten von realen Objekten, hier Gebäuden, berücksichtigt.

Im Gegensatz dazu geht der hier entwickelte objektorientierte Modellierungsansatz von abstrakten Objekten aus, die ein ganzes Gebäude oder Teile davon repräsentieren. Diese abstrakten Objekte besitzen Eigenschaften, die Gegebenheiten oder Randbedingungen von realen Gebäuden so gut als möglich berücksichtigen sollen. Diese Eigenschaften parametrisieren teilweise die Geometrie und Topologie des Objektes und definieren die Abhängigkeiten zu einem Parentobjekt, falls diese existiert. Weiter müssen die Parentobjekte Strukturierungsmöglichkeiten kennen, die weit über das einfache Zusammenfassen von Objekten gehen, um die Unterobjekte untereinander korrekt abzustimmen. Ausserdem übernehmen die Objekte die konkrete Repräsentation mit primitiven geometrischen Objekten, in Abhängigkeit ihrer Eigenschaften und Definition. Damit ist es möglich, das Gebäude *top-down* zu modellieren, indem von einem abstrakten Gebäude ausgegangen und dieses mit passenden Unterobjekten sukzessive genauer modelliert wird.

Die Hauptmotivation der objektorientierten Gebäudemodellierung ist dem Benutzer die Modellierung eines vollständigen Gebäudes mit möglichst wenigen Parametern und Objekten zu ermöglichen, indem Eigenschaften von realen Gebäuden so weit als möglich berücksichtigt werden, ohne aber die Flexibilität der Modellierung wesentlich einzuschränken.

6.2 Der Modellierungsablauf

Die Abbildung 6.1 gibt einen Überblick über den Modellierungsablauf.

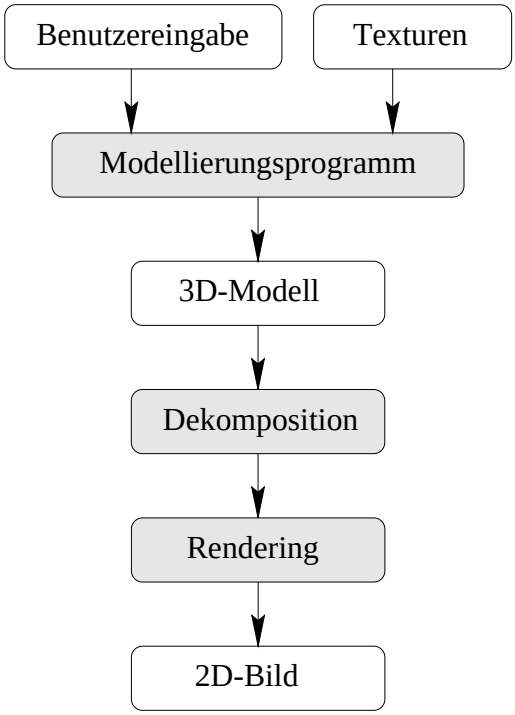

Abb. 6.1: Der Modellierungsablauf.

Der Modellierungsablauf unterscheidet sich nur unwesentlich von dem des geometrischen Modellierens. Die Visualisierung enthält zusätzlich die Dekomposition, die die Szene für Renderingalgorithmen zugänglich macht.

6.3 Eingabe und Bereitstellung der Daten

Dem Benutzer stehen grundsätzlich die gleichen Eingabemöglichkeiten zur Verfügung wie beim geometrischen Modellieren. Die Szene wird vom Benutzer mit Hilfe von Eingabegeräten aus Objekten aufgebaut. Neben den herkömmlichen geometrischen Objekten und Strukturierungsmöglichkieten stehen zusätzliche Objekte zur Verfügung, die eine hohe Abstraktion aufweisen und mit entsprechenden Unterobjekten weiter spezifiziert werden können. So kann der Benutzer seine Szene wie mit einem Baukastensystem modular aufbauen, ohne sich um das Zusammenspiel der Unterobjekte zu kümmern. Die Szene kann natürlich auch mit herkömmlichen Objekten aufgebaut werden, nur dass die Verwaltung und die Abhängigkeiten der Unterobjekte vom Benutzer umgesetzt werden müssen.

Für die Gebäudemodellierung wird ein abstraktes Objekt für die Repräsentation von Gebäuden eingeführt. Daneben werden zusätzliche Unterobjekte definiert, wie beispielsweise das Dach. Der Benutzer kann so seine Szene mit diesen abstrakten Objekten einfach aufbauen und mit den dazu vorgesehenen Unterobjekten weiter ausstatten.

Das Importieren von Daten spielt hier eine marginale Rolle, denn die Daten liegen in den meisten Fällen gar nicht in der geforderten abstrakten Form vor und müssen vom

Benutzer speziell aufbereitet werden, um die Vorteile des objektorientierten Modellierens voll auszunutzen.

6.4 Repräsentationsmodell und seine Strukturen

Die Szene wird mit einem hierarchischen Modell repräsentiert und ist als azyklischer, gerichteter Graph (DAG) (Abb. 6.2) organisiert. Unter einem Objekt werden nicht nur

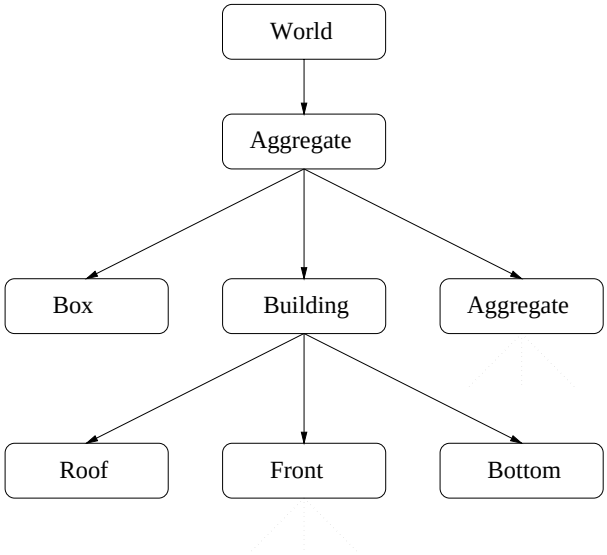

Abb. 6.2: Der Szenengraph.

geometrische Objekte sondern auch Lichtquellen oder Kameras verstanden. Die Objekte selber werden mit Parametern definiert, die der natürlichen Definition nahekommen. Eine Lichtquelle wird beispielsweise durch ihre Position und Leuchtkraft definiert. Die tatsächliche Repräsentaion in Form eines Draht-, Flächen- oder Volumenmodells wird vom Objekt selber realisiert. Damit wird die Verantwortung der Repräsentation dem Objekt zugeteilt und erlaubt eine saubere Trennung der Definition und der Visualisierung der Szene.

Da ein Objekt auch Unterobjekte enthalten kann, wird ein hierarchischer, strukturierter Aufbau der Szene möglich. Das Objekt (Aggregat) übernimmt die Verwaltung der Unterobjekte und muss für die korrekte Repräsentation sorgen. Neben den eigentlichen Objekten kennt der objektorientierte Ansatz auch Attribute, wie geometrische Transformationen oder Texturen, die nicht als eigenständige Knoten im Szenengraph repräsentiert werden, sondern als Bestandteile im Objekt selber enthalten sind. Die Attribute ermöglichen die genauere Spezifizierung der Eigenschaften eines Objektes und beschränken sich nicht nur auf einen speziellen Objekttyp, sondern können falls vorgesehen auf alle Objekte angewendet werden, wie beispielsweise die Transformation. Es gibt aber auch Systeme, die keine solche Trennung von Attributen und Objekten kennen. Die Attribute werden dort als selbständige Objekte betrachtet, wie in The Inventor Mentor von [Wern 94] implementiert.

Die Daten und die Verantwortung der Realisation liegen beim Objekt. Dies entspricht der objektorientierten Philosphie. Es wird eine Unabhängigkeit zwischen den Objekten

und den verarbeitenden Komponenten angestrebt, indem Manipulationen möglichst über wohldefininerte, allgemeine Methoden geschehen. Damit ist es möglich, Dienste (Methoden) von einer Menge von Objekten zu verlangen, ohne über genaues Wissen über die einzelnen Objekte zu verfügen. Als typische Dienste seinen hier das Erzeugen einer Dekomposition des Objektes in einfachere Objekte oder das Schneiden des Objektes mit einem Strahl erwähnt. Diese allgemeinen Dienste (Methoden) werden wenn möglich in einer allgemeinen Basisklasse definiert und teilweise auch implementiert. Auf diese Weise kann sichergestellt werden, dass alle davon abgeleiteten Klassen und die davon instanzierten Objekte diese Dienste anbieten. Normalerweise arbeitet man mit mehreren Hierarchiestufen, um allgemeine Dienste kaskadenartig zu implementieren.

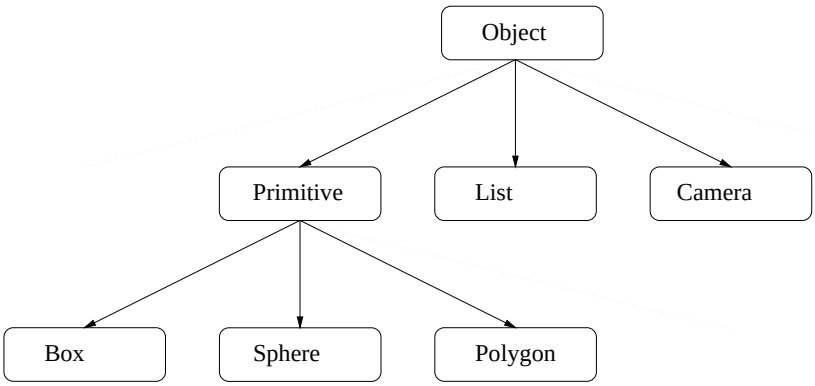

Abb. 6.3: Die Objekthierarchie.

6.5 Visualisierung

Die Visualisierung besteht wie im rein geometrischen Ansatz aus dem Abbilden (*rendering*) der Objekte in den Bildraum. Je nach Realisation können die Objekte mit ihren Attributen und den Beleuchtungsmodellen und Schatten umgesetzt werden. Der objektorientierte Ansatz unterscheidet sich dadurch, dass die Visualisierung (*rendering*) ein Objekt nicht umbedingt umsetzen können muss, denn das Objekt bietet über eine Methode (Dienst) eine alternative Darstellung in Form einer Dekomposition an. Damit ist es möglich einen *renderer* zu entwickeln, der beispielsweise nur Dreiecke umsetzen kann und bei jedem anderen Objekte eine Dekomposition verlangt, die auch weitere Dekompositionen mit sich bringen kann.

Das Umsetzten der Szene geschieht durch Traversieren des Szenengraphen, wobei die Traversierung über die allgemeinen Dienste der Objekte geht. Beim Besuchen eines Objektes muss es sicherstellen, dass neben einer alternativen Darstellung und der korrekten Realisation auch die Attribute entsprechend umgesetzt werden. Die Traversierung wird normalerweise von Objekten eingleitet, wie der Szene selber oder Aggregaten, die nicht direkt umgesetzt werden können oder unbekannt sind. So kann ein Raytracer einen Strahl aussenden, ohne selber die Szene nach den getroffen Objekten zu untersuchen. Die Objekte leiten den Strahl weiter, bis ein konkretes Objekte getroffen wird, oder brechen ab, falls das Objekt mit all seinen Unterobjekten sicher nicht getroffen wird. Weiter können Attribute wie Texturen den weiteren Verlauf des Strahls beeinflussen, indem eine Textur

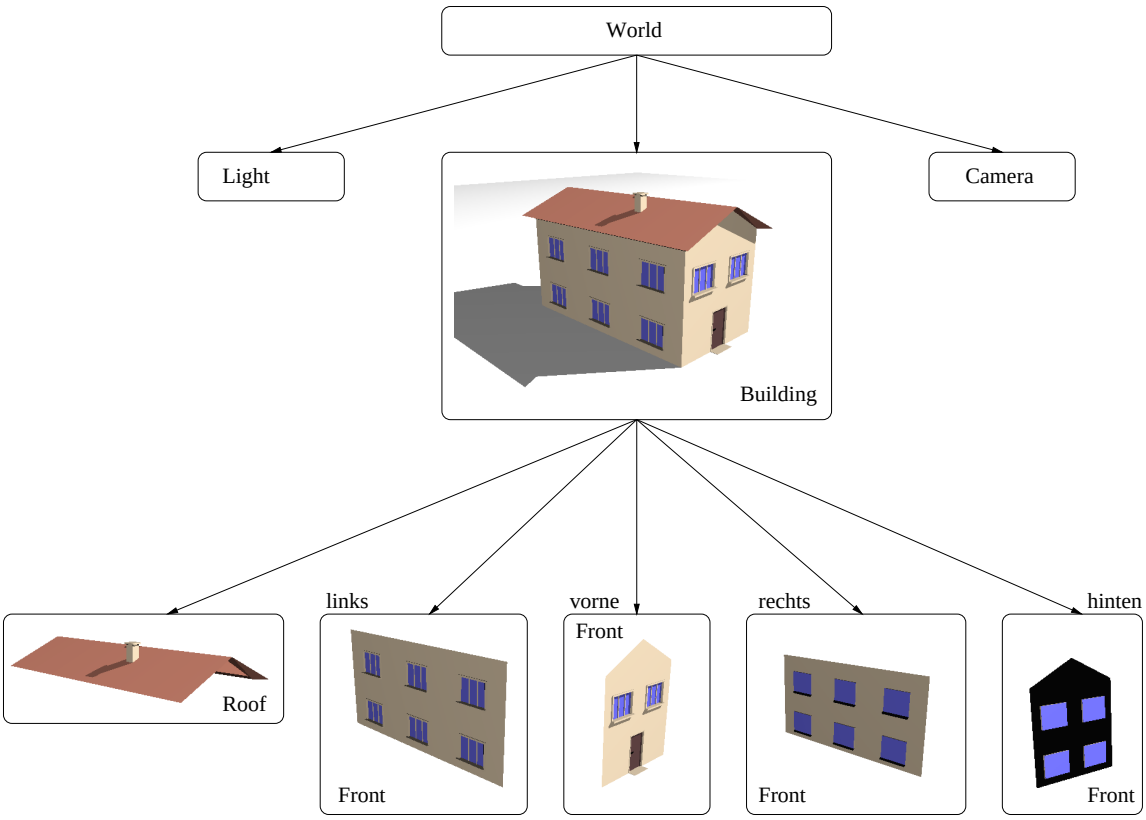

Abb. 6.4: Haus mit Unterobjekten modelliert.

neue Strahlen erzeugen oder den Strahl umlenken kann. Damit übernimmt die Textur die Auswertung, ohne dass sich der Raytracer darum kümmern muss.

Die Abbildung 6.4 zeigt ein Haus, das mit einem Objekt `Building` und Unterobjekten `Roof` und `Front` modelliert wurde. `Building` übernimmt die Verantwortung über die Unterobjekte und bestimmt Lage und Grösse der Unterobjekte. Die Unterobjekte liefern anhand von Spezifikationen von `Building` die passende Repräsentation mit geometrischen Objekten.

6.6 Möglichkeiten und Grenzen

Die abstrakten Objekte erlauben ein einfaches Modellieren von komplexen Objekten, ohne sich um das Zusammenspiel der Unterobjekte zu kümmern. Mit diesem Baukastensystem kann der Benutzer nach dem *top-down*-Verfahren seine Szene modellieren. Neben der einfacheren Modellierung, sind nachträgliche Änderungen problemlos möglich. Die Objekte kümmern sich um die Konsistenz, die durch das Verändern von Parametern der Objekte verloren gehen kann. Bei einem Gebäude kann beispielsweise der Grundriss verändert werden und das Dach wird seine Form mitverändern, ohne dass der Benutzer sich beteiligen muss.

Das Baukastensystem liegt auch im Trend mit den heute erstellten Gebäuden, die vorwiegend aus einzelnen Modulen konzeptioniert und gebaut werden. Dieses Konzept wurde in der Architektur unter dem Begriff *design patterns* [Gam 95] eingeführt und von

der Informatik für das OO-Design übernommen.

Bei der Visualisierung müssen die abstrakten Objekte in eine für den *renderer* verständliche Form umgesetzt werden. Das Umsetzen kann mit OO-Methoden elegant gelöst werden, indem das Objekt selber für die korrekte Umsetzung verantwortlich gemacht wird. Da in einem solchen System jedes einzelne Objekt die Verantwortung für seine tatsächliche Realisation trägt, können neue abstrakte Objekte eingeführt werden, ohne dass das ganze System neu überarbeitet werden muss. Diese Philosophie wurde in BOOGA erfolgreich implementiert [Streit 97].

Die Nachteile entsprechen denen eines Baukastensystems. Der Benutzer ist an die vordefinierten abstrakten, modularen Objekte gebunden und kann nur bedingt das Zusammenspiel zwischen den Unterobjekten verändern. Aufwendig ist die Suche nach einer guten Strukturierung und Aufteilung der komplexen Objekte in einfachere Unterobjekte, ohne dass allzu viel an Allgemeinheit verloren geht. Normalerweise müssen der Aufgabenstellung angepasste Beschränkungen gemacht werden, um eine klare Struktur zu erreichen. Für Gebäude kann man beispielsweise von einem Grundriss und einer Höhe ausgehen und mit Unterobjekten wie einem Dach oder einer Front weitere Details spezifizieren.

Kapitel 7

Modellierungsansatz

In diesem Kapitel wird gezeigt, wie ein eigener objektorientierter Modellierungsansatz (vgl. Kapitel 6) für Gebäude entwickelt wurde und wie dieser in einem objektorientierten Grafik-Framework wie *BOOGA* integriert wurde. Dazu werden die nötigen Begriffe eingeführt und die Objekthierarchie und die Visualisierung unter *BOOGA* erklärt.

7.1 Die Modellierung unter BOOGA

Das Grafik-Framework *BOOGA* ist objektorientiert und wie schon in der Einleitung erwähnt, kann es 2D- und 3D-Welten mit Hilfe von Komponenten verarbeiten. Da die Komponenten und auch das restliche Framework verschiedene *design patterns* [Gam 95] verwenden, wird es einfacher, das bestehende Framework zu verstehen und weiter zu entwickeln. Jedoch wird eine hohe Anforderung an den Entwickler gestellt, was dessen Wissen und Umgang mit *design patterns* und objektorientierter Programmierung im allgemeinen betrifft. Ein für diese Arbeit essentielles *design pattern*, das *proxy pattern*, wird in Kapitel 10 erklärt. Für weitere Erklärungen betreffend *design patterns* sei hier auf das Buch von E. Gamma [Gam 95] verwiesen.

Die Welt unter *BOOGA* wird als gerichteter, azyklischer Graph repräsentiert. Die Elemente des Graphen sind Objekte, wobei zwischen primitiven geometrischen Objekten, Aggregaten und uneigentlichen geometrischen Objekten wie virtuellen Kameras oder Lichtquellen nicht unterschieden wird. Szenen werden mit diesen Objekten modelliert, indem primitive geometrische Objekte (Klasse `Primitive3D`) wie Kugeln, Quader oder NURBS[1] mit Aggregaten (`Aggregate3D`) wie einer Liste (`List3D`) oder 3D-Gitter (`Grid3D`) zu komplexen Objekten kombiniert werden. Diese wiederum können beliebig weiter zusammengesetzt werden und erlauben so ein Strukturieren (baumartig [2], hierarchisch) der Szene.

Alle Objekte werden aufbauend auf einer Basisabstraktion mit identischen Schnittstellen modelliert, damit können alle Objekte vom Benutzer (*client*) über die gleiche Schnittstelle manipuliert werden. Es entfällt die umständliche Typenprüfung durch Verwendung von *tags* und langen *switch*-Blöcken. *BOOGA* enthält die Klasse `Objekt3D`, die die Basisabstraktion aller 3D-Objekte übernimmt (Abb. 7.1). `Primitive3D` übernimmt die

[1] **N**on-**U**niform **R**ational **B**-**S**plines.

[2] Die Bedingungen für eine Baumstruktur werden einzig von Mehrfachreferenzen nicht erfüllt.

Abstraktion der geometrischen Objekte, die eine Flächennormale kennen und normalerweise eine alternative Darstellung mit Hilfe einer Dekomposition anbieten. `Aggregate3D`, das das allgemeine Zusammenfassen von mehreren Objekten zu einem komplexen Objekt ermöglicht und auf dem *composite pattern* baut, übernimmt die Verantwortung und die Verwaltung der Unterobjekte, ohne dass sich der Benutzter speziell darum kümmern muss. Neben den für die Gebäudemodellierung neu entwickelten Objekten gibt es `Shared3D`, das Mehrfachreferenzen auf ein beliebiges Objekt erlaubt, und eine hier nicht erwähnte Vielzahl von anderen Objekten und Abstraktionen.

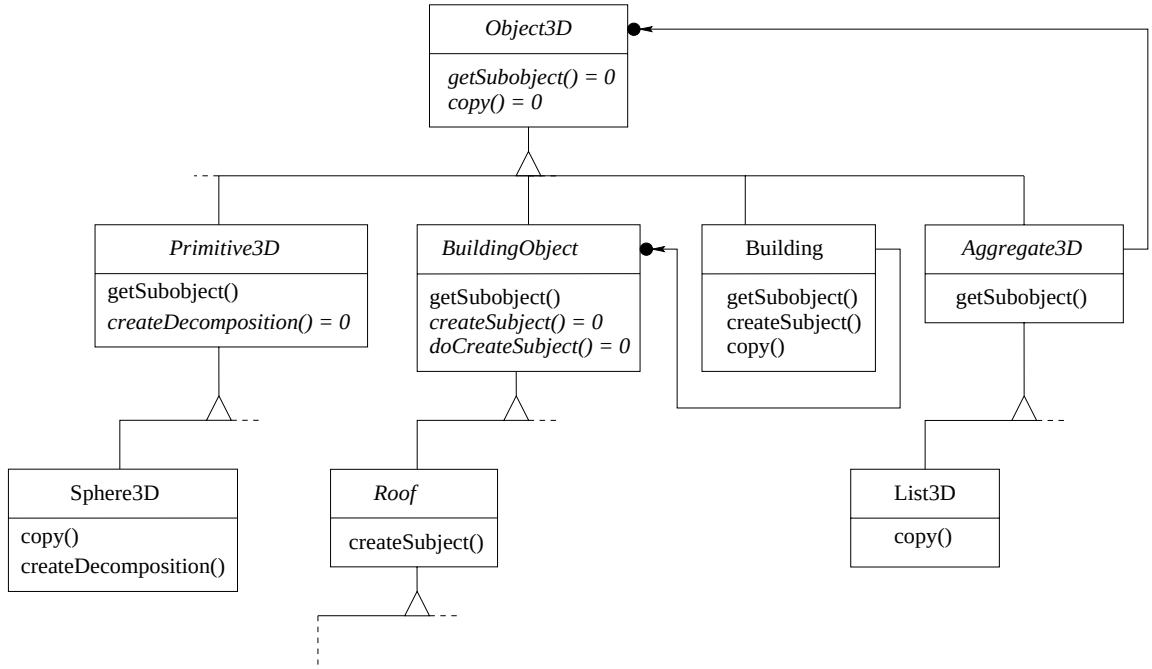

Abb. 7.1: Die Objekthierarchie in BOOGA.

Die Vorteile einer Basisabstraktion und der Verwendung des *composite pattern* für Aggregate ermöglichen es, neue Objekte auch nachträglich in die Objekthierarchie aufzunehmen. Die neu definierten Objekte fügen sich automatisch in die bestehende Struktur ein. Die schon implementierten Objekte bedürfen normalerweise keiner Anpassung, nur die neuen Objekte müssen den Zwecken entsprechend implementiert werden. Ausserdem ist mit einer Basisabstraktion eine flexible und objektunabhängige Modellierung der Welten möglich, weil Aggregate[3] im Normalfall beliebige Objekte aufnehmen und nicht für eine spezielle Gruppe von Objekten ausgelegt sind.

Weiter kennt *BOOGA* Attribute, die ein Objekt genauer spezifizieren und zusätzliche Eigenschaften des Objektes definieren. Die Attribute werden im Szenengraphen nicht als selbständige Objekte repräsentiert (vgl. The Inventor Mentor [Wern 94]), sondern werden vom entsprechenden Objekt verwaltet und umgesetzt. Neben allgemeinen Attributen wie Transformationen oder Texturen, die in der Basisabstraktion verankert sind, gibt es auch solche, die nur für ein spezielles Objekt gelten können. Als Beispiel sei das Flag erwähnt, das eine offene oder geschlossene Definition des Zylinders ermöglicht. Diese

[3] Auch Objekte, die den Zugriff oder die Verwaltung von Unterobjekten übernehmen.

saubere Trennung von Attributen und Objekten ermöglicht eine eindeutige Definition des Geltungsbereiches eines Attributes und erleichtert somit auch deren Manipulation.

7.2 Die Visualisierung unter BOOGA

Die Visualisierung ist eine Komponente, die eine 3D-Welt in eine 2D-Welt mit Hilfe einer Renderingtechnik überführt (Abb. 1.1). Die Ausgabe selber wird von einer anderen Komponente (2D → 2D) übernommen, die beispielsweise das Resultat in einem Window oder auf einem Drucker ausgeben kann.

Das Überführen einer 3D-Welt in eine 2D-Welt besteht im wesentlichen aus dem Traversieren des Szenengraphen und dem Auswerten der einzelnen Knoten. Die Traversierung übernimmt das Bindeglied zwischen Szenengraph und Komponenten. Beim Traversieren werden die einzelnen Knoten besucht und die Objekte der Komponente zur Verarbeitung bereitgestellt. Die Komponente versucht, falls möglich, das Objekt zu verarbeiten. Anderenfalls wird sie eine Dekomposition verlangen oder die Kontrolle an die Traversierung zurückgeben. Die Traversierung entscheidet unter Berücksichtigung des Rückgabewertes der Komponente über den weiteren Verlauf der Traversierung des Szenengraphen. Mit dem Rückgabewert kann die Komponente Einfluss auf die Traversierung nehmen und beispielsweise beim Eintreffen einer Benutzereingabe die Traversierung zum Abbrechen zwingen. Die Aufgabe der Traversierung beschränkt sich aber nicht nur auf die Bereitstellung der Objekte, sondern übernimmt auch allgemeine Aufgaben, wie die Berechnung der kumulierten Transformationsmatrizen, das Aufsammeln von Texturen oder das Clipping.

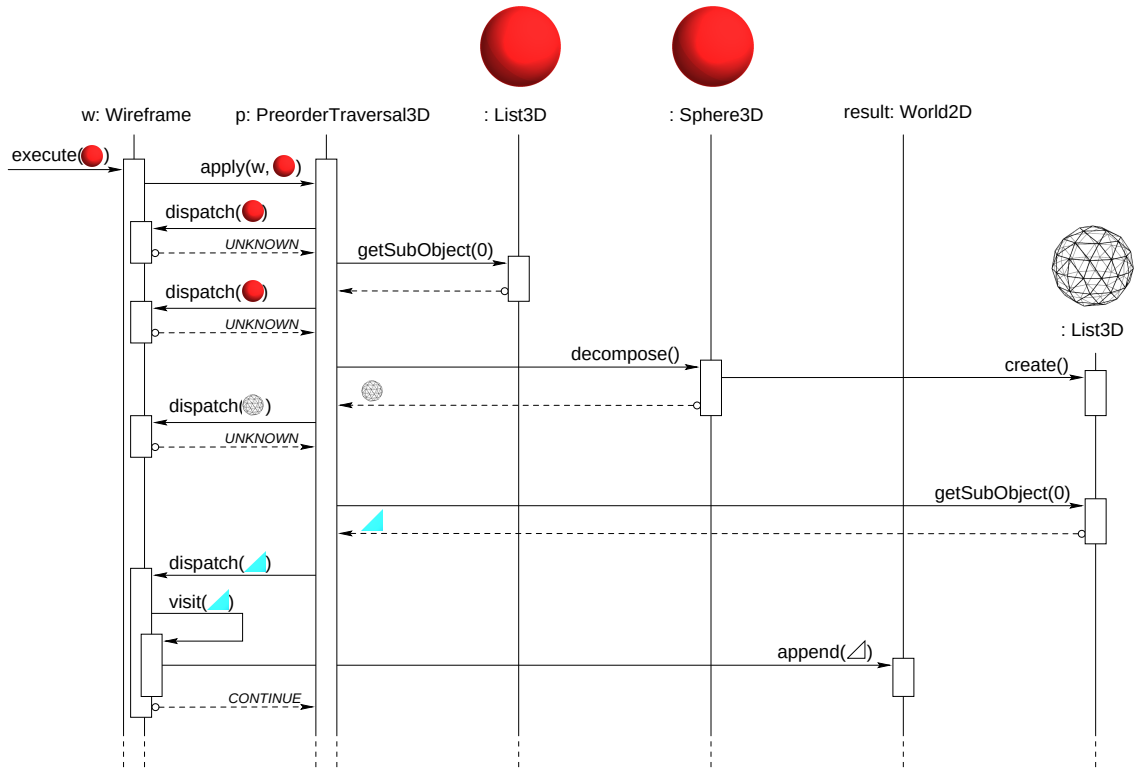

Abb. 7.2: Die Visualisierung von Aggregaten und geometrischen Objekten.

Die Abbildung 7.2 zeigt die Umsetzung einer Szene bestehend aus einer Liste und einer Kugel als Wireframe. Die Applikation `wireframe`, die hier nur Dreiecke verarbeiten kann, versucht als erstes die Liste zu verarbeiten. Diese Liste kann aber nicht direkt umgesetzt werden, deshalb wird eine Traversierung der Unterobjekte der Liste mit dem Rückgabewert `UNKNOWN` von `wireframe` eingeleitet. Die Traversierung liefert als nächstes eine Kugel. Da `wireframe` nicht in der Lage ist eine Kugel direkt zu verarbeiten, wird nun eine Dekomposition der Kugel verlangt. Die Dekomposition bestehend aus Dreiecken kann nun unter Berücksichtigung der Projektionsart und der Kameraposition abgebildet werden, indem die Dreiecke als drei Strecken in der 2D-Welt repräsentiert werden.

Die hier vorgestellte Technik ermöglicht eine Unabhängigkeit zwischen den verarbeitenden Komponenten und den Objekten, die eine Szene repräsentieren. So können neue, komplexe Objekte eingeführt werden, ohne dass die Komponenten selber angepasst werden müssen. Die Objekte übernehmen die konkrete Repräsentation mit primitiven geometrischen Objekten. Auf diese Art wurden beispielsweise NURBS [Bäch 95] für *BOOGA* implementiert, indem die Klasse die ganze Intelligenz der Interpretation enthält und die Flächen oder Kurven, falls nötig, mit Dreiecken bzw. Strecken repräsentieren kann. Im Gegensatz dazu kann die Intelligenz auch in einer spezialisierten Komponente ausgelagert werden, mit dem Nachteil, dass andere Komponenten die Objekte nicht ohne die spezialisierte Komponente korrekt verarbeiten können.

7.3 Der Modellierungsansatz

Zu Beginn der Arbeit stand der Modellierungsansatz nicht fest, weil die objektorientierte Einbettung in *BOOGA* im Vordergrund stand. Eine schlechte Einbettung wäre auch im Widerspruch zum Gebot der Wiederverwendbarkeit gestanden, weshalb der vorliegende Ansatz sich nach und nach herauskristallisierte.

Am Anfang der Arbeit stellten sich folgende Fragen:

- Wie kann ein Gebäude auf einfache Art beschrieben werden, ohne dass man sich auf einen speziellen Gebäudetyp festlegen muss ?
- Wie können die Gebäudemodellierung möglichst einfach in *BOOGA* integriert werden und bestehenden Mechanismen ausgenutzt werden ?
- Wer übernimmt die Interpretation der Gebäudeobjekte, die Objekte oder eine spezialisierte Komponente ?

