1-1-2004

The vjAvatar library: a toolkit for development of avatars in virtual reality applications

Justin Allen Hare

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
Hare, Justin Allen, "The vjAvatar library: a toolkit for development of avatars in virtual reality applications" (2004). Retrospective Theses and Dissertations. 20575.
https://lib.dr.iastate.edu/rtd/20575

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
The vjAvatar Library: A toolkit for development of avatars in virtual reality applications

by

Justin Allen Hare

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Carolina Cruz-Neira, Major Professor
Yan-bin Jia
Adrian Sannier
James Oliver

Iowa State University
Ames, Iowa
2004

Copyright © Justin Allen Hare, 2004. All rights reserved
Graduate College
Iowa State University

This is to certify that the Master's thesis of

Justin Allen Hare

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
# TABLE OF CONTENTS

**LIST OF FIGURES**

- v

**ABSTRACT**

- vi

**CHAPTER 1 – INTRODUCTION**

- 1
  - Research Problem 1
  - Statement of Purpose 2
  - Scope of Work 3

**CHAPTER 2 – REQUIREMENTS FOR AN AVATAR TOOLKIT**

- 5
  - Performance 5
  - Versatility 6
  - Ease of Use 7
  - Dynamic Motion 9
  - Extensibility 10

**CHAPTER 3 – CHARACTER REPRESENTATION METHODS**

- 11
  - Static Models 11
  - Keyframed Models 12
  - Video Avatars 12
  - Billboarded Images 14
  - Skeletal Models 15
  - Hybrids 17

**CHAPTER 4 – AVAILABLE SOLUTIONS**

- 18
  - Film and Game Development Software 18
  - JACK 19
  - SGI Performer 20
  - DI-Guy 20
  - Cal3D 21

**CHAPTER 5 – CONCEPT AND DESIGN**

- 22
  - Design Overview 23
  - Avatar Creation 24
  - Resource Management with the Avatar Factory 25
  - Controlling Avatar Motion 28
  - User-Definable Functionality with Plug-ins 28

**CHAPTER 6 – TECHNICAL DESIGN**

- 30
  - Character Animation System 30
Virtual Reality Framework 30
Graphics Library 32

CHAPTER 7 — IMPLEMENTATION AND FEATURES 34
Extensions to Cal3D 34
Resource Management 39
Time Synchronization 43
Math in vjAvatar 44
Character Skinning 46
Pluggable Functionality 46
Dynamic Motion Plug-in Implementation 49
Scene Graph Integration 50

CHAPTER 8 — RESULTS AND EXAMPLES 52
The Virtual Hindu Temple 52
The Multi-mission Aircraft (MC2A) 54
Template Application Collection 56
AvatarNET 57
Performance Metrics 58

CHAPTER 9 — CONCLUSIONS 62
Performance 62
Versatility 62
Ease of Use 63
Dynamic Motion 64
Extensibility 64

CHAPTER 10 — FUTURE WORK 65
Facial Animation 65
Speech Communication 65
Component Separation 66
Animation System Abstraction 67

REFERENCES 68

ACKNOWLEDGEMENTS 72
LIST OF FIGURES

Figure 1. A video avatar in a 3D environment 13
Figure 2. Image series making up a billboarded avatar 14
Figure 3. Skeletal avatar with mesh overlay 16
Figure 4. The avatar factory with prototypes, resources, and instances 26
Figure 5. Integration of base libraries into a VR application with vjAvatar 33
Figure 6. Different skeletal models with some of the same joints 37
Figure 7. Avatar resources: skeleton, mesh, animation, and material 40
Figure 8. Illustration of right- and left-handed rotation axes 45
Figure 9. Worshippers in the virtual Hindu temple 53
Figure 10. Avatars working inside the MC2A aircraft 55
Figure 11. Networked collaboration on an engineering task 57
Figure 12. Networked combat inside AvatarNET 58
Figure 13. Memory usage using an increasing number of textured avatars 59
Figure 14. Memory usage using an increasing number of non-textured avatars 60
Figure 15. Application framerate decreasing as number of avatars increases 61
Figure 16. Library component separation with glAvatar and vjAvatar 66
ABSTRACT

The potential of animated characters in virtual environments is great. They can be used to populate an environment and give it a life-like feel, or to represent real users in networked collaborative environments, becoming a vehicle of human-to-human communication.

Many researchers have worked with using animated characters in virtual environments for a variety of purposes. The approaches, for the most part, have been ad hoc solutions that are were not designed for general-purpose use. Many issues involved in creating a reusable implementation that can be used in the wide variety of VR systems have not been addressed.

This thesis discusses requirements for a character toolkit that can be used for many different purposes in VR applications, the design and implementation of it, problems encountered, and the experiences and knowledge gained from developing real applications with it.
CHAPTER 1 - INTRODUCTION

Research Problem

Virtual reality has become a sophisticated field with a wide array of practical applications – from industrial design to historical research and military planning. Advances in computation power and graphical capabilities have allowed VR developers to create virtual environments with ever-increasing detail. Developers are constantly striving for more immersive environments.

Using virtual humans is one of the best ways to engage users and make the environment more believable. Developers can give a life-like feel to their environments by adding animated characters with a variety of purposes.

Animated characters – or “avatars” – can be used to populate an environment. The presence of other people is something that we often take for granted in the real world. In virtual environments, human characters can add a realistic feeling of presence that helps the user to believe that the environment is real and not just a computer simulation.

Avatars are commonly used to facilitate human-to-human communication in networked virtual environments. Users from different places in the world can enter a shared virtual environment and communicate with other users in a variety of ways. At the most basic level, avatars can show the presence of other users. They can act as a placement marker to show where various users are at and what they are doing. More complex implementations can show detailed information about a user, such as body movements, gestures, facial expressions and speech.

Avatars can also be used as a vehicle for conveying information to users. Regular desktop applications typically do this through command line text output or popup message boxes. But in VR applications, these features are either nonexistent or disrupt the user’s interaction with the environment. It is important, above all, to maintain the user’s sense of

---

1 The term “virtual human” generally refers to any humanoid character, whether it is computer-controlled or represents a live user.

2 The terms “character” and “avatar” will often be used interchangeably in this text and encompass any kind of animated figure in an environment.
immersion and avatars can help with this. They can appear as a natural part of the environment and are a good method of communication.

There have been a number of implementations of avatars in virtual environments, the majority of which are ad hoc solutions. They are designed for a specific purpose and are not easily reusable. Because they are usually created from scratch, duplication of effort is common.

With the rise of consumer 3D video cards and the proliferation of computer graphics in movies, there are many commercial solutions that address the problems of creating and using 3D animated characters. Many of these offer excellent features, but are geared towards game development or movie production and do not lend themselves well to being used in complex virtual reality systems. Additionally, these packages are often priced out of reach for many academic budgets.

VR developers need an avatar toolkit that is designed to meet the special needs of VR applications and the systems they run on. Such a toolkit would aid developers by not forcing them to create an application-specific implementation. It would alleviate the problems of dealing with the differences between various VR systems. It would also allow a developer to customize functionality for their needs, which cannot be known ahead of time.

**Statement of Purpose**

The research presented in this document addresses the problems of integrating avatars into virtual environments with the specification and implementation of a toolkit to construct, display, and control avatars. We call this toolkit the *vjAvatar library*. This toolkit allows the addition of animated characters into VR applications in a way that lets developers focus on the interactivity of the characters rather than dealing with the specifics of complex VR systems.

During the course of development, we analyzed the practical requirements of integrating avatars into VEs, looked at previous VR work as well as game and movie production, designed and implemented the toolkit based on these requirements, and tested its effectiveness during the course of various kinds of VR application development.
Scope of Work

1. Define needs for avatars in virtual environments

   We looked at avatars from a practical standpoint – that of a developer wanting to create and integrate characters into their interactive VR applications. We wanted to define requirements and desirable features in an objective manner so that existing approaches could be compared and contrasted alongside one another. These requirements would play a big role in the design and implementation of the toolkit.

2. Analyze existing work

   There has already been a great deal of work done regarding avatars in virtual environments. We examined previous work in VR research toward avatars and looked at what researchers were trying to accomplish through their use. We looked at how various VR researchers implemented avatars for particular uses and analyzed their strengths and weaknesses.

   It was also apparent that there was a lot of crossover work done in the commercial sector with animated characters in modern cinema and video games. We surveyed an array of available packages aimed at these markets to see how their techniques could be applied to virtual environments.

3. Design and implement the vjAvatar library

   Based on our requirements and analysis of existing work, we began the design of the toolkit. This included deciding on a feature set, laying out the software architecture, and defining a programming interface. Once a solid design had been reached, the implementation of the toolkit began. During the first implementation, some technical limitations became apparent and necessitated changes to the toolkit design.
4. Test in VR applications

In order to gauge how effective the vjAvatar library would be at fulfilling the needs of VR developers, we looked at several different projects that had a need for avatars. These projects used avatars for widely varied purposes, allowing vjAvatar to be tested in a variety of arenas. Due to vjAvatar and the VR projects being located in the same lab, the toolkit was built with a great deal of developer feedback. In fact, some of the projects were developed in parallel with vjAvatar; the project development often directly drove the iterative design and implementation process. In many cases, we were able to refine and add features almost interactively as project needs became apparent.
CHAPTER 2 – REQUIREMENTS FOR AN AVATAR TOOLKIT

There are a number of key requirements that are important for an avatar toolkit that is meant to work with virtual reality applications.

- **Performance** – The toolkit must be able to perform fast enough for real time applications and make efficient use of computing resources.
- **Versatility** – It must be able to function on many different kinds of VR systems.
- **Ease of Use** – It should be easy to integrate avatars into virtual environments.
- **Dynamic Motion** – There needs to be a way to programmatically animate avatars at runtime.
- **Extensibility** – Developers should be able to extend the functionality of the toolkit to fit the needs of their applications.

**Performance**

Fast performance is perhaps the most important requirement for an avatar toolkit. An avatar can impact the overall performance of an application by taking time and resources that would otherwise be available for other parts of an application. The more visual complexity that an avatar possesses, the more time it will take the application to draw the avatar. Different avatar representations (discussed in chapter 3) vary greatly in the amount of memory used. Additional computation time can be consumed when an avatar is animated and motion needs to be calculated.

In virtual reality applications, fast performance is essential to maintain the user’s sense of immersive and interactivity. The frame rate, or the number of images generated per second, is one of the best benchmarks for determining the performance of a VR application. Therefore it is critical that the performance of each individual avatar is as efficient as possible to avoid bogging down the overall performance of the entire application. The faster
that the avatar toolkit can update and render characters to the screen, the more this will help achieve a high frame rate. The bare minimum needed to maintain the user’s sense of immersion is typically 12-15 Hz [47].

