UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

# The Design of a Generic Signing Avatar Animation System

by

Jaco Fourie

*Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Engineering at the University of Stellenbosch*

Department of Mathematical Sciences (Computer Science)
University of Stellenbosch
Private Bag X1, 7602 Matieland, South Africa

Supervisors:

Dr. L. van Zijl    Prof. B. Herbst    Prof. P.J. Bakkes

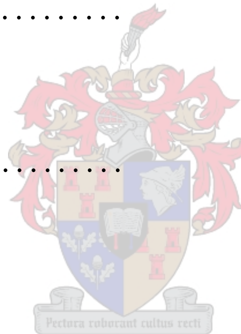
November 2006

# Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature: . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
J. Fourie

Date: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .



# Abstract

## The Design of a Generic Signing Avatar Animation System

J. Fourie

*Department of Mathematical Sciences (Computer Science)*
*University of Stellenbosch*
*Private Bag X1, 7602 Matieland, South Africa*

Thesis: MScEng (AM)

November 2006

We designed a generic avatar animator for use in sign language related projects. The animator is capable of animating any given avatar that is compliant with the H-Anim standard for humanoid animation. The system was designed with the South African Sign Language Machine Translation (SASL-MT) project in mind, but can easily be adapted to other sign language projects due to its generic design.

An avatar that is capable of accurately performing sign language gestures is a special kind of avatar and is referred to as a *signing avatar*. In this thesis we investigate the special characteristics of signing avatars and address the issue of finding a generic design for the animation of such an avatar.

# Samevatting

**Die Ontwerp van 'n Generiese Gebaretaalkarakter Animasiestelsel**

J. Fourie

*Departement Wiskundige Wetenskappe (Rekenaarwetenskap)*
*Universiteit van Stellenbosch*
*Privaatsak X1, 7602 Matieland, Suid-Afrika*

Tesis: MscIng (TW)

November 2006

Ons het 'n generiese karakteranimasiestelsel ontwikkel vir gebruik in gebaretaal verwante projekte. Die animasiestelsel het die vermoë om enige karaktermodel wat met die H-Anim standaard versoenbaar is, te animeer. Die animasiestelsel is ontwerp met die oog op gebruik in die *South African Sign Language Machine Translation* (SASL-MT) projek, maar kan maklik aangepas word vir ander gebaretaalprojekte te danke aan die generiese ontwerp.

'n Karaktermodel wat in staat is om gebare akkuraat te maak is 'n spesiale tipe karaktermodel wat bekend staan as 'n *gebaretaal avatar (Engels : signing avatar)*. In hierdie tesis ondersoek ons die spesiale eienskappe van 'n gebaretaal avatar en beskou die soektog na 'n generiese ontwerp vir die animering van so 'n karaktermodel.

# Acknowledgements

> ‘‘The rule is, jam tomorrow and jam yesterday -- but never jam today.’’
> ‘‘It MUST come sometimes to ‘jam-today,’’’ Alice objected.
> ‘‘No, it can’t,’’ said the queen. ‘‘It’s jam every other day: today isn’t any OTHER day, you know.’’
>
> – *Lewis Carol “Through the Looking-Glass”*

A project of this magnitude is never a one-man enterprise and many individuals deserve acknowledgement. I will start by thanking my mentor and study leader, Dr. Lynette van Zijl, for the hard work and patience without which this project would not have reached completion. I thank my parents for their assistance (both financially and emotionally) and encouragement that kept me going to the end.

I would also like to specially thank Dr. D. Cunningham and Prof. W. Straßer for the assistance and guidance they gave me during my research in Tübingen.

Most of all, I thank the Lord God for giving me the ability and for making this all possible.

Pectora roborant cultus recti

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not to be attributed to the NRF.

# Contents


**Declaration** i

**Abstract** ii

**Samevatting** iii

**Acknowledgements** iv

**Contents** v

**List of Figures** vii

**1 Introduction** 1

1.1 Thesis Outline . . . 2

**2 Literature Overview** 3

2.1 Signing Avatars . . . 4

2.2 Other Signing Avatar Projects . . . 5

2.2.1 ViSiCAST . . . 5

2.2.2 The Auslan Tuition System . . . 6

2.2.3 The SYNENNOESE Project . . . 7

2.2.4 Vcom3D Sign Smith Studio . . . 9

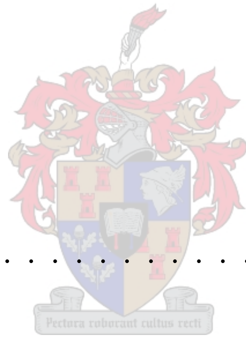
2.2.5 The Thetos Project . . . 9

2.3 Notation . . . 11

2.3.1 Stokoe . . . 11

2.3.2 Sutton SignWriting . . . 14

2.3.3 HamNoSys . . . 17



Pectora roborant cultus recti

2.3.4 The Nicene Notation . . . . . . . . . . . . . . . . . . . . . . 19
2.4 Avatar Animation Systems . . . . . . . . . . . . . . . . . . . . . . 20

**3 Design Issues 25**
3.1 Pluggable Avatars . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.1.1 The EAI Approach . . . . . . . . . . . . . . . . . . . . . . . 26
3.1.2 The VRML File Loader Approach . . . . . . . . . . . . . . . 28
3.2 Pluggable Input Notation . . . . . . . . . . . . . . . . . . . . . . 30
3.3 Generic Animator . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.4 Development Environment . . . . . . . . . . . . . . . . . . . . . 33

**4 Design and Implementation 35**
4.1 The Three-level Design . . . . . . . . . . . . . . . . . . . . . . . 36
4.2 The Parser . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.3 The Animator . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.4 The Renderer . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.5 Optimisations . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

**5 Results 52**
5.1 Issues Relating to Functionality . . . . . . . . . . . . . . . . . . . 52
5.1.1 Discrepancies between Real and Generated Gestures . . . 53
5.1.2 Generic Animation in Varying Situations . . . . . . . . . . 56
5.1.3 The Input Notation Design . . . . . . . . . . . . . . . . . . 57
5.1.4 Disadvantages of Euler Angles . . . . . . . . . . . . . . . . 58
5.1.5 Restrictions on Joint Rotations . . . . . . . . . . . . . . . . 59
5.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

**6 Conclusions 67**

**A The SignSTEP DTD 69**

**Bibliography 72**

# List of Figures

2.3.1 The phrase ”Don't know” in Stokoe notation [17]. . . . . . . . . . . . . 12
2.3.2 The ASL phrase “Don't know” [4]. . . . . . . . . . . . . . . . . . . . . . 13
2.3.3 The phrase “Don't know” in SignWriting notation. . . . . . . . . . . . 15
2.3.4 The sign for “difficult” transcribed with HamNoSys [23]. . . . . . . . 17
2.3.5 The ASL sign for “difficult” [23]. . . . . . . . . . . . . . . . . . . . . . 18
2.4.1 The H-Anim hierarchy of joints. Taken from [13]. . . . . . . . . . . . . 24

3.1.1 The VRML EAI interface. . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.2.1 A diagram of the proposed method to implement pluggable notations. 31

4.1.1 The generic signing avatar animator design. . . . . . . . . . . . . . . . 36
4.2.1 An example of nested queues. . . . . . . . . . . . . . . . . . . . . . . . 39
4.2.2 The parsing process. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.3.1 An example of a partial Java3D scene graph. . . . . . . . . . . . . . . 43
4.3.2 The animation action is added to the scene graph. . . . . . . . . . . . 44
4.5.1 Telemetry provided by the Netbeans profiler. . . . . . . . . . . . . . . 49
4.5.2 The amount of heap memory used as a function of time. . . . . . . . 49

5.1.1 The difference between *weight* and *maybe*. . . . . . . . . . . . . . . . . 54
5.1.2 A comparison of the “Thank you” gesture. . . . . . . . . . . . . . . . . 55
5.1.3 The SASL sign for “home” using two different avatars. . . . . . . . . 57
5.1.4 An example illustrating that rotation is not commutative. . . . . . . . 59
5.1.5 The neutral position. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
5.1.6 Rotation about the $x$-axis is possible. . . . . . . . . . . . . . . . . . . . 61
5.1.7 Rotation about the $y$-axis is possible. . . . . . . . . . . . . . . . . . . . 62
5.1.8 Rotation about the $z$-axis is not possible. . . . . . . . . . . . . . . . . . 62

Pectora roborant cultus recti

5.1.9 Now, rotation about the $z$-axis is possible. . . . . . . . . . . . . . . . . . . 63
5.2.1 A screen shot from the sign editor tool. . . . . . . . . . . . . . . . . . . . 66

# Chapter 1

# Introduction

```
Science is built up with facts, as a house is
with stones. But a collection of facts is no more
science than a heap of stones is a house.
```

*– J.H. Poincaré*

The South African Sign Language Machine Translation project (SASL-MT), is an ongoing project at the Department of Computer Science at the University of Stellenbosch [35]. The aim of the project is to develop a prototype system that can translate English text into South African Sign Language.

The objective of this study is the design and implementation of the signing avatar animation system that forms part of the SASL-MT project. The signing avatar system forms the back-end of the SASL-MT project and receives as input the sign language gestures that were translated from English text. The gestures are received in a textual representation called an interlingual notation [14, 15, 25]. The system then uses the gestures to animate a realistic anthropomorphic model also known as an *avatar*.

Previous machine translation systems for sign language can be found in current literature [12, 14] and we will discuss these in more detail in chapter 2. A common characteristic shared by these existing systems is that the avatar animation system cannot function independently from the rest of the encompassing ma-

chine translation system. Typically, this means that the system can only animate specific custom built avatars. The gestures that it can animate are also limited by the specific sign language that is translated to by the encompassing system.

In this study we propose a design for a generic signing avatar animation system. The system is not constrained by limiting it to specific sign languages or to custom built avatars. The system also functions completely independently from the translation system of which it forms a part. An implementation of this design is also provided and the results thereof evaluated.

## 1.1 Thesis Outline

Chapter 2 is a overview of the applicable literature. Previous signing avatar systems are discussed and evaluated. Other machine translation projects for sign language are also evaluated.

Chapter 3 investigates the design issues that influenced the design of the signing avatar system. The impact that the choice of development environment has, is explored. We also explain the need for pluggable avatars and input scripts. The design of the generic animator, the module that is central to the system, is also discussed.

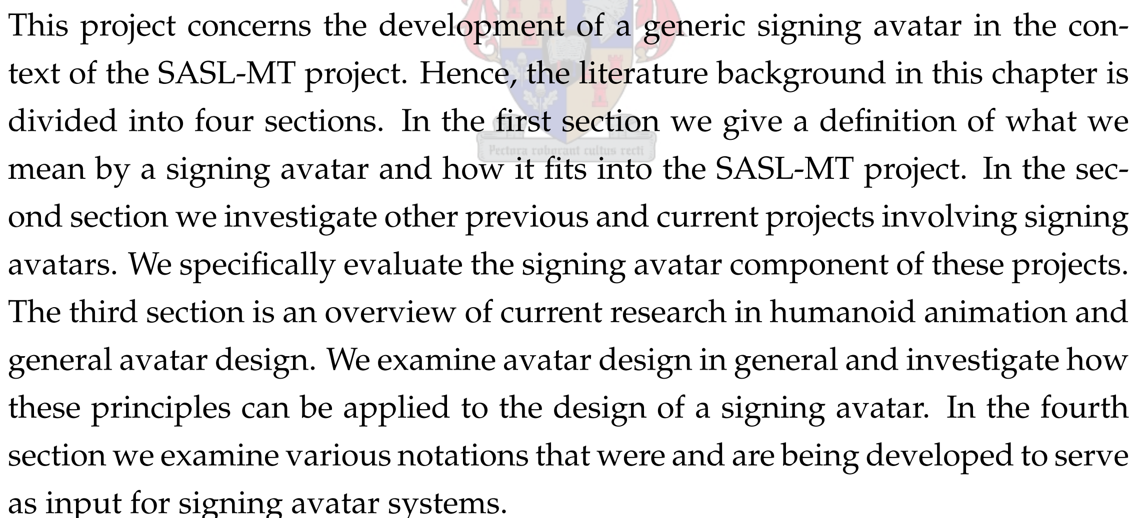
In chapter 4 we look at the implementation of the system that was discussed in chapter three. The system is divided into three separate modules: the parser, the animator and the renderer. These three modules function independently from each other and communicate using certain interfaces that will be discussed. Possible optimisations to the system are also investigated.

In chapter 5 the results of the system are evaluated. We examine the level of pluggability in both the avatar and the input script. The most important result is realistic animations that equate to recognisable sign language gestures. The recognisability of these results is evaluated experimentally.

# Chapter 2

# Literature Overview

> `Knowledge is of two kinds: We know a subject ourselves, or we know where we can find information about it.`
>
> – *Samuel Johnson, 1775*



This project concerns the development of a generic signing avatar in the context of the SASL-MT project. Hence, the literature background in this chapter is divided into four sections. In the first section we give a definition of what we mean by a signing avatar and how it fits into the SASL-MT project. In the second section we investigate other previous and current projects involving signing avatars. We specifically evaluate the signing avatar component of these projects. The third section is an overview of current research in humanoid animation and general avatar design. We examine avatar design in general and investigate how these principles can be applied to the design of a signing avatar. In the fourth section we examine various notations that were and are being developed to serve as input for signing avatar systems.

## 2.1 Signing Avatars

The term avatar, in our context, refers to the computer modelled representation of a human being. Research into virtual reality human modelling and animation has progressed greatly in terms of accuracy. It is now possible to model and animate a virtual human being accurately enough so that gestures as subtle as a change in facial expression can be noticed and understood by a real human observer [12]. As we will see, this is of particular interest to the development of signing avatars.

A signing avatar is used to communicate sign language gestures. Therefore, it requires the ability to accurately reproduce any movements that a human signer can perform. The movements that make up sign language gestures consist of hand, arm and body movements combined with facial expressions [12, 25]. We call these movements *sign language gestures* and divide them into two separate categories called manual and non-manual gestures. The manual gestures are those that are performed using the arms, hands and fingers, while the non-manual gestures are those performed using facial expressions and other body movements. Sign language gestures are fine motor movements, such as one would find in high budget animated films. Gross motor movement, such as one normally finds in computer games and internet 3D chat-rooms, would be insufficient for accurate recognition of gestures. Fine motor movements require a higher level of articulation in the avatar. Higher levels of articulation allow for more complex movements but are more difficult to animate. A signing avatar would need a high level of articulation and a sufficiently advanced animation system to control its movements.

Pectora roborant cultus recti

In the section that follows we take a critical look at previous signing avatar projects. We investigate important design decisions like choice of notation, computational animation models, avatar structural models and implementation platform.

## 2.2 Other Signing Avatar Projects

In the following five sections we evaluate and discuss five systems that make use of a signing avatar to create sign language gestures. In most of these systems the avatar animation is only a small component in a larger machine translation project, but for the purposes of this study we concentrate on the avatar animation component itself. The five systems that we discuss are: the VisiCAST Translator, the Auslan Tuition System, the SYNENNOESE project, the Thetos project and the Vcom3D Sign Smith Studio.

### 2.2.1 ViSiCAST

The VisiCAST translator was created as part of the European Union's VisiCAST project at the University of East Anglia [12]. Its aim was to translate English text into British Sign Language (BSL) and serve as a research vehicle for translation to German and Dutch Sign Language.

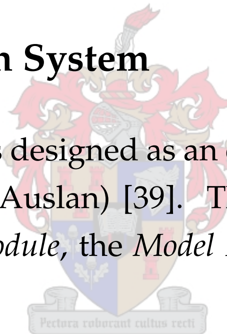
Before the English text is translated into BSL, it is first translated into an interlingual notation called SiGML (Signing Gesture Markup Language) [15]. SiGML is then translated into animations that represent the BSL gestures. SiGML is an XML encoded version of the HamNoSys notation (see section 2.3.3) for describing sign language gestures. The SiGML gestures are then sent to the *Animgen animation synthesiser* which is the animation engine for the VisiCAST translator.

Since Animgen is part of a commercial project, the details of the design are not publicly released. To the author's knowledge, for each frame in the animation sequence, the rotation for each joint in the avatar is computed. These frame-by-frame rotation values are then applied to the avatar and rendered onto the screen. Due to the lack of sufficient facial joints for accurate expressions, the avatar used is not H-Anim compliant (see section 2.4 for more on the H-Anim standard). For prototyping, Animgen can also generate animations in the form of a VRML [18] avatar that is H-Anim compliant, but which lacks accurate facial features.