Zu Beginn war unklar, wie ein Gebäude überhaupt beschrieben werden sollte. Deshalb wurde in einem ersten Schritt mit einem einfachen Objekt gearbeitet, um erste Erfahrungen zu sammeln. Das neue Objekt besass eine quaderähnliche Geometrie und entsprach funktionell einem primitiven geometrischen Objekt, ähnlicher der Kugel. Damit lag die ganze Intelligenz für die Erzeugung der dazugehörigen geometrischen Struktur in einem einzigen Objekt. Auch wenn die Quaderstruktur die Repräsentation von einfachen rechteckigen Häuser ermöglichte, konnten Häuser mit beliebiger Grundfläche nur mit einer grossen Anzahl von zusätzlichen Parametern realisiert werden. Dies führte dazu, dass die dazugehörige Klasse mit den immer zahlreicheren Parametern sich rasch aufblähte und unübersichtlich wurde. Neben dem Fehlen einer ansprechenden Beschreibung von Dächern, war die Parametrisierung der Gebäude umständlich und nur wenig intuitiv.

Deshalb wurde in einem zweiten Schritt von einem abstrakten Gebäude ausgegangen, das nur durch eine Grundfläche und eine Höhe definiert wird. Weitere Eigenheiten des Gebäudes werden durch zusätzliche Objekte definiert. Daraus entwickelte sich eine Art von Baukastensystem, das auf der Philosophie der objektorientierten Modellierung aufbaut. Dies ermöglicht dem Benutzer ein einfaches und allgemeines Gebäude mit spezialisierten Objekten, hier Gebäudeobjekte genannt, genauer zu modellieren. Die Gebäudeobjekte wurden so ausgelegt, dass sie auch Unterobjekte verwalten können, um eine individuelle Modellierung der Gebäudeobjekte selber zu ermöglichen. Diese Schachtelung erlaubt eine intuitive *top-down*-Modellierung eines Gebäudes. Der Benutzer parametrisiert seine Objekte in Abhängigkeit des Parentobjektes[4] und muss nur noch einige wenige freie Parameter definieren, die nicht schon von den Abhängigkeiten[5] zwischen den Objekten gegeben sind. Die Umsetzung in konkrete geometrische Objekte und ein korrektes Zusammenspiel[6] zwischen den einzelnen Objekten wird von diesen selber getragen, indem ein Objekt an seine Unterobjekte die nötigen Informationen mitteilt und unter Umständen auch mit den Unterobjekten kommuniziert. Ein korrektes Zusammenspiel macht es möglich, dass nachträgliche Änderungen eines Parameters in einem einzelnen Objekt, wie das Verschieben einer Ecke der Grundfläche, an die betroffenen Objekte weitergereicht wird und diese sich ihrer geometrischen Realisation entsprechend anpassen und die Änderungen auch ihren eigenen Unterobjekten mitteilen. Es entfällt das aufwendige Anpassen aller betroffenen Objekte durch den Benutzer.

Dieses hierarchische Baukastensystem ermöglichte es, die Definition eines Gebäudes auf mehrere Objekte zu verteilen und Klassen zu entwickeln, die sich einem speziellen Problem der Modellierung annehmen. Dabei übernimmt ein allgemeines und abstraktes Gebäude als Basiseinheit die Hauptverantwortung für die korrekte Erzeugung der entsprechenden geometrischen Realisation, indem den Gebäudeobjekten die nötigen Informationen, wie Grundfläche und Höhe mitgeteilt werden. Die Unterobjekte passen sich dann selbständig an. Damit kann ein korrektes Zusammenspiel aller Objekte sichergestellt werden. Weiter kann in einem Objekt ein normiertes lokales Koordinatensystem eingeführt werden, was eine unabhängige, relative Modellierung, die auch auf Unterobjekte übertragen werden kann, ermöglicht.

Im Gegensatz zur objektorientierten Modellierung, die Intelligenz der Erzeugung der konkreten Repräsentation den einzelnen Objekten zuweist, stand auch die Entwicklung einer eigenen Sprache für die Beschreibung der Gebäude zur Diskussion. Dieser Lösungsansatz erwies sich aber für interaktive Anwendungen, wie Editoren, als zu umständlich, weil die Manipulation von schon bestehenden Beschreibungen eine genaue Kenntnis der Sprache voraussetzt. Diese Sprachkenntnisse müssten dann in allen Komponenten, die Gebäude verarbeiten sollen, enthalten sein. Damit müssten bei einer eventuellen Spracherweiterung alle betroffen Komponenten angepasst werden, was das Gebot der einfachen Erweiterbarkeit missachten würde. Ein weiterer Nachteil ist auch der Abstraktionsverlust, der bei einer konkreten Umsetzung in geometrische Objekte entsteht. Die Beschreibung des Gebäudes kann nicht mehr anhand der geometrischen Objekte rekonstruiert werden. Dazu wären

[4]Ein Parentobjekt übernimmt die Verwaltung seiner Unterobjekte, wobei die Abhängigkeit der Unterobjekte vom Parentobjekt stark variieren kann. Im Szenengraph entspricht dies dem Vater.

[5]Eine Front beispielsweise wird sich entsprechend der Höhe des Gebäudes verhalten.

[6]Unter dem Zusammenspiel wird beispielsweise die Sicherstellung des Berührens zweier Mauern in einer Ecke verstanden.

einige zusätzliche Hilfskonstrukte nötig, die die Beschreibung verwalten können. Ausserdem muss die Beschreibung nach jeder Veränderung auf Konsistenz überprüft und neu interpretiert werden.

7.4 Repräsentation und Definition eines Gebäudes

Der Modellierungsansatz beruht auf einem hierarchischen Baukastensystem, das zwei grundlegende Objekte kennt. Einerseits gibt es ein Objekt, das einem allgemeinen und minimalen Gebäude entspricht und als Aggregat der Gebäudeobjekte (Def. 7.1) dient. Anderseits gibt es Gebäudeobjekte (Def. 7.2), die ein Gebäude oder auch ein Gebäudeobjekt genauer modellieren und von dessen Parentobjekt abhängen.

Definition 7.1 *Ein Gebäude definiert sich aus einem zur xy-Ebene planaren einfachen Polygon, dem Grundriss, und einer positiven Höhe (Abb.7.3). Zusätzlich kann das Polygon Löcher enthalten, die als einfache Polygone definiert werden und Innenhöfe repräsentieren. Die Polygone in die xy-Ebene projiziert bilden ein Rechtssystem.*

Definition 7.2 *Ein Gebäudeobjekt ist ein Objekt, das die genauere Modellierung übernimmt und keinem konkreten geometrischen Objekt entsprechen muss. Es ist immer Unterobjekt eines Gebäudes oder eines anderen Gebäudeobjektes und die konkrete Realisation mit geometrischen Objekten wird von dessen Parentobjekt teilweise vorgegeben.*

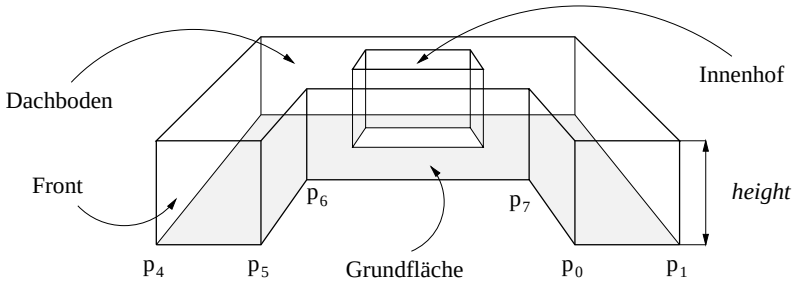

Abb. 7.3: Geometrie und Parametrisierung des Gebäudes.

Das Gebäude kann als Basis oder Grundeinheit des Baukastensystems angesehen werden. Die Definition entspricht einem Polyeder, das durch die Grundfläche mit deren Löchern und der Höhe aufgespannt wird[7]. Das Polyeder stellt ein minimales Gebäude dar, das keine weiteren Objekte enthält. Dabei werden für die Darstellung nur die Seitenflächen, die Mauern, berücksichtigt, um einen möglichst allgemeinen Ansatz anzustreben.

Die Gebäudeobjekte sind die eigentlichen "Bauelemente", die auf der Grundeinheit basierend eingesetzt werden, und die eine genauere Modellierung des Gebäudes ermöglichen. Sie übernehmen beispielsweise die Modellierung eines Daches oder einer Mauer mit all ihren Fenstern. Diese Objekte leben auf einer Oberfläche des Polyeders oder kennen eine andere Beziehung zum Parentobjekt. Damit kann ein lokales, normiertes Koordinatensystem eingeführt, was ermöglicht, die Lage, Ausrichtung und/oder die Grösse relativ zu

[7]Entspricht der Definition eines *translational sweep*-Körpers entlang der z-Achse.

definieren. Der Benutzer kann so ein Gebäudeobjekt mit einigen wenigen Parametern vollständig definieren. Es entfallen das mühsame Positionieren im WKS und das Ausrichten mit all den anderen Objekten. Die Positionierung von Fenstern beispielsweise kann relativ zur Seitenfläche des Polyeders geschehen und ist somit unabhängig von Ort und Grösse der Seitenfläche. Das Gebäudeobjekt erhält für die Interpretation die nötigen Informationen, um die Fenster im WKS korrekt zu positionieren.

7.5 Grobdesign der Objekte

Eine erste grundlegende Einteilung der Objekte wurde im letzten Abschnitt schon gemacht. Die Klasse `Building`, direkt von `Object3D` abgeleitet, übernimmt die Funktionalität der Grundeinheit, dem Gebäude aus Definition 7.1. Die Gebäudeobjekte aus Definition 7.2 übernehmen die Spezialisierung der Modellierung.

Das Gebäude als Grundeinheit entspricht einem erweiterten Aggregat und bedient sich, wie alle anderen Aggregate, des *composite pattern*. Neben der Verwaltung der Gebäudeobjekte kann es bei Bedarf (*on demand*) aber auch eine geometrische Struktur erzeugen. Diese Möglichkeit wird dann verwendet, wenn ein Default–Gebäude erzeugt werden soll. Das Erzeugen bei Bedarf ist mit Hilfe des *proxy pattern* gelöst. Weiter übernimmt `Building` die Verwaltung der wichtigen Informationen, der Geometrie des Gebäudes (Grundriss und Höhe). Diese Informationen werden bei einer Interpretation in konkrete geometrische Objekte an die Unterobjekte, die Gebäudeobjekte, weitergegeben. Dies geschieht durch Weiterreichen des Zeigers des Gebäudes an die Unterobjekte, die sich dann selbständig die nötigen Informationen heraussuchen. Damit kann sichergestellt werden, dass sich auch alle Unterobjekte alle nötigen Informationen akquirieren können. Weiter übernimmt die Klasse `Building` auch die Prüfung der Konsistenz seiner Definition mit all seinen Unterobjekten und enthält weitere Methoden, die die Manipulation der Parameter, wie auch seiner Gebäudeobjekte ermöglicht.

Für die Gebäudeobjekte wurde eine dreistufige Objekthierarchie eingeführt (Abb. 7.4). Die erste Stufe der Hierarchie bildet die Klasse `BuildingObject`, die als Basisabstraktionsklasse auftritt und direkt von `Object3D` abgeleitet ist. Damit kann sichergestellt werden, dass alle Gebäudeobjekte über ein einheitliches Interface verfügen und sich in die bestehende Objekthierarchie von *BOOGA* problemlos einfügen. `BuildingObject` übernimmt die Verwaltung der Abhängigkeit zum Gebäude und verwaltet auch die aktuelle Realisation mit geometrischen Objekten in Form einer Dekomposition in einer eigenen Struktur. Damit besitzen auch alle davon abgeleiteten Klassen die Funktionalität eines Caches[8]. Ausserdem werden auf dieser Stufe `Object3D`-typische Dienste, wie das Schneiden eines Strahls mit einem Objekt, an die Dekomposition weitergereicht. So ist sichgestellt, dass auch alle davon abgeleiteten Klassen diese Dienste anbieten.

Die Forderung, dass die normalerweise aufwendige Realisation erst bei Bedarf erzeugt wird und teilweise vom Parentobjekt abhängt, entsprechen weitgehend den Funktionalitäten eines *proxy*[9]. Deshalb kennt die Basisabstraktion `BuildingObject` folgende drei

[8] Der Cache ist in diesem Zusammenhang ein typische Grundfunktionalität, die in einer Basisklasse implementiert gehört, um unnötige Codedublizierung zu vermeiden.

[9] Ein *proxy* übernimmt im allgemeinen die Beschaffung und/oder die Verwaltung von Objekten, ohne dass sich der Benutzer darum kümmern muss.

Methoden, die für das *proxy pattern* typisch sind:

- `createSubject()` gibt eine Realisation mit geometrischen Objekten in Form einer Dekomposition zurück.
- `getSubject()` gibt die momentan aktuelle Realisation zurück. Dabei wird falls möglich auf eine schon bestehende Dekomposition zurückgegriffen. Anderenfalls wird mittels `createSubject()` die Struktur erzeugt und in einer eigenen Struktur gespeichert.
- `subjectChanged()` löscht eine bestehenden Dekomposition und impliziert beim späteren Zugriff ein erneutes Erzeugen der Dekomposition mittels `getSubject()`.

Die zweite Stufe der Objekthierarchie der Gebäudeobjekte (Abb. 7.4) übernimmt die zusätzliche Abstraktionsebene, die Gebäudeobjekte gleichen Typs[10] zu Gruppen[11] zusammenfasst. Diese Klassen dieser Hierarchiestufe übernehmen allgemeine Funktionalitäten, die für die einzelnen Gruppen typisch sind. Diese mehrstufige Hierarchie erlaubt auch Mechanismen, die weitgehend Codeduplizierungen verhindern. Für die Gebäudemodellierung wurde ein in *BOOGA* weit verbreiterter Mechanismus verwendet. Die Abstraktionsklasse besitzt für die Erzeugung der Realisation einen konstanten Teil, der in `createSubject()` implementiert wird. Der variable Teil wird entsprechend in den konkreten Gebäudeobjekten in `doCreateSubject()` implementiert, wobei die konkreten Gebäudeobjekte die letzte Stufe der Hierarchie bilden. Die konkreten Gebäudeobjekte decken, nebst dem Gebäude (`Building`), folgendes der Gebäudemodellierung ab:

- `Bottom`: Modellierung von Fussböden oder dem Grundriss des Gebäudes.
- `Face`: Detaillierte Modellierung von Teilbereichen einer Front.
- `Front`: Definition der Geometrie einer Front.
- `Roof`: Modellierung von Dächern aller Art.
- `Snatch`: Automatisches Positionieren von Objekten bezüglich anderen Objekten.

Ein solches Design der Objekthierarchie erlaubt auch nachträglich neue konkrete Gebäudeobjekte ohne grösseren Aufwand einzufügen, denn die neuen Objekte erben die nötigen Eigenschaften und Methoden der Abstraktionsklasse. In Falle eines neuen Typs von Gebäudeobjekt muss natürlich als erstes ein allgemeines Objekt gefunden werden, das diesen Typ von Gebäudeobjekt repräsentiert. Bei der nicht immer einfachen Suche und Entwicklung von neuen Abstraktionsklassen sollte ein tragender Leitgedanke [Streit 97] von *BOOGA* nicht vergessen werden:

> *Flexibilität ist nur dann erwünscht, falls sie auch benötigt wird.*

Die Namengebung der Klassen der Gebäudeobjekte wurde möglichst uniform gestaltet, indem der Klassenname einer konkreten Klasse mit dem Präfix des Namens der Abstraktionsklasse beginnt. So können Objekte und Klassen einfach im Hierarchiekonzept lokalisiert werden, und der Name wiedergibt auch die Ableitungshierarchie.

[10] Der Typ hängt vom zu repräsentierenden Objekt ab.
[11] Ein Gruppe umfasst beispielsweise alle Arten von Dächern.

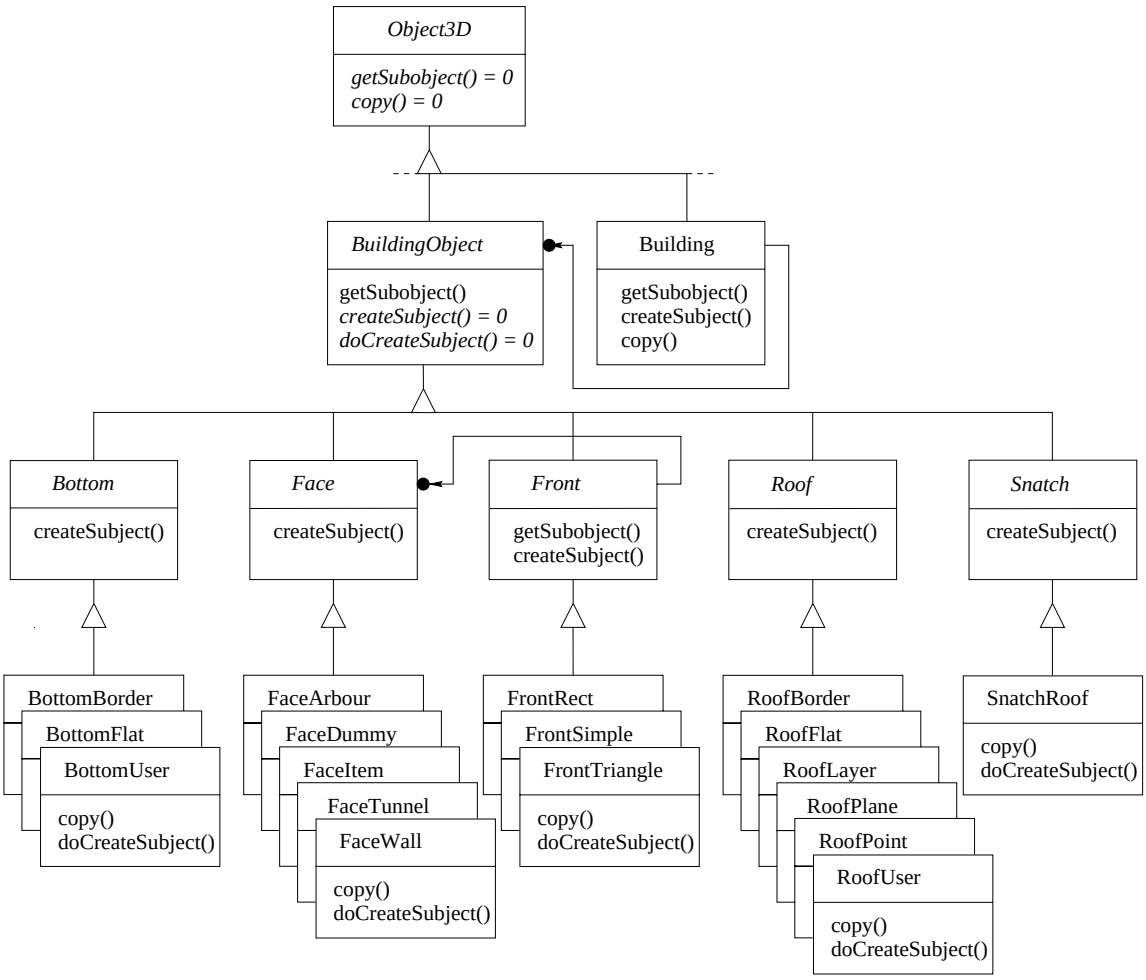

Abb. 7.4: Die Objekthierarchie der Gebäudeobjekte.

Diese dreistufige Objekthierarchie erlaubt eine Zuteilung der Funktionalitäten, die hierarchiegerecht ist und somit unnötige Codedublizierungen verhindert. Mit der Ableitung von `Object3D` stehen auch alle Möglichkeiten, von der Erzeugung eines Objektes anhand einer Liste von Parametern mittels `make()`, bis zum schon erwähnten Schneiden des Objektes mit einem Strahl mittels `doIntersect()`, zur Verfügung. Damit kann die Entwicklung der Objekte und ihrer Klassen hauptsächlich auf das Problem der Realisation der dazugehörigen geometrischen Strukturen beschränkt werden.

7.6 Strukturgraph eines Gebäudes

Der Szenengraph eines konkreten Beispiels (Abb. 7.5) soll die Struktur (vgl. Arbeit von [Hab 96] und Abb. 8.6) der Modellierung eines Gebäudes erläutern und die Abhängigkeiten zwischen den einzelnen Objekten, vom Gebäude, bis zu einem primitiven geometrischen Objekt, aufzeigen.

In einem ersten Schritt wird die Grundfläche mit allen Innenhöfen und der Höhe des Gebäudes mit Hilfe von `Building` definiert. Damit lassen sich schon die Mauern des Hauses

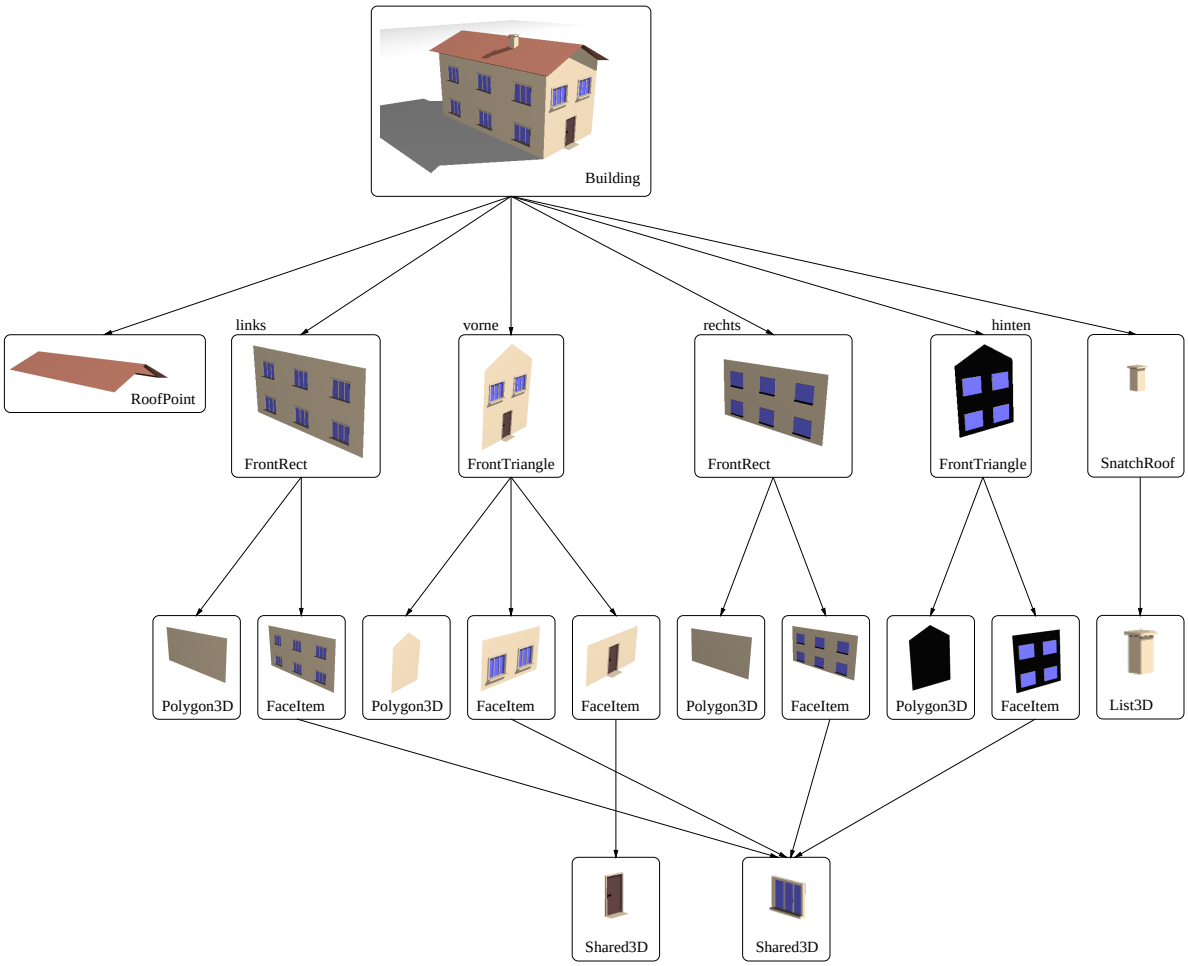

Abb. 7.5: Szenengraph eines Hauses.

mit Rechtecken darstellen. Dies entspricht dem Default–Gebäude ohne Dach und Boden.

In einem zweiten Schritt werden die Gebäudeobjekte in `Building` eingefügt. Das Beispiel zeigt ein Haus mit einem Dach, vier Fronten, bestehend aus Teilfronten, und einem Schornstein. Das Dach ist mit `RoofPoint` modelliert, das eine Definition mit Punkten des Dachgiebels ermöglicht. Der Schornstein besteht aus ein paar primitiven geometrischen Objekten und ist mittels `SnatchRoof` auf das Dach gesetzt worden. Dabei ist die z-Koordinate in Abhängigkeit des Daches und der horizontalen Lage des Schornsteins von `SnatchRoof` selbständig ermittelt worden. Die Fronten des Hauses sind mit zwei verschieden Arten von Fronten modelliert worden. `FrontRect` modelliert die strikt rechteckigen Fronten. Der Giebel ist mit `FrontTriangle` definiert. Damit lassen sich Fronten mit mehreren Giebeln und auch Kellermauern modellieren. Allgemein werden Fronten mit einem einfachen Polygon repräsentiert. Die Fronten wiederum können genauer modelliert werden, indem Teile der Fronten mit Teilfronten (`Face`) definiert werden. Die Türe und die Fenster, die aus einzelnen primitiven Objekten zusammengesetzt sind, sind hier mit `FaceItem` auf den Fronten eingesetzt worden. Die Vorderfront beispielsweise besteht aus zwei Teilfronten, die die Türe und die zwei Fenster des ersten Stockes anordnen. `FaceItem` übernimmt

allgemein die mehrfache Anordnung[12] eines beliebig zusammengesetzten Objektes und, falls nötig, werden auch die entsprechenden Löcher herausgeschnitten.

Die Objekthierarchie (Abb. 7.4) zeigt die möglichen Gebäudeobjekte. Bis jetzt ist die Teilfront das einzige Gebäudeobjekt, das nicht direkt am Gebäude hängt, sondern von einem anderen Gebäudeobjekt verwaltet wird, der Front. Ein weiterer Typ von Gebäudeobjekt, der hier nicht eingesetzt wurde, ist `Bottom`, der die Modellierung des Bodens ermöglicht. Genauere Definitionen und Möglichkeiten der Gebäudeobjekte werden im Kapitel 13 erklärt und besprochen.

[12] Die Anordnung ist äquidistant und in zwei Dimensionen möglich.

Kapitel 8

Zwei Anwendungsbeispiele

Dieses Kapitel gibt eine praxisorientierte Einführung in die Gebäudemodellierung mit *BOOGA* und soll die Möglichkeiten der objektorientierten Modellierung aufzeigen. Das erste ausführlich erklärte Beispiel zeigt den Modellierungsvorgang mit BSDL und soll dem Benutzer als Einstieg in die Gebäudemodellierung dienen. Das zweite Beispiel zeigt anhand einer kleinen Applikation Möglichkeiten der Verarbeitung von Gebäuden in einer Szene.

8.1 Gebäudemodellierung in BSDL

Das hier vorgestellte Beispiel erklärt die Gebäudemodellierung anhand eines einfachen Hauses.

Die Geometrie des Gebäudes

Für die Grundfläche des Hauses wird ein Rechteck von der Grösse 20×40 gewählt. Die Grundfläche liegt in der xy-Ebene und ist achsenparallel zum Koordinatensystem. Die Höhe des Hauses ist 20. Damit ist die Geometrie des Gebäudes bestimmt und definiert das Default–Gebäude (Abb. 8.1). Ausserdem wird dem Haus eine Textur `rock` zugewiesen, die als Default–Textur des Gebäudes und dessen Unterobjekte dient. Zusätzlich ist noch ein Rasen mit einer Box und einer grünen Farbe `green` modelliert.

```
// Texturen fuer das Haus und den Rasen
define rock whitted {
  ambient [220/255,200/255,170/255];
  diffuse [220/255,200/255,170/255];
}
define green whitted {
  ambient [.3,.4,.2];
  diffuse [.3,.4,.2];
}
```

```
// Die Geometrie des Gebaeudes
building (20,[-10,-20, 0],[ 10,-20, 0],[ 10, 20, 0],[-10, 20, 0]){
  rock;
}

// Der Rasen
box([-1000,-300,-1],[1000,1000,0]){green;}
```

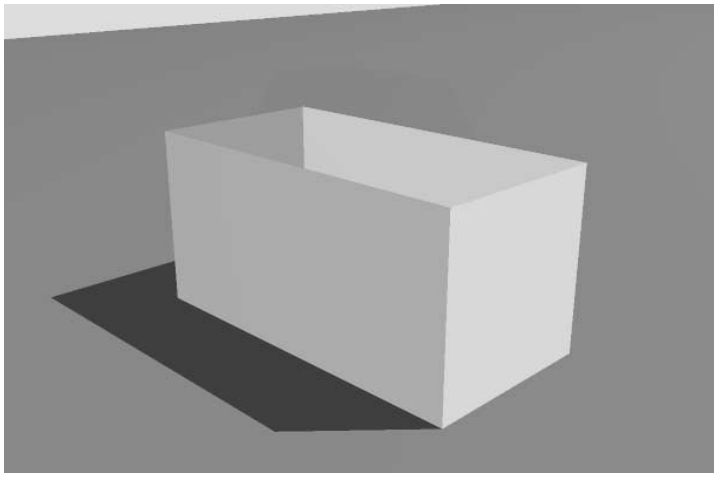

Abb. 8.1: Haus als Default–Gebäude repräsentiert.

Die einzelnen Fronten

Die Geometrie der Fronten ist einfach gehalten und soll einen einfachen Dachstock mit abgeflachtem Dachgiebel modellieren. Die längeren Seiten des Hauses werden mit dem Objekt `FrontRect` definiert, das nur eine rechteckige Form kennt. Mit $bottom = 0$ und $top = 1$ wird eine Front der Grösse der zugehörigen Seitenflächen des Default–Gebäude definiert. Für die hintere und die vordere Front wird ein abgeflachter Dachgiebel mit `FrontTriangle` modelliert. Dieser wird mit zwei zusätzlichen Punkten `[0.3,1.25]` und `[0.7,1.25]` definiert, wobei die Koordinaten relativ zur entsprechenden Seitenfläche eingeben werden.

```
building (20,[-10,-20, 0],[ 10,-20, 0],[ 10, 20, 0],[-10, 20, 0]){
  rock;
  // (frontindex,polygonindex,...)
  fronttri(0,0,[0.3,1.25],[0.7,1.25]);  // hinten
  // (frontindex,polygonindex,bottom,top)
  frontrect(1,0,0,1);                   // links
  fronttri(2,0,[0.3,1.25],[0.7,1.25]);  // vorne
  frontrect(3,0,0,1);                   // rechts
}
```

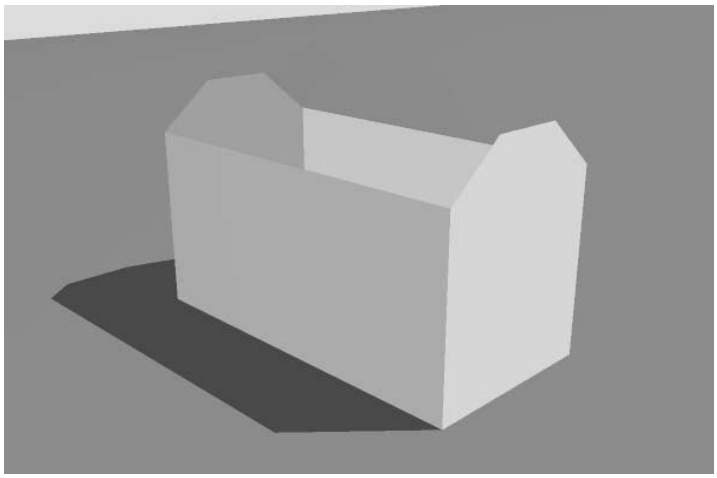

Abb. 8.2: Haus aus leeren Fronten bestehend.

