Procedural Character Generation

Implementing Reference Fitting and Principal Components Analysis

Spring 2007

Aalborg University

Aalborg Universitet
Department of Computer Science, Frederik Bajers Vej 7E, DK 9220 Aalborg Øst

Title: Procedural Character Generation: Implementing Reference Fitting and Principal Components Analysis

Abstract:

This report examines the area of procedural generated content, more specifically procedural generation of humanoid characters. The work presented extends upon an analysis of which methods are suitable for generating such models. The analysis suggest that procedurally generating humanoids is best done using principal components analysis. Only the generation of the surface mesh is considered in the analysis i.e. no texture data is generated. Based on the analysis we present a design for an application which can be used by graphics artist in association with their modeling editor. The application utilizes principal components analysis, reference fitting and genetic algorithms to provide an easy method for generating models. Based on the application implemented we conclude that it is possible to generate humanoid models using principal components analysis, however there is room for improvement. The test carried out do little to show the degree of variation which can be achieved in the generation process. Likewise the method developed for making the model generation intuitive is not successful, given the small amount of test data.

Project period:
February. 1st 2007 - June. 8th 2007

Project group:
d628a

Group members:
Bach, Esben
Madsen, Andreas

Supervisors:
B. Madsen, Claus
Bangsø, Olav

Copies: 6

Page count: 91

Preface

We would like to thank Camilla Volkersen, a professional graphics artist at Pollux Gamelabs [Vol], for her assistance in clarifying modeling concepts, and providing valuable information on the process of character modeling in a game context. Also thanks goes to our supervisors Claus B. Madsen and Olav Bangsø for valuable input throughout the project.

Table of Contents

1 Introduction

The problem of creating enough content for computer games has been described by Will Wright of EA Games as *The Mountain of Content* problem. This report presents work related to procedurally generating content for computer games, helping to reduce the mountain of content to be created. More specifically the target of attention is humanoid characters. This report is the second part of a master thesis, continuing on the work laid out in [BM07]. *This*, the first, chapter is a general introduction as well as a recap of the previous report, ending with a description of the overall implementation goals. Chapter 2 presents a system definition and describes how a piece of software can be used in the model creation process. Furthermore a class design is presented, further highlighting the important classes, as well as describing how the different components fit together. The third chapter presents an explanation of the important methods utilized in the software, such as reference fitting and principal components analysis. Chapter 4 describes a series of test designed to showcase performance of the software. Chapter 4 also presents a discussion of the results and status of the software. Finishing off the last Chapter concludes on the results, and suggests possible areas of improvement.

1.1 Procedural Content

The term *Procedural Generation* (or procedural synthesis) refers to the process of generating "something" procedurally, i.e. letting an algorithm generate "something" instead of doing it by hand. In the case of video games *Procedural Content Generation* refers to creating graphical content such as levels, characters, props and other visual objects. Traditionally procedural content have been used because computers were limited in both available memory and storage (both hard drives and installation media). Large custom build levels and graphics artwork

could not fit into memory, instead pseudo random number generators were used in order to create these as the game progressed. One of the earliest games [Spu03] to use these techniques extensively was *Elite* by Acornsoft (Now known as Frontier Developments[Bra]). Games have come to rely more and more on custom graphics and sound created by artists, mainly because customized procedural content is difficult to create, and therefore also expensive. However the demo scene [Wik06] has kept up, showing what is possible with procedural generated content. One of the most impressive examples to date is *.kkrieger* [Fab04] developed by the German demo group Fabrausch in 2004. *.kkrieger* is a short game featuring nothing but procedural content, using no more than $96kb$ of disk space (it is however very CPU intensive). Electronic Arts is currently working on a new game called *Spore* [MA] where the entire world is procedurally generated, using only limited amounts of artist generated content.

1.1.1 Procedural Content Today

Procedural content is regaining interest in video games for two main reasons; due to technical capabilities of modern consoles and high-end home PC's, many highly detailed objects can appear in a single scene. Also games can utilize storage media with large capacity (Gigabyte sized) to house larger game worlds and stories. As clones of houses, characters and landscapes do not fit into a realistic environment, more graphics artists are needed to create this varying content, thus increasing development cost. Secondly some technical limits still exists such as the I/O bus speeds, it is hard to keep enough content in memory due to the sheer size of the content data. Reading speeds of media such as hard drives and optical discs is becoming a problem[Wil]. If content can be produced runtime the required I/O traffic load can be reduced by many orders of magnitude. That is, reading a few seed parameters from disc instead of reading 200 megabytes of vertices and textures will free bandwidth for other uses or reduce the system waiting time for I/O interrupts. Since *Elite* it has not become any easier to create customizable procedural content (i.e. procedural content that the artist can decide every little detail on). Consequently procedural content generation has been applied as middleware, using algorithms to create the basic structures and then loading textures and other information created by artists, in effect creating a hybrid between procedural content and artist produced content. Speedtree [Vis] is one such software package to create hybrid content. Trees are procedurally generated but leaf textures can be loaded from disc, enabling artist to use pictures of real foliage. Another approach utilized in *Spore*, is to create tools where algorithms help generate the basic structure of models, and details added later [Max]. This does not solve any problems with regards to I/O, however it can help reduce the time needed by artists to create all the required content.

The work presented here and in [BM07] focuses on finding one or more methods which can be used to reduce the production time when creating simple hu-

manoid characters. The overall goal is to end up with an application which can let artists create large amounts of models easily and by specifying simple intuitive parameters.

1.2 Modeling Process

The following presents a short motivating description of the steps involved in modeling characters for a video game.

In general character modeling can be broken into a series of steps:

1. Creating and importing 2D Sketches of the character
2. Building a surface mesh
3. Building and inserting a skeleton hierarchy (rigging)
4. Binding together mesh and skeleton hierarchy (skinning)
5. Adding textures and displacement maps to the mesh
6. Creating animations
7. Creating level of detail models

The process of creating characters often starts with 2D sketches to provide a look and feel of the character. Such sketches can be used in the creation of the model, assisting in building the surface mesh. Building a surface mesh for low polygon characters is most commonly done using a method called box modeling. As the name suggests the idea is to form the character basics with boxes, and hereafter modifying these boxes to form the final character. The idea with 2D sketches is that during the last phase of the mesh building, the vertices can be moved such that they align with the lines on the scanned 2D drawings.

If the mesh should be easy to animate, it needs a skeleton. To make certain that the animations behave correctly, the skeleton should have a carefully designed hierarchy. The effect of such a hierarchy is that when one bone is transformed, all its children are also transformed. The structure of the hierarchy is anatomically dependent. Another important aspect is the constraints that should be in place in order to adhere to common anatomical constraints, such as an elbow not bending in the wrong direction. Upon finishing the skeleton it needs to be attached to the surface mesh. This process, known as skinning, is needed in order to determine which vertices should be transformed when the bones are transformed, and just as important how much they should be transformed. Additionally 2D colour maps known as textures are required to give the model details such as skin colour, eyes and simple clothes.

The point of creating a skeleton in the model, is so that the model can be animated. Each animation sequence is a series of movements of bones over a period of time, normally measured in frames. A character model often has several sequences such as; walking, running, jumping, crouching and others specifically needed for the characters. All this work should possibly be repeated to create lower detailed models in order to reduce the strain on the graphics adapter. The idea is that the low detail models can be used when the object moves away from the camera.

A modeling assistant tool could be used in order to reduce the time of model creation. Several approaches can be taken when creating such a system. Among these is the approach of letting the user choose a series of parameters and then via some procedural method, generate a number of models. It has the advantage of being completely automatic once started, but likely to be nontransparent and hard to use. Other approaches includes automating subtasks, visual representation of parameters in real time, a user guided system or combinations of these. In general a compromise between control, transparency and automation should be reached which allows the artist to form the output (control), while knowing what each parameter does (transparency and intuitiveness) and with as little work as possible (automation). Also an important aspect is the reuse of simple animations such as walk, run and jump.

1.3 Procedural Methods

There are several methods which might be suitable for creating procedural content. In the previous part of this project [BM07], two approaches were analyzed focusing on different ideas. One approach is based on analyzing a large set of examples, finding common information between the models. The other approach is based on the idea of constructing models using a series of construction rules and object primitives. The following is a short description of these two approaches and the issues that arise as a consequence of choosing either method.

1.3.1 Principal Components Analysis

Principal Components Analysis is a statistical process, and have been used successfully to model facial expressions [BV03, BV99] and humanoid models [SMT04, SMT03a, BV99] using example data. The idea is to isolate the components that best describes a data set and then relying on the user to specify these components. One requirement of principal component analysis is that the data must be normalized, i.e. in the case of a model they should all have the same number of vertices, and each vertex should map to the same abstract physical attribute. A limitation of this method is that it only captures the most dominant features of the example data, so if a single model has huge ears, this feature is likely to

disappear if the example dataset is large. Furthermore the parameters, which the user needs to specify, are unintuitive.

In general principal components analysis provides a high degree of automation, and with some additional method, to make parameters intuitive, it also provides transparent control. However it requires a fair amount of initial work.

1.3.2 Reference Fitting

The principal components analysis requires normalized models, thus reference fitting is introduced. Reference fitting can be used to normalize example data. The idea is to transform an instance of some reference model so that it resembles an example model. The process contains two steps:

1. Rough Skeleton Fitting
2. Fine Mesh fitting
    (a) Vertex projection
    (b) Mesh relaxation

The first part relies on so called skeleton driven deformation and assumes the reference model is skinned. Each bone is transformed(1) so that landmark locations, of the reference model, best matches the example model. The second step is to make a fine adjustment(2) of the vertices. This step is divided into two steps; each vertex is projected to nearest vertex or plane(a). Hereafter the mesh are relaxed(b), i.e. the deformation in the projected mesh is minimized.

1.3.3 Genetic Algorithms

In the context of procedural character generation, genetic algorithms can have at least two uses, one is to avoid the normalization problem for principal components analysis. Secondly it might be used to construct entire models on its own. Genetic algorithms rely on a fitness function and as such it cannot be completely automated since the artist needs control. To alleviate the problem of creating a fitness function, the user can be prompted to choose optimal solutions for each generation of the algorithm. Genetic Algorithms can be used in multiple scenarios and ways. Firstly it can be used prior to principal components analysis to create example models, to help avoid the normalization issue. Secondly it can be used after the component analysis, to modify and specialize models, alleviating the problem that only dominant features are present on the output. Lastly it can also be used without the component analysis in order to generate models and then modify them to achieve individual character traits.

The heart and soul of genetic algorithms is the mutation operations. Several schemes can be used with varying effect.

Single Vertex Mutation A model can be considered as a collection of vertices, encoding each vertex into the chromosomes opens the possibility of affecting each vertex separately. Such single vertex mutation results in a highly flexible method, however it gives little control to the artist. Furthermore it is not automatic, and it takes a long time to converge to a proper result.

General Mutation Operations A combination of a selection scheme and a mutation action can be used to perform more abstract and general operations on a vertex collection. An example of a selection scheme could be to select all vertices included in a sphere with center v and radius r, where v is a random selected vertex from the model. The actual mutation operation could be to move all selected vertices away from the center of the sphere, giving the effect of a spherical structure appearing on the model. This approach can be designed such as to give the user more control than with single vertex mutation. However convergence is still low and if the user is not familiar with genetic algorithms, it is likely to be no better than single vertex mutation.

Combining Predefined Objects Another approach is to consider models, not as a collection of vertices, but as a hierarchy of predefined objects. Mutation can be performed on the individual objects, and these can in turn be placed in the hierarchy according to a set of rules that defines the objects legal placements in the hierarchy. To mutate the objects simple landmarks can be defined to adjust size and form. It requires some work to define these objects, their landmarks and the allowed hierarchy. However it can increase artist control and possibly ensure faster convergence. Additionally if the artist can specify simple fitness functions for the small objects which should be used, the process can be made completely automatic. The disadvantage is that it is hard to generate normalized models using this approach.

1.4 Summary and Goals

The goal is to design and implement an application which can be used for creating humanoid models. It seems that by adopting principal components analysis, it is possible to assist artists in the process of creating simple character models. In short a few general goals should be reached to provide a useful model generation tool:

Easy mesh generation The mesh of a model should be easily generated, requiring less work than manual modeling using a 3D modeling editor.

Possibility of mass generation It should be possible to mass produce varried models.

Reusable animations As many models are produced, simple animations should be resuable for most of the models.

Using principal components analysis, to solve some of these problems, presents some new technical obstacles which should be overcome. The following is a list of general goals important if an application should be useful for creating models

Reference Fitting Implementing a reference fitting method is essential in the reuse of existing models for the example database.

Format export facilities A tool relying entirely on its own format description is hard pressed when it come to an actual use. As such the tool should provide export facilities to known formats, or save data in an open format like VRML or similar.

Automatic PCA The PCA should be automatic, requiring little to no input from the user. This includes a robust component selection method.

Database persistence The application must facilitate reuse of a trained PCA databases between sessions. This requires the possibility to save and load trained PCA data.

Interactive model generation Interaction with the PCA should be done visually, for instance via the use of sliders to adjust parameters and generating new models.

Intuitive parameter adjustment As principal components are unintuitive, a method should exist for mapping these unintuitive values to more intuitive parameters.

Batch model generation For mass and large scale production batch execution is needed. That is, a method for specifying ranges of values used for generating models.

2

Software Analysis & Design

This chapter presents a description of the system definition and requirements, along with possible usage scenarios. Based on these a class and component design is described.

2.1 System Definition and Requirements

The following is a description which defines a system, capable of using principal components analysis for creating models, and its general requirements. The section covers the problem domain, the primary objects and possible usage scenarios such an application should support.

2.1.1 Problem Domain

Game development often requires many 3D models, typically divided into different classes of importance and detail; player characters and main plot centric characters, minor plot characters, and lastly characters which make the world seem alive and populated. The last class has mixed importance, without it the game quickly becomes unbelievable, but creating hundreds of detailed models takes time. Instead this group of models often ends up being duplicates with minor variations, such as skin color. This work, even though simplified, still takes time and is manual labor. The goal is to develop a system to aid in the process of creating these, unimportant, characters so that the production costs are low and the resulting group of models is varied and large.

In the first part of this thesis [BM07] it is proposed that:

> Principal Components Analysis can be used in conjunction with Reference Fitting as parts of a tool, reducing production time of low detail character models.

Based on this proposition the primary objective of the system can be formulated as:

> To reduce overall character model generation time, by making reference fitting and principal components analysis, of 3D character models, readily available for graphics artists.

In general the application should support a new work flow for creating simplistic characters. That is the artist should use this application and its provided functionality instead of going through the modeling steps presented in Section 1.2 on page 11. The new work flow is split into two main phases; one a data preparation phase, and two a model synthesis phase. As presented in Figure 2.1 the preparation phase consists of multiple steps; a normalization, an analysis and an annotation mapping. The normalization step relies on an implementation of reference fitting. The analysis is a principal components analysis of a model database. Lastly the annotation mapping step is there to provide intuitive parameters for the synthesis process.

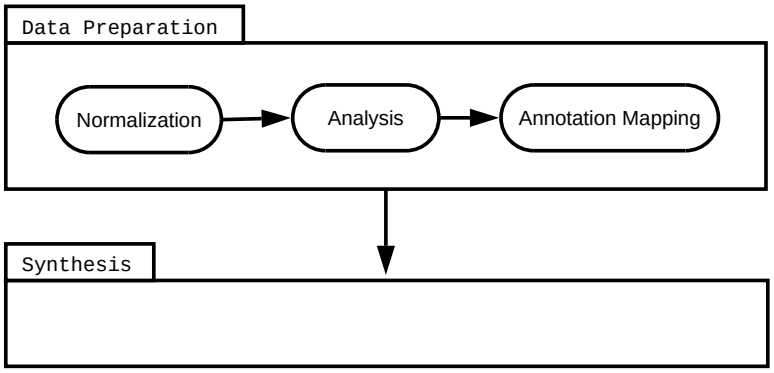


Figure 2.1: The work flow of the preparation phase

Synthesized models should adhere to a common reference model, so that animations can be reused on large sets of characters. In general the application should affect three areas in a game production; the approach to how simplistic models are created, how models are normalized before creation and the amount of time and resources allocated to create such simple models.

2.1.2 Use Cases

The following is a description of the use cases that an application is intended to support. It is desirable that the application supports all required processes; normalization of the models, analysis, mapping and lastly model synthesis i.e. extracting and rendering models. Figure 2.2 on the facing page presents a possible menu layout giving the user the possibility to start anywhere in the process that he or she might desire.

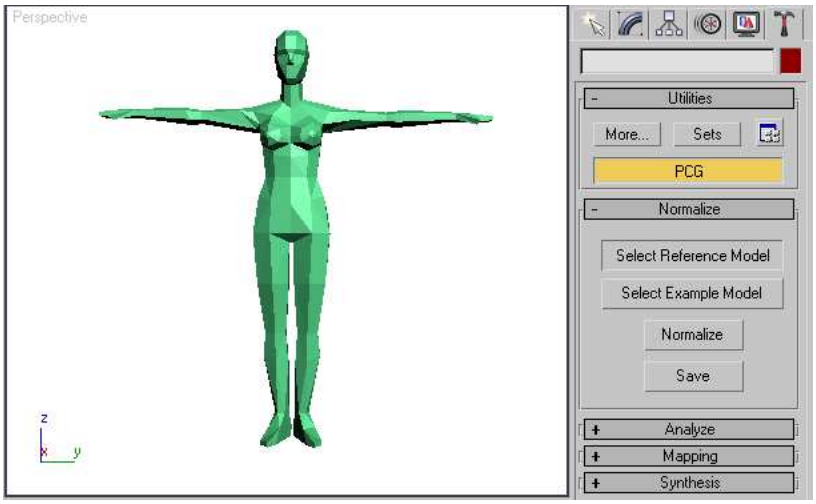
Figure 2.2: Main menu of the implemented 3D Studio Max plugin

Data Preparation

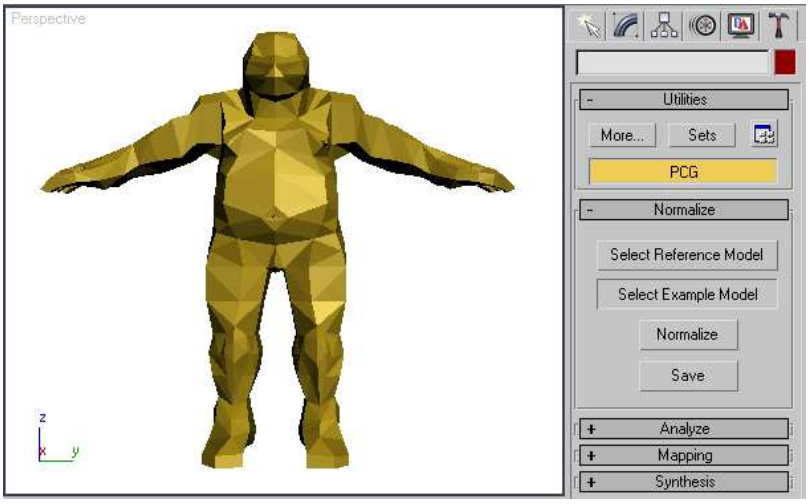
Preparing data is a matter of normalizing all example data, feeding this normalized data to the analyzer component and possibly training an annotation mapping for the synthesis process. In order to provide flexibility the preparation phase should be separated into three general actions; normalization, analysis and annotation mapping, in order to provide flexibility. For instance, if a series of models have been normalized either by the application or by some other means, they can be analyzed without having to run them through the normalizing component.

