Generative Modellierung der Burg im Kondertal

Studienarbeit im Fachbereich Informatik
Universität Koblenz-Landau
Verfasser: Sebastian Stolz
tlotr@uni-koblenz.de
Betreuer: Prof. Dr. Dietrich Paulus

5. Februar 2008

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Siedlungsgeschichte im Rhein-Mosel-Dreieck reicht zurück bis in die römische Zeit. Entlang der beiden großen Flüsse finden sich zahlreiche Beispiele historischer Architektur. Besonders häufig trifft man auf sogenannte „wüst gefallene“ Wehranlagen. Eine Wüstung ist im weitesten Sinne „jede aufgegebene menschliche Niederlassung“ [1].
In diese Kategorie lässt sich auch die ehemalige Burganlage im Kondertal einordnen, die sich auf dem Nordwest-Ausläufer des Hinterberges befindet (siehe Abbildung 1.1). Entstanden etwa im frühen 12. Jahrhundert diente sie vermutlich der Kontrolle von Verkehrswegen, die die in unmittelbarer Nähe gelegene Moselfurt nutzten. Auch die Silber- und Bleierzvorkommen der Region, die in Bergwerken abgebaut wurden, und natürlich die Erztransporte konnten unter den Schutz der Burg gestellt werden. Im Frühjahr 2005 begannen archäologische Untersuchungen und Vermessungen, die zum jetzigen Zeitpunkt noch andauern. Erste Ergebnisse wurden in einer Magisterarbeit des Institutes für Kunstwissenschaften der Universität Koblenz vorgestellt. Um eine genauere Vorstellung der Burganlage zu erhalten sollte ein Computermodell erstellt werden. Die praktische Umsetzung dieses Modells ist Thema der vorliegenden Studienarbeit.

1.1 Aufgabenstellung

Im Vorfeld der Studienarbeit wurden verschiedene Ansätze und Lösungsmöglichkeiten diskutiert. Von der Erstellung eines „einfachen 3D-Modells“ mittels einer dazu mächtigen Software kam man schnell ab. Stattdessen sollte das Ziel der Arbeit ein Programm

[1] (Janssen 1975)

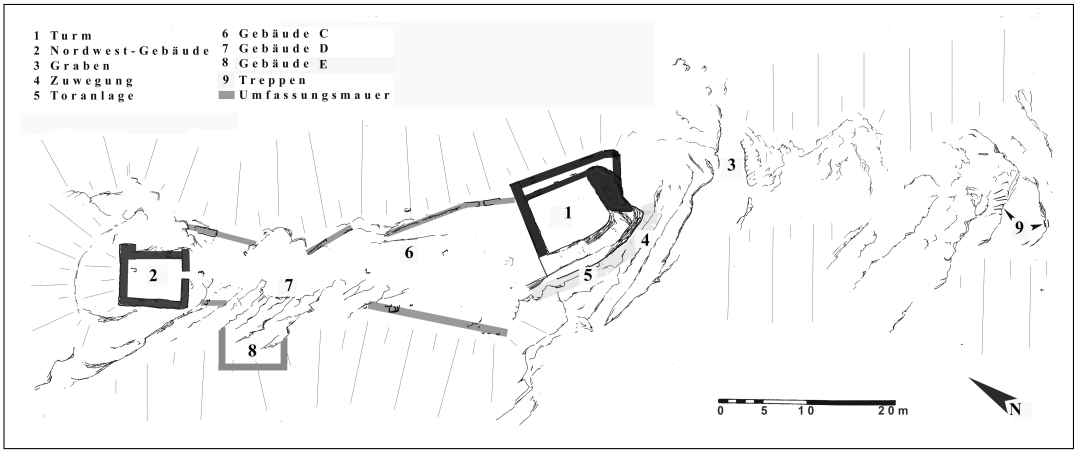


Abbildung 1.1: Grundriss der Burg im Kondertal

sein, dass es dem Benutzer ermöglicht die Burganlage interaktiv aufzubauen und in beliebiger Form zu verändern. Anhand der oben erwähnten ersten Ergebnisse archäologischer Untersuchungen kann dann beispielsweise ein Kunsthistoriker die Festung am Bildschirm „erbauen“.
Zunächst soll der Nutzer an einen vormodellierten Grundriss gebunden sein. Diese Beschränkung kann in weiteren Stadien des Programmes eventuell aufgehoben werden. Zusammenfassend lassen sich folgende Anforderungen an die Studienarbeit formulieren:

- Erarbeiten der ersten Ergebnisse kunsthistorischer Forschung
- Modellierung eines einfachen Grundrisses der Burg
- Einbinden von Interaktionselementen
- Bereitstellung einer graphischen Oberfläche

1.2 Werkzeuge

Um den Forderungen nach interaktiver Veränderung von Modellparametern nachzukommen wurde beschlossen das Modell durch sogenanntes „Prozedurales Modellieren“/ „Generatives Modellieren“ (siehe Kapitel 2) aufzubauen. Dieses Paradigma wird von der stackbasierten *Generative Modeling Language* (GML siehe 3) umgesetzt, die auch das hauptsächliche Werkzeug der Arbeit darstellt. Die Benutzeroberfläche des Programms

wurde mit C # erstellt. Als Entwicklungsumgebung diente das .NET Framework, da somit eine hohe Portabilität gewährleistet war [2].
Da die GML - Engine *OpenGL* zur Darstellung des Codes benutzt, wurde eine auf OpenGL basierende Ausgabeanzeige notwendig. Diese stellte das *TAO - Framework*[3] zur Verfügung.

1.3 Zum Aufbau der Arbeit

Im Anschluss an die Einleitung wird in **Kapitel 2** die Idee der generativen Modellierung vorgestellt. Neben einer Definition sollen vor allem die Möglichkeiten betrachtet werden, die dieser Ansatz dreidimensionale Formen zu beschreiben mit sich bringt. **Kapitel 3** befasst sich mit der *Generative Modeling Language* - einer stackbasierten Scriptsprache, die generatives Modellieren möglich macht. Nach dieser Einführung werden in
Kapitel 4 die Interaktionselemente der GML, die so genannten *Gizmos* beschrieben. Die Ergebnisse der Studienarbeit sind in **Kapitel 5** zu finden. Abschließend wird im letzten Teil der Arbeit (**Kapitel 6** und **Kapitel 7**) ein Ausblick und ein Fazit gegeben.

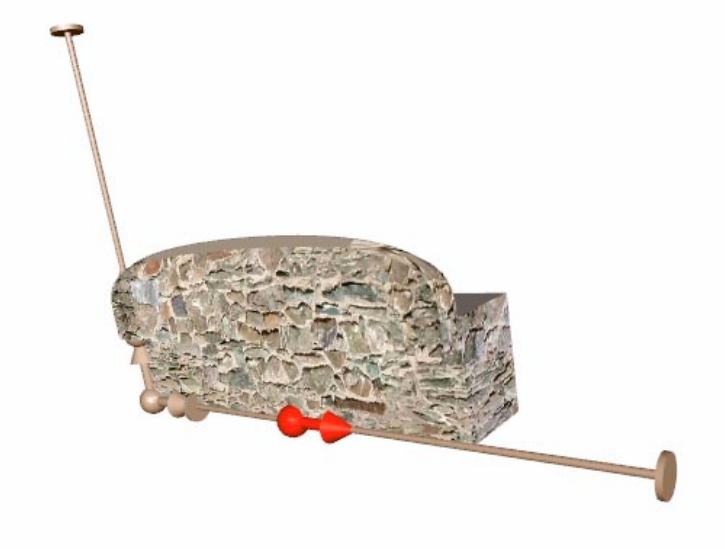


Abbildung 1.2: GML-Modell mit Gizmo

[2] http://msdn2.microsoft.com/de-de/netframework/default.aspx
[3] (TAO)

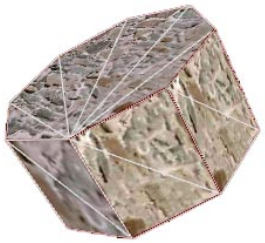
Kapitel 2

Generative Modellierung

Die Einsatzmöglichkeiten dreidimensionaler Modelle vergrößern sich von Jahr zu Jahr. Ob in der medizinischen Bildverarbeitung, der Konstruktion, der Forschung oder auch im Bereich von Film und Fernsehen. Das Erstellen von 3D-Modellen ist in der Regel teuer, langwierig und erfordert speziell ausgebildete Fachleute. Hinzu kommt das Problem wachsender Modellgrößen auf der einen und begrenzter Änderbarkeit bzw. Wiederverwendbarkeit auf der anderen Seite. Oft ist es jedoch der Fall, dass aus einem Objekt durch geringe Eingriffe ein völlig neues Objekt entstehen kann.
Die generative Modellierung versucht diesen Nachteilen entgegenzuwirken. Sie greift die Frage auf, wie man Modelle schnell erstellen kann, ohne dass ihre Repräsentation zu groß wird und wie man eine hohe Wiederverwendbarkeit erreicht.