Das Dach

Das Dach soll zu den Seiten des Dachgiebels etwas abgeflacht sein und die Geometrie der Fronten vollständig berücksichtigen. Dazu wird das Dachobjekt `RoofPoint` verwendet, das neben einem Dachvorsprung die Möglichkeit der Definition von zusätzlichen Dachpunkten bietet. Diese Punkte definieren Punkte des Dachgiebels. Die zusätzlichen Punkte `[0,17,7.5]` und `[0,17,7.5]` liegen innerhalb des Grundrisses und erzeugen ein auf den Seiten teilweise abgeflachtes Giebeldach. Der Dachvorsprung ist 2. Dem Dach ist noch eine Textur `dachRot` zugeordnet.

```
building (20,[-10,-20, 0],[ 10,-20, 0],[ 10, 20, 0],[-10, 20, 0]){
  rock;
  roofpoint(2,[0, 17,7.5],[0, 17,7.5]) {dachRot;}
  fronttri(0,0,[0.3,1.25],[0.7,1.25]);  // hinten
  frontrect(1,0,0,1);                   // links
  fronttri(2,0,[0.3,1.25],[0.7,1.25]);  // vorne
  frontrect(3,0,0,1);                   // rechts
  snatch([5,6]){chimney;}   // Der Kamin
}
```

Weiter wird ein einfacher Kamin `chimney` mit zwei Boxen modelliert. Der Kamin wird mit dem Objekt `SnatchRoof` auf das Dach aufgesetzt und entsprechend der lokalen Neigung des Daches um die z-Achse gedreht. Das lokale Koordinatensystem des Kamins wird nach $(5, 6, z)$ verschoben, wobei `SnatchRoof` den dazugehörigen z-Wert durch Schneiden des Daches mit der Geraden $(5, 6)$ bestimmt. Der Kamin wird so gedreht, dass die x-Achse und die Fallinie des Daches bezüglich der xy-Ebene in die gleiche Richtung zeigen.

```
define chimney list {
  box ([0,-.75,0],[-1.5,.75,3]);
  box ([0.25,-1,3],[-1.75,1,3.25]);
  rock;
}
```

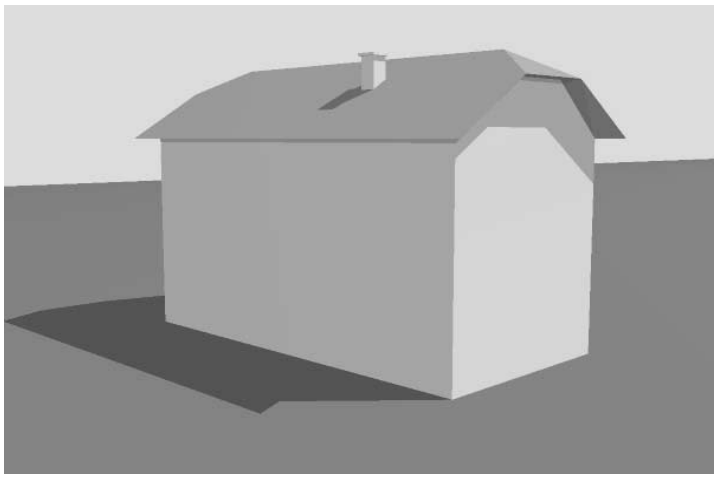

Abb. 8.3: Haus mit Dach und Kamin.

Die Teilfronten

Der letzte Modellierungsschritt besteht aus dem Einsetzen von Teilfronten in die Fronten. Dazu werden erst einmal zwei Fenstertypen `window` und `windowSmall` (Abb. 8.4) benötigt, wobei hier nur `window` vollständig definiert wird.

```
define window list {
  box([-10,-8,2],[10,-7,-1]);       // unten
  box([-10,8,0.5],[10,7,-1]);       // oben
  box([-10,8,0.5],[-9,-7,-1]);      // links
  box([10,8,0.5],[9,-7,-1]);        // rechts
  box([-3-.2,-8,-.5],[-3+.2,8,-1]); // linke Leiste
  box([3-.2,8,-.5],[3+.2,-8,-1]);   // rechte Leiste
  box([-10,-8,-1],[10,8,-1.1]){glasDunkel;} // Glas
  rock;  // Textur
  scale[0.3,0.3,0.3];
}
```

Abb. 8.4: Die beiden Fenster `window` (links) und `windowSmall` (rechts).

In die rechte Front `frontrect(3,0,0,1)` und die linke Front `frontrect(1,0,0,1)` werden 3×2 Fenster mit `FaceItem` eingesetzt. Da die Teilfront die ganze Front (Seitenfläche) ausfüllen soll, wird die Grösse auf $\vec{from} = [0,0]$ und $\vec{to} = [1,1]$ gesetzt. Mit dem Attribut `hole` werden die Löcher für die Fenster anhand der *boundingbox* selbständig bestimmt. Ausserdem wird der Koordinatenursprung des Fensters etwa in die Mitte des

Elementes[1] der Teilfront verschoben, da der Ursprung des Fensters in der Mitte liegt.

```
frontrect(1,0,0,1){  // links
  // (from,to,column,row)
  faceitem([0,0],[1,1],3,2){
    window;
    hole;
    displacement[0.5,0.4];  // relative Translation
  }                         //  bezueglich dem Element
}

frontrect(3,0,0,1){  // rechts
  faceitem([0,0],[1,1],3,2){
    window;
    hole;
    displacement[0.5,0.4];
  }
}
```

Die hintere Front $(0, 0)$ wird in gleicher Weise mittels `FaceItem` mit 2×2 Fenstern modelliert. Ausserdem wird ein zusätzliches kleineres Fenster im Giebel eingesetzt. Die Teilfront ragt zwar etwas heraus, aber das Fenster liegt noch vollständig innerhalb der Front. Damit ist es der Front möglich, das Fenster korrekt zu berücksichtigen.

```
fronttri(0,0,[.3,1.25],[.7,1.25]){  // hinten
  faceitem([0,0],[1,1],2,2){
    window;
    hole;
    displacement[0.5,0.4];
  }
  faceitem([0.3,1],[.7,1.3],1,1){
    windowSmall;
    hole;
    displacement[0.5,0.3];
  }
}
```

Die vordere Front besteht aus drei Teilfronten. Die Türe `door` (siehe Anhang A.1) wird in die untere Hälfte der Front (Seitenfläche) eingesetzt. Die zwei Fenster werden mit einer zweiten Teilfront in der oberen Hälfte eingesetzt. Das kleinere Fenster `windowSmall` wird wie im Falle der hinteren Front modelliert. Hier muss beachtet werden, dass sich die Teilfronten nicht überlappen, weil für die einzelnen Teilfronten Rechtecke in der Front

[1] `FaceItem` wird in 3×2 Elemente aufgeteilt.

herausgeschnitten werden, was zu unerwünschten Effekten führen könnte.

```
fronttri(2,0,[.3,1.25],[.7,1.25]){  // vorne
    faceitem([0,0.5],[1,1],2,1){ // 1. Stock mit 2 Fenstern
      window;
      hole;
      displacement[0.5,0.4];
    }
    faceitem([0,0],[1,0.5],1,1){  // Tuere
      window;
      hole;
      door;
      displacement[0.5,0];
    }
    faceitem([0.3,1],[.7,1.3],1,1){ // Fenster im Dachgiebel
      windowSmall;
      hole;
      displacement[0.5,0.3];
    }
  }
```

Ausserdem wird mit `BottomBorder` ein Rand zwischen den beiden Stockwerken modelliert.

```
  // (ledge, borderwidth, borderheight, borderdepth)
  bottomborder(0.5,0.5,0.5,0.1){
    height(9); // zwischen den Stockwerken positioniert
  }
```

Mit der Hinzunahme einer Kamera und einer Lichtquelle kann nun die Szene mit einer geeigneten Applikation wie `flythrough` oder `raytrace` visualisiert werden (Abb.8.5und 9.2 (farbig)) und steht für die weitere Verarbeitung bereit. Die vollständige Beschreibung der Szene ist im Anhang A.1 zu finden.

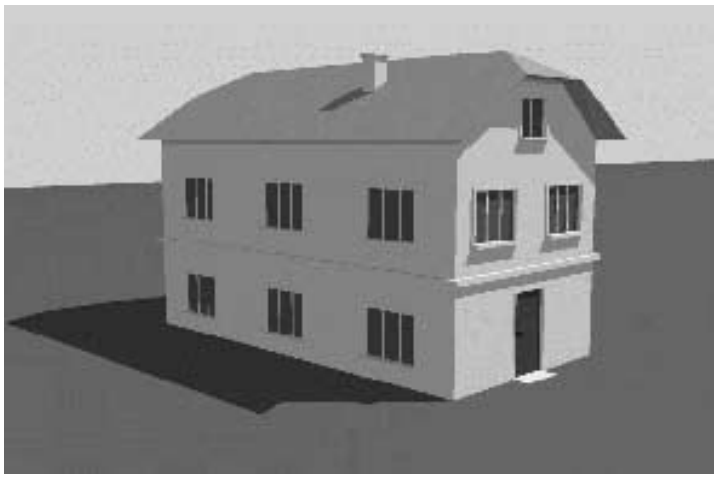

Abb. 8.5: Das fertig modellierte Haus.

Die Szene kann selbstverständlich noch detaillierter modelliert oder die Objekte können mit aufwendigen Texturen versehen werden, die der Oberfläche eine Struktur verleihen, um

eine realistischere Ausgabe zu erzielen. Die Ausgabe des Beispiels zeigt aber, dass schon mit einem korrekten geometrischen Modell ohne Texturen gute Resultate erzielt werden. Deshalb zahlt sich auch eine erhöhter Aufwand für die Modellierung der Geometrie in den meisten Fällen aus.

Als mögliche Weiterverarbeitung sei hier das Erzeugen des Strukturgraphen der Szene (Abb. 8.6) erwähnt. Damit lässt sich der hierarchische Aufbau der Szene visualisieren und die Abhängigkeiten zwischen den Objekten wiedergeben. Die Abbildung 8.6 zeigt den Strukturgraphen der Beispielsszene und stellt den genauen Aufbau des Hauses dar. Die Ausgabe ist mit dem Strukturbrowser *wxBrowser* von [Hab 96] erzeugt worden.

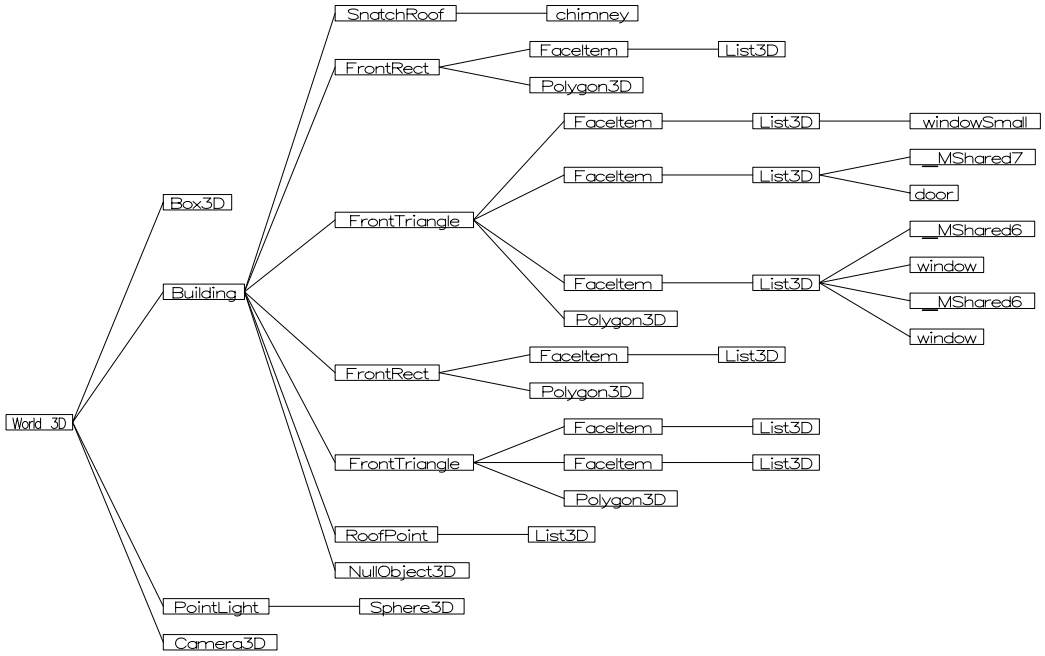

Abb. 8.6: Strukturgraph der Beispielsszene, mit `wxBrowser` erstellt.

8.2 Verarbeitung von Gebäuden

Das hier vorgestellte Beispiel erklärt anhand einer kleinen Applikation eine mögliche Verarbeitung von Szenen mit Gebäuden. Die Applikation `buildingPlacer` soll eine Szene mit Gebäuden einlesen, die Gebäude korrekt ins Geländemodell plazieren und anschliessend die Szene wieder in eine Datei zurückschreiben. Dazu wurde die Applikation `bsdlWriter` als Grundlage verwendet, die eine Szene einlesen und anschliessend zurückschreiben kann.

Einlesen der Szene

Für das Einlesen einer Szene wird in einem ersten Schritt dem Parser alle zulässigen Schlüsselwörtern angemeldet, wobei dies mit einer Komponente und/oder explizit in der

Applikation geschehen kann. Damit ist eine Trennung von applikationsspezifischen und allgemeinen Schlüsselwörtern möglich. Die Schlüsselwörter für die Gebäudemodellierung werden bis auf ein paar Einzelfälle vollständig in dafür zuständigen Komponente, dem Parser, angemeldet. So ist gewährleistet, dass alle Applikationen in BOOGA Szenen mit Gebäuden korrekt lesen können.

In einem zweiten Schritt wird die Szenenbeschreibung geparst und die entsprechende Welt (Szene) im Speicher aufgebaut. Dabei können fehlerhafte Beschreibungen unterschiedliche Parsermeldungen provozieren, wobei Warnungen übergangen werden und schwerwiegendere Fehler zu einem Abbruch der Applikation führen können. Bei einem Gebäude beispielsweise wird bei einer negativen Gebäudehöhe jegliche Realisation des Gebäudes unterdrückt und eine entsprechende Fehlermeldung ausgegeben. Es liegt dann beim Benutzer, eine positive Höhe zu definieren. So können schon viele Fehler beim Einlesen erkannt werden, die sonst erst bei der Bearbeitung der Szene auftreten und die Applikation möglicherweise zum Absturz führen können.

Bearbeitung der Szene

Ist die Szene einmal eingelesen und die Welt aufgebaut, kann diese nun verarbeitet werden. Die hier vorgestellte Applikation soll Gebäude in das bestehende Gelände der Szenen plazieren. Dazu müssen die Gebäude zuerst aus der Welt aufgesammelt werden. Das Aufsammeln geschieht mit einem *collector*, der einen bestimmten Objekttyp durch Traversierung der Welt aufsammelt. Hier kann aber nicht auf einen Standard-*collector* zurückgegriffen werden, weil auch Gebäude, die in einem anderen Gebäude enthalten sind, aufgesammelt werden. Damit würden auch diese Gebäude neu positioniert werden und die Modellierung der Gebäude, die diese enthalten würden, verändert. Dazu wurde ein eigener *collector* konzipiert, der beim Erreichen eines Gebäudes im Szenengraph nicht mehr tiefer herunter steigt (*prune*).

```
CollectBuilding buildingCollector;
buildingCollector.execute(world3D);

for (buildingCollector.first(); !buildingCollector.isDone(); buildingCollector.next())
  buildingCollector.getObject()->turnOff();

world3D->getObjects()->computeBounds();
```

Für das weitere Vorgehen werden alle Gebäude ausgeschaltet, so dass sie unsichtbar sind und damit von keinem Strahl getroffen werden. So kann man sich das Entfernen und das Wiedereinsetzen der Gebäude aus bzw. in die Szene ersparen. Die jetzt noch sichtbaren Objekte bilden das Gelände, auf dem die Gebäude positioniert werden sollen.

Nun wird für jedes einzelne Gebäude und für jeden einzelnen Eckpunkt der Grundfläche ein Strahl erzeugt. Der Strahl mit Richtung $(0, 0, -1)$ wird oberhalb des Geländes (maximaler z-Wert der *boundingbox*) und den x- und y-Werten des Punktes der Grundfläche ausgesendet, wobei allgemeine Transformationen des Gebäudes berücksichtigt werden müssen. Der tiefste getroffene Geländepunkt dient als Referenzpunkt des Gebäudes bezüglich der z-Koordinate. Somit ist die benötigte Translation mit einer einfachen Vektorsubtraktion bestimmbar.

```
for (buildingCollector.first(); !buildingCollector.isDone(); buildingCollector.next()){
  building = buildingCollector.getObject();
  tmp = building->getVertices();
  z = 0;
  for(long j=0;j<tmp.count();j++){
    c = tmp.item(j)*(building->getTransform().getTransMatrix());
    ray = Ray3DFactory::createRay(a+Vector3D(c.x(),c.y(),0),b);
    if (world3D->getObjects()->intersect(*ray) &&
        (j == 0 || c.z() - ray->getHitPoint().z() > z))
      z = c.z()-ray->getHitPoint().z();
    delete ray;
  }
  if (!equal(z,0,.001)){ // Damit wird unnoetiges Positionieren unterdrueckt
    building->addTransform(TransMatrix3D::makeTranslate(0,0,-z));
    count++;
  }
}
```

In einem letzten Schritt werden alle Gebäude wieder aktiviert, um den Ausgangszustand der Szene zu erreichen.

```
for (buildingCollector.first(); !buildingCollector.isDone(); buildingCollector.next())
  buildingCollector.getObject()->turnOn();
```

Zurückschreiben der Szene

Die Welt kann mit den neu positionierten Gebäuden mit Hilfe einer Standardkomponente von *BOOGA* als Szenenbeschreibung zurückgeschrieben werden. Dabei ist zu beachten, dass die ursprünglichen Textformatierungen und alle Kommentare der ursprünglichen Szenenbeschreibung in BSDL verloren gehen. Ausserdem kann das Zurückschreiben nicht in jedem Fall vollständig gelingen, weil einige Klassen diesen Dienst noch nicht anbieten.

Die vollständige Applikation `buildingPlacer` ist im Anhang A.3 zu finden und wurde für die Szene `bern.bsdl3` (Abb. 9.1) eingesetzt, da alle Gebäude auschliesslich in der xy-Ebene definiert wurden.

Kapitel 9

Farbbilder

Dieses Kapitel beinhaltet einige Szenen mit Gebäuden, die die Möglichkeiten der Gebäudemodellierung illustrieren.

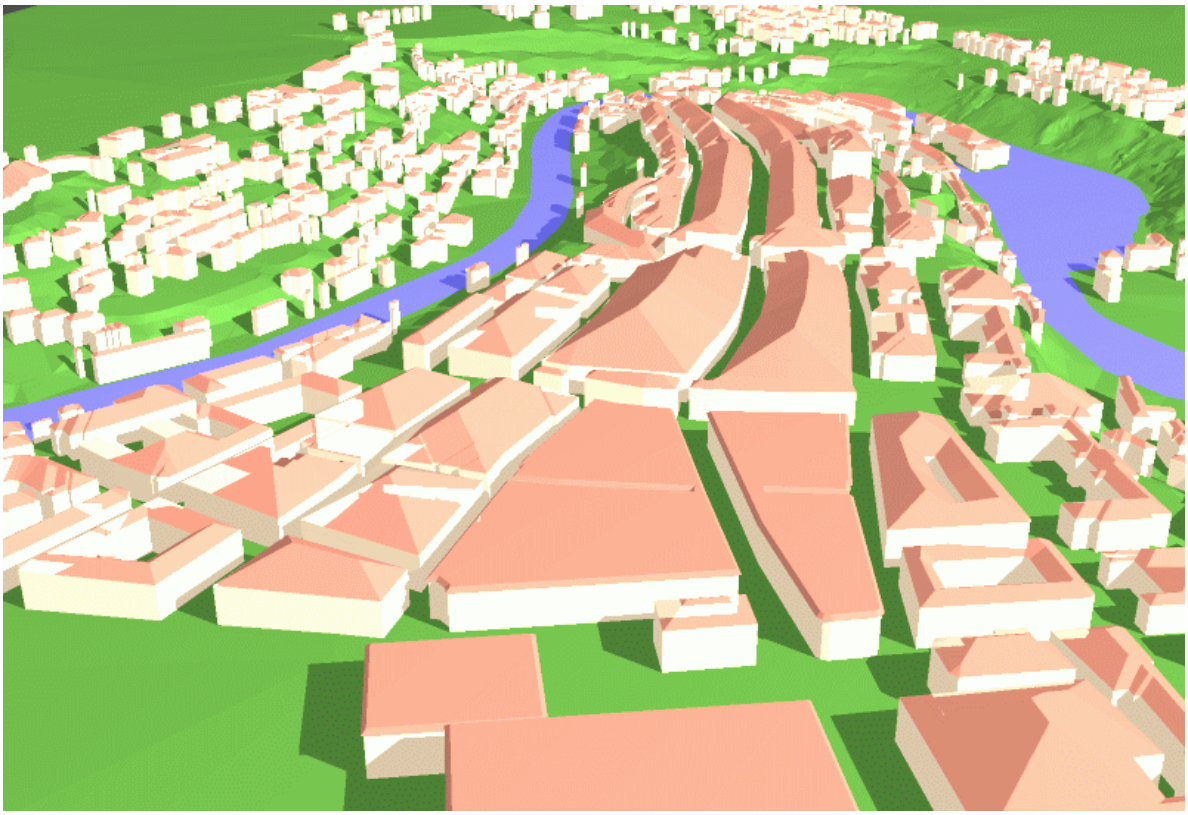

Abb. 9.1: Altstadt von Bern (`bern.bsdl3`).

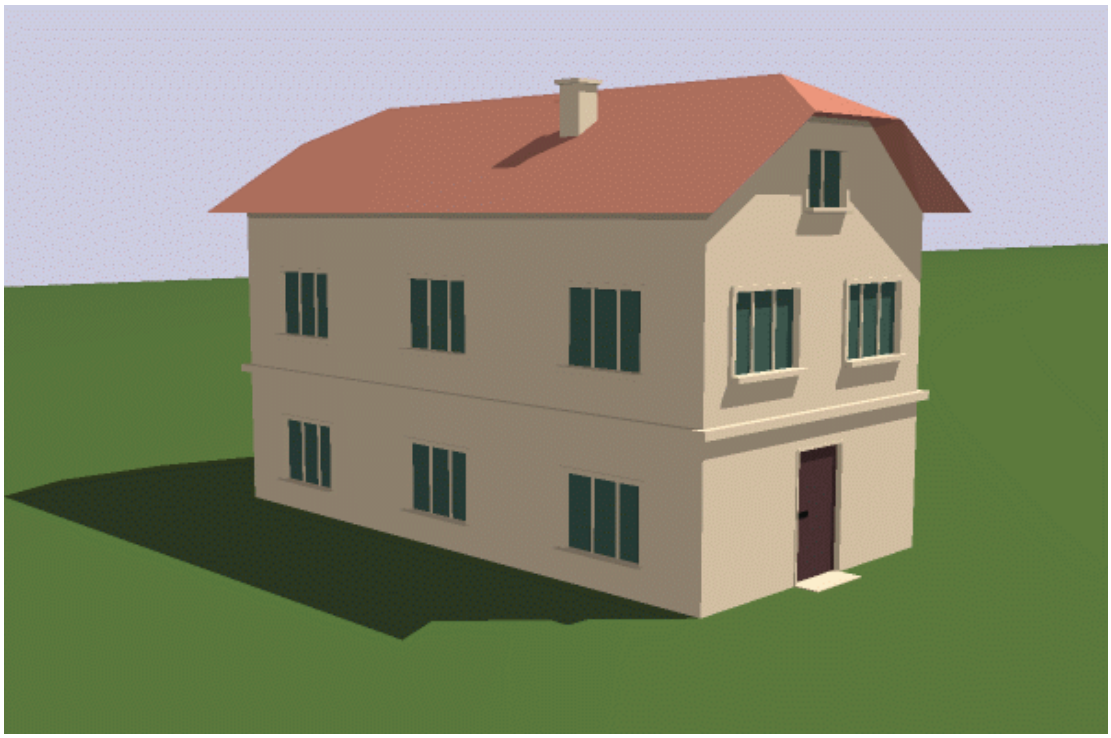

Abb. 9.2: Einfaches Haus (`building_nice.bsdl3`).

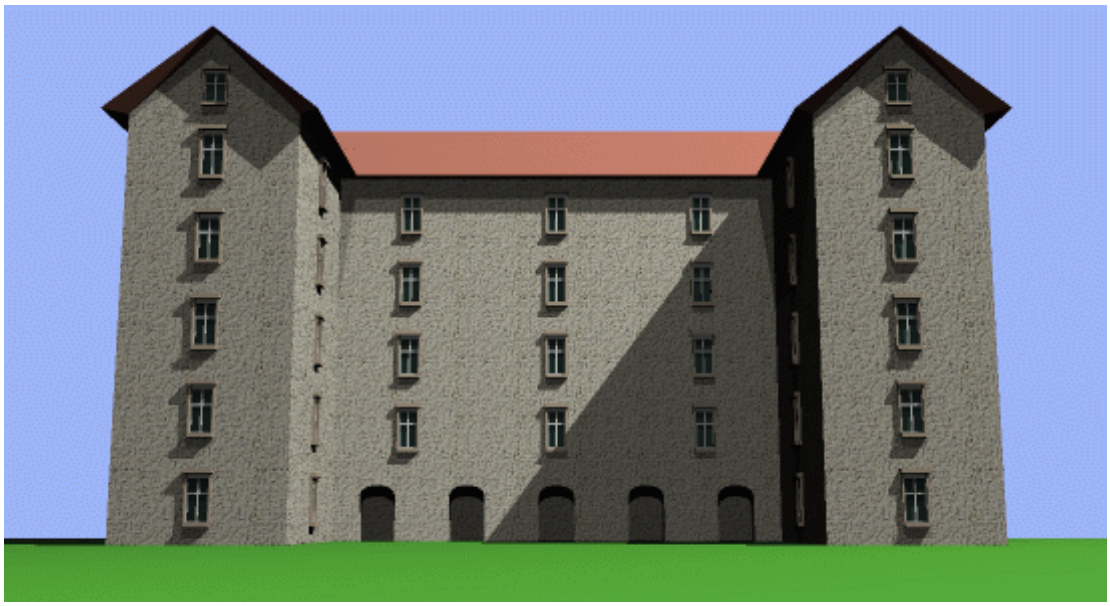

Abb. 9.3: Kaserne mit einer Laube (`building_arbour.bsdl3`).

Abb. 9.4: Zytglogge von Bern (`building_zeit.bsdl3`).

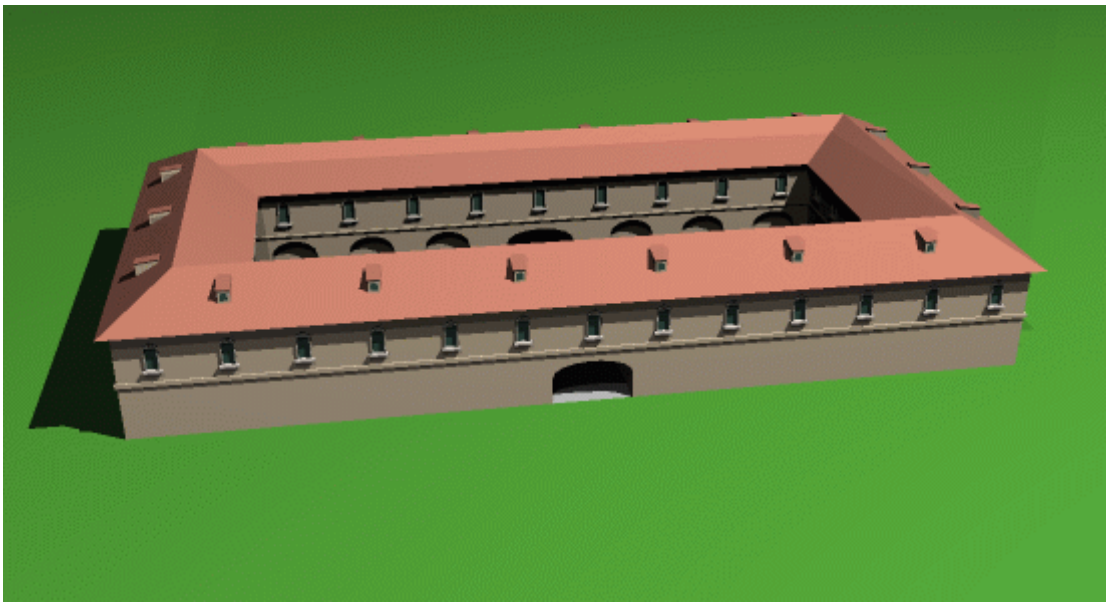

Abb. 9.5: Gebäude mit Durchgang und Lauben (`building_example.bsdl3`).

Abb. 9.6: Das IAM der Universität Bern (`uni.bsdl3`).

Teil III

Implementierung der Gebäudemodellierung unter BOOGA

Kapitel 10

Design Patterns

Dieses Kapitel geht kurz auf *design patterns* ein und erklärt das für die Implementation des objektorientierten Modellierungsansatzes mit BOOGA wichtige *proxy pattern* [Gam 95].

> *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.* Christopher Alexander [Alex 77], Architekt.

10.1 Einführung und Motivation

Das Entwickeln von objektorientierter Software ist nicht einfach, und das Entwickeln von objektorientierter wiederverwendbarer Software ist um so schwerer. Es müssen passende Objekte gefunden und mit einer sinnvollen Auflösung in Klassen aufgeteilt werden. Das Design der Software sollte einerseits das Problem spezifisch lösen, aber anderseits für Änderungen und Neuerungen offen sein. Die Praxis hat gezeigt, dass ein flexibles und wiederverwendbares Design nicht im ersten Anlauf gelingt, sondern durch einen iterativen Prozess und alten Lösungen gefunden werden kann.

Erfahrene Entwickler wissen, dass man das Rad nicht neu erfinden und besser auf bestehenden Lösungen aufbauen sollte. Dabei werden beim Design gewisse Muster (*patterns*) von Klassen und Kommunikationen zwischen Objekten immer wieder eingesetzt. Diese *patterns* lösen spezielle Probleme und erlauben eine flexibles und wiederverwendbares Design. Der Entwickler kann so mit schon bewährten *patterns* neue Probleme elegant lösen, ohne die *patterns* neu zu entdecken.

Design patterns vereinfachen die Wiederverwendung von erfolgreichen Designs und Architekturen. Sie sind auch bei der Suche nach alternativen Lösungen nützlich, um ein wiederverwendbares und flexibles Design anzustreben. Zusätzlich erleichtern *design patterns* das Benennen von Klassen und Objekten, insbesondere ermöglicht es Entwicklern eine gemeinsame, klar definierte Design-Sprache zu sprechen. Damit können Entwickler schneller und sicherer ein gutes Design für ihre Software entwickeln.

10.2 *Proxy Pattern*

Das *proxy pattern* ist ein Platzhalter oder Vertreter, das den Zugriff auf eine Objekt kontrolliert. Für den Benutzer ist das *proxy* durchsichtig, d.h. der Benutzer greift via dem *proxy* auf das Objekt zu. Das *proxy pattern* erlaubt das Zugreifen oder das Anfragen auf ein Objekt zu steuern, indem bei einem Aufruf interne Arbeiten verrichtet werden und entsprechend reagiert werden kann. Damit können beispielsweise Interfaces geschrieben werden, die mit einer Übersetzungstabelle arbeiten, oder intelligente Platzhalter für das Einbinden von Bildern in Textdokumenten (Abb. 10.1) entwickelt werden, die ein Bild nur bei Bedarf in den Hauptspeicher laden. Weiter übernimmt das *proxy* normalerweise die Verantwortung über das Objekt, d.h. es muss sich um die Initialisierung, die Erzeugung und das Vernichten des Objektes kümmern.