Normalizing The action of normalizing a model should work by choosing a reference model, and choosing one or more example models to normalize. The result is displayed in the editor with the possibility for manual adjustments, and, the now normalized, model can be saved. Figure 2.3 on the next page shows how this process can work in 3D Studio Max. Having the possibility to choose a reference model provides flexibility to the application, as it can be hard to fit a human reference model to any class of models.

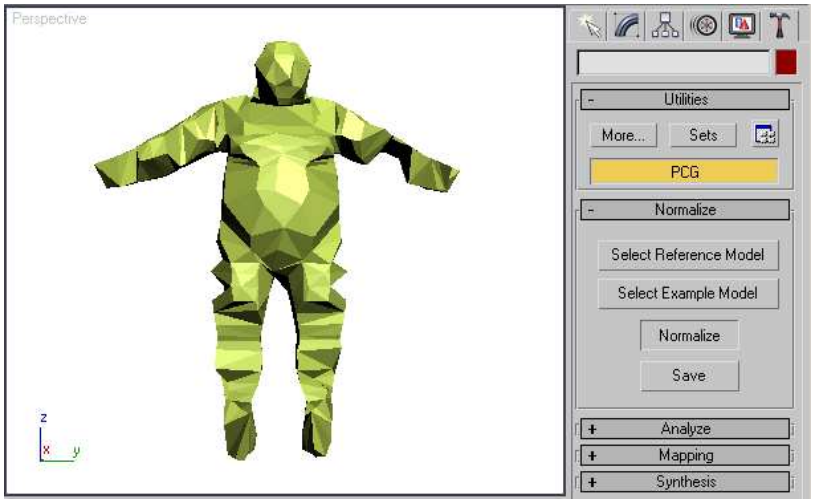
Analyzing Data Analyzing data should be a matter of adding a set of normalized models to a list. Given this list of models the analyzing component can be initiated, with the assumption that all models are normalized. Upon completion of the analysis, the user is presented with an interface for adjusting the number of principal components, and a display showing how increasing or decreasing this amount changes the result. Figure 2.4 on page 21 shows how such an interface could look using 3D Studio Max, note no option for choosing components is shown. The user must be able to save the end result in order to use it for annotation mapping or model synthesis.



(a) Selecting a Reference Model



(b) Selecting an Example Model



(c) Normalize the Example Model

Figure 2.3: Normalizing a model using the implemented 3D Studio Max plugin

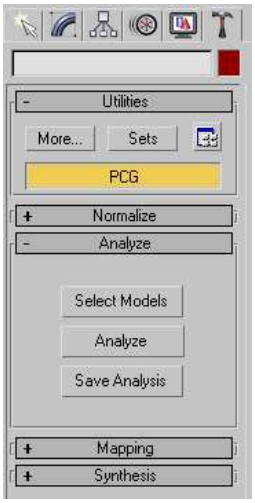


Figure 2.4: Analyzing models using the implemented 3D Studio Max plugin

Creating Annotation Mapping The process of creating annotation mappings is two fold, first a series of annotations should be created on all of the input data, secondly the actual mapping must be learned after the analysis process has been completed. When the user is creating annotations for a model, he or she should load a model and specify a series of parameters. For instance the user could specify that there should be parameters regarding height, weight, width, and race. Then for each model he or she has to provide values that specify the parameters of a particular model. If such an annotation mapping is present during the analysis phase, the user should have the option of letting the program estimate which annotation parameters maps to which analysis parameters. The idea behind this process is to convert some intuitive annotations to the unintuitive principal components of the analysis step.

Model Synthesis

Having prepared model data, it should be used in the model synthesis process. Loading an analysis database(with or without annotation mappings), the user should be presented with a series of spinners and a view-port. The view-port contains a preview of the model output given the current configuration of the sliders. As the spinners are adjusted the output changes accordingly. Ideally the dataset should determine the number of sliders and their meaning, in the case of humanoid models this will often be physical parameters. As the user finishes adjusting the sliders the model should be presented in the default view-port of the 3D modeling suite, available for further manual adjustment if required. Figure 2.5 on the next page displays a user interface for synthesizing a model based on analyzed data. The resulting model is displayed in the 3D Studio Max view port, while each of the spinners adjust the parameters of the synthesis.

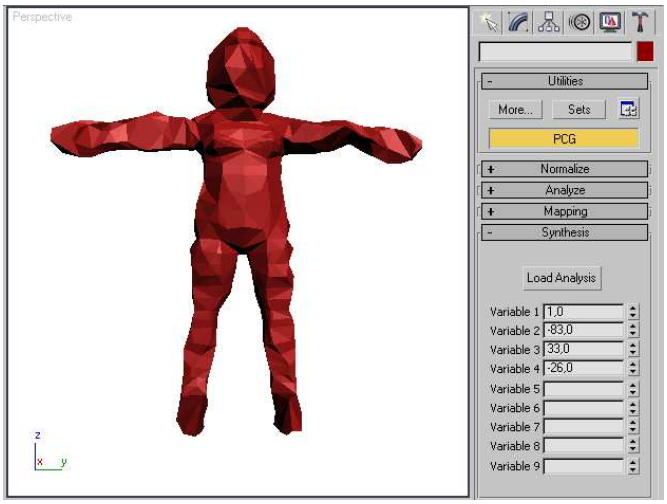


Figure 2.5: Model synthesis using the 3D Studio Max plugin

Batch jobs

The process of analyzing data is not suitable for large data sets, importing and normalizing, possibly, hundreds of models is tedious work. Likewise synthesizing hundreds of models from the analyzer dataset is not straight forward. To overcome these obstacles, a batch execution mode should be provided. The batch mode execution should have no need for a 3D modeling editor, but still be able to use the results from such an editor. Instead of specifying single models and precise physical parameters, entire collections and ranges should be specified either in a file format or from a simple command-line interface. In essence this works as a machine automated process for all possible activities.

2.2 Classes

Having presented the overall requirements and possible usage scenarios, we move on to the design of the system. The following section presents a system design, and a description of the most important classes in the design. The entire class diagram is presented in Figure 2.6 on the facing page. The remainder of the classes are described in Appendix A.

A state diagram is included for most of the classes, the purpose of these diagrams is to show the steps a class goes through in order to fulfill its purpose. Specifics regarding member attributes and methods can be found on the homepage as referred to in Appendix B.

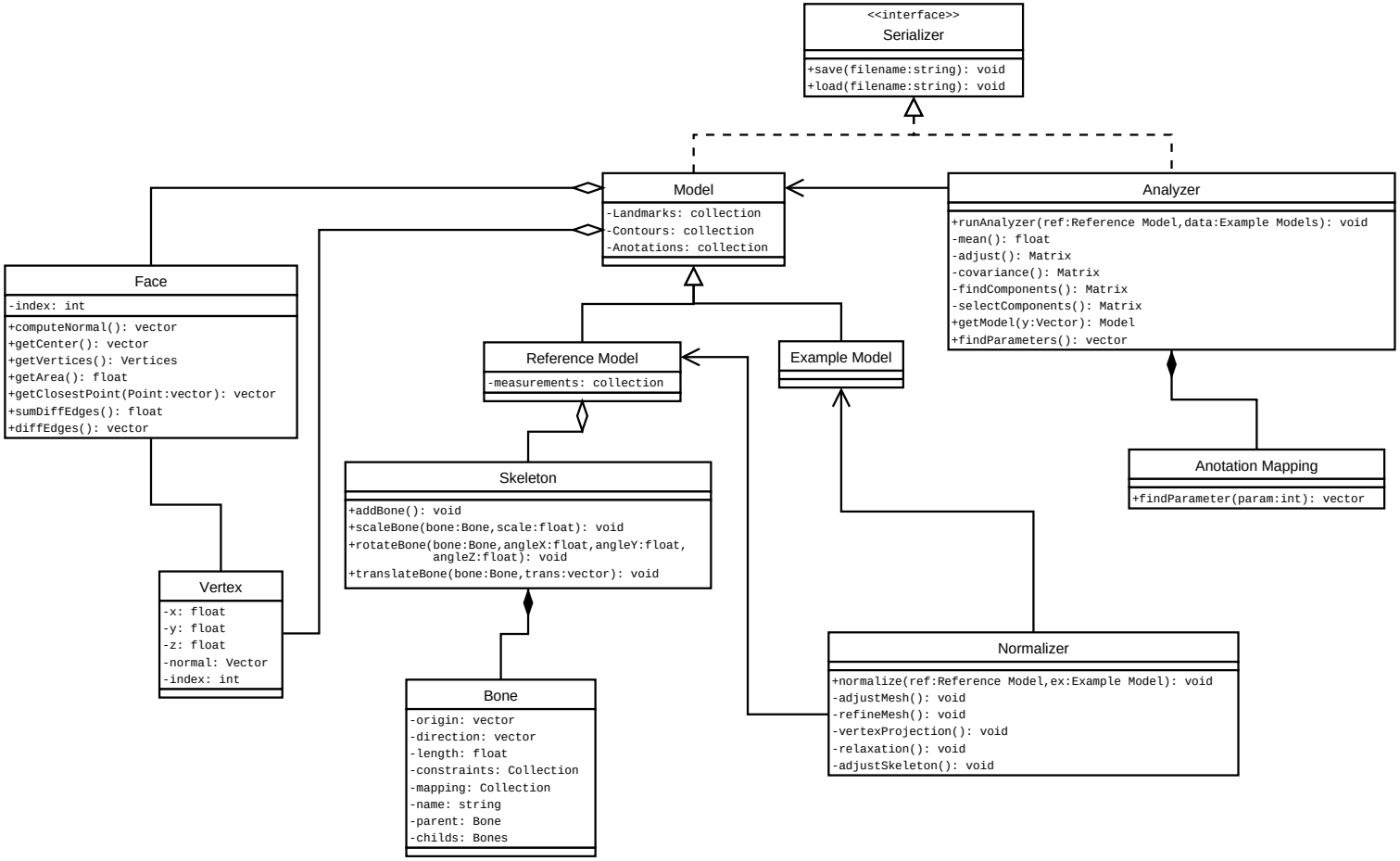
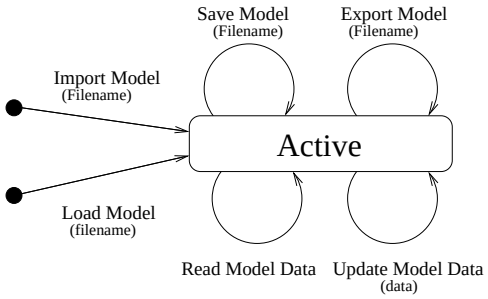


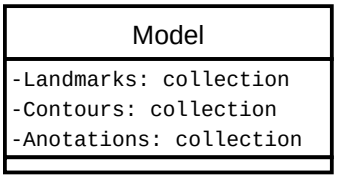
Figure 2.6: The class diagram

2.2.1 Model

Model is an abstract class used to gather the common properties of *Example Model* and *Reference Model*. A *Model* contains a number of faces and vertices. All models are annotated with physical parameters, landmarks, indicating specific points on the model such as the top and bottom vertex. A model also has contours, that is multiple vertices representing for instance a shoulder or an eye. These contours are specified by the user and are in essence landmarks with more than one vertex. Finally a model has annotations to describe some subjective artistic parameters of the model. The distinction between example and reference models is necessary for the *Normalizer* and *Analyzer* classes. *Model* implements the *Serializer* class (See Appendix A.7 on page 78) in order to provide persistence.



(a) The state diagram for the Model class

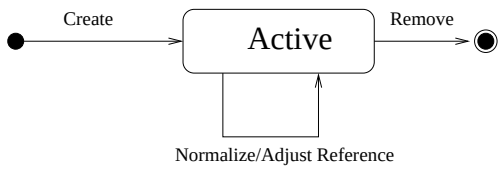


(b) The Model Class

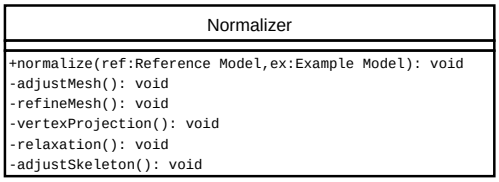
Figure 2.7: The State and Class diagram of the class Model

2.2.2 Normalizer

The *Normalizer* class is used to normalize example models such that they can be used by the *Analyzer* class. Specifically it is an implementation of a reference fitting procedure where an *Example Model* is the target model to be matched and a *Reference Model* is the source to modify in order to get a normalized example. For details about reference fitting see Section 3.2 on page 37.



(a) The state diagram for the Normalizer class

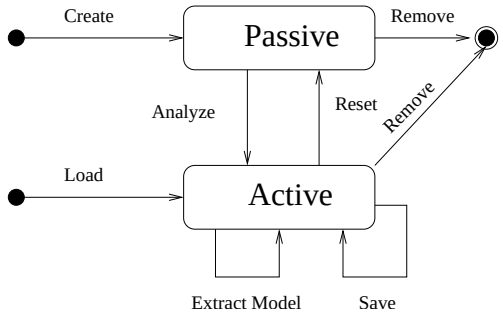


(b) The Normalizer class

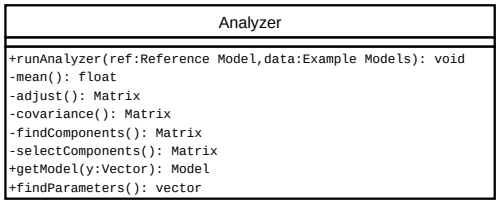
Figure 2.8: The State and Class diagram for the Normalizer class

2.2.3 Analyzer

The *Analyzer* class is used to select important components of the example data. Specifically it is a principal components analysis (PCA) implementation. Based on *Example Models* a transformation is calculated that can be used to synthesize different models from the analyzed space. Details of the PCA process can be found in Section 3.1 on page 27. The *Analyzer* class implements the *Serializer* class (See Appendix A.7 on page 78) in order to provide persistence of the analyzed data.



(a) The state diagram for the Analyzer class

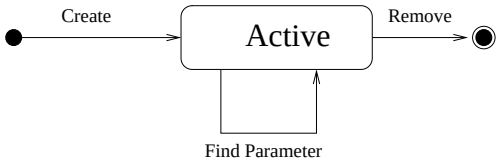


(b) The Analyzer class

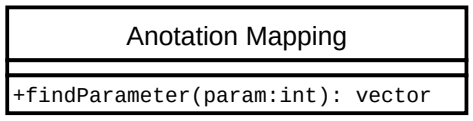
Figure 2.9: The State and Class diagram for the Analyzer class

2.2.4 Annotation mapping

The *Annotation Mapping* class is used to map the selected components from the *Analyzer* class to intuitive parameters. These parameters can be described as two types. One is physical or anthropometric, such as height, which can be measured based on the landmarks defined in the model. The other kind is a subjective type, called artistic parameters, based on the artistic annotations in the model, i.e. how much those the model belong to a given race.



(a) The state diagram for the Annotation Mapping class



(b) The Annotation Mapping class

Figure 2.10: The State and Class diagram for the Annotation Mapping class

2.3 Components

The design presented in Section 2.2 on page 22 concerns the mechanics of the application. As the application should be usable using a graphical interface and a batch mode interface, the implementation is separated into several components. The essential mechanics, as described in Section 2.2, are placed in a `Mechanics` component, exposing an interface for input and output data. The graphical and batch mode interfaces utilizes the `Mechanics` component to provide a work flow and intuitive approach to the processes involved. These interfaces are placed in an `Interface` component. As the core library and interface component need common functions a third component is introduced. The `Functions` component contains the common function such as importers and exporters, i.e. format conversion from and to the internal representations of the `Mechanics` component. The idea is that a common internal representation can be fed to the importer and exporter modules and methods can be implemented to support a variety of file formats to export or import.

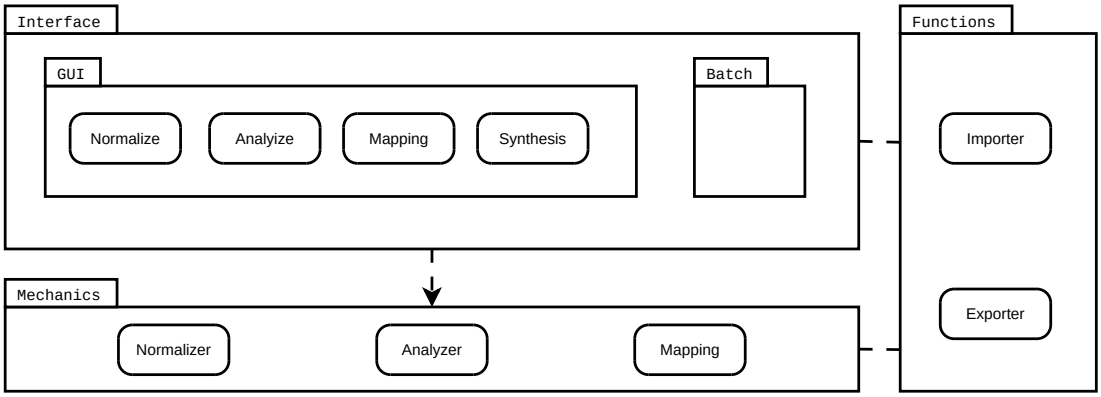


Figure 2.11: The component design

Figure 2.11 presents the relationship between these components. By separating the functionality into components and defining an interface between them, modules can be replaced or expanded as needed, essentially providing extensibility and modularity.