2.1 Definition

Vereinfacht lässt sich sagen, dass Generative Modellierung ein alternativer Ansatz zur Beschreibung von dreidimensionaler Form ist. Ein Modell wird nicht mehr durch geometrische Primitive repräsentiert, sondern durch eine Funktion, aus der sich das Modell berechnen lässt. Abbildung 2.1 zeigt ein einfaches Beispiel:
Um das gesamte (hier natürlich sehr simple) Modell zu speichern genügt es die Funktion auf der rechten Seite der Abbildung abzulegen. Das heißt, dass der Konstruktionsprozess selber und nicht mehr nur das Ergebnis gespeichert wird.



```
deleteallmacros newmacro clear

/bruchstein1 setcurrentmaterial
(0,0,0) (0,-1,0) 1.0 8 circle
1 poly2doubleface
(0,1,3) extrude
```

Abbildung 2.1: Geometrisches Primitiv und prozedurale Beschreibung

2.2 Möglichkeiten

Welche Möglichkeiten ergeben sich aus diesem Paradigmenwechsel? Der augenscheinlichste Vorteil ist natürlich die geringe Modellgröße, die nicht mehr linear von der Dateigröße abhängt. Abbildung 2.2 zeigt links das Modell eines Fensters, welches in einer hohen Auflösung etwa 7 Millionen Dreiecke besitzt. Das Modell des Doms auf der rechten Seite der Abbildung beinhaltet unter anderem 70 dieser Fenster und lässt sich aus nur 130 KB Code erzeugen.

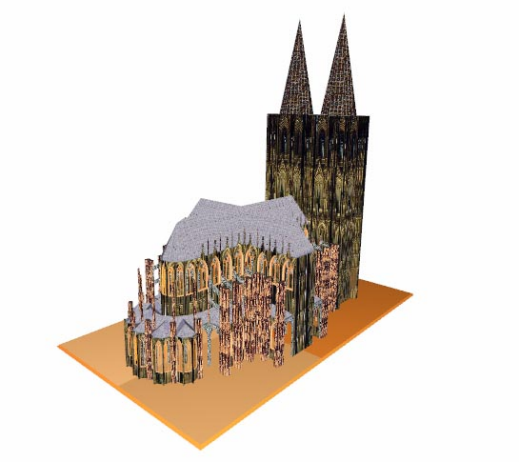


Abbildung 2.2: Beispiel-Modell erstellt mit GML, dargestellt mit der GMLMFC, Autor Sven Havemann

An diesem Beispiel lässt sich ein weiterer Vorteil generativer Modellierung veranschaulichen: Das einfache Verändern eines fertiggestellten Modells. Da die Kathedrale aus Funktionen generiert wird, die mit Parametern rechnen, lassen sich Änderungen am Modell durch Modifikation eben dieser Parameter leicht realisieren. Möchte man beispielsweise alle Fenster der Kathedrale etwas größer machen, reicht es aus, den Parameter, der die Fenstergröße festlegt, zu modifizieren. Ein weiterer Vorteil, den die prozedurale Repräsentation von Modellen mit sich bringt liegt darin begründet, dass viele Objekte eine gemeinsame Struktur haben und damit aus einer gemeinsamen Funktion erstellt werden können. Ein Beispiel zeigt Abbildung 4.2: Aus einem Stuhl lässt sich durch einige wenige Änderungen ein Tisch formen.

Abbildung 2.3: Ein Modell - zwei Repräsentationen: Stuhl und Tisch

Die generative Repräsentation ermöglicht es demnach auch, bereits gelöste Konstruktionsaufgaben wiederzuverwenden. Diese können je nach Anwendung in Bibliotheken gesammelt werden.
Eine praktische Umsetzung dieser Theorie der Generativen Modellierung stellt die *Generative Modeling Language* dar.

Kapitel 3

Generative Modeling Language

Die *Generative Modeling Language* , kurz GML, ist eine stackbasierte Programmiersprache, die eine generative Modellbeschreibung ermöglicht. Sie wurde im Rahmen einer Dissertation von Dr.-Ing. Sven Havemann entwickelt. Eine ausführliche Beschreibung der Sprache wird demnach in seiner Arbeit[1] gegeben[2]. Sie ähnelt in Syntax und Aufbau der Sprache *Postscript* von Adobe[3].

3.1 Eigenschaften

GML bringt eine auf OpenGL basierende Runtime - Engine (Abbildung 3.1) mit und erlaubt damit eine Beschreibung parametrisierter 3D-Modelle und deren Darstellung.
Die vier Bestandteile der Modeling Engine werden in den folgenden Unterabschnitten etwas genauer erklärt.

[1](Havemann 2005)
[2]http://www.digibib.tu-bs.de/?docid=00000008
[3](pos)

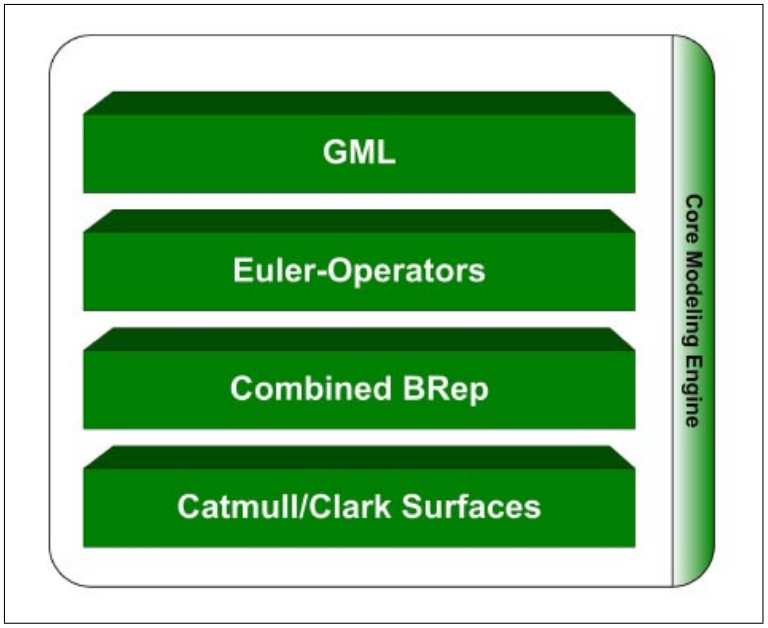


Abbildung 3.1: GML Modeling Engine [HAV05]

3.1.1 GML - Interpreter

Die oberste Schicht stellt den GML Interpreter dar (Abbildung 3.2). Da sich Aufbau und Syntax der GML an Adobe's *Postscript* anlehnen, werden auch hier vier interne Stacks benutzt:

- **operand stack**: Stellt Argumente für Operatoren bereit und speichert Ergebnisse ab.
- **dictionary stack**: Hier werden „dictionaries" abgelegt, in denen der Interpreter Ausdrücke sucht.
- **execution stack**: Hier werden ausführbare Objekte abgelegt.
- **graphics state stack**: Auf diesem Stack werden die Eigenschaften von Objekten (Farbe, Kantenschärfe...) abgelegt.

Diese Schicht stellt die eigentliche *Generative Modeling Language* dar. Ihre Syntax sowie einige Beispiele werden in den Abschnitten 3.2 und 3.3 gezeigt.