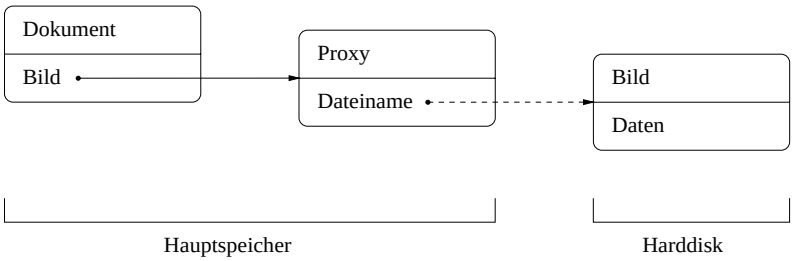

Abb. 10.1: Das *virtual proxy* als Platzhalter für ein Bild.

Die folgende Aufzählung zeigt übliche Einsatzgebiete eines *proxy pattern*:

- Das *remote proxy* ermöglicht eine lokale Repräsentation eines Objektes eines anderen Adressraumes.
- Das *virtual proxy* erzeugt ein aufwendiges, komplexes Objekte auf Anfrage (*on demand*), oder schreibt ein Objekt nur falls nötig zurück (*copy-on-write*).
- Das *protection proxy* kontrolliert den Zugriff auf das Objekt entsprechend den Zugriffsrechten.
- Das *smart reference* ist ein Zeiger, der bei einem Zugriff auf das Objekt zusätzliche Aktionen ausführt.

Kapitel 11

Texturen

Das Kapitel befasst sich mit dem Texturieren von Oberflächen und beschränkt sich auf *2D texture mapping* mit Polygonen.

11.1 Polygon in R^3 texturieren

Das Texturieren ist eine Methode, die die Eigenschaften der Oberfläche eines Körpers von Punkt zu Punkt verändert, um ihr im Erscheinungsbild Details zu verleihen. Damit lassen sich geometrische Details nachempfinden, ohne diese wirklich geometrisch modellieren zu müssen. Diese Variation von Punkt zu Punkt kann durch Abbilden von Mustern (2D *texture mapping*) auf die Oberfläche eines Objektes geschehen oder durch lokales Stören der geometrischen Struktur der Oberfläche erzielt werden.

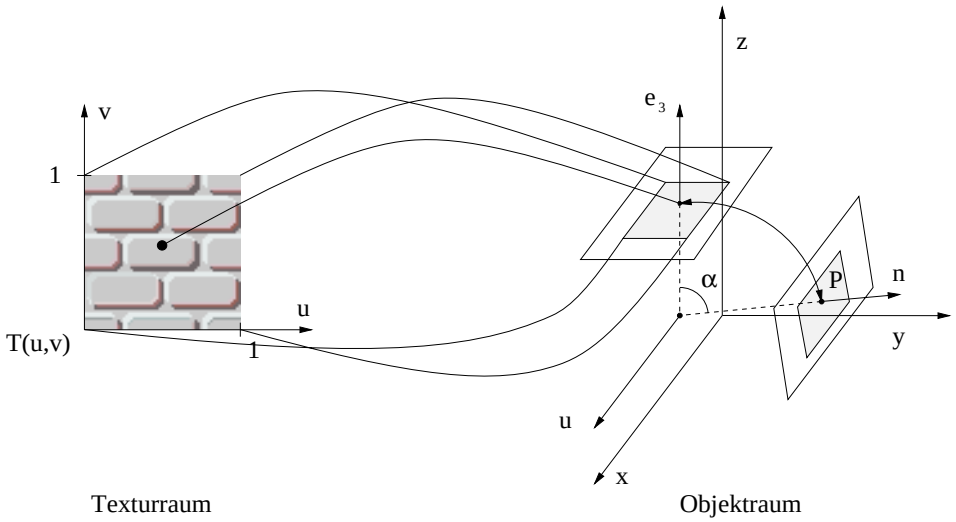

Abb. 11.1: Abbildung zwischen Texturraum und Objektraum.

Die Idee des 2D *texture mapping* besteht darin, dass eine 2D Textur im Texturraum $T(u, v)$ mit $0 \leq u, v \leq 1$ abgebildet wird. Die Normalisierung des Texturraumes $T(u, v)$ erlaubt die Unabhängigkeit zwischen dem Abbilden der Textur (*mapping*) und dem Texturbeschrieb, insbesondere können so Bilder ($bitmap[i, j]$) als eine Menge von $n_x \times n_y$

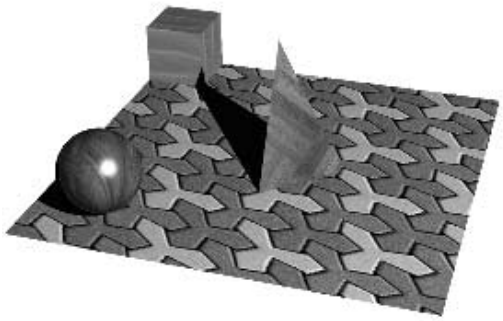

Abb. 11.2: Szene mit texturierten Objekten.

Punkten ($bitmap[i,j]$) auf Objekte abgebildet werden. Der Texturwert $T(u,v)$ lässt sich wie folgt bestimmen:

$$T(u,v) = bitmap[\lfloor u(n_x - 1)\rfloor, \lfloor v(n_y - 1)\rfloor] \tag{11.1}$$

Die Schwierigkeit des *mapping* hängt stark von der Geometrie des Objektes ab, wenn aus einem Punkt $P(x,y,z)$ im Objektraum auf den Texturwert $T(u,v)$ geschlossen werden muss. Die Zuordnung wird mit einer *mapping*-Funktion

$$m \;:\; (x,y,z) \rightarrow (u,v) \tag{11.2}$$

definiert.

Der hier behandelte Spezialfall erlaubt ein einfaches *mapping*, weil das Polygon als planare Oberfläche angesehen wird. Das *mapping* lässt sich mit Hilfe einer Rotation und einer Projektion auf den 2D-Fall zurückführen, indem das Polygon in eine zur xy-Ebene parallele Lage gebracht wird und die z-Koordinate weggelassen wird.

Die normierte Rotationsache $\vec{u}$ lässt sich mit dem Normalvektor $\vec{n}$ des Polygons auf folgende Weise bestimmen:

$$\vec{u} = \begin{cases} (1,0,0) & : \; \vec{n} = (0,0,0) \\ \frac{(-n_y,n_x,0)}{|(-n_y,n_x,0)|} & : \; \text{sonst} \end{cases} \tag{11.3}$$

Der Rotationswinkel α entspricht dem Zwischenwinkel von $\vec{n}$ und dem Einheitsvektor e_3.

$$\alpha = cos^{-1}(n_z) \tag{11.4}$$

In einem ersten Schritt wird der Punkt $P(x,y,z)$ des Polygons in die richtige Lage gebracht, indem $\vec{u}$ und α in die Formel der allgemeinen Rotation eingesetzt:

$$\begin{aligned} x' &= (u_x^2 + \cos(\alpha)(1 - u_x u_x), u_x u_y (1 - \cos(\alpha)), u_y \sin(\alpha)) \cdot (x,y,z) \\ y' &= (u_x u_y (1 - \cos(\alpha)), u_y u_y + \cos(\alpha)(1 - u_y^2), -u_x \sin(\alpha)) \cdot (x,y,z) \end{aligned} \tag{11.5}$$

Mit dem Weglassen der z-Koordinate wird der Punkt von R^3 nach R^2 projiziert. In einem zweiten Schritt wird der Punkt (x', y') am Ursprung um β gedreht, mit (s_x, s_y) skaliert und durch anschliessendes Weglassen der ganzzahligen Werte in den Texturraum $T(u, v)$ abgebildet. Damit lässt sich die Textur beliebig drehen, skalieren und fortsetzen.

$$(u, v) = (\lfloor (x' \cos(\beta) - y' \sin(\beta)) s_x \rfloor, \lfloor (x' \sin(\beta) + y' \cos(\beta)) s_y \rfloor) \tag{11.6}$$

Nun kann der Texturwert mit (11.1) aus dem Bild herausgelesen werden.

Bemerkung

Das hier vorgestellte Texturieren von Polygonen kann auch auf andere geometrische Objekte angewendet werden, falls von jedem Punkt auch die dazugehörige Normale $\vec{n}$ bestimmt werden kann. Bei gekrümmten Flächen können unerwünschte Verzerrung (vgl. Kugel, Abb. 11.2) entstehen und die *mapping*-Funktion liefert in der Umgebung von $\vec{n} = (0, 0, -1)$ keinen stetigen Übergang.

Kapitel 12

Vergleich von geometrischen Objekten

In diesem Kapitel werden geometrische Objekte aus R^2 und R^3 miteinander verglichen [PrSh 88, Rour 94].

12.1 Schnitt zweier Strecken in R^2

Der Schnitt (Abb. 12.1) zweier Strecken $\overline{P_1P_2}$ und $\overline{Q_1Q_2}$ wird in R^2 wie folgt bestimmt:

$$\text{Sei} \quad p(\vec{t}) = P_1 + \vec{a} \cdot t \qquad \text{mit} \quad \vec{a} = P_2 - P_1,\ t \in [0,1] \text{ und} \tag{12.1}$$

$$\vec{q}(s) = Q_1 + \vec{b} \cdot s \qquad \text{mit} \quad \vec{b} = Q_2 - Q_1,\ s \in [0,1], \tag{12.2}$$

$$\text{so muss } \vec{p}(t) = \vec{q}(s) \text{ gelten.} \tag{12.3}$$

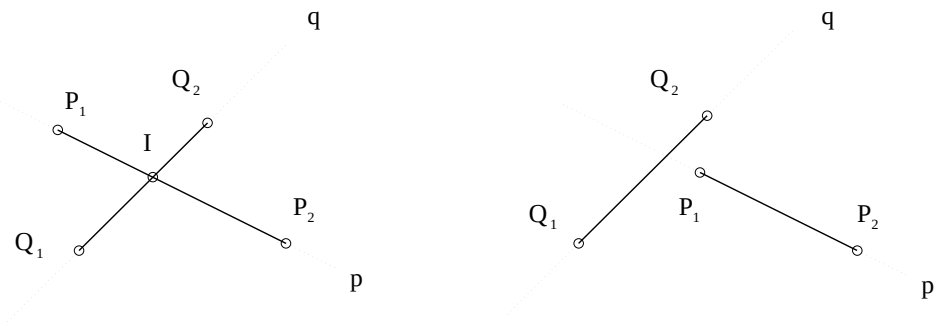

Abb. 12.1: Schnitt zweier Strecken.

Im Falle eines Schnittes ist folgende Bedingung erfüllt (vgl. $orient(A,B,D)$ aus [Rour 94]):

$$orient(P_1,P_2,Q_2) \cdot orient(P_2,P_1,Q_1) \geq 0 \ \wedge$$
$$orient(Q_1,Q_2,P_2) \cdot orient(Q_2,Q_1,P_1) \geq 0, \tag{12.4}$$

$$orient(A,B,C) = \begin{cases} 1 & : \overline{ABC} \text{ positiv orientiert} \\ -1 & : \overline{ABC} \text{ negativ orientiert} \\ 0 & : \text{sonst} \end{cases} \quad \text{wobei} \tag{12.5}$$

Der Schnittpunkt I lässt sich durch lösen der Gleichung (12.3) mit (12.1, 12.2) bestimmen. Das Auflösen nach t ergibt:

$$t = \frac{a_y(P_{1x} - Q_{2x}) - a_x(P_{1y} - Q_{2y})}{b_x a_y - b_y a_x} \quad \text{mit} \quad b_x a_y - b_y a_x \neq 0 \tag{12.6}$$

Falls $b_x a_y - b_y a_x = 0$ gilt, erhält man keinen Schnittpunkt, weil die beiden Strecken $\overline{P_1P_2}$ und $\overline{Q_1Q_2}$ kollinear sind. Anderenfalls ist t in $\vec{p}(t)$ eingesetzt der gesuchte Schnittpunkt I.

12.2 Schnitt zweier Strecken in R^3

Der Schnitt (Abb. 12.2), falls vorhanden, zweier Strecken $\overline{P_1P_2}$ und $\overline{Q_1Q_2}$ wird in R^3 wie folgt bestimmt:

$$\text{Sei} \quad \vec{p}(t) = P_1 + \vec{a} \cdot t \quad \text{mit} \quad \vec{a} = P_2 - P_1,\ t \in [0,1] \text{ und} \tag{12.7}$$

$$\vec{q}(s) = Q_1 + \vec{b} \cdot s \quad \text{mit} \quad b = Q_2 - Q_1,\ s \in [0,1], \tag{12.8}$$

$$\text{so muss} \quad p(t) = q(s) \text{ gelten.} \tag{12.9}$$

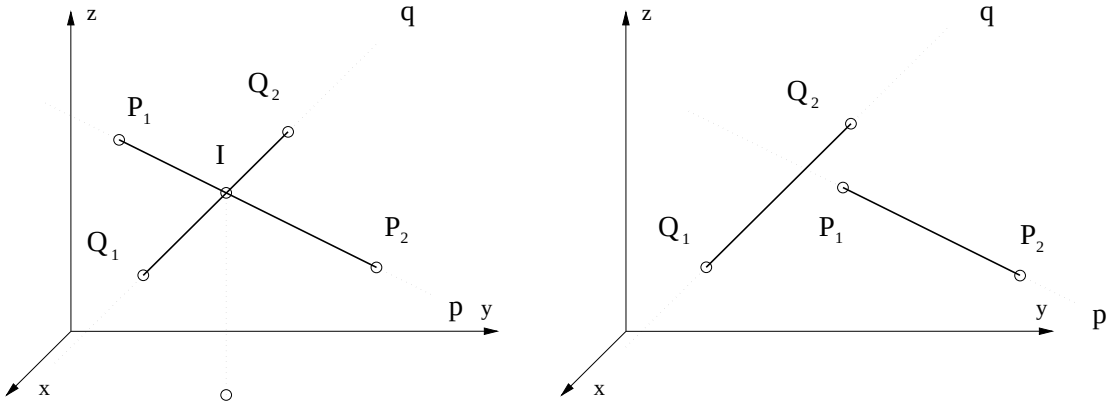

Abb. 12.2: Schnitt zweier Strecken.

Falls sich die Strecken schneiden, müssen sie in einer Ebene liegen. Diese Eigenschaft lässt sich durch Vergleichen der Flächennormalen der vier Dreiecke $\overline{P_1P_2Q_2}$, $\overline{P_2P_1Q_1}$, $\overline{Q_1Q_2P_2}$ und $\overline{Q_2P_1Q_1}$ überprüfen.

$$\text{Sei} \quad \vec{n}_1 = normal(P_1, P_2, Q_2), \tag{12.10}$$

$$\vec{n}_2 = normal(P_2, P_1, Q_1), \tag{12.11}$$

$$\vec{n}_3 = normal(Q_1, Q_2, P_2), \tag{12.12}$$

$$\vec{n}_4 = normal(Q_2, Q_1, P_1), \tag{12.13}$$

$$normal(A,B,C) = \begin{cases} \frac{\vec{n}}{|\vec{n}|} & : |\vec{n}| > 0 \\ (0,0,0) & : \text{sonst} \end{cases} \quad \text{mit} \quad \text{und } \vec{n} = (C-B) \times (A-B) \tag{12.14}$$

Im Falle eines Schnittes ist mindestens eine der folgenden Bedingungen erfüllt:

$$\vec{n}_1 = \vec{n}_2 = \vec{n}_3 = \vec{n}_4 \wedge |\vec{n}_1| > 0 \tag{12.15}$$

$$\vec{n}_1 = \vec{n}_3 \wedge |\vec{n}_1| > 0 \wedge |\vec{n}_2| = |\vec{n}_4| = 0 \tag{12.16}$$

$$\vec{n}_1 = \vec{n}_4 \wedge |\vec{n}_1| > 0 \wedge |\vec{n}_2| = |\vec{n}_3| = 0 \tag{12.17}$$

$$\vec{n}_2 = \vec{n}_3 \wedge |\vec{n}_2| > 0 \wedge |\vec{n}_1| = |\vec{n}_4| = 0 \tag{12.18}$$

$$\vec{n}_2 = \vec{n}_4 \wedge |\vec{n}_2| > 0 \wedge |\vec{n}_1| = |\vec{n}_3| = 0 \tag{12.19}$$

Die Indizes x', y' und z' sind so zu wählen, dass $|a_{x'}|, |a_{y'}| \geq |a_{z'}|$ gilt. Der Schnittpunkt I lässt sich durch lösen der Gleichung (12.9) mit (12.7, 12.8) bestimmen. Das Auflösen nach t ergibt:

$$t = \frac{a_{y'}(P_{1x'} - Q_{2x'}) - a_{x'}(P_{1y'} - Q_{2y'})}{b_{x'}a_{y'} - b_{y'}a_{x'}} \ , \quad \text{mit} \quad b_{x'}a_{y'} - b_{y'}a_{x'} \neq 0 \tag{12.20}$$

Falls $b_{x'}a_{y'} - b_{y'}a_{x'} = 0$ gilt, erhält man keinen Schnittpunkt, weil die beiden Strecken $\overline{P_1P_2}$ und $\overline{Q_1Q_2}$ kollinear sind. Anderenfalls ist t in $p(t)$ eingesetzt der gesuchte Schnittpunkt I.

12.3 Punkt in Polygon in R^2

Die Frage, ob ein Punkt innerhalb oder ausserhalb eines Polygons liegt (Abb.12.3), kann auf ein Zählen der Schnitte des Polygon mit einem vom Punkt ausgehenden Strahl zurückgeführt werden. Falls die Anzahl der Schnitte ungerade ist, befindet sich der Punkt innerhalb. Anderenfalls ist der Punkt ausserhalb des Polygons.

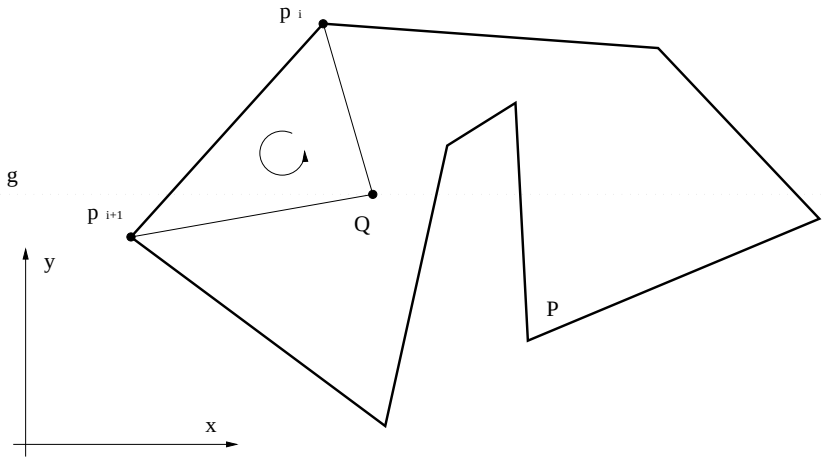

Abb. 12.3: Punkt innerhalb Polygon.

Gegeben seien der Punkt Q und das Polygon $P = (p_0, p_1, p_2, \cdots, p_{n-1})$ mit $p_i = (x_i, y_i)$. Sei eine Gerade g parallel zur x-Achse und durch Q definiert. Kante i $\overline{p_i p_{((i+1) \ mod \ n)}}$, die g links von Q schneidet, erfüllt folgende Bedingung:

$$\begin{aligned}&(y_i > Q_y \geq y_{((i+1) \ mod \ n)} \wedge area(p_i, p_{((i+1) \ mod \ n)}, Q) > 0) \ \vee \\ &\quad (y_{((i+1) \ mod \ n)} < Q_y \leq y_i \wedge area(p_i, p_{((i+1) \ mod \ n)}, Q) < 0)\end{aligned} \tag{12.21}$$

mit

$$area(A, B, C) = (C - B) \times (A - B) \tag{12.22}$$

Der Punkt Q liegt innerhalb von P, falls die Summe

$$\sum_{i=0}^{n-1} \begin{cases} 1 & : \text{ Kante } i \text{ erfüllt (12.21)} \\ 0 & : \text{ sonst} \end{cases} \tag{12.23}$$

ungerade ist. Anderenfalls liegt der Punkt ausserhalb.

12.4 Schnitt zweier Polygone in R^3

Der Schnitt I (Abb. 12.4) zweier planarer Polygone $P = (p_0, p_1, p_2, \cdots, p_{n-1})$ mit $p_i = (x_i, y_i, z_i)$ und $Q = (q_0, q_1, q_2, \cdots, q_{m-1})$ mit $q_j = (x_j, y_j, z_j)$ sei konvex und es kann ausgeschlossen werden, dass ein Eckpunkt eines Polygons innerhalb von I liegt. Der Schnitt I lässt sich auf den Durchschnitt von $\overline{AB}$ und $\overline{CD}$ zurückführen:

$$I = \overline{AB} \cap \overline{CD} \tag{12.24}$$

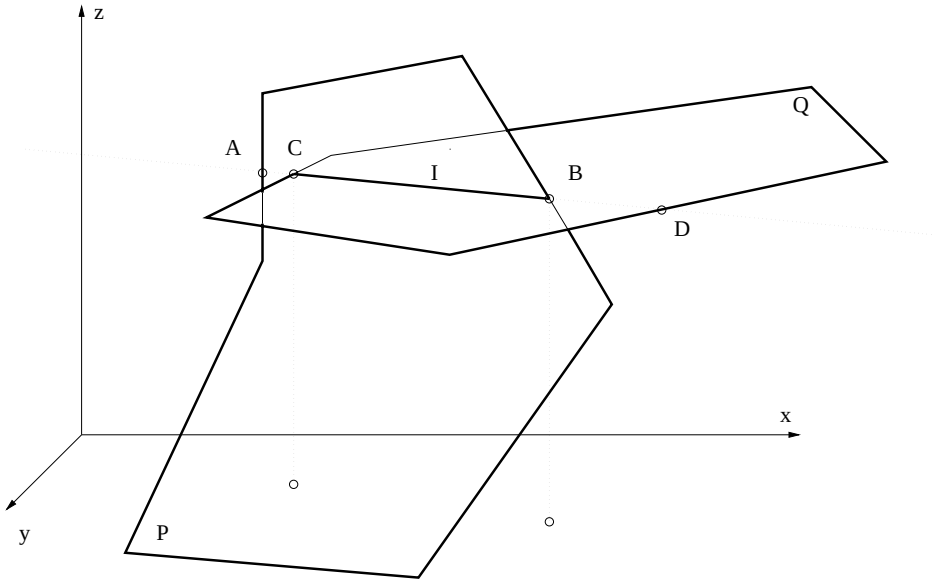

Abb. 12.4: Schnitt zweier Polygone.

$\overline{AB}$ und $\overline{CD}$ werden durch schneiden der Polygone mit der Ebene des verbleibenden Polygons bestimmt:

$$\overline{AB} = P \cap E_q, \qquad \text{wobei } Q \in E_q \tag{12.25}$$

$$\overline{CD} = Q \cap E_p, \qquad \text{wobei } P \in E_p \tag{12.26}$$

Der Schnitt eines Polygons P' und einer Ebene E' lässt sich auf das Schneiden der Kanten des Polygons und der Ebene zurückführen. Mit der Voraussetzung, dass $P \cap Q$ konvex ist und $\forall i \forall j \; p_i, q_j \notin I^{int}$ gilt, muss auch $P' \cap E'$ konvex sein und $\forall i \; p'_i \notin (P' \cap E')^{int}$

gelten. Da $P' \cap E'$ konvex ist und $\forall i\ p'_i \notin (P' \cap E')^{int}$ gilt, gibt es höchstens zwei Kanten $\overline{p'_i p'_{((i+1)\ mod\ n)}}$, die folgende Gleichung erfüllen:

$$E' : xn'_x + yn'_y + zn'_z + D' = 0, \tag{12.27}$$

$$\text{mit } (x, y, z) = p'_i + \vec{a} \cdot t,\ t \in]0, 1] \text{ und } \vec{a} = p'_{((i+1)\ mod\ n)} - p'_i \tag{12.28}$$

Das Auflösen nach t ergibt:

$$t = \frac{\vec{n} \cdot U - \vec{n} \cdot p'_i}{\vec{n} \cdot \vec{a}}, \qquad \text{wobei } \vec{n} \cdot \vec{a} \neq 0,\ U \in E' \tag{12.29}$$

Falls Zähler und Nenner von (12.29) Null sind, liegt die Kante in der Ebene. In diesem Fall wird $p'_{((i+1)\ mod\ n)}$ als Schnittpunkt angenommen. Anderenfalls erhält man den Schnittpunkt durch Einsetzen von t in den Term $p'_i + \vec{a} \cdot t$ unter Berücksichtigung von $t \in]0, 1]$ und $\vec{n} \cdot \vec{a} \neq 0$.

Kapitel 13

Definition der Objekte

Dieses Kapitel erklärt alle für die Gebäudemodellierung notwendigen, neu eingeführten Objekte und ihre entsprechenden Klassen. Neben der Definition wird eine Sprachdefinition gegeben und anhand eines Beispiels in BSDL[1] erläutert.

13.1 Einführung

Die Definition der Objekte und die Entwicklung deren Klassen baut auf dem Modellierungsansatz des Kapitels 6. Die Modellierung von Gebäuden unter *BOOGA* kann einerseits mit BSDL in Form einer Textdatei beschrieben werden und mit einer Applikation wie `flythrough` eingelesen und visualisiert werden. Anderseits kann die Welt mittels einer Applikation erzeugt werden, indem diese die nötigen Objekte erzeugt und den Szenengraph selbständig aufbaut. Eine einmal erzeugt Welt kann dann auf vielfältige Weise mit weiteren Komponenten verarbeitet werden.

Die einzelnen Definitionen geben die genauen Parametrisierungen und Funktionalitäten der Objekte wieder. Es werden die Abhängigkeiten zwischen Gebäuden, anderen Gebäudeobjekten oder allgemeinen `Object3D`-Objekten erklärt. Bei den abstrakten Klassen wird auf die für die einzelnen Gruppen von Gebäudeobjekten typischen Funktionalitäten und Abhängigkeiten eingegangen. Dabei werden die grundlegenden Mechanismen zwischen den Objekten aufgezeigt. Ausserdem werden die meisten Objekte anhand eines Beispiels mit entsprechender Ausgabe erläutert. Für den genauen Aufbau mit allen Methoden und Members der einzelnen Klassen sei hier auf die Implementation verwiesen.

13.2 Das Gebäude (`Building`)

Die konkrete Klasse `Building` entspricht der Definition 7.1 aus dem Kapitel 7. Das Gebäude definiert sich aus einer Grundfläche mit Innenhöfen und der Höhe des Gebäudes. Die Definition erzeugt ein Polyeder (Abb. 13.1), das aus Seitenflächen, einem Boden und einem Deckel besteht. Das Objekt `Building` kennt folgende Funktionalitäten und Eigenschaften:

[1] **B**OOGA **S**cene **D**escription **L**anguage,
`http://iamwww.unibe.ch/~fcglib/WWW/Vorlesungen/3D/booga/bsdle.ps.gz`.

- Die Klasse `Building` dient als Definition eines allgemeinen Gebäudes anhand der Grundfläche mit all seinen Innenhöfen und der Höhe des Gebäudes. Das Gebäude kennt ein Default–Gebäude, das aus den Seitenflächen des aufgespannten Polyeders besteht und bei Bedarf[2] erzeugt wird.
- Sie führt eine eindeutige, zweidimensionale Numerierung der einzelnen Seitenflächen (Fronten) ein (vgl. Abschnitt 13.6).
- Sie verwaltet die Unterobjekte und nimmt die Funktionalität eines Aggregates wahr. Es werden neben Gebäudeobjekten, auch allgemeine `Objekt3D`-Objekte akzeptiert.
- Sie übernimmt die Verantwortung für die korrekte Erzeugung der geometrischen Realisation der Unterobjekte. Das Gebäude übergibt seinen Gebäudeobjekten die für die Erzeugung der Struktur relevanten Informationen und Daten.
- Sie Überprüft seine Definition und die Konsistenz seiner Unterobjekte.
- Sie stellt eine Cache-Struktur zur Verfügung, um ein eventuelles Default–Gebäude zwischenzuspeichern.
- Sie gibt Probleme wie das Schneiden eines Strahles oder das Berechnen der *boundingbox* an die Unterobjekte und an eine eventuelles Default–Gebäude weiter.
- Sie enthält Operationen, die das Manipulieren der Definition des Gebäudes vereinfachen. Damit lassen sich nachträglich Ecken einfügen, ersetzen oder sogar löschen. Die Unterobjekte werden entsprechend angepasst und falls nötig gelöscht.
- Sie kann ein Gebäude mit all seinen Unterobjekten mit einer leeren, geometrischen Struktur repräsentieren. So kann ein ganzes Gebäude unsichtbar gemacht werden.

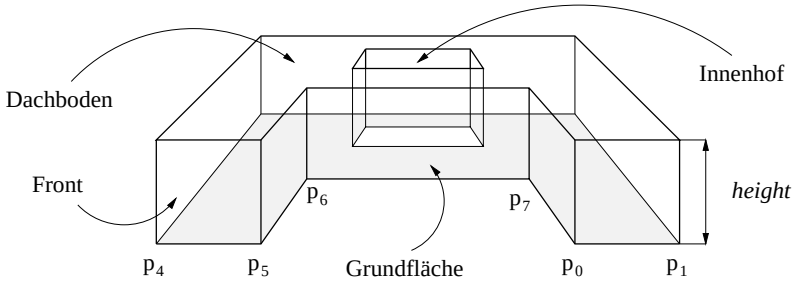

Abb. 13.1: Geometrie und Parametrisierung des Gebäudes.

[2]Funktionalität des *proxy pattern*.

Abb. 13.2: Beispiel eines einfachen Gebäude ohne Unterobjekte.

Sprachdefinition

`building` $(height, \vec{p}_0, \vec{p}_1, \vec{p}_2, \ldots, \vec{p}_{n-1})$
`{[hole` $(\vec{h}_0, \vec{h}_1, \vec{h}_2, \cdots, \vec{h}_{n-1})$`;][` *buildingobjects*`;][`*objects*`;][on;|off;][` *textures*`;]`
`[`*transforms*`;]}`

height > 0 definiert die Höhe des Gebäudes und dessen Fronten. Die $\vec{p}_i \in R^3$ definieren die Ecken der Grundfläche und bilden ein einfaches und planares Polygon, das parallel zur xy-Ebene liegt. Die Innenhöfe werden wie die Grundfläche als einfache Polygone mit dem Attribut `hole[...]` mit $\vec{h}_i \in R^3$ definiert, wobei die Grundfläche und die Innenhöfe in der gleiche Ebene liegen müssen. Das Überlappen von Innenhöfen wird nicht geprüft. Falls das Gebäude keine einzige Teilfront (Unterobjekt der Front) besitzt, erzeugt es ein Default–Gebäude, das nur aus den Seitenflächen des aufgespannten Polyeders besteht. Anderenfalls wird eine leere Struktur[3] erzeugt. *buildingobjects* sind beliebige[4] Gebäudeobjekte, wobei auch allgemeine Objekte (*objects*) als Unterobjekte akzeptiert werden. Mittels `on` und `off` lässt sich das Gebäude mit all seinen Unterobjekten sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur und *transforms* entspricht einer allgemeinen Transformation.

Beispiel

Das folgende Beispiel modelliert ein Gebäude (Abb. 13.2) mit rechteckiger Grundfläche und zwei Innenhöfen. Die Grundfläche liegt in der xy-Ebene und die Abbildung zeigt das von `Building` erzeugte Default–Gebäude, da das Gebäude keine Unterobjekte enthält.

```
building(5, // Hoehe des Gebaeudes
  [20,10,0],[-20,10,0],[-20,-10,0],[20,-10,0]){ // Grundflaeche
  // rechter Innenhof
  hole([15,5,0],[15,-5,0],[5,-5,0],[5,5,0]);
  // linker Innenhof
  hole([-15,5,0],[-5,5,0],[-5,-5,0],[-15,-5,0]);
  aBuildingObejct;  // ein beliebiges Gebaeudeobjekt
  ...
  aTexture;      // Textur des Gebaeudes und
                 //  Default-Textur der Unterobjekte
  aTranslation;  // lokale Transformation
}
```

[3] Mit `NullObject3D` realisiert.
[4] Teilfronten (`Face`) werden zwar akzeptiert, aber ergeben keinen logischen Sinn.