In conclusion, the relevance that the ViSiCAST translator has to the avatar ani-

mation system proposed by this thesis is found in the concept of the interlingual notation. The interlingual notation divides the VisiCAST translator into a component that translates English into SiGML and a component that translates SiGML into animated gestures. The advantage of having separate modules is that the module that translates from SiGML to animated gestures does not need to be changed if the system is modified to translate from other oral languages. For example, if the VisiCAST translator is modified to translate German into BSL, only the module that translates into SiGML needs to be replaced. It is exactly this kind of module based, generic design that we aim for in this thesis. However, this strategy is only as generic as the interlingual notation itself. The notation needs to be tested by transcribing a variety of different sign languages to ensure that it is capable of representing any gesture. This aspect of notation will be discussed further in section 2.3 when we investigate different notations including SiGML.

### 2.2.2 The Auslan Tuition System



The Auslan Tuition System was designed as an educational tool for the teaching of Australian Sign Language (Auslan) [39]. The system is divided into three parts: the *Human Modelling Module*, the *Model Rendering Module* and the *Model Interpolation Module*.

The Human Modelling Module is responsible for the creation and rendering of the humanoid model. The model itself is defined using a proprietary XML format. A purely rotational hierarchical kinematic tree is built from the model information. This kinematic tree is similar to the scene graph structure found in VRML and scene graph based 3D graphics libraries such as Java3D. The nodes of the kinematic tree represent joints in the humanoid. In this way a structure similar to the joint hierarchy defined in the H-Anim standard is formed.

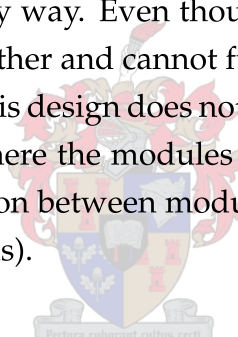
The Rendering Module is responsible for displaying the avatar graphically. It accomplishes this task by using the OpenGL 3D graphics library [19]. The kinematic tree is traversed and each node in the tree is associated with a polygonal model representing the surface of the body segment that follows that node. Each

segment is rendered as it is encountered in the tree using OpenGL polygonal primitives.

Smooth and visually pleasing animation is created by the Model Interpolation Module. Since most of the animation tree is purely rotational, effective interpolation between rotations is all that is needed for smooth animation.

The Auslan Tuition System is the only system that we discuss that does not make use of an interlingual notation. The system was not designed to be generic but rather to be fast and effective in the visualisation of Auslan. The avatar was custom designed for use only in this system and is not user customisable. Since there is no clear intermediate step between the animation of the avatar and the rest of the system, it would take significant modification to adapt the system to other sign languages.

In conclusion, the Auslan Tuition System is different from the other systems that we discuss in almost every way. Even though the design is module based, the modules depend on each other and cannot function independently from the other modules. As was seen, this design does not lead to a generic system. In this thesis we propose a design where the modules are kept independent from one another by restricting interaction between modules to clearly defined interfaces (see chapter 4 for further details).




### 2.2.3 The SYNENNOESE Project

The SYNENNOESE project is a Greek national project addressed to Greek Deaf pupils in primary schools [3]. The aim is to create a Greek Sign Language signing avatar as an educational tool for early primary school pupils.

The designers of the project chose the STEP [11] language (Scripting Technology for Embodied Persona) as an interlingual notation to interact with the avatar. As we saw in the VisiCAST translator, translation to the interlingual notation and animation of the avatar are two separate steps in the project. Unlike SiGML, STEP does not describe gestures by using a pre-defined set of hand shapes, but rather uses more generic movements. STEP was designed to describe any type

of human motion by representing the motion as a collection of joint translations and rotations. The STEP notation is not as compact as SiGML, but can describe almost any possible human motion and thus supports all sign languages equally well.

Rendering of the avatar is done in two different ways. The first method implemented was to use an H-Anim compliant VRML avatar and a VRML browser. The animation is embedded as JavaScript code in the VRML avatar and is displayed using readily available VRML browsers. One advantage of this approach is that the animated result can easily be made into a standard HTML page and published on the internet. In this way the system is immediately accessible to any user with a VRML enabled browser.

Another advantage of the first rendering method is that any H-Anim compatible avatar can be embedded with the JavaScript animations and can be rendered in exactly the same way. One must realise, however, that the end user that wants to use his own H-Anim avatar would not be able to use his custom avatar before the JavaScript modifications to the avatar have been done. Usually, these modifications cannot be done by the user and needs to be done by the programmer. This effectively limits the user to use only the avatars that the system provides.

The second method that the SYNENNOESE project used to render the avatar is by modelling the avatar using the MPEG-4 SNHC standard [22]. Unlike H-Anim that only provides for basic expressions, the MPEG-4 SNHC standard fully supports facial animation. The MPEG-4 SNHC standard is a set of body and facial animation parameters that can be used to animate avatars using body animation parameter (BAP) and facial animation parameter (FAP) players. By using BAP and FAP players, H-Anim compatible avatars can also be animated. However, to fully take advantage of the advanced facial animation supported by the MPEG-4 SNHC standard, the facial model of the H-Anim avatar needs to be altered. The advantage gained by supporting a pure H-Anim model is that a user of the system can now use his custom built H-Anim avatar without having to change it in any way. This enables the system to be easily used in a variety of applications. Note that this advantage is gained at the expense of more realistic facial animations gained from the altered H-Anim model.

The SYNENNOESE system was designed to be more generic than most other systems that we investigated. The use of the STEP notation as interlingual notation makes the animation system more generic, since any sign language that can be represented in STEP notation can be animated using this system. The avatar renderer is also designed generically as the animation can be applied to any H-Anim avatar with little or no modification. Rendering of the animations are only restricted by the capabilities of the VRML browser or the MPEG-4 player. A system that is as generic in its design as this one, but without the need for BAP/FAP players or VRML browser plugins, is what we are aiming for in this study.

### 2.2.4 Vcom3D Sign Smith Studio

The Vcom3D Sign Smith Studio was developed as an authoring tool for creating multimedia that incorporate sign language gestures [36]. Since this is a commercial product, no design or implementation details are available. Instead, we discuss the features of the product and evaluate it purely on its ability to accurately render sign language gestures.

The software allows the user to construct sign language gestures using a library of 2000 hand gestures and facial expressions. The user can then export his gestures as video files that can be played back without the need of the software. Sign Smith Studio also gives the option of exporting gestures as animated VRML models that can be embedded in a HTML web page and viewed using a VRML compliant web browser.

The animations are smooth and accurate and the authoring interface is intuitive and easy to use. The user can choose between twelve different avatars to sign the constructed gestures, but the system does not support custom made avatar models. This is the only disadvantage to an otherwise well designed system.

### 2.2.5 The Thetos Project

The purpose of the Thetos project was to improve the social integration of the Polish Deaf Community into the larger community of Polish speaking people [6].

The project was designed to translate Polish text into Polish Sign Language gestures.

Similar to the previous translation systems that were discussed, the designers of the Thetos translator chose to separate their system into two components. The first component performs a full linguistic analysis of the textual input and translates it to an interlingual notation. The interlingual notation used in this system is the Szczepankowski's gestographic notation [6]. The Szczepankowski's gestographic notation was designed to be easy to read and understand. It was designed this way in order to simplify the task of compiling the gesture dictionary. At the same time it is also descriptive enough that all gestures can be accurately defined.

The second component is responsible for the animation of the avatar. It accomplishes this task by interpreting the gesture notation and generating from it the key frames that make up the finished animation. The key frames specify static configurations of the avatar, together with the time intervals needed to pass from one configuration to the next. Each configuration is defined as a set of joint angles that is similar to the angles specified in the H-Anim standard. Smooth motion is achieved by interpolating all the rotation angles in time. In this way intermediate key frames are generated and the motion appears smoother.

For the purposes of this study, the most interesting component of the Thetos system is the interlingual notation that is used. The notation is described as being a trade-off between the SignWriting notation (see section 2.3.2) that is simple to use by humans, and the HamNoSys/SiGML notation (see section 2.3.3) that is detailed and easy to parse by computer. This notation would be a practical choice for the input notation of our system.

The choice of notation is important and affects both the interface to the system and the ability of the system to animate a variety of sign languages. In the section that follows we will further investigate interlingual notations and the effect that the choice of notation has on an avatar animation system.

## 2.3 Notation

In order to modularise and simplify the task of machine translation to sign language, most developers of a machine translation system have proposed a written form of sign language. One must realise that such abstractions of real sign language will cause some detail of the language to be lost [12]. Deciding which detail can acceptably be sacrificed is of cardinal importance and makes the design of the script notation a critical step.

Sign language has no standard written form but has its own unique grammar. This means that a system that aims to translate a spoken language like English into sign language, will need a computational linguistic component to generate a script that controls the movement of the avatar. The best notation for such a script is still an open area of research.

The notations discussed in this section are all known as interlingual notations. By interlingual notation we refer to the notation that is used to describe a sign language phrase [25]. This notation will be interpreted by a computer program, which will then generate the appropriate animations. In the rest of this section we investigate four different interlingual notations, namely, the Stokoe notation, Sutton SignWriting, HamNoSys and the Nicene notation.

### 2.3.1 Stokoe

The Stokoe notation [26] was one of the earliest (1976) description methods for sign language. It was designed by the linguist Dr. William Stokoe with the intention of having a written representation of sign language to aid in linguistic research on sign language.

The Stokoe notation is a written form of sign language that consists of a combination of Roman letters and invented symbols. It divides all gestures into four parts: a hand shape, a movement, a place of articulation and an orientation. The gesture is written in a near-linear fashion with the place of articulation indicated first, followed by the hand shape, orientation and any movement indicators. An example of the American Sign Language (ASL) phrase “Don’t know” tran-

scribed in Stokoe notation can be seen in figure 2.3.1. The figure clearly points out the four parts of the pictograph.

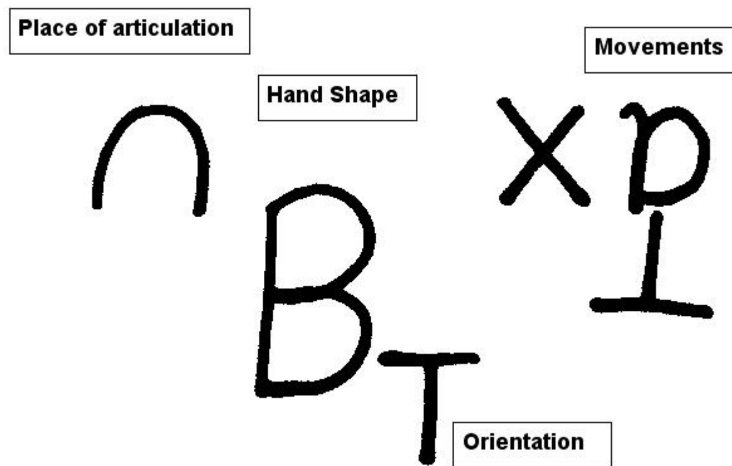



**Figure 2.3.1:** The phrase "Don't know" in Stokoe notation [17].

The first part of the pictograph is the place of articulation. It indicates where in the signing space[1] the gesture should be "articulated". In the example in figure 2.3.1, the place of articulation indicates that the gestures should be articulated over the area in front of the forehead. The second part of the pictograph is the hand shape. The Stokoe notation uses hand shapes based on the international one-handed finger spelling hand shapes. In our example the $B_T$ glyph[2] indicates that the international one-handed finger spelling hand shape for the letter "B" should be used. The subscripted "T" indicates the orientation of the hand. The final part of the pictograph is the movement indicators. In our example there are three separate movement indicators. The "X" indicates that contact is made. The glyph that looks almost like the letter "D" indicates that movement is directed away from the signer. The $\perp$ glyph shows that the palm is turned down as movement is made away from the signer. Figure 2.3.2 on page 13 shows an illustration of this sign.

[1]The signing space is the space in front of the signer where sign language gestures are performed or articulated. It extends in an arc from the left of the signer to the right and reaches from the top of the head to the waist [4].

[2]By *glyph* we mean a character or symbol that is used to represent a movement,

**Figure 2.3.2:** The ASL phrase "Don't know" [4].

Stokoe notation lacks completeness, since its set of possible hand shapes is not sufficient to describe all the gestures found in sign languages [25]. The system also lacks finger orientation information as well as information on non-manual signs like facial expression and movement of the shoulders. Smith and Edmondson [25] showed that the movements are too vague to be accurately reproduced by a computer, given only the information captured by the notation.

In conclusion we see that the Stokoe notation would not be a practical choice for our interlingual notation. To be used as a computational base for our system, the interlingual notation needs to be well defined and should never be vague or incomplete. Stokoe provided a written form for sign language and showed that sign language is not just a signed version of English or "pictures in the air" but was structured like any other human language with its own unique grammar. The Stokoe notation forms a basis for two of the notations that we investigate in the sections that follow and clearly shows the complexity and non-triviality of

orientation or hand shape in a particular notation.

the design of such a notation.

### 2.3.2 Sutton SignWriting

Sutton SignWriting was invented by the dancer and movement notator Valerie Sutton [29]. The notation was developed when the University of Copenhagen asked her to adapt her notation for recording dance steps. They wanted a notation to record sign language gestures for linguistic research. Her notation was called DanceWriting and was the predecessor to what later became SignWriting.

SignWriting takes a unique approach to the recording of sign language gestures. Unlike most other notations such as Stokoe and HamNoSys (see section 2.3.3), it does not use of a set of pre-determined hand shapes. Instead it uses the schematic "see and draw" approach where the only goal is to record movement without even needing to know that language is being recorded.

We use figure 2.3.3 on page 15 to illustrate the way gestures are recorded using the SignWriting notation. Once again, the gesture is divided into four logical parts namely: hand shape, movement, location and orientation.

Instead of the normal taxonomic approach to hand shape definition, SignWriting uses a schematic or pictorial approach. The taxonomic approach would define a finite number of hand shapes and assign an arbitrary symbol for each hand shape. We saw this approach in the Stokoe notation where Roman characters were used to represent hand shapes. The problem with this approach is that all possible hand shapes have to be matched to one of the symbols in this finite set, even if some of them do not match exactly. The schematic approach of SignWriting solves this problem by defining hand shapes as schematic diagrams where each part of the hand is represented independently. In the example of figure 2.3.3 on page 15, the hand shape refers to a straight outstretched palm with all the fingers straight and pointing in the same direction.

Orientation is indicated by taking advantage of the fact that the back of the hand is darker than the palm. If the hand shape is coloured in black it indicates that the back of the hand is turned towards the signer. If the hand shape is not coloured

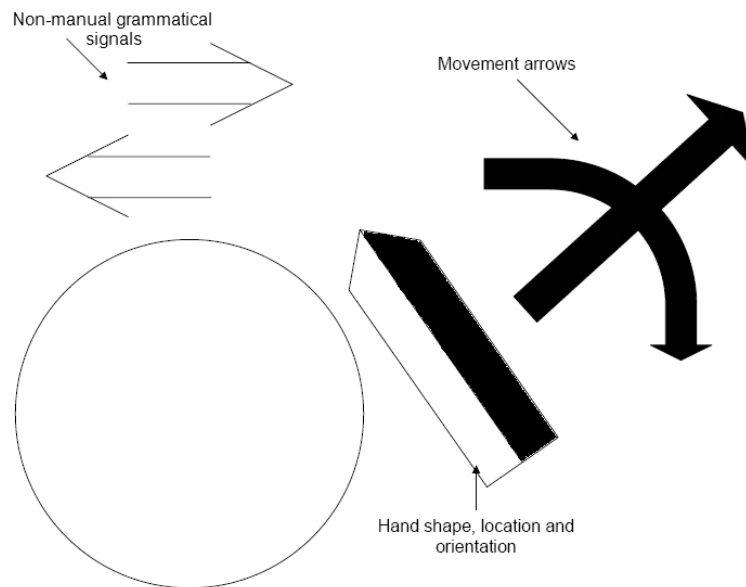



**Figure 2.3.3:** The phrase "Don't know" in SignWriting notation.

in and left white, it indicates that the palm of the hand is turned towards the signer. If the hand shape is half coloured it indicates that the hand is turned half way between the previous two positions. Orientation is also indicated by the fact that the symbols can be rotated to point in any direction. In this way the hand shape can be rotated to point in a specific direction with relation to the signer's head. This intuitive approach simplifies reading the signs. In our example, the half coloured hand shape indicates that the side of the hand is turned towards the signer. Notice that the hand shape is also rotated to indicate the direction that the fingers are pointing.

The visual approach of SignWriting is most clearly seen in the way it indicates location. It has no arbitrary symbols for location as other notations have. Instead of the characters being written linearly from left to right, the characters are written in whatever relationship they actually take in the sign. If the hands appear on top of each other in the actual sign, the hand shapes are written down one underneath the other. In this way one can say that the symbol for location in SignWriting is the image itself. In our example, the large circle indicates the signer's head. The hand shape is therefore located at the top-right side of the signer's head.