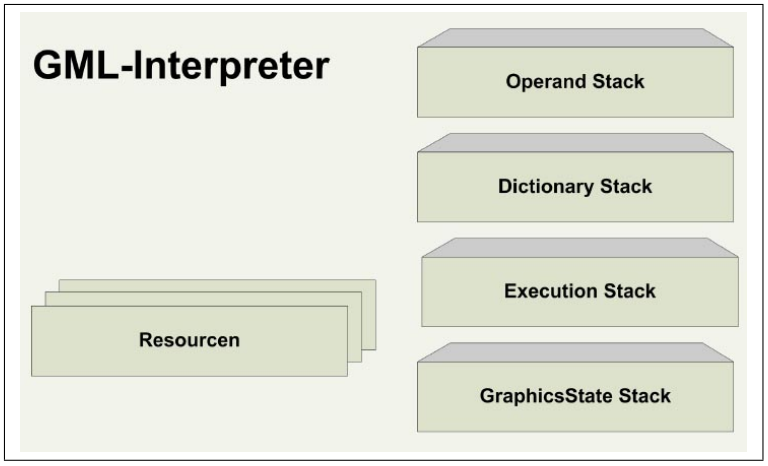


Abbildung 3.2: GML Interpreter [HAV05]

3.1.2 Euler Operatoren

Die Euler Operatoren erlauben einen gesicherten Zugriff auf die unter ihnen liegende Netzstruktur. Gesichert deshalb, weil sie die Konsistenz des Netzes nicht gefährden.
Die GML besitzt 13 Euler Operatoren zur Manipulation des Netzes. Dieses besteht aus Kanten, Eckpunkten und Flächen. Zusätzlich besitzen die drei Elemente je ein Attribut:

- Kanten eine Schärfedefinition
- Eckpunkte eine Position
- Flächen ein Material

Fünf Euler Operatoren dienen nun der Erzeugung der Netzelemente. Jeder der fünf besitzt einen inversen Operator und zusätzlich gibt es drei Operatoren zum Ändern der Attribute. Abbildung 3.3 zeigt die Euler Operatoren der GML.

3.1.3 Combined BRep

Die Combined BRep (kurz cB-rep) bieten eine Datenstruktur zur Darstellung von Netzen. Sie ermöglichen es planare Flächen und Freiformgeometrie (beispielsweise abgerundete Kanten) in einer Datenstruktur zusammenzufassen.

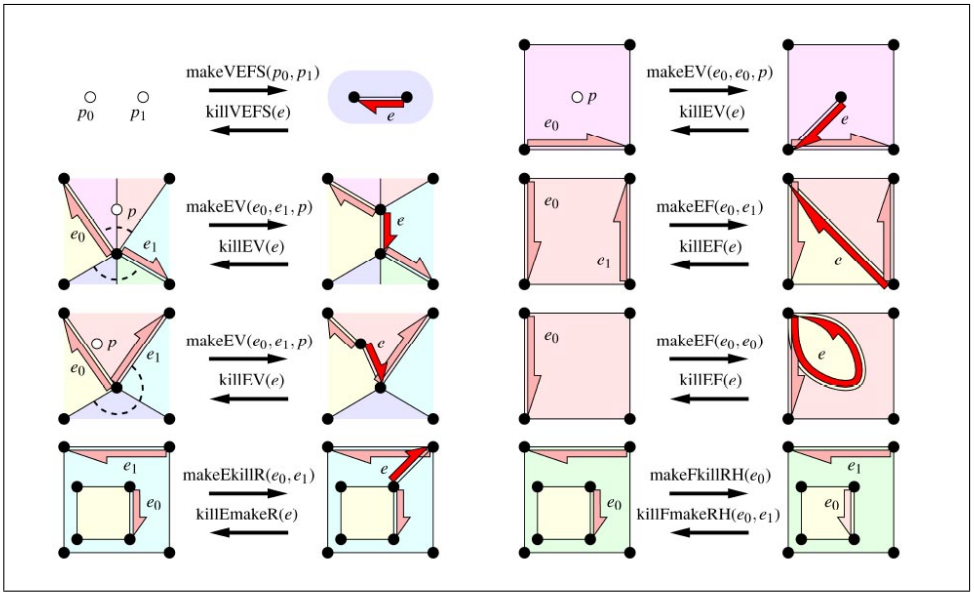


Abbildung 3.3: Euler Operatoren [HAV05]

3.1.4 Catmull/Clark Oberflächen

Die unterste Schicht stellt der Grafik-Hardware die bereits aufbereiteten Primitive bereit. Die GML verwendet dazu Unterteilungsflächen vom Typ Catmull/ Clark.
Aus den grob vorgegebenen Kontroll-Meshes der Combined BReps werden glatte Oberflächen geschaffen. Dazu wird das Netz aus Kontrollpunkten iterativ verfeinert.

„Mit den hier vorgestellten Techniken kann man Modelle einfach an beliebigen Stellen ändern und die Tesselierung wird „on the fly“ nur dort neu berechnet und zwar in genau der Auflösung die benötigt wird. Das ist DIE Grundvoraussetzung für interaktive Modellierung.“ [Hav05]

3.2 Syntax

Die *Generative Modeling Language* ist eine sehr einfach aufgebaute, stackbasierte Sprache. Sie ähnelt in Form und Aufbau einem Postscript - Programm. Jede Funktion nimmt die von ihr benötigten Eingabeparameter vom Stack, verarbeitet sie und legt ein oder auch mehrere Ergebnisse auf den Stack zurück. Diese Ergebnisse (und natürlich evtl. auch alles was darunter abgelegt wurde) werden zu den Eingabeparametern der nächsten Funktion.

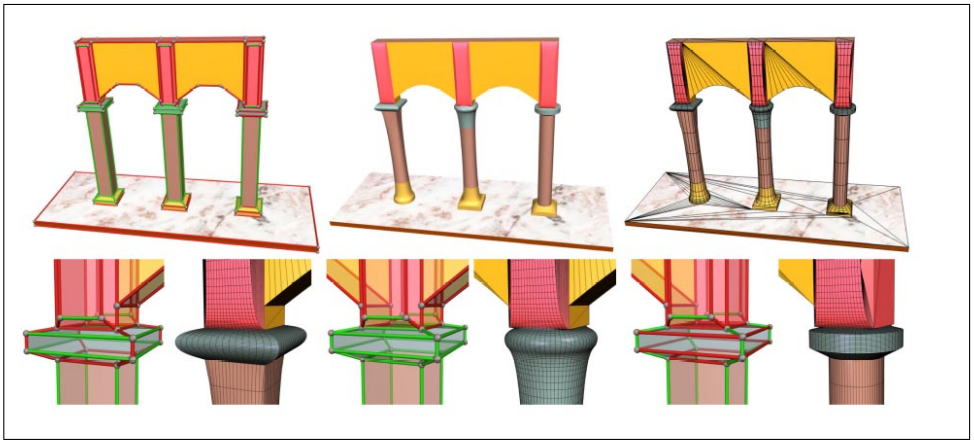


Abbildung 3.4: Combined BRep - Gleiches Kontrollnetz mit unterschiedlichen Kanteneigenschaften [HAV05]

Da die GML sehr einfach gehalten ist, wird sie von ihrem Entwickler nur semantisch mit Hilfe von zwölf Regeln beschrieben. An dieser Stelle soll als Beispiel die erste Regel genannt werden:

GML code consists of individual tokens, separated by stop characters such as white-spaces. (GML Language Rules, Rule 1)

Eine detaillierte Beschreibung aller Regeln findet sich in (Havemann 2005).

3.2.1 Token

Generell werden zwei Arten von Token unterschieden. Es gibt Literale und Ausführbare Anweisungen (Executables). Bei der Ausführung eines GML Programms werden Literale einfach auf den Operanden-Stapel gelegt. Ausführbare Anweisungen werden auf den Execution-Stack gelegt und ausgeführt. Literale sind numerische Werte, Felder und Assoziativfelder. Beispiele für einfache Werte sind:

- 42
- „Apfel“
- (1,0.5,6)

Ein Feld ist eine Zusammenfassung von anderen Datenobjekten beliebigen Typs.
Für Felder existieren eine ganze Reihe von Operatoren. Das folgende Beispiel bildet aus

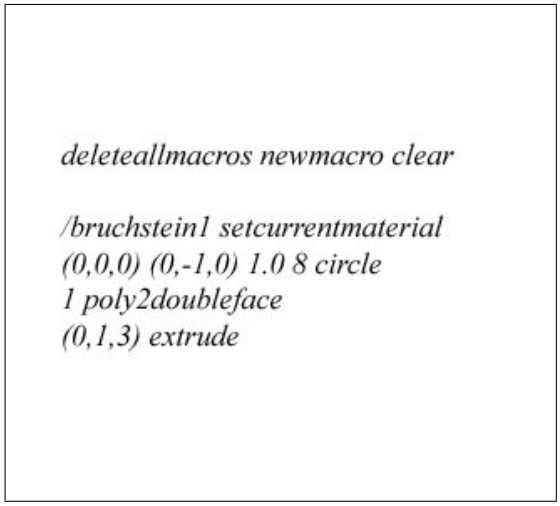


Abbildung 3.5: Einfache GML - Anweisungen

den obersten vier Elementen auf dem Stack ein Feld:

(0,0,5) (5,0,5) (5,5,5) (0,5,5) 4 array

Assoziativfelder (Dictionaries) sind Tabellen mit Objektpaaren. Ein Schlüssel ist jeweils mit einem Wert verknüpft. Der GML-Interpreter schaut im Assoziativfeld nach, ob ein bestimmter Schlüssel vorhanden ist und holt sich dann den dazugehörigen Wert. In Abbildung 3.6 wird ein „Objekt“ *exampleDict* erzeugt und auf dem Dictionary Stack gespeichert.