3

Techniques and Implementation Design

The following chapter presents some of the methods used in the *Analyzer*, *Normalizer* and *Annotation Mapping* classes. Starting with principal components analysis and continuing with reference fitting. The chapter ends with the design of a fitness function for the *Annotation Mapping* class.

3.1 Principal Components Analysis

Principal Components Analysis (henceforth PCA) is a technique used for dimensionality reduction of data. The method is a linear transformation of an example set using new basis vectors called the datasets *principal components*. PCA is well suited for finding the most important components of a dataset, i.e. the components that best describes it. For one, two, and three dimensional data, this process is fairly simple and can be done by producing graphs representing the data visually. However for data with high dimensionality, such visual representations are hard to create, and thus PCA can be a good solution.

The following section is a description of using PCA in practice and how it is used in the *Analyzer* class presented in Section 2 on page 17. The examples presents how to calculate eigenvectors and eigenvalues, how to choose eigenvectors, and how to reconstruct the original data and lastly how to get "new" data from the PCA space. The mathematical expressions and calculations are accompanied by graphical representation where appropriate. The section describes *how* a PCA can been implemented and not *why* each step is needed, for this purpose, a more thorough mathematical explanation of the required steps can be found in [Moe01, Wik03]. The data presented in Table 3.1 on the following page will be used as the main example dataset.

	2D									
a	2.5	0.5	2.2	1.9	3.1	2.3	2.0	1.0	1.5	1.1
b	2.4	0.7	2.9	2.2	3.0	2.7	1.6	1.1	1.6	0.9

Table 3.1: Example data with two dimensions, each column represents one sample

Representing Model Data

A single model represent a sample for the example set to be analyzed. Representing a model is a matter of arranging the coordinates of all vertices into a column vector, i.e. the x, y, z coordinates of the first vertex represents the three first elements while the x, y, z coordinates of the second vertex represents elements four, five and six. Thus a model with 700 vertices results in a sample vector with 2100 elements, or dimensions.

3.1.1 PCA as a Transformation

M	:	Dimension of input models, i.e. number of elements in each example vector
N	:	Number of input models
$\mathbf{X}$	:	A matrix of all example models, $\vec{\boldsymbol{x}}$ with size $M \times N$
$\vec{\boldsymbol{x}}$	:	A column vector representing a single example model, with size $M \times 1$
$\vec{\boldsymbol{\mu}}$	:	A column vector of M elements, containing the mean of example models along each dimension. $M \times 1$
$\mathbf{B}$	:	Example data with the mean subtracted. $M \times N$
$\vec{\boldsymbol{b}}$	:	A column vector $\vec{\boldsymbol{x}} - \vec{\boldsymbol{\mu}}$. $M \times 1$
$\mathbf{C}$	:	The Covariance matrix, a square and symmetric matrix. $M \times M$
$\mathbf{V}$	:	All Eigenvectors of the Covariance matrix. $M \times M$
$\mathbf{A}$	:	The selected principal components.
$\vec{\boldsymbol{g}}$	:	A cumulative energy vector of the eigenvectors

Table 3.2: PCA Legend

PCA is a linear transformation with orthonormal basis vectors (i.e. the vectors are perpendicular and of unit length) and can be expressed as a translation followed by a rotation. This transformation can be written as:

$$\vec{\boldsymbol{y}} = \mathbf{A} \cdot (\vec{\boldsymbol{x}} - \vec{\boldsymbol{\mu}}) \tag{3.1}$$

where $\vec{\boldsymbol{x}}$ is the input, or example, data (corresponding to a column in Table 3.1). $\vec{\boldsymbol{\mu}}$ denotes the mean of the dataset $\mathbf{X}$. Where $\mathbf{X}$ consists of all vectors of the example data $[\vec{\boldsymbol{x_1}}, \vec{\boldsymbol{x_2}}, \ldots, \vec{\boldsymbol{x_N}}]$. The matrix $\mathbf{A}$ is orthogonal and contains the eigenvectors, $\vec{\boldsymbol{e_i}}$, of the covariance matrix of the example data $\mathbf{X}$. Intuitively $\vec{\boldsymbol{y}}$ can be thought of as a coordinate for finding a model instance, $\vec{\boldsymbol{x}}$, in the PCA space spanned by the vectors of $\mathbf{A}$. Examining Figure 3.1 and Table 3.1 the example data, $\mathbf{X}$, is represented by the blue x-markers and each marker constitutes a sample $\vec{\boldsymbol{x}}$. Each of the lines is a column vector of $\mathbf{A}$ with origin $\vec{\boldsymbol{\mu}}$.

The idea is to describe the variance in $\mathbf{X}$ with as few parameters as possible. This is done by transforming the data to a new coordinate system, with $\mathbf{A}$ as

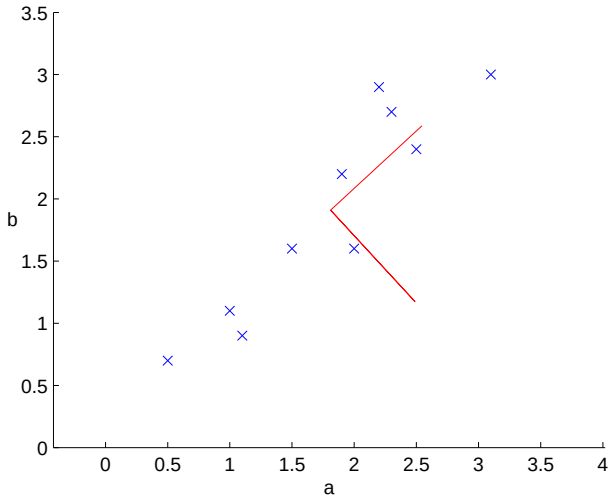


Figure 3.1: Graphical representation of the 2D Example data (the x markers) and Eigenvectors (the lines) from Table 3.1

basis vectors. The task is to find all the eigenvectors and associated eigenvalues of the dataset, and to choose the most relevant ones to constitute **A**. It should be noted that the eigenvector with the highest eigenvalue is also the one that describes the most variance. Examining Figure 3.1, it is possible to see that most of the variance of the data lies parallel with one of the eigenvectors. So for two dimensional data, it might not be necessary to calculate the eigenvectors and eigenvalues. However when dimensionality increases it can be harder to determine the axes of variance.

Doing A Principal Components Analysis

First order of business is to calculate the mean of the example data along each dimension. As in Equation (3.1) the mean is denoted $\overrightarrow{\boldsymbol{\mu}}$. Equation (3.2) shows the formula for calculating $\overrightarrow{\boldsymbol{\mu}}$

$$\overrightarrow{\boldsymbol{\mu}} = \frac{1}{N}\sum_{i=1}^{N}\overrightarrow{\boldsymbol{x_i}} \tag{3.2}$$

where $\overrightarrow{\boldsymbol{\mu}}$ is the mean vector. Each element of the vector contains the mean in a given dimension, i. If the data is three dimensional, e.g. (x, y, z), then the first element of $\overrightarrow{\boldsymbol{\mu}}$ contains the mean of all x values. N is the number of input models.

Using the 2D example data from Table 3.1 on the facing page in the equation the resulting vector becomes:

$$\overrightarrow{\boldsymbol{\mu_{2D}}} = \begin{bmatrix} 1.81 \\ 1.91 \end{bmatrix}$$

Figure 3.2 contains a plot of the example data and the mean.

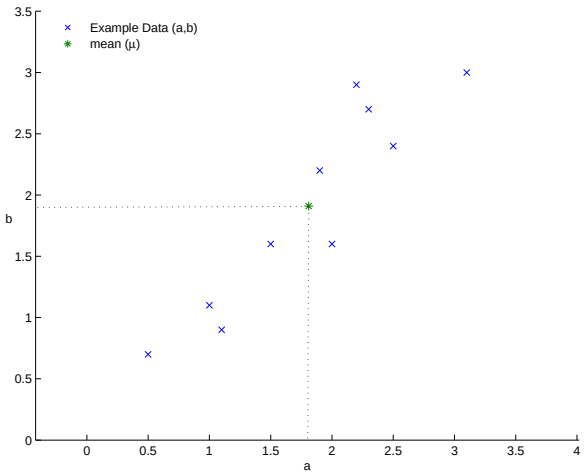


Figure 3.2: 2D Example Data and Mean μ

Next the mean should be subtracted from the example data, this is a simple calculation as displayed in Equation (3.3),

$$\overrightarrow{\boldsymbol{b_i}} = \overrightarrow{\boldsymbol{x_i}} - \overrightarrow{\boldsymbol{\mu}} \tag{3.3}$$

where $\overrightarrow{\boldsymbol{b_i}}$ is a vector used to create the matrix $\mathbf{B}$, of the same dimensionality as $\mathbf{X}$, containing the adjusted vectors $[\overrightarrow{\boldsymbol{b_1}}, \overrightarrow{\boldsymbol{b_2}}, \ldots, \overrightarrow{\boldsymbol{b_N}}]$. Again using the example data and the calculated $\overrightarrow{\boldsymbol{\mu_{2D}}}$ the new matrix becomes:

$$\mathbf{B_{2D}} = \begin{bmatrix} 0.69 & -1.31 & 0.39 & 0.09 & 1.29 & 0.49 & 0.19 & -0.81 & -0.31 & -0.71 \\ 0.49 & -1.21 & 0.99 & 0.29 & 1.09 & 0.79 & -0.31 & -0.81 & -0.31 & -1.01 \end{bmatrix}$$

The $\mathbf{B}$ matrix is needed to calculate the covariance matrix, $\mathbf{C}$, which is essential for the calculation of the eigenvectors and values. The covariance matrix, describes the correlation of the example data. I.e. the linear association between the dimensions of the data. The Covariance matrix is a square matrix of dimensions $M \times M$ and is calculated as shown in Equation (3.4).

$$\mathbf{C} = \frac{1}{N-1} \cdot \mathbf{B} \cdot \mathbf{B^T} \tag{3.4}$$

Using the calculated matrix $\mathbf{B_{2D}}$ in Equation (3.4) the covariance matrix becomes

$$\mathbf{C_{2D}} = \begin{bmatrix} 0.6166 & 0.6154 \\ 0.6154 & 0.7166 \end{bmatrix}$$

Now that the covariance matrix has been calculated, it is time to calculate the eigenvectors and eigenvalues. The details of this operation can be found in [Moe01]. At least two methods for eigenvalue decomposition are commonly used; Jacobi and Householder. Numerical routines for these methods can be found in [WRB86], and the Newmat C++ library [Dav] contains an implementation of both, as do many matrix algebra software packages. Jacobi is a reliable method, albeit slow. Householder is considered fast when compared to the Jacobi method. However the Householder method has some borderline cases where it becomes inaccurate [WRB86].

For some datasets there can be several eigenvectors solutions, at least one of these is always orthogonal if the matrix is symmetric [Wik07]. If the transformation in Equation (3.1) should be reversible, the eigenvectors must be chosen so that the matrix of eigenvectors, $\mathbf{V}$, is orthogonal.

Calculating eigenvectors and values for the example data using the Jacobi method yields two vectors,

$$\mathbf{V_{2D}} = \begin{bmatrix} 0.6779 & 0.7352 \\ -0.7352 & 0.6779 \end{bmatrix}$$

Figure 3.3 displays the example data with the eigenvectors with the mean added.

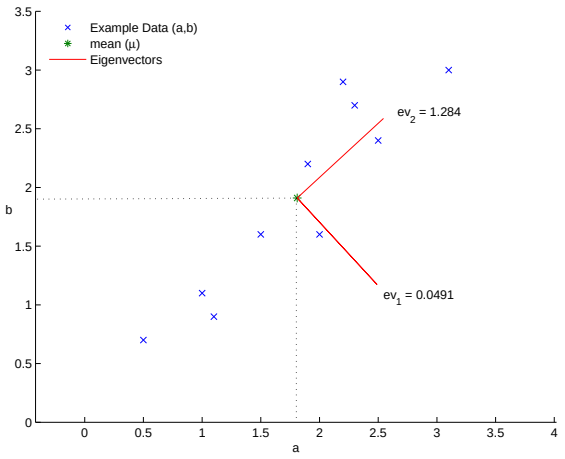


Figure 3.3: 2D Example data, eigenvectors with added mean, and eigenvalues for each vector

Choosing Eigenvectors

Returning to Equation (3.1), it is now possible to find $\overrightarrow{\boldsymbol{y}}$ for each of the example observations. Figure 3.4 on the following page presents the data in its transformed state. $\mathbf{A}$ contains all the eigenvectors sorted by eigenvalue, so $\overrightarrow{\boldsymbol{e_2}}$ has the highest

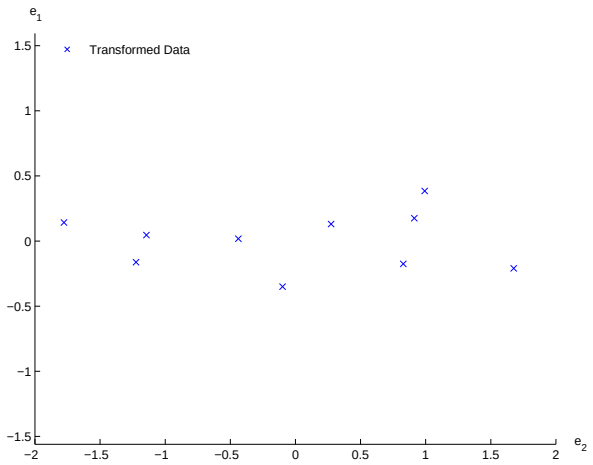


Figure 3.4: 2D data transformed

eigenvalue. As can be observed on both Figure 3.4 and 3.3 on the previous page, the most variance lies along $\overrightarrow{\boldsymbol{e_2}}$ so it is likely that the eigenvector $\overrightarrow{\boldsymbol{e_1}}$ can be ignored. For two dimensional data the probable axis of reduction can be depicted on a graph, but as dimensions increase it becomes harder to find a decent graphical representation. This brings about the question of how to choose the right eigenvectors.

If all eigenvectors are chosen to represent the transformation, the dimensionality of the data is the same as before, and no advantage is gained. However removing the eigenvectors which carry little variance in the data, it is possible to reduce the dimensionality. Several methods exists, with various complexity and results. According to [Moe01], the different methods tend to converge toward choosing the same set of components for high dimensional data. As such the *Analyzer* class implements the fast and straightforward *m-method*. The idea is to calculate the m value or *energy* of the eigenvectors and choose the eigenvectors that contains most of the cumulative energy. Equation (3.5) shows the formula for calculating the energy. It is assumed that the eigenvectors are ordered by their eigenvalues in descending order.

$$g_m = \sum_{i=1}^{m} ev_i \tag{3.5}$$

where m is the dimensionality of the data, g_m is the energy of the m'th eigenvector $\overrightarrow{\boldsymbol{e_m}}$. ev_i is the eigenvalue of eigenvector $\overrightarrow{\boldsymbol{e_i}}$, i.e. g_2 contains $ev_1 + ev_2$. Note that eigenvectors with an eigenvalue of 0 represents no variance in the data, and can be removed without further investigation.

Using $\overrightarrow{\boldsymbol{g}}$ the total energy represented by a specific number of eigenvectors

can be calculated as

$$\text{energy} = \frac{g_i}{g_m} \cdot 100 \tag{3.6}$$

where g_i is the energy of the i'th vector, and g_m is the maximum energy yield.

Vectors $\overrightarrow{\boldsymbol{e_1}} \ldots \overrightarrow{\boldsymbol{e_{max}}}$ should be chosen so that energy term is less than or equal to some threshold. Determining the required size of the energy threshold is up to the user and depends on the requirements for the results and the complexity of the data. These chosen vectors makes up the matrix $\mathbf{A}$.

$$\mathbf{A} = [\overrightarrow{\boldsymbol{e_1}} \ldots \overrightarrow{\boldsymbol{e_{max}}}] \tag{3.7}$$

Extracting Data

Having chosen the principal components, it is time to extract data from the PCA space. Restoring the original data can be achieved by rewriting Equation (3.1) as shown in Equation (3.8).

$$\overrightarrow{\boldsymbol{x}} = (\mathbf{A^T} \cdot \overrightarrow{\boldsymbol{y}}) + \overrightarrow{\boldsymbol{\mu}} \tag{3.8}$$

This can be done because $\mathbf{A}$ is an orthogonal matrix and thus $\mathbf{A}^{-1} = \mathbf{A}^T$. If $\mathbf{A}$ does not contain all eigenvectors, $\overrightarrow{\boldsymbol{x}}$ is only an estimate and the transpose of $\mathbf{A}$ is not strictly correct since a non-square $\mathbf{A}$ does not have an inverse.

Using the 2D Example data, it seems that one eigenvector is sufficient to describe the dataset.

$$\begin{aligned} \mathbf{V_{2D}} &= \begin{bmatrix} 0.6779 & 0.7352 \\ -0.7352 & 0.6779 \end{bmatrix} \\ \mathbf{A_{2D}} &= \begin{bmatrix} 0.7352 & 0.6779 \end{bmatrix} \end{aligned}$$

$\mathbf{V_{2D}}$ contains all the eigenvectors and $\mathbf{A_{2D}}$ is the selected component sorted as a row vector.

Using Equation (3.1), $\overrightarrow{\boldsymbol{y}}$ is calculated using matrix $\mathbf{A_{2D}}$, and for the first example observation in the 2D data, this results in the value

$$\overrightarrow{\boldsymbol{y}} = \begin{bmatrix} 0.7352 & 0.6779 \end{bmatrix} \cdot \begin{pmatrix} 2.5 - 1.81 \\ 2.4 - 1.91 \end{pmatrix} = 0.8394$$

Using Equation (3.8) it is possible to get the estimated data back.

$$\overrightarrow{\boldsymbol{x'}} = \left(\begin{bmatrix} 0.7352 \\ 0.6779 \end{bmatrix} \cdot 0.8394 \right) + \begin{pmatrix} 1.81 \\ 1.91 \end{pmatrix} = \begin{pmatrix} 2.4271 \\ 2.4790 \end{pmatrix}$$

The values of $\overrightarrow{\boldsymbol{x'}}$ is not exactly the same as those of the original $\overrightarrow{\boldsymbol{x}}$ but that is to be expected as half of the dimensions have been removed.

As demonstrated, by supplying one value, $\overrightarrow{\boldsymbol{y}}$, it is possible to get a two dimensional vector $\overrightarrow{\boldsymbol{x'}}$ which is close to the original data, effectively granting a simplified space.