SignWriting uses arrow symbols to indicate movement. Just as with location, SignWriting takes advantage of the spatial arrangement of the symbols to add detail. Arrows are rotated to indicate the path that the movement should take and arrows that indicate circular motion are curved in the appropriate direction. Complex movement like looping is indicated by curving the tail of the arrow in on itself. If vertical movement instead of forward-backward movement is to be indicated, the tail of the arrow doubles. The arrowhead also changes to indicate whether the movement should be done with the left, right or both hands. The arrows shown in the example of figure 2.3.3 indicate that the hand rotates outward as it moves away from the signer.

The last aspect that we will discuss on the SignWriting notation is non-manual grammatical signals. These non-manual gestures are important to the meaning of sign language adverbs, relative clauses and other important grammatical constructs. Accurate representation of these constructs would not be possible without non-manual gestures. The non-manual grammatical signals consist mostly of facial expression but also includes the movement of the shoulders, head and body. The two double tail arrows of our example in figure 2.3.3 is a non-manual gesture indicated by a sideways head shake.

The graphical nature of SignWriting makes it a practical notation for easy human understanding and reading. However, it is this graphical nature that also makes the notation difficult to parse with a computer. The notation will need to be adapted to use only standard ASCII characters or be encoded in a markup language such as XML to be practical for computer animation. Such an adaptation has been done and the result is SWML (SignWriting Markup Language) [29]. SWML is a XML version of the SignWriting notation and was designed with the digitisation of SignWriting in mind. SWML would be a practical choice for our input notation. However, the author that has to transcribe gestures in this notation would still require knowledge of the SignWriting notation since SWML is merely an XML adaptation of SignWriting.

### 2.3.3 HamNoSys

HamNoSys was developed by researchers at the University of Hamburg in Germany [15, 23]. It was designed to be used by sign language researchers to record sign language gestures.

Like most other sign language notations, HamNoSys uses hand shape, position, orientation and movement for the description of gestures. Just like the Stokoe notation, a sign is transcribed linearly from left to right as can be seen in the example of figure 2.3.4. An illustration of this sign can be seen in figure 2.3.5.

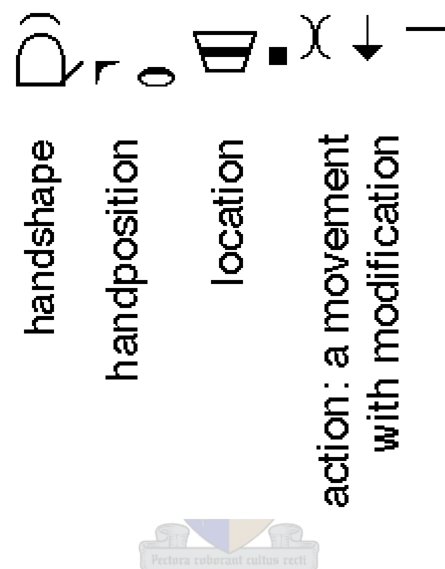



**Figure 2.3.4:** The sign for "difficult" transcribed with HamNoSys [23].

There are twelve standard hand shapes in the HamNoSys notation. These standard hand shapes can be modified by bending or moving individual fingers for more complicated hand shapes. This iconic approach is similar to what we saw in SignWriting but is not nearly as customisable.

Location is described by defining a set number of positions on and around the human body. The position indicators also contain information on the distance from the specified position where the sign should be articulated. Several hundred positions on the human body have been defined in HamNoSys.

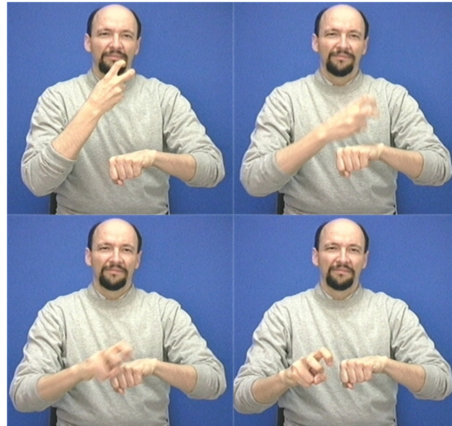


**Figure 2.3.5:** The ASL sign for "difficult" [23].

Orientation is specified in two ways. Firstly, "extended finger direction" refers to the direction the fingers would be pointing if they were straight. The twenty-six possible values of "extended finger direction" have been defined as the directions from the centre of a cube to its face centres, edge midpoints and vertices. The second way orientation is specified, is with palm orientation. Palm orientation can be one of eight values corresponding to the directions from the centre of a square to its edge midpoints and vertices.

The movement descriptions used in HamNoSys are varied and can take many forms. Movement through space can be in a straight line, curved, circular or directed to a specific location or body position. Straight line movement can be in any of the twenty-six directions defined for "extended finger direction" and is indicated by an arrow pointing in the applicable direction, for example (→). Curved movement is indicated by an arc following the movement arrow and can be oriented in any of the eight directions defined for palm orientation. Wavy or zigzag movement is indicated by wavy arrows (⇝) and circle arrows (↻). Other possible movements include wrist oscillation about three different axes and movement called "fingerplay" where fingers are waggled as if crumbling something between the fingers and thumb [14]. Further possibilities are found by combining these movements sequentially or in parallel.

Non-manual grammatical signals are not implicitly supported by HamNoSys.

Pectora roborant cultus recti

Its main scope is arm and finger movement which it describes well. Head movement is the only non-manual grammatical signal supported and is indicated by a circle (◯), representing the head, preceding the movement indicators.

HamNoSys is used as an interlingual notation for the VisiCAST project [15]. Since the notation uses special characters that are not easily parsed with a computer, SiGML was defined [14] (see section 2.2.1). SiGML (Signing Gesture Markup Language) is an XML encoded version of HamNoSys and contains exactly the same amount of information as HamNoSys. A tool used to translate HamNoSys into SiGML has been developed and is used to translate all HamNoSys transcriptions to SiGML prior to insertion into the VisiCAST system. This approach has the advantage of having both a notation that is easily understandable by human readers and one easily parsed by computers. The only disadvantage is the extra step of translation that is needed.

### 2.3.4 The Nicene Notation

The Nicene notation was developed by Smith and Edmondson [25]. It was designed specifically with computer representation in mind. This notation does not suffer from the problems that the previous notations suffered from, as it is not based on hand shapes and is designed in such a way as to be completely general and able to describe almost any gesture. The notation is divided into three layers: thought, word and deed.

The first layer, called the thought layer, provides a rich description of the sign on a high level. It is composed of six vectors: two for the hands, two for the arms, one for the face and one for the movement of the sign. The two hand configuration vectors are defined using the Stokoe notation, as it is accurate enough for this high level description and is easy to use. The arm configuration vectors contain information on wrist, elbow and shoulder positions. It also defines any points of contact from the hands. The non-manual features of the sign is covered by the face vector and includes information on lip, mouth, tongue, eye, cheeks and nose movement. The last vector is the movement vector that describes the trajectory of the hands and arms through the configurations described by the

other vectors. This vector also contains information on wiggling, repetition of movement or any contact made during these movements.

The second layer is called the word layer and takes the anatomical vectors from the thought layer and converts them to matrix form with numerical parameters. The matrix parameters define angles for every joint of the five fingers of the hands. The arm vectors are also translated into joint-angle matrices with three angles for the shoulder, two angles for the elbow and two angles for the wrist. The movement vector is composed of one or more velocity vectors which are in turn composed of a speed and a direction vector. The direction vector is a three dimensional vector that describes the path that the hands will follow. The vector retains any information on repetition of movement and possible points of contact.

The final layer, called the deed layer, ensures smooth transition between subsequent signs by bringing in the temporal aspects of the sign. It links the trajectories and target positions by attaching them to points along a timeline and using the movement vectors from the previous layer to link between targets. Computer interpolation is used were movement is unspecified.

Like the SignWriting notation, the Nicene notation was designed to be generic. It was designed to be able to represent any sign language gesture from any sign language.

Pectora roborant cultus recti

In summery, a generic notation is needed if we are to design a generic signing avatar system. The notation also needs to be easy to parse using a computer. It is exactly for this reason that SWML was developed as a computer readable version of the SignWriting notation.

## 2.4 Avatar Animation Systems

There is an ongoing interest in the development and realistic animation of humanoid avatars [33]. Applications for research into humanoid avatar animation include entertainment, computer graphics and multimedia communication [33]. More sophisticated avatars are used in military applications where avatars are

used in realistic battle simulations [2]. Realistic avatar animation have also been developed to simulate the movements of athletes [9].

We start our discussion on avatar animation systems by investigating the most general avatar animation systems. We do this in order to find the most generic solutions and the issues that are common to all avatar animation systems in general.

Yang, Petrui and Whalen proposed an hierarchical control system for the animation of avatars [38]. In their control system, they use avatars that are compatible with the H-Anim[13] standard for humanoid animation (see figure 2.4.1 on page 24 for the H-Anim skeleton). The control hierarchy is a three-tier system with the lowest layer responsible for joint and segment movements and the highest layer interpreting the storyboard-based behaviour script.

In the lowest layer of the hierarchy, the control system directly manipulates the humanoid joints defined by the H-Anim standard. The H-Anim standard defines a humanoid as a collection of predefined *joints* and *segments*. According to this standard individual humanoids differ only in the shape of their segments and the position of their joints. No assumptions are made about the avatar appearance or the type of application in which it is used. The appearance of the avatar is determined by the texture that is applied to the segments and is not defined in the standard. This means that a generic avatar animation system can be created by simply making sure that the system animation is based on the H-Anim standard.

The second layer of the hierarchy defines basic actions (such as run, jump and walk) using the joint movements defined in the lower layer. These basic actions are then combined together in a storyboard in the highest layer of the hierarchy using some user-driven behaviour script. The avatars themselves are defined in the VRML language. VRML is a well-known language used for describing interactive 3D objects [33]. VRML is not a programming language and can only define simple behaviours. This makes VRML an impractical language for the implementation of high-level complex animations. Yang *et al.* proposed two means of solving this problem.

One approach is to use the external authoring interface (EAI) to communicate with the VRML nodes using a high level programming language like Java to write the animations. For every joint, defined as VRML nodes in the model, a Java animation script is written that runs in its own thread, waiting for input from the master animation system. The master animation system is written in Java and controls the H-Anim joints through the EAI Java animation scripts.

This approach is cumbersome in many ways. With this approach the user interface to the system is a VRML web browser. The VRML browser loads the VRML avatar, that in turn loads the Java animation scripts through the EAI. This step already compromises the generality of the animation system, since the avatar needs to be *injected*[3] with the Java script nodes that represent the rest of the animation system. Also, since each joint is controlled separately in its own thread by the applicable Java script class, perfect timing in the joint movements are needed for smooth and realistic looking animations. The overhead that the slow interface between Java and VRML creates in rendering time, causes the system to slow down considerably and makes real-time animation impossible without high-end dedicated graphics processing [38].

The second approach that Yang suggested is to use the Java3D API [28] and a VRML *file loader*. Java3D is a high level scene graph-based 3D API that runs on Java and seamlessly integrates with a Java animation system. In this approach the VRML objects are first loaded using a VRML file loader. Yang *et al.* used the CyberVRML97 [30] loader. The loader converts the VRML object into a Java3D scene graph that can be rendered without the need for a VRML browser using Java and Java3D. This means that the model, and the animation system that animates it, are now seamlessly combined into one environment. With this approach the joint controller can communicate directly with the H-Anim joints without communication overhead. Generality is not lost, since any VRML file can be loaded using the file loader, and once the model is converted into a Java3D scene graph the animation system works the same way for all avatars. This approach is therefore more practical in our situation and is discussed further in chapter 4.

[3]We use the term *inject* to refer to the action of adding references of the corresponding Java animation scripts to the various joints in the VRML model.

Whalen [20] conducted research into the effective animation of Java3D H-Anim models. A Java3D model representation is a directed acyclic graph (DAG) of geometry and control nodes. To animate such a model, one simply inserts movement nodes at the appropriate place into the graph. In the system that Whalen describes, Java3D motion interpolaters are used to smooth the animation. These interpolation nodes are inserted with the movement nodes (called `Transform` nodes in Java3D) at the appropriate place in the scene graph. In this case the appropriate place is the parent node of the H-Anim joint that is to be animated. The same child-parent relationship that VRML nodes have in the H-Anim standard is also present in the Java3D implementation. Just as with a VRML scene graph, transforms done on a parent node also affects all the children of that parent in a Java3D scene graph. This is an important feature of the scene graph structure and allows one to easily translate from the H-Anim joint hierarchy to the corresponding scene graph in Java3D.

In conclusion, Yang, Petrui and Whalen showed that a generic animation system can be constructed using the H-Anim standard. If the assumption can be made that the joints specified by the standard are always present, animations can be performed on any avatar. An efficient animation strategy was also proposed using a file loader and the Java3D API. This animation strategy can be combined with Whalen's independent research on effective animation in Java3D using motion interpolaters. This thesis discusses exactly such a system and further details can be found in chapters 3 and 4.

Pectora roborant cultus recti

In the next section the design issues that were specific to the design of a generic signing avatar system for the SASL-MT project will be discussed. These issues will be resolved by using the techniques that were discussed in the investigation of the signing avatar systems presented in this chapter.

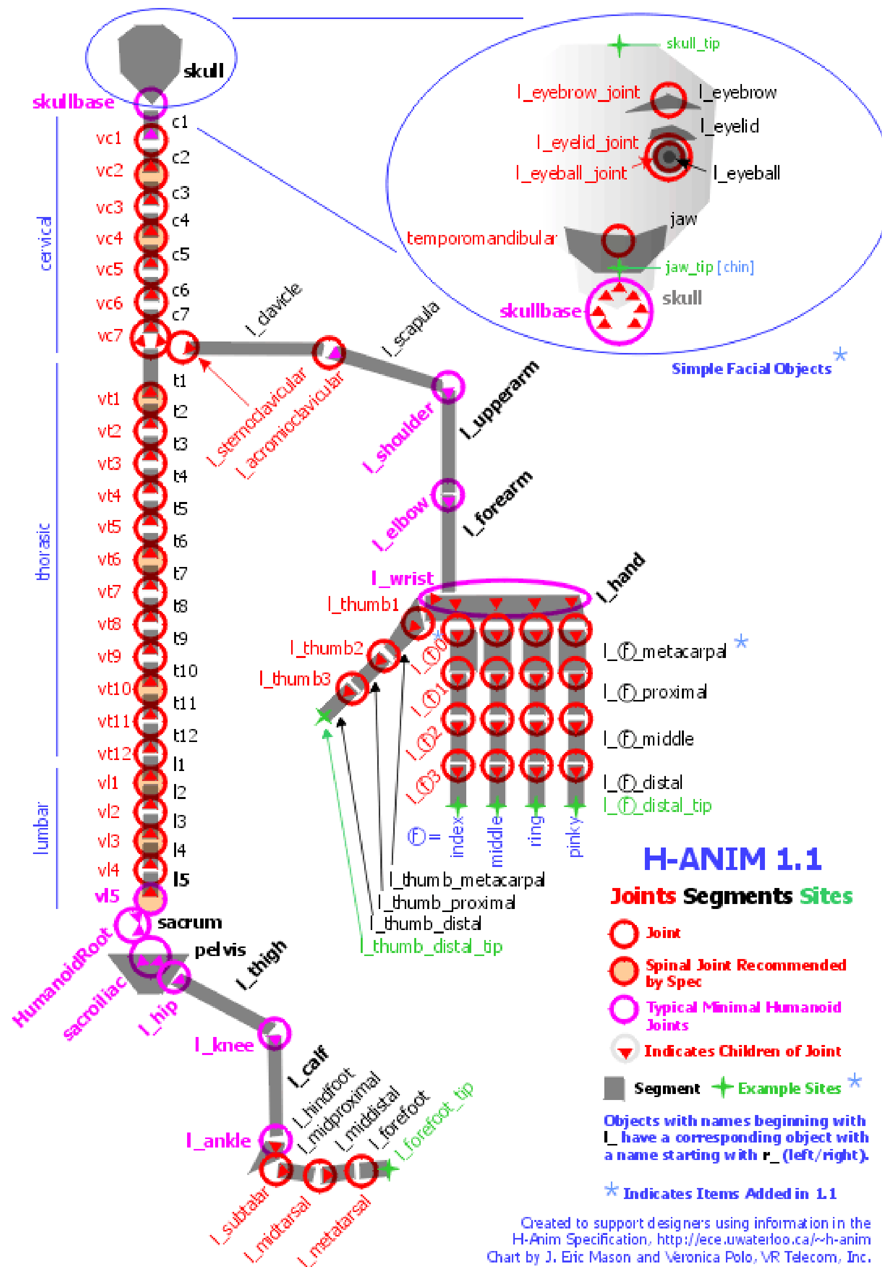



**Figure 2.4.1:** The H-Anim hierarchy of joints. Taken from [13].

# Chapter 3

# Design Issues

```
   Design and programming are human activities;
forget that and all is lost.
```

*– B. Stroustrup, 1991*

In this chapter we investigate the three key features that led to the design of the avatar animator that is the topic of this thesis. We discuss why these features are important and provide possible ways to implement them.