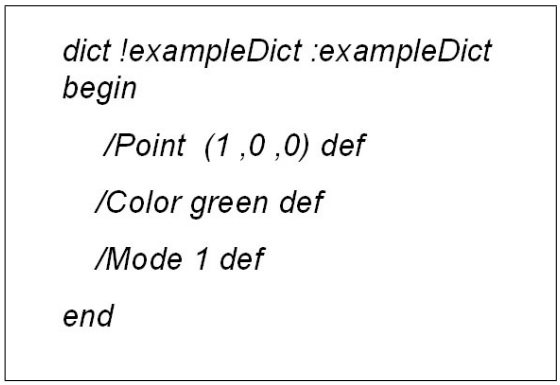


Abbildung 3.6: Assoziativfeld (Dictionary) mit Schlüssel-Wert-Paaren

Assoziativfelder können auch dazu benutzt werden um GML - Programme in objektorientierter Art und Weise zu verfassen. Abbildung 3.7 zeigt ein Objekt mit Variable und Funktion. Der Aufruf von *exampleObject.data* gefolgt von *exampleObject.function* würde zunächst die 12 auf den Stack legen, anschliessend eine weitere 12 auf den Stack legen und dann die beiden obersten Stackelemente vom Stack nehmen und multiplizieren. Das Ergebnis 144 wird nun als oberstes Element auf den Stack gelegt.

```
/exampleObject dict def
exampleObject begin
    /data   12              def
   /function { dup mul }    def
end

exampleObject . data
exampleObject . function
```

Abbildung 3.7: Ein Assoziativfeld als Objekt mit Variable und Funktion

3.2.2 Kontrollstrukturen

Als vollwertige Programmiersprache besitzt GML auch Kontrollstrukturen für Verzweigungen und Schleifen.

repeat{
%führe Anweisung X aus
}5

Diese Schleife führt beispielsweise fünfmal die Anweisung X aus.

3.2.3 Register

Register sind eine zusätzliche Speichermöglichkeit und können jeden beliebigen Datentyp aufnehmen. Sie verhindern ein umständliches Hantieren mit dem Stack um an gewünsch-

te Daten zu gelangen. Sie werden mit dem Schlüsselwort *usereg* aktiviert.

Die Anweisung *(0,1,0) !myVector* speichert den Vektor (0,1,0) im Register *!myVector*.
Die Anweisung *:myVector* legt das Register auf den Stack. Enthielte das Register eine ausführbare Anweisung so würde sie nun direkt ausgeführt werden.
Die Anweisung *;myVector* legt das Register auf den Stack ohne es auszuführen.

3.2.4 Makros

Makros erlauben eine Gruppierung von Euler-Operatoren. Das aktive Makro „protokolliert“ alle Euler-Aufrufe. Abbildung 3.8 zeigt typische Makro - Anweisungen.

```
%löscht alle Eulermakros des Modells
deleteallemacros

%erzeugt ein neues leeres Makro
newmacro

% löscht ein Makro und macht Änderungen
% am Modell rückgängig
deletemacro
```

Abbildung 3.8: GML Makro - Anweisungen

Makros werden eingesetzt, um Teile eines Modells zu modifizieren. Dabei bleibt der Rest des Modells unbeeinflusst.

3.3 Beispiele

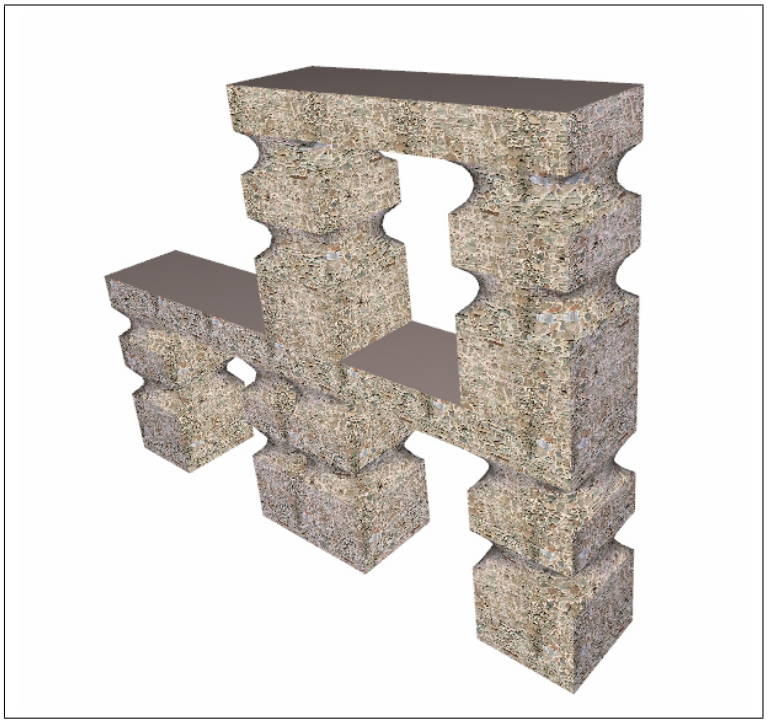


Abbildung 3.9: Beispiel eines GML - Modells

Am Beispiel des Bogenganges aus Abbildung 3.9 soll ein kurzer Einstieg in GML gegeben werden.
Ausgenommen den praktrischen Beispielen aus diesem letzten Abschnitt und den Abschnitten zuvor stammen die Informationen dieses Kapitels aus der Dissertation von Herrn Dr.-Ing. Sven Havemann[4], die auch im Internet zu finden ist[5].

[4](Havemann 2005)
[5]http://www.digibib.tu-bs.de/?docid=00000008

```
deleteallmacros newmacro clear

(0,0,5) (5,0,5)  (5,5,5) (0,5,5)
4 array
1 poly2doubleface
```

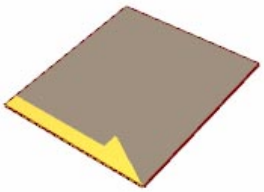


Abbildung 3.10: Anlegen eines neues Makros und erstellen einer Fläche aus einem Feld mit 4 Elementen. Ein Pfeil markiert die aktuelle Kante.

```
/bruchstein1 setcurrentmaterial

(0,4,3) extrude
(0,2,1) extrude
(0,2,3) extrude
(0,2,1) extrude
(0,2,3) extrude
```

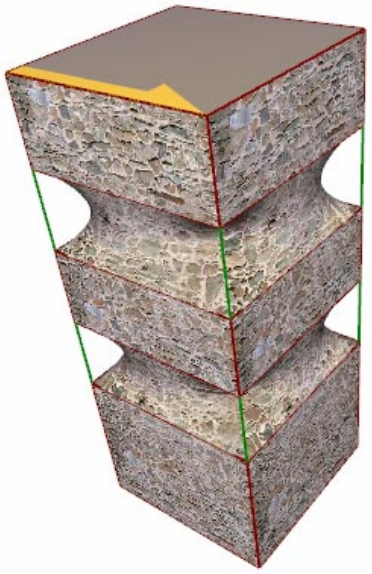


Abbildung 3.11: Aus der Fläche wird eine Säule mit Textur „bruchstein1“.

```
faceCCW
vertexCW
(0,7,3) extrude
(0,5,3) extrude
faceCCW
vertexCW
(0,2,1) extrude
(0,2,3) extrude
(0,2,1) extrude
(0,2,3) extrude
(0,2,3) extrude
```