3.1.2 The Analyzer Class

The *Analyzer* class implements principal components analysis in order to provide a simplified space for specifying a model. The values of $\overrightarrow{\boldsymbol{y}}$ can be adjusted, thus giving different output models as a result. Figure 3.5 presents the process of

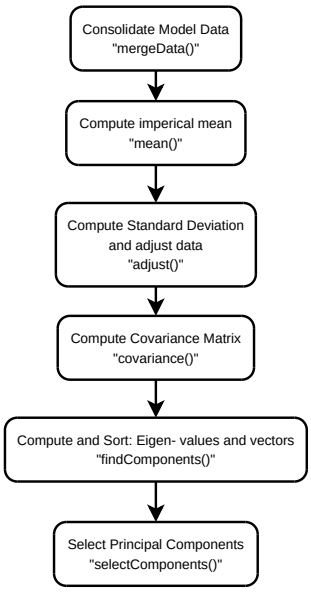
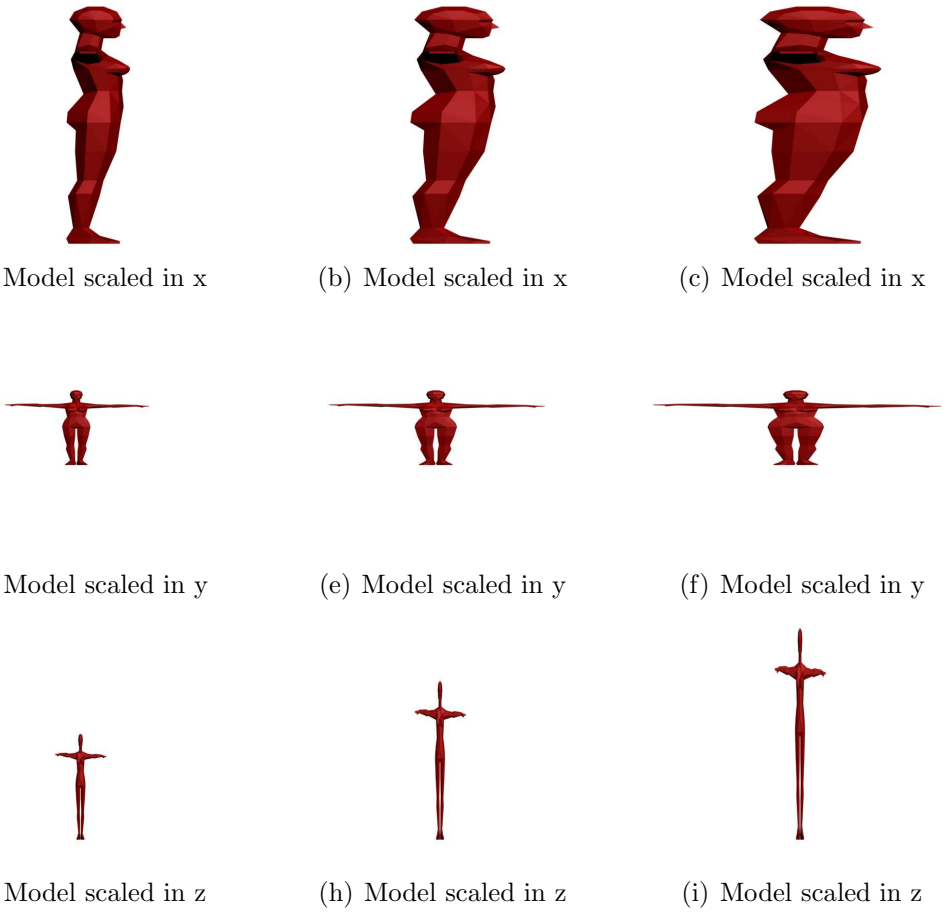


Figure 3.5: PCA process diagram

the *Analyzer* class, when the analysis procedure is initiated. Each step refers to a method name in the implementation. The flow is straight forward and has no user interaction. The consolidation step arranges all imported models into a matrix of example data, corresponding to the $\mathbf{X}$ matrix. The step of selecting the principal components, is an implementation of the m-method, and uses an energy value supplied by the user. Once the analysis have been completed, the resulting transformation can be saved for subsequent program execution. Upon creation, either from a load or from a complete analysis, the transformation matrix is made available to the GUI component, so the user can extract data based on the analysis.

The following example uses the *Analyzer* implementation to determine scale factors in a model, i.e. instead of specifying new coordinates for all vertices, only the scale along each axis should be specified. For this example 10 models have been used as input, each being scaled along the x, y or z axis. Figure 3.6 on the next page presents a render of these models. Using these models $\mathbf{A}$ is calculated and components are chosen using the aforementioned m-method. Varying the input data, $\overrightarrow{\boldsymbol{y}}$, with the values $(0,0,0)$,$(-500,0,0)$,$(0,-500,0)$, and $(0,0,2000)$, the extracted models can be rendered as presented in Figure 3.7 on page 36. A video capture of the procedure can be found on the homepage as referred to in Appendix B.



(a) Model scaled in x (b) Model scaled in x (c) Model scaled in x

(d) Model scaled in y (e) Model scaled in y (f) Model scaled in y

(g) Model scaled in z (h) Model scaled in z (i) Model scaled in z

Figure 3.6: Some of the example models used as input

(a) Model using $\mathbf{y} = (0, 0, 0)$

(b) Model using $\mathbf{y} = (-500, 0, 0)$

(c) Model using $\mathbf{y} = (0, -500, 0)$

(d) Model using $\mathbf{y} = (0, 0, 2000)$

Figure 3.7: Synthesized models with four different configurations of the $\mathbf{y}$ vector

3.2 Reference Fitting

An essential requirement for principal components analysis, is that all input data must be normalized, i.e. all models must have the same number of vertices, and the same vertices in each model should map to the same landmarks or contours. Instead of requiring the user to create normalized models, reference fitting can be applied[SMT03b]. Reference fitting is a method whereby one mesh can be changed to look like that of another model. The overall idea is that a common reference model is copied and changed so that it resembles each of the user supplied example models.

This section describes the steps used to normalize a model as it is done in the *Normalizer* class presented in Section 2.2.2 on page 24. The basic requirements for the reference fitting process is, an example model and a reference model. In order to provide reusable animations on the output data, the reference model is assumed to be rigged and skinned. Another assumption is that the example models contain a series of landmarks and contours identical to the ones present on the reference model. These landmarks and contours are needed for guiding the fitting process.

Reference fitting is a two part process: one is to change the pose, and position of the reference model by moving the skeleton, relying on skeleton driven deformation. The second part is to fit the mesh of the reference model to the example model. The following is a detailed description of the process and the parts involved.

3.2.1 Skeleton Fitting

The first step is skeleton fitting. It is the process of deforming the reference model such that it lines up with the example model. Also the deformation should result in the models having resembling poses. The deformation is achieved by translating, rotating and scaling the bones in the skeleton of the reference model. As the bones are being transformed the surface mesh should be deformed with the skeleton, this is known as skeleton driven deformation. Skeleton driven deformation is the reason that the reference model must be skinned. If there is no skinning information no connection exists between the skeleton and the surface mesh, thus the vertices can not be moved with the skeleton.

The landmarks defined on both reference and example model, are used to guide the skeleton deformation. The skeleton is placed correctly if all landmarks of the reference model, has the exact same position as those of the example model. Each bone can be associated with one or more vertices, which in turn can be associated with one or more landmarks. For each bone an error is calculated, for each of the landmarks that it is associated with. The error is the distance between the landmark of the reference model and that of the example model, as displayed on Figure 3.8 on the following page. The error function for a bone is

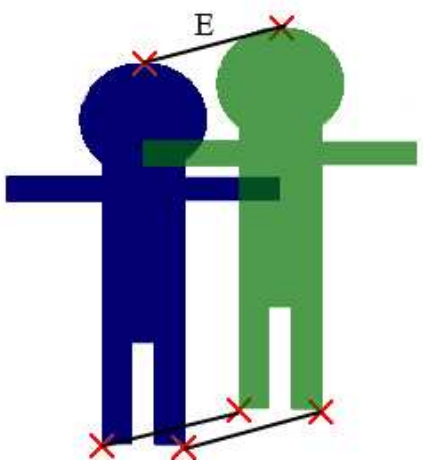


Figure 3.8: The error of a bone is measured as the distance between two landmarks

defined as presented in Equation (3.9).

$$E_b = \sum_{i=1}^{n} \left\| L_i - L_i^{'} \right\|, \tag{3.9}$$

where E_b is the error of the bone b, n is the number of landmarks influenced by the bone. L_i is the position of landmark i on the example model, and $L_i^{'}$ is the position of landmark i on the reference model when the skeleton transformation has been applied.

The idea is to find a transformation, which yields the least error over all bones in the skeleton. According to Hyewon Seo and Nadia Magnenat-Thalmann[SMT03a] the problem of minimizing the error can be solved using a direction set method called Powell's method[PTVF92].

The transformation can be any combination of the aforementioned transformations; translation, rotation and scale. However the translation transformation is only valid when applied to the root bone, as this is the only bone without a parent. Translating a bone with a parent would result in a gap in the skeleton, leaving two bones disconnected. Knowing this, the transformation is restricted to rotation and scale on all but the root bone.

The actual process of finding a transformation for all bones is done by traversing the hierarchy, starting at the root, minimizing the error of each bone at each step. The reason why such a top-down approach can be applied is that all bones inherits the transformation of their parents, as it is also described in Section 3.2.4 on page 41. A suitable solution is found when the collective error is below some user defined threshold, or if the error stops converging. Note that if a bone is not associated with any landmarks, the error will be minimized for the landmark of all its children. This is done to ensure that all parts of the skeleton is moved. For instance if the root bone has no landmarks, it will be moved according to

the landmarks of its children in order to avoid misshaped models. It must be noted that when traversing the skeleton hierarchy, in either a breadth or depth first manner, the model can be skinned in a way such that the surface will be deformed.

3.2.2 Mesh Fitting

The second part of reference fitting is the process of fine tuning the position of the mesh, in order to capture small details. This process, known as mesh fitting, is split into two separate steps: First all vertices of the reference mesh are projected onto the closest point of the example mesh. Secondly the projected mesh is relaxed, that is, an attempt to reduce the deformation of each face in the mesh. These two steps are repeated until the reference mesh is satisfactory.

Vertex Projection

Each vertex in the reference model, is projected onto the closest point on a face, edge or vertex of the example data. This projection indicates the new position of that vertex. The effect is depicted on Figure 3.9. Each time the step is repeated, only the vertices which have moved in the relaxation step are projected again. In

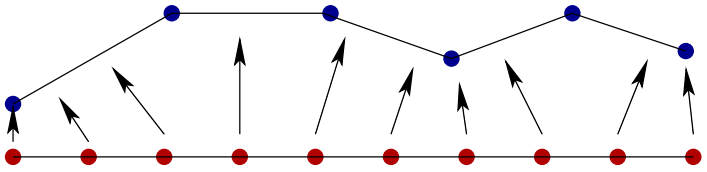


Figure 3.9: Projection of vertices onto a mesh

the implementation the process of finding the nearest point examines the distance from one vertex in the reference model, to every face in the example model.

Relaxation

Relaxation is used to minimize the deformation of the mesh after the vertices have been projected. The deformation of a given face is calculated as:

$$E_f(f) = \sum_{i=1}^{n} \left\| e_i - e_i' \right\| \tag{3.10}$$

where $E_f(f)$ is the deformation of a face f, n is the number of edges in the face, e_i is the i'th edge on the original face and e_i' is the i'th edge on the deformed face. The edges of a face can be seen on Figure 3.10 on the next page. The idea is that a face should keep its original dimensions if at all possible.

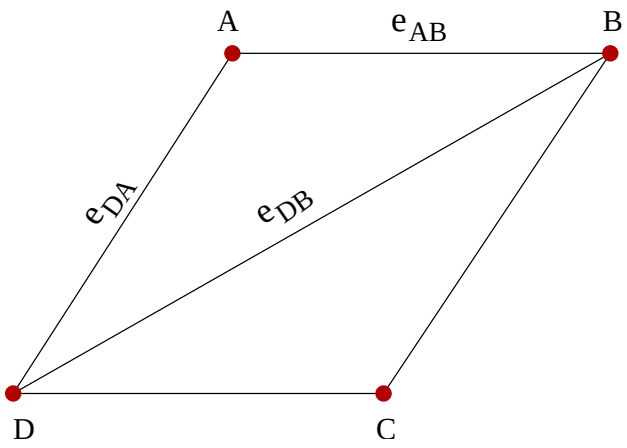


Figure 3.10: Edges of a face

An iterative gradient descent method can be used to minimize Equation (3.10)[PTVF92]. However if no further restrictions are made, each face will return to its original position, and the projection has no effect. To prevent this an extra constraint is introduced: If a vertex belongs to a contour, the vertex must be coplanar to all other vertices belonging to the same contour. In practice this is achieved by calculating an average plane of each contour on the example model, and projecting the vertices onto this plane. The constraint can be written as an error function, as presented in Equation (3.11)

$$E_c(c) = \sum_{j=1}^{k} \left\| p_j - v_j \right\| \tag{3.11}$$

where $E_c(c)$ is the distance between vertices, on example and reference model, which belongs to the same contour. n is the number of vertices associated with the contour. v_j is the j'th vertex on the reference model and p_j is the position of the vertex once it has been projected to the average plane of the contour of the example model.

By combining the two Equations (3.10) and (3.11) a common error function for the face deformation can be constructed as:

$$E(M) = \sum_{i=1}^{n} E_f(f_i) + \sum_{j=1}^{k} E_c(c_j) \tag{3.12}$$

where $E(M)$ is the error for a model M, n is the number of faces in M, and k is the number of contour in M. This error can be minimized using an iterative gradient descent method as suggested previously.

3.2.3 The Normalizer Class

The *Normalizer* class implements reference fitting in order to provide simple means for normalizing models. The process of normalizing a model requires but two models, an example and a reference, each of which must contain landmarks and contours. Once initiated the process is fully automatic and requires no user interaction. Figure 3.11 presents the process of normalizing a model. As it can

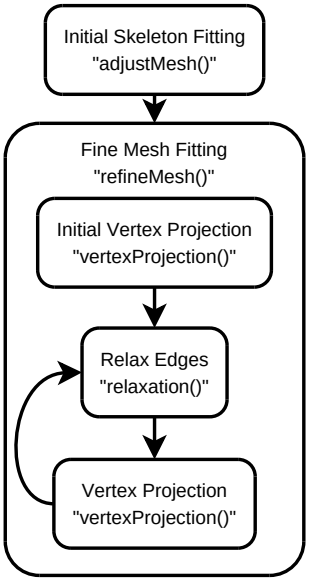


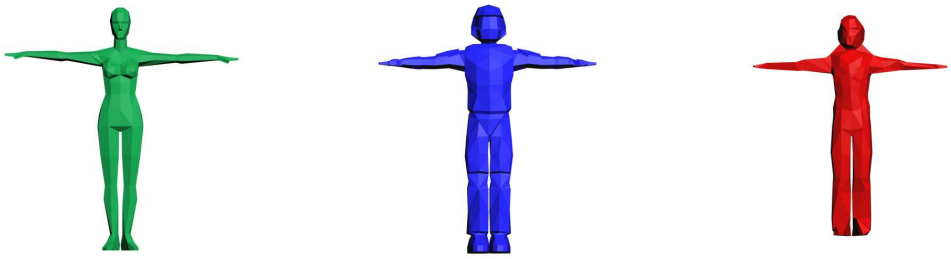
Figure 3.11: Reference fitting process diagram

be seen from the figure the process of fine mesh fitting, involves three steps, an initial projection, and hereafter repeated relaxation and projection. The "Vertex Projection" step also involves a check to determine if the change in error of the resulting mesh is sufficiently low. Each step refers to a method name in the implementation.

Figure 3.12 on the next page presents an example using the *Normalizer* class. First model from the left is the reference model, the second is the target example model, and the third model is the reference model after the reference fitting procedure.

3.2.4 Simple Skeleton Driven Deformation

Part of the reference fitting process moves or rotates the skeleton of the reference model and assumes, that by doing so the vertices follow suite. In order for this to be true some sort of skeleton driven deformation is needed. For the skeleton adjustments described in Section 3.2.1 on page 37 three operations are needed; translation, rotation and scale. The following presents simple versions of these



(a) The Reference Model (b) The Example Model (c) The Reference Model fitted to the Example Model

Figure 3.12: An example of using the *Normalizer* class

operations. For each vertex, v, it is assumed that there is a connection to one or more bones. Each of these connections has a weight w, where $0 \leq w \leq 1$

Bone Translation

Moving the bone is a matter of translating its origin point. Given a translation vector $\overrightarrow{\boldsymbol{t}}$ all vertices, v, belonging to the bone, b, are moved using Equation (3.13) resulting in new vertices v'.

$$v' = v + \overrightarrow{\boldsymbol{t}} \cdot w \tag{3.13}$$

Translating the bone is a similiar process:

$$b'_{origin} = b_{origin} + \overrightarrow{\boldsymbol{t}} \tag{3.14}$$

Notice the weight is only relevant to the translation of the vertices and not the bone.

Translating a bone also requires all bones lower in the hierarchy to be translated. In practice only the root bone is ever translated directly, otherwise bones would become detached resulting in a deformed skeleton, not usable for animation purposes.

Bone Scaling

Figure 3.13 on the next page illustrates the concept of scaling a bone along its direction vector. Consider a vertex, v, belonging to a bone, $\overrightarrow{\boldsymbol{b}}$, as a vertex has no size but only a position it cannot be scaled. Instead it should be moved in relation to the bone and the other vertices, increasing the distance to its neighbor vertices. Equation (3.15) shows how to calculate the new position of v denoted v'.

$$v' \quad = \quad v + \Delta\overrightarrow{\boldsymbol{v}} \cdot w \tag{3.15}$$

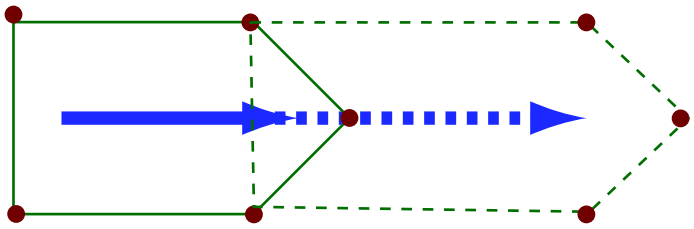


Figure 3.13: Scaling a bone

where

$$\Delta\overrightarrow{\boldsymbol{v}} = (\text{proj}_{\overrightarrow{\boldsymbol{b}}} v - b_{origin}) \cdot (s - 1)$$

where $\text{proj}_{\overrightarrow{\boldsymbol{b}}} v$ is the projection of v onto $\overrightarrow{\boldsymbol{b}}$, b_{origin} is the origin point of $\overrightarrow{\boldsymbol{b}}$ and s is the scaling factor.