The first feature we investigate is pluggable avatars. By pluggable avatars we mean that the user should be able to provide his own custom avatar and that the system must not be constrained to animate only a set number of proprietary avatars.

The second feature investigated is pluggable input notation. Similar to the first feature, this refers to the ability of the user to describe the gestures to be animated in a notation of his choosing. We want the input notation to be as flexible as possible and users should be able to introduce a new input notation into the system with as little modification to the system as possible.

The final feature we discuss is generic animation. This is the most important feature of the system and the aim is to animate an abstract representation of an

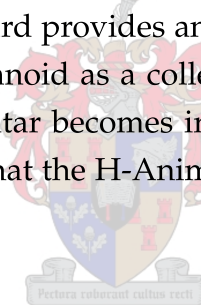
Pectora roborant cultus recti

avatar in such a way that it is completely independent from both the notation used to instruct it, and the avatar that is animated.

At the end of this chapter we also investigate various development environments to find the one most suitable for the implementation of our animation system. Specifically, we discuss the advantages and disadvantages of various programming languages and 3D graphics libraries.

## 3.1 Pluggable Avatars

In chapter 2 we investigated the methods that other animation systems used to implement pluggable avatars. In both the ViSiCAST and the SYNENNOESE systems the H-Anim standard was used as a reasonable constraint on the choice of avatar. The animator could animate any avatar as long as it was compliant with the node hierarchy set by the H-Anim standard. As we explained in section 2.4 on page 20, the H-Anim standard provides an abstract way to describe any humanoid. By describing a humanoid as a collection of segments and joints, the size and appearance of the avatar becomes irrelevant. In the sections that follow we investigate two ways that the H-Anim standard can be used to provide pluggable avatar functionality.




### 3.1.1 The EAI Approach

H-Anim compliant avatars are usually built using VRML [33] (Virtual Reality Modelling Language) or the XML encoding of VRML called X3D [33]. VRML models are typically displayed by embedding them in normal HTML web pages. To correctly view these web pages, they need to be opened using a VRML compliant web browser. VRML compliant web browsers are readily available and most can be downloaded for free[1].

Because the animation capabilities of VRML is too simplistic for the complex

[1]Blaxxun Interactive is one good example and can be downloaded from `http://www.blaxxun.com` .

animations needed in signing avatar systems, developers typically use the External Authoring Interface (EAI) [18] to animate VRML avatars. The EAI enables developers to use VRML script nodes to communicate with the VRML model using other programming languages like Java or C++. This allows the developer to build the avatar animator using a language that is more suited to the task or that is easier to program. The avatar is animated through script nodes that reference the external avatar animator. This process is illustrated in figure 3.1.1.

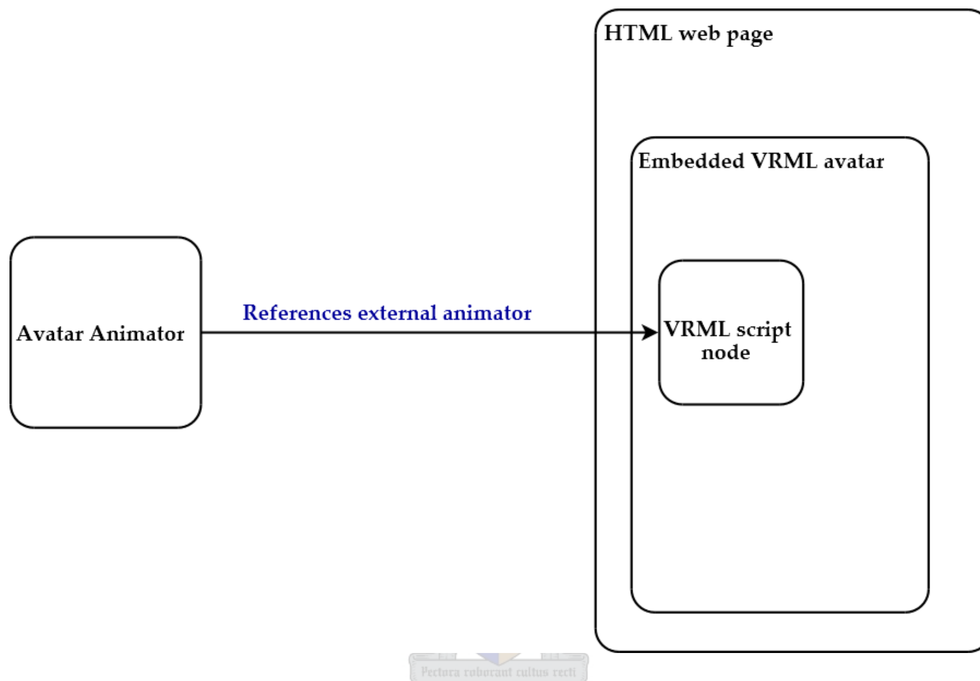



**Figure 3.1.1:** The VRML EAI interface.

One disadvantage of using the EAI is that the entry point into the signing avatar system is now the avatar itself. The user accesses the system by typing the URL of the VRML embedded web page into his VRML compliant browser. This means that all interfacing must be done through the VRML model itself. The controls that allow the user to choose gestures and perhaps avatar appearance will need to be implemented as VRML interactions and embedded into the avatar itself. A workaround for this disadvantage was proposed by the designers of the SYNENNOESE project [3]. Interfacing with the system can be done by designing the system in such a way that it dynamically configures the avatar and its

animations beforehand, based on previous user input. Before the avatar is rendered, the user first chooses which gestures should be signed and also possibly the avatar appearance. After this data is submitted to the animator, the avatar is configured and the user is redirected to the correctly configured avatar.

Another disadvantage of the EAI approach was pointed out by Yang, Petrui and Whalen [38]. They showed that the animation frame rate suffers from the overhead caused by the EAI. They proposed another method that does not use the EAI and showed that a much higher frame rate can be accomplished by not using the EAI. In the section that follows we will investigate this method.

The solution presented in this section does not allow for completely pluggable avatars, since the VRML avatar has to be embedded with the correct script nodes in order to correctly reference the external avatar animator. A user will not be able to use his own custom built avatar before the necessary modifications have been made to it. The only way to make this solution truly pluggable is to add an extra step to the software that would automatically embed the user provided avatar with the necessary nodes beforehand.

### 3.1.2 The VRML File Loader Approach

Another way of displaying VRML models is to convert them to some other suitable format and delegate the rendering of the model to the most convenient rendering mechanism for that format. This conversion is typically done using VRML file loaders. Suitable formats to convert to are formats that share the VRML hierarchical scene graph structure. By converting to such a format the hierarchical structure of the model is kept in the original configuration. This is important when working with H-Anim humanoids since the structure of the H-Anim skeleton as an hierarchy of VRML nodes is critical to the correct animation of the avatar.

One format that is convenient for the reasons given above is the Java3D [28] scene graph format. Java3D is a 3D graphics library for the Java programming language. It is based on an hierarchical scene graph structure that is similar to the scene graph structure used in VRML. When a VRML model is converted into

a Java3D model, the structure of the scene graph is kept intact. The relationship that the VRML nodes have with each other in the VRML scene graph is not changed and the same relationship can be found in the corresponding Java3D scene graph.

The main advantage of converting to a Java3D scene graph is that, once the scene graph is converted, all the advanced functionality of the Java3D library is available to the developer. Also, the rendering of the animated avatar can now be done without the need for a VRML browser and the EAI is not needed. Since the EAI is not needed, the avatar does not need to be embedded with extra script nodes. This means that any avatar can be loaded and animated without the need for any further modifications. Therefore, pluggable avatars are possible without the need for any extra programming.

The disadvantage of using a VRML file loader is the computational overhead of the loading process. The conversion between formats is a computationally expensive operation, since the entire scene graph needs to be traversed and rebuilt in a node-by-node fashion. An H-Anim compliant VRML avatar that is sufficiently articulated for accurate animation of sign language typically consists of hundreds of nodes. The conversion of such an avatar from one format to another can significantly increase the time that it takes the animator to load a new avatar. This also severely increases the memory footprint of the animation system. Notice, however, that the overhead only affects the initial loading time of a new avatar and that the animation frame rate is not affected.

In conclusion, we discussed two possible approaches for using the H-Anim standard to provide pluggable avatar functionality for our signing avatar animator. The VRML file loader is easier to implement and offers a cleaner and more elegant design at the cost of an increased loading time. As we mentioned, the overhead caused by the file loader only affects the initial loading time and the frame rate of the animations is not affected. The EAI solution does not suffer from computationally expensive format conversions, but does suffer from the overhead caused by the EAI. This overhead does not affect loading times significantly but it does slow down the frame rate of the animations. It is for this reason, and also for the sake of a neater design, that we opted to use a VRML file

loader in the implementation of our avatar animator. Details on how this was implemented can be found in chapter 4.

## 3.2 Pluggable Input Notation

Several notations for the representation of sign language have been suggested. We investigated a few of these in section 2.3 on page 11. Sign language linguists are still debating on which of these notations is the best for use as an input notation to a signing avatar animator. For this reason it would be a useful feature in an avatar animation system if the choice of input notation were pluggable.

In this study we propose the following method to implement pluggable input notations. Instead of having the animator directly parsing the input notation, we use an *interface notation*. The interface notation resides between the input notation and the animator. It is designed to be easy to parse by the animator and does not need to be user readable. Our notation is primarily a list of joints and their corresponding movements. It also includes temporal information that consists of rotation speed and start times for the various joint movements. The start times are synchronised by specifying them relative to a global clock.

The input notation is parsed by a separate module. This module generates instructions for the animator in the form of the interface notation. It is the responsibility of the parser module to translate the input notation into the simplified instructions of the interface notation. In this way pluggable input notations are achieved by introducing new input notation parsers into the system. The animator is designed to work using input from the interface notation and can thus animate using instructions from any input notation that can be parsed into the interface notation. The process is illustrated diagrammatically in figure 3.2.1 on page 31.

The interface notation should not be limited by the input notation but should be designed with the animator in mind to ensure that all animations producible by the animator can be represented in the interface notation. The purpose of the interface notation is to separate the animator from the input notations in

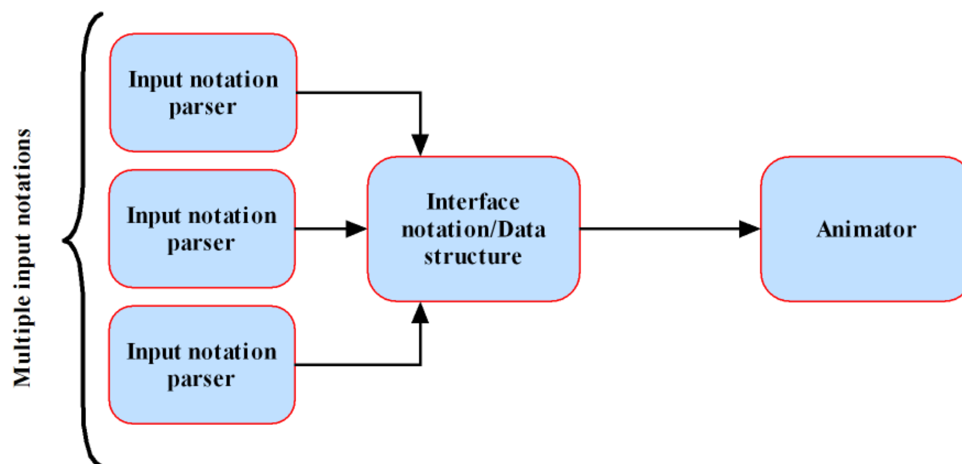



**Figure 3.2.1:** A diagram of the proposed method to implement pluggable notations.

order to remove all coupling between the animation algorithm and any *specific* input notation. Notice that since the interface notation is never authored by a human, it can be implemented as an internal data structure and does not need to be parsed in file form. The parsing process is significantly faster if it does not require any disc access, and is done completely in memory.

The critical component in the implementation of pluggable notations, is the design of the interface notation itself. Whether the notation is implemented as a data structure in memory or as a notation that is written to a file, it must be able to represent any possible animation. The temporal aspects of the animation must also be accurately recorded. All this must be done in as compact a way as possible to minimise the computational overhead. In sections 4.2 and 4.3 we will discuss the implementation that was used in our avatar animator.

## 3.3 Generic Animator

In the previous two sections we showed how to make our animation system more generic by adding pluggable functionality. In this section we investigate the characteristics that the animator needs in order to function in such a pluggable environment.

The animator has two input sources. It receives input from the interface notation that is converted into animations, and it receives input in the form of an avatar model that is to be animated. The primary responsibility of the animator is to build animations on the avatar using the instructions that come from the interface notation.

Even though the avatar model is pluggable, the animator assumes that all avatar models follow a known standard. The standard that we opted to use in our implementation is the H-Anim standard. If the animator can assume that all avatars are H-Anim compliant then it knows that certain joints are always present and can always be referenced using predetermined names that are set by the standard. For example, if the instructions from the interface notation indicates that the left shoulder should be rotated, the animator knows that this can be done by rotating the joint that is referenced by the name "`l_shoulder`" in the avatar [2].

It is important to realise that the H-Anim standard defines multiple levels of articulation and that some joints do not need to be defined if the level of articulation is low. For example, figure 2.4.1 on page 24 defines all the joints in a humanoid that has the highest level of articulation. A humanoid that has a lower level of articulation would normally only have a few of the joints of the spinal column defined, and none of the joints of the fingers. This is the level of articulation that is typically found in 3D Internet chat rooms. However, this level of articulation is too low for all but the most basic sign language gestures. Typically, signing avatars have a high level of articulation close to the maximum that H-Anim provides.

The animator needs to check whether the user-selected avatar is of high enough articulation for the desired animations. If the desired animations require joints that are not defined in the user selected avatar, the user is notified with a warning that indicates the specific joint that needs to be defined. Therefore, the animator can only animate avatars that are sufficiently articulated and are H-Anim compliant.

If we follow the approach recommended by Yang, Petrui and Whalen (see sec-

[2] See figure 2.4.1 on page 24 for a list of H-Anim reference names.

tion 3.1.2) and use VRML file loaders, it is important that the joint references from the original VRML model are kept intact. Not only must the joint hierarchy be kept intact, but the reference names that were set by the H-Anim standard still have to refer to the same joints that were referred to in the original VRML model. Most VRML loaders only focus on the geometry and the relationship between nodes of the model and do not implicitly keep name references intact. Many loaders, for example the Xj3D [34] loader, store the named references separately and do not load them by default. In this case a hash table can be constructed using the information provided by the loader. The hash table maps joint names to the corresponding nodes created by the loader. In this way the standard H-Anim joint names can be used to refer to the applicable nodes created by the loader. The way that VRML loaders are used to create workable models is discussed in more detail in sections 4.2 and 4.3.

Once the model has been constructed, the instructions from the interface notation can be used to generate animations. Each joint in the avatar can be moved at its own speed independent from the speed of other joints. This means that each joint has its own animation clock that determines start and end times for the animations of that joint. A global animation clock is also created to control the global animation speed and also to synchronise the separate animations of each joint. Synchronisation is done to ensure that the start and end times of consecutive animations flow smoothly from one animation to the next. Once all the animations have been synchronised to the global animation clock they can be applied to the avatar and rendered onto the screen.

## 3.4 Development Environment

When designing avatar animation systems, the choice of development environment is driven by the language in which the avatar is modelled. In our case this is VRML. In section 3.1 we mentioned two ways that VRML models can be incorporated into an animation system.

If the EAI approach is used to animate the VRML avatar, 3D animation libraries are not needed since all the animation is done by the VRML engine. The VRML

nodes are controlled directly through external scripts (see figure 3.1.1 on page 27). The only factor remaining in the development environment is the choice of programming language. This depends on the languages that the EAI enabled VRML browser supports as script languages. Most EAI enabled VRML browsers provide Java EAI libraries and only support Java or JavaScript as script language.

The developer has more flexibility when the VRML file loader approach is used. As we mentioned in section 3.1.2, the VRML model gets converted to another 3D modelling format. The choice of this format is the primary factor in the choice of programming language for the animator.

The format in question is usually an entire 3D graphics library. Two of the most popular 3D graphics libraries today are OpenGL [19] and DirectX [31]. Both of these have been tested and proven in industry and are good choices for developing an avatar animator. Both libraries have C++ bindings and C++ is the most popular choice when working with these libraries. OpenGL can also be used in Java by using wrapper classes, but suffers from a slight decrease in performance due to the overhead caused by the wrapper classes. The disadvantage of converting our VRML model to an OpenGl or DirectX model is that neither OpenGl nor DirectX defines models using the scene graph structure seen in VRML. As we mentioned before, it is important for the design of the animator that the scene graph structure is kept intact and that the relationship that joints have with each other in the VRML model is not lost.