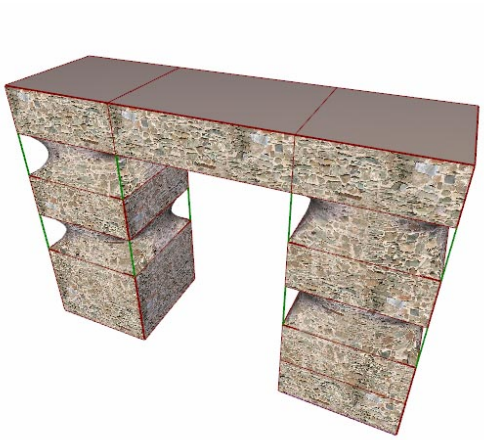


Abbildung 3.12: Durch neue Wahl der jeweils aktuellen Kante wird ein Tor gebaut.

```
/drawArchway
{
...
}def

2{
drawArchway
}repeat
```

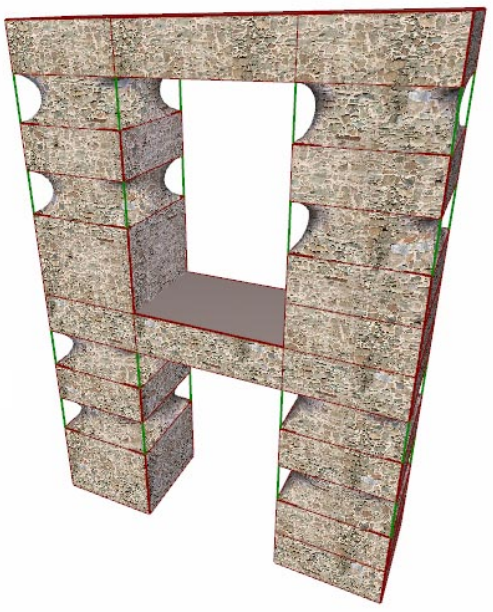


Abbildung 3.13: Der Torbogen als Funktion und zweimal aufgerufen.

Kapitel 4

Interaktion

Interaktion steht für die Möglichkeiten des Anwenders mit dem Programm zu interagieren und die Daten und damit das Modell zu modifizieren. Dies bedeutet im Fall von GML, dass nach einer Aktion des Nutzers der Programmcode interaktiv generiert/ verändert werden muss. Als wiedererkennbare Werkzeuge zur Interaktion benutzt GML sogenannte *Gizmos* .

4.1 Gizmos

Ein *Gizmo* ist ein Symbol für eine Interaktion mit dem Modell. Es ist selber Teil der Szene, mit der es interagiert. Meist ist ein *Gizmo* an ein bestimmtes Objekt gebunden, dass es modifizieren soll.

Abbildung 4.1: Linear - Slider Gizmo

Abbildung 4.1 zeigt ein Beispiel eines Gizmos, dass die Länge einer Mauer vergrößert oder verkleinert. Ein wichtiger Aspekt ist der Wiedererkennungswert eines solchen Werk-

zeugs beim Anwender. Trifft er in einer Szene auf ein „Pfeil - Gizmo“ so wird er die Wirkung dieses Pfeils direkt absehen können.

Wie schon in Kapitel 2.2 und Abbildung 4.2 gezeigt, können viele Objekte eine gemeinsame Struktur haben. *Gizmos* werden benutzt, um aus einem Objekt ein anderes zu erzeugen. Nur eine Repräsentation der Struktur muss gespeichert werden.

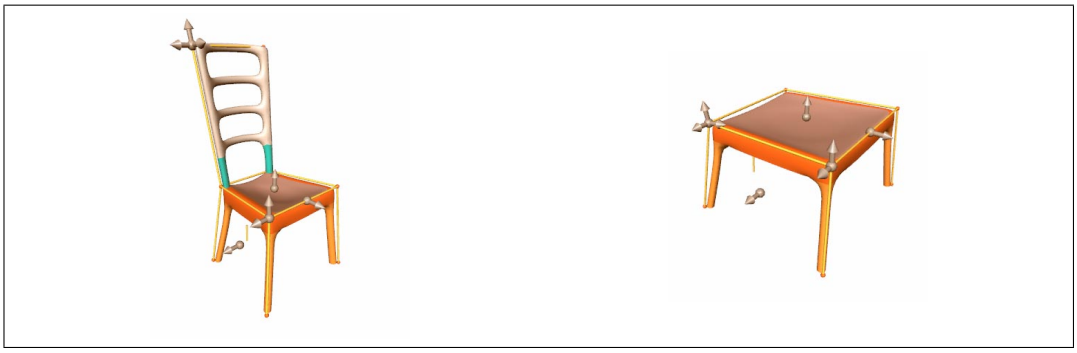


Abbildung 4.2: (a) Stuhl mit Gizmos (b) Tisch mit Gizmos

Kapitel 5

Die Burg im Kondertal

Nachdem in den bisherigen Abschnitten die Technik beschrieben wurde, soll in diesem Teil der Arbeit die praktische Umsetzung gezeigt werden. Das historische Hintergrundwissen lieferte eine Magisterarbeit des Instituts für Kunstwissenschaft der Universität Koblenz[1]. Wie bereits in Kapitel 1 erwähnt, erstreckte sich die Burganlage über den gesamten Nordwestausläufer des Hinterberges im Kondertal und diente hauptsächlich der Überwachung von wichtigen Transportwegen und Bergwerken.

5.1 Geschichte

Die Entstehungszeit der Burg lässt sich in etwa auf das frühe 12. Jahrhundert festlegen. Dies belegen zum einen Keramikfunde, die auf diese Zeit datiert wurden und zum anderen auch Eigenschaften der Architektur. So entstand in dieser Zeit beispielsweise der Typus des befestigten Wohnturmes und die Wohngebäude selber wurden in einer Größe errichtet, die auf Fernwirkung ausgerichtet war.
Während der ersten Hälfte des 12. Jahrhunderts gehörte das Kondertal bereits zum Besitz des Bistums von Trier. Dies wurde 1018 in einer Schenkungsurkunde Kaiser Heinrichs II. an Erzbischof Poppo von Trier festgelegt, die den gesamten Koblenzer Königshof in den Besitz des Erzbischofs übergehen lies. Lediglich das Recht des Bergbaus wird in der Urkunde nicht erwähnt, was darauf schließen lässt, dass der Kaiser sich dieses Recht selber vorbehielt. Dessen Verwalter, der Pfalzgraf, hatte die Interessen des Kai-

[1](Markus 2007)

sers zu wahren und musste neben dem Bergbau unter anderem auch die Handelsstraßen beaufsichtigen. Die Burgen der Pfalzgrafen liegen nun jeweils auf Bergen, von denen man wichtige Straßen beherrschen kann[2]. Besonders an den Moselübergängen hatten die Pfalzgrafen Aufsichtsrechte, so an den Fähren bei Hatzenport und Burgen sowie an Furten, wie zum Beispiel in Treis, in Alken und eventuell bei der Burg im Kondertal. Die drei zuletzt genannten Burgen weisen in Topographie, Zeitstellung und Baubefund auffällige Gemeinsamkeiten auf[3]. Der Schutz der Erzmienen und deren Siedlung innerhalb einer kaiserlichen Enklave sprechen für eine Burggründung vor 1150, um die kaiserlichen Interessen zu wahren, denn erst 1158 verlieh Kaiser Friedrich I. „alle Bergwerksrechte innerhalb des gesamten erzbischöflichen Besitzes“[4] dem Kurfürsten von Trier.

5.2 Architektur

Zunächst wurde in Anlehnung an die archäologischen Untersuchungen ein Grundriss erstellt. Abbildung 5.1 zeigt die Umfassungsmauer der Burg, die mit einem Gizmo in Höhe und Breite modifiziert werden kann.

Abbildung 5.1: Burgmauern als Grundriss: Ein Slider kann Höhe und Breite der Mauer variieren.