Scaling the actual bone is a matter of multiplying it with the scaling factor.

$$\overrightarrow{\boldsymbol{b'}} = \overrightarrow{\boldsymbol{b}} \cdot s \tag{3.16}$$

After the bone has been scaled, all bones deeper in the hierarchy need to be translated. The translation vector is found by subtracting the two end locations of the bone being scaled, the end position before the scale and after the scale.

Bone Rotation

Given 3 angles, $\theta_x\theta_y\theta_z$, the vertex $\overrightarrow{\boldsymbol{v}}$ is rotated around the origin of the bone using Equation (3.17).

$$\overrightarrow{\boldsymbol{v'}} = \begin{bmatrix} 1 & 0 & 0 & bx \\ 0 & 1 & 0 & by \\ 0 & 0 & 1 & bz \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot R_{\theta_x} R_{\theta_y} R_{\theta_z} \cdot \begin{bmatrix} 1 & 0 & 0 & -bx \\ 0 & 1 & 0 & -by \\ 0 & 0 & 1 & -bz \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \overrightarrow{\boldsymbol{v}} \tag{3.17}$$

where bx,by and bz are coordinates of the bones origin and $R_{\theta_x} R_{\theta_y} R_{\theta_z}$ is the rotation matrix [vVB04] constructed as:

$$R_{\theta_x} R_{\theta_y} R_{\theta_z} = \begin{bmatrix} CyCz & -CySz & Sy & 0 \\ SxSyCz + CxSz & -SxSySz + CxCz & -SxCy & 0 \\ -CxSyCz + SxSz & CxSySz + SxCz & CxCy & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.18}$$

where

$$\begin{aligned} Cx &= \cos(\theta_x \cdot w) & Sx &= \sin(\theta_x \cdot w) \\ Cy &= \cos(\theta_y \cdot w) & Sy &= \sin(\theta_y \cdot w) \\ Cz &= \cos(\theta_z \cdot w) & Sz &= \sin(\theta_z \cdot w) \end{aligned}$$

This operation is illustrated on Figure 3.14.

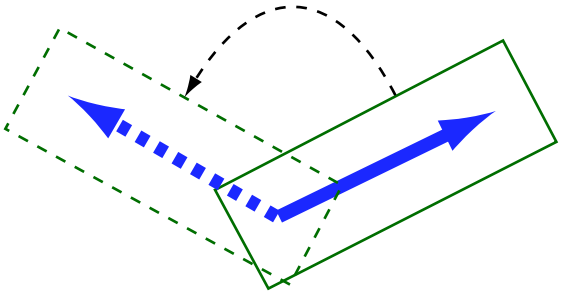


Figure 3.14: Rotating a bone

Like the other operations, rotating a bone also requires updating the hierarchy. Each bone and vertex should be rotated with the same angle and axis as the original, and it should be done around the same point.

3.3 Annotation Mappings

Given a series of models, the *Analyzer* class will provide a number of principal components usable for creating models easily. However the effect of adjusting these components is not necessarily intuitive. To overcome this, some method, for ascertaining the meaning of the component, is needed. Assuming that each model has landmarks which can be used to measure anthropometric parameters such as height, hip girth, waist girth etcetera, then we propose that these can be mapped to principal components using genetic algorithms.

The following is a description of the design used in the *Annotation Mapping* class for mapping physical parameters to principal components. Specifically it concerns the design of a fitness function for a genetic algorithm [Mit97]. It is assumed that measurable landmarks and contours are present in the models synthesized by the *Analyzer* class. The sole goal of the mapping implementation is to find the mapping of anthropometric parameters to principal components, this method is not usable for more subjective and artistic observation mappings.

3.3.1 Designing the Genetic Algorithm

As the principal components analysis discovers linear dependencies, it should be possible to find the components which affect a measurable parameter, if such compoennts exists. Since components can affect the parameter with various degree the goal for the fitness function is to discover scale factors or weights for each of the principal components.

First of all the genetic algorithm needs a population and some measure of fitness for each specimen in the population. We define a specimen as a vector

denoted $\overrightarrow{\boldsymbol{s}}$, where each element is a weight, w. A population is a series of specimens mutated and combined from the previous generation.

The fitness measure of a specimen is slightly more complicated. Given an analysis from the *Analyzer* class, and any one model from the examples used to create the analysis, a configuration $\overrightarrow{\boldsymbol{y}}$ for that single model can be found (See Equation (3.1)). This configuration is used to synthesize a model. This model is used for reference of the anthropometric parameters. A target value, t, is specified for each of the anthropometric parameters. It can be user supplied or it can be a multiple of the value from the initial model, for the *Annotation Mapping* class, the later is adopted to avoid user interaction.

Next each parameter mapping is learned in turn, i.e. each parameter is isolated so, for instance, adjusting height does not adjust hip girth. The anthropometric parameters are found for each specimen, $\overrightarrow{\boldsymbol{s}}$, by multiplying it with the initial configuration $\overrightarrow{\boldsymbol{y}}$, and then synthesizing a new model using the weighted configuration.

$$\overrightarrow{\boldsymbol{c}} = \overrightarrow{\boldsymbol{s}} \cdot \overrightarrow{\boldsymbol{y}}$$

$$\overrightarrow{\boldsymbol{x}} = \mathbf{A}^{\mathbf{T}} \cdot \overrightarrow{\boldsymbol{c}} + \overrightarrow{\boldsymbol{\mu}}$$

Determining the actual fitness value of a specimen is a matter of measuring the distance between parameter values of the initial model, the current model, and the target value, of the parameter being learned. The fitness of a specimen, $f(s)$, is calculated as shown on Equation (3.19).

$$f(s) = |t - p| + \sum_{i=0}^{n} |p_i - p_i'| \tag{3.19}$$

p is the value of the anthropometric parameter that should be learned, t is the target value for p, n is the number of other parameters, p_i is a parameter of the initial model, and p_i' is a parameter of the model synthesized using $\overrightarrow{\boldsymbol{c}}$.

The effect of Equation (3.19) is that the specimen is punished for adjusting parameters other than the one being learned, and it is awarded for approaching t. It should be noted that the "optimal" value of $f(s)$ is 0.

Once the fitness $f(s)$ has reached an appropriate value, the specimen that received the best fitness is stored and saved as a mapping to be used in the the synthesis. The entire process is repeated for each anthropometric parameter.

4

Results

This chapter presents a series of tests for the methods described in the previous chapter. Following the tests is a general status summary and discussions of the drawbacks and possibilities of the implementation.

4.1 Test System and Procedures

The following are the technical specifications of the test system (see Table 4.1), and a general description of the test cases. The test cases described are carried

System Name	bart.cs.aau.dk
CPU Cores (#)	4
Frequency (Mhz)	2200
Memory (mb)	4096
Bogomips/CPU (#)	4387
OS	RHEL4
CPU Name	AMD Opteron 275

Table 4.1: Specification of the test systems

out without the GUI component wherever possible, i.e. via a console program without loading 3D Studio Max. This is to avoid any overhead not relevant to the analysis and normalization process, and to demonstrate the speed when running in batch mode, where performance is assumed to be optimal. Each test case is described with a purpose and information of how the test was carried out, where appropriate a short table presents the test parameters.

4.1.1 Input Data

For the non-visual performance test, i.e. the ones that compares different mathematical implementations, random data is generated instead of hand made exam-

ples to ensure a varied input distribution. The following section describes how these random example models are generated. First the number of models, N, and the dimensionality, M. For this example N is chosen to be 100 and M is 5000. A matrix $\mathbf{R}$ of dimensions $M \times N$ i.e. 5000×100, is filled with random floating point numbers between 1 and 10.

$$\mathbf{R} = M \times N \text{ matrix of random numbers}$$

As principal components analysis is used to to find linear dependencies, entirely random data is of little use. So a matrix $\mathbf{Q}$ is constructed as linear combinations of $\mathbf{R}$.

$$\mathbf{Q_n} = a_1 \cdot r_1 + \ldots + a_n \cdot r_n$$

Q_n is the n'th column of Q, r_n is the n'th column of R. The scalars, $a_1 \ldots a_n$, are random numbers in the range $[-1; 1]$. So Q is a $M \times N$ matrix of linear combinations of R and each column in Q represents one example model.

4.2 General Observations

The following is a few general observations which influences the performance of the application.

4.2.1 Single Threaded Implementation

All parts of the application is implemented as a single thread, i.e. no parallel calculations, this means that only one cpu (or core) is ever used to do the calculations. Decrease in calculation times can be achieved by distributing the matrix calculation, either by multi-threading the application or by distributing the calculations for several computers. Depending on the amount of input data, and details of the models to be normalized, this can be a desirable improvement.

4.2.2 Memory Consumption

As a matrix is needed for calculation it is loaded into memory in its entirety, and intermediate results are kept in memory till the computation is done. If the amount of data is sufficiently large the memory is maxed out and swapping occurs. Swapping drastically reduces the performance of the application. As distributing the computations involves splitting matrices into smaller pieces, this can help alleviate the problem, however only to a point. Matrices should be loaded into memory as needed, and intermediate results only kept as long as needed, i.e. in the case of matrix multiplication it should be sufficient to load a smaller number of columns at a time instead of the entire matrices.

4.3 Test 1: Performance Jacobi vs Householder

The *Analyzer* class implements two methods for finding eigenvectors and eigenvalues, the test presented in this section is designed to determine the performance of the two methods. The test is two fold; the first is used to measure the two methods against each other, the second part measures the increase in running time as the number of models and dimensions increase.

4.3.1 Test 1a: Description

To measure the performance of the two methods against each other, the *Analyzer* is loaded with 100 example models (random generated as described in Section 4.1.1 on page 47), each having a dimensionality of 2000. Running time, in second, is meassured for each method execution. To eliminate external factors, such as changes in cpu load and hard-drive searches, each method is run 10 times, and the average across all 10 iterations is used as a result. The simple performance measure is the difference in time between the two methods. Table 4.2 presents an overview of the test parameters used in this test.

Test parameters	**Amount**
Input models	100
Dimensions	2000
Test runs	10

Table 4.2: Summary of the parameters used to test the performance of Jacobi and Householder methods

4.3.2 Test 1a: Results

As the Householder method is in general deemed a faster method [WB64, Dav] than Jacobi, it is expected that it is also the case for random generated data. Furthermore it is expected that the speed does not change significantly when changing the data, i.e. with different random data samples.

Figure 4.1 on the following page presents the running time for each of the 10 test runs, while Table 4.3, shows the average values. By examining Figure 4.1 on

Method	**Avg Time(seconds)**
Jacobi	5029.025
Householder	467.036

Table 4.3: Average running time for Jacobi and Householder

the following page and Table 4.3 it can be observed, that the Householder method

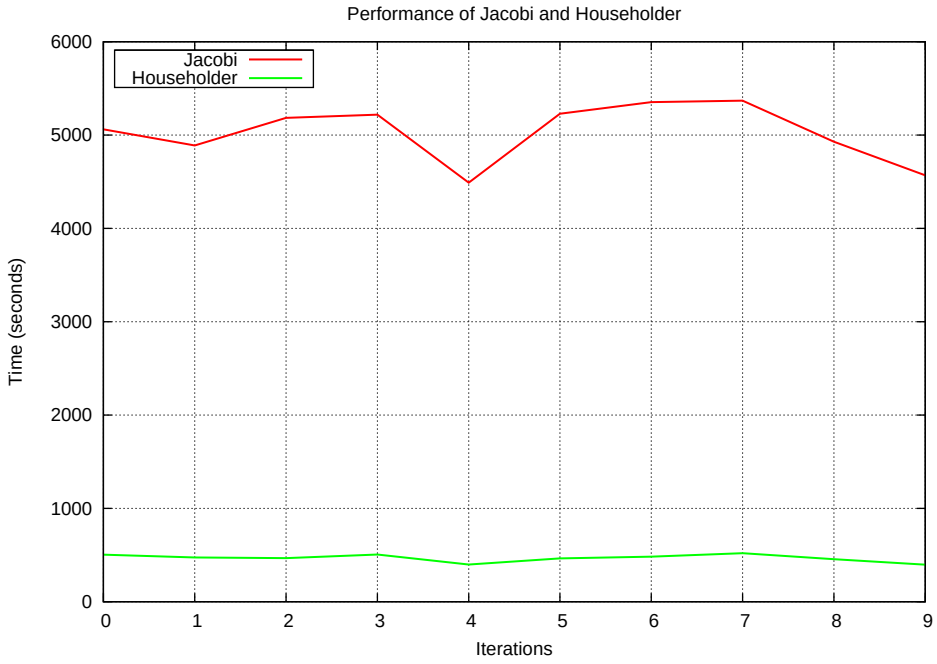


Figure 4.1: Timing results for each iteration of the Jacobi and Householder performance test

is significantly faster than the Jacobi method at 2000 dimensions. Furthermore the biggest difference in running time is approximately 900 seconds, for Jacobi, which can be accounted to differences in cpu load (multiple users system).

4.3.3 Test 1b: Description

The second part of the performance test should determine what happens when dimensions are increased, and similar when the amount of input data is increased. The actual test is performed by running each method a number of times, storing the same performance parameter as in Test 1a, i.e. running time in second. Additionally at each iteration the random generated data will be increased either in dimensionality or in input size.

For the Jacobi method there are two test executions, each having 10 iterations. In the first execution the starting configuration is: 10 input models with a dimension of 10. Each iteration the dimension is increased by 10. The second test execution has a start configuration of 10 models with a dimension of 100, at each iteration the number of input models is increased with 10. Two similar tests are carried out using the Householder method.

4.3.4 Test 1b: Results

Eigenvector calculations are carried out on the covariance matrix (See Section 3.1.1 on page 29), which is a square matrix with the same size as the dimensions of the data. Because of this it is expected that increasing dimensions will increase time of calculations, while the amount of input data should have no effect in this calculation. Figure 4.2 presents the results from increasing the dimensions. It

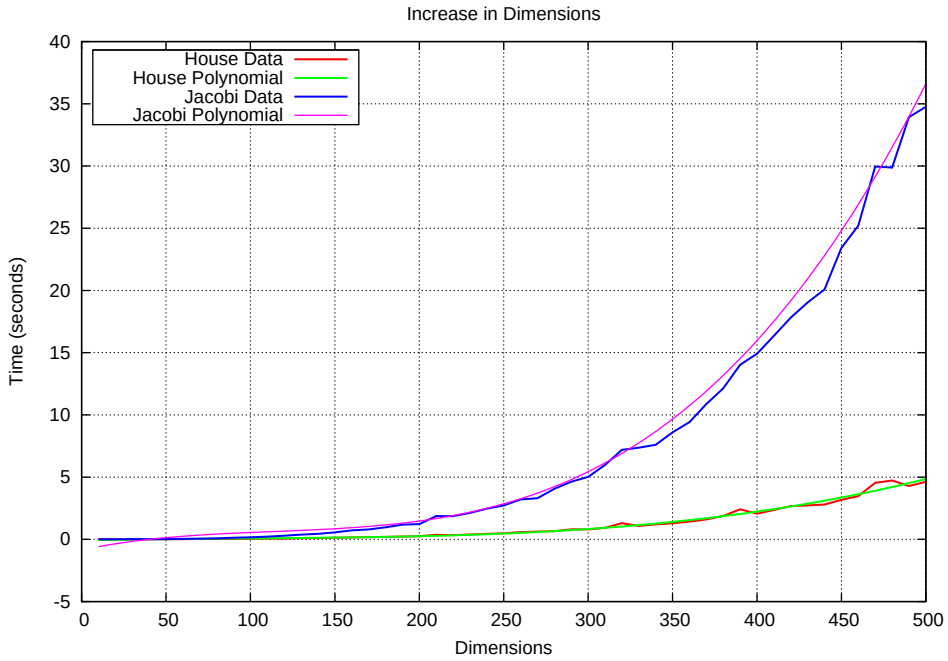


Figure 4.2: Average timing results for different sizes of example data

is clear from the previous test that householder is the fastest method, this test indicates that the running time of Jacobi also increases more than Householder. The two polynomials on the figure are:

$$\begin{aligned} \text{Jacboi}: \quad & f(x) = 0.00000058 \cdot x^3 - 0.0002 \cdot x^2 + 0.0281 \cdot x - 0.8453 \\ \text{Householder}: \quad & f(x) = 0.000000067 \cdot x^3 - 0.0000203 \cdot x^2 + 0.0030 \cdot x - 0.0923 \end{aligned}$$

The results of Test 1 indicates that Householder is the preferred method for finding eigenvectors and eigenvalues. As the dimensionality of the data increases, the difference becomes more outspoken. Increasing the dimensionality of the data increases the running time as the size of the covariance matrix is based on the data dimensionality. This indicates that running time can be improved by minimizing vertices in the normalized models, i.e. by using the least detail reference model possible when normalizing the input data. As expected, increasing the amount of input data has negligible effect on the overall running time as the heavy computation lies in finding the eigenvectors.

4.4 Test 2: Accuracy of Householder

As mentioned previously, in Section 3.1.1 on page 29, the Householder method for finding eigenvectors and eigenvalues is, in some cases, less accurate than the classical Jacobi method. The test presented in this section is designed to measure the amount of inaccuracy to determine if Householder is a suitable method compared to Jacobi.

4.4.1 Test 2: Description

In this test it is assumed that Jacobi is the accurate solution and Householder is the method to be tested. The test is carried out on some random data, $\mathbf{Q}$, consisting of 100 models each having a dimensionality of 2000. The eigenvectors of $\mathbf{Q}$ is determined using both Jacobi and Householder. If the space produced by the method is identical, the eigenvectors should be pairwise parallel, i.e. the angle, θ , between two eigenvectors with the same index should have a value of 0 or 180 when measured in degrees. The angle is calculated as presented in Equation (4.1).

$$\cos(\theta) = \frac{\mathbf{j_i} \bullet \mathbf{h_i}}{\|\mathbf{j_i}\| \cdot \|\mathbf{h_i}\|} \tag{4.1}$$

where j_i and h_i is the i'th eigenvector calculated with Jacobi and Householder respectively.