In this situation a better choice of 3D graphics library is Java3D. Java3D is a scene graph based 3D graphics library for the Java programming language. Since it is already scene graph based, it is much simpler to transform VRML models into Java3D models than it is to transform VRML models to OpenGl or DirectX models. As we will see in chapter 4, we opted not to use an OpenGl/C++ or DirectX/C++ combination for our development environment but rather chose to use a Java3D/Java environment. The animator was designed in Java and acts on a Java3D model that was converted from an H-Anim VRML avatar using a VRML file loader.

# Chapter 4

# Design and Implementation

> `There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies; the other way is to make it so complicated that there are no obvious deficiencies.`
>
> – *C.A.R. Hoare, 1985*

Pectora roborant cultus recti

In this chapter our implementation of an avatar animator is discussed in detail. The design issues of chapter 3 were weighed against each other and a design was constructed. The aim of this chapter is to provide a detailed design for a generic avatar animation system in such a way that the reader can easily customise the design and implement his own avatar animation system.

In section 4.1 we give an overview of the design of the animation system, and discuss the motivations that led to this design. The design is then divided into three parts that we discuss separately in the three sections that follow. In the last section we investigate the computational bottlenecks that increase the loading times and decrease the animation frame rate. We investigate possible ways in which these bottlenecks can be mitigated or completely bypassed.

## 4.1 The Three-level Design

As discussed in chapter 3, an important part of a generic design is the pluggability of the avatar model and input notation. In our design we opted to use the VRML file loader approach of section 3.1.2 to provide pluggable avatar functionality. We used the interface notation approach of section 3.2 to provide pluggable input notation functionality.

The most logical way to combine these two approaches with a generic animator is in a modular design. We propose a three-part design that consists of a parser, a renderer and an animator. The parser module interprets the input notation and communicates with the animator through an interface notation. The renderer module serves a dual purpose. Its first responsibility is to provide the animator with a model that it can animate. This is accomplished by converting a VRML model into a Java3D model using a VRML file loader. The second responsibility of the renderer is to set up a 3D canvas on which the animated avatar can be rendered. The canvas has to be set up so that the user can rotate and translate the avatar to the most suitable viewpoint for the specific gesture that is signed. Figure 4.1.1 illustrates the modular design and the interfaces that are used for communication between the three modules.

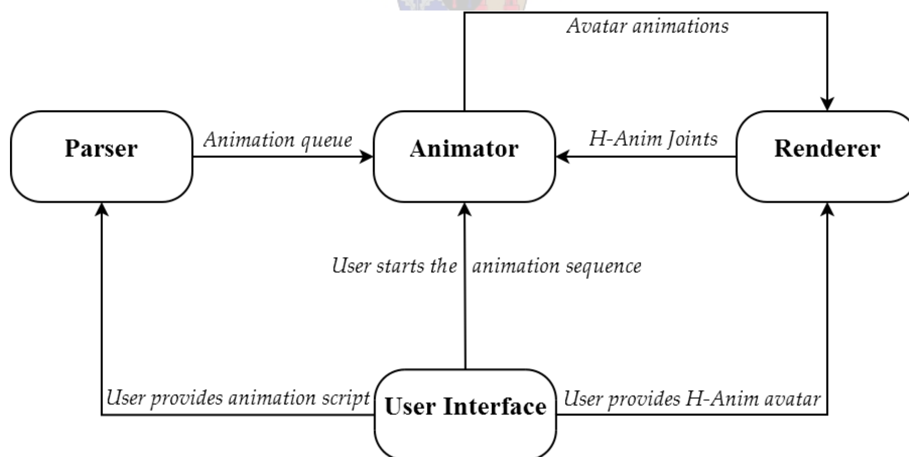



**Figure 4.1.1:** The generic signing avatar animator design.

From figure 4.1.1 we notice that the interface between the parser and the animator is called an *animation queue*. The animation queue is the data structure that serves as the interface notation in our implementation. In the next section we investigate the parser in more detail and we also explain the way in which the animation queue is used as an interface notation.

## 4.2 The Parser

As we have already mentioned, it is the responsibility of the parser to interpret the input notation and generate instructions in the form of the interface notation. Our parser module is primarily made up of four Java classes, namely:

- **NotationParser**: This is a Java abstract class[1] and represents the attributes and actions that are common to all parser implementations. Any specific parser implementation has to extend this class to be useful for the animator.
- **StepParser**: This is an example implementation of the `NotationParser` class. Specifically, this is a parser for the SignSTEP notation that we developed to demonstrate our system.
- **AnimationQueue**: This class represents the data structure that serves as interface notation for the animator.
- **AnimationAction**: This class forms part of the interface notation and represents the smallest possible part of an animation.

We start our discussion with the animation queue. In essence, the animation queue is a first-in-first-out (FIFO) linked list of animation actions represented by the `AnimationAction` class. The animation queue is a temporal queue since actions at the front of the queue happen before actions that are at the back. If there are actions that should execute concurrently, this is accomplished by setting a special flag in the animation action itself.

[1] In Java, an abstract class is a class that cannot be instantiated but is used as a common source of inheritance for any classes that extend it.

The animation action data structure contains information that describes the animation itself. This includes the joint name that the animation should act on, the axis and angle of rotation, the animation speed and various other flags that indicate special actions. One such important flag indicates that the action is concurrent with respect to the animation actions that precede it.

When multiple lists of sequential animation actions must be executed concurrently with respect to each other, nested animation queues are needed. For example, consider the situation where we have two lists of sequential actions. The first list specifies that the left elbow should be rotated followed by a rotation of the left wrist. The second list specifies the exact same movements but for the right elbow and wrist. If we want the left side and the right side of the body to move concurrently, these two lists have to be inserted into the primary animation queue as two separate queues, nested inside the primary animation queue. This nested queue structure requires two separate animation actions. Both actions have the concurrent flag set and both have the complex flag set. The complex flag indicates to the animator that the animation action contains a nested queue structure. We illustrate this process in the diagram of figure 4.2.1 on page 39. In our example the nested queues from the two animation actions would contain the elbow and wrist rotations for the left and right side of the body respectively.

We developed the SignSTEP notation to demonstrate how an animation queue can be built from an input script. The SignSTEP notation is based on the STEP and XSTEP notation for humanoid animation [11]. We customised the XSTEP notation to meet our needs and called the result SignSTEP.

The XSTEP notation was designed to be a general notation for the animation of humanoids. It supports high level dynamic logic for the control of humanoids. For example, XSTEP supports the `if-then-else` and `do-repeat` constructs that are commonly found in traditional programming languages. It is also possible to embed meta-language statements for high-level interactions into the script[2]. For our purposes we did not need this added functionality and wanted to keep the script as simple as possible. We adapted XSTEP in order to make it easier to

[2]We refer to the file in which the gestures are specified, as the input script. This is the input to our system and is transcribed using the input notation.

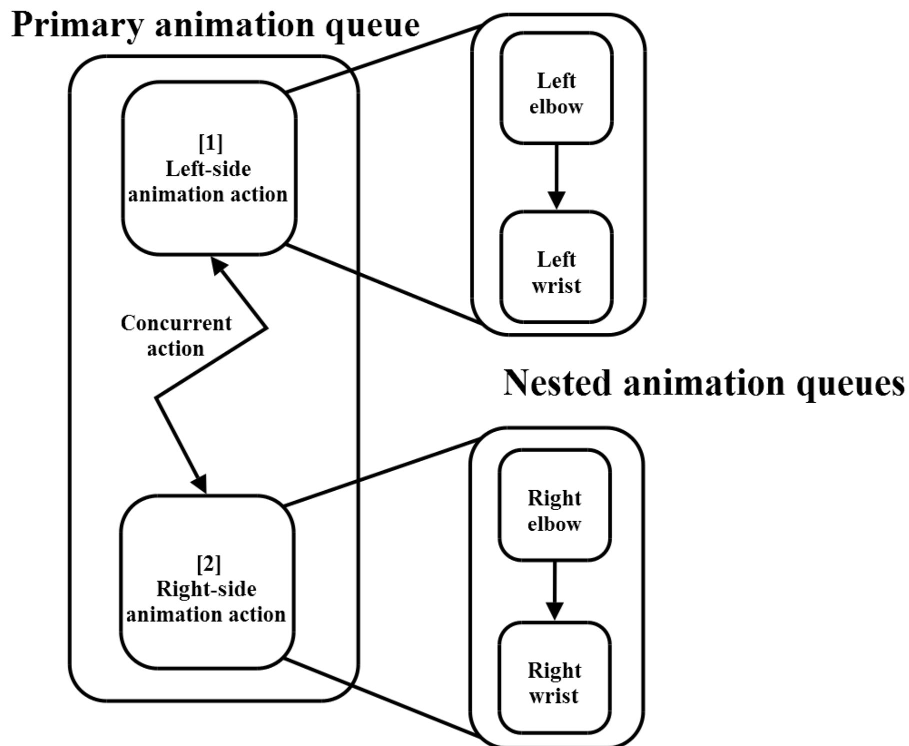



**Figure 4.2.1:** An example of nested queues.

parse by removing some of the high-level functionality. However, we also had to add some extra joint definitions in order to support the fine motor movement that our system requires. After the adaptations have been made, the result is a streamlined version of XSTEP that supports fine motor movements such as individual finger rotations.

The four most important elements of the SignSTEP notation are the `seq`, `par`, `turn` and `trans` elements. The `seq` and `par` elements are grouping elements that specify whether the action elements that are nested inside them, should be executed concurrently or sequentially. Elements nested inside a `seq` element are executed sequentially and elements that are nested inside a `par` element are executed concurrently. The two action elements are the `turn` and `trans` elements. The `turn` element represents a joint rotation and must contain `speed` and `rotation` elements that specify the speed, axis of rotation and angle for the rotation action. The `trans` element is similar and specifies a joint translation. It

must contain speed and dir elements that specify the speed, direction and distance of translation. The interested reader can refer to Appendix A, that contains the full SignSTEP DTD (Document Type Definition), for more detail.

SignSTEP is an XML notation and is therefore easily parsed using readily available XML libraries [32]. The Java Standard Developers Kit[3] (Java SDK) contains built in libraries for the creation and parsing of XML documents.

The parsing process is made up of two steps. First, the data from the SignSTEP input script is converted into a tree structure using standard Java XML parsers. This tree structure is then recursively traversed and converted into an animation queue with the corresponding animation actions correctly inserted. A diagram depicting this process can be seen in figure 4.2.2.

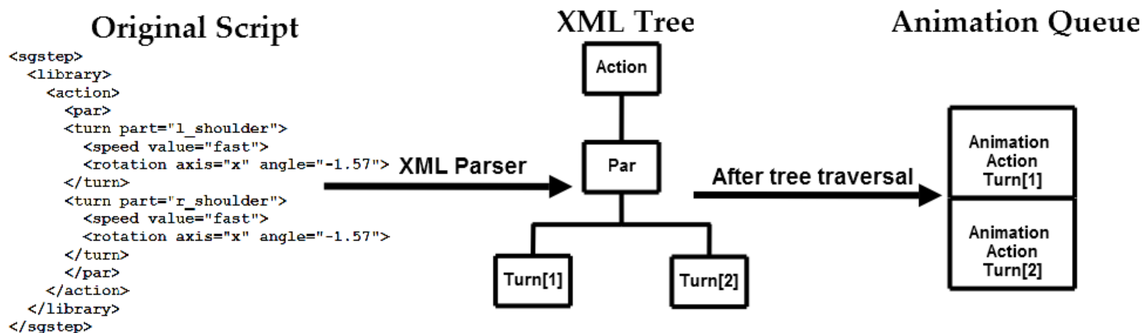



**Figure 4.2.2:** The parsing process.

In the SignSTEP example script of figure 4.2.2, only two rotations are specified. They are represented as *turn* elements in the script and appear as *turn* nodes in the XML tree. Notice that both *turn* elements are nested inside a *par* element. The *par* element indicates that the actions nested inside it must execute concurrently. This is also indicated in the XML tree as a *par* node that acts as the parent node of the turn nodes. If the actions were sequential instead of concurrent, the *par* node and *par* element would be replaced by a *seq* node and a *seq* element. The parser generates an animation action for each of the turn elements and inserts them into an animation queue to be sent to the animator. Notice that the parser

[3]The Java SDK can be downloaded from http://www.java.sun.com .

has to query the parent node of the `turn` element when generating animation actions. The parent node indicates whether the animation action is concurrent or sequential and hence also indicates whether nested queues are needed.

## 4.3 The Animator

As we have seen in figure 4.1.1 on page 36, the animator module receives two input streams. It receives input from the parser in the form of an animation queue and it receives input from the renderer in the form of a Java3D model and a hash table of joint references. In turn, the animator has to use this input and generate an animated avatar model that is sent to the renderer for rendering to the screen.

The animator module consists of two Java classes:

- **AnimationBuilder**: This is the central part of the animator module and contains the main algorithm that loops through the animation queue and applies the animation actions to the avatar model.
- **AnimationTracker**: This class is responsible for synchronisation. It tracks and records joint movements throughout the animation sequence. It maintains the position of all joints at all times and is also responsible for ensuring that no joint is ever rotated or translated past its minimum or maximum range.

The animation process starts in the `AnimationBuilder` class. The animator is initialised by setting the global animation clock to zero and creating a new animation tracker. The animation tracker is initialised by setting all joints to their default positions and by initialising the lookup table that stores the minimum and maximum rotations for all joints.

The primary animation loop starts by iterating over all the animation actions in the animation queue that originated from the parser. Each action is first checked to establish whether it is concurrent or sequential and whether it contains nested queues. If the action contains a nested queue, the global animation

clock is frozen and a temporary local clock is created for the nested queue. Once the actions of the nested queue finish, the global clock is resumed and iteration through the primary animation queue can continue. Based on the current value of the global clock and depending on whether the animation action is concurrent or sequential, the start time for the animation action is calculated. The end time is then calculated based on the start time and the animation speed that was specified in the current animation action.

Once the start and end times have been set, the motion interpolators can be constructed. We used the `PositionInterpolator`, `RotationInterpolator` and `Alpha` Java3D classes to construct interpolators (see [27] for details). The interpolators are constructed in such a way that the animations are as smooth as possible and that the transition from one animation to the next do not appear unsteady.

The next step is to update the joint position in the animation tracker and to check whether the new position is within the tolerated limits. This is done by first calculating the expected position of the joint after the projected animation, based on the current position stored in the animation tracker. This value is then compared to the global minima and maxima for that joint. If the projected value falls outside the tolerated limits, the user is informed and the minimum or maximum value for that joint is used instead.

Before we can explain how the animation is applied to the avatar model, we must first discuss scene graphs. Previously, we mentioned that Java3D uses a scene graph to represent 3D models. Animation of a 3D model is accomplished by adding certain nodes to the scene graph of the model. A scene graph is a directed acyclic graph (DAG) that represents a 3D scene [27].

For the purposes of our explanation, a scene graph can contain any one of three different kinds of nodes, namely, group nodes, geometry nodes or behavioural nodes. Geometry nodes represent the geometry of the model, or more frequently parts of the model, while group nodes assemble geometry nodes, that logically fit together, into groups. For example, all the geometry from the fingers of the left hand can be assembled together in a group node called the left hand. Figure 4.3.1 on page 43 shows an example of a scene graph in Java3D. The H-Anim nodes are

examples of geometry nodes while the transform nodes are group nodes. The animation nodes are examples of behavioural nodes.

**Figure 4.3.1:** An example of a partial Java3D scene graph.

Behavioural nodes are needed when the model is animated. Since a scene graph is a directed graph, changing a node affects all its children nodes but does not affect the rest of the graph. If we refer back to our left hand example, adding an animation that moves the left hand group would also move the fingers of the left hand. This seems logical, but the automatic animation of the fingers would fail were it not for the relationship that the geometry nodes have with each other in the scene graph. If the fingers of the left hand were not children nodes of the left hand group, animating the left hand would leave the fingers stationary.

Depending on the VRML file loader used in the renderer module, the animator adds either two or four nodes to the scene graph for every animation action in the animation queue. In VRML each group node has a centre of rotation that specifies around which point in 3D space a joint should rotate. Some VRML file loaders create Java3D scene graphs that are optimised to keep the memory footprint as small as possible. One of the side effects of this optimisation is that the original centre of rotation from the VRML model is lost. This implies that the animator first has to translate the joint to its centre of rotation before the

animation can be correctly applied. After the animation is applied, the animator then has to transform the joint back to its original position relative to the rest of the model. We show an example of this process in figure 4.3.2.