Der Grundriss wurde aufgrund einiger Probleme mit der GML vereinfacht dargestellt (siehe Abbildung 1.1).
Anschliessend wurde nach und nach die übrige Anlage hinzugefügt. Im Nordwesten der Burganlage befand sich ein Gebäude größeren Umfangs, das wohl das Hauptgebäude darstellte. Es lag auf einem natürlichen Felsplateau und bildete einen Teil der Umfassung. Schon aus weiter Ferne war dieses Gebäude sichtbar, weshalb es wohl einen repräsentativen Charakter gehabt haben muss. Die drei weiteren Gebäude waren vermutlich eine Küche, eine Schmiede sowie ein kleineres Wohnhaus. Ihre Positionen können durch

[2](Gerstner 1941)
[3](Markus 2007)
[4](Michel 1963)

entsprechende Gizmos verändert werden. Der Turm kann zusätzlich noch in der Höhe modifiziert werden. Einen Brunnen besaß die Burg nicht und das Wasser sowie alle übrigen lebensnotwendigen Dinge mussten über einen schmalen Pfad herauf getragen werden.

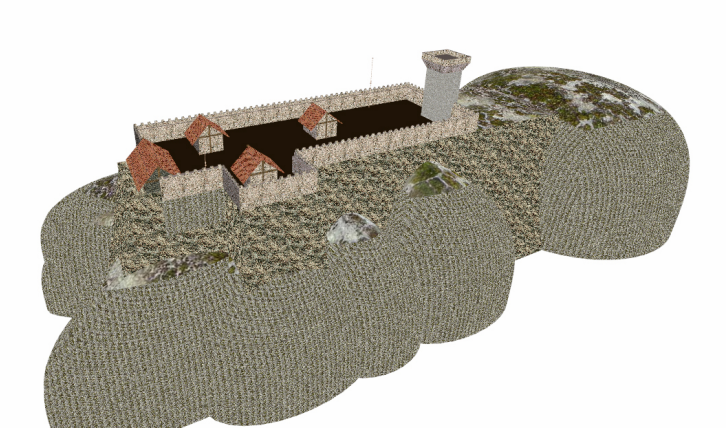


Abbildung 5.2: Die Burg im Gelände

5.3 Details der Programmierung

Nachfolgend werden einige zentrale Stellen des Programm-Codes näher erläutert. Zunächst sollen die Gizmos betrachtet werden. Wie jedes Gizmo benötigt das *Linear - Slider - Gizmo* sogenannte *Callback-Funktionen*, die die Benutzereingaben auswerten. In diesem Fall sind dies *mouse events*, bei denen drei Fälle unterschieden werden können: *click*, *drag* und *release*. Abbildung 5.3 zeigt den GML - Code, der das Verhalten des Sliders bestimmt.

Die äußeren *if-statements* prüfen, welches Ereignis ausgelöst wurde (*click*, *drag* oder *release*). Im ersten Fall (Zeilen 1-9) wird die aktuelle Position des Sliders gespeichert. Während einem *mouse drag* wird diese dann neu berechnet (Zeilen 10-22). Der Operator *intersectSTPQ_2line* berechnet den dem Slider am nächsten liegenden Punkt mit Hilfe der Achse des Sliders und dem Strahl, den der Mausklick verursacht hat. Da der Operator drei Ergebnisse liefert jedoch nur eins benötigt wird, werden die übrigen einfach vom Stack entfernt. Das Loslassen der Maustaste schliesslich fixiert die neue Position des Sliders (Zeilen 23-30). Die *Callback-Funktionen* werden im Register *!linearSliderCallbackFkt* gespeichert.

```
1: {
2:    mouseEvent 1 eq
3:    {
4:     begin
5:     /hitpointOffset hitpoint currentPosition sub def
6:     /isActive 1 def
7:     updateMovingBNSColor
8:     end
9:    } if
10:   mouseEvent 2 eq
11:   {
12:    begin
13:    pNear
14:    pFar
15:    startPosition hitpointOffset add
16:    endPosition hitpointOffset add
17:    intersectSTPQ_2line pop pop /t edef pop
18:    t 0 lt { /t 0 def } if
19:    t 1 gt { /t 1 def } if
20:    end
21:    execute
22:   } if
23: mouseEvent 3 eq
24:   {
25:    begin
26:    /isActive 0 def
27:    updateMovingBNSColor
28:    end
29:   } if
30: } !linearSliderCallbackFkt
```

Abbildung 5.3: Callback Funktionen des Linear - Slider

Abbildung 5.4 zeigt die weitere Anwendung eines *Slider - Gizmo*. Die Zeilen 2-4 definieren den Slider */linearSlider_WallHeight*, der in den Zeilen 6-9 eine Position, sowie einen Maximal-/ Minimalwert zugewiesen bekommt. Danach wird er „aktiviert“ (Zeile 10) und das mit der „perform - Methode“ berechnete Ergebnis der Veränderung wird im Register *!wallHeight* gespeichert (Zeile 11). Im letzten Teil des Codeauszugs sieht man dann die Verwendung dieses Registers.

Die Benutzeroberfläche beschränkt sich momentan noch auf einen OpenGL - Anzeige - Frame und einen Bereich der für weitere GUI - Elemente angelegt wurde. In den Kapiteln 6 und 7 soll näher erläutert werden, weshalb noch nicht weitere Elemente zur Handhabung des Programms im *Control - Panel* erstellt wurden.
Der Benutzer ist in der Lage das gesamte Modell zu drehen und in beliebiger Weise zu

```
1:   %  LinearSlider for height of walls
2:    0.0
3:    linearSlider
4:    /linearSlider_WallHeight edef
5:    ....
6:   10
7:    1
8:    (10,0,15)
9:    (10,0,5)
10:   linearSlider_WallHeight begin perform end
11:   !wallHeight
12:  ....
13:   % wallFront
14:   (-5,4,0) dup getX 5 add putX !blf
15:   (-5,4,0) dup getX 20 add putX !brf
16:   (-5,4,0) dup getX 20 add putX dup getZ :wallHeight add putZ !trf
17:   (-5,4,0) dup getX 5 add putX dup getZ :wallHeight add putZ !tlf
```

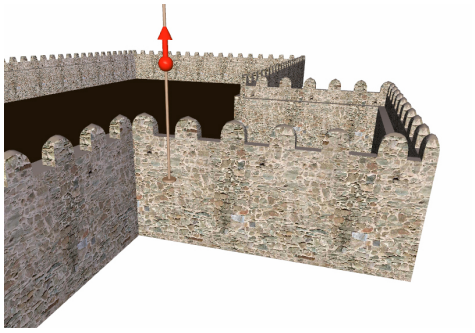


Abbildung 5.4: Code für den *Linear - Slider* und Ergebnis im Programm

verschieben. Auch das Ein- und Auszoomen ist möglich. Dazu wurden dem OpenGL - Frame *Event - Handler* zur Verfügung gestellt. Abbildung 5.6 zeigt das Control Panel und den OpenGL - Frame.
Es folgen einige Beispielabbildungen des Programms zum jetzigen Zeitpunkt.
Die Auswertung der Ergebnisse erfolgt anschließend in den Kapiteln 6 und 7.