13.3 Das Gebäudeobjekt (`BuildingObject`)

Die abstrakte Klasse `BuildingObject` ist die Basisabstraktion der Gebäudeobjekte aus Definition 7.2. Sie übernimmt allgemeine Funktionen, die für alle Gebäudeobjekte typisch sind:

- Die Klasse `BuildingObject` dient der gemeinsamen Verankerung aller Gebäudeobjekte. So können alle Gebäudeobjekte über eine standardisierte Schnittstelle angesprochen werden.
- Sie verwaltet den Zeiger des Gebäudes, dem das Gebäudeobjekte untergeordnet ist. Es wird sichergestellt, dass jedes Gebäudeobjekt zu jeder Zeit auf alle nötigen Daten zugreifen kann[5].
- Sie nimmt die Funktion eines *proxy* wahr. Es leitet die Erzeugung der dazugehörigen geometrischen Struktur, der Dekomposition, erst bei Bedarf ein (`createSubject()`).
- Sie nimmt die Verwaltung eines Caches wahr, der für die Dekomposition des (konkreten) Gebäudeobjektes konzipiert ist.
- Sie gibt Probleme wie das Schneiden eines Strahles (`doIntersect()`) oder das Berechnen der *boundingbox* (`doComputeBounds()`) an die Dekomposition weiter.
- Sie kennt keine lokalen Transformationen, weil diese bei der Erzeugung der Dekomposition überschrieben werden. Normalerweise wird die Tranformationsmatrix von den abstrakten Klassen entsprechend berechnet und gesetzt (`doCreateSubject()`).
- Sie kann Gebäudeobjekte mit all deren Unterobjekten mit einer leeren, geometrischen Struktur repräsentieren. So kann ein Gebäudeobjekt mit all seinen Unterobjekten unsichtbar gemacht werden.

13.4 Der Boden (`Bottom`)

Die abstrakte Klasse `Bottom` übernimmt die Abstraktion aller Objekte, die in Abstimmung mit der Grundfläche des Gebäudes erzeugt werden oder von ihr abhängen. Die Klasse bietet das Attribut `height` an, das die Positionierung bezüglich der z-Achse ermöglicht. Die genaue Abhängigkeit zwischen dem Gebäudeobjekt und der Grundfläche ist in den konkreten Klassen `BottomBorder`, `BottomFlat` und `BottomUser` in der Methode `doCreateSubject()` implementiert. Die folgende Abbildung 13.3 zeigt die Klassenhierarchie, die wichtigsten Methoden und Daten der einzelnen Klassen.

[5]Mit einem Zeiger auf das Gebäude sind auch logische Abhängigkeiten zwischen Gebäudeobjekten realisierbar (siehe Bsp. `FaceTunnel`), die mit einem rein hierarchischen Baukastensystem gar nicht möglich sind.

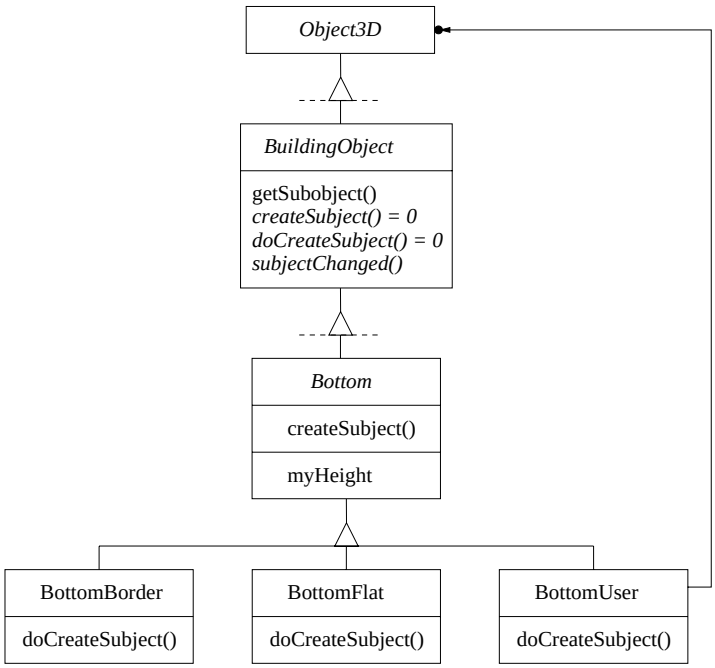

Abb. 13.3: Die Hierarchie von `Bottom`.

13.4.1 `BottomBorder`

Die konkrete Klasse `BottomBorder` modelliert anhand von vier Parametern einen Rand mit Boden (Abb. 13.4), der in Abstimmung mit der Grundfläche erzeugt wird. Innenhöfe werden für die Erzeugung der Ränder nicht berücksichtigt. Damit lassen sich "balkonähnliche Rundgänge" um das Gebäude modellieren. Im Szenengraph ist `BottomBorder` ein Sohn von `Building`.

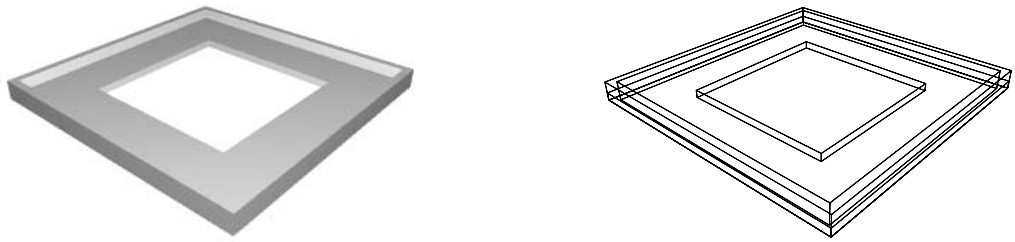

Abb. 13.4: Beispiel von `BottomBorder`.

Sprachdefinition

`bottomborder` $(ledge, borderwidth, borderheight, borderdepth)$
`{[height(`*height*`);][on;|off;][`*textures*`;]}`

Erzeugt einen Boden mit der Dicke $borderdepth > 0$, der um $ledge > 0$ aus dem Grundriss herausragt (Abb. 13.5). $borderheight > 0$ definiert die Höhe und

borderwidth > 0 die Breite des Randes. Das Attribut `height` definiert eine Translation entlang der z-Achse bezüglich der Grundfläche um *height*. Mit `on` und `off` lässt sich das Objekt sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur. Der Parameter *ledge* soll mit Umsicht gewählt werden, weil bei nicht-konvexer Grundfläche ein sich selber schneidender Rand entstehen kann.

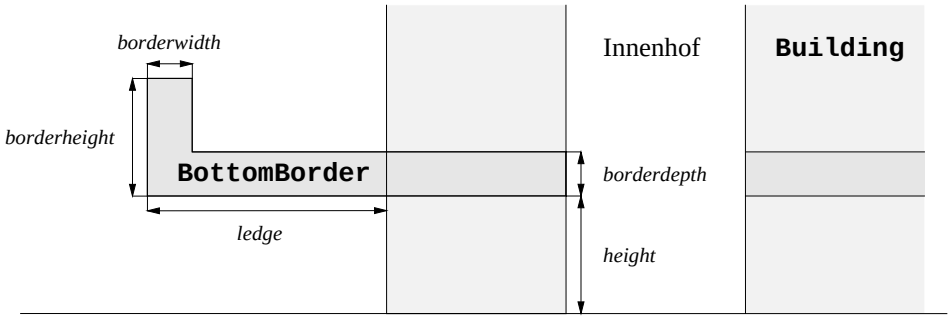

Abb. 13.5: Parametrisierung von `BottomBorder`.

Beispiel

Das folgende Beispiel modelliert den Rundgang von Abbildung 13.4.

```
building (15,[-30,-30, 0],[ 30,-30, 0],   // Hoehe und Grundflaeche
             [ 30, 30, 0],[-30, 30, 0]){  //  des Gebaeudes
  // Innenhof
  hole([-25,-25,0],[-25,25,0],[25,25,0],[25,-25,0]);
  ...
  // (ledge,borderwidth,borderheight,borderdepth)
  bottomborder(5,2,5,2){
    height(2);
  }
  aTexture;  // Default-Textur aller Gebaeudeobjekte
             //  und des Default-Gebaeudes
}
```

13.4.2 BottomFlat

Die konkrete Klasse `BottomFlat` modelliert einen Boden in Abstimmung mit der Grundfläche. Damit lassen sich mit dem Attribut `height` auch die Fussböden der einzelnen Stockwerke problemlos modellieren. Im Szenengraph ist `BottomFlat` ein Sohn von `Building`.

Sprachdefinition

`bottomflat` {`[height(`*height*`);][on;|off;][`*textures*`;]`}

Kennt keine eigenen Parameter und erzeugt ein Polygon, das identisch mit der Grundfläche und all seinen Innenhöfen ist. Das Attribut `height` definiert eine Translation des modellierten Bodens entlang der z-Achse bezüglich der Grundfläche. Mit `on` und `off` lässt sich das Objekt sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur.

Beispiel

Das Beispiel modelliert einen Boden, der eine Textur enthält und um 1 entlang der z-Achse verschoben ist.

```
building(...){  // Gebaeudedefinition
  bottomflat{
    height(1);  // Translation entlang z-Achse
    aTexture;   // Textur des Bodens
  }
  ...
}
```

13.4.3 BottomUser

Die konkrete Klasse `BottomUser` positioniert ein beliebiges Objekt (`Object3D`) in Abstimmung mit der Grundfläche. So können benutzerdefinierte Böden oder andere Objekte in Abhängigkeit der z-Koordinate der Grundfläche und dem Attribut `height` positioniert werden. Im Szenengraph ist `BottomUser` ein Sohn von `Building`.

Sprachdefinition

`bottomuser` {`[`*anObject*`;][height(`*height*`);][on;|off;][`*textures*`;]`}

Kennt keine eigenen Parameter. *anObject* ist ein Objekt vom Typ `Objekt3D` und wird entsprechend der Grundfläche und dem Attribut `height` positioniert. Mit `on` und `off` lässt sich das Objekt sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur. Lokale Transformationen werden wie bei allen anderen Gebäudeobjekten überschrieben.

Beispiel

Das Beispiel zeigt eine Kugel mit Radius 1, die genau auf der Ebene der Grundfläche liegt.

```
building(...){ // Gebaeudedefinition
  bottomuser{
    sphere(1,[0,0,10]); // beliebiges Objekt
    height(-9); // Translation entlang z-Achse
  }
  ...
}
```

13.5 Das Dach (`Roof`)

Die abstrakte Klasse `Roof` übernimmt die Abstraktion aller Objekte, die in Abstimmung mit dem Dachboden (Abb. 13.1) erzeugt werden oder von ihm abhängen und Dächer aller Art erzeugen. Die Klasse verwaltet, falls erwünscht, die Grösse des Dachvorsprungs (*ledge*) und kann die dazugehörigen Eckpunkte bestimmen, die ein Layer (Abb. 13.10) um das Polygon des Dachbodens bilden. Ausserdem kennt `Roof` das Attribut `height`, das die Positionierung bezüglich der z-Achse ermöglicht und die lokale Transformationsmatrix aller konkreten Klassen definiert. Die genaue Abhängigkeit zwischen dem Gebäudeobjekt, dem Dachboden und den Fronten ist in den konkreten Klassen `RoofBorder`, `RoofFlat`, `RoofLayer`, `RoofPlane`, `RoofPoint` und `RoofUser` in der Methode `doCreateSubject()` implementiert. Die folgende Abbildung 13.6 zeigt die Klassenhierarchie, die wichtigsten Methoden und Daten der einzelnen Klassen.

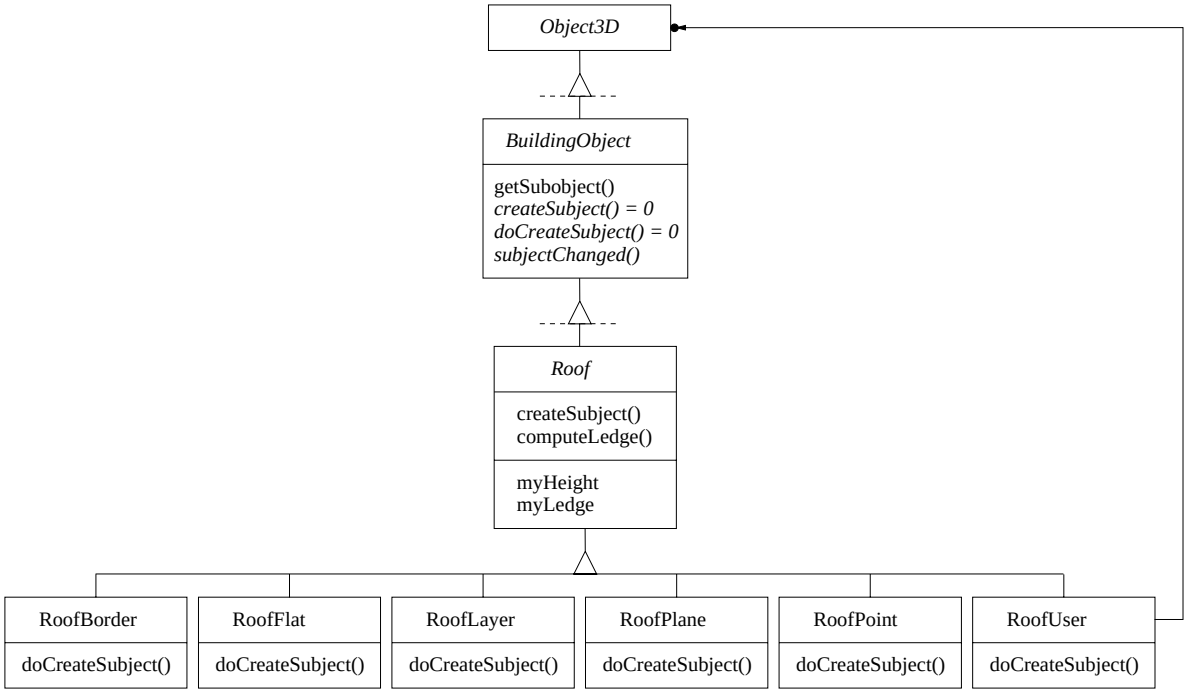

Abb. 13.6: Die Hierarchie von `Roof`.

13.5.1 `RoofBorder`

`RoofBorder` modelliert anhand von vier Parametern einen Rand mit Boden (Abb. 13.7), der in Abstimmung mit dem Dachboden erzeugt wird. Innenhöfe werden für die Erzeugung der Ränder nicht berücksichtigt. Damit lassen sich "balkonähnliche Rundgänge" um das Gebäude oder Aussichtsplattformen modellieren. Im Szenengraph ist `RoofBorder` ein Sohn von `Building` und unterscheidet sich von `BottomBorder` dadurch, dass die Positionierung vom Dachboden abhängt.

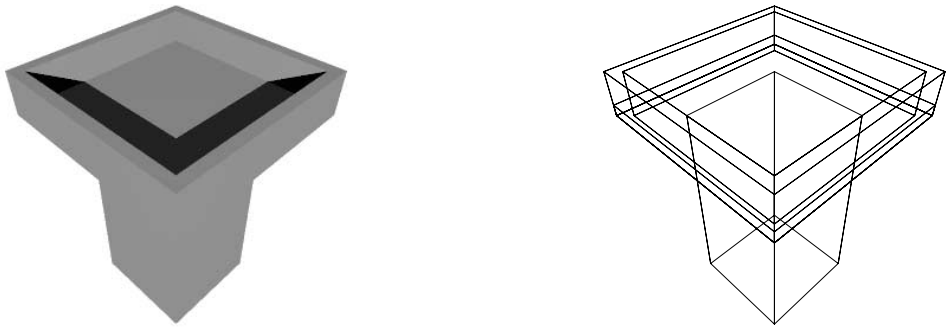

Abb. 13.7: Aussichtsplattform mit `RoofBorder` modelliert.

Sprachdefinition

`roofborder` (*ledge*, *borderwidth*, *borderheight*, *borderdepth*)
{`[height(`*height*`);][on;|off;][`*textures*`;]`}

Erzeugt einen Boden mit der Dicke *borderdepth* > 0, der um *ledge* > 0 aus dem Grundriss herausragt (Abb. 13.8). *borderheight* > 0 definiert die Höhe und *borderwidth* > 0 die Breite des Randes. Das Attribut `height` definiert eine Translation entlang der z-Achse. Mit `on` und `off` lässt sich das Objekt sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur. Der Parameter *ledge* soll mit Umsicht gewählt werden, weil bei nicht-konvexer Grundfläche ein sich selber schneidender Rand entstehen kann.

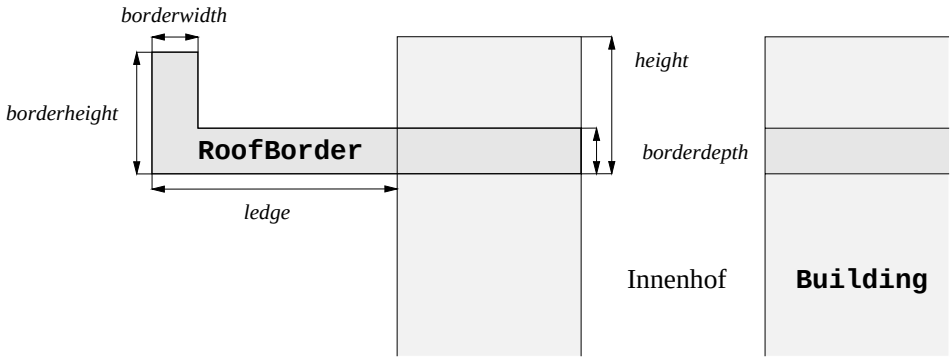

Abb. 13.8: Parametrisierung von `RoofBorder`.

Beispiel

Das folgende Beispiel modelliert einen Turm mit Aussichtsplattform (Abb. 13.7). Der Turm wird von `Building` erzeugt und die Plattform ist mit `RoofBorder` realisiert.

```
building (10,[-2.5,-2.5, 0], [ 2.5,-2.5, 0],  // Hoehe und Grundflaeche
             [ 2.5, 2.5, 0], [-2.5, 2.5, 0]){ // des Turmes
  // (ledge, borderwidth, borderheight, borderdepth)
  roofborder(2,0.5,2,0.5){
    height(0);  // keine Translation
  }
  ...
}
```

13.5.2 `RoofFlat`

Die konkrete Klasse `RoofFlat` modelliert ein Flachdach in Abstimmung mit dem Dachboden, ohne herausragende (benutzerdefinierte) Fronten zu berücksichtigen. Mit *ledge* ist die Modellierung eines Dachvorsprungs möglich. Es ermöglicht das einfache Modellieren von Flachdächern unter Berücksichtigung aller Innenhöfe. Im Szenengraph ist `RoofFlat` ein Sohn von `Building`.

Abb. 13.9: Beispiel eines einfachen Gebäude mit Flachdach.

Sprachdefinition

`roofflat` (*ledge*) {`[height(`*height*`);][on;|off;][`*textures*`;]`}

> Erzeugt ein Flachdach mit Dachvorsprung anhand des Dachbodens. *ledge* definiert die Grösse des Dachvorsprungs und ein Polygon, hier Layer genannt, mit Kanten, die zu den Kanten des Dachbodens parallel verlaufen und in einem konstanten Abstand *ledge* stehen (Abb. 13.10). Für $ledge > 0$ ragt das Dach heraus und für $ledge < 0$ entsprechend hinein[6]. Das Dach wird mit einem Polygon realisiert, das alle Innenhöfe als Löcher berücksichtigt. Das Attribut `height` definiert eine Translation des Daches entlang der z-Achse. Mit `on` und `off` lässt sich das Dach sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur. Der Parameter *ledge* soll mit Umsicht gewählt werden, weil bei nicht-konvexer Grundfläche ein sich selber schneidendes Dach entstehen kann.

[6]Für eine korrekte Bestimmung des Layers wird eine Rechtsorientierung vorausgesetzt.

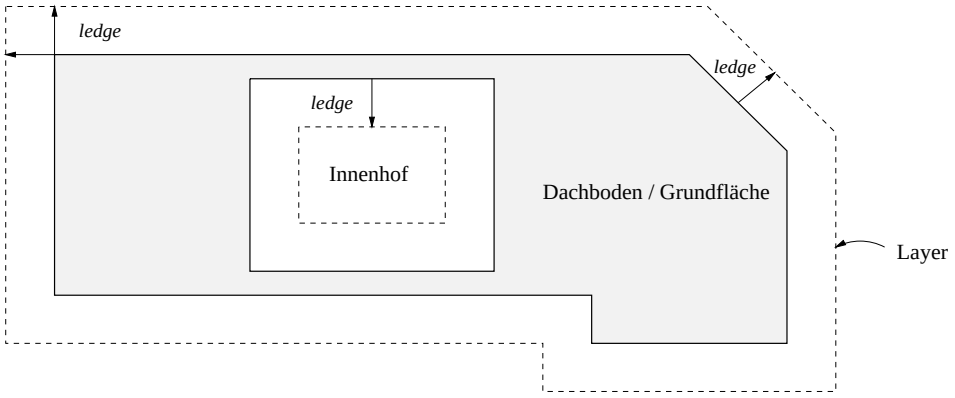

Abb. 13.10: Die Definition des Dachvorsprungs (Grundriss).

Beispiel

Das Beispiel modelliert das Flachdach mit einem Dachvorsprung von 1 aus Abbildung 13.9.

```
building(5,[20,10,0],[-20,10,0],[-20,-10,0],[20,-10,0]){ // Gebaeudedefininition
  hole([ 15,5,0],[ 15,-5,0],[ 5,-5,0],[  5,5,0]);        // rechter Innenhof
  hole([-15,5,0],[ -5,5,0],[-5,-5,0],[-15,-5,0]);        // linker Innenhof
  roofflat(1){   // Dachvorsprung
    roofTexture; // Textur des Daches
  }
  ...
  aTexture;  // Textur des Gebaeudes
}
```

13.5.3 RoofLayer

Die konkrete Klasse `RoofLayer` modelliert ein Dach anhand von Polygonen, die das Skelett des Daches bilden (wie Höhenkurven in einem Geländemodell). Die Oberfläche des Daches wird aus einzelnen "Streifen" zusammengesetzt, die durch benachbarte Polygone definiert werden. Die einzelnen Polygone, hier Layers genannt, werden in Abstimmung mit dem Dachboden und der Geometrie der Fronten parametrisiert. Es ermöglicht eine einfache Modellierung eines Daches, ohne die genaue Geometrie des Gebäudes und der benutzerdefinierten Fronten zu kennen. Im Szenengraph ist `RoofLayer` ein Sohn von `Building`.

Sprachdefinition

`rooflayer` $(\vec{l_0}, \vec{l_1}, \vec{l_2}, \ldots, \vec{l_{n-1}})$ {`[height(`*height*`);][on;|off;][`*textures*`;]`}

Die $\vec{l_i} = (ledge, height, flatness)$ definieren ein Polygon in Abhängigkeit des Dachbodens und der Geometrie der benutzerdefinierten Fronten. *ledge* definiert das Herausragen (vgl. Dachvorsprung Abb. 13.10) des Layers in der xy-Ebene. *height* definiert

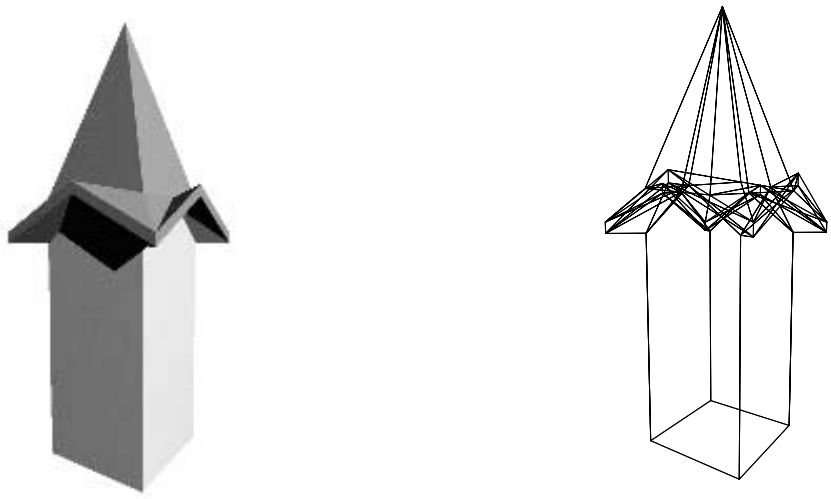

Abb. 13.11: Beispiel von `RoofLayer`.

die Höhe des Layers relativ zum Dachboden. $flatness \in [0, 1]$ gibt den Grad der Berücksichtigung von benutzerdefinierten Fronten wieder, die über den Dachboden herausragen. Diese Fronten bilden neben dem Dachboden ein zweites Polygon (Abb. 13.12), das nicht planar ist und aus dem Polygon des Dachboden durch "Herausziehen" entsteht. 0 entspricht einer vollständigen und 1 keiner Berücksichtigung der herausragenden Fronten. Die Zwischenwerte werden durch Interpolation der zwei Polygone bestimmt. Das Attribut `height` definiert eine Translation des Daches entlang der z-Achse. Mit `on` und `off` lässt sich das Dach sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur. Der Parameter *ledge* soll mit Umsicht gewählt werden, weil bei nicht-konvexer Grundfläche ein sich selber schneidendes Layer entstehen kann.

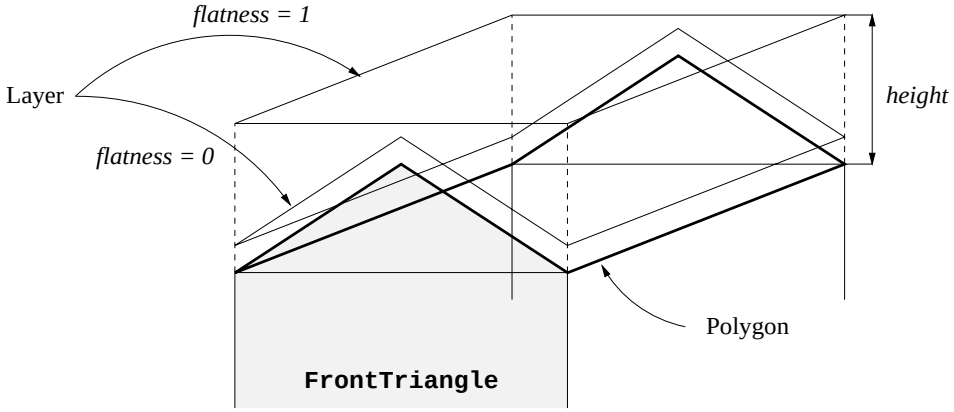

Abb. 13.12: Das Polygon und Layer mit benutzerdefinierten Fronten.

Beispiel

Das Beispiel zeigt einen Turm mit Dach (Abb. 13.11). Der Turm ist mit vier Fronten (`FrontTriangle`) modelliert, die in der Mitte eine Spitze (Giebel) aufweisen. Das Dach ist mit vier Layers modelliert, wobei die ersten zwei Layers die Höhe der Spitzen vollständig berücksichtigen. Die letzten zwei Layers orientieren sich an der Geometrie des Dachbodens, wobei das letzte Layer aus einem einzigen Punkt besteht.

```
building(20,[5,5,0],[-5,5,0],[-5,-5,0],[5,-5,0]){ // Hoehe und Grundflaeche
  // Definition des Daches mit Layers
  // ([ledge, height, flatness],...)
  rooflayer([2.5,0,0],[2.5,1,0],[0,4,1],[-5,20,1]){
    aTexture;
  }
  // Definition der Front: (frontindex,polygonindex,[x,y],...)
  fronttri(0,0,[0.5,1.2]);  // [0.5,1.2] definiert
  fronttri(1,0,[0.5,1.2]);  //  eine Spitze in der
  fronttri(2,0,[0.5,1.2]);  //  Mitte der Front
  fronttri(3,0,[0.5,1.2]);  //  mit absoluter Hoehe 20 * 0.2
  aTexture;
}
```

13.5.4 RoofPlane

Die konkrete Klasse `RoofPlane` modelliert ein Dach mit konstanter Neigung. Das Dach hängt nur vom Dachboden des Gebäudes ab. Die zugehörigen Polygone des Daches werden durch paarweises Schneiden von Halbebenen erzeugt. Dazu wurde ein eigener Algorithmus entwickelt, der weiter unten in groben Zügen erklärt wird und auf Schnitten von Objekten aus Kapitel 12 aufbaut. Die Definition erlaubt eine einfache und wirkungsvolle Modellierung von Dächern. Im Szenengraph ist `RoofPlane` ein Sohn von `Building`.

Abb. 13.13: Beispiel von `RoofPlane`.

Sprachdefinition

`roofplane` $(ledge, \alpha)$ {`[height(`*height*`);][on;|off;][`*textures*`;]`}

Erzeugt ein Dach mit konstanter Neigung $5° \leq \alpha \leq 85°$ und mit einem Dachvorsprung $ledge > 0$. Das Dach ist aus sich paarweise schneidenden Halbebenen definiert, die eine gemeinsame Kante mit dem Polygon des Dachbodens besitzen und mit der Ebene des Dachbodens den Winkel α einschliessen. Die Definition mit Halbebenen ist in seltenen Fällen bei einspringenden Ecken nicht eindeutig, deshalb kann nicht immer ein vollständig deckendes Dach garantierten werden. *height* definiert die Höhe des Layers relativ zum Dachboden. Das Attribut `height` definiert eine Translation des Daches entlang der z-Achse. Mit `on` und `off` lässt sich das Dach sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur. Der Parameter *ledge* soll mit Umsicht gewählt werden, weil bei nicht-konvexer Grundfläche ein sich selber schneidendes Dach entstehen kann.

Algorithmus

Der Algorithmus baut auf Halbebenen auf, die mit einer Kante des Dachbodens und einem konstanten Winkel α definiert werden.

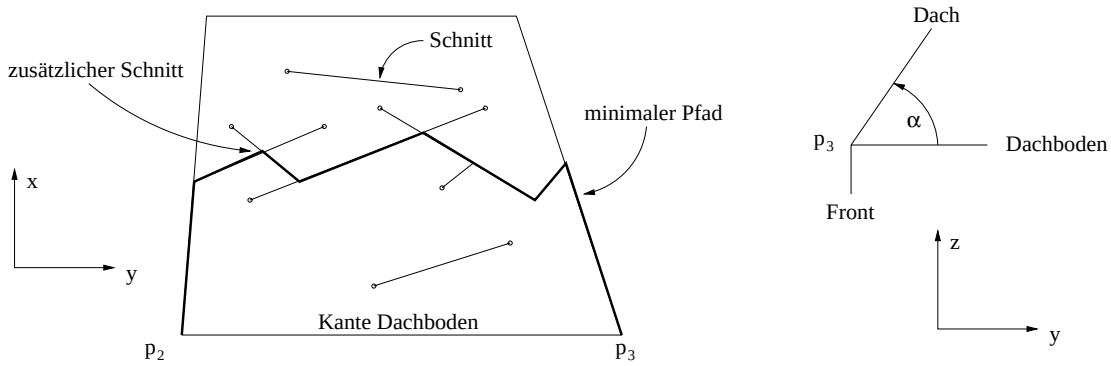

Abb. 13.14: Bestimmung des Polygons (links) mit Winkel α (rechts).