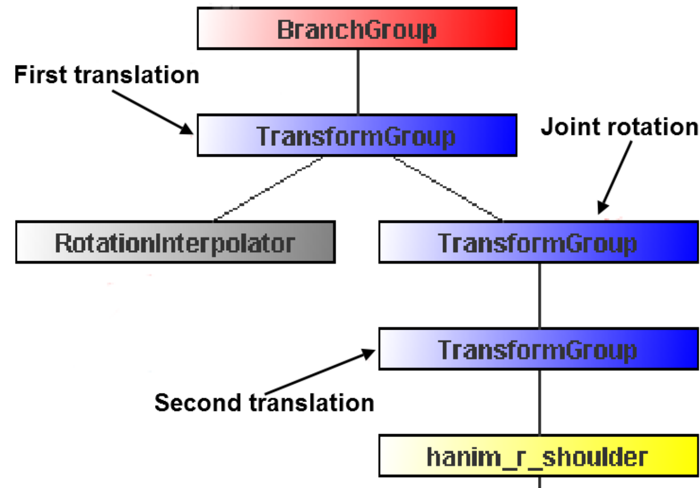



**Figure 4.3.2:** The animation action is added to the scene graph.

In the example of figure 4.3.2 we add a rotation to the right shoulder joint using a VRML loader that does not keep the original VRML centre of rotations intact. We see that three transform groups are added as parents of the joint we are animating. The first and second transform groups translate the centre of rotation as explained in the previous paragraph, while the group referred to as the joint rotation is the actual animation as specified in the animation action. The fourth node that is added is the rotation interpolator that acts on the joint rotation group to create a moving animation. This node can be added either as a parent or a sibling of the joint rotation group on which it acts. If the VRML file loader indicated to the animator that it does in fact keep the original centre of rotation intact, only the rotation interpolator and the joint rotation nodes are needed.

At this point the main animation loop has finished one iteration. The primary animation clock is then updated and the animator moves on to the next animation action in the animation queue. For each animation action the animator goes through the three stages explained in the previous paragraphs, namely:

1. The start time and end time are calculated and an interpolator is constructed.
2. The validity of the rotation or translation is checked using the animation tracker.
3. The scene graph is edited by adding animation nodes as parents of the joint to be animated.

Once all the animation actions in the animation queue have been processed, the animated avatar model is sent to the renderer module for rendering to the screen.

## 4.4 The Renderer

The renderer module has a two-fold responsibility. Firstly, it must provide the animator module with a Java3D model of the avatar. This is done by converting a VRML model using a VRML file loader. Secondly, it must create a platform independent 3D canvas for the rendering of the animated avatar.

The renderer module consists of two Java classes, namely:

- **AvatarLoader**: This is a Java abstract class and represents the functionality that all renderer implementations should provide to the animator module. All renderer implementations must implement this class in order to integrate with the animator module.
- **VRMLLoader**: This is an implementation of the `AvatarLoader` class that loads H-Anim compliant VRML models.

Pectora roborant cultus recti

Our implementation is divided into two steps. The first step is to create a 3D canvas and initialise it with an empty scene. Since this is a simple scene, we can use the Java3D helper class `SimpleUniverse` [27] to create a scene. In order to give the user the ability to rotate and move the avatar in 3D space, we add some navigational behaviours to the scene.

Three navigational behaviours were added to allow the user to move the avatar using the mouse. Java3D provides three classes that add basic mouse behaviour to a scene, namely: `MouseRotate`, `MouseTranslate` and `MouseZoom` [27]. The combination of rotation, translation and zoom functionality allows the user to view the avatar from every angle and at any distance.

Now that we have created a 3D scene, we have to load an avatar model. We do this using the Xj3D VRML loader package [34]. Xj3D is a library that provides loaders for both VRML and the newer XML version of VRML called X3D. The Xj3D project is, however, still under development and the version that we used (version M10) still has some outstanding issues and lacks some important functionality. For example, the loader only supports raw files and cannot load compressed models. VRML models can be rather large and it is common practice to distribute them in a compressed form. The reader may refer to chapter 5.2 for more detail on Xj3D (version M10) and the recently released Xj3D (version 1.0).

Another important feature that the current version lacks, is the ability to customise the construction of the Java3D model. With the default configuration, the Java3D model is constructed in such a way that the H-Anim joint name references are lost. Also, the model is automatically optimised in such a way that the joint centre of rotations are lost.

In order to compensate for the loss of H-Anim references, we construct a hash map of all the nodes in the model and use the H-Anim joint names as keys to the applicable nodes in the map. The `getNamedObjects` method from the Java3D `Scene` class is used to create this map. The hash map is then sent with the model to the animator module to be used as reference. We use a hash map in order to minimise the lookup time of the nodes during the animation phase. Increasing the lookup time during the animation phase would decrease our frame rate during animation and would cause a loss in performance.

Compensating for the loss of the joint centre of rotations is a more difficult problem. Once again, we construct a hash map that is sent with our avatar model from the renderer to the animator. Similar to our previous hash map, we also use the joint names as keys, but now they refer to 3D coordinates instead of

nodes in our model. We fill this hash map by iterating over all the nodes in the original model and extracting the `center` field from each of them. The `center` field contains the centre of rotation from the original VRML model. A vector of coordinates is then constructed from this information and is inserted into the hash map.

Before the model is ready for animation, we have to address one last issue. When the Xj3D loader constructs the Java3D model, it optimises the scene graph by using `Link` nodes. `Link` nodes minimise the amount of memory used by the scene graph by re-using parts of the scene graph that are identical [27]. `Link` nodes are always leaf nodes in a scene graph but refer to a special kind of group node called a `SharedGroup` node. The `SharedGroup` node has no parent node but can have child nodes. In other words, a `SharedGroup` node can never be added as a child node anywhere in the scene graph but can act as a parent node for other nodes. The only way a `SharedGroup` node can affect the scene graph is through the `Link` node that references it. By using multiple `Link` nodes to refer to the same `SharedGroup`, the section of the scene graph that has the `SharedGroup` as parent, is re-used instead of duplicated. However, since the `SharedGroup` node has no parent, it cannot be referenced from the root of the scene graph using traditional tree traversal algorithms that are based on the child-parent relationship that all nodes in the scene graph have with each other. Hence, the use of `Link` nodes in our scene graph makes traditional tree traversal algorithms impossible. Therefore, we must traverse the model to remove all `Link` nodes, before the model can be animated.

Pectora roborant cultus recti

The removal of `Link` nodes from the scene graph is done as a post-processing step after the scene graph has been created. `Link` nodes are removed by traversing the entire scene graph and replacing each `Link` node with a `BranchGroup` node. A `BranchGroup` node is a type of group node (see section 4.3) and is used to assemble nodes that logically fit together into one group. For every `Link` node that is encountered in the scene graph, we construct a `BranchGroup` node that contains the same children nodes as the `SharedGroup` node that is referred to by the `Link` node. The `Link` node, and the `SharedGroup` node that it refers to, is then deleted from the scene graph and replaced with the `BranchGroup` node that was created using the children nodes of the `SharedGroup` node. This is a computa-

tionally expensive process and is only performed when memory optimising file loaders such as Xj3D are used.

Once the canvas has been initialised and the model has been made ready for animation, the model and the two hash maps are made available to the animator module. When the animator has finished adding the necessary nodes for animation, the model becomes ready and the renderer can render the model to the 3D canvas that was configured for this purpose. The user can then move the avatar around inside the canvas and display the animations when ready.

## 4.5 Optimisations

The 20-80 rule of software engineering states that 80% of the runtime will take place in 20% of the code. This system is no exception to that maxim. We used the Netbeans profiler[4] from Sun Microsystems to analyse our program in order to find the routines that would benefit most from optimisation. The profiler showed that the majority of time is spent in the routine that uses the Xj3D file loader to load in the VRML model. Figure 4.5.1 shows a screen shot of the CPU telemetry window of the profiler. Here we see that 71.1% of the total CPU cycles are spent in the `loadfile` method of the renderer module. By total CPU cycles we refer to the total time that the CPU spends on processing one cycle of our program, that is the time it takes to load an avatar, load an animation and render the animated model to the screen.

We also used the profiler to find the data structures that are most responsible for the large memory footprint that our system uses. The profiler showed that the `XObject[]` object, that forms part of the XPath package, is mostly to blame. The XPath package is used to query the XML tree that is built from our input script in the parser module. Figure 4.5.2 on page 49 shows a graph of the heap memory used by our program as a function of time. We see that at a specific time the amount of heap memory used suddenly jumps from about 15Mb to about 35Mb. This point in time corresponds to the point where the parser module starts the

[4] http://www.netbeans.org

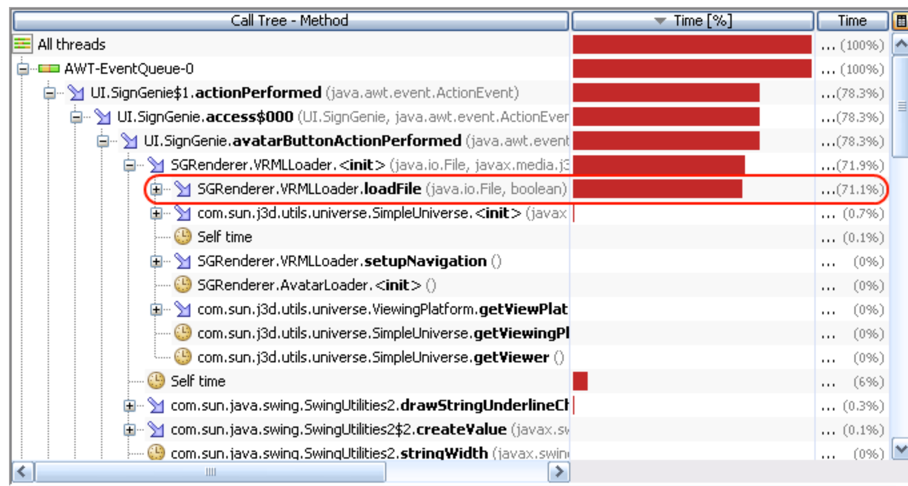



**Figure 4.5.1:** Telemetry provided by the Netbeans profiler.

XML parser to build the XML tree.

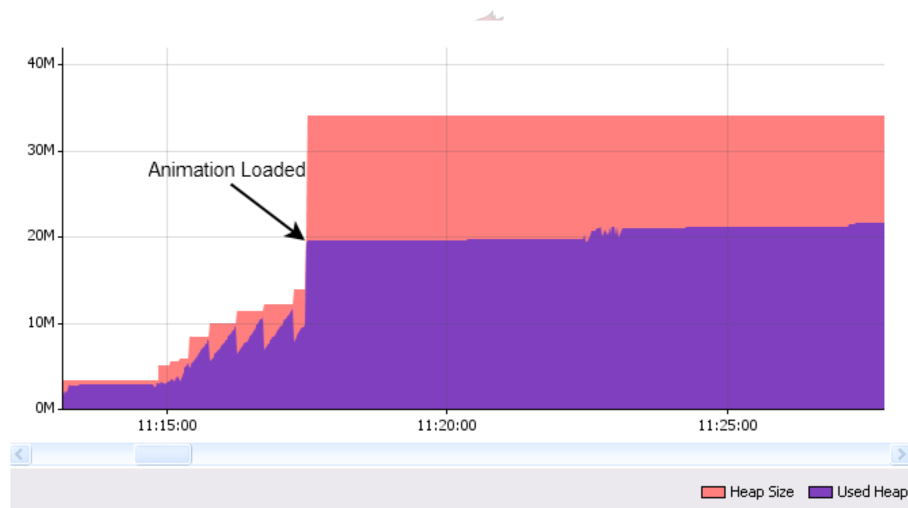



**Figure 4.5.2:** The amount of heap memory used as a function of time.

We conclude that our animation system can benefit from two optimisations. Firstly, the `loadfile` method of the renderer module must be optimised for speed. The two primary factors that make this method computationally expensive are the access to secondary memory that has to be made, and the conversion between formats. Disk access is inherently slow, and therefore we need to rather

concentrate on optimising the conversion process between the VRML model and the Java3D model. This conversion is done using the Xj3D loader followed by some post-processing that includes removal of the `Link` nodes (see section 4.4). At the time when the renderer was implemented, the Xj3D loader was still under development and the newest release at that time was version M10. Since that time a new version of Xj3D was released (version 1.0). It is not known at this time whether version 1.0 performs better than version M10 or whether it offers the possibility of creating a Java3D model without using `Link` nodes. If version 1.0 does perform better, modifying our renderer to use version 1.0 rather than version M10 would significantly improve our overall performance. We address this issue further in our section on future work in the next chapter (see section 5.2).

The second optimisation that will be of benefit is the optimisation of the memory usage of the XML parser that we use in the parser module. If we can decrease the memory footprint of our program, less memory will need to be cached to the hard drive by the operating system. The less memory that needs to be cached by the operating system, the better the performance of our program will be. One way to decrease the amount of memory our parser module uses, would be to switch to a different kind of XML parser. XML parsers come in two varieties, namely, SAX (Simple API for XML) and DOM (Document Object Model) parsers [10]. Currently, we use a DOM parser, since the tree structure that a DOM parser builds is a highly intuitive base from which we can construct our animation queue. However, it is this tree structure that is responsible for our large memory footprint. A SAX parser is event driven and does not build a tree structure like a DOM parser does. If we adapt our parser module to use a SAX parser instead of a DOM parser, we will gain performance due to the decrease in memory footprint. However, we must not forget that using a SAX parser would complicate our parser module, since we no longer have a XML tree from which we can easily build our animation queue. This adaptation would only be sensible if the performance gain due to the use of the SAX parser outweighs the decrease in performance due to the extra calculations that need to be done.

In summery, we proposed two optimisations that could benefit our signing avatar animation system. Neither optimisation is guaranteed to improve performance,

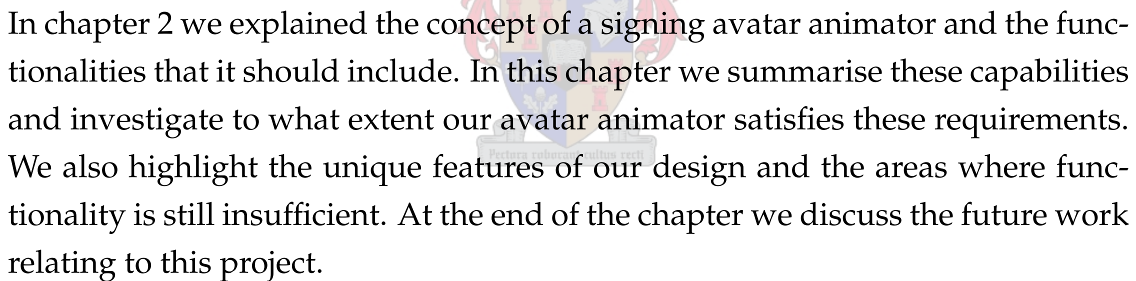
but will need to be implemented and compared with the unoptimised version to determine whether the performance was improved. The implementation of these optimisations is included in the future work of this project.

# Chapter 5

# Results

```
   One does not discover new lands without
consenting to lose sight of the shore for a very
long time.
```

– *André Gide*



In chapter 2 we explained the concept of a signing avatar animator and the functionalities that it should include. In this chapter we summarise these capabilities and investigate to what extent our avatar animator satisfies these requirements. We also highlight the unique features of our design and the areas where functionality is still insufficient. At the end of the chapter we discuss the future work relating to this project.

## 5.1 Issues Relating to Functionality

In this section we discuss the most important issues that relate to the functionality of our animation system. We discuss these issues from both the perspective of the end user of our system and from the perspective of the designer that aims to adapt our system to function in other sign language projects. More specifically, we concentrate on the following issues:

- possible discrepancies between real and generated gestures;
- generic animation in situations where multiple avatars with varying relative dimensions are used;
- the choice of input notation;
- restrictions on various joint rotations; and
- multiple rotations on a single joint that are not commutative.

### 5.1.1 Discrepancies between Real and Generated Gestures

The primary objective of a signing avatar animator is to generate recognisable animated sign language gestures given an appropriately articulated avatar model and a sufficiently descriptive gesture input script. For a gesture to be recognisable, the discrepancies between the generated gesture and the gesture as signed by a real person, have to be kept to a minimum. Discrepancies can be classified into two categories. In the first category we have discrepancies that might make the gesture unrecognisable but will not change the meaning of the gesture. These discrepancies are also found between different signers and only make the gesture unrecognisable in severe cases. In the second category we find the discrepancies that can change the meaning of the gesture. These discrepancies are the most important to avoid. The most common situation where this kind of discrepancy occurs, is in gestures where the non-manual components of the sign are important. It is often the case that the only difference between two signs is the non-manual component. Non-manual gestures are linguistically meaningful and are often used to indicate that a phrase is a question or to stress an idea [4]. For example, the only difference between the SASL sign for *weight* and *maybe*, is in the non-manual component of the sign (see figure 5.1.1 on page 54). The position and motion of the hands and arms are exactly the same for both *weight* and *maybe*. However, in the sign for *maybe*, the head is tilted to the side, the facial expression changes and the shoulders are raised upward. This example demonstrates the importance of non-manual gestures and indicates that a system that

supports only limited non-manual gestures would not be useful in situations such as these.