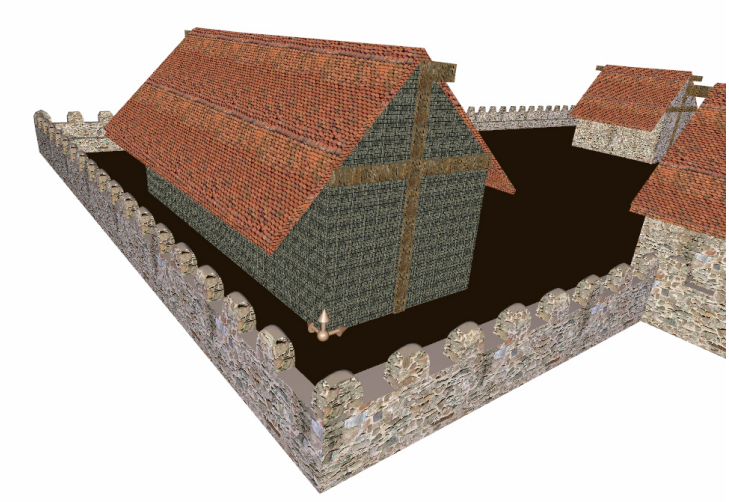


Abbildung 5.5: Haus mit „Positions-Gizmo“

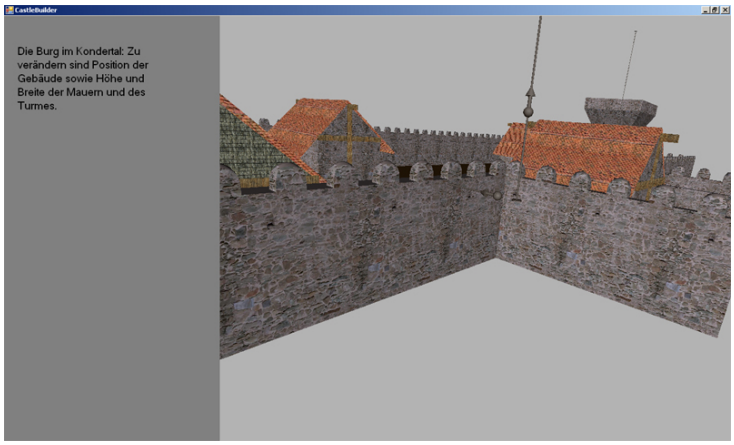


Abbildung 5.6: Control Panel und OpenGL - Frame

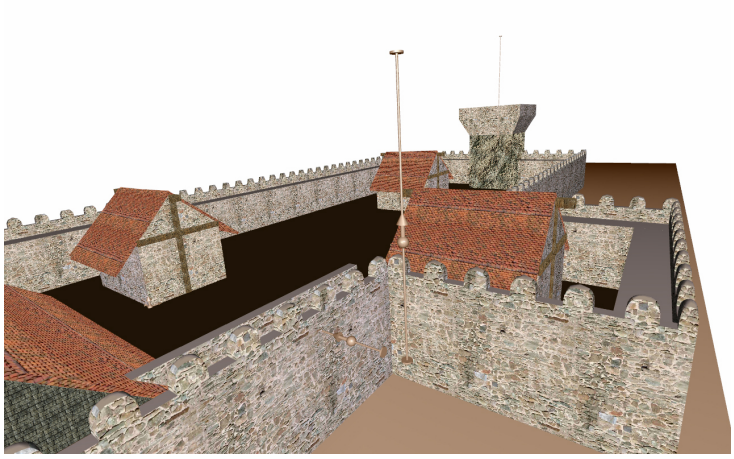


Abbildung 5.7: Beispiel: Burg mit Gebäuden und Turm

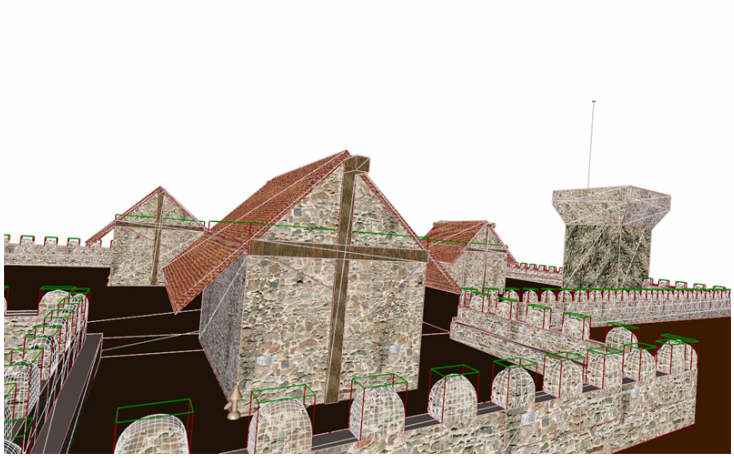


Abbildung 5.8: Beispiel: Control-Mesh und Tesselierung

Kapitel 6

Verbesserungsmöglichkeiten

Zum jetzigen Zeitpunkt sind sicher noch nicht alle Möglichkeiten, die die *GML* im Kontext der Aufgabenstellung bietet, ausgenutzt. Als turing-mächtige Sprache kann sie theoretisch alle denkbaren Anforderungen erfüllen.
Die Modelle der Gebäude wirken zur Zeit noch etwas behelfsmäßig, bedingt durch die zugrunde liegende Datenstruktur. Diese wurde gewählt, um die *Gizmos* nutzen zu können. Zahlreiche Versuche die Daten in anderer Weise bereit zu stellen endeten stets in Fehlermeldungen
Dadurch musste jedes Gebäude einzeln modelliert werden, was den Ideen der generativen Modellierung eigentlich widerspricht. Auch eine Kommunikation zwischen *Control - Panel* und *OpenGL - Frame* scheiterte aus diesem Grund. So kann das *Control - Panel* momentan nur Hinweise zur Nutzung des Programms bieten, aber keine Werkzeuge anbieten.
Eine weitere Ergänzungsmöglichkeit wäre das Einbinden weiterer *Gizmos*, beispielsweise zur Korrektur des Mauerverlaufs oder zur Wahl verschiedener Gebäudetypen. Als Vorbild kann hier das *Castle Construction Kit* [1] dienen. Dieses arbeitet mit einem Szenegraph im Hintergrund und bietet viele der für diese Arbeit wünschenswerten Funktionen an. Abbildung 6.1 beispielsweise zeigt den „Mauer - Editor“ des Programms. Leider stand lediglich eine abstrakte Beschreibung und nicht der Code des *Castle Construction Kit* zur Verfügung.
Das Benutzen der *GMLMFC*[2], einer speziellen Entwicklungsumgebung für *GML* würde wahrscheinlich die besten Ergebnisse für ein Projekt wie die generative Modellierung der

[1] (Havemann u. a.)
[2] http://www.generative-modeling.org/GenerativeModeling/software

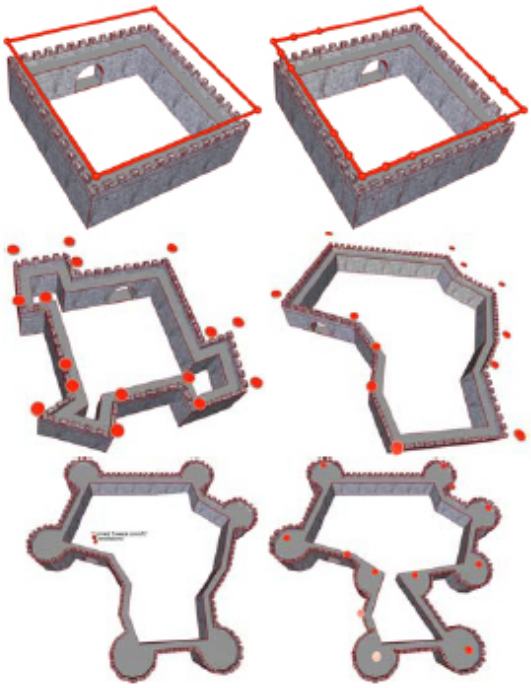


Abbildung 6.1: „Mauer - Editor“ des *Castle Construction Kit* (Havemann u. a.)

Burg im Kondertal liefern. Sie bringt neben einer Anzeige des aktuellen Stackinhalts auch eine Exportfunktion in das *.obj - Format* mit. Dies hieße jedoch auf eine ansprechende Benutzeroberfläche zu verzichten. Denkbar wäre es jedoch das so gewonnene Modell in eine Landschaft einzubinden, die mit einem dafür geeigneten Tool erstellt wurde. Die Eigenarten des Kondertals (siehe Abbildung 6.2) könnten damit besser hervorgehoben werden und die Wahl der Lage der Burg wäre offensichtlicher.