Avatars often have to react to their surrounding environment or interact with a user. When they do so, they should respond fast enough to maintain the interactive nature of the application. For instance, if a user bumps into an avatar standing near him, the user will expect the avatar to fall down or move out of the way as soon as the collision occurs — not 5 seconds later.

When avatars are used to represent the actions of real users in networked environments, fast response times are essential to facilitate communication and cooperation between participants. If two users are working together to lift a large beam in the assembly of a building, their movements need to be registered by the avatars as soon as the users move. If the users are seeing delayed movements of each other, then their physical reactions will also be delayed, making cooperation extremely difficult.

When performance is degraded, the effects of cybersickness [47] can result. Aside from losing their sense of immersion, users can also experience eyestrain, dizziness, and nausea.

Memory usage can also become an issue on less powerful computers or with applications that push the boundaries of a computer’s capabilities. By efficiently managing the resources of its avatars, the toolkit can help performance by using as small of a memory footprint as possible. This allows applications to be run on less powerful computers and developers to apply the computing power to other parts of their applications.

Versatility

An avatar toolkit needs to be usable on a diverse array of computer systems. Virtual reality systems can be put together in a wide variety of configurations. This includes everything from a standard desktop PC with a keyboard and mouse to a high-end supercomputer with multiple screens and a complex motion tracking system. The toolkit should be designed to work with as many of these systems as possible.
There is a delicate balance between versatility and performance, and the two characteristics are often at odds with each other. For example, multi-threaded computation could potentially enhance performance of the toolkit, but this could limit application developers to using particular threading software and/or operating systems. The choice of a graphics library, such as OpenGL or Direct3D, can produce the same limitations. Additionally, there are many differences between compilers that could be supported. Some compilers do not fully support all of the features of a programming language, or exhibit significant performance drops when certain features are used. One particular compiler might have the capability of code optimization where another may not and there could be a performance gap between the two.

Cross-platform capability is also desirable, since there is no standard operating system for virtual reality development. Windows, Linux, and Irix are among the most commonly used, but VR development is not limited to them. The more operating systems the toolkit can support, the more people it can potentially benefit.

The avatar toolkit also needs to be versatile enough to handle any arbitrary type of avatar that a developer would want to use. VR applications are even more diverse than the systems they are designed to run on. One developer might want human-like avatars to represent real users. Another might want avatars that look like automotive assembly plant robots. Yet another developer might use avatars to represent autonomous military aircraft. The toolkit should allow for anything the environment designer has in mind.

Ease of Use

Designing the avatar toolkit to be easy to use goes a long way towards making it valuable to developers. The less work they have to do to integrate characters into their environments, the better. Making the toolkit easy to use affects several key areas - programming, character creation, and long-term compatibility.

Visually appealing avatars are inherently difficult to create because they are complex. In terms of graphical complexity, they can be just as detailed as a painting or sculpture, and require some degree of skill to create. They are difficult to control because each body part
has the capacity to move independently. Coordinated motion needs to be orchestrated in a manner that appears fluid and lifelike. Considered from an application developer’s standpoint, avatars are difficult to manage in terms of computation resources. There is a limited amount of memory in which to store avatars, and they must be rendered quickly enough to maintain real-time speeds.

Many of the details of animating an avatar can be encapsulated into a high-level programming interface. This allows the developer to manipulate an avatar using intuitive commands, lowering the learning curve and speeding up the development process. This can also drastically cut down the lines of code required to integrate avatars into an environment and make the code more readable.

Integrating the toolkit with a common graphics API simplifies development by freeing the programmer from having to write code that renders their characters to the screen. Rendering code can be included in the toolkit that is already optimized and makes use of some of the more advanced features of a graphics library. This allows the characters to be rendered quickly without requiring the developer to be an expert at computer graphics.

It should also be easy for developers to create new characters that are customized for their applications. One of the best ways to allow this is to integrate the toolkit with a 3D modeling package. This gives developers the power to leverage all of the tools that the 3D modeling software provides, making it possible to create believable characters in a relatively short time period. It also allows artists with no experience in virtual reality development to contribute their talent and vision to an application.

Long-term compatibility is also a desirable goal. Software libraries inevitably go through a number of revisions throughout their lifetime. New features are added, bugs are fixed, and processes are streamlined. When upgrading the avatar toolkit, it should be easy to transition from an old version to the new one. This will encourage developers to always use the latest version available, which most likely offers the best performance and richest feature set.
Dynamic Motion

The interactive nature of virtual environments demands that an avatar toolkit support dynamic motion. The concept of dynamic motion refers to any kind of character movement or animation that is generated at runtime. This is different from scripted animation, which is entirely defined ahead of time—the character merely acts out predetermined animations. One example of using dynamic motion would be to automatically twist a character's head to look at a bird as it flies through the trees.

Avatars that represent real users in virtual environments need to mimic the actions of the users, which simply cannot be predetermined. Perhaps a user is wearing a full body suit that can track the movements of her arms and legs, and an avatar is used to represent her in a networked virtual environment. Dynamic motion will have to be applied to the avatar to make it have the exact movements of the user.

Avatars often need to react in a realistic manner to their surroundings and any users that occupy the virtual space. For example, an avatar could be used to represent a crash test dummy in a vehicle collision simulation. To get an accurate depiction of what happens to the dummy during a car crash, the avatar needs to move according to the laws of physics as the car’s cab buckles and bends around it. Assuming the operator has control of how fast the car is going, changing the vehicle’s speed will change the shape of the car as it crashes. This makes it impossible to create a scripted animation of the dummy that accurately represents its motions for every possible crash outcome.

Body language can also be a method of communication, conveying attitude, disposition, and emotion. A nervous person might appear jittery. A hyperactive child might bounce up and down. An older person might appear to have poor posture. Ken Perlin has demonstrated the expressive capacity of avatars' body language [4]. Such body motions would most likely be implemented as dynamic motion rather than scripted animations.

In many cases, scripted animation and dynamic motion need to be blended together for effect. As important as dynamic motion is for avatars in virtual environments, scripted animations are still extremely useful. Going back to the example of a character watching a bird fly through the forest, he could also be walking along a trail at the same time. The
developer could use a scripted walking animation for the character's body while applying dynamic motion to its head to keep it looking at the bird.

**Extensibility**

It should be easy for developers to extend the avatar toolkit to support functionality suited to their own needs. It is unreasonable to believe that an avatar toolkit could include enough features that would satisfy each and every developer that wanted to use animated characters in their VR application. It is therefore a better idea to allow developers to define their own functionality if they want.

This can be accomplished by creating a "pluggable" architecture, in which certain pieces of the toolkit's functionality can be swapped out for code written by the developer. This allows developers to write their own custom functionality without having to modify the toolkit's source code. Different plug-ins can be switched in and out at runtime, providing a great deal of flexibility.

There are several pieces of necessary functionality that are good candidates for allowing plug-ins. Dynamic motion, for instance, is one of the major requirements for the toolkit. Many types of dynamic motion programming have to be customized for specific avatars that have been created by the application developer. Specialized rendering plug-ins would be beneficial to developers who want to use special effects and advanced rendering techniques that the base toolkit doesn't support. Custom character loading plug-ins could also help set up non-standard characters that use developer-created features.

Perhaps the best way to add extensibility to the toolkit is to release it under an open source license. This gives developers the ultimate in customizability — they can modify and adapt the base code to do anything they need. Developers could, for example, modify the toolkit to handle video avatars or billboarded characters. They could even alter it to support a different graphics API, or extend it to work on a non-supported platform.
CHAPTER 3 – CHARACTER REPRESENTATION METHODS

There are many different ways to go about implementing animated characters for real-time applications. The variety of methods offers a tradeoff between realism, performance, utility, and ease of implementation. It is difficult to say whether one technique is strictly better than any other - each has its own advantages and disadvantages. The most popular representations are:

- Static models
- Keyframed models
- Video avatars
- Billboarded images
- Skeletal models

Static Models

One of the simplest methods is to use static (non-animated) models to visually represent an avatar and give it a location in the environment. This is easily implemented, can allow for any kind of character that the designer wants, and does not require much computation overhead.

DIVE [11] used static models to represent “actors”, which could be computer-controlled characters or avatars representing live users in networked virtual environments, which was the primary focus of the DIVE project. The static models could move around the environment space, but were not animated.

The ALICE project [13] uses a combination of static models and transformations. Avatars can be comprised of various body parts, which can be manipulated independently to achieve realistic motion programmatically.

Non-animated models are very limited in their expressiveness and realism. Since one of the goals of vjAvatar is to add life-like feeling to virtual environments, restricting the developer to static models is something to avoid.
The NICE project [10] extended static models to approximate the location and arm motions of users interacting with their virtual environments.

**Keyframed Models**

Static models can be enhanced using a keyframed animation technique. Much like traditional cell animation that has been the mainstay of the cartoon industry in the past, keyframes can be applied to 3D models. An artist creates a series of freeze-frame shots that, when viewed rapidly one after another, make the model appear to be in motion. The capability of animation allows developers to use more expressive and life-like characters in their environments.

This presents a resource problem in real-time environments, however. In order to maintain the perception of fluid animation, at least 10-15 keyframes are required per second. Since each of these keyframes contains a full description of the character, the memory consumed can quickly get out of hand. Excessive load times and poor performance can result. Keyframed animation also limits us to playing back predefined animations.

**Video Avatars**

One of the most popular and highly researched techniques for creating avatars in virtual environments is video avatars. This involves capturing real video of users with a camera, processing it, and putting it into the virtual environment (see Figure 1). This can be as simple as texture mapping a flat video image onto a surface, as described in [7] or as complex as using cameras to get a full 3D view of a user, as in [9]. Additionally, streaming audio can be incorporated to allow voice communication. The Virtual Harlem Project [12] used video avatars to bring old photographs to life in an interactive tour of Harlem, New York, in the 1920s and 30s.

Most of the research on video avatars has been regarding networked virtual environments where live users can communicate and collaborate with each other using avatars as a medium. As they can convey facial and gestural information very well via
photorealistic video, these avatars can very accurately depict the person whom they represent. Combined with voice audio, video avatars come as close to face-to-face communication as is possible in a virtual environment.