**Figure 5.1.1:** The difference between *weight* and *maybe*.

In figure 5.1.2 on page 55 we give an example of the ASL gesture “Thank you”. The top section of the figure shows the gesture, as signed by a person fluent in ASL, captured at two separate time steps. The bottom section of the figure shows the same gesture, as generated by our animator, captured at approximately the same time steps. We use this example to point out common discrepancies that is found between our generated gestures and the actual gesture signed by a real person.

Pectora roborant cultus recti

As expected, we notice several discrepancies of the first category. At certain time steps, the exact position of the hand, as well as the direction that the fingers are pointing in, differ to some extent. This is mostly due to insufficient accuracy in the input script, rather than a lack of accuracy in the animation process. Discrepancies of this nature can easily be solved by fine-tuning the input script in the areas where there are differences.

A more serious discrepancy is found in the facial expressions that accompany the manual gesture. When signed by a real person, the signer adds subtle yet important facial expressions to the manual gesture – for example, the smile at the end of the gesture. The generated gesture does not include any facial expression. As we mentioned, facial expression is an important linguistically meaningful com-

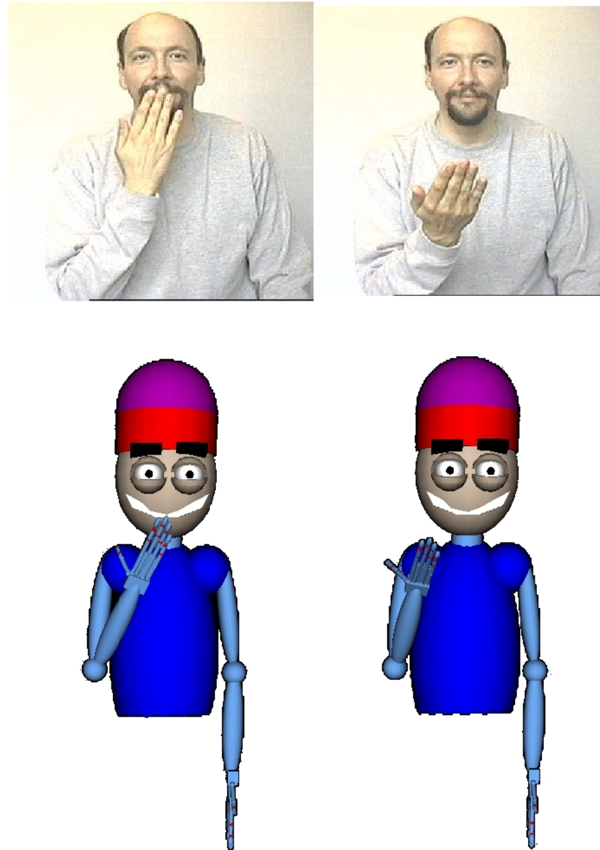


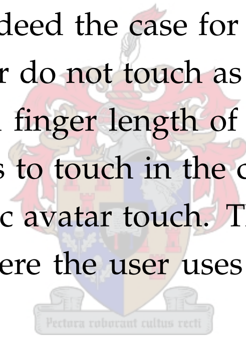
**Figure 5.1.2:** A comparison of the "Thank you" gesture. Taken from [37].

ponent of sign language, but in our example the lack of facial expression does not change the meaning the sign. However, it might make the sign unrecognisable especially when used as part of a larger phrase. Sign language without facial expression is at best equivalent to a speaker speaking in a monotone unchanging voice and at worst ambiguous and unrecognisable [4].

Currently, our animator only supports simple non-manual gestures such a head tilts and shoulder movements. The lack of support for facial expression limits our animator to only a few gestures in which the facial expression plays no role in the meaning of the gesture. We discuss support for facial expression further in section 5.2.

### 5.1.2 Generic Animation in Varying Situations

In section 3.1 on page 26 we discussed our aim to add pluggable avatar functionality to our design. Part of our goal was that the system should generate the same animations on any compliant avatar given the same input script. However, we only succeeded in this goal when the avatar that was used, followed the normal humanoid dimensions. This problem becomes especially apparent when cartoon and realistic avatars are used interchangeably. We give an example of such a situation in figure 5.1.3. The avatar on the left [7] is modelled following normal humanoid dimensions and is considered realistic in that sense, while the avatar on the right is modelled using cartoon dimensions and has exaggerated hand and head size. We transcribed a SignSTEP input script for the SASL sign "home" and applied the same script to both the cartoon avatar and the realistic one. The script was designed with the cartoon avatar in mind and should show straight hands with the fingers just touching at the centre of the body. From the figure we notice that this is indeed the case for the cartoon avatar but that the fingers of the humanoid avatar do not touch as they do for the cartoon avatar. This is due to the exaggerated finger length of the cartoon avatar. The elbow rotation that causes the fingers to touch in the cartoon avatar is not enough to make the fingers of the realistic avatar touch. The input script therefore has to be fine tuned in the event where the user uses multiple avatars with varying relative dimensions.



Notice that this problem can always be resolved by fine tuning the input script when non-standard dimensions are used. The problem is not due to lack of ability to animate generically, but is rather a lack of ability to dynamically fine-tune the input script, depending on the avatar used. If the parser had prior knowledge of the avatar model, this type of dynamic altering of the input script would be possible. However, this would make the parser dependent on the renderer and compromise the pluggability of the parser module.

**Figure 5.1.3:** The SASL sign for "home" using two different avatars.

### 5.1.3 The Input Notation Design

The input notation that we developed to demonstrate our animation system (SignSTEP) forces the author of the script to have knowledge of the coordinate system that our animator uses. The author has to specify each joint rotation in the gesture as a combination of rotations about the $x$, $y$ and $z$ axes. In most cases this is a non-trivial operation and usually requires that the author goes through several test-and-fine-tune iterations. Ideally we would like to have a notation in which the user can specify basic actions that make up gestures instead of specifying joint rotations. This would allow the author to transcribe gestures efficiently without prior knowledge of the coordinate system. In section 5.2 we discuss a sign editor tool that is currently being developed in the SASL-MT project, that will allow the author to specify joint rotations in a more user friendly manner.

### 5.1.4 Disadvantages of Euler Angles

Regardless of the input notation used, the animator is based on the use of Euler angles [8] for the specification of joint rotations. Euler angles are the simplest way to specify a rotation and only requires an axis ($x$, $y$ or $z$) and an angle to be specified. It is also the easiest notation to visualise conceptually and would therefore be a sensible choice for the notation used in the interface.

However, there are disadvantages to specifying rotations using Euler angles. There are two main issues that one should be aware of when specifying angles using Euler notation. Firstly, notice that all rotations are specified using only three parameters that represent three actions performed sequentially:

(1). a rotation about the $x$ axis,

(2). a rotation about the $y$ axis, and

(3). a rotation about the $z$ axis.

It is important to notice that the order in which these operations are performed is not commutative. The animator was designed to build up any rotation about an arbitrary axis by dividing the rotation into three separate rotations about the $x$, $y$ and $z$ axes. The author of an input script has to be aware of this and has to carefully specify rotations in the correct order to avoid unexpected results from the animator. Refer to the example in figure 5.1.4 on page 59. In the first situation a vector, indicated by a red bar on the $x$-axis, is first rotated about the $z$-axis, and then about the $x$-axis. It ends in a position indicated by the green bar on the $z$-axis. We show that this sequence of rotations is not commutative by switching around the order in the second situation. This time the vector is first rotated about the $x$-axis but since the vector is positioned on the $x$-axis this has no effect. The second rotation about the $z$-axis now causes the vector to end on the $y$-axis instead of the $z$-axis as before. The situation that we encountered the second time, when a vector lying directly on the $x$-axis was rotated about the $x$-axis, illustrates our next issue, called gimbal lock.

Gimbal lock occurs when a vector is rotated such that it lies directly on either the $x$, $y$ or $z$ axes [24]. Any further rotations about this axis will then have no effect

and one degree of freedom is lost from the rotation transformation. This is what happened in our example of figure 5.1.4. The rotation about the $x$-axis had no effect since the vector was lying directly on the $x$-axis.

We can solve the problem of gimbal lock by representing rotations using quaternions instead of Euler angles. A quaternion is a 4-dimensional complex number that specifies rotation using three degrees of freedom [5, 16]. Quaternions are not as intuitive to visualise as Euler angles and is in that sense more difficult to work with, but they do not suffer from gimbal lock. Also, since a rotation is not divided into an $x$, $y$ and $z$ component, the non-commutativity of rotation is not a problem. The Java3D library supports rotations specified as quaternions and modification of the animator module to use quaternions instead of Euler angles would be possible. We initially designed the animator to use the more intuitive Euler angles for their intuitive simplicity, but a sensible adaptation would be to modify the animator to optionally use quaternions instead of Euler angles.

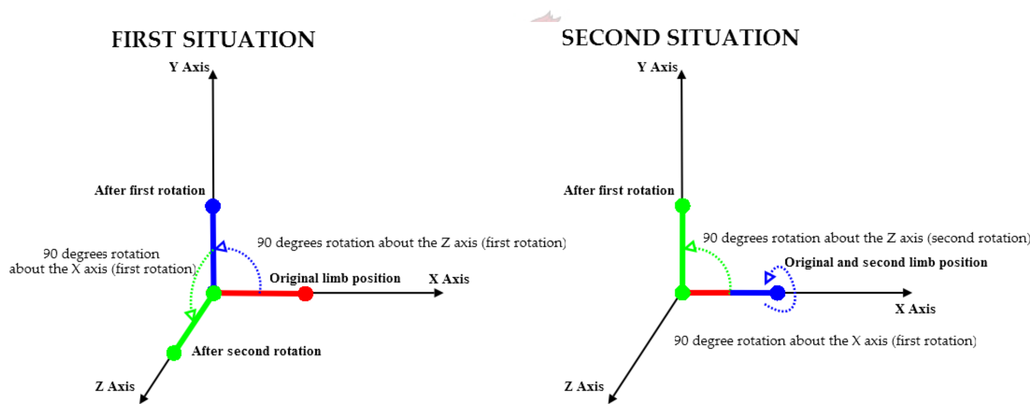



**Figure 5.1.4:** An example illustrating that rotation is not commutative.

### 5.1.5 Restrictions on Joint Rotations

The last issue that we discuss concerns the restrictions that the animator places on the rotation of joints. By restricting joint rotations to certain minimum and maximum angles, the animator prevents the generation of joint rotations that would be impossible for a real human signer. When the animator encounters

a rotation that it deems impossible, it issues a warning and uses the maximum allowable angle instead.

An example of an impossible rotation would be the head rotating $360^\circ$about an axis that is perpendicular to the floor. In other words, the avatar would be turning its head around all the way till it faces forward again. Clearly, the head should not be able to rotate by more than $90^\circ$in each direction.

However, when some joints are rotated to a certain degree, it becomes easier to describe gestures when the restrictions on that joint are relaxed. In the following example we show that the elbow joint is an example of such a joint. Suppose that the elbow starts in a position where it is not rotated at all (see figure 5.1.5). In this position the arm is straight with the upper and lower arm in line. In this position the elbow cannot rotate about the axis that is perpendicular to the chest (see figure 5.1.8 on page 62), but can rotate in both the other axes (see figures 5.1.6 and 5.1.7 on pages 61 and 62).

Now suppose that we rotate the elbow by $90^\circ$ so that the lower arm is perpendicular to the upper arm. From this position the elbow joint rotates about all three axes and the restriction of the rotation about the axis perpendicular to the chest should be relaxed. We illustrate this situation in figure 5.1.9 on page 63. Where it previously was not possible to rotate about the $z$-axis, it now becomes possible. Notice that the $z$-axis appears to be the same axis as the $y$-axis in figure 5.1.7. These axes are not the same but appear this way because the whole local coordinate system of the joint rotates by $90^\circ$ when the elbow rotates.

In the current implementation we relax the restrictions as much as possible to simplify the transcription of gestures. However, this means that seemingly impossible rotations can sometimes be generated. The system lacks a way to dynamically alter the restrictions on rotations, depending on the current position of a joint.

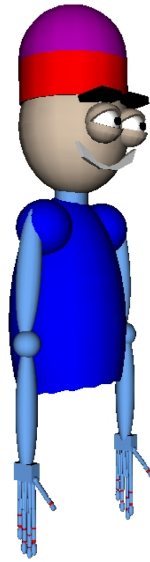


**Figure 5.1.5:** The neutral position.

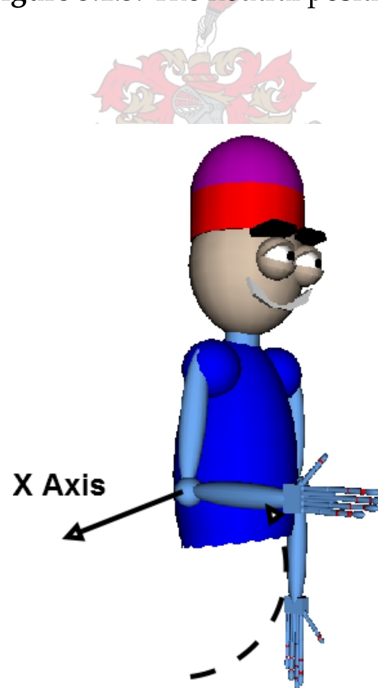



**Figure 5.1.6:** Rotation about the $x$-axis is possible.

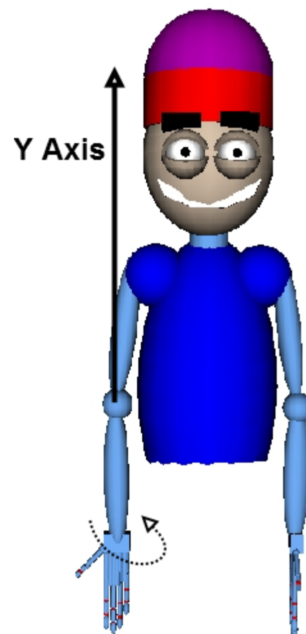



**Figure 5.1.7:** Rotation about the $y$-axis is possible.

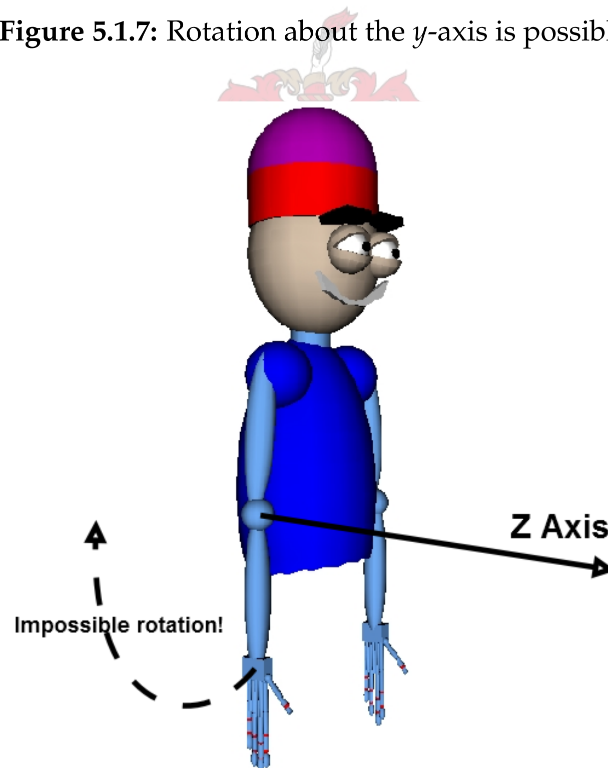



**Figure 5.1.8:** Rotation about the $z$-axis is not possible.

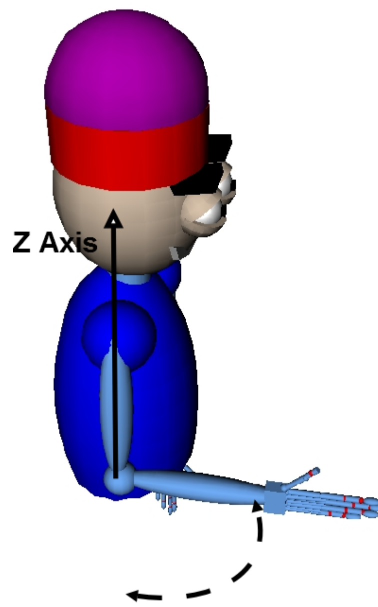



**Figure 5.1.9:** Now, rotation about the $z$-axis is possible.

## 5.2 Future Work

In the previous section we highlighted issues that can be improved by future work. The most important functionality that is still lacking from our animation system is support for facial expressions. The H-Anim standard that we use as a standard for all avatars only support severely limited joints for facial animation. As we can see in figure 2.4.1 on page 24, H-Anim only supports joint positions for the eyebrow, eyelid, eyeball and jaw. These joints are not sufficient to express a smile, a frown or the puffing of the cheeks. In order to add facial expression functionality to our animation system, we have two options. One option would be to extend the H-Anim standard to include more facial joints. The second option would be to use a different standard other than H-Anim, for example the MPEG-4 SNHC standard [22] (see section 2.2.3). We chose to rather concentrate on the first option as it would cause the least modification to the rest of our animation system.