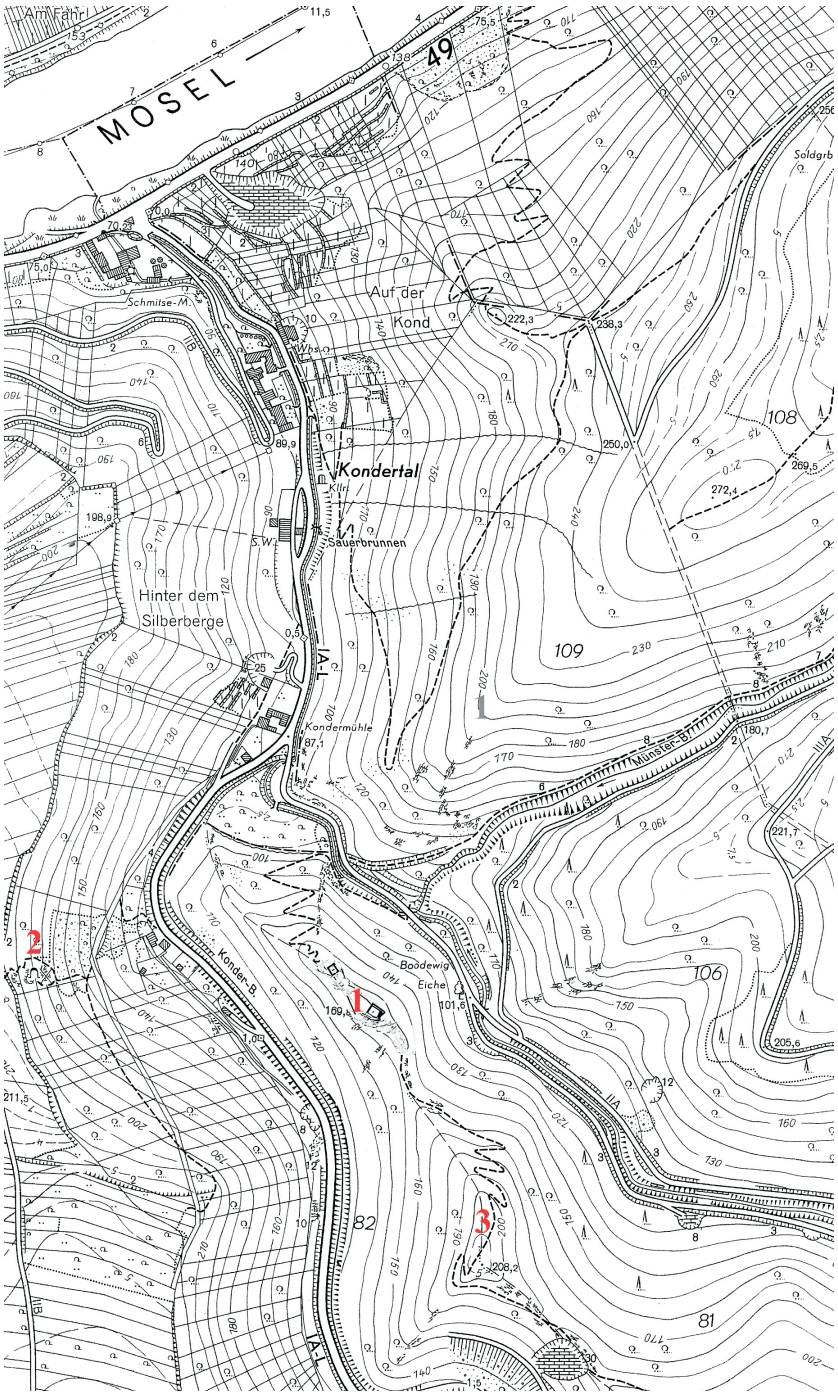


Abbildung 6.2: Das Kondertal und die Lage der Burg (Markus 2007)

Kapitel 7

Persönliche Einschätzung und Fazit

Die generative Modellierung ist eine Technik, die in der Zukunft sicherlich große Beachtung erfahren wird. Ihr innovativer Umgang mit großen 3D - Modellen kann in vielen Bereichen von großem Nutzen sein. Mit der für diese Arbeit benutzten *Generative Modeling Language* stand ein sehr mächtiges Werkzeug zur Verfügung. Leider stellte es sich als sehr schwierig für mich heraus die Sprache selbstständig zu erarbeiten. Dies lag rückblickend wohl unter anderem daran, dass ich den GML - Code von Anfang an mit Hilfe des *GML - Framework* für C++ ausgeführt habe. Dort gibt es scheinbar noch einige zusätzlich mögliche Fehlerquellen, die auch nach intensivem Suchen nicht gefunden bzw. behoben werden konnten. Zunächst sollte man demnach das GML - Modell in einer geeigneten Umgebung, beispielsweise der *GMLMFC* (siehe Abbildung 7.1), entwickeln und erst im Endstadium die Anbindung an eine graphische Oberläche in Angriff nehmen.

Die Beschäftigung mit einer aktuellen Idee der Computergraphik war zu jedem Zeitpunkt motivierend und trotz der nicht optimalen Ergebnisse konnte ich viele praktische Erfahrungen in gleich mehreren Bereichen sammeln:

- Zum Einbinden des mit C++ entwickelten *GML - Framework* in C# war es nötig, die Module mit Hilfe von *SWIG (Simplified Wrapper and Interface Generator)*[1] zu wrappen.
- Da das Programm unter C# entstand, fand ich einen Einstieg in diese aktuelle Programmiersprache, die in gewisser Weise eine Mischung aus C++ und Java darstellt und mittlerweile in vielen Bereichen verwendet wird. Auch in das Erstellen einer

[1](swi)

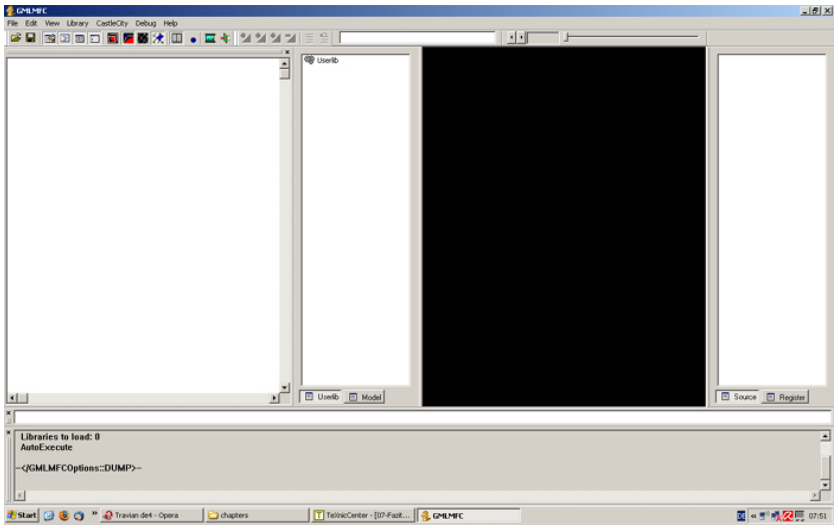


Abbildung 7.1: GMLMFC

graphischen Oberfläche mit C# und dem *.NET - Framework* konnte ich mich einarbeiten, auch wenn diese im Endergebnis des Programms nicht sehr bedeutend ausfällt.

- Die Idee der generativen Modellierung gab einen interessanten Einblick in ein aktuelles Thema der Computergraphik.
- Mit der *Generative Modeling Language* lernte ich den Umgang mit einer stackbasierten Programmiersprache.

Zum jetzigen Zeitpunkt entsprechen die Ergebnisse der Arbeit nicht den Vorstellungen, die ich zu Beginn hatte. Die Idee die Burg interaktiv zu modellieren ist zu erkennen, jedoch sind momentan die Möglichkeiten der Interaktion noch zu gering und das Modell an sich ist noch zu einfach gehalten (zu den Gründen siehe auch Kapitel 6). Die Verbindung von historischer Forschung und Informatik bietet aber noch eine Vielzahl von Möglichkeiten und wird für mich persönlich auch weiterhin von Interesse bleiben.

Literaturverzeichnis

[pos] *Adobe Postscript*. http://www.adobe.com/

[swi] *SWIG*. http://www.swig.org

[TAO] *The Tao - Framework*. http://taoframework.com/

[Gerstner 1941] GERSTNER, Ruth ; (Hrsg.): *Die Geschichte der lothringischen und rheinischen Pfalzgrafschaft von ihren Anfängen bis zur Ausbildung des Kurterritoriums Pfalz*. Rheinisches Archiv 40, 1941

[Havemann 2005] HAVEMANN, Dr.-Ing. S.: *Generative Mesh Modeling*, Technische Universität Braunschweig, Diss., 2005

[Havemann u. a.] HAVEMANN, Sven ; BERNDT, Rene ; GERTH, Björn ; FELLNER, Dieter W.: *Castle Construction Kit*. – 3D Modeling for Non-Expert Users (2005)

[Janssen 1975] JANSSEN, W.: *Studien zur Wüstungsfrage im fränkischen Altsiedelland zwischen Rhein, Mosel und Eifelnordrand*. Köln Rheinland-Verlag, 1975

[Markus 2007] MARKUS, Yvonne. *Die mittelalterliche Besiedelung im Rhein-Mosel-Dreieck*. Magisterarbeit. 2007

[Michel 1963] MICHEL, F. ; (Hrsg.): *Die Geschichte der Stadt Koblenz im Mittelalter*. Trautheim, 1963