In einem ersten Schritt werden die Halbebenen mit ihren benachbarten Halbebenen und mit der *boundingbox* des Gebäudes unter Berücksichtigung eines Dachvorsprungs geclippt [Fell 92]. Damit werden die Halbebenen auf konvexe Polygone reduziert. In einem zweiten Schritt werden dann die Polygone paarweise miteinander geschnitten [Rour 94, PrSh 88]. Die Schnitte werden anhand des Algorithmus aus Abschnitt 12.4 bestimmt. Jedes Polygon besitzt dann eine zugehörige Menge von Schnitt–Strecken. Die Schnitt–Strecken werden weiter paarweise miteinander geschnitten (vgl. Abschnitt 12.2), um Schnitte zwischen anderen Polygonen zu berücksichtigen. Die nun vorliegende Menge von Schnitt–Strecken bilden einen Pfad, der ein minimales Polygon definiert (Abb. 13.14, links). Wobei der Startpunkt und der Endpunkt des Pfades dem Startpunkte bzw. dem Endpunkt der Kante des Dachbodens entsprichen. Das minimale Polygon ist zugleich auch das gesuchte Polygon. Der Pfad kann mit einem einfachen Backtracking-Algorithmus bestimmt werden.

Es hat sich gezeigt, dass das die Reduktion (Clipping der Halbebenen mit der *boundingbox* und der benachbarten Halbebenen) auf konvexe Polygone die Menge der Schnitte stark reduziert und viele unerwünschte Schnitte verhindert, die von einem einfachen Backtrack-Algorithmus nicht immer erkannt werden können. Die Reduktion wirkt sich bei einspringenden Ecken in seltenen Fällen nachteilig aus, da zuviel geclippt wird und so Löcher im Dach entstehen können.

Beispiel

Das Beispiel zeigt ein Dach (Abb. 13.13) mit einer konstanten Neigung von 40°. In diesem Fall wurde kein Dachvorsprung modelliert, weil die einspringenden Ecken neben dem Turm schon bei kleinen positiven Werten von *ledge* Probleme verursachen.

```
building (20, [-1282.35, -21.343, 0], [-1285.17, -11.289, 0],
               ...
              [-1303.87,  23.019, 0], [-1292.31, -23.813, 0]){
  roofplane (0, 40){
    roofTexture;
  }
  aTexture;
}
```

13.5.5 RoofPoint

Die konkrete Klasse `RoofPoint` modelliert ein Dach mit Punkten, die den Dachgiebel repräsentieren. Die Oberfläche des Daches entspricht einem Flachdach, dem einzelne Punkte aus der Ebene des Dachbodens entlang der z-Achse "herausgezogen" wurden. Dies ermöglicht ein freie Modellierung des Daches, ohne dass die Struktur des Daches, abgesehen vom Dachvorsprung, von der Geometrie des Gebäudes abhängt. Im Szenengraph ist `RoofPoint` ein Sohn von `Building`.

Abb. 13.15: Beispiel von `RoofPoint`.

Sprachdefinition

`roofpoint` $(ledge, \vec{p}_0, \vec{p}_1, \vec{p}_2, \ldots, \vec{p}_{n-1})$ {`[height(`*height*`);][on;|off;][`*textures*`;]`}

> Definiert ein vollständig deckendes Dach mit Dachvorsprung von *ledge*, das die Punkte $\vec{p}_i \in R^3$ enthält und die Geometrie der Fronten vollständig berücksichtigt. Die Oberfläche wird mit einer Triangulation[7] unter Berücksichtigung der zusätzlichen Punkte $\vec{p}_i$ erzeugt, wobei $\vec{p}_i$ bezüglich des Dachbodens definiert wird. Nachträglich wird das erzeugte Dach von unerwünschten Abschnürungen (Abb. 13.16) befreit. Das Attribut `height` definiert eine Translation des Daches entlang der z-Achse. Mit

[7] Die Triangulation ist mit der Software `Triangle` V1.3 (Juli 19, 1996) von J.R. Shewchuk, Carnegie Mellon University, Pittsburgh, Pennsylvania, `jrs@cs.cmu.edu` gelöst.

`on` und `off` lässt sich das Dach sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur. Der Parameter *ledge* soll mit Umsicht gewählt werden, weil bei nicht-konvexer Grundfläche ein sich selber schneidendes Dach entstehen kann.

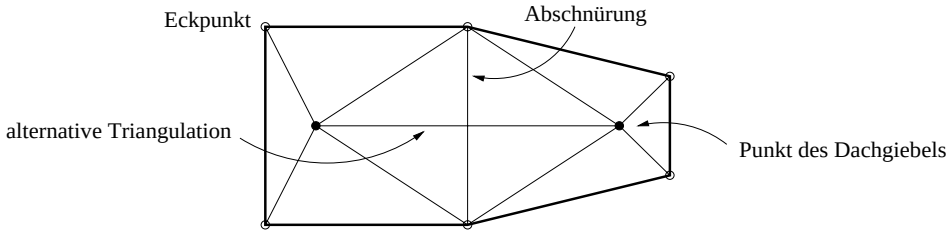

Abb. 13.16: Abschnürung eines Daches.

Beispiel

Das Beispiel modelliert das Gebäude aus Abbildung 13.15. Das Dach ist mit zwei Punkten mit einer Erhebung von 10 definiert. Die Punkte liegen nicht in der Mitte des Grundrisses, was ein Dach mit unterschiedlichen Neigungswinkeln zur Folge hat.

```
building (10,[-10,-20, 0],[ 10,-20, 0],   // Grundflaeche und
             [ 10, 20, 0],[-10, 20, 0]){  //  Hoehe des Gebaeudes
  roofpoint  (2,[-7.5,-15,10],  // Dachvorsprung und Punktes
                [-7.5, 15,10]){, // des Dachgiebels
    aTexture;  // Textur des Daches
  }
}
```

13.5.6 `RoofUser`

Die konkrete Klasse `RoofUser` positioniert ein beliebiges Objekt (`Object3D`) in Abstimmung mit dem Dachboden. So können benutzerdefinierte Dächer oder andere Objekte in Abhängigkeit der z-Koordinate des Dachbodens und dem Attribut `height` positioniert werden. Im Szenengraph ist `RoofUser` ein Sohn von `Building` und unterscheidet sich von `BottomUser` dadurch, dass die Positionierung vom Dachboden abhängt.

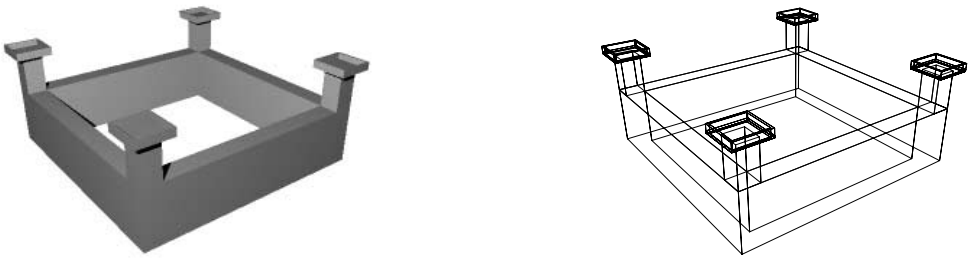

Abb. 13.17: `RoofUser`.

Sprachdefinition

`roofuser {[`*anObject*`;][height(`*height*`);][on;|off;][` *textures*`;]}`

Kennt keine eigenen Parameter. *anObject* ist ein Objekt vom Typ `Objekt3D` und wird entsprechend des Dachbodens und dem Attribut `height` positioniert. Mit `on` und `off` lässt sich das Objekt sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur. Lokale Transformationen werden wie bei allen anderen Gebäudeobjekten überschrieben.

Beispiel

Das folgende Beispiel zeigt eine Burg mit quadratischer Grundfläche und vier Türmen (Abb. 13.17). Die Türme sind mit `Building` modelliert und `RoofUser` übernimmt die Positionierung der Türme auf der Grundmauer.

```
// Definition des Turmes
define tower
  building (10,[-2.5,-2.5,0],[2.5,-2.5,0],[2.5,2.5,0],[-2.5,2.5,0]){
     roofborder(2,0.5,2,0.5);
  }

// Definition der Burg
building (15,[-30,-30, 0], [ 30,-30, 0],   // Hoehe und Grund-
             [ 30, 30, 0], [-30, 30, 0]){  //  flaeche der Burg
  // Innenhof
  hole([-25,-25,0],[ -25,25,0],[25,25,0],[25,-25,0]);
  // Die Vier Tuerme auf den Ecken der Grundmauer
  roofuser {tower {translate[-27.5,-27.5,0];}}
  roofuser {tower {translate[ 27.5,-27.5,0];}}
  roofuser {tower {translate[ 27.5, 27.5,0];}}
  roofuser {tower {translate[-27.5, 27.5,0];}}
  aTexture // Textur der Burg und der Tuerme
}
```

13.6 Die Front (Front)

Die abstrakte Klasse `Front` übernimmt die Abstraktion aller Objekte, welche die Fronten (Seitenflächen des Polyeders) genauer modellieren. Die Front besitzt folgende typische Eigenschaften und Funktionalitäten:

- Die Klasse verwaltet Unterobjekte, nimmt die Funktionen eines Aggregates wahr und übernimmt das Propagieren von Aufgaben wie `doIntersect()` oder `doComputeBounds()`. Neben den erwarteten Teilfronten (`Face`), die einen Teil der Front genauer modellieren, werden auch allgemeine `Object3D`-Objekte akzeptiert.

- Die Front wird mittels der Numerierung der Seitenflächen des aufgespannten Polyeders zugeordnet. Die eindeutige Numerierung der Seitenflächen des Polyeders (Abb. 13.18) besteht aus zwei Indizes. Der erste Index $frontindex \geq 0$, wobei $frontindex \in N$, entspricht dem Index der Kante des Polygons. Der zweite Index $polygonindex \geq 0$, wobei $polygonindex \in N$, definiert das Polygon, wobei $polygonindex = 0$ für das Polygon des Grundrisses steht und $polygonindex > 0$ für die Innenhöfe stehen, in Abhängigkeit der Eingabereihenfolge. Die Abhängigkeit von der Seitenfläche gilt auch für alle Unterobjekte, speziell für die Teilfronten.

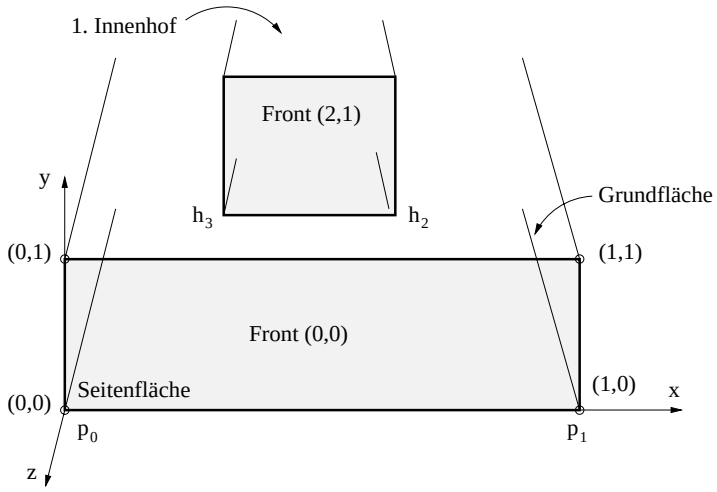

Abb. 13.18: Die eindeutige Numerierung von Fronten.

- Jede Front liegt in der gleichen Ebene wie die ihr zugehörige Seitenfläche (Rechteck) des Polyeders. Es wird anhand der Seitenflächen ein lokales und normiertes Koordinatensystem für die Definition der Front und ihren Unterobjekten eingeführt (Abb. 13.18), so dass die Seitenfläche das Einheitsquadrat repräsentiert.

- Mit dem lokalen Koordinatensystem werden bei der Erzeugung der Dekomposition (`createSubject()`) auch alle Unterobjekte in die Ebene der Seitenfläche transformiert. Der Ursprung des Objektes wird in die linke untere Ecke der Seitenfläche überführt, wobei die x- und y-Achse in die Kanten der Seitenfläche übergehen und die z-Achse der Flächennormalen entspricht[8]. Es wird keine Skalierung auf das normierte Einheitsquadrat durchgeführt.

- Die von `Front` abgeleiteten Klassen ermöglichen eine individuelle Definition der Geometrie der Fronten. Die Geometrie der Front überdeckt die Seitenfläche vollständig, wobei sie nur oben oder unten herausragen darf und in der Ebene der Seitenfläche liegt. So kann die Front bezüglich ihrer Geometrie nur vertikal "erweitert" werden. Die Definition der Geometrie bezieht sich auf das lokale Koordinatensystem.

- Die komplexen Fronten, wie `FrontRect` und `FrontTriangle`, können die "erweiterten" Seitenflächen in Form eines Polygons selber erzeugen. Dabei wird für Teilfronten

[8]Damit ist sichergestellt, dass jedes Unterobjekte unabhängig von der Geometrie des Gebäudes definiert werden kann.

ein entsprechendes Rechteck herausgeschnitten. Bei einer Teilfront, die aus der Front herausragt, wird eine Alternative verlangt, die auch aus mehreren kleineren Löchern bestehen kann. Das Polygon, das die Front repräsentiert, wird wie bei allen anderen Gebäudeobjekten im Cache von `BuildingObject` abgelegt. Das Polygon kann mit den Attributen `walloff` und `wallon` unterdrückt bzw. sichtbar gemacht werden.

- Die Front ist für das Zusammenspiel der Unterobjekte verantwortlich und muss gegebenenfalls ihr nicht untergeordnete Objekte bei der Erzeugung der Dekomposition berücksichtigen (vgl. Klasse `FaceTunnel`). Deshalb kommt der Front auch eine Mittelrolle zwischen Gebäude und Teilfronten zu[9].

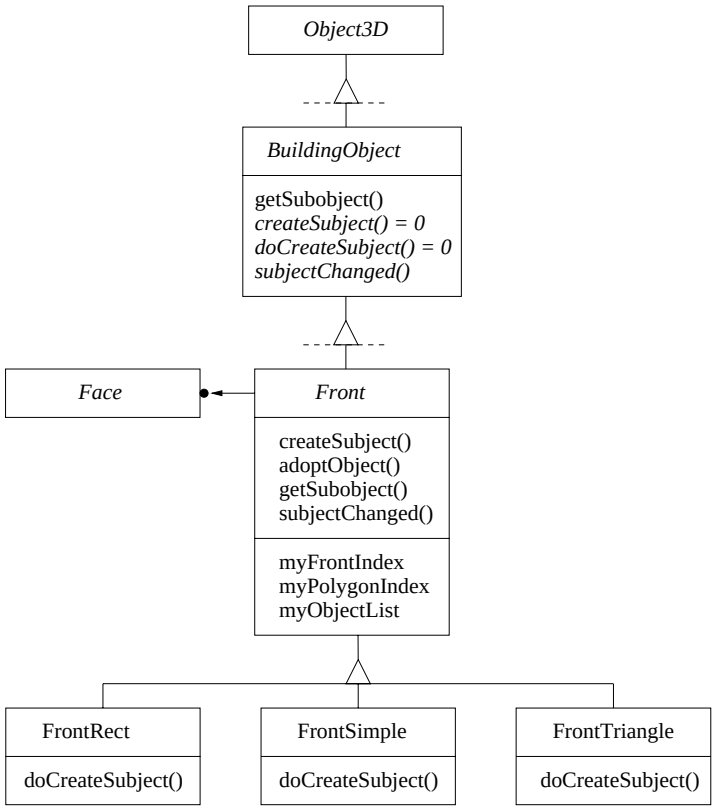

Abb. 13.19: Die Hierarchie von `Front`.

Die genaue Modellierung der Fronten ist in den konkreten Klassen `FrontSimple`, `FrontRect` und `FrontTriangle` in der Methode `doCreateSubject()` implementiert. Die Abbildung 13.19 zeigt die Klassenhierarchie, die wichtigsten Methoden und Daten der einzelnen Klassen.

13.6.1 `FrontSimple`

Die konkrete Klasse `FrontSimple` ist eine einfache Front, die nur Unterobjekte verwaltet und selber keine Repräsentation in Form eines Polygons kennt. So können beliebige Objekte in die Front eingesetzt werden, ohne auf die Geometrie der Front und des Gebäudes

[9]Diese Tatsache weicht die strenge Hierarchie des Baukastensystems auf.

Rücksicht zu nehmen. Damit nimmt diese Klasse nur eine Funktion als reines Aggregat wahr. `FrontSimple` ohne Unterobjekte kommt einer geometrisch leeren Struktur gleich und ist im Szenengraph ein Sohn von `Building`.

Sprachdefinition

`front` $(frontindex, polygonindex)$ {`[`*faces*`;][`*objects*`;]`
`[wallon;|walloff;][on;|off;][`*textures*`;]`}

faces sind die Teilfronten (`Face`), die verwaltet werden. *objects* sind allgemeine `Object3D`-Objekte, die auch als Unterobjekte akzeptiert werden. Mit $frontindex$ und *polygonindex* wird die Abhängigkeit zur entsprechenden Seitenfläche definiert. Die Attribute `wallon` und `walloff` haben keinen Einfluss auf `FrontSimple`, da sowieso kein Polygon erzeugt wird. Mit `on` und `off` lässt sich das Objekt mit seinen Unterobjekten sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur.

Beispiel

Das Beispiel zeigt ein Gebäude mit zwei Fronten. Die Front $(0, 0)$ hängt von der ersten Kante des Polygons der Grundfläche ab. Die andere Front $(1, 3)$ definiert sich mit der zweiten Kante des Polygons des dritten Innenhofes.

```
building (5,  // Hoehe
  [20, 10, 0], [-20, 10, 0], [-20, -10, 0], [20, -10, 0]) { // Grundflaeche
  hole(...);  // 1. Innenhof
  hole(...);  // 2. Innenhof
  hole(...);  // 3. Innenhof
  front(0,0){   // 1. Front der Grundflaeche
    aFace;        // Teilfront
    ...
  }
  front(1,3){   // 2. Front des 3. Innenhofes
    ...
  }
}
```

13.6.2 `FrontRect`

Die konkrete Klasse `FrontRect` modelliert eine rechteckige Front, wobei sie unten und/oder oben herausragen kann und die entsprechende Seitenfläche immer vollständig überdecken muss. Die Front erzeugt neben der Verwaltung der Unterobjekte eine eigene Repräsentation in Form eines Polygons, das der Definition der Front, einem Rechteck entspricht. Dabei werden für die Teilfronten, falls nötig, die entsprechenden Löcher aus dem Polygon herausgeschnitten, ohne Überlappungen der Teilfronten zu prüfen[10]. Falls eine Teilfront

[10]Das Überlappen kann zwar einfach erkannt werden, kann aber nur mit aufwendigen Protokollen gelöst werden.

aus der Front herausragt, wird eine Alternative von der Teilfront verlangt. Mit der Alternative können Löcher für die Objekte der Teilfront direkt in der Front herausgeschnitten werden. Damit ist eine optimale Modellierung mit Teilfronten auch im Randbereich der Fronten gewährleistet.

Weiter berücksichtigt die Front auch indirekt erzeugte Teilfronten, die nicht als Unterobjekt der betroffenen Front auftreten. Dies ermöglicht die Modellierung von Objekten, die sich nicht nur auf eine Front beziehen, sondern von mehreren Front abhängen können, wie beispielsweise der Tunnel (`FaceTunnel`). Damit wird aber die strenge Hierarchie auf Kosten der Übersicht etwas gelockert. `FrontRect` ist im Szenengraph ein Sohn von `Building`.

Abb. 13.20: Beispiel von unterschiedlich herausragenden Fronten.

Sprachdefinition

`frontrect` (*frontindex*, *polygonindex*, *bottom*, *top*)
{[*faces*;][*objects*;][`wallon;`|`walloff;`][`on;`|`off;`][*texture*;]}

faces sind die Teilfronten (`Face`), die verwaltet werden. *objects* sind allgemeine `Object3D`-Objekte, die auch als Unterobjekte akzeptiert werden. Mit *frontindex* und *polygonindex* wird die Abhängigkeit zur entsprechenden Seitenfläche definiert. $bottom \leq 0$ und $top \geq 1$ definieren unten bzw. oben das Herausragen der Front relativ zur Höhe des Gebäudes. Es wird ein rechteckiges Polygon erzeugt, wobei für alle Teilfronten die entsprechenden Löcher herausgeschnitten und, falls nötig, Alternativen angefordert werden. Die Attribute `wallon` und `walloff` machen das Polygon sichtbar bzw. unsichtbar. Mit `on` und `off` lässt sich das Objekt mit seinen Unterobjekten sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur.

Beispiel

Das Beispiel modelliert das Gebäude von Abbildung 13.20 und zeigt alle Kombinationen des Herausragens von `FrontRect`.

```
building (5, [20, 10, 0], [-20, 10, 0],      // Definition
             [-20, -10, 0], [20, -10, 0]) {  // des Gebaeudes
  aTexture;                   // beliebige Textur
  frontrect (0,0, 0.0,1.0);  // Standard Front [0,0],[1,1]
  frontrect (1,0,-0.5,1.0);  // ragt unten um 0.5 heraus
  frontrect (2,0,-0.5,1.5);  // ragt oben und unten um 0.5 heraus
  frontrect (3,0, 0.0,1.5);  // ragt oben  um 0.5 heraus
};
```

13.6.3 `FrontTriangle`

Die konkrete Klasse `FrontTriangle` definiert eine frei modellierbare Front mit Hilfe eines Polygonzugs, wobei nur der untere und obere Teil der Front mit Hilfe eines Polygonzugs modellierbar sind. So können beispielsweise Mauern für die Modellierung einer Burg definiert werden.

Die Front erzeugt neben der Verwaltung der Unterobjekte eine eigene Repräsentation in Form eines Polygons, das den oberen und den unteren Polygonzug enthält. Dabei werden wie im Falle von `FrontTriangle` die gewünschten Löcher für Teilfronten herausgeschnitten, ohne Überlappungen der Teilfronten zu prüfen. Dies ermöglicht eine optimale Modellierung in den Randbereichen, wie unter `FrontRect`. Weiter werden auch die indirekten Teilfronten entsprechend berücksichtigt und ermöglichen Objekte, die sich auf mehrere Fronten beziehen, umzusetzen. `FrontTriangle` ist im Szenengraph ein Sohn von `Building`.

Abb. 13.21: Beispiel von unterschiedlich modellierten Fronten.

Sprachdefinition

`frontri` $(frontindex, polygonindex, \vec{p}_0, \vec{p}_1, \vec{p}_2, \ldots, \vec{p}_{n-1})$
{[*faces*;][*objects*;][`wallon;`|`walloff;`][`on;`|`off;`][*texture*;]}

faces sind die Teilfronten (`Face`), die verwaltet werden. *objects* sind allgemeine `Object3D`-Objekte, die auch als Unterobjekte akzeptiert werden. Mit $frontindex$ und *polygonindex* wird die Abhängigkeit zur entsprechenden Seitenfläche definiert. Die Punkte $\vec{p}_i \in R^2$ definieren die zusätzlichen Punkte[11] des Polygons der Front relativ zur Seitenfläche (dem normierten lokalen Koordinatensystem). Für $\vec{p}_{iy} \leq 0$ oder $\vec{p}_{iy} \geq 1$ wird ein Punkt unten bzw. oben definiert. Die Punkte werden nach der x-Komponente sortiert[12] und definieren so den oberen und den unteren Polygonzug. Es wird ein Polygon erzeugt, das die zwei Polygonzüge enthält und alle zulässigen Löcher für Teilfronten berücksichtigt. Die Attribute `wallon` und `walloff` machen das Polygon sichtbar bzw. unsichtbar. Mit `on` und `off` lässt sich das Objekt mit seinen Unterobjekten sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur.

[11] Das Polygon besteht schon aus den Punkten $(0,0), (1,0), (1,1)$ und $(0,1)$.
[12] Der Sortieralgorithmus ist stabil.

Beispiel

Das Beispiel modelliert die Abbildung 13.21 und enthält vier Fronten, wobei die Definition der zusätzlichen Punkte in korrekter Reihenfolge definiert wurden.

```
building (5, [20, 10, 0], [-20, 10, 0],      // Definition
             [-20, -10, 0], [20, -10, 0]) {  // des Gebaeudes
  fronttri (0,0,[.4,-.5],[.6,-.5]);               // ragt unten heraus
  fronttri (1,0,[0,1.5],[.25,1],[.75,1],[1,1.5]); // an den Enden erhoeht
  // definiert eine Burgmauer
  fronttri (2,0,[0.0,1.5],[0.2,1.5],[0.2,1.0],
                [0.4,1.0],[0.4,1.5],[0.6,1.5],[0.6,1.0],
                [0.8,1.0],[0.8,1.5],[1.0,1.5]);
  // Standard Front [0,0],[1,1] ohne zusaetzliche Punkte
  fronttri (3,0);
};
```

13.7 Die Teilfront (Face)

Die abstrakte Klasse `Face` übernimmt die Abstraktion aller Objekte, die einen Teil einer Front, deren Unterobjekte sie sind, genauer modellieren. Die Teilfront selber "lebt" auf ihrer zugehörigen Front und besitzt folgende Eigenschaften und Funktionalitäten:

- Die Teilfront nimmt auf der entsprechenden Front eine rechteckige und zur Seitenfläche parallele Fläche ein. Die Definition der Teilfront geschieht bezüglich dem lokalen Koordinatensystem der Front (Abb. 13.22). Das Einheitsquadrat entspricht der zugehörigen Seitenfläche. Der Ursprung liegt in der linke unteren Ecke und die z-Achse entspricht der Flächennormalen. So kann garantiert werden, dass Teilfronten sich unabhängig von der Grösse der Fronten definieren lassen.

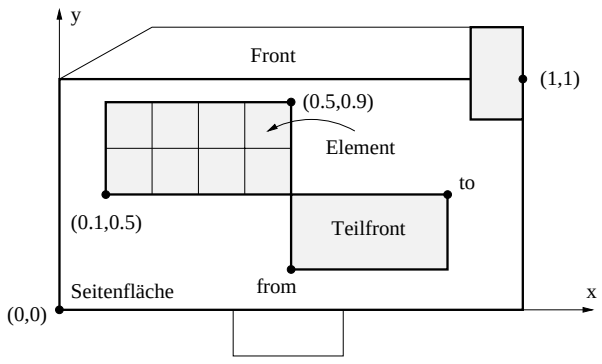

Abb. 13.22: Parametrisierung der Teilfront in der Front.

- Sie kennt eine Rechteckige und gleichförmige Aufteilung der Teilfront in Spalten (*column*) und Zeilen (*row*), den Elementen.
- Sie positioniert die Teilfront bezüglich der entgegengesetzten Richtung der z-Achse mit dem Attribut `depth`.

- Sie verwaltet einen Zeiger auf die zugehörige Front, und sie stellt sicher, dass auf die Daten der Front zugegriffen werden kann. Weiter ermöglicht `Face` einen Zugriff auf das Gebäude über die zugehörige Front[13].

Die genaue Modellierung der Teilfronten ist in den konkreten Klassen `FaceArbour`, `FaceDummy`, `FaceItem`, `FaceTunnel` und `FaceWall` in der Methode `doCreateSubject()` implementiert. Die folgende Abbildung 13.23 zeigt die Klassenhierarchie, die wichtigsten Methoden und Daten der einzelnen Klassen.

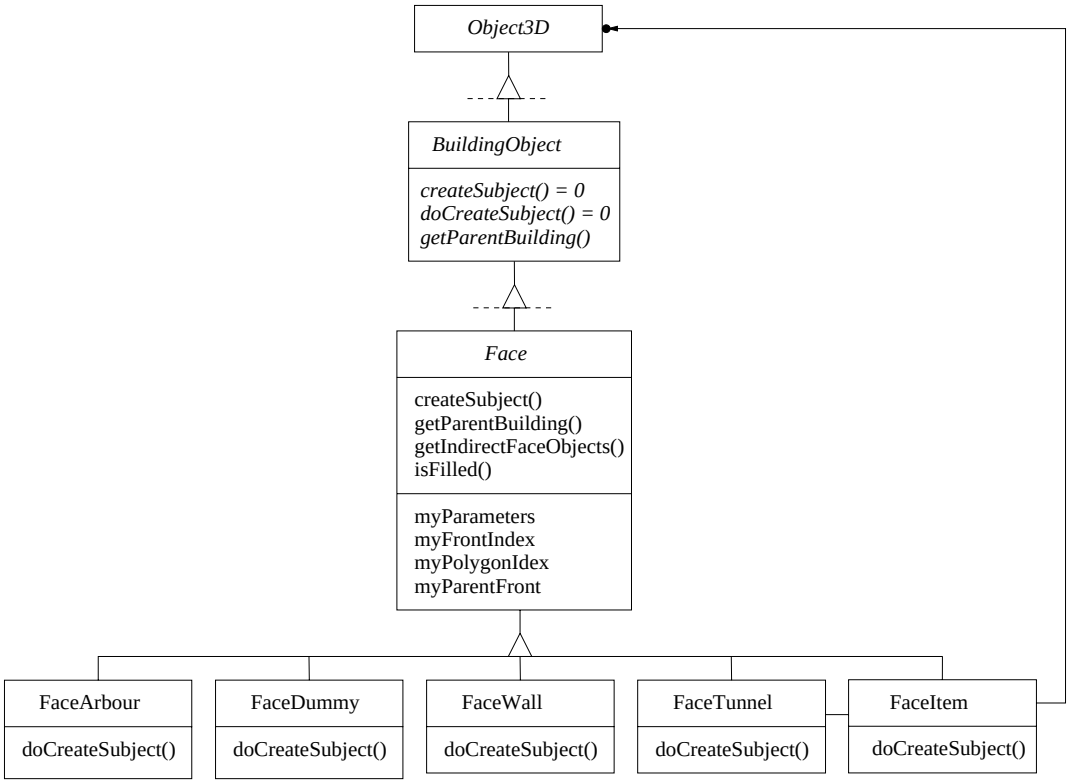

Abb. 13.23: Die Hierarchie von `Face`.

13.7.1 `FaceArbour`

Die konkrete Klasse `FaceArbour` modelliert eine Laube mit den dazugehörigen Bögen und dem Gang. Die Anzahl der Bögen ist frei wählbar. Die Bögen werden mit einer Bézier-Kurve definiert. Zusätzlich wird eine Innenwand erzeugt, um den Gang zu modellieren. `FaceArbour` ist im Szenengraph ein Sohn von `Front`.

[13]Der Zeiger auf das Gebäude ermöglicht es logische Abängigkeiten zwischen Gebäudeobjetken zu realisieren, die quer zur hierarchischen Struktur verlaufen.

Abb. 13.24: Eine Laube mit drei Bögen und Gang.

Sprachdefinition

`facearbour` $(\vec{from}, \vec{to}, column)\{$
$\{$`[arbourdepth(`$arbourdepth = 5$`);]`
`[bezierparm1(`$b1 = 1$`);]`
`[bezierparm2(`$b2 = 0.5$`);]`
`[columndepth(`$columndepth = 0.1$`);]`
`[columnwidth(`$columnwidth = 0.2$`);]`
`[numberofpoints(`$n = 7$`);]`
`[wallon;|walloff;][depth(`$depth$`)][on;|off;][`$texture$`;]`$\}$

$\vec{from} \in R^2$ und $\vec{to} \in R^2$ definieren das Rechteck bezüglich dem lokalen Koordinatensystem der zugehörigen Front. $column > 0$, wobei $column \in N$, definiert die Anzahl der Bögen. $arbourdepth > 0$ definiert die Tiefe der Laube, d.h. die Breite des Ganges. $b1 \geq 0$ und $b2 \geq 0$ parametrisieren das Kontrollpolygon einer kubischen Bézier-Kurve in R^2, wobei $b1$ und $b2$ relativ zur Höhe von `FacArbour` interpretiert werden (Abb. 13.25). $columndepth > 0$ definiert die Tiefe der Bögen relativ zu $arbourdepth$ und $0 < columnwidth < 0.5$ entspricht der Breite des Portals relativ zur Breite des Elementes. $numberofpoints > 0$, wobei $numberofpoints \in N$, definiert die Anzahl der Interpolationspunkte der Bézier-Kurve. Die Innenwand der Laube kann mit den Attributen `walloff` und `wallon` unterdrückt bzw. sichtbar gemacht werden. `depth` verschiebt die Teilfront nach innen ($depth > 0$) oder aussen ($depth < 0$). Mit `on` und `off` lässt sich das Objekt sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur.