Accurate facial animation is a non-trivial extension to the H-Anim standard, since the extensions have to be generic and function correctly with any facial model. An example extension would be to add non-uniform rational basis-splines (NURBS) [8] surfaces to the face of the avatar. Barker [1] has shown that high quality facial expressions can be created from a face model that is based on NURBS surfaces. However, he also mentions certain limitations of NURBS surfaces. For example, certain facial movements like the puffing of the cheeks are difficult to model using NURBS surfaces based on muscle models. Accurate models for the tongue and teeth cause similar problems. The addition of the NURBS surfaces would need to be assimilated into the concept of a joint and surface, since the H-Anim standard is based on the fact that any avatar can be described as a standard collection of joints and surfaces. If NURBS surfaces are added to the avatar model without defining them as a collection of joints that function similarly to the rest of the standard collection of joints, generic animation of the face would not be possible.

Another possibility would be to add multiple joints, that only translate instead of rotate, to the face of the H-Anim joint hierarchy. Every joint is associated with a surface that represents facial muscles and segments of skin. By translating a

Pectora roborant cultus recti

collection of these joints, one would be able to animate facial expression. The accuracy of these animations depends on the number of joints that are available in the face. If more joints are available, more accurate animations can be created. However, as more joints are added to the face, the creation of the input script that describes the facial expression, becomes more complex. Since a human author has to transcribe the input script, it quickly becomes impractical to add more joints.

The question of how to extend the H-Anim standard to include facial expressions while maintaining the generic design of the standard, is one that requires further research. This question must be answered before this design can be viable, since a signing avatar requires facial animation in order to accurately communicate sign language gestures.

A sign editor tool that will allow the user to generate input scripts, without the need to manually work out the individual joint rotations that make up a gesture, is currently being developed for the SASL-MT project [21]. The sign editor will allow the user to create gestures using a GUI interface by directly manipulating the limbs of the avatar and exporting the result as an input script to the animation system. This tool will facilitate the creation of sign language gestures, since the author would not need any prior knowledge of the coordinate system or the angle notation that the animation system uses. The tool is under development but we include a screen shot from the current version in figure 5.2.1.

In the previous section we mentioned three other less important improvements that can be made to the animation system. Firstly, future developers should find a way to analyse the avatar model in order to determine the relative dimensions of the humanoid it represents. This information should then be used to fine-tune the animations specified in the input script in order to prevent the problem mentioned in section 5.1.2. If this can be accomplished, input scripts would be truly generic in the sense that they can be applied to any avatar without the need for manual fine-tuning. The second improvement would be to adapt the main animation module of our system to use quaternion angles instead of Euler angles. The more intuitive Euler angle notation could still be used in the input script, since Euler angles can be converted into quaternion representation us-

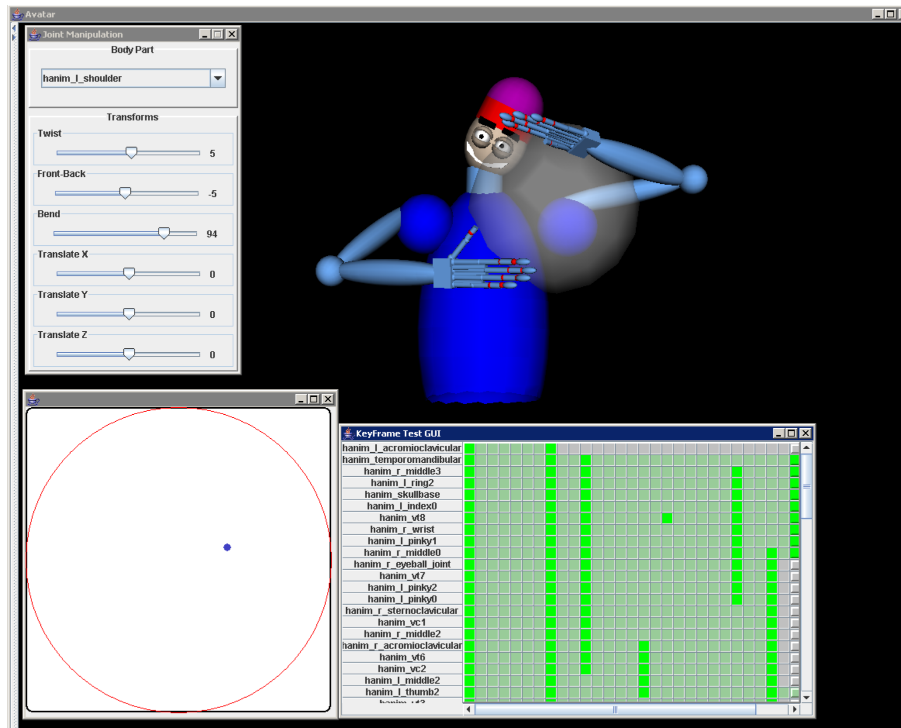



**Figure 5.2.1:** A screen shot from the sign editor tool.

Pectora roborant cultus recti

ing the Java3D library. However, adapting the animator module to rather use quaternion representation would eliminate the gimbal lock problem and would therefore generate more robust animations. It would also simplify integration into other sign language projects, since users that wish to add parser modules would not have to be concerned with the individual axes of rotation and the order in which they should be processed. Lastly, if a way can be found to dynamically relax restrictions on joint rotations, the system would be made more robust since users would not be able to accidentally generate impossible animations.

# Chapter 6

# Conclusions

In everything one must consider the end.

*– J. de la Fortaine (1621-1695)*

The aim of this study was the design of a generic avatar animator for use in sign language projects. Our design is aimed at the SASL-MT project [35] but, due to its design, can easily be adapted for use in other sign language projects.

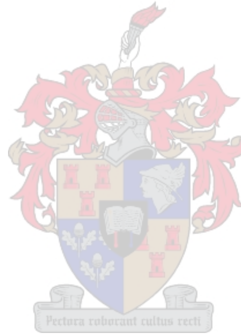
The primary difference between our design and most other avatar animator designs, lies in the fact that our design is not specific to any sign language or to any avatar appearance. The purpose was to design an avatar animator that is generic enough to animate any reasonable avatar and to accept input in any gesture notation with minimal adaptation.

In order to be as generic as possible, our design was separated into three modules that can operate independently from each other (see figure 4.1.1 on page 36). The *animator* module is central to our design and receives input from both the other modules, namely the *parser* and the *renderer*. Both the parser and renderer module are pluggable. This means that they can easily be replaced by different implementations without changing any code in the rest of the system. This is a useful feature in the event where the system is to be adapted to use a different input notation or a different avatar standard.

In our current system we adopted the H-Anim standard [13] for humanoid animation as our standard for reasonable avatars. This means that any avatar that conforms to the H-Anim standard can be animated with our current system.

We designed a simple gesture notation for illustrative purposes based on the XSTEP notation [11] for humanoid animation, and called it *SignSTEP*. We used XML to define SignSTEP and implemented a parser module that uses a readily available XML parser to parse the input into the queue structure that the animator module requires.

We demonstrated that the system can produce recognisable gestures on H-Anim compliant avatars using the SignSTEP notation. Due to the lack of facial expression support, we cannot animate all gestures and some gestures produce higher quality animations than others. Many enhancements can be made to improve the functionality and performance of the system but the primary goal of the project, namely a working design, was accomplished.



# Appendix A

# The SignSTEP DTD

```
<!--
        This is the SignSTEP DTD
        Author: Jaco Fourie
        E-mail: jfourie@cs.sun.ac.za

        This DTD is based on the XSTEP DTD that can be found at
        http://wasp.cs.vu.nl/step/xstep/translation/step.dtd
        Original authors of STEP can be contacted at http://wasp.cs.vu.nl/step/xstep/
-->


<!ENTITY % BodyPart     "(l_shoulder|r_shoulder|l_hip|r_hip|l_elbow|r_elbow
                          |l_thumb1|l_thumb2|l_thumb3|r_thumb1|r_thumb2|r_thumb3
                          |l_index1|l_index2|l_index3|r_index1|r_index2|r_index3
                          |l_middle1|l_middle2|l_middle3|r_middle1|r_middle2|r_middle3
                          |l_ring1|l_ring2|l_ring3|r_ring1|r_ring2|r_ring3
                          |l_pinky1|l_pinky2|l_pinky3|r_pinky1|r_pinky2|r_pinky3
                          |l_knee|r_knee|l_wrist|r_wrist|l_ankle|r_ankle
                          |humanoidRoot|humanoid|skullbase|sacroiliac
                          |lower_neck|upper_neck)">
<!ENTITY % Speed        "(fast|slow|intermedia|very_fast|very_slow)">
<!ENTITY % Action       "(seq | par | turn | trans)">

<!ENTITY % TransElement1              "((dir),(speed))">
<!ENTITY % TransElement2              "((speed),(dir))">
<!ENTITY % TransElement         "(%TransElement1; | %TransElement2;)">

<!ENTITY % TurnElement1         "((rotation),(speed))">
<!ENTITY % TurnElement2         "((speed),(rotation))">
<!ENTITY % TurnElement          "(%TurnElement1; | %TurnElement2;)">
```

```
<!ELEMENT signstep (head?, library+)>

<!ELEMENT head (world?,start?)>

<!ELEMENT world EMPTY>
<!ATTLIST world
        url CDATA #REQUIRED
        >

<!ELEMENT start EMPTY>
<!ATTLIST start
        action CDATA #REQUIRED
        library CDATA #IMPLIED
        >

<!ELEMENT set_tempo EMPTY>
<!ATTLIST set_tempo
        value CDATA #REQUIRED
        >


<!ELEMENT library (action)* >
<!ATTLIST library
        name CDATA #IMPLIED
        >
```

```
<!ELEMENT action %Action; >
<!ATTLIST action
        name CDATA #IMPLIED>



<!ELEMENT turn %TurnElement;>
<!ATTLIST turn
        part %BodyPart; 'l_shoulder'>

<!ELEMENT trans %TransElement;>
<!ATTLIST trans
        part %BodyPart; 'l_shoulder'>


<!ELEMENT time EMPTY>
<!ATTLIST time
        value CDATA #REQUIRED
        unit (second|minute|beat) #REQUIRED>
```

```
<!ELEMENT speed EMPTY>
<!ATTLIST speed
        value %Speed; #REQUIRED>


<!ELEMENT dir EMPTY>
<!ATTLIST dir
        axis CDATA #REQUIRED
    distance CDATA #REQUIRED>


<!ELEMENT rotation EMPTY>
<!ATTLIST rotation
        axis CDATA #REQUIRED
        angle CDATA #REQUIRED>


<!ELEMENT par (%Action;)+ >

<!ELEMENT seq (%Action;)+  >
```

# Bibliography

[1] Barker, D. (2005). Computer Facial Animation for Sign Language Visualisation. Master's thesis, University of Stellenbosch. `http://www.cs.sun.ac.za/~dbarker/`.

[2] Dutton, J. A. (2001). Developing Articulated Human Models from Laser Scan Data for Use as Avatars in Real-Time Networked Environments. Master's thesis, U.S. Naval Postgraduate School. `http://www.stormingmedia.us/authors/Dutton_James_A_.html`.

[3] Efthimiou, E., Sapountzaki, G., Fortinea, S., and Karpouzis, K. (2004). Developing an e-Learning Platform for the Greek Sign Language. *Lecture Notes in Artificial Intelligence*, 3118:1107–1113.

[4] Fant, L. (1994). *The American Sign Language Phrase Book*. Contemporary Books, Lincolnwood, Chicago.

[5] Foley, J., Van Dam, A., Feiner, S., and Hughes, J. (1997). *Computer Graphics Principles and Practice*. Addison-Wesley Publishing Company.

[6] Francik, J. and Fabian, P. (2002). Animating Sign Language in the Real Time. In *20th IASTED International Multi-Conference on Applied Informatics (AI2002)*, pages 276–281.

[7] H-Anim Examples (2004). `http://www.ballreich.net/vrml/h-anim/h-anim-examples.html`.

[8] Hill, F. J. (2001). *Computer Graphics Using OpenGL*. Prentice Hall, Upper Saddle River, NJ 07458.

[9] Hodgins, J. K. (1998). Animating Human Motion. *Scientific American*, 278(3):64–69.

[10] Horstmann, C. (2006). *Big Java*. John Wiley & Sons.

Pectora roborant cultus recti

[11] Huang, Z., Aliens, A., and Visser, C. (2005). STEP: A Scripting Language for Embodied Agents. `http://wasp.cs.vu.nl/step/step.html`.

[12] Huenerfauth, M. P. (2002). A Survey and Critique of American Sign Language Natural Language Generation and Machine Translation Systems. Technical report, Computer and Information Sciences University of Pennsylvania. `http://www.seas.upenn.edu/~matthewh/research.html`.

[13] Humanoid Animation Group, T. (2006). The H-Anim Homepage. `http://www.h-anim.org`.

[14] Kennaway, R. (2002). Synthetic Animation of Deaf Signing Gestures. *Lecture Notes in Artificial Intelligence*, 2298:146–157.

[15] Kennaway, R. (2003). Experience with and Requirements for a Gesture Description Language for Synthetic Animation. *Lecture Notes in Artificial Intelligence*, 2915:300–311.

[16] Klingener, F. (2001). Summary of Dual and Quaternion Mathematics for Kinematics. `www.BrockEng.com/VMech/Quaternions/kinemath.pdf`.

[17] Martin, J. (2003). A Linguistic Comparison: Two Notation Systems for Signed Languages. Technical report, Western Washington University.

[18] Nadeau, D. R. (1998). The VRML Repository. `http://vrml.sdsc.edu/`.

[19] OpenGL (1998). The OpenGL Repository. `http://www.opengl.org`.

[20] Petriu, M., Georganas, N., and Whalen, T. (2003). Development of a Humanoid Avatar in Java3D. In *Proceedings of the IEEE International Workshop on Haptic, Audio and Visual Environments and their Applications (HAVE'2003)*, volume I, pages 2033–2036.

[21] Potgieter, D. (2006). A Sign Editor Tool. `http://www.cs.sun.ac.za/~dpotgieter/`.

[22] Preda, M. and Prêteux, F. (2002). Critical Review on MPEG-4 Face and Body Animation. In *Proceedings of the IEEE International Conference on Image Processing (ICIP 2002)*, pages 505–508.

[23] Prillwitz, S., Leven, R., Zienert, H., Hanke, T., and Henning, J. (2005). HamNoSys version 4.0: HamNoSys Homepage. `http://www.sign-lang.uni-hamburg.de/Projekte/HamNoSys/HNS4.0de/Inhalt.html`.

[24] Selman, D (2002). *Java3D Programming*. Manning Publications.

[25] Smith, K. C. and Edmondson, W. H. (2004). The Development of a Computational Notation for Synthesis of Sign and Gesture. *Lecture Notes in Artificial Intelligence*, 2915:312–323.

[26] Stokoe, W. (1976). *A Dictionary of American Sign Language on Linguistic Principles (New Edition)*. Linstock Press, Silver Spring, Maryland.

[27] Sun Microsystems (2000). *Getting Started with the Java3D API*. Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

[28] Sun Microsystems (2006). The Java3D API. `http://java.sun.com/products/java-media/3D/`.

[29] Sutton, V. (1996). Sign Writing Web Site. `http://www.SignWriting.org`.

[30] The CyberVRML97 Homepage (2005). `http://www.cybergarage.org/vrml/cv97/cv97java/index.html`.

[31] The DirectX Homepage (2006). `http://www.microsoft.com/directx`.

[32] The W3C Consortium (2001). The W3C XML Homepage. `http://www.w3c.org/xml`.

[33] The Web3D Consortium (2006). Web3D Homepage. `http://www.web3d.org/`.

[34] The Xj3D Development Team (2005). The Xj3D Homepage. `http://www.xj3d.org`.

[35] Van Zijl, L. (October 2006). South African Sign Language Machine Translation Project. In *8'th International ACM SIGACCESS Conferance on Computers and Accessibility (ASSETS'06)*.

[36] Vcom3D (2006). The Vcom3D Homepage. `http://www.vcom3d.com`.

[37] www.lifeprint.com (2006). `http://www.lifeprint.com/asl101/pages-signs/t/thankyou.htm`.

[38] Yang, X., Petriu, D., Whalen, T., and Petriu, E. (2005). Hierarchical Animation Control of Avatars in 3D Virtual Environments. *IEEE Transactions for Instrumentation and Measurement*, 54:1333–1341.

Pectora roborant cultus recti

[39] Yeates, S., Holden, E.-J., and Owens, R. (2003). An Animated Auslan Tuition System. *Machine Graphics & Vision International Journal*, 12(2):203–214.