4.4.2 Test 2: Results

If all the eigenvectors are pairwise parallel the resulting transformations should be equally good, thus it is only a matter of performance which method is used. The expectations, for these test results, are that most of the eigenvectors are parallel, however perhaps with minor deviations in special cases. Figure 4.3 on the next page presents the result of the test carried out on a dataset of 100 models with 2000 dimensions in each. The error plotted is calculated as

$$E = 1 - abs(\cos(\theta))$$

When $\cos(\theta)$ is 1 or -1 the vectors are parallel, i.e. providing an error of 0. Ideally all vectors should be parallel indicating identical transformations, however as it can be seen by examining Figure 4.3 on the facing page this is not the case. In fact the average error is 0.912. Further examination shows that eigenvectors with associated eigenvalues above 0 are parallel, i.e. has an error of 0. Since the eigenvectors are sorted this explains why the last 100 eigenvectors has an error of 0 while the rest are random. As described in Section 3.1 on page 27 all eigenvectors with an eigenvalue of zero can be ignored as they contain no information about the structure of the data. This result is an indication, that

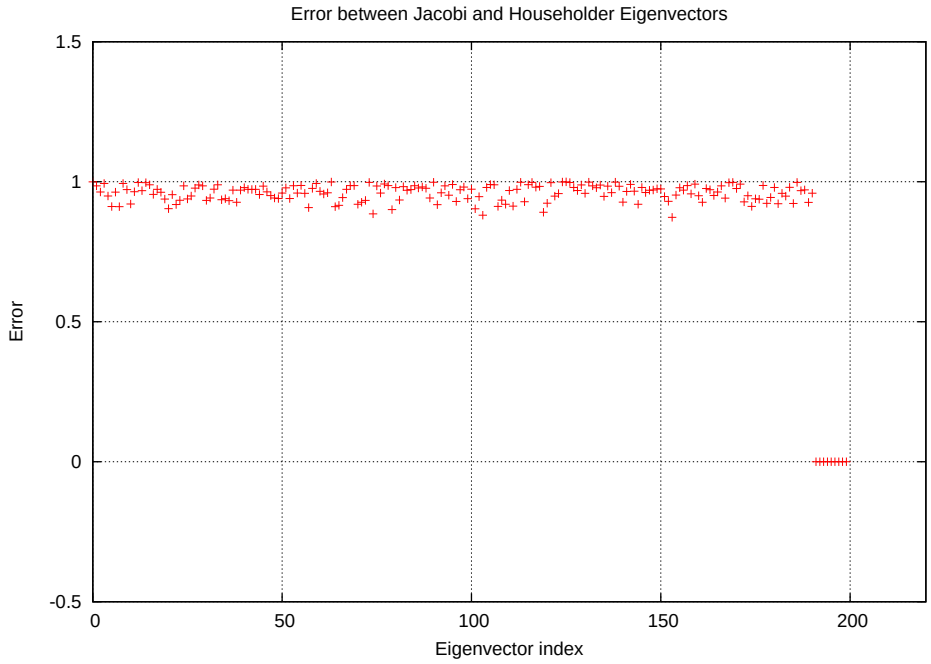


Figure 4.3: Error between pairs of Jacobi and Householder eigenvectors, only every 10th result is presented

the Householder method is suitable for finding eigenvectors in the *Analyzer* class. The actual implementation of the class has an option to switch between Jacobi and Householder in case of problems with the Householder method.

4.5 Test 3: Performance of Mesh Fitting

The test presented in this section is designed to measure the performance of the mesh fitting implementation of the *Normalizer* class.

4.5.1 Test3: Description

To get an idea of the performance of the mesh fitting procedure, random data can not be applied. Instead a common reference model, with 761 vertices, is used. 4 different models of humanoids with varying number of vertices are normalized. The reference model has been changed so it is placed in the same position as the example models. This is done because the skeleton fitting procedure has not been implemented. Table 4.4 on the following page presents the metrics of the example models as well as the reference.

Model	Vertex count	Face count	Contour Count
Woman (Reference)	761	1518	30
Soldier	869	1269	30
Master Chief	2182	3896	30
Bomberman	1914	3700	30
Fatman	660	1316	30

Table 4.4: Vertex, Face and Contour count for the test models

4.5.2 Test3: Results

Figure 4.4 shows the change in deformation as the reference model is changed to fit the examples. The deformation value is calculated using Equation (3.12). As

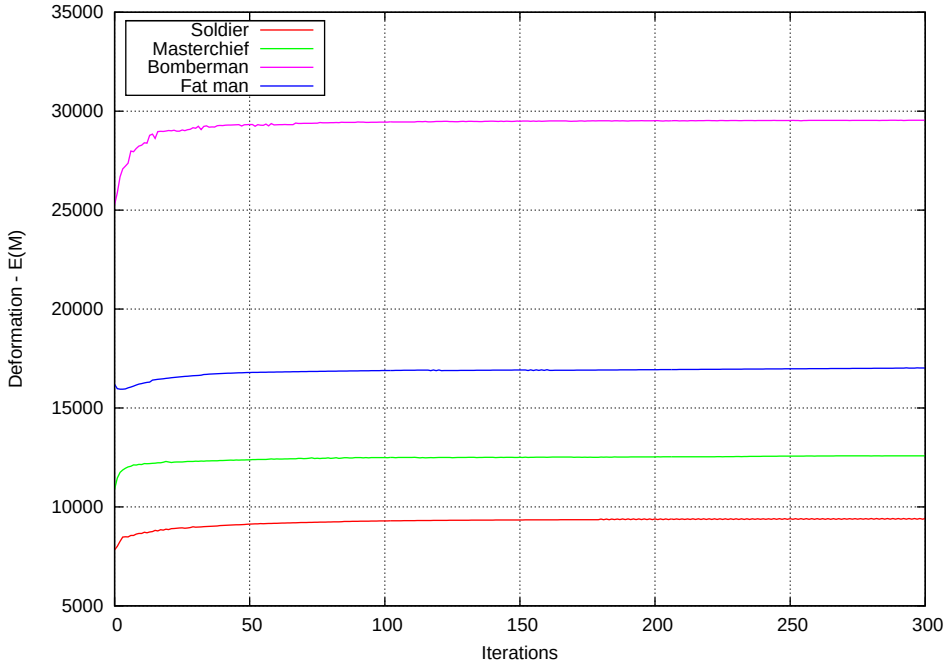
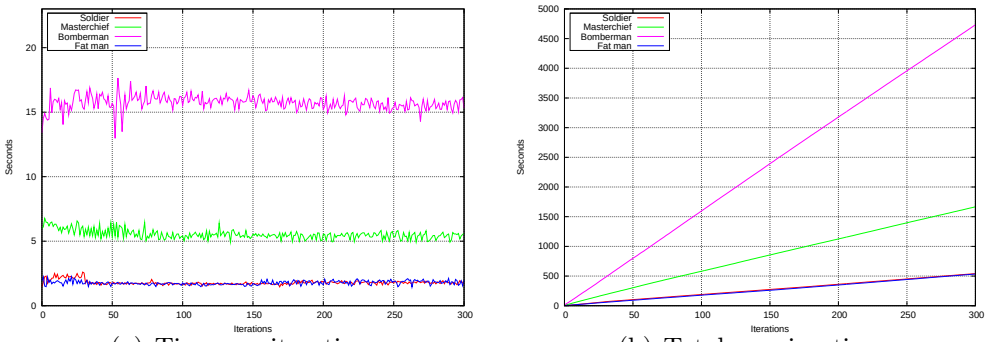


Figure 4.4: Deformation for the models

the figure shows, the deformation levels out after 50 iterations. The deformation levels out when the vertices have been placed "correctly" and the relaxation stops changing the vertex position significantly.

Figure 4.5 on the facing page shows the running time for each of the example models. Looking at Figure 4.5(a) on the next page it can be seen that the time of each iteration neither increase or decrease as the fitting progresses. This observation is confirmed by examining Figure 4.5(b) on the facing page which shows a linear increase in total running time

Examining the deformation, the running time and the number of vertices, of the different models, it can be observed that the running time is dependant on



(a) Time per iteration (b) Total running time

Figure 4.5: Running time for the test models

mainly two things; how many vertices the example model has and how much the reference model needs to be deformed. The first is partly explained by the implementation for projecting the reference model, it is a brute force aproach which calculates the distance to all faces of the example model. The indication that the larger the deformation the longer the running time, can be due to the gradient descent method involved in the relaxation. If the deformation is large, it is likely to need more iterations to converge.

4.6 Test4: Model Normalization

The following is a simple test of the reference fitting process. No matter how fast, the reference fitting process is best judged by visual quality. The following section presents a gallery created using one common reference model, and a fixed amount of contours. Table 4.4 on the preceding page contains information about the number of contours on each of the models, as well as the amount of vertices on the original model. As with the performance test the pose of the reference model has been manually adjusted.

4.6.1 Test4: Results

As the goal is to produce low detail models, the reference model used in this test only contains 761 vertices. Therefore it is likely that the fitting produces models with lower quality than the originals.

Figure 4.7 on page 57 shows a fitting of the Soldier model after 1, 2, 3, 5, 10 and 20 iterations of the mesh fitting procedure. Examining Figure 4.7(e) and 4.7(f), i.e. the fitted model after 5 and 10 iterations of the mesh fitting procedure, there are subtle changes in the face and vertex positions. Furthermore the head is different on the two models. The changes that occur after 10 iterations are visually negletible, and thus the soldier model only needs around 10 fitting

iterations. Fitting the models presented in Table 4.4 on page 54 a maximum of 20 iterations are needed for all models.

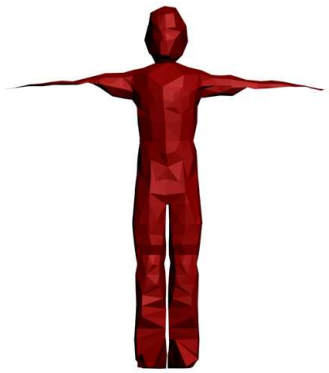
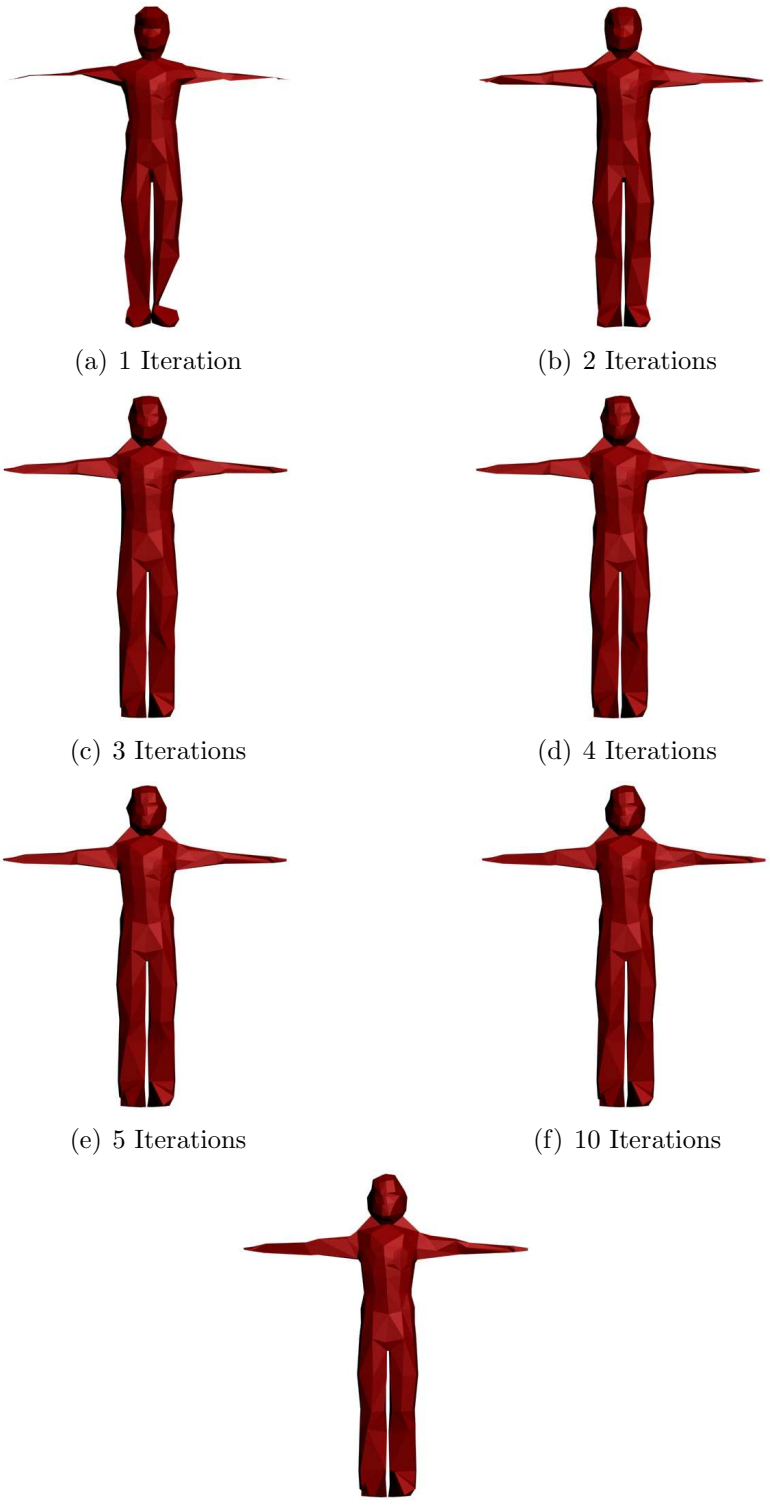


Figure 4.6: Soldier model with no contours around the arms

Figure 4.8 on page 58 displays the example models along with their normalized counterpart. The figure shows that models with varying nature can be fitted using a common reference.

So far the example have all been defined with 30 contours, Figure 4.6 presents a render of the Soldier model with only 14 countours. The 16 contours determining the position of the arms have been removed.

As it can be observed the Soldier in Figure 4.6 have flat arms, thus these contours are needed to get a good result. The nature of the reference and example model determines the number of contours needed. The number of contours can also be changed for each pair of reference and example model, as they are only used for guiding the fitting process.



(a) 1 Iteration

(b) 2 Iterations

(c) 3 Iterations

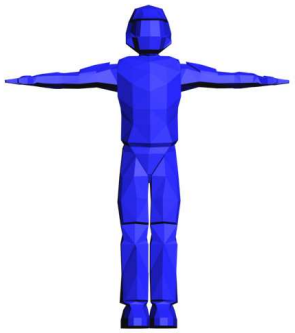
(d) 4 Iterations

(e) 5 Iterations

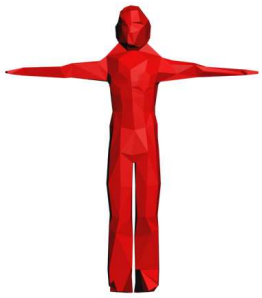
(f) 10 Iterations

(g) 20 Iterations

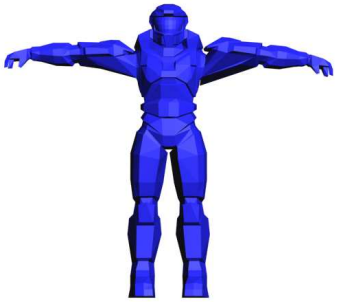
Figure 4.7: Fitting the Soldier model



(a) Example Soldier

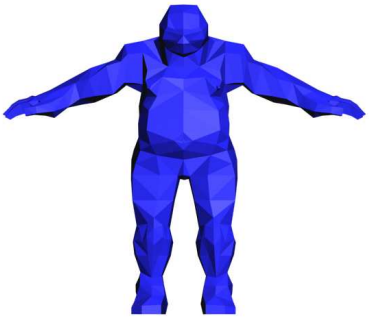


(b) Reference Fitted Soldier



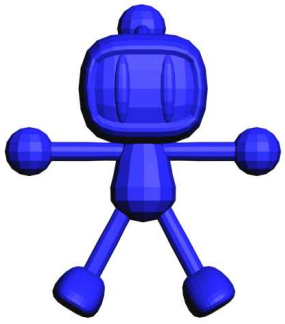
(c) Example Master Chief

(d) Reference Fitted Master Chief



(e) Example Fatman

(f) Reference Fitted Fatman



(g) Example Bomberman

(h) Reference Fitted Bomberman

Figure 4.8: Example models normalized with the normalizer class

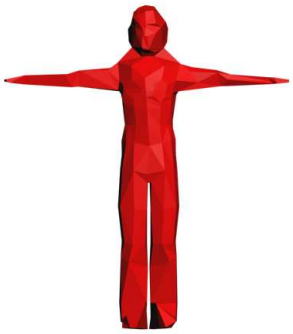
4.7 Test5: Model Synthesis

This section presents models synthesized by the *Analyzer* class, much like the previous section the idea is to present visual results of the principal components analysis. The models used in the analysis are the same ones used for testing the *Normalizer* class. The analysis is limited to one instance of the Woman (the initial reference model), Soldier, Master Chief, Bomberman and Fatman models. To make an extensive test more models are needed, however we have been unable to acquire an amount of models large enough to do extensive testing.

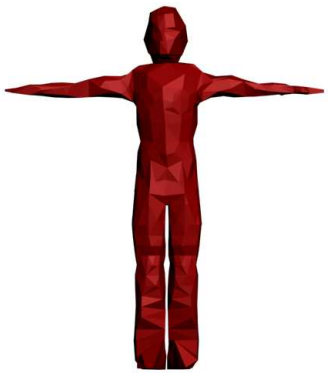
4.7.1 Test5: Results

A simple test is to determine if the models used for the analysis can be synthesized given their reduced parameter space. For each input model a vector $\overrightarrow{\boldsymbol{y}}$ is found. This $\overrightarrow{\boldsymbol{y}}$ vector is used to synthesize a model which should resemble the input model. Figure 4.9 on the following page presents the models used as input and the synthesized models. The normalized models consists of 761 vertices which equals to dimensionality of 2283, the synthesized models have been created using only four parameters.

Synthesizing the input models is not sufficient because we already had those models from the beginning. Figure 4.10 on page 61 presents models synthesized using four different analysis databases. The models have been synthesized using a limited number of models, thus the variance seems somewhat limited. Nonetheless, as it can be observed by examining the figure, models different from the originals can indeed be synthesized.