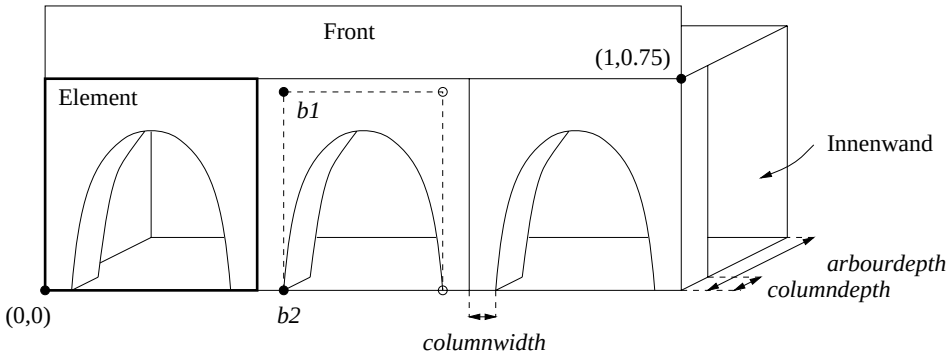

Abb. 13.25: Die Parametrisierung von `FaceArbour`.

Beispiel

Das Beispiel modelliert eine Laube mit drei Bögen und einer Innenwand (Abb. 13.24).

```
building (5, [20, 10, 0], [-20, 10, 0],      // Definition
             [-20, -10, 0], [20, -10, 0]) {  // des Gebaeudes
  aTexture;          // beliebige Textur
  frontrect (0,0, 0.0,1.0){
    // Laube mit drei Boegen
    facearbour([0.2,0],[0.8,0.75],3);
  }
}
```

13.7.2 FaceDummy

Die konkrete Klasse `FaceDummy` übernimmt die Repräsentation von Teilfronten, die von Teilfronten selber erzeugt werden, aber einer anderen Front zugehören. Diese indirekten Teilfronten werden für die Modellierung von Teilfronten eingesetzt, die logisch von mehreren Fronten abhängen. Die Fronten übernehmen dann mit Hilfe von `Building` das Aufsuchen der zugehörigen Teilfronten. `FaceTunnel` beispielsweise erzeugt eine indirekte Teilfront für die korrekte Represätation des Ausganges in der gegenüberliegenden Front. `FaceDummy` ist nur für interne Zwecke bestimmt und ist im Szenengraph nicht sichtbar.

13.7.3 FaceItem

Die konkrete Klasse `FaceItem` positioniert und vervielfacht innerhalb der Teilfront ein beliebiges Objekt entsprechend der Anzahl Spalten und Zeilen. Neben dem Attribut `depth`, kennt `FaceItem` Attribute, die die Positionierung sowohl im Randbereich der Teilfront, wie auch bezüglich der einzelnen Elemente erleichtern. Weiter kann in jedem Element ein Loch für das Objekt herausgeschnitten werden. Mit dem Herausschneiden von Löchern, ist eine Einsicht in das Gebäude möglich. So kann beispielsweise eine vollständige Fensterfront bestehend aus mehreren Stockwerken modelliert werden. Ausserdem werden, falls die Teilfront über die Front herausragen sollte, die Löcher der positionierten Objekte als Alternative angeboten. Damit kann die Front die noch zulässigen Löcher aus der Front selber herausschneiden und eine optimale Modellierung im Randbereich garantieren. `FaceItem` ist im Szenengraph ein Sohn von `Front`.

Abb. 13.26: Fensterfront.

Sprachdefinition

`faceitem` $(\vec{from}, \vec{to}, column, row)$
`{[`*anObject*`;][hole;|hole` $(\vec{h}_0, \vec{h}_1, \vec{h}_2, \cdots, \vec{h}_{n-1})$`;]`
`[displacement[`$\vec{displacement}$`;]`
`[leftcolumn;|rightcolumn;|bothcolumn;|insidecolumn;]`
`[bottomrow;|toprow;|bothrow;|insiderow;]`
`[wallon;|walloff;][depth(`*depth*`)][on;|off;][`*textures*`];}`

$\vec{from} \in R^2$ und $\vec{to} \in R^2$ definieren das Rechteck bezüglich dem lokalen Koordinatensystem der zugehörigen Front. $column, row > 0$, wobei $column, row \in N$, definieren die Aufteilung der Element in Spalten und Zeilen. *anObject* ist das zu positionierende Objekt vom Typ `Objekt3D`. Das Attribut `hole` erzeugt ein Loch für jedes einzelne Element anhand der *boundingbox* des zu positionierenden Objektes. `hole` erlaubt auch eine explizite Form der Definition. Das Loch wird im lokalen Koordinatensystem des Objektes in der xy-Ebene mit einem einfachen Polygon `hole` $(\vec{h}_0, \vec{h}_1, \vec{h}_2, \cdots, \vec{h}_{n-1})$ mit $\vec{h}_i \in R^3$ definiert. Mit $\vec{displacement} \in R^2$ ist eine relative Translation des Objektes bezüglich des einzelnen Elementes möglich, wobei Löcher mitverschoben werden. Die Attribute `leftcolumn`, `rightcolumn`, `bothcolumn` und `insidecolumn` definieren die Positionierungsart bezüglich der Spalten. `leftcolumn` (Default–Wert) und `rightcolumn` positionieren das Objekt bezüglich der linken bzw. rechten Seite des zugehörigen Elementes. `bothcolumn` erzeugt am rechten Rand der Teilfront eine zusätzlich Spalte mit Objekten. `insidecolumn` lässt die linke Spalte von Objekten weg. Die Attribute `bottomrow`, `toprow`, `bothrow` und `insiderow` definieren, analog zur Positionierungsart bezüglich Spalten, die Positionierungsart bezüglich Zeilen. Der Default–Wert ist `bottomrow`. Die Attribute `walloff` und `wallon` unterdrücken bzw. erzeugen ein Rechteck entsprechend der Grösse der Teilfront und berücksichtigen eventuelle Löcher der positionierten Objekte. `depth` verschiebt die Teilfront nach innen ($depth > 0$) oder aussen ($depth < 0$). Mit `on` und `off` lässt sich das Objekt sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur.

Beispiel

Das Beispiel zeigt eine Fensterfront (Abb. 13.26), die aus 8×2 Fenstern besteht. Für die Fenster werden die Löcher anhand der *boundingbox* herausgeschnitten. Ein Kugel wurde mit Hilfe der Attributen `bothcolumn` und `bothrow` in allen Ecke der einzelnen Elemente

positioniert.

```
building (5, [20, 10, 0], [-20, 10, 0],      // Definition
             [-20, -10, 0], [20, -10, 0]) {  // des Gebaeudes
  aTexture;         // beliebige Textur
  frontrect (0,0, 0.0,1.0){
    // [from],[to],row,column
    faceitem([0.2,0.05],[0.8,0.8],8,2){
      window;                 // ein Fenster
      displacement [0.5,0.5]; // Verschiebt den Ursprung des Objektes
                              // in die Mitte des Elements
      hole;     // Loch bezueglich der Boundingbox
    }
    // plaziert die Kugel 3*9 Mal in allen Ecken der Elemente
    faceitem([0.2,0.05],[0.8,0.8],8,2){
      sphere(0.4,[0,0,0]); // eine Kugel
      bothcolumn; // plazieren rechts und links des Elementes
      bothrow;    // plazieren oben und unten des Elementes
    }
  }
}
```

13.7.4 FaceTunnel

Die konkrete Klasse `FaceTunnel` modelliert einen Durchgang, der einem Tunnel entspricht und zwei Fronten miteinander verbindet. Ein Durchgang wird in einer Front definiert und mit einem Verweis auf die gegenüberliegende Front versehen. Das Portal des Durchganges wird mit einer Bézier-Kurve definiert, das sich wie die Bögen von `FaceArbour` parametrisieren lässt. Zusätzlich kann der Boden des Durchganges unterdrückt werden. `FaceTunnel` kümmert sich auch um die gegenüberliegende Teilfront und erzeugt mittels `FaceDummy` eine entsprechende Teilfront für die gegenüberliegende Front. `FaceTunnel` kennte eine Alternative, falls die Teilfront und/oder die indirekte Teilfront herausragen sollte. Im Szenengraph ist `FaceTunnel` ein Sohn von `Front`.

Abb. 13.27: Zwei Durchgänge.

Sprachdefinition

`facetunnel` $(\vec{from}, \vec{to}, \vec{otherfrom}, \vec{otherto}, frontindex, polygonindex)$
`{ [bezierparm1(`$b1 = 1$`);]`

`[bezierparm2(`$b2 = 0.5$`);]`
`[columnwidth(`$columnwidth = 0.2$`);]`
`[numberofpoints(`$n = 7$`);]`
`[wallon;|walloff;][depth(`$depth$`)][on;|off;][`$textures$`];}`

$\vec{from} \in R^2$ und $\vec{to} \in R^2$ definieren das Rechteck bezüglich dem lokalen Koordinatensystem der zugehörigen Front. $other\vec{f}rom \in R^2$ und $other\vec{t}o \in R^2$ definieren das Rechteck der Teilfront der gegenüberliegenden Front. $frontindex, polygonindex \geq 0$, wobei $frontindex, polygonindex \in N$, definieren die gegenüberliegende Front. $b1 \geq 0$ und $b2 \geq 0$ parametrisieren das Kontrollpolygon einer kubischen Bézier-Kurve in R^2, wobei $b1$ und $b2$ relativ zur Höhe von `FaceTunnel` interpretiert werden (Abb. 13.25). $0 < columnwidth < 0.5$ entspricht der Breite des Portals relativ zur Breite des Elementes. $numberofpoints > 2$, wobei $numberofpoints \in N$, definiert die Anzahl der Interpolationspunkte der Bézier-Kurve. Der Boden des Durchganges kann mit den Attributen `walloff` und `wallon` unterdrückt bzw. sichtbar gemacht werden. `depth` verschiebt die Teilfront nach innen ($depth > 0$) oder aussen ($depth < 0$). Mit `on` und `off` lässt sich das Objekt sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur.

Beispiel

Das Beispiel zeigt ein Gebäude mit Innenhof und zwei Durchgängen, die mit `FaceTunnel` modelliert sind (Abb. 13.27). Die Durchgänge sind ohne Boden und sind mit benutzerdefinierten Werten für die Parametrisierung des Kontrollpolygons der Bézier-Kurve modelliert.

```
building (5, [20, 10, 0], [-20, 10, 0],      // Definition
             [-20, -10, 0], [20, -10, 0]) {  // des Gebaeudes
  // Innenhof
  hole ( [10, -5, 0], [-10, -5, 0], [-10, 5, 0],[10, 5, 0]);
  frontrect (0,0){
    // Durchgang von Fornt (0,0) zu (2,1)
    facetunnel([0.25,0],[0.75,1],[0,0],[1,1],2,1){
      walloff;           // ohne Boden
      numberofpoints(9); // Interpolationspunkte
      bezierparm1(0.9);  // Bezierkontrollpunkte
      bezierparm2(0.5);
    }
  }
```

```
  // Durchgang von Fornt (0,1) zu (2,0)
  frontrect (0,1){
    facetunnel([0,0],[1,1],[0.25,0],[0.75,1],2,0){
      walloff;           // ohne Boden
      numberofpoints(9); // Interpolationspunkte
      bezierparm1(0.9);  // Bezierkontrollpunkte
      bezierparm2(0.5);
    }
  }
  ...
}
```

13.7.5 FaceWall

Die konkrete Klasse `FaceWall` modelliert ein Rechteck und eignet sich für das Ausfüllen von rechteckigen Teilen von Fronten, die beispielsweise verschieden gefärbt sein sollen. Im Szenengraph ist `FaceWall` ein Sohn von `Front`.

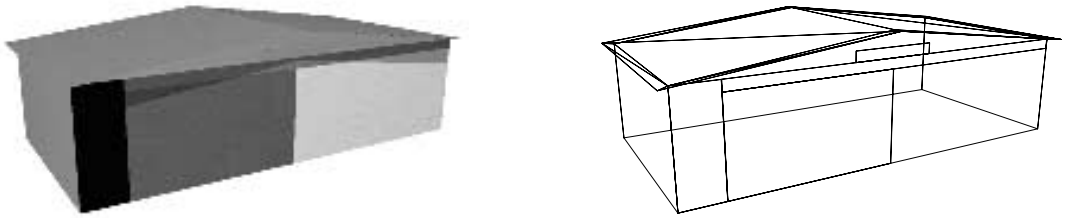

Abb. 13.28: Verschiedenfarbige Front mit `FaceWall` modelliert.

Sprachdefinition

`facewall` $(\vec{from}, \vec{to})$ { `[depth(`*depth*`)][on;|off;][`*textures*`;}`

Erzeugt anhand von $\vec{from} \in R^2$ und $\vec{to} \in R^2$ ein Rechteck bezüglich dem lokalen Koordinatensystem der zugehörigen Front. Mit dem Attribut `depth` kann die Teilfront nach innen ($depth > 0$) oder aussen ($depth < 0$) verschoben werden. Mit `on` und `off` lässt sich das Objekt sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur.

Beispiel

Das Beispiel zeigt eine Front mit Giebel, die vier verschiedenfarbige Teilfronten besitzt (Abb. 13.28).

```
building (5, [20, 10, 0], [-20, 10, 0],      // Definition
             [-20, -10, 0], [20, -10, 0]) {  // des Gebaeudes
  aTexture;          // beliebige Textur
  fronttri (0,0,[0.5,1.25]){
    // 4 Teilfronten mit verschiedenen Farben
    facewall([0.0,0.0],[0.1,1.0]){black;}
    facewall([0.1,0.9],[1.0,1.0]){red;}
    facewall([0.1,0.0],[0.5,0.9]){blue;}
    facewall([0.4,1.0],[0.6,1.1]){gold;}
  };
  ...
}
```

13.8 Das Positionieren (Snatch)

Die abstrakte Klasse `Snatch` ermöglich das Positionieren von Objekten bezüglich anderen Objekten. `Snatch` übernimmt das Einsammeln der Objekte, die für das Positionieren des Objektes berücksichtigt werden sollen, wobei das genaue Vorgehen des Einsammelns in den konkreten Klassen festgehalten wird. Weiter wird durch Schneiden eines Strahls mit den zu berücksichtigenden Objekten die konkrete Position im Raum bestimmt. Die Abbildung 13.29 zeigt die Klassenhierarchie, die wichtigsten Methoden und Daten der einzelnen Klassen.

13.8.1 SnatchRoof

Die konkrete Klasse `SnatchRoof` vereinfacht die Positionierung von allgemeinen Objekten bezüglich Dächern. `SnatchRoof` setzt ein Objekt anhand gegebener xy-Koordinaten genau auf das betroffene Dach auf. Weiter wird das Objekt entsprechend der Neigung des Daches bezüglich der z-Achse gedreht. Damit entfällt das nachträgliche Ausrichten des Objektes bezüglich des Daches.

Sprachdefinition

`snatch` $(\vec{position})$ {`[`*anObject*`;][on;|off;][`*texture*`];}`

> Verschiebt *anObject* vom Typ `Object3D` anhand der xy-Kooridnaten[14] $\vec{position} = (x', y') \in R^2$ nach (x', y', z), wobei der z-Wert durch Schneiden der Gerade (x', y') mit dem Dach bestimmt wird. Bei mehreren Schnittpunkten wird der Maximalwert genommen. Die Ausrichtung des Objektes wird durch die Neigung des Daches im Punkt (x', y', z) bestimmt. Das Objekt wird bezüglich der z-Achse so gedreht, dass

[14] Die Koordinaten beziehen sich auf die des Gebäudes.

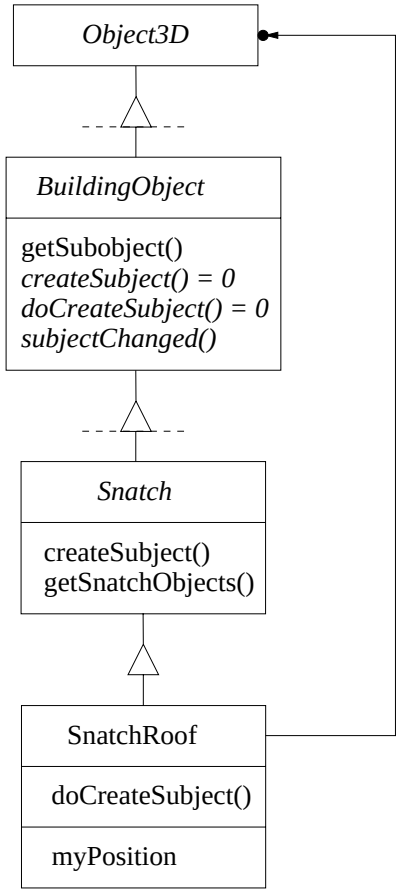

Abb. 13.29: Die Hierarchie von `Snatch`.

die x-Achse bezüglich der xy-Ebene in die gleiche Richtung wie die Fallinie zeigt. Mit `on` und `off` lässt sich das Objekt sichtbar bzw. unsichtbar machen. *textures* ist eine allgemeine Textur.

Beispiel

Das Beispiel zeigt ein Gebäude mit vier Dachluken, die mit `Snatch` positioniert werden (Abb. 13.30). Die Dachluken werden entsprechend der Fallinie des Daches ausgerichtet.

```
// Definition der Dachluke
define dormer
building (4, [-10, -5, 0], [0, -5, 0], [0, 5, 0], [-10, 5, 0]) {
  aTexture;
  fronttri (1, 0, [0.5, 1.5]);
  fronttri (3, 0, [0.5, 1.5]);
  roofpoint (1) {roofTexture;}
}
```

```
building (15, [40, 20, 0], [-40, 20, 0],      // Definition des
              [-40, -20, 0], [40, -20, 0]) {  // Gebaeudes
  aTexture;
  roofplane (2, 30) {roofTexture;}
  snatch ([35, 0])  {dormer;}
  snatch ([-35, 0]) {dormer;}
  snatch ([0, 15])  {dormer;}
  snatch ([0, -15]) {dormer;}
}
```

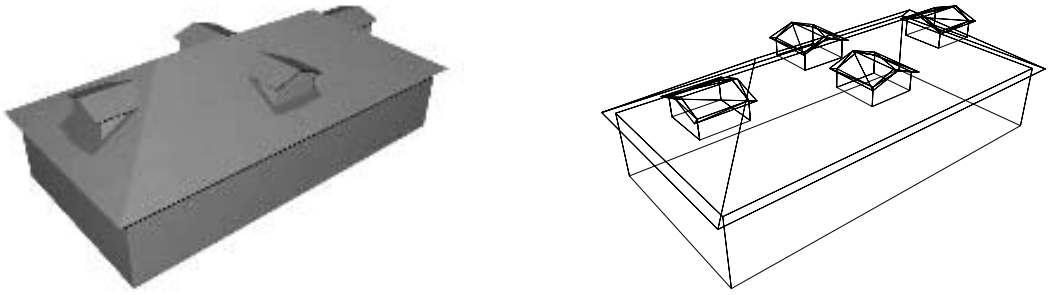

Abb. 13.30: Positionieren und Ausrichten eines Objektes mit `Snatch`.

Kapitel 14

Tips und Tricks zur Gebäudemodellierung

Dieses Kapitel gibt einige Hinweise, die den praktischen Umgang mit der Gebäudemodellierung erleichtern. Die Modellierung von Gebäuden erfordert genau so wie die allgemeine Modellierung Übung und Erfahrung. Hier sind die wichtigsten Erfahrungen, die im Verlauf dieser Arbeit gesammelt wurden, kurz zusammengefasst:

- Für die Modellierung einer ansprechenden Szene sind immer noch eine gute Idee und Phantasie nötig.
- Der Einstieg in die Gebäudemodellierung gestaltet sich am einfachsten durch das Studium und das Manipulieren von schon bestehenden Gebäuden.
- Die Geometrie des Gebäudes spielt eine entscheidende Rolle für das Erzielen von ansprechenden Ergebnissen. Es hat sich erwiesen, dass schon mit einfach modellierten Gebäuden, die aus einem Dach bestehen, bescheiden modellierten Fronten und ohne Texturen auskommen, gute Resultate möglich sind.
- Texturen eignen sich sehr gut für die Repräsentation von Oberflächenstrukturen. Sonst sollten die Texturen eher zurückhaltend eingesetzt werden, da nicht immer bessere Resultate erzielt werden.
- Komplexe Gebäude lassen sich oftmals leichter durch das Zusammensetzen von mehreren dafür einfacheren Gebäuden realisieren. Dabei kann das Zusammensetzen mit einem Aggregat wie `List3D` geschehen, oder indem die Gebäude mit `BottomUser`, `RoofUser` und `SnatchRoof` beliebig verschachtelt werden (vgl. `building_zeit.bsdl3`).
- Bei umfangreicheren Gebäuden können in der Modellierungsphase ganze Fronten oder Gebäude mit dem Attribut `off` unterdrückt werden, um die Visualisierung zu beschleunigen.
- Das Default–Gebäude wird normalerweise nur erzeugt, falls das Gebäude keine Teilfronten (`Face`) besitzt. Ein unerwünschtes Default–Gebäude kann wie folgt unterdrückt werden, wobei die Definitionen von Fronten von anderen Objekten wie Dächern immer noch berücksichtigt werden:

```
front(0,0){
  off;
  facewall([0,0],[1,1]);
}
```

- Bei mehrfachen Definitionen einer Front, was in der Regel nicht unbedingt empfehlenswert ist, wird als erstes `FrontTriangle`, dann `FrontRect` und zuletzt `FrontSimple` berücksichtigt, wobei bei gleichem Fronttyp die Eingabereihenfolge entscheidet.
- Bei den Fronten `FrontTriangle` und `FrontRect` ist zu beachten, dass sich die Teilfronten nicht überlappen, wobei das Herausragen aus der Front zugelassen ist. Für `FaceItem` sind beliebige Überlappungen möglich, falls mit `walloff` die Erzeugung des zugehörigen Polygons unterdrückt wird.
- Für die genauere Modellierung der Fronten mit `FaceItem` ist es hilfreich, die zu positionierenden Objekte im Ursprung zu definieren, und zwar so, dass alle Objekte etwa gleich gross sind. Die Objekte können nachträglich immer noch transformiert oder mit dem Attribut `displacement` relativ zum Element positioniert werden. Damit können die Objekte für andere Gebäude wiederverwendet werden, und der Benutzer kann sich eine kleine Sammlung von Objekten anlegen.

Kapitel 15

Schlussbemerkungen

In diesem Kapitel werden die verschiedenen Vor- und Nachteile der Gebäudemodellierung aufgeführt, die sich im Verlauf der Arbeit ergeben haben. Weiter werden noch einige Erweiterungsvorschläge der Gebäudemodellierung erwähnt und allgemeine Verbesserungsmöglichkeiten des Grafik-Frameworks *BOOGA* vorgeschlagen, die die Verwendbarkeit in der Praxis weiter steigern können.

15.1 Zusammenfassung der Ergebnisse

Im ersten Teil der Arbeit wurden drei wichtige Modellierungsansätze betrachtet. Der zweite Teil befasste sich mit einem selbstentwickelten objektorientierten Modellierungsansatz, der anhand von Beispielen und Farbbildern erklärt bzw. illustriert wurde. Im dritten Teil wurden die nötigen Grundlagen, Algorithmen und Begriffe eingeführt und die selbstentwickelten Objekte erklärt.

Das Konzept der objektorientierten Modellierung mit eigens entwickelten Objekten, die eine Modellierung nach dem Baukastenprinzip ermöglichen, bringt klar eine Vereinfachung mit sich. Das vorgestellte Beispiel (Abschnitt 8.1) und die Beispiele im Anhang verdeutlichen, dass die Modellierung nun ein grosses Stück intuitiver geworden ist, und der Benutzer seine Gebäude nicht mehr aus "einfachen" Listen zusammenbauen muss, sondern mit spezialisierten Objekten aufbauen kann.

Die Objekte übernehmen hier dem Benutzer die Organisation und die hierarchische Anordnung der Elemente, aus denen ein Gebäude modelliert werden soll. Diese Objekte besitzen spezielle Eigenschaften, die aus der Natur hervorgehen können oder per Definition festgelegt sind. So ist beispielsweise festgelegt, dass eine Front sich immer anhand der Geometrie des Gebäudes orientieren muss und die nötige, lokale Transformation berechnet. Damit kann auch sichergestellt werden, dass sich alle Unterobjekte der Front am richtigen Ort befinden, ohne dass sie selber eingreifen müssen.

Hier kommt nun die hierarchische Struktur der Szenen unter *BOOGA* zum Tragen, die es ermöglicht, ein Objekt zu transformieren, ohne sich um die Unterobjekte zu kümmern. Diese Idee wurde für die Gebäudemodellierung weiterentwickelt, indem von einem allgemeinen Gebäude ausgegangen wird. Das Gebäude selber kann mit weiteren Gebäudeobjekten genauer modelliert werden, indem ihm entsprechend Unterobjekte zugewiesen werden. Diese Objekte verhalten sich entsprechend dem Gebäude und ihren Eigenheiten

und passen ihre geometrische Realisation an. Dabei werden Veränderungen immer an die Unterobjekte weitergereicht. Es hat sich im Verlauf der Arbeit jedoch gezeigt, dass diese einseitige Kommunikation zwischen den Objekten zu streng ist und einer Aufweichung bedarf, um Objekte zu entwickeln, die nicht nur genau von einem Objekt abhängen.

Hierzu wurden einige neue Protokolle eingeführt, die es ermöglichen, auch zwischen zwei Objekten gleicher Hierarchiestufe zu kommunizieren. Damit war es möglich, Objekte wie einen Tunnel, der eine Beziehung zu zwei Fronten besitzt, aber nur in einer Front gleichzeitig definiert sein kann, zu entwickeln. Diese Aufweichung der strengen hierarchischen Struktur ging teilweise auf Kosten der Übersichtlichkeit, kann aber mit der gewonnenen Flexibilität vertreten werden.

Abschliessend lässt sich sagen, dass das hier vorgeschlagene hierarchische Baukastensystem mit der Möglichkeit der *top-down*-Modellierung und seiner Objekthierarchie einen soliden Lösungsansatz darstellt. Dazu beigetragen hat neben einer sorgfältig aufgebauten, mehrstufigen Objekthierarchie der Gebäudeobjekte auch die Trennung zwischen Gebäude und Gebäudeobjekten, den eigentlichen "Bauelementen". Das Gebäude übernimmt hier die Hauptverantwortung der Gebäudemodellierung und versucht, eine in sich abgeschlossene Einheit darzustellen. Die dreistufige Objekthierarchie und ihre Einbettung in BOOGA hat sich als solid und erweiterbar erwiesen und ermöglicht auch eine optimale Ausnutzung von schon vorhandenen Möglichkeiten des Frameworks.

15.2 Erfahrungen bei der Entwicklung

Mit den schon vorhandenen Klassenbibliotheken von BOOGA wurde die Entwicklung der Gebäudemodellierung stark erleichtert. Es konnten etliche Funktionen aus dem Framework übernommen werden, oder Klassen dienten als mögliche Grundlage für Weiterentwicklungen. Mit der zusätzlichen Tatsache, dass die Gebäudemodellierung in der Klasse `Object3D` verankert ist, hängt sie von BOOGA sehr stark ab und kann ohne die entsprechenden Bibliotheken nicht bestehen.

Mit der umfangreichen Vielfalt von Applikationen standen schon zu Beginn der Arbeit alle nötigen Verarbeitungs- und Visualisierungshilfen zur Verfügung. Es entfiel die aufwendige Entwicklung einer eigenen Sprache zur Beschreibung von Szenen wie auch deren Visualisierung. Damit konnte sich die Arbeit hauptsächlich auf die Entwicklung der Gebäudemodellierung beschränken, auch wenn teilweise Aufgaben anfielen, die mit der Modellierung nur am Rande etwas zu tun hatten.

Im Zuge dieser Arbeit wurde BOOGA betreffend Schnitte von geometrischen Objekten in R^2 und R^3 stark erweitert. Ausserdem wurde eine Textur implementiert, die Polygone mit einer Bitmap einfärbt. So konnte ein Beschleunigungsfaktor von zwei bis drei gegenüber einer Beschreibung durch die Shading Language BSL von [Teu 96] erreicht werden.

Die fehlende Dokumentation zu BOOGA führt für einen neuen Entwickler unabdingbar zu einer längeren Einarbeitungszeit, speziell bei umfangreicheren Entwicklungen oder solchen, die stark von BOOGA abhängen. Deshalb war das schon in einer Projektarbeit erworbene Wissen über BOOGA von grossem Nutzen. Ratschläge und Erklärungen wurden bei den Autoren und anderen Mitentwicklern eingeholt. Weiteres Wissen musste oftmals anhand von Beispielen selber erarbeitet werden und führte zu einem eingehenden Studium des Frameworks, wobei dies rückblickend einen grossen Lerneffekt mit sich

brachte.

Für die Entwicklung unter BOOGA hat sich eine "intelligente" Programmierumgebung wie das Entwicklertool SNiFF+ sicher bezahlt gemacht, da das Framework einen Umfang erreicht hat, der sonst nicht mehr überblickbar wäre. Das Tool SNiFF+ ermöglicht das Arbeiten mit umfangreichen Klassenhierarchien in C++ durch die Bereitstellung verschiedener Hilfsmittel. Nebst einem Editor zur Programmentwicklung besitzt SNiFF+ verschiedene Informationsbrowser, die ein schnelles Auffinden von Klassen ermöglichen und die Orientierung im Framework stark erleichtern. Gerade wegen der fehlenden Dokumentation wäre der Zugang ohne dieses leistungsfähige Tool nur schwer möglich.

Für die Entwicklung und das Verstehen von BOOGA war das Buch von [Gam 95] von Vorteil, da das Framwork häufig *design patterns* einsetzt. Nicht zuletzt deshalb besitzt BOOGA ein klares Design und ist einfach erweiterbar. Mit einem eingehenden Verständnis wird der Zugang zu BOOGA deutlich erleichtert und ermöglicht eine gemeinsame Designsprache der Entwickler untereinander.

Die objektorientierte Entwicklung hat sich für die Gebäudemodellierung ausgezahlt. Gerade mit einer geschickt gewählten Objekthierarchie konnte in vielen Fällen Codeduplizierung für die Gebäudemodellierung vermieden werden. Auch hat eine sinnvolle Verankerung in BOOGA dazu beigetragen, dass viele Funktionen direkt übernommen werden konnten und nicht noch einmal implementiert werden mussten. Dies setzt gute Kenntnisse und Erfahrung der objektorientierten Programmierung, speziell unter C++, voraus.

15.3 Erfahrung beim Einsatz der Gebäudemodellierung

Die in dieser Arbeit entwickelte Gebäudemodellierung eignet sich vorzüglich, um kleinere Gebäude wie Häuser zu modellieren. Dies kann mit einem einfachen Texteditor in der BOOGA-Sprache BSDL geschehen. Der Benutzer kann seine Gebäude mit einigen wenigen Objekten modellieren, die sich einfach parametrisieren lassen. Dabei kann der Benutzer je nach Gutdünken seine Gebäude mit den entsprechenden Objekten genauer modellieren. Hier zahlt sich nun die Modellierung nach dem Baukastenprinzip aus, die es erlaubt, in einem ersten Schritt nur die wichtigsten Sachen wie Grundfläche und Höhe zu modellieren. In einem weiteren Schritt können dann immer noch Details modelliert werden.

Bei umfangreicheren Gebäuden und Szenen hat sich gezeigt, dass Gebäudemodellierung nach wie vor möglich ist, auch wenn vieles merklich langsamer und träger wird. Dies ist darauf zurückzuführen, dass BOOGA zu Entwicklungszwecken konzipiert ist und Übersicht der Optimierung vorgezogen wird, um eine hohe Wiederverwendbarkeit zu erreichen. Dies konnte in den meisten Fällen durch den Einsatz von Hardwarebeschleunigern oder mit Hilfe von parallelen Computern[1] aufgefangen werden. Ausserdem sind beim Raytracing durch die Aufteilung des Szenenraumes mit `Grid3D` gewaltige Leistungssteigerungen erreicht worden.