Cameras, however, are limited to capturing flat 2D images that are awkward to integrate into a 3-dimensional environment. In many types of VR systems, such as one that uses stereo imaging, flat images clash with other 3D objects and disturb the user's sense of immersion because they cannot convey depth information. Video avatar techniques that build complete 3D models can alleviate this problem, but are significantly more difficult to implement and require more equipment such as motion tracking gear or extra cameras.

![Figure 1. A video avatar in a 3D environment](image)

One further disadvantage is that video avatar systems are primarily designed to represent live users or play back the recorded actions of real people. There is not much leeway to allow for dynamic changes to the avatars at runtime. Essentially, video avatars are scriptable but not very programmable. There is also a lack of artistic flexibility since everything is geared towards video feeds. While it is easy to create a video avatar based on a
real person, it would be relatively time-consuming for an artist to create one from scratch frame by frame.

**Billboarded Images**

When the first 3D video games started being produced in the mid-90s, such as Doom [15] and Duke Nukem 3D [16], animated billboarded images were a common way of representing characters. Billboarded images offer a very computationally efficient way of adding characters into a three-dimensional space. Instead of drawing a full 3D representation of a character consisting of hundreds or thousands of polygons, a single polygon can be drawn with an image pasted on it (like a billboard advertisement). This polygon is constantly rotated to always face the viewer, no matter where it is.

Typically, the image pasted on the billboard changes based on which direction the character is facing relative to the viewer. There are usually front, back, and several side images taken from different angles (see Figure 2).

![Figure 2. Image series making up a billboarded avatar](image)

Since only a single polygon is being used, the billboard technique gives very good performance compared with equivalent 3D models. Even some modern games such as Diablo II [18] and Medieval: Total War [19] still use billboarded images in order to accommodate
large numbers of characters on the screen at one time while maintaining acceptable frame rates.

Billboarded characters can be problematic, however, when the viewer is seeing the character from above or below. Billboards typically only rotate about a single axis, and the closer the viewer gets to this axis, the more likely that the character will look like a flat surface rather than a 3D object. This is known as axial billboarding and is commonly used to represent objects such as trees [45].

A notable extension of basic billboarded characters comes from the field of image-based rendering [50]. By using a higher number of directional samples and an image blending algorithm that "morphs" between adjacent samples, it is possible to create characters that are extremely realistic and appear to be in 3D. As with the basic billboarding method, the amount of detail is only limited by the choice of image resolution. With the blending algorithm, the switch between samples is much less noticeable when the character is rotated. This allows for highly detailed characters that are relatively light on computation.

**Skeletal Models**

Skeletal animation combines keyframed animation of a simple skeleton-like structure with a method of mapping a 3D model around the skeleton's pose [51]. The mapping method is called "skinning" because the model's mesh — or skin — is being wrapped around the bones of the skeleton [5] (see Figure 3). As the skeleton changes its pose, so does the skin. This causes the skin to bend, stretch, and shrink in different places to match the skeleton's pose. Conceptually, skeletal models function analogously to the way real humans' bodies work. Skeletal animation has been used in the computer animation industry for years, but has only recently been introduced in the realm of real-time graphics — most notably in video games.
Skeletal animation offers many advantages over other character animation techniques. Compared to other methods, it is generally easier to animate skeletal characters since only their bones need to be manipulated. A character might have thousands of polygons to make up its mesh, but only a couple of dozen bones. This takes a lot of work off the shoulders of the animator.

Each bone consists simply of a coordinate system transformation – a translation and a rotation. The bones are arranged in a hierarchy, starting with a root bone that marks the origin of the model. Each bone can have any number of child bones. These child bones inherit their parent’s transformation and add their own to the chain. This results in a cascading of transformations that define the overall shape of the skeleton.

Animation of the bones is typically done using keyframes, where the state of individual bones is stored as translation/rotation pair. In three-dimensional space, movement and rotation can be interpolated between different values. Using keyframe interpolation, perfectly smooth animation can be achieved regardless of the actual frame rate that the
application is operating at. This can be important when consistency is needed between computer systems that have differing performance.

Skeletal animation allows developers to build up collections of smaller animations that can be combined to make a wide variety of actions. Because bones inherit their parent’s transformations, it is possible to animate individual parts of a character. For instance, a character’s legs could be animated to make him walk, while a separate animation makes the character’s arm wave. Both of these animations can be used individually or they can be used at the same time, making the character wave while he’s walking.

Compared with keyframe animation of a standard 3D model, skeletal animation needs much less disk and memory storage, but often requires a higher cost in terms of computation time. The memory requirements are lower because the 3D mesh vertex data only needs to be stored once and the bones’ keyframes that make up an animation take up very little space. A full description of the mesh has to be generated each time the character moves, however. This takes a significant amount of processing power and can limit the number of characters that can be active at any given time.

Most importantly, skeletal characters naturally fit in with 3D environments where other character implementations do not, such as some video avatars. Skeletal characters can be constructed of arbitrary shape to conform to the designer’s wishes. This allows a high degree of flexibility and a wide base of applications.

Hybrids

In some cases, it is desirable to use more than one method in tandem. For example, the implementation of virtual humans in VLNET environments uses a combination of video and skeletal methods [8]. Video images are used to represent the faces of avatars, with 3D skeletal bodies that mimic the arm motions and gestures of users.
CHAPTER 4 – AVAILABLE SOLUTIONS

There are already a number of commercially available libraries that can be used for character animation. Many of them are targeted towards film and video game development, but could potentially be applied to virtual reality applications. There is still a great deal of research to be done with avatars, on both the academic and industrial sides. There is a need for a tool that can serve as a research test bed for avatars.

Film and Game Development Software

Kaydara’s Motionbuilder [25] software is one of the most popular packages for doing skeletal model animation, and is commonly used for films, television shows, and video games. Kaydara also offers motion capture software that allows characters to be animated in real time based on the motions of a real person. This could be especially useful in networked virtual environments where the body movements of participants need to be shown to other users.

Granny 3D from Rad Game Tools [26] has a set of functionality that is more in line with the goals of vjAvatar. Granny 3D is specifically designed for integrating animated characters into video games, and follows many of the same design principles that vjAvatar does. It allows artists to use a common 3D modeling package – such as 3D Studio MAX or Maya - to do all the character creation, eliminating the need for character designers to be adept at both modeling and programming. It is also designed to be cross-platform and independent of any specific graphics API. The programming interface includes advanced features like inverse kinematics and collision detection, making dynamic motion possible.

One problem with these libraries is the proprietary nature of their software development kits, which can limit the kinds of systems that can use them. As discussed earlier, VR systems can be more complex than the typical gaming or workstation machine, with multi-pipe video output or clustered configurations. If the software library is not designed with these possibilities in mind, it may be impossible to use them in VR environments. Also, games are usually only played on Windows or Mac operating systems,
so some SDKs will only work on these platforms. Granny 3D, for its part, comes with source code for its SDK and can be modified to suit specific needs.

These libraries are geared for professional development groups working on movies and video games. One of the key differences from these two uses of avatars with virtual reality is that for VR there very little corners that can be cut. On a movie, very complex avatars are rendered off-line and edited for the final cut. On a game, developers can make assumptions on the line of sight of the player, as well as the sequence of events that may happen. In a VR environment, everything is dynamic, in particular when the avatars are tied to an actual human user in a collaborative space. We can’t predict what the avatar will do next, and neither can we pre-compute any avatar “looks”. Most (if not all) of today’s avatar commercial tools do not address the spontaneous nature of VR avatars. Additionally, there is still quite a a great deal of research that needs to be done about the role of avatars in VR, so having an Open Source tool can facilitate this task.

JACK

One of the most famous and widely used implementations of virtual humans is Jack [14], originally created by Dr. Norman Badler and now a commercial product [43] licensed by EDS. Jack is a highly detailed computer model of a human being and its biomechanical movements. It is designed to be very accurate at reflecting the movements of real humans and the effects of strain and discomfort while they are performing actions.

Jack is primarily used in industry for ergonomics testing and human factors experiments to assist in product designs and establishing workplace task procedures. It is well suited to engineering design and testing, but may be overkill for simply adding animated virtual humans into an environment. Developers are limited to using only a couple of different virtual humans, and cannot create arbitrary characters to add to the environment. The human models, being highly detailed and accurate in their movements, also consume a significant portion of computation power. Unless a developer is working with human factors in virtual environments, Jack is probably not the best solution for the job.
SGI Performer

SGI's Performer scene graph [32] is not necessarily a character animation library per se, but it has support for billboarded characters and may be suitable for some developers. Recent versions of Performer have support for image-based rendering of billboarded characters [50] as discussed in the previous section. Performer is a well-established library for 3D graphics development and has a strong following in the VR community. It would be a good choice for those developers who simply want to drop background characters into their environments, without needing true 3D or dynamic motion.

DI-Guy

DI-Guy from Boston Dynamics [52] is a commercial package that directly addresses the needs of VR developers. At the core of DI-Guy is a 3D character animation library specifically designed for developers of interactive simulations, especially military training simulators. The library can fulfill many of the needs of virtual reality developers right out of the box. It allows developers to easily drop 3D characters into their environments and have them perform a wide variety of actions.

With lots of ready-made characters and animations, it can be especially useful for developers who don't want to learn the nuances of 3D character animation. Since most of the details are hidden behind the DI-Guy programming interface, the learning curve to integrate DI-Guy into your application is very reasonable. Other notable features include programmable motion, automatic level-of-detail, and animation blending. It also has support for several common graphics APIs, including OpenGL, DirectX, and SGI Performer.

One of the major drawbacks to using DI-Guy is the limited range of human characters. Boston Dynamics software has been primarily targeted for military and defense simulation purposes [1]. It comes as no surprise, then, that most of the available characters are related to military institutions. There are a few civilian and athlete models available, but are limited in variety. The only way to create new custom characters and animations is though a Boston Dynamics consultant.
Combining this limited application domain with an expensive price tag prevents DI-Guy from being a top choice for general VR development. Military VR developers, however, will likely find it to be a good solution.

Cal3D

Cal3D [3] is an open source alternative to commercial skeletal animation solutions that provides a lot of flexibility in how it can be used. Its core consists of a stable skeletal animation system that is fast enough for real time applications. Cal3D offers similar features to DI-Guy with animation blending and level-of-detail controls, as well as a simple spring system implementation can be used for cloth and hair modeling.

One of Cal3D's most attractive features is the exporter plug-ins for the popular modeling package, Discreet's 3D Studio MAX [28]. Many virtual reality applications can benefit from characters specifically tailored to them, but characters can only be as good as their creators. Because 3D Studio is such a commonly used tool, many computer artists are able to create characters using the same familiar methods they have used in the past without a steep learning curve.