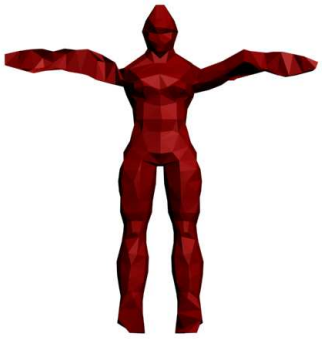


(a) Reference Fitted Soldier



(b) Synthesized Soldier

(c) Reference Fitted Master Chief

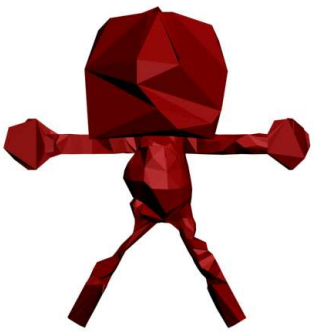


(d) Synthesized Master Chief

(e) Reference Fitted Fatman

(f) Synthesized Fatman

(g) Reference Fitted Bomberman



(h) Synthesized Bomberman

Figure 4.9: Input models and their synthesized counterparts

(a) Synthesized model from a pca based on 2 models

(b) Synthesized model from a pca based on 3 models

(c) Synthesized model from a pca based on 4 models

(d) Synthesized model from a pca based on 5 models

Figure 4.10: Synthesized models using different analysis databases

4.8 Test6: Annotation Mapping Performance and Quality

This section presents a test of the *Annotation Mapping* class. Mainly it concerns the convergence time for mapping the parameters, and the quality of the mapping achieved.

4.8.1 Test6: Description

To test the running time and quality of the *Annotation Mapping* class, a training has been executed, recording the error, number of iteration and the running time. Table 4.5 shows the parameters used for the genetic algorithm, these parameters can be adjusted possibly yielding a result faster.

Parameter	Value
Analysis Database	5 Models
Target	120% of actual value
Population size	1000
Number of Crossovers	100
Number of Randomized Specimen	100
Mutation chance	50%

Table 4.5: Parameters of the genetic algorithm used in the annotation mapping class

4.8.2 Test6: Results

Figure 4.11 on the next page presents the fitness value of the best gene at each iteration for the three parameters, height, arm length and stomach girth. The best gene is preserved until a better gene is found, thus there is no guarantee that the value changes at each iteration. The fitness function, used to score the genes, is implemented as described in Section 3.3.1 on page 44. Remember that the best fitness value is 0. As it can be seen, by examining the figure, the fitness function converges after around 60 iterations. This is no guarantee as it may never converge if no mapping exists, or if the mapping relies on many parameters being changed. The analysis database used only allows for adjusting four parameters which also affects the number of needed iterations.

Figure 4.12 on the next page presents the running time of each iteration. As it can be observed there is almost no change in running time and a single iteration completes in approximately 8.3 seconds, given the current population size and mutation parameters.

A mapping has been learned for height, arm length and stomach girth. Figure 4.13 on page 64 presents four models, an initial model with no adjustments,

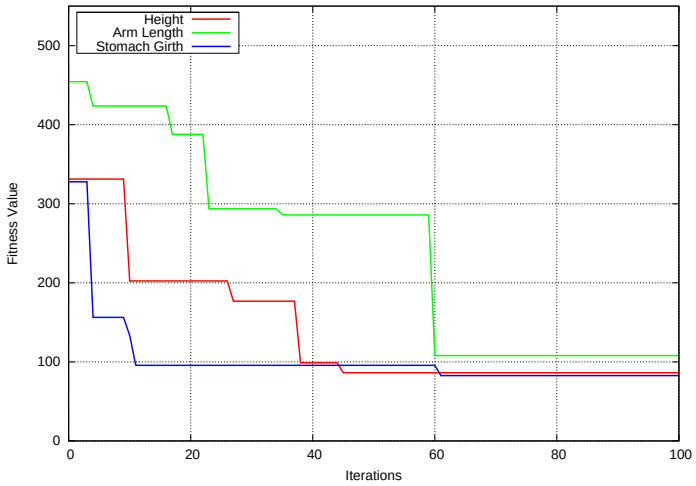


Figure 4.11: Value of the fitness function for the best gene

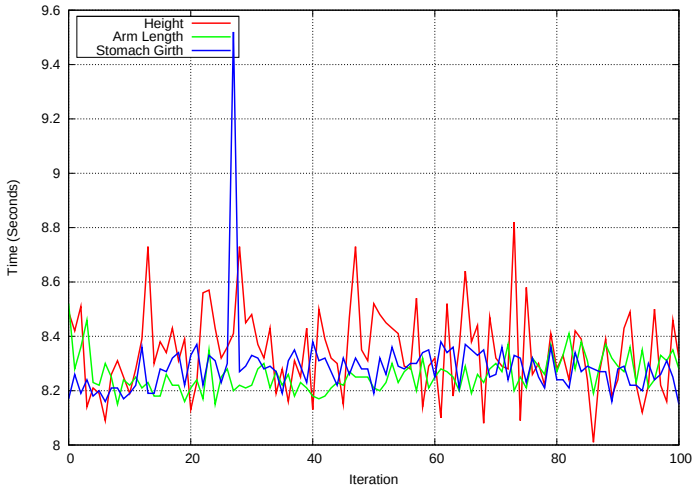
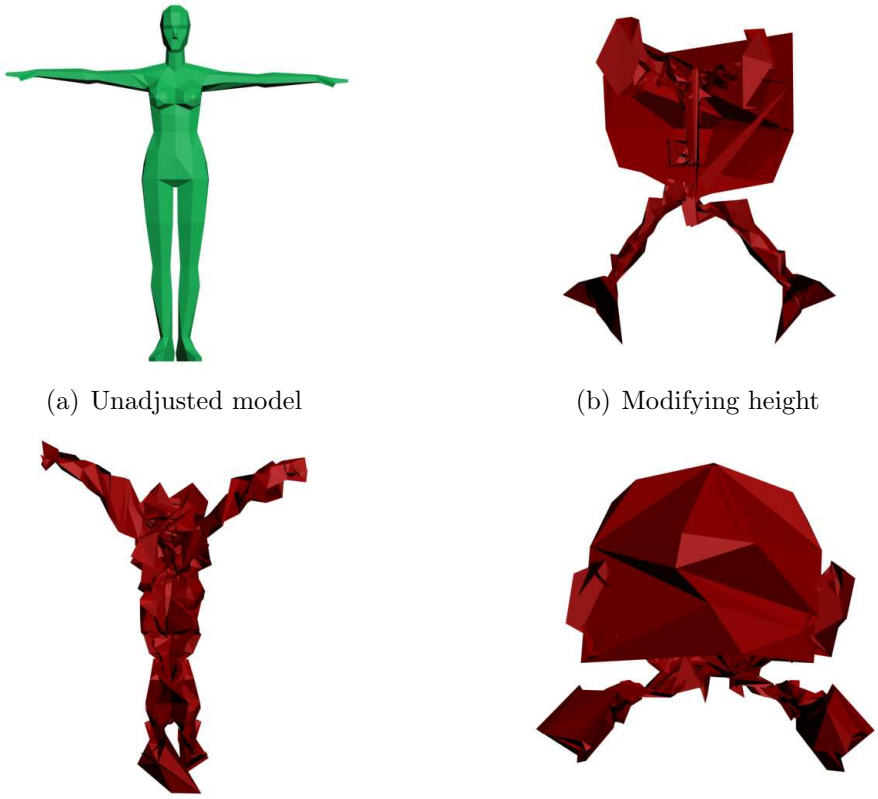


Figure 4.12: Running time of each iteration

a model with the height adjusted, one with arm length adjusted and finally one with the stomach girth adjusted. As it can be observed from the figure, the



(a) Unadjusted model

(b) Modifying height

(c) Modifying arm length

(d) Modifying stomach girth

Figure 4.13: Models synthesized by adjusting antropometric parameters

adjustments does indeed change the models anthropometrics. However it also changes a lot of other stuff. The problem is likely caused by the amount of parameters and the lack of models with different anthropometrics. It seems that the mapping class is far from useful in its current state, however testing with a larger model base is needed to be conclusive.

4.9 Implementation Status and Discussion

The following is a status on the implementation laid out in Section 2.2 and Chapter 3 and a discussion of the test results, limitations and possibilities the current implementation sets.

4.9.1 Mechanics Library

The framework, or `Mechanics` component, has been implemented as described in Section 2.2 on page 22. However not all methods are fully implemented, the skeleton fitting in the *Normalizer* class is still unfinished. Likewise the *Annotation Mapping* class has no implementation for learning artistic parameters. The implemented components includes a PCA implementation in the *Analyzer* class, a fine mesh fitting procedure in the *Normalizer* class and a genetic algorithm to learn a mapping from principal components to anthropometric parameters. Furthermore the basics of skeleton fitting has been implemented, in the form of operations for simple skeleton driven deformation.

The mechanics library implementation is known to run on Linux and Microsoft Windows when compiled using GCC Version 3.4.6 and the Visual Studio 2005 compilers respectively.

4.9.2 Interface and Functions

An example of a graphical interface has been implemented in the form of a 3D Studio Max plugin. The plugin can handle data normalization, analysis, mapping and model synthesis. It does so by interfacing with the methods of the `Mechanics` component. Furthermore the plugin relies on the `Functions` component to convert data from the internal data structure to the 3D Max format, and back again. The plugin implementation has little to no input validation, for instance it is possible to start an analysis on unnormalized models, which will result in the analysis class aborting without a result. The desired behavior would be for the analyzer to be unavailable until normalized models have been loaded. The 3D Studio Max plugin is known to run on Windows using 3D Studio Max 8.

A simple batch interface has been implemented for running tests and like the 3D Studio Max plugin it relies on the `Functions` and `Mechanics` components. The batch interface does not provide any means for specifying ranges for the PCA as described in the original goals. The batch interface is known to run under the same conditions as the `Mechanics` library.

Common for both interfaces is that no system has been provided for specifying landmarks, contours, or anthropometric annotations.

4.9.3 Landmarks and Contours

In the first part of this project [BM07] it was suggested that, automatic landmark techniques be used for the aid of the reference fitting process. An alternative to such a process is a graphical user interface which allows for easy manipulation of landmarks and contours on a model.

In Sections 3.2 and 3.3.1 regarding reference fitting and annotation mappings, it is assumed that landmarks and contours exists for all models. In the imple-

mentation (see Appendix B both landmarks and contours can be defined for each model independently. However the procedure requires manual editing of the model file using a text editor. This is neither easy nor quick, as it requires that the artist is able to identify the index of each vertex and associate it with a landmark or a contour.

No further work has gone into examining if landmarks and contours can be placed automatically. For the reference fitting process the use of automated landmarks might be suitable, as they are used for guiding the reference mesh. However to learn the anthropometric parameters, the landmarks should have intuitive meaning. Otherwise the annotation mapping would need to be redesigned. All in all for production use a facility, for manipulating landmarks and contours is still needed, even though it means updating all example models.

4.9.4 Normalizing Data

The *Normalizer* class should be an implementation of the reference fitting method described in Section 3.2 on page 37, however the actual implementation used for testing only involves fine mesh fitting, and not the skeleton fitting process. The *Normalizer* class does contain an implementation of the operations needed to deform the mesh, but no method for minimizing the error when adjusting the skeleton.

The testing done with only the mesh adjustments indicates that, by manually adjusting the pose of the reference model, the example models can be matched closely. The end result is dependent on the number of contours, and vertices in the two models. Once the examples have been prepared the process itself is fairly quick, only requiring a few minutes for each model.

In the current implementation the mesh fitting procedure does not account for overlapping faces. This means that faces can cross creating artifacts on the normalized model which are not present on the example. Likewise when determining where to project a vertex, the distance is calculated from the vertex to every face on the model, in order to find the closest one. This is an inefficient procedure with $O(n^2)$ complexity. A possible solution to alleviate these errors and inefficiencies, is to use collision detection techniques, placing the vertices in a suitable data structure[VT00, Ebe05, vVB04]. The collision detection itself would make sure to avoid moving face through each other. As many collision detection algorithms relies on special data structures, the time used for finding the closest vertex can possibly be reduced by utilizing these data structures.

For the *Analyzer* class to work correctly the normalized example models must changed to have the same pose. The current implementation provides no such functionality. However an approach is to inverse the transformation found during the skeleton fitting process. Alternatively a skeleton fitting step can be added to the normalization process. I.e the normalized example model should have its skeleton fitted to that of the reference model. No work has gone into examining

if either of these methods are viable in practice.

4.9.5 Analyzing and Synthesizing Models

As we have been unable to acquire actual 3D models for testing, and as the *Normalizer* class have yet to be fully implemented, the testing has been carried out on a limited number of models.

No test has been created to find out how many models are required before the data set becomes sufficiently large, to create reasonable results. Likewise it has not been tested when the dataset becomes too large to get reasonable results. To ascertain if the proposed solution fulfills the requirements, test should be carried out using production ready model data.

The test cases described in Section 4.7 on page 59 indicates that models can be synthesized easily. However with the small amount of model data few models can be synthesized that actually looks like humanoids. This can possibly be alleviated, a little, by included the same models in different scales.

4.9.6 Annotations and Mapping

Previously [BM07] it was suggested that, mapping intuitive parameters to principal components should utilize an incremental neural network with a special radial basis function. As the current design only considers anthropometric parameters, such a non-transparent approach it not necessary. Instead simpler search techniques can be used, in this case genetic algorithms. However finding anthropometric parameters is not sufficient as more artistic parameters are needed. Future work must go into examining such a mapping, and here the suggested RBF network seems like a viable solution.

Section 3.3 on page 44 describes the design of a genetic algorithm for learning a mapping between annotations and the principal components. As with the landmarks and contours, these annotations can be defined independently for each model using a text editor. Again this is not a quick solution and a graphical interface is needed for production use.

The tests carried out in Section 4.8 on page 62 indicates that the convergence time of the genetic algorithm is relatively fast, however the test only has room for adjusting four parameters, which makes for a small search space. The results also indicates that the mapping fails entirely, the anthropometrics are rightly adjusted, however other parts of the model, for which no measurements exists, are also adjusted resulting in unusable results.

It is possible that no mapping exists between a parameter and one or more principal components, in this case the genetic algorithm will never converge. Safeguards can be developed which allows the algorithm to terminate. However as a mapping cannot be found, the principal components will have no intuitive meaning, giving rise to the question if they should be left out or the subject

of another mapping routine. In [SMT04] a similar mapping is created using a linear regression model. In general it seems that the anthropometric parameter mapping can be found using any search method, the only requirement being that each parameter can be locked and found separately.

Determining Artistic Parameters

As mentioned the mapping class only determines anthropometric parameters. Several methods might be suitable for learning the more artistic parameters. Such methods are likely to require either an absolute annotation, or a weighted annotation of each example model. I.e. a model is either an orc or it is not. Or all models are orcish to some degree. It is possible that a link exists between the anthropometric parameters and the artistic parameters. Having already determined the anthropometric parameters these, could be used by a mapping routine. Figure 4.14 illustrates the idea behind mapping via the anthropometric parameters. Further work must go into examining the use of RBF networks for finding

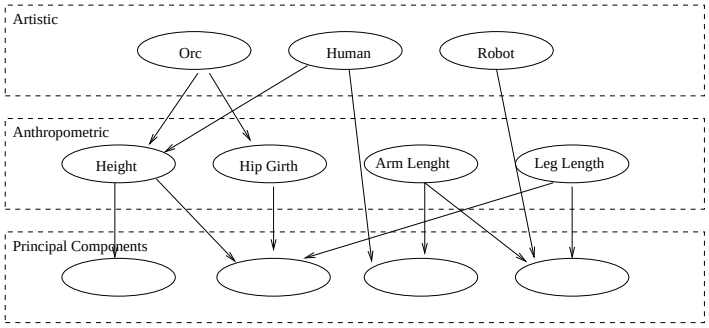


Figure 4.14: Artistic mapping using anthropometric parameters

these artistic mappings. In the current implementation the unintuitive principal components are still exposed to the artist.

5 Conclusion

Rounding off, this chapter presents an evaluation of this thesis. The first part of this thesis we examined the possibilities for generating procedural models. One possibility was to base the model generation on example databases and another was to glue together primitives using a series of rules. We have opted to look into the first possibility, using a example models, as the analysis showed it was likely to solve most of the requirements presented. This report, the second part of the thesis, has been concerned with the implementation of said method. We have presented a series of goals, both technical and general, which an application should fulfill in order to be useful in a game production by a graphics artist. We have presented requirements, design and test of an application which should fulfill these goals. The following sections presents discussions on how we consider the goals fulfilled, and what must and can be improved.

5.1 Implementation Goals

In Chapter 1 we presented a series of application goals, some necessary for the application to be useful, others as necessary consequences of the methods applied. The following is a recap of these goals and how well they have been achieved. It should be noted that the conclusions are based on tests using only a few models.

5.1.1 Implementation of Reference Fitting

Reference fitting should be implemented in order to provide easy means for normalizing model data. The idea behind normalizing model data is to enable the reuse of exisiting models. Even though some manual work is needed for each model, the normalization process works for different models as shown in the tests. Changing pose and position of the reference model should be done man-

ually in the current implementation, however this is a small job in any mordern 3D editor. The current implementation does have it limitations however, small details like eyes and ears are hard to match without a large amount of landmarks and contours. Additionally the landmarks and contours need to be assigned manually. Overall the reference fitting implementation works as intended, however it still requires some amount of work.

5.1.2 Automatic Principal Components Analysis

The core facility of the application is an automated process of analysing a series of models, a prerequisite for generating new models. The principal components analysis has been implemented as described in Section 3.1 on page 27. It works as intended and on a mordern computer the running time is acceptable. No tests have been carried out to determine if the component selection algorithm is robust, and efficient. Neither does the graphicl implementation, the 3D studio max plugin, expose a means to select specific components. However during the tests there seemed to be no problems in the results that the m-method provided.

5.1.3 Format Facilities and Database Persistence

Since the application is geared toward game production use, the models should be readily availble in one or more common formats. The current implementation provides facilities to convert from an internal format to the common x3d format, and back again. Furthermore an `Interface` component have been developed which can convert to a 3D Studio Max scene environment, from which models can be exported to a variety of formats. The framework also allow for easy addition of extra format conversion methods. As both x3d and 3D Studio Max are commonly used formats the model data can be considered easily available. Another requirement was that the analysis database should be available between session, i.e. that there should be no need to analyze the entire database of models, each time a few models should be synthesized. A simple text file system has been provided for saving and loading the database. Thus allowing for different databases to be used in the same session, or moving and sharing the database. We consider these solutions to satisfy the goals of having readily available models, and a reusable analysis database.