Nebst den Laufzeitproblemen hat sich die Beschaffung der Daten als eigentliches Hauptproblem für die Modellierung von grösseren Szenen entpuppt. Ausser den Grundrissen der Gebäuden werden auch Geländemodelle benötigt, um eine reale Szene nachzubil-

[1] `http://www.sp.unibe.ch/DeepRay/`.

den. Der Einsatz von Datensätzen aus der realen Welt wie beispielsweise den Grundrissen eines Stadtteils ist unumgänglich. Hier sei das Beispiel der Altstadt von Bern (`bern.bsdl3`) erwähnt, das anhand einer digitalen Karte und einem Konvertierungsprogramm erstellt worden ist.

Die Modellierung unter BSDL hat gezeigt, dass mehrseitige Beschreibungen von Gebäuden noch überblickbar sind. Dies kommt nicht zuletzt davon, dass die Gebäudemodellierung dem Benutzer eine hierarchische Modellierung nach dem Baukastenprinzip vorschreibt, die nicht allzu flach ausfällt. Auch wenn sich unter BSDL Teile weitgehend wiederholen können, wurde darauf verzichtet, solche Regelmässigkeit auszunutzen, weil die Objekte sonst ihre Überblickbarkeit verlieren und für den Benutzer nur schwer einsetzbar würden. Damit ist auch ein einigermassen sinnvolles Gleichgewicht zwischen den Funktionalitäten der einzelnen Objekte und der Komplexität der Parametrisierung erzielt worden.

Die mit Hilfe der Gebäudemodellierung erzeugten Bilder zeigen auf, dass mit relativ kleinem Aufwand schon ansehliche Ergebnisse erzielt werden können. Bei der Modellierung wurde die Erfahrung gemacht, dass die korrekte Geometrie des Gebäudes und eine vernünftige Beleuchtung die wichtigsten Voraussetzungen für ansprechende Resultate sind. Deshalb wurde oftmals auf Texturen verzichtet, weil die Wahl und die Aufbereitung der Textur nicht immer einfach ist und Aufwand und Ertrag oftmals in einem schlechten Verhältnis stehen. Die Texturen sind ausschliesslich für die Repräsentation von Oberflächenstrukturen eingesetzt worden. Ausserdem entstehen bei der Repräsentation von grösseren Teilen eines Gebäude mittels Texturen unerwünschte "Tapeteneffekte", speziell bei Texturen, die nicht-planare Objekte darstellen.

Die getroffene Wahl der Objekte für die Gebäudemodellierung hat sich in der Praxis als ansprechend erwiesen. Damit lassen sich die meisten Gebäude mit ihren Eigenheiten modellieren. Dazu stehen ein Gebäude und 17 Gebäudeobjekte zur Verfügung. Damit können beispielsweise ein Flachdach, ein Durchgang zwischen zwei Innenhöfen oder eine komplette Fensterfront mit den zugehörigen Löchern für die Fenster modelliert werden. Neben den hauptsächlich für die Realisation von geometrischen Strukturen zuständigen Objekten gibt es auch drei Objekte (`BottomUser`, `RoofUser` und `SnatchRoof`), die sich nur der Positionierung eines allgemeinen Objektes annehmen. Dabei bestimmen die `Snatch`-Objekte die Positionierung und die Ausrichtung anhand einer konkreten geometrischen Realisation einer bestimmbaren Teilmenge von Objekten.

15.4 Beurteilung der objektorientierten Gebäudemodellierung

Die objektorientierte Gebäudemodellierung ist sicherlich eine mögliche Alternative zu bekannten Modellierungsansätzen. Die in den Abschnitten 1.1 und 6.1 gestellten Forderungen konnten zur vollen Zufriedenheit erfüllt werden, wie folgende Auflistung zeigt:

- Gebäude können mit spezialisierten und abstrakten Objekten modelliert werden, die mit einigen wenigen Parametern auskommen. Dies ermöglicht dem Benutzer ohne grösseren Aufwand, ein Gebäude mit einigen wenigen Objekten ("Bauelementen") zu modellieren.

- Das hierarchische Baukastensystem mit seinen "Bauelementen" erlaubt dem Benutzer eine intuitive *top-down*-Modellierung seiner Gebäude, indem der Benutzer seine Gebäude mit abstrakten Objekten, die einem realen Gebäude oder einem Teil davon entsprechen, modelliert. Ausserdem wird der Aufbau des Gebäudes grösstenteils vorgegeben.

- Eigenheiten und Eigenschaften realer Gebäude sind in den abstrakten Objekten realisiert. Dies ermöglicht Objekte zu entwickeln, die nur mit einer kleinen Anzahl von Parametern auskommen, ohne dabei die Modellierungsmöglichkeiten wirklich einzuschränken.

- Die Objekte übernehmen die Realisation mit konkreten geometrischen Objekten und stimmen sich gegenseitig ab. Nachträgliche Veränderungen einzelner Objekte werden auch von den anderen Objekten des Gebäudes berücksichtigt.

- Das offene Design der Objekthierarchie mit mehreren Stufen hat sich im Verlauf der Entwicklung als sehr geeignet erwiesen. Beim Hinzufügen von neuen Objekten mussten nur in seltenen Fällen Änderungen vorgenommen werden und Codedublizierung konnte in den meisten Fällen vermieden werden.

- Die in Kapitel 9 abgebildeten Beispiele und ihre entsprechenden BSDL-Dateien des Anhangs A zeigen, dass mit minimalem Aufwand ansprechende Ergebnisse erzielt werden können.

Abschliessend lässt sich sagen, dass sich der objektorientierte Modellierungsansatz sehr gut eignet, die Modellierung einer Gruppe von Objekten, hier Gebäuden, zu vereinfachen, indem die Eigenschaften und Eigenheiten der realen Objekte in den abstrakten Objekten festgehalten und implementiert werden.

15.5 Mögliche Erweiterungen

Es bieten sich einige Erweiterungsmöglichkeiten der Gebäudemodellierung und des Frameworks *BOOGA* an, die nachstehend aufgeführt sind:

- In der Praxis hat sich gezeigt, dass bei grösseren Szene wie `bern.bsdl3` mit 618 Gebäuden, die nur aus einem Default–Gebäude und einem Dach bestehen, sogar mit einem Hardwarebeschleuniger interaktives Arbeiten kaum mehr möglich ist. Hier müssen neue Verfahren eingesetzt werden, die die Ausgabe beschleunigen. Eine Möglichkeit ist das Einführen eines *Level-Of-Details*, der die Sichtbarkeit, die Reihenfolge der darzustellenden Objekte und den Detaillierungsgrad des Objektes anhand der Entfernung zum Betrachter bestimmt. Eine weitere Möglichkeit besteht aus dem Aufbereiten der Szene (*pre-processing*), so dass die Ausgabe stark vereinfacht werden kann. Dazu würden sich beispielsweise ein BSP[2]-Baum oder der feudale Algorithmus von [Chen 96] eignen.

[2] **B**inary **S**pace-**P**artitioning.

- Für grössere Szenen wäre ein binäres Format von Vorteil, um einerseits das Einlesen der Szene zu beschleunigen und anderseits die dazu notwendigen Speichermedien zu schonen. Die Datei `bern.bsdl3` beispielsweise ist 1.1 MB gross.

- Auch wenn die Modellierung von Gebäuden stark vereinfacht wurde, ist immer noch die Beschaffung der grundlegenden geometrischen Daten der Gebäude das Hauptproblem. Im Fall der Altstadt Bern [Lim 88] konnte glücklicherweise auf eine Karte in digitalem Format zurückgegriffen werden.

- Eine mögliche Erweiterung, die zu Beginn der Arbeit vorgesehen war und aus Zeitgründen nicht implementiert werden konnte, ist das Bestimmen von einfachsten Daten aus Aufnahmen von Gebäudefronten, um die Modellierung der Fronten zu erleichtern.

- Das Editieren von primitiven Objekten ist mit der Applikation `wxEdit` möglich. Leider werden Gebäude mit ihren Eigenheiten kaum berücksichtigt. Dazu wäre ein eigens dafür entwickeltes Modul denkbar, das das Editieren und Manipulieren der Gebäude und seiner Unterobjekte erlaubt. Nebst einer Einbindung in die Applikation `wxEdit` könnte dies auch durch ein Java-Applet gelöst werden.

- Auch wenn in dieser Arbeit ein Gebäude und 17 Gebäudeobjekte für die Modellierung entwickelt wurden, sind Erweiterungen möglich. Denkbar wären Objekte, die die Modellierung der Ecken des Gebäudes ermöglichen. So könnte beispielsweise das Herausragen des Mauerwerkes in einer Ecke modelliert werden.

- Im Bereich des Zusammenspiels der einzelnen Gebäudeobjekte[3] sind noch Verbesserungen möglich, wobei hier neu Wege bestritten werden müssen. Einerseits wäre ein *pre-processing* möglich, das die Modellierung des Gebäudes genauer untersucht und, falls nötig, Anpassungen vornehmen würde. Anderseits wäre auch die Modellierung der Gebäude mit CSG möglich, was das Herausschneiden von Löchern wesentlich erleichtern würde.

Die hier aufgezeigten Beispiele von möglichen Erweiterung betreffen BOOGA direkt. Auf jeden Fall sollen die an dieser Stelle gemachten Vorschläge zeigen, dass die Gebäudemodellierung und BOOGA noch entwicklungsfähig sind.

[3] Beispielsweise werden zwei sich kreuzende Durchgänge nicht erkannt und entsprechend angepasst.

Literaturverzeichnis

[Abdel 93] R. Abdelhamid. *Das Vieweg LaTeX–Buch: eine praxisorientierte Einführung.* Vieweg, 2. verb. Auflage, 1993.

[Abra 91] S. Abramowski, H. Müller. *Geometrisches Modellieren.* BI Wissenschaftsverlag, 1991.

[Alex 77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel. *A Pattern Language.* Oxford University Press, New York, 1977.

[Bäch 95] R. Bächler. *Entwurf und Implemtierung einer NURBS-Library.* IAM Universität Bern, 1995.

[Bieri 95] H. Bieri. *Vorlesung Geometrisches Modellieren.* IAM Universität Bern, 1995.

[Chen 96] H.M. Chen, W.T. Wang, *The Feudal Priority Algorithm on Hidden-Surface Removal*, SIGGRAPH '96 Conference Proceedings, 1996.

[Debe 96] P.E. Debevec, C.J. Taylor, J. Malik, *Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-Based Approach*, SIGGRAPH '96 Conference Proceedings, 1996.

[Fell 92] W. D. Fellner. *Computergrafik.* BI Wissenschaftsverlag, 1992.

[FvDFH 90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice.* Addison Wensley, second edition, 1990.

[Eber 94] D.F. Ebert, F.K. Musgrave, D. Peachey, K. Perlin, S. Worley. *Texturing and Modeling, A Procedural Approach.* Academic Press, 1994.

[Gam 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object–Oriented Software*, Addison Wesley, 1995.

[Gou 71] H. Gouraud, *Computer Display of Curved Surfaces*, Ph.D. Thesis, University of Utah, 1971.

[Hab 96] P. Habegger, *Ein grafischer Strukturbrowser*, IAM Universität Bern, 1996.

[Hild 97] A. Hildebrand, *Von der Photographie zum 3D-Modell*, Springer-Verlag, 1997.

[Kru 13] E. Kruppa, *Zur Ermittlung eines Objektes aus zwei Perspektiven mit innerer Orientierung*, Sitz.-Ber. Akad. Wiss., Wien, 1913.

[Lim 88] F. Limbach. *Die schöne Stadt Bern.* Benteli Verlag Bern, 1988.

[Pho 75] B.T. Phong, *Illumination for Computer Generated Pictures*, Communications of the ACM, Volume 18, 1975.

[PrSh 88] F. P. Preparata, M. I. Shamos, *Computational Geometry, an Introduction.*, Springer-Verlag, 1988.

[Rour 94] J. O'Rourke, *Computational Geometry in C*, University of Cambridge, 1994.

[StrMas 96] A. Streilein, S. A. Mason, *Photogrammetric reconstruction of buildings for 3D city models*, South African Journal of Surveying and Mapping, Vol. 23, Part 5, August 1996.

[Streit 97] C. Streit, *BOOGA, ein Komponentenframework für Grafikanwendungen*, IAM Universität Bern, 1997.

[Strou 92] B. Stroustrup, *Die C++–Programmiersprache*, Addison Wesley, 1992.

[Suth 63] I.E. Sutherland, *Sketchpad: A man-machine graphic communication system*, AFIPS SJCC 23, 1963.

[Teu 96] T. Teuscher, *Shading Languages*, IAM Universität Bern, 1996.

[Ups 89] S. Upstill, *The RenderMan^{TM} Companion, A Programmer's Guide to Realistic Computer Graphics*, Addison Wesley, 1989.

[Wern 94] J. Wernecke, *The Inventor Mentor*, Addison Wensley, second edition, 1994.

[Whi 80] T. Whitted, *An Improved Illumination Model for Shaded Display*, Communications of the ACM, Volume 23, Number 6, 1980.

Anhang A

Beispieldateien

Dieses Kapitel enthält zwei vollständige Beschreibungen von Szenen mit einem Gebäude. Weiter ist der Source-Code der Applikation `buildingPlacer` angegeben.

A.1 `building_nice.bsdl3`

Die folgende Beschreibung wiedergibt die im Abschnitt 8.1 erklärte Modellierung eines einfachen Hauses mit Fenstern, Türe, Dach und Kamin.

```
/*
 * $RCSfile: building_nice.bsdl3,v $
 *
 * A simple example of a building with a roof, bottom, faces
 * and fronts. It contains a snatch object.
 *
 * Copyright (C) 1997, Thierry Matthey <matthey@iam.unibe.ch>
 *                     University of Berne, Switzerland
 *
 * All rights reserved.
 *
 * This software may be freely copied, modified, and redistributed
 * provided that this copyright notice is preserved on all copies.
 *
 * You may not distribute this software, in whole or in part, as part of
 * any commercial product without the express consent of the authors.
 *
 * There is no warranty or other guarantee of fitness of this software
 * for any purpose.  It is provided solely "as is".
 *
 * ------------------------------------------------------------------------
 *  $Id: building_nice.bsdl3,v 1.1 1997/04/15 08:37:43 matthey Exp $
 * ------------------------------------------------------------------------
 */
using 3D;

// Definition of the camera and the lightsources

camera {
  perspective {
    resolution (600, 400);
```

```
    eye [60,60,20];
    lookat [0,0,10];
  }
  background [.8,.8,.9];
};

pointLight (.8, [1,1,1]) { position [-500,500,1000]; }
pointLight (.2, [1,1,1]) { position [500,500,100]; }
ambientLight (.5, [1,1,1]);

// Definition of the colors

define green whitted { ambient [.3,.4,.2]; diffuse [.3,.4,.2]; }
define black whitted { ambient [.1,.1,.1]; diffuse [0,0,0]; }
define parkett whitted { ambient [.3,.2,.2]; diffuse [.3,.2,.2]; }
define glasDunkel whitted {
  ambient [60/255,90/255,80/255];
  diffuse [60/255,90/255,80/255];
}
define dachRot whitted {
  ambient [180/255,115/255,97/255];
  diffuse [180/255,115/255,97/255];
}
define rock whitted {
  ambient [220/255,200/255,170/255.1];
  diffuse [220/255,200/255,170/255];
}

// Definition of the window

define window list {
  box([-10,-8,2],[10,-7,-1]);
  box([-10,8,0.5],[10,7,-1]);
  box([-10,8,0.5],[-9,-7,-1]);
  box([10,8,0.5],[9,-7,-1]);
  box([-3-.2,-8,-.5],[-3+.2,8,-1]);
  box([3-.2,8,-.5],[3+.2,-8,-1]);
  box([-10,-8,-1],[10,8,-1.1]){glasDunkel;}
  rock;
  scale[0.3,0.3,0.3];
}

// Definition of the small window

define windowSmall list {
  box([-8,-7,2],[8,-6,-1]);
  box([-8,7,0.5],[8,6,-1]);
  box([-8,7,0.5],[-7,-6,-1]);
  box([8,7,0.5],[7,-6,-1]);
  box([-.2,-7,-.5],[.2,7,-1]);
  box([-8,-7,-1],[8,7,-1.1]){glasDunkel;}
  rock;
  scale[0.25,0.25,0.3];
}

// Definition of the door
```

```
define door list {
  box([-7,0,0.2],[-7+.5,25,-1]);
  box([7,0,0.2],[7+.5,25,-1]);
  box([-7,0,5],[7,0.2,-1]);
  box([-7,25,0.2],[7,25-.5,-1]);
  box([-7,0,-1],[7,25,-1.1]){parkett;}
  box([-5,12,-1],[-2,13,-.8]){black;}
  rock;
  scale[0.3,0.3,0.3];
}

// Definition of the chimney

define chimney list {
  box ([0,-.75,0],[-1.5,.75,3]);
  box ([0.25,-1,3],[-1.75,1,3.25]);
  rock;
}

// The nice house

building (20,[-10,-20, 0],[ 10,-20, 0],[ 10, 20, 0],[-10, 20, 0]){
  // Roof defined by points and ledge 2
  roofpoint(2,[0, 17,7.5],[0,-17,7.5]) {
    dachRot;
  }
  // Default-texture
  rock;
  // Definiton of the border
  bottomborder(0.5,0.5,0.5,0.1){height(9);}
  // Back-front
  fronttri(0,0,[.3,1.25],[.7,1.25]){
    faceitem([0,0],[1,1],2,2){
      window;
      hole;
      displacement[0.5,0.4];
    }
    faceitem([0.3,1],[.7,1.3],1,1){
      windowSmall;
      hole;
      displacement[0.5,0.3];
    }
  }
  // Left-front
  frontrect(1,0,0,1){
    faceitem([0,0],[1,1],3,2){
      window;
      hole;
      displacement[0.5,0.4];
    }
  }
  // Front-front
  fronttri(2,0,[.3,1.25],[.7,1.25]){
    faceitem([0,0.5],[1,1],2,1){
      window;
      hole;
      displacement[0.5,0.4];
```

```
    }
    faceitem([0,0],[1,0.5],1,1){
      window;
      hole;
      door;
      displacement[0.5,0];
    }
    faceitem([0.3,1],[.7,1.3],1,1){
      windowSmall;
      hole;
      displacement[0.5,0.3];
    }
  }
  // Right-front
  frontrect(3,0,0,1){
    faceitem([0,0],[1,1],3,2){
      window;
      hole;
      displacement[0.5,0.4];
    }
  }
  // Placing the chimney
  snatch([5,6]){chimney;}
}

// grass
box([-1000,-300,-1],[1000,1000,0]){green;}
```

A.2 building_example.bsdl3

Diese Beispiel modelliert das Gebäude von Abbildung 9.5. Es enthält einen Innenhof und acht Fronten. Die Dachluken werden mit dem Snatch-Objekt `SnatchRoof` auf das Dach positioniert. Weiter enthält das Gebäude neben Fenstern auch Lauben und zwei Durchgänge.

```
/*
 * $RCSfile: $
 *
 * An example with arbours, tunnels and snatch-objects.
 *
 * Copyright (C) 1997, Thierry Matthey <matthey@iam.unibe.ch>
 *                     University of Berne, Switzerland
 *
 * All rights reserved.
 *
 * This software may be freely copied, modified, and redistributed
 * provided that this copyright notice is preserved on all copies.
 *
 * You may not distribute this software, in whole or in part, as part of
 * any commercial product without the express consent of the authors.
 *
 * There is no warranty or other guarantee of fitness of this software
 * for any purpose.  It is provided solely "as is".
 *
 * ------------------------------------------------------------------------
 *  $Id: building_example.bsdl3,v 1.5 1997/04/15 08:37:34 matthey Exp $
 * ------------------------------------------------------------------------
 */

using 3D;

// Definition of the camera and the lightsource

camera {
  perspective {
    eye [100,600,300];
    lookat [0,0,0];
    resolution (550, 300);
  }
  background [157/255,180/255,253/255];
};

pointLight (1.0, [1,1,1]) { position [-500,500,1000]; }
pointLight (0.2, [1,1,1]) { position [200,800,1000]; }
pointLight (0.2, [1,1,1]) { position [200,800,100]; }

// Definition of the colors and textures

define green2         whitted { ambient [.2,.2,.2]; diffuse [.3,.6,.2]; }
define grey           whitted { ambient [.2,.2,.2]; diffuse [.7,.7,.7]; }
define sandstein      whitted { ambient [.3,.2,.25];diffuse [220/255,200/255,170/255];}
define sandsteinWeiss whitted { ambient [.4,.4,.4]; diffuse [220/255,210/255,200/255];}
define glasDunkel     whitted { ambient [.1,.1,.1]; diffuse [60/255,90/255,80/255];}
define dachRot        whitted { ambient [.1,.1,.1]; diffuse [180/255,115/255,97/255];}
//define fronttexture polygonmapper( "rock10.jpg",0.03,0.03,0);
```

```
// Definition of the dormer

define dormer
  building(6,[-10,-2.5,0],[0,-2.5,0],[0,2.5,0],[-10,2.5,0]){
    sandstein;
    fronttri(0,0,[0,1.25]);
    frontrect(1,0,0,1){
      faceitem([.1,.1],[.9,.9],1,1){
        displacement[.5,.5];
        box([-1.5,-2,-.1],[1.5,1.4,-0.01]){glasDunkel;}
        hole([1.5,-2],[1.5,0.5],[1.37109,0.89375],[1.03125,1.175],
             [0.550781,1.34375],[0,1.4],[-0.550781,1.34375],
             [-1.03125,1.175],[-1.37109,0.89375],[-1.5,0.5],[-1.5,-2]);
        // Path of the hole:
        //  1.5 -2.0 m
        //  1.5 0.5 l
        //  1.5 1.7 -1.5 1.7 -1.5 0.5 c
        //  -1.5 -2.0 l
      }
    }
    fronttri(2,0,[1,1.25]);
    frontrect(3,0,0,1.25);
    rooflayer ([0.5,0,0],[0.5,0.1,0]){dachRot;}
  }

// Definition of the window

define window list {
  // Border of the window
  box([-3,-6,-1],[-2.5,6,1]){sandsteinWeiss;}; // left
  box([-4,-6,-1],[-2.5,6,0]);
  box([3,-6,-1],[2.5,6,1]){sandsteinWeiss;};   // right
  box([4,-6,-1],[2.5,6,0]);
  box([-4,-6,-1],[4,-5,2]){sandsteinWeiss;};   // bottom
  box([-3,6,-1],[3,5.5,1]){sandsteinWeiss;};   // top
  // Arc of the window
  box([0,0,-1],[3.5,-.5,1]){
    rotateZ(30);
    translate[-3,6,0];
    sandsteinWeiss;
  }
  box([0,0,-1],[-3.5,-.5,1]){
    rotateZ(-30);
    translate[3,6,0];
    sandsteinWeiss;
  }
  box([-3,5.5,-1],[3,8,0.01]){sandsteinWeiss;};
  box([-4,5.5,-1],[4,8,0]);
  // Glas of the window
  box([-3,-5,-1],[3,5.5,-.5]){glasDunkel;}
}

// Definition of the Building

building (50, [200, 100, 0], [-200, 100, 0], [-200, -100, 0], [200, -100, 0]) {
  sandstein;
```

```
frontrect (0, 0, 0, 1) {
  facetunnel ([0.425, 0], [0.575, 0.5],[0.4, 0], [0.6, 0.5], 2, 1) {
    bezierparm1 0.9;
    numberofpoints 9;
    walloff;
  };
  faceitem([0,0.5],[1,1],13,1){
    window;
    hole;
    displacement[0.5,0.5];
  }
};
frontrect (1, 0, 0, 1){
  faceitem([0,0.5],[1,1],7,1){
    window;
    hole;
    displacement[0.5,0.5];
  }
}
frontrect (2, 0, 0, 1) {
  facetunnel ([0.425, 0], [0.575, 0.5],[0.4, 0], [0.6, 0.5], 0, 1) {
    bezierparm1 0.9;
    numberofpoints 9;
    walloff;
  };
  faceitem([0,0.5],[1,1],13,1){
    window;
    hole;
    displacement[0.5,0.5];
  }
};
frontrect (3, 0, 0, 1){
  faceitem([0,0.5],[1,1],7,1){
    window;
    hole;
    displacement[0.5,0.5];
  }
}
frontrect (0, 1, 0, 1){
  facearbour([0,0],[0.4,0.5],3);
  facearbour([0.6,0],[1,0.5],3);
  faceitem([0,0.5],[1,1],9,1){
    window;
    hole;
    displacement[0.5,0.5];
  }
}
frontrect (1, 1, 0, 1){
  facearbour([0,0],[1,0.5],3);
  faceitem([0,0.5],[1,1],4,1){
    window;
    hole;
    displacement[0.5,0.5];
  }
}
frontrect (2, 1, 0, 1){
  facearbour([0,0],[0.4,0.5],3);
```

```
    facearbour([0.6,0],[1,0.5],3);
    faceitem([0,0.5],[1,1],9,1){
      window;
      hole;
      displacement[0.5,0.5];
    }
  }
  frontrect (3, 1, 0, 1){
    facearbour([0,0],[1,0.5],3);
    faceitem([0,0.5],[1,1],4,1){
      window;
      hole;
      displacement[0.5,0.5];
    }
  }
  roofplane(5,30){dachRot;}
  hole ([150, -50, 0],[-150, -50, 0],[-150, 50, 0],[150, 50, 0]);
  bottomflat{height(0.01); grey;}
  bottomborder(1,1,2,1){height(25);}
  // Front
  snatch([-200+400/26 + 400/13*1,92.5]){dormer;}
  snatch([-200+400/26 + 400/13*3,92.5]){dormer;}
  snatch([-200+400/26 + 400/13*5,92.5]){dormer;}
  snatch([-200+400/26 + 400/13*7,92.5]){dormer;}
  snatch([-200+400/26 + 400/13*9,92.5]){dormer;}
  // Back
  snatch([-200+400/26 + 400/13*11,92.5]){dormer;}
  snatch([-200+400/26 + 400/13*1,-92.5]){dormer;}
  snatch([-200+400/26 + 400/13*3,-92.5]){dormer;}
  snatch([-200+400/26 + 400/13*5,-92.5]){dormer;}
  snatch([-200+400/26 + 400/13*7,-92.5]){dormer;}
  snatch([-200+400/26 + 400/13*9,-92.5]){dormer;}
  snatch([-200+400/26 + 400/13*11,-92.5]){dormer;}
  // Left
  snatch([-192.5,-100+200/14 + 200/7*1]){dormer;}
  snatch([-192.5,-100+200/14 + 200/7*3]){dormer;}
  snatch([-192.5,-100+200/14 + 200/7*5]){dormer;}
  // Right
  snatch([192.5,-100+200/14 + 200/7*1]){dormer;}
  snatch([192.5,-100+200/14 + 200/7*3]){dormer;}
  snatch([192.5,-100+200/14 + 200/7*5]){dormer;}
};

// ground
box([-1000,-1000,-1],[1000,1000,0]){green2;}
box([-150, -50, -1], [150, 50, 0.01]){grey;}
```

A.3 buildingPlacer.C

Die Applikation ermöglicht es, Gebäude in das bestehende Gelände korrekt einzusetzen (vgl. Abschnitt 8.2). Es wird eine Szene eingelesen, die Gebäude werden positioniert und anschliessend wird die Szene wieder in eine Datei zurückgeschrieben.

```
/*
 * $RCSfile: buildingPlacer.C,v $
 *
 * Copyright (C) 1997, Thierry Matthey <matthey@iam.unibe.ch>
 *                     University of Berne, Switzerland
 *
 * All rights reserved.
 *
 * This software may be freely copied, modified, and redistributed
 * provided that this copyright notice is preserved on all copies.
 *
 * You may not distribute this software, in whole or in part, as part of
 * any commercial product without the express consent of the authors.
 *
 * There is no warranty or other guarantee of fitness of this software
 * for any purpose.  It is provided solely "as is".
 *
 * ------------------------------------------------------------------------
 *  $Id: buildingPlacer.C,v 1.5 1997/1/28 13:50:32 matthey Exp $
 * ------------------------------------------------------------------------
 */

#include <string.h>  // strcmp()
#include <stdlib.h>  // atoi()

#include "CollectBuilding.h"
#include "booga/object/Ray3D.h"
#include "booga/object/Ray3DFactory.h"
#include "booga/building/Building.h"
#include "booga/component/Parser3D.h"
#include "booga/component/BSDLParserInit.h"
#include "booga/component/PrintWorld3D.h"
#include "booga/component/ConfigurationHandlers.h"
#include "booga/component/BSDL3DWriter.h"
#include "booga/component/SingleFileStore.h"
#include "booga/component/MultiFileStore.h"

static void usage(const RCString& name);
static void parseCmdLine(int argc, char* argv[], char& mode,
                         RCString& in, RCString& out);

int main(int argc, char* argv[])
{
  //
  //  Setup world.
  //
  Configuration::setOption(Name("Report.ErrorStream"),Name("cerr"));

  initBSDLParserGlobalNS();
  initBSDLParser3DNS();
```

```
RCString in, out;
char mode = ' ';

parseCmdLine(argc, argv, mode, in, out);

//
// Read scene
//
World3D* world3D = new World3D;
Ray3D* ray;
List<Vector3D> tmp;
Parser3D parser;
parser.setFilename(in);
parser.execute(world3D);

//
// Collect the buildings and switch them off
//
CollectBuilding buildingCollector;
buildingCollector.execute(world3D);

cerr << "Number of buildings found:  " << buildingCollector.count() << endl;

for (buildingCollector.first(); !buildingCollector.isDone(); buildingCollector.next())
  buildingCollector.getObject()->turnOff();

world3D->getObjects()->computeBounds();

//
// Place the buildings
//
Real z;
Building* building;
long count = 0;
Vector3D a(0,0,10+world3D->getObjects()->getBounds().getMax().z());
Vector3D b(0,0,-1);
Vector3D c;

for (buildingCollector.first(); !buildingCollector.isDone(); buildingCollector.next()){
  building = buildingCollector.getObject();
  tmp = building->getVertices();
  z = 0;
  for(long j=0;j<tmp.count();j++){
    c = tmp.item(j)*(building->getTransform().getTransMatrix());
    ray = Ray3DFactory::createRay(a+Vector3D(c.x(),c.y(),0),b);
    if (world3D->getObjects()->intersect(*ray) &&
        (j == 0 || c.z() - ray->getHitPoint().z() > z))
      z = c.z()-ray->getHitPoint().z();
    delete ray;
  }
  if (!equal(z,0,.001)){
    building->addTransform(TransMatrix3D::makeTranslate(0,0,-z));
    count++;
  }
}
cerr << "Number of buildings placed: " << count << endl;
```

```
  //
  // Switch the buildings on
  //
  for (buildingCollector.first(); !buildingCollector.isDone(); buildingCollector.next())
    buildingCollector.getObject()->turnOn();

  //
  // Write BSDL format.
  //
  DocumentStore* docuStore = NULL;
  if (mode == 'm') {
    docuStore = new MultiFileStore(out);
  } else {
    docuStore = new SingleFileStore(out);
  }

  BSDL3DWriter writer(*docuStore);
  writer.execute(world3D);

  delete world3D;

  return 0;
}

void parseCmdLine(int argc, char* argv[], char& mode, RCString& in, RCString& out)
{
  int next = 1;
  mode = ' ';
  if ((argc == 2 && !strcmp(argv[1], "-h")) || argc == 1) {
    usage(argv[0]);
    exit(0);
  }

  if (!strcmp(argv[1], "-m")) {
    mode = 'm';
    next++;
  }

  if (next >= argc) return;
  in = argv[next];
  next++;

  if (next >= argc) return;
  out = argv[next];
  next++;
}

void usage(const RCString& name)
{
    cerr << "Usage: " << name << " [-m] [in-file [out-file]]\n";
    cerr << " where:\n";
    cerr << "  -m             : (optional) multiple file output\n";
    cerr << "                      One file per shared object or material\n";
    cerr << "  in-file        : (optional) filename of input\n";
    cerr << "  out-file       : (optional) filename of output\n";
}
```