The code itself is written in platform-independent C++ with extensive use of the standard template library [30]. It can be used on many different platforms such as Windows, Linux, IRIX, and Mac OSX. It can also work with any graphics API, since it doesn't specifically deal with visualizing characters itself.

Cal3D’s flexibility is also a drawback when working with it. Developers have to write their own code to load and compose characters, manage shared resources, and write rendering code. Cal3D is packaged with only a few sample characters. Overall, a lot of work is required on the part of both developer and artist before characters can be used in Cal3D applications.
CHAPTER 5 – CONCEPT AND DESIGN

As previously stated, the main goal of the research presented in this document is to design and develop an avatar toolkit that makes it simple for developers to integrate animated characters into their VR applications. This was the driving force behind the design of a toolkit, which we call the vjAvatar library.

The features of the vjAvatar library can be categorized as follows:

- Creation of an avatar’s visual appearance
- Resource management at runtime
- Integrating avatars into a variety of VR systems
- Control of animation and dynamic motion
- Supporting user-defined functionality
- Rendering to the screen

From experience in video games, we know that introducing real-time animated 3D characters is a challenging and complex task. The animation of 3D characters includes the real-time execution of the physics/kinematics model associated with the character, as well as the real-time resolution of lighting for the character.

As an example of that complexity, consider the main character, Tommy Vercetti, in the game Vice City [17]. When Tommy is evading police and collecting protection money, there is a significant amount of complex code running in the background to keep his legs moving in time with each other and updating the surface normals of his pants to make them reflect light properly. These actions are not only restricted to that one character – many other characters in the game may perform similar actions. The implementation of these actions does not necessarily have to be individually coded for each character.

A good toolkit for animated 3D character should provide the developer with enough controls so that characters like Tommy can be manipulated using calls like “run straight forward” or “point the gun in this direction”. Many of the steps necessary to execute these
actions can be hidden from the developer so that they can make their avatars perform actions with simple intuitive commands.

**Design Overview**

This has guided the design of the vjAvatar toolkit. Its goal is to enable application developers to create virtual environments populated with multiple avatars with little coding effort. The vjAvatar library handles the animation model, geometry, and lighting information of the avatars so that VR developers can focus their efforts in the "choreography" of the avatars and their role in the application.

At a high level, vjAvatar can be considered a resource management system for avatars. It focuses on eliminating wasteful memory usage and computation whenever possible. Many avatars may have shared attributes, such as their body geometry or walking motion. An easy approach would load each avatar individually, without taking into account any duplication. In systems with small memory, this would limit the number of avatars that can be used in an application.

For instance, let's say Warren is developing a VR police training simulator and wants to include several SWAT team avatars. Their faces may be different, but their combat uniforms are all exactly alike. Warren can “paint” the clothes of the SWAT team members with the same image. Although each of the avatars is distinct and independent of the others, the vjAvatar library will recognize that they reference the same image and will only load it into memory once, rather than loading it once per character. In this example, vjAvatar not only handles the task of loading images for the developer, it also avoids duplicating load time and storage space.

The wide variety of VR system configurations and platforms had a strong influence on how the vjAvatar library was implemented. VR applications are commonly run on very different systems, including desktop computers, large projection systems, stereo monitors, head-mounted displays, and CAVE-like systems. Our goal was to design a toolkit that could be used with any of these systems without requiring the developer to manage the details of programming for any individual system.
Avatar Creation

The development of the physical appearance of a character is a very time consuming task and requires skills different than those of programming the character's actions in a virtual environment. We can think of several approaches to build the avatars: brute force programming, combinations of ready-made pieces, and development through a 3D modeling package.

The brute force method would require a programmer to actually code the avatar's structure through command of a graphics API such as OpenGL. Obviously this is an approach that should be avoided. No one, especially an artist, wants to build characters from scratch in code — this would be tedious, time-consuming, and unintuitive.

Another alternative is to have a selection of ready-made pieces (heads, torsos, arms, etc.) that the developer could use to construct characters. This makes it easy for non-artistic users to put together functional characters quickly — a sort of Frankenstein rapid prototyping tool. But this places limits on the kinds of characters that developers can use in their environments. The overall bone structure would have to be predetermined, and any body pieces used would have to conform to the size and shape of the skeleton.

The use of a 3D modeling package is a good way to create complex characters, and this is one of our design requirements for the development of the vjAvatar library. The ability to use a 3D modeling package kills two birds with one stone. First, it permits trained artists to contribute their skills without forcing them to learn programming. Second, it allows developers to create any avatar that they want, without being limited to specific kinds of characters.

Having a way to use a 3D modeling program allows artists who have experience with it to leverage the power and tools it offers. This can result in better overall quality in the characters as well as reduced development time while not restricting users to predefined content.

Secondly, one of the deficiencies of some of the other avatar solutions is that they are only focused on human characters. Although they play a big role in making richer virtual
environments and they are commonly used to represent live participants in multi-user environments, human-like characters are not the only kinds of figures that are useful in virtual environments.

For example, if in your application you walk along a city’s streets long enough, you will undoubtedly see people walking dogs, alley cats scavenging, and sewer rats scurrying in the shadows. Skeletal animation can be used to create all manner of beings – from domestic fauna to giant robots and desert lizards to eight-legged alien invaders. A skeletal animation system could even be used to animate a toaster popping up a piece of burnt bread.

The key here is to allow the figures to be constructed of any arbitrary bone and mesh structure. This lets the environment designers be as creative as they want when creating characters to use, without being restricted to human-like skeletal structures.

**Resource Management with the Avatar Factory**

To address the avatar resource management issues, vjAvatar introduces the concept of the “avatar factory”. This paradigm is analogous to a factory line: all avatars are created through the factory’s assembly lines and depend on it for access to their resources. The factory, in turn, controls access to the resources to prevent conflicts - such as when one avatar wants to read a shared texture while another one is still loading it (see Figure 4).

Under this unique approach, the construction of an avatar is a two-step process: loading an avatar prototype into the factory and instantiating the prototype to create the individual avatars for us in the application.
Prototype and Instances

It is common in a real time computer graphics application to have multiple copies of a single avatar in the scene, each performing different actions at the same time. While they may be functioning independently, these copies still share some of the same data. This shared data can be represented in a "prototype" of an avatar, while the "instances" of the avatar contain the data necessary to function independently.

A prototype can be thought of as a non-functional model of an avatar. The prototype contains all of the data that can be shared between instances of an avatar type. This includes the default skeletal structure, a description of how to map mesh data onto the skeleton, and lists of materials and animations that the avatar can use. It serves as a basis for the creation of an instance.
The avatar instance contains information that can vary between independently operating avatars. This includes information about running animations, world space position and orientation, dynamic motion modifiers, and storage space for posed mesh data.

For instance, prototype A can be the basis of 2 different instances with different skins, indicated by different colors in the diagram.

The following are some examples of when avatar instances consult the prototype for information:

- When an animation is running, the keyframes are continuously pulled from the prototype’s animation data. Only the current animation state is stored in the instance.
- Any materials used during rendering are found through the prototype. No material data is stored in an instance.
- When no animations are running on an instance, the avatar’s skeleton is returned to its default pose, which is contained in the prototype.

**Multiple Avatar Instances**

Once the assembly line has been set up, the developer can tell the factory to make new instances of the prototype. These instances are fully functional characters and can be viewed by the developer as independent of each other, though they may use some of the same resources. Even if two avatars are built from the same prototype, they can be animated and rendered separately. The factory will handle their shared resources in the background without requiring the developer to manage them.

Having multiple instances of the same character are useful in many cases. Perhaps one of Warren’s SWAT team training scenarios involves raiding a drug lord’s heavily defended estate and the officers will use tear gas to subdue the criminals before moving in. In this case, the officers’ avatars will be wearing gas masks. Since they’re already wearing the same uniforms, Warren can just design a single SWAT avatar with a gas mask, load the prototype into the factory once, and create one instance for each officer on the team. He still has the ability to manipulate each officer independently; one can be breaking down the front
door while the others rappel down from the building’s roof. All of the gas mask-wearing SWAT avatars will still share the same resources, which makes Warren’s training simulator more efficient by reducing the file loading time and memory consumption.

**Controlling avatar motion**

The vjAvatar library is designed to let developers easily animate the motions of avatars, without limiting the level of control the developer has over them. It is easy to create simple characters and make them walk around in a VR environment, requiring a minimum of coding effort. Yet access is provided to the lower-level parts of the characters so that developers can programmatically determine their movements rather than be limited to scripted animations.

This keeps vjAvatar’s learning curve very shallow, allowing new developers to get an avatar up and running very quickly without having to learn the complexities of character animation. But advanced developers who have special needs are able to manipulate the avatar at runtime in any way they want. There is a fair balance between ease of use and flexibility.

**User-Definable Functionality with Plug-ins**

VjAvatar seeks to be even more pliable. Going beyond giving developers full control of an avatar’s body, key parts of the library are “pluggable”, meaning that they can be swapped in and out with other pieces that affect how the avatar functions or appears. Some of these important pieces that developers would want to change include character loaders, motion modifiers, texture loaders, and renderers. By allowing runtime plugging of different modules, we make vjAvatar very flexible - developers can change the way things work without modifying the source code of the library itself.

Plug-in functionality can be especially nice when new versions of the libraries are released. Say, for instance, Ronald the programmer has created a VR haunted house application and he wants ghostly avatars that have a greenish glow around them. The
growing effect cannot be represented in any model format, so he needs special rendering
code that draws this. If vjAvatar did not support extendible rendering functions, he might
change the library’s source code in order to draw the glow.

This is initially acceptable, but what happens when the next version of vjAvatar is
released that includes more features that Ronald is interested in? In order to take advantage
of the new version, Ronald has to pick out the changes he’s made to his current version and
port them over to the new one. This can be a lot of work. If Ronald had written a rendering
plug-in instead of modifying the library, the version porting would require very little work.

The support of developer add-ons also promotes reusability. By alleviating the need
for developers to modify the library source code to fit their individual needs, we can have a
wider user base of unmodified vjAvatar libraries. These add-ons can then be distributed and
used by other people.
CHAPTER 6 – TECHNICAL DESIGN

To address the design approach described above, the vjAvatar library needed a base framework to handle character animation as well as a framework to integrate the avatars in a virtual environment. We surveyed existing open source tools in those areas to identify existing systems that we could build from and therefore focus our efforts on the issues of managing avatars in virtual environments.

Character animation system

First and foremost, vjAvatar needed a 3D skeletal animation system. In order to avoid reinventing the wheel, it builds on top of the open source library Ca13D [3]. Ca13D provides us a stable skeletal animation system that is versatile enough to be used on a wide variety of VR systems. Since it is released as open source software, we were able to modify it to suit the needs of VR developers.

One of Ca13D’s most important features is its support for creating characters in Discreet’s 3D Studio MAX [28], a widely used 3D modeling package. Its runtime library was also written in a way that made it highly adaptable to different operating systems, compilers, and system configurations.

Ca13D was not going to work for us right out of the box. It had been designed with games in mind, not virtual reality. Several key changes had to be made to the library for it to fulfill our needs (see Implementation section for details). Originally, Ca13D could not support some of the most common VR platforms, any kind of multi-screen configuration, or any character motion programming. In the spirit of the open source community, these changes were submitted back to the Ca13D project and eventually became part of the library.

Virtual reality framework

VjAvatar also makes use of VR Juggler [2], a development framework for virtual reality applications. While most video games are only run on a single computer/single screen
system, VR applications can be run on a wide variety of systems and distributed environments. They run the gamut from simple desktop setups to the most complex multi-pipe and clustered systems. VR Juggler abstracts many of the features of these complex systems and provides tools and functions to properly handle any configuration with the same code. Developing vjAvatar as an extension of VR Juggler is advantageous because of its reputation in the VR community. Juggler has existed as open source software since the beginning of 2000 and has garnered a growing number of users and contributors worldwide. It has strong support from academia, industry, and military and has a growing user base in each of these areas.

**Code once, run everywhere**

Major considerations in the design of vjAvatar were made to follow the core ideologies of VR Juggler. One of Juggler’s main features is that applications can be written once and run on a variety of systems with no code changes whatsoever. The vjAvatar library, as an extension of VR Juggler, also needed to facilitate this philosophy.

Using the tools that VR Juggler provides, the vjAvatar library has been written in such a way that all of the system-specific details are handled internally, without the developer having to deal with them. An application’s code will be exactly the same whether it will run on a high-end CAVE system [24] or an aging laptop PC.

VR Juggler applications can also be written in a cross-platform manner. This means that an application can be compiled and executed on computers with different architectures and operating systems with no change in source code or functionality. Juggler facilitates this through a runtime abstraction library called VPR (VaPoR, or VR Juggler Portable Runtime). VPR provides a common way to access various kinds of system-level functionality that differs from system to system. This includes things like threading, timers, synchronization, socket networking, and serial communication.

VjAvatar is also designed to be cross-platform. Since the library can make use of VPR’s functionality to deal with system-to-system differences, it “inherits” VPR’s cross-platform capabilities. That is, vjAvatar should work on any platform that VPR supports.
Ca3D is also cross-platform software written in C++ and can be used on practically any system with a compiler compatible with the Standard Template Library (STL) [30].

**OpenGL Graphics Library**

OpenGL was really the only choice when it came to deciding on a graphics API. It is by far the most popular and has widespread support on most modern platforms. Microsoft’s Direct3D is a popular competitor, but is only supported on Windows operating systems. OpenGL has been widely used for many years, and implementations tend to be stable, mature, and with high performance.

Since the Ca3D library is independent of any graphics API, relying on the developer to take the character data and draw it to the screen, efficient rendering methods had to be implemented. VR Juggler supports OpenGL graphics, as well as several scene graph libraries: SGI Performer [32], OpenSG [33], and OpenSceneGraph [34]). Because each of these is built on top of OpenGL, the vjAvatar library can also render in the same application window without causing conflicts.
Figure 5. Integration of base libraries into a VR application with vjAvatar
CHAPTER 7 - IMPLEMENTATION AND FEATURES

Extensions to Cal3D

In order to use Cal3D for our base skeletal animation library, several changes had to be made. Because of the different platforms that VR developers use, Cal3D needed to support more hardware architectures and operating systems. It also needed to support more complex hardware configurations, such as those will multiple screens and stereo viewing. Dynamic motion programming was also a priority due to the interactive nature of VR applications, but Cal3D had not originally been written to support anything other than scripted animations. And finally, to better facilitate motion programming, component naming was added.

_Cross-platform Compatibility in Cal3D_

First, Cal3D had originally been written to be platform-independent, but not architecture-independent. One of the ways that computer architectures can differ from each other is their “endianness”. This refers to the ordering of bytes in memory when a multi-byte number is stored. Most modern architectures are either big- or little-endian. In big-endian systems, such as SPARC or Apple’s PowerPC, the most significant byte of the number is stored first, followed by the others in descending order of significance. On little-endian systems, such as x86 PCs or VAX, the byte ordering in memory is just the opposite [39].

For example, if there is a 32-bit number 0xAAABCCDD, it would be stored as AA BB CC DD on a big-endian system but as DD CC BB AA on a little-endian system. This became a problem because of Cal3D’s binary file formats. When the files were read in on big-endian systems, the byte ordering for all of the data was reversed, resulting in unusable Cal3D objects. Most of Cal3D’s users develop in Windows or Linux, both PC-based operating systems, so the problem had not cropped up before. But a large number of virtual reality developers work on SGI Irix systems, which typically run on big-endian MIPS systems.
Endian-detection code was introduced into the Ca13D library that automatically tests the endianness of a system and adjusts the byte ordering if necessary. Several months after the endian handling code was introduced to the library, equivalent XML file formats and loaders were introduced. This also provided another way around the problem because of the text-based nature of XML. Text is composed of strings of characters, and each character is represented by a single byte, making them immune to the endian issue. As an added benefit, the XML files are human readable, which can be useful during debugging.

**Multi-threaded Rendering with Cal3D**

Ca13D was originally created as part of a game development project called Worldforge [40]. As such, it was designed with video games in mind. One of the biggest differences between virtual reality applications and standard video games, as mentioned before, is the multi-screen nature. Ca13D doesn’t handle graphics rendering, but it does have some generic functionality to assist the developer in creating rendering code.

In particular, the Ca13D “renderer” class can iterate through all of a character’s body parts and allow external access to the vertex data necessary to draw the character on a screen. Unfortunately, the renderer was written in such a way that only one thread can safely use it at a time. But in certain VR systems, there can be many threads drawing a character at the same time. Without changing Ca13D’s renderer code, a multi-screen system would have to serialize the drawing process — rendering one screen at a time — and causing a significant performance drop.

Ca13D’s renderer was modified to allow concurrent access by any number of threads. Each rendering thread maintains information about which of the character’s body parts it is currently working on, rather than having a single location for this information. This allows each rendering thread to operate simultaneously without interfering with one another.

**Dynamic Motion with Cal3D**
Cal3D also had to be modified to facilitate dynamic motion. Originally, Cal3D had only the functionality to play back predefined animations created in a 3D modeling program. Its animation mixer takes care of manipulating the characters’ bones according to which animations are currently running. But there was no easy way for developers to modify a character’s skeleton manually at runtime.

It is not a matter of simply setting a bone’s transformation. For performance reasons, Cal3D stores bone information other than just its local transformation, including its global transformation. This data is dependent on the bone’s local transform as well as the transforms of all ancestor bones in the skeletal hierarchy. Therefore, when a bone’s local transformation changes, its children’s data also needs to be updated.

Functions were introduced into Cal3D to allow developers to manually set bone transformations and automatically update any data that is affected by the changes. This allows dynamic motion plug-ins to be written more easily and takes some of the bookkeeping responsibilities off of the developer.

As mentioned previously, all of these changes were submitted back to the Cal3D project and eventually integrated into the base library. This is good for the users of both Cal3D and vjAvatar. Cal3D users get some additional functionality and portability; vjAvatar users can avoid using a specialized version of the Cal3D library and can use future versions of it as they are released.

**Naming Avatar Components with Cal3D**

One non-critical feature that was added to Cal3D is the ability to access bones and animations by their names rather than their unique identification numbers. This may seem like a minor change, but it goes a long way when a developer is programming dynamic motion. A number of skeletons may be very different in overall structure, but share some similar joints or equivalent animations (such as walking). These similarly named features can be used to write dynamic motion functions that can be used on many avatars, even if they have greatly varying skeletal structures (see Figure 6).
As an example, John is developing a military training simulator in anticipation of a war against killer robots. He has models of a human resistance fighter and an evil robot with tank treads instead of legs. Their underlying skeletal structures are very different, but they both have a head and neck.

![Figure 6. Different skeletal models with some of the same joints](image)

John wants to write a dynamic motion plug-in that can make a character's neck twist to make the head follow moving objects – whatever object the character is paying attention to at any given time. This is simple enough; John just has to rotate the neck bone in the correct direction. But since the characters have very different bone structures, it is unlikely that their neck bones will have the same ID numbers. Without named bones, John’s dynamic motion plug-in would have to specifically know the ID number of the neck bone for each and every character that it would be used on. This bloats the code and limits the reusability of the plug-in, since it would have to be modified for each new character it is applied to.

Since John can reference the bones by name rather than ID, he could write the plug-in to look up the “NECK” bone and twist it to face in the desired direction. This simplifies his code by alleviating the need to check which character is being operated on and selecting a hardcoded ID number. It also makes the plug-in highly reusable; any character with a “NECK” bone defined can have the plug-in attached to it.
The bones start with the names given to them in the 3D modeling package, and they are exported as part of the Cal3D skeleton file format. When this file is loaded into memory, these names are automatically associated with the generated identification numbers. You can also add new name associations and modify them at runtime through the Cal3D API. Animations have no automatically generated associations and must be set manually using function calls.

Cal3D has had support for assigning names to bones in the past (though not animations). But the implementation was very poor on performance. When a developer requested a bone by its name, an iterative search was performed over the list containing all of the bones, comparing strings against each other. This can be a slow operation, especially when the skeleton has a large number of bones and the lookup is occurring for every screen refresh.

A map from the Standard Template Library [30] was added that associates bone names to their integer identification numbers. This provides a very fast lookup, efficient enough for dynamic motion plug-ins to use even when they are looking up named bones every frame of animation.

When animations are loaded into memory, they are also given ID numbers based on the order in which they are loaded. Named animations can be just as useful as named bones when programming dynamic motion. Going back to John's war with the evil robots, he wants to have another dynamic motion plug-in that makes one character chase after another. Depending on the situation, there might be a killer robot chasing after one of the resistance fighters or vice versa.

One way to accomplish this is to trigger an animation that turns the chasing character to face the other one, trigger a running animation, and constantly move the character towards the target at some speed. Because different characters will have different IDs for their running animations, the plug-in needs a way to decide which animation to trigger.

To help with this, the same mapping technique was applied to named animations as was to bones. If John looks up a named animation called “RUN” instead of using a hardcoded ID number, he can apply the chasing plug-in on any character that has a “RUN” animation.
Resource management

Perhaps the most important feature that vjAvatar provides is its resource management. One of the core philosophies of VR Juggler is to allow developers to write applications just once and have the same code work on any platform or hardware configuration that Juggler supports. We sought to maintain this ability for applications that use vjAvatar as well.

The vjAvatar library's resource management component was written to deal with platform- and hardware-specific details automatically without requiring application developers to worry about them. This frees developers from a great deal of work, helps ensure that their application code will be cross-platform, and lets them focus on creating and using their avatars.

Components of a Skeletal Model

Avatars are broken up into individual components, which are stored separately from each other. This is advantageous because multiple avatars can share some of the same underlying components, providing some degree of reusability. Part of the resource management system involves loading avatars from disk and storing them in memory in a way that avoid duplicate file loading and minimizes the memory footprint.

Avatars are made up of the following components (see Figure 7):

1. **Skeleton** – This is the definition of the overall structure and shape of an avatar, consisting of a hierarchy of bones. These bones are represented with a length and orientation in 3D space. There is a single skeleton per avatar.

2. **Meshes** – A mesh is a group of connected vertices forming solid surfaces (polygons) in 3D space that make up the shape of an avatar. There can be a collection of meshes
that, when put together, form the outer shape of the avatar, or a single mesh that forms its entire body.

3. **Materials** – Materials specify how different parts of a mesh are supposed to look to a user. Surface color, light reflecting properties, and/or images are contained in a material.

4. **Animations** – These are descriptions of motion that can animate some or all of a skeleton’s bones.

![Avatar resources: skeleton, mesh, animation, and material.](image)

The avatar factory stores all of these resources separately, makes sure that there are no duplicates loaded, and manages access to pieces that are shared between multiple avatars. This ensures that memory usage is as efficient as possible at runtime and minimizes loading time.
Concurrent Access to Avatars

VR Juggler provides several key functions to vjAvatar to allow the library to function on any kind of VR system. In some VR configurations, such as a CAVE, there may be multiple graphics windows rendering an avatar at the same time. For instance, when multiple screens are used and are adjacent to each other, half of an avatar might appear on each screen. VjAvatar must guarantee that avatars across the multiple windows are synchronized and acting as a single object when it is being rendered in two separate threads/windows. To allow concurrent rendering without causing conflicts, the avatar factory had to be responsible for controlling concurrent access to resources. This is done through the use of blocking mutexes. In order to get at a resource, such as a texture image, a mutex lock must first be "acquired". If some other entity has already acquired the lock, then any others must wait until it is released.

Unfortunately, it is not easy to implement a mutex entirely in software. These are usually implemented directly in a computer's architecture in order to guarantee concurrent access. But then vjAvatar would need to know how to use a mutex on each kind of system it is meant to run on. Instead, vjAvatar uses mutexes from VPR (VR Juggler Portable Runtime), which abstracts the concept of mutexes on a wide variety of systems. In this way, vjAvatar can create and use these locks with a single interface masking the underlying system architectures that the code will actually resolve to.

On a shared memory system with multiple graphics pipes, such as a CAVE-like system, we encounter problems where certain objects need to be used not just once, but once per graphics pipe. Take texture images for instance, which have to be loaded onto every video card. The card assigns them an identification number, which needs to be referenced later in order to use the texture. This is what is known as a context-specific variable. That is, the variable can hold a different value for each graphics context.

Fortunately, VR Juggler has a utility class for dealing with context-specific variables of any type. With the GLContextData class, we can use a single variable and have it automatically handle the setting and retrieval of the values depending on which context it is
called from. This makes the library more system-independent and greatly simplifies the code that handles context-sensitive data.

**Texture Management**

Often the avatars that are the most realistic and life-like are the ones that have the most detail. To achieve visual acuity without using an excessive number of polygons, which could result in poor performance, artists use a technique called “texturing”. Images are painted onto the skin of the avatar to give it more depth, coloring, and detail. For instance, a picture of a human’s face might be used on an avatar’s head to make it appear more like the person it’s representing.

Because textures play such an important role in making convincing avatars, it is important that the vjAvatar library be capable of managing them. To handle the elements of texture images, a texture manager is used.

Textures are notoriously difficult to manage properly in a virtual environment, especially on systems with multiple graphics pipes. Textures differ from other resources that an avatar might use because they have to be loaded onto each video card in the system, not just the system’s memory, before they can be used. During this process, vjAvatar needs to ensure that the texture is loaded onto each video card without loading duplicates into main memory.

The image file containing the texture must be loaded into memory first. This occurs the first time an avatar’s rendering function requests access to it, commonly known as “lazy instantiation”. It is possible that multiple avatars use the same texture. In such a case, the texture manager recognizes this and only loads the image once. Subsequently, the avatars share the texture, which cuts down on load time and memory usage. Access is controlled through an exclusive lock. Whenever an avatar wants to use a texture, it has to acquire the lock and no other objects can use the texture until the avatar releases it.

Once the texture resides in memory, it must be loaded onto the video card. OpenGL references every texture by an ID number that it generates [46]. To use the texture when rendering, a developer must reference this number. On systems with multiple graphics pipes,
however, there is no guarantee that each pipe will generate the same ID number, making it impossible to simply store a single ID number for each texture.

The texture manager makes use of VR Juggler’s context-sensitive variables, which automatically store separate data for each OpenGL graphics context. Each context corresponds to an individual video card in the system. These variables can only be referenced from a thread that can perform graphics rendering. When the variable is referenced, VR Juggler checks which thread is referencing it, and resolves the data depending on which graphics context the thread is tied to.

Through the use of texture loading plug-ins, the vjAvatar library provides great flexibility to character artists. There are a wide variety of image formats that can be used to store textures, such as JPEG, bitmap, and TGA. Each format has different advantages and capabilities, such as compression ratio, color depth, and transparency.

Loading images is not always a trivial task, especially for formats like JPEG that have complex compression schemes. Programming image loaders presents a great deal of work. Fortunately, there are already several open source software libraries available that can load most common formats. A framework is provided that allows a developer to easily add support for any image library that they might want to use. In addition, support for DevIL [36] and Corona [37] has been built into vjAvatar in the form of texture loading plug-ins. These plug-ins are optional to use.

The vjAvatar library comes with native support for bitmapped textures purely for convenience, to eliminate any more external library dependencies. This also promotes the cross-platform nature of vjAvatar by providing a basic image loader that will work on platforms that other image libraries might not support.

**Time Synchronization**

Clustered systems offer the unique problem of time synchronization. Although their graphics buffer swapping is automatically synchronized by VR Juggler, the individual node updates and drawing routines can take very different amounts of time, depending on the performance of particular nodes. The animation of avatars is highly dependent on time,
especially because standard animation relies on the interpolation of still-shot keyframes. Even differences of a few microseconds can make a fast-moving character appear out of sync.

Gadgeteer, the device configuration library that VR Juggler uses, has already tackled the problem of synchronization across a clustered system. Each input device, such as a motion tracker or digital button, has to distribute its input information to every node in the cluster configuration. This includes a time signature that tells when the latest input information originated.

Gadgeteer provides a method to poll its devices for the cluster-synchronized time. The vjAvatar library relies on this time signature to keep track of the passage of time, and uses this time measurement when updating characters. Every VR Juggler application will have at least one device (that represents the user’s position), so this approach will work in all cases.

**Math in vjAvatar**

One of the most important modules that vjAvatar makes use of is the Generic Math Template Library (GMTL), a full featured collection of 3D graphics math classes that VR Juggler uses. The vjAvatar library uses GMTL to a great extent. All of its transformations are stored with GMTL’s data types and all of the operations involving 3D math are performed using GMTL. Cal3D already has its own 3D math classes, but they consist of only the basics and do not offer a lot of functionality. Using GMTL gives developers much more flexibility in how they act on the avatars and lets them use the same classes that they would already be using in a standard VR Juggler application.

Cal3D is an oddity in that the coordinate system its math classes use is a "left-handed" system, whereas most computer graphics texts deal with "right-handed" rotations (see Figure 10). Conversion functions had to be implemented to convert between Cal3D’s rotation types and GMTL’s equivalent rotation types so that computations could be made using both of them.
To illustrate the difference between left- and right-handed rotations, take a person’s right hand. Point the thumb straight up while curling the fingers as if around a coffee mug. The thumb represents an axis of rotation and the fingers show which direction a positive-valued rotation will rotate. So in a right-handed coordinate system, rotations greater than 0 degrees will rotate counter-clockwise and anything less than 0 degrees will rotate clockwise.

If the same posing is applied to a person’s left hand, it is clear that the rotations are now applied in the opposite direction. That is, positive-valued rotations will move clockwise and negative-valued rotations will move counter-clockwise about the axis. This is the opposite of the right-handed system (see figure 8).

An attempt was made to convert the Cal3D library itself to use GMTL for its math library. A performance boost was anticipated, and it would pave the way for future development of basic collision detection functionality. But it turned out to actually cause a significant performance drop. Further analysis and discussion elucidated the fact that GMTL was designed to take advantage of some particular compiler optimizations in order to provide the high performance that it boasts. Unfortunately, not every C++ compiler supports these kinds of optimizations. The majority of Cal3D users were not using them, so the plan for switching to GMTL was shelved.
Character Skinning

Another feature that gives developers more flexibility is the ability to skin characters at runtime. This is akin to a person changing their clothes or putting on makeup. It allows users to alter the way characters look by changing what is rendered on their surfaces.

Ca13D has built-in support for runtime skinning through the use of “material sets”. These are collections of materials that account for the full surface area of a character’s model. By switching between material sets, a developer can drastically change the way a character appears when rendered. Alternatively, individual material “threads” can be changed within a material set when only part of the character needs a different material. For instance, switching shirts or facial expressions are common examples of this.

Pluggable Functionality

The vjAvatar library offers a number of ways for developers to extend and customize its functionality. Without modifying the base library, developers are able to change the way vjAvatar behaves in the many areas, including rendering, character motion, and resource allocation.

Much of this customization can be done through the use of developer-written plug-ins. The plug-ins work through the use of C++ class inheritance coupled with the use of virtual functions. This means that the library recognizes a “type of” plug-in and can use it through a common interface, but it doesn’t need to know exactly what the plug-in is doing. This is resolved when the VR developer compiles their application.

Dynamic Motion

Dynamic motion plug-ins can be used to programmatically change an avatar’s body pose, skeletal motion, or placement within the environment. This is different from simply executing predefined animations; the motion is generated at runtime, possibly as a reaction to other objects in the environment.
Dynamic motion is useful for creating realistic interaction between avatars and the surrounding environment. Collision detection and response is commonly used in VR applications to make an environment seem more real. Instead of allowing an avatar’s hand to pass through a wall while walking near it, a dynamic motion plug-in could detect this collision and move the hand’s position to the outer edge of the wall, as would happen in real life.

**Rendering Methods**

Developers can change the way their avatars are drawn graphically to the screen. This is useful when special effects are necessary to complete the look of an avatar. For example, let’s say Chang is creating a VR medieval fantasy environment that includes a huge dragon avatar that breathes fire and smoke at will. Smoke and fire cannot be represented in most common 3D model file formats, including Ca13D’s files.

Chang could write a custom rendering plug-in that would draw billowing smoke and an animated stream of fire along with the regular mesh of his dragon, without any limitations on which OpenGL features he can use.

**Avatar Loading**

Another customizable area of the vjAvatar library is the avatar loading functionality. This is the code that reads the configuration files that contains the locations of all the separate Ca13D files that make up a complete character as well as other information. A developer might want to extend the configuration file format to include other options and information.

As an example, consider a football player avatar that can don the uniforms of any NFL team simply by changing which material set he is using. A developer might create an extension to the file format that specifies which team the avatar belongs, along with an avatar loader plug-in that recognizes the team designation and automatically selects the correct material set.
Configuration file format extensions might also be used to automatically load other plug-ins not included in the core vjAvatar library, such as developer-defined rendering or dynamic motion plug-ins.

**Texture Loading**

The vjAvatar library supports the use of custom texture loading plug-ins. The ability to texture 3D models with images is a necessity in real time computer graphics. There are a wide variety of image formats that vjAvatar could support; each has its own advantages and disadvantages, but there is no clearly superior format.

Rather than attempting to write image-loading code for a number of popular formats, we were able to leverage existing software libraries that specialize in image loading and manipulation. The vjAvatar library comes with optional texture loading plug-ins that use Corona [37] and DevIL [36], two open source image libraries that load many commonly used image formats. Developers are free to choose one of the provided texture loaders at runtime or they can write their own if they want to use other image formats not directly supported through vjAvatar.

Unlike other plug-ins, defined using C++ class inheritance, texture loading plug-ins are defined with function pointers. This is because there may be multiple instances of other kinds of plug-ins, but there is usually only a need for a single texture loader.

**Avatar Specialization**

For developers who need even more advanced functionality that cannot be accomplished through the various kinds of plug-ins, specialized avatars may be necessary. A developer does this by defining a new "type of" avatar that can have any arbitrary functionality as long as it fits in with the avatar factory's work pipeline.

This again uses the concept of C++ class inheritance. The developer can create a new class that inherits from the vjAvatar class. An example of this is included in the vjAvatar library, called the bipedAvatar. It makes the assumption that an avatar has a human-like
skeletal structure with a head, 2 arms, and 2 legs. Functionality is built into the bipedAvatar class that can automatically move the avatar's body to mimic the motions of a live human user.

**Dynamic Motion Plug-in Implementation**

Dynamic motion plug-ins have to fit in with the pipeline of updating an avatar and getting it ready for rendering. In the vjAvatar library, the process operates in this order:

1. Update animations (if any)
2. Update skeletal pose
3. Apply dynamic motion
4. Update physique
5. Write vertex arrays for rendering

During step 1, skeletal animation frames are generated by interpolating between the closest pair of keyframes, based on the current time. If an animation is finished, it will be stopped and removed from the animation queue. To update the skeletal pose in step 2, the animation frames will be taken and blended together to form a full skeletal pose.

After this has been done, dynamic motion plug-ins can modify the skeleton if they need to. It is necessary to apply dynamic motion after animations have already been blended because Cal3D’s animation code will override any changes that were made, setting the skeleton’s pose based solely on the blended animations.

Updating the physique of an avatar involves taking the mesh that makes up the model and “wrapping” it around the skeleton. This process is also known as “skinning” [5]. Each vertex in the mesh is associated with one or more of the bones of the skeleton. During the skinning process, each vertex is modified based on where its influencing bone(s) are located. This causes the mesh to stretch and bend just like a person’s skin does when they move. Surface normal vectors also need to be calculated for each vertex, which are necessary for proper lighting in OpenGL.
Once all the vertex data has been calculated, it can be written in a form that is useful for rendering. For our purposes, vjAvatar’s rendering code simply writes all the data into one large array. The vertex location, surface normals, and texture coordinates (if any) are interlaced to take advantage of OpenGL features that can speed up the process of submitting the mesh data to the video card.

Scene Graph Integration

It was desirable for vjAvatar to be supported in Juggler applications that use any of its supported scene graphs (Performer, OpenSG, OSG), but this presented several technical problems. Scene graphs are complex pieces of software that attempt to optimize the graphics rendering process. They often make use of rendering techniques that can conflict with other rendering code, such as vjAvatar’s OpenGL rendering. We have had extensive opportunities to work with OpenSG, so this discussion is based on our experiences with this particular scene graph, although the issues are common across others as well.

OpenSG handles textures in a way that conflicted with vjAvatar’s texture handling. In OpenGL, every texture that is loaded into video RAM has an identification number associated with it. A developer typically makes a function call to generate one of these ID numbers for each texture. Following this process ensures that each texture gets a unique ID number.

OpenSG, however, manages its own texture ID numbers without generating them through OpenGL. This is a valid approach because there is usually only one OpenGL process running on a graphics card at any given time. But this causes vjAvatar’s generated IDs to frequently conflict with IDs generated internally by OpenSG. This results in vjAvatar’s textures overwriting OpenSG’s textures and vice versa. The user will often see a radically altered environment, with avatar faces plastered on a wall or an avatar’s clothes painted like the floor.

To work around this problem, vjAvatar supports a special texturing mode that switches it from having OpenGL generate texture IDs to generating its own, similar to the
approach OpenSG uses. The vjAvatar library uses a different number space to select IDs from, ensuring that they will not conflict with OpenSG.

A context mode switch was added to vjAvatar to support applications that cannot use context-sensitive data. This was required to make vjAvatar compatible with SGI’s Performer scene graph. As discussed earlier, vjAvatar makes extensive use of context-specific data. But VR Juggler does not support context-sensitive data in Performer-based applications (as of version 2.0 Alpha 3).

When context-sensitive data is disabled, a single variable is used instead of one that maintains different data for each graphics context. This allows the application to be compiled and executed, but data integrity cannot be guaranteed at runtime when multiple OpenGL applications are running on the same graphics pipe at the same time.
CHAPTER 8 – RESULTS AND EXAMPLES

VjAvatar has already been used in several projects at Iowa State University’s Virtual Reality Applications Center [27]. The timing of the development of the vjAvatar library presented a unique and fortunate opportunity. At the same time that development got underway, a number of other projects were in the works that stood to benefit greatly from a 3D character library. As such, the vjAvatar library was designed and implemented from the ground up based on a great deal of developer feedback and field-testing.

The projects were varied in scope, purpose, and application. There were different target operating systems, character types, VR devices, and graphics APIs to name a few. These differences resulted in amassing a body of testing that rooted out problems early on and ensured that vjAvatar would be robust on many platforms and VR systems.

The Virtual Hindu Temple

The first major project to make use of the vjAvatar library was the Virtual Hindu Temple project [20], a collaboration between Iowa State’s Department of Philosophy and Religious Studies and its Virtual Reality Application Center.

The project’s goal is to create an educational tool for students wanting to learn about the religious practices of the Hindu people. The belief is that being immersed in such a virtual environment can supplement a person’s understanding of the culture in a way that reading textbooks cannot. A virtual experience also alleviates the need for journeying to far-away lands or time traveling to past historical eras.
The centerpiece of the environment in the highly detailed recreation of the central chamber of the Radharaman Temple, a 16th century Hindu temple that still stands today in Vrindavan, India. Many distinctive features of architecture from the period are represented – from the checkered floor tiling to the colorful hanging tapestries and column supports made of white stone (see Figure 9).

A crowd of Krishna worshippers, complete with traditional Indian clothing, populates the Hindu temple. Men are clad in flowing robes and sandals; women are wearing a variety of vivid saris (lightweight garments consisting of a skirt and head covering common among Indian and Pakistani women). The environment recreates the religious practices and people in great detail.

This is a good example of where Cal3D’s exporter plug-ins were put to use. Although many talented programmers were available from the Virtual Reality Applications Center, the project coordinators needed to recruit someone who could solidify their historic and artistic vision into 3D avatars. Since 3D Studio MAX is such a commonly used tool, many 3D art
students are trained in its use. A student artist with no former experience in VR was able to create each of the Indian worshippers from scratch in 3D Studio MAX.

The temple’s characters come to life through the use of the vjAvatar library. The avatars are entirely scripted, following predefined paths and animations to act out a contemporary worship ritual.

The Multi-mission Aircraft (MC2A)

Another project using avatars to populate the environment is the Multi-mission Aircraft (MC2A) project. The United States Air Force has a new airplane coming into service in the near future called the Multi-sensor Command and Control Aircraft [41]. It is designed to be a highly configurable aircraft capable of performing a wide array of mission goals and is based off the Boeing 767 airframe. Ultimately, it is intended to replace existing reconnaissance aircraft.

To be able to support many different mission roles, the MC2A is built so that its internal components are easily replaceable. It can make use of a variety of electronic equipment packages adapted to particular missions, allowing the aircraft to replace the functionality that is currently filled by several different aircraft (AWACS, JSTARS, ACCC) [42].

The MC2A’s VR application acts as a rapid prototyping tool. Users are immersed in an Air Force base environment featuring a large hangar and a highly detailed model of the MC2A. They can add different kinds of electronic equipment modules inside the aircraft, move them around to get the best fit, and test the setup to ensure that the human operators will have enough space to work or maneuver inside the hull and cockpit.

VjAvatar is used to visualize the human operators and mechanics that will be working with the aircraft (see Figure 10). Outside the MC2A, an Air Force mechanic character is used to show what parts of the plane can be accessed without assistance of a lift or ladder. Inside, users can place avatars in front of consoles and in the empty spaces where electronic equipment does not reside. The people in charge of setting up the aircraft can gauge how well
any given setup allows the operators to do their jobs and if there is enough space for them to move about the cabin.

![Avatars working inside the MC2A aircraft](image)

**Figure 10. Avatars working inside the MC2A aircraft**

This is an example of where certain ergonomic aspects of a system are prototyped and tested, but extremely realistic modeling was not necessary. EDS’s commercial JACK software [43] could have been used, and is even marketed for use on such projects. But in this case only a rough estimate of whether human operators could fit in the spaces created by different aircraft configurations was needed. Using JACK would have been a pricey option while most of its extensive capabilities would have gone unused.

DI-Guy would have been a more appropriate commercial choice, especially considering the military nature of the application. But the animations of the mechanics working on the aircraft and the operators working at their consoles would still have had to be custom made. On top of DI-Guy’s licensing costs, the environment developers would have to pay a Boston Dynamics consultant to make the animations. The vjAvatar library fulfilled all the project’s requirements without the commercial pricing.
**Template Application Collection**

The Template Applications Collection [23] is a project that seeks to provide a basis upon which many different VR applications can be built. Although these applications can be used for any number of purposes – engineering, art, entertainment, etc. – they usually have similar aspects that end up getting implemented for every application, resulting in a lot of redundant work on the part of the developers. These common aspects include user navigation, model loading, object interaction, and collision detection.

By starting with the Template Applications Collection, VR developers can avoid having to re-implement these pieces of functionality and concentrate on the aspects of development that are unique to their individual applications. Let’s say a developer is working on a VR architecture program that allows people to preview their 3D CAD models in life-size scale and walk around to get a sense of what the space feels like before actually constructing their buildings. With the Template Applications Collection, the developer can make use of a model loader to bring the CAD models into the virtual environment. He can use the Collection’s navigation code to allow the user to walk around the model, and set it to automatically use collision detection, which keeps the user anchored to the floor and prevents them from walking through walls. In fact, the architecture application is almost completed aside from tweaking movement speeds and setting up controls. This has saved an enormous amount of development time.

The Collection also provides collaborative functionality for developers who want to build networked applications where multiple users can occupy the same virtual environment and interact with each other without being in the same physical location (see Figure 11). The vjAvatar library is used to integrate avatars representing live users in the shared environment. 3D avatars appear in the scene wherever a real person resides, showing their position and orientation.
One of the first demonstrations of the collaborative capabilities of the Template Applications was a manufacturing assembly simulation. In this application, multiple users in remote locations work together to assemble the major parts of a virtual tractor. They can pick up the pieces of the tractor and try to connect them together just as a factory worker might. The avatars' heads are twisted around to show what the user is looking at, giving participants an idea of what the others are working on.

**AvatarNET**

The AvatarNET project combines elements of virtual reality, psychology, and video games. It is a fully immersive VR first-person shooter game, in which players compete by sneaking around and shooting each other in a large building with lots of places to hide (see Figure 12).

The game is part of a psychology experiment designed to test how violent video games affect the human psyche. Studies in the past have suggested that violent video games can make test subjects more aggressive for a short time [49]. The goal of this experiment is to
test whether a fully immersive game — one that makes users feel like they are actually part of it — can have a more profound effect on a test subject’s aggressiveness.

Figure 12. Networked combat inside AvatarNET

The players move around the environment searching for other players to shoot at. They are able to maneuver throughout the game area, moving up and down stairs, running through hallways, and ducking behind tables. Players aim and shoot with a realistic toy handgun that is motion-tracked by the VR system. Blood curdling screams are heard whenever a flying bullet hits someone.

In much the same way as the Template Applications Collection, vjAvatar is used to visualize the other players in the game. In addition to representing player locations, death animations are used whenever a player is killed.

**Performance Metrics**

We were able to gauge the general performance characteristics of the vjAvatar library by carrying out a series of controlled tests that attempted to isolate vjAvatar from the application overhead as much as possible.

When vjAvatar is integrated into an actual application, it is difficult to tell how it is performing because it has to share resources with the rest of the application. Some
applications constantly allocate and free resources, or widely vary the amount of computation
time they use at runtime. This can affect vjAvatar's performance in turn.

The test applications use only OpenGL for graphics (as opposed to a scene graph). No
objects occupied the virtual environments except for the avatars that the particular tests
called for. No rendering occurred except for that of avatars.

All tests were conducted on the same PC machine with a single graphics pipe. The
system has a 2.4 GHz Pentium 4 processor with 2 GB of memory, as well as an nVidia
GeForce 3 graphics card with 64 MB of VRAM. We ran the test applications on a RedHat
Linux 8 operating system.

We devised a test to measure the efficiency of vjAvatar's memory usage when a
number of avatars occupied the same environment. We compared this with numbers that
resulted without the use of the avatar factory. The test consisted of adding increasing
numbers of a single textured avatar to the environment. Having multiple instances of the
same avatar does not necessarily degrade performance linearly (see figures 13 and 14), since
many resources are shared between the individual instances.

![Graph showing memory usage](image)

Figure 13. Memory usage using an increasing number of textured avatars
The use of the avatar factory drastically reduces the amount of memory needed, as shown in figure 13. It should be noted that when very few avatars are used, it is possible that slightly more memory is used in practice. This is a result of the overhead of the avatar factory. As more instances are added that are built from the same prototype, or from prototypes that share resources, the reduction in memory usage of the avatar factory is clear.

In the preceding test, we used an avatar that made use of a large amount of texture memory. In fact, storage for the avatar's textures comprised most of the memory cost. The next test was very similar, but used a non-textured avatar. This test shows the memory usage curve as a result of other shared resources (skeletons, animations, and meshes), which require significantly less memory than texture images.

![Graph showing memory usage with and without the avatar factory](image)

Figure 14. Memory usage using an increasing number of non-textured avatars

The memory usage curve of figure 14 appears similar to that of figure 13, but note the different scale of the memory usage axis. The second test showed less of a difference
between linear memory usage and vjAvatar's usage, but the difference was significant nonetheless.

Memory consumption is not the only consideration for performance. We also looked at how an increasing number of avatars would affect the overall framerate of an application. In this test, we increasingly added textured avatars into the environment and executed an animation on each of them (see Figure 15).

![Figure 15. Application framerate decreasing as number of avatars increases](image)

Although not shown on the chart, the application was able to redraw the screen at approximately 1130 Hz when no avatar instances were being used. It should also be noted that standard computer monitors and projectors are not usually capable of refresh rates in excess of 120Hz, and typically operate much lower than that.
CHAPTER 9 - CONCLUSIONS

The vjAvatar library served a very important role for several major applications (see chapter 9). Much of its usefulness can be attributed to its fulfilling of the design requirements specified in chapter 2.

Performance

Applications were able to make use of many avatars at the same time, while maintaining real time performance. As expected, performance scaled down as the visual complexity (number of polygons) of the avatars’ meshes increased. The majority of computation time is spent generating the vertices of the avatars’ meshes after each skeleton’s pose had been set.

A noticeable drop in frame rate was experienced when a large number of texture images were being used at the same time. After some investigation, it was discovered that more texture memory was being used than the video card had onboard. When this occurred, textures were continually being swapped in and out of video memory, resulting in a much slower performance.

Versatility

The vjAvatar library exhibited versatility in the variety of graphics rendering libraries that it could be integrated with: standard OpenGL, the OpenSG scene graph, and SGI’s Performer scene graph. VjAvatar worked most easily with OpenGL applications. As discussed in chapter 8, specific changes had to be made in order to properly work with OpenSG.

Some shortcomings were apparent with SGI Performer. Performer has a radically different rendering pipeline that has many threads running asynchronously, and VR Juggler does not support context-sensitive data in Performer applications, which made it impossible to handle textures correctly. It was only possible to integrate avatars into Performer
applications by storing duplicate copies of avatar instances, one for each graphics pipe. This worked, but was wasteful of memory.

Applications were developed that operated on a variety of system configurations and platforms. The array of system configurations included desktop and laptop computers, large stereo projection systems, clustered computers with multiple screens, and CAVE-like systems with motion tracking systems.

**Ease of Use**

Application programming was made easier, since all of the resource management was done in the background. Developers were able to integrate characters with only a few lines of code. They were able to focus their time and energy on creating better avatars and designing their interactivity without having to expend effort on dealing with the details of writing system-independent code.

The ability to write the same code that would run on all systems aided development by allowing testing on desktop computers, even though the applications were targeted to run on more complex systems. Access to the large projection systems is very limited due to the many people wanting to use them. Developers were able to perform the majority of development on regular desktops and test the same code on the large projection systems only rarely.

Several of the applications required custom avatars suited to particular virtual environments. Through the use of Cal3D’s exporters, 3D modelers were able to create new avatars using 3D Studio MAX. Developers who can write software as well as use 3D modeling tools were in short supply, and the creation of animated character could have been a major bottleneck impeding the project workflow. Being able to use a popular modeling program allowed development teams to recruit computer artists who had no previous experience with virtual reality applications. These artists were able to easily bring conceptual characters from the sketchbook to the 3D environment.
Dynamic Motion

Dynamic motion plug-ins were used in several ways to facilitate avatars interacting in real time. This was exhibited by movement functions for computer-controlled characters as well as avatars representing live users, with motions that mimic those of live users in networked virtual environments.

Extensibility

The open source nature of the vjAvatar library made it possible for developers to easily expand and build new tools on top of the library. VjAvatar has been released under the Lesser GNU Public License (LGPL) [29] as part of the VR Juggler Toolbox [31], which makes it available to the larger VR community, including commercial and industrial interests. During the course of application development, it became apparent that several extensions would be useful alongside vjAvatar.

One extension of vjAvatar is a set of Python bindings for the library, written by Patrick Hartling. Python is an interpreted programming language that is gaining widespread popularity among developers because of its powerful object-oriented nature and ease of use. Since VR Juggler already had Python bindings written for it [38], this allows users to develop entire VR applications including avatars using only the Python programming language. The Python bindings for vjAvatar, while not a required piece of the library, are now packaged along with the source code for those who want to make use of them.

Another useful extension that is under development is Alan Fischer’s scripting library. Although the vjAvatar library makes it easier for developers to program animated characters in their VR applications, the scripting library goes even further. Virtual environment designers can create a shorthand script similar to that of a screenplay, specifying which characters to use, when and where they should move around the environment, and which animations to execute. This makes it possible for non-programmers to get involved in the creation of a scripted VR environment. It also allows designers to change the way characters act out their roles without code changes or even recompiling the code.
CHAPTER 10 – FUTURE WORK

Despite the success of the vjAvatar toolkit, there are still a number of ways it could be made more useful for VR developers.

Facial Animation

In order to better facilitate communication between live users in a networked virtual environment, facial animation could be further developed. As of this writing, Cal3D has a simple implementation of “mesh morphing”. This allows, among other things, rudimentary facial animation through a set of freeze frame