5.1.4 Interactive Model Generation Using Intuitive Parameters

Another important requirement is that an artist must be able to generate models easily. The most intuitive is to represent the result visually, and letting the user adjust intuitive parameters which affects the result at runtime. The principal components analysis allows for runtime synthesis of models, and the 3D Studio

Max plugin utilizies this to present the result of a configuration, directly in the users viewport. Effectively this provides interactive model generation. Regarding the intuitive parameters, a method has been designed to map, user placed, annotations to the less intuitve parameters provided by the analysis. This mapping sequence is less than perfect and is not guaranteed to work, however in some cases it can be used to determine which principal components can be used to change anthropometric parameters. The conclusion must be that interactive model generation has been achieved, however a lot of work is needed if the parameters should be intuitive.

5.1.5 Batch Model Generation

One of the goals is to provide interactive model generation, another is to provide massive amounts of models in order to be able to populate game worlds with varying characters. To provide this functionality batch based model generation should be developed. No batch interface meeting the requirements have been implemented. However the framework is designed so that such an interface is easily constructed, and indeed a console based interface have been implemented for the analysis and normalization procedures. This goal is not satisfied, however only minor work is needed for the current console application to become satisfactory.

5.2 Production Use

We proposed to reduce production time by making principal components and reference fitting availble for use by graphics artists. Based on the relatively small amount of models that was used for testing no conclusive verdict can be made. However current results indicates that it is indeed possible to generate models quickly. As initially required meshes are easily generated using pca, and it requires less work than constructing new models manually. Likewise models can indeed be mass produced given the development of a batch application, however with the small amounts of models tested so far, results suggest that the models are likely to resemble each other. Whether this changes significantly as the number of example models remains to be seen.

We suggested that a requirement for production use was the possibility to reuse animations on a group of models. No work has gone into showing whether this is actual possible. However all models synthesized from a specific analysis database has a common skeleton, and all likewise they all have the same pose and skinning information. Thus it seems likely that atleast simple animations can be reused. This intuition is based on the assumption that the mesh is not moved significantly in the normalization procedure.

So is the current implementation usable for actual use in a game production? The answer is, as of now, a maybe. Further test must be done in order to deter-

mine if the synthesis procedure provides enough variance to be usable. Research should go into mapping artistic parameters to principal components, as this is as subject not thouroghly examined in this report. Furthermore the reference fitting process should be finalized, and likewise for the console based application. The base application is already portable and is known to run on multiple systems, including unix and linux variants, our oppinion is that this is an important aspect. If the normalization, analysis and mapping procedures are all carried out at the same time the process is likely going to take some time. Many companies have render farms running linux and unix systems so computing power is already available, ensuring an even faster process. Overall the implementation can be considered partly succesfull at reducing model generation times, but improvements are still needed.

5.3 Improvements

Finishing off the following presents possible improvements to the methods and the implementation, some have been discussed or mentioned in the report, others are possible expansion not yet touched upon.

Landmark and contour interface Implementation of a graphical interface, possibly in the 3D Studio Max plugin, used to place landmarks, annotations and contours.

Automatic Landmarks Implementation of automated landmarks for guiding the reference fitting.

Better annotation mapping Research and implementation of a more general mapping method.

Database expansion suggestions A tool for mapping model into the PCA space and out again, helping the user to figure out if the database should be extended or if the model can already be represented with the current information.

Texture data Research of how texture data can be included in the analysis procedure.

Parallel computation Implementation of parallel computation methods for pca, reference fitting and annotation mapping.

Adding primitives after synthesizing Research and implementation of a method for adding primitives to the synthesized model in order to provide extra details and variations.

Appendix

Appendix A
Class descriptions

A.1 Vertex

A *Vertex* is a single point in 3D space. It is made up of three coordinates and a vector, representing the position and direction of the normal respectively. A vertex is used in conjunction with other vertices to form one or more *Faces*. A

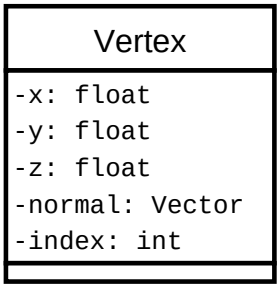


Figure 1: The Vertex class

Vertex is basically a point in space, as such it has no behavior, other than it can be created and removed.

A.2 Face

A *Face* is a surface spanned by three or more vertices. A *Face* contains a vector which represents the surface normal. A series of *Faces* is used to describe the surface area of a model. As vertices are transformed any *Face* associated with them also "moves", as such a *Face* cannot be moved, scaled or rotated directly, instead it relies on the transformation of its vertices. A *Face* is basically an association of vertices, as such it has no behavior, other than it can be created and removed.

A.3 Bone

A *Bone* is a vector in 3D space used for *Vertex* deformation. It consists of an origin point, a direction and a length. A *Bone* contains a collection of references to vertices. When a *Bone* is rotated, so is each *Vertex* that it references. To model the concept of joints with limited degree of freedom, a *Bone* includes constraints on how many degrees it can be adjusted in any given direction. The degrees stored in this collection is relative to the origins normal. A *Bone* also know who the parent are and which *Bones* are the children.

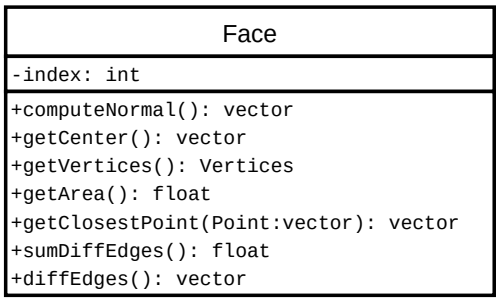
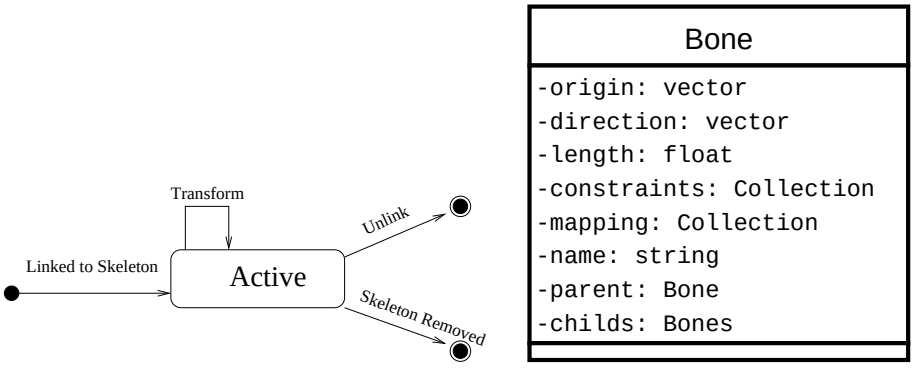


Figure 2: The Face class



(a) Bone State Diagram (b) The Bone Class

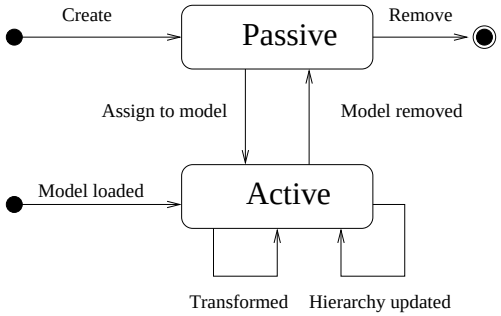
Figure 3: The Bone class

Bone Behavior

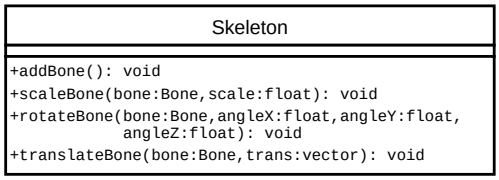
When a bone is created, it is assigned a starting position or a parent bone. It is also giving a length and a direction. Now the bone need to know its mapping of vertices and the constraints. When this is done the bone is ready to receive scale, rotation, and translation operations form the skeleton.

A.4 Skeleton

The purpose of the *Skeleton* class is to function as a hierarchy of *Bones*, determining the parent-child relationship. This hierarchy is needed because moving and rotating a bone should affect child bones.



(a) Skeleton State Diagram



(b) The Skeleton Class

Figure 4: The Skeleton class

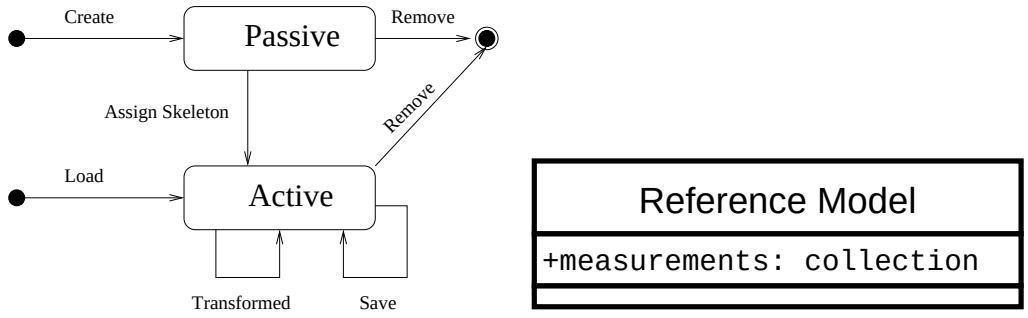
Skeleton Behavior

Upon creation of a skeleton it contains no *Bones*, the act of linking a *Bone* also creates it. When unlinking a *Bone* instance it is also removed. If a skeleton is removed so is all its associated bones. I.e. no bones can exist without a skeleton.

A skeleton can perform scale, rotation, and translation operations on the bone assigned to the skeleton, as described in 3.2.4 on page 41.

A.5 Reference model

A *Reference model* is a specialization of *Model*. It contains a skeleton so that skeleton driven deformation [LCF00] can be applied to the surface mesh and Measurements that can be used in *Annotation Mapping*. During the fitting process the *Reference Model* is the source to modify, also the analyzer modifies the surface of a *Reference Model* when generating models.

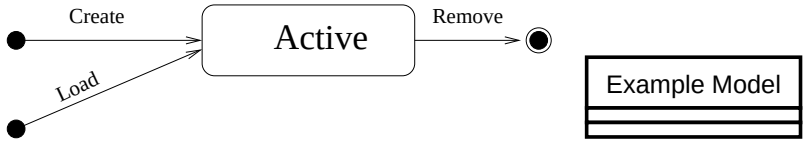


(a) Reference Behavior Diagram (b) Reference Model

Figure 5: The Reference Model class

A.6 Example Model

Example Model is a specialization of the class *Model*. It is used to store the data of a supplied example model for use in the analyzer. An *Example Model* is different from a *Reference Model* in that it contains no skeleton or measurements, and it is the target model for the *Reference Model* during the fitting process.

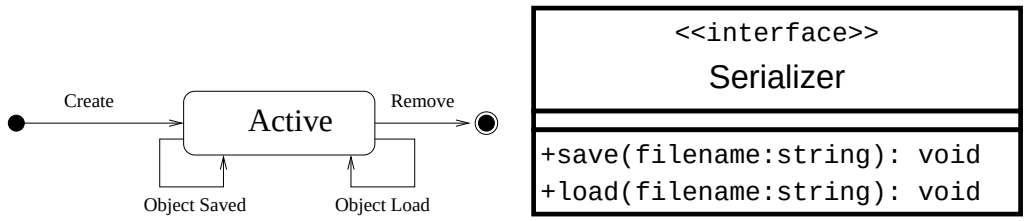


(a) The state diagram for the Example Model (b) The Model Class class

Figure 6: The State and Class diagram for Example Model

A.7 Serializer

The *Serializer* is an interface implemented by the *Analyzer* and *Model* classes. Implementing this interface enables object serialization. That is object can be saved and loaded from disk, thus data can be reused between sessions without converting to external formats. Classes implementing the *Serializer* interface can at any point choose to save (Serialize) an instance of itself to the disk, indicated by the "Object Saved" transition. Likewise if an object of the same type exists on disk, it can be loaded, thus overwriting any current states set in the object with the serialized values. The object on disk must be of the same type as the one in memory. The actions "Create" and "Removed" happens during creation and removal of the implementing class respectively.



(a) The state diagram for the Serializer class (b) The Serializer class

Figure 7: The State and Class diagram for the Serializer class

Appendix B

Source Code and Example data

The homepage for this project http://www.ofn.dk/pcg/ contains documentation for the source code, binaries and example data used in this and the previous paper. All renders and graphs can also be found at this page. A video showing the use of the synthesis process can also be found. The sourcecode documentation is provided by doxygen and is available as a downloadable PDF file and for online browsing.

Appendix C

Project Resume

The following is a short project resume included in order to fulfill the rules and regulations.

This report examines the area of procedural generated content, more specifically procedural generation of humanoid characters. The work presented extends upon an analysis of which methods are suitable for generating such models. The first chapter recaps the previous work, shortly presenting the concept of character modeling, and outlining the ideas behind principal components analysis and genetic algorithms in the context of generating humanoid models. The second chapter continues to present a systems definition and presents the possible uses of the system. Furthermore a design is presented which should fulfill the requirements of the system definition. Chapter three continues with the implementation, specifically discussing the methods used to achieve the goals. Principal Components Analysis is presented in detail along with a necessary prerequisite known as reference fitting. The classes using these methods are presented and specific implementation choices explained. Lastly the chapter presents the design of a fitness function for a genetic algorithm, this genetic algorithm is used to approximate a mapping between abstract annotations and concrete components of the analysis. Chapter four proceeds to present tests of the implemented methods, focusing on the performance and quality of the methods described in chapter three. The test results are discussed and it is concluded that the mapping class is insufficient given the current tests, and that the Householder method for finding eigenvectors is the best, as it is fast and accurate. The results from the reference fitting methods are satisfactory, as reasonable low detail models can be normalized. Chapter five, the last chapter, concludes upon the work presented and shortly lists a few areas of improvements to the application. The implemented application fulfills most of the technical goals laid out in the first chapter, however according to the current tests it fails to solve the problem of providing intuitive parameters to synthesize models. Furthermore there has been no examination of how well animations can be reused - however it is assumed that for models synthesized from the same database this should be the case. As far as the original goal of producing a production ready tool for graphics artists, the implementation still needs work and as mentioned the annotation mapping is faulty. However the idea of using principal components analysis and reference fitting are likely to be useful.

Bibliography

[BM07] Esben Bach and Andreas Madsen. Procedural character generation. Master's thesis, Aalborg University - Department of Computer Science, Janurary 2007. First Part.

[Bra] David Braben. Frontier developments ltd. http://www.frontier.co.uk. Online; accessed November 2006.

[BV99] Volker Blanz and Thomas Vetter. A morphable model for the synthesis of 3d faces. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 187–194, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[BV03] Volker Blanz and Thomas Vetter. Face recognition based on fitting a 3d morphable model. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 25(9), September 2003.

[Dav] Robert Davies. *NewMat C++ Matrix Class*, 10th edition. Available at http://robortnz.net.

[Ebe05] David H. Eberly. *3D Game Engine Architecture; Engineering real-time applications with Wild Magic*. Morgan Kaufmann Publishers, 2005.

[Fab04] Fabrausch. .kkrieger. http://www.theprodukkt.com/kkrieger, 2004.

[LCF00] J.P. Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: A unified approach to shape interpolation and skeleton-driven deformation. *SIGGRAPH*, 2000.

[MA] Maxis and Electronic Arts. Spore website. http://www.spore.com. Online; accessed November 2006.

[Max] Maxis. Spore editor presentation. http://www.spore.com/images/movies/spore_editor_hi.mov. Online; accessed November 2006.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, international edition, 1997.

[Moe01] Thomas B. Moeslund. Principal component analysis - an introduction. *Technical Report CVMT*, 2001.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[SMT03a] Hyewon Seo and Nadia Magnenat-Thalmann. An automatic modeling of human bodies from sizing parameters. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 19–26, New York, NY, USA, 2003. ACM Press.

[SMT03b] Hyewon Seo and Nadia Magnenat-Thalmann. An automatic modeling of human bodies from sizing parameters. In *I3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 19–26, New York, NY, USA, 2003. ACM Press.

[SMT04] Hyewon Seo and Nadia Magnenat-Thalmann. An example-based approach to human body manipulation. *Academic Press Professional*, 2004.

[Spu03] Francis Spufford. Masters of their universe. *The Guardian*, 2003. Online; accessed November 2006.

[Vis] Interactive Data Visualization. Speedtree. http://www.speedtree.com/. Online; accessed November 2006.

[Vol] Camilla Volkersen. Artist at pollux gamelabs. http://www.polluxgamelabs.com/. Personal Informational Contact.

[VT00] Pascal Volino and Nadia Magnenat Thalmann. Accurate collision response on polygonal meshes. In *CA '00: Proceedings of the Computer Animation*, page 154, Washington, DC, USA, 2000. IEEE Computer Society.

[vVB04] James M. van Verth and Lars M. Bishop. *Essential Mathematics for Games and Interactive Applications - A Programmer's Guide*. Elsevier, 2004. ISBN: 1-55860-863-X.

[WB64] White, Paul A. and Brown, Robert R. A comparison of methods for computing the eigenvalues and eigenvectors of a real symmetric matrix. *Mathematics of Computation*, 18(87):457–463, July 1964.

[Wik03] Wikipedia. Principal components analysis — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Principal_component_analysis&oldid=16060068, 2003. Online; accessed November 2006.

[Wik06] Wikipedia. Demoscene — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Demoscene&oldid=86511968, 2006. Online; accessed November 2006.

[Wik07] Wikipedia. Symmetric matrix — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Symmetric_matrix&oldid=124545480, 2007. Online; accessed May 2007.

[Wil] Mark Wilson. Next gen consoles held up by last gen disc speeds. http://www.gizmodo.com/gadgets/entertainment/next-gen-consoles-held-up-by-last-gen-disc-speeds-196976.php. Online; accessed December 2006.

[WRB86] J. H. Wilkinson, C. Rinsch, and F. L. Bauer. *Handbook for Automatic Computation: Linear Algebra (Grundlehren Der Mathematischen Wissenschaften, Vol 186)*. SpringerVerlag, 1986.

List of Figures

List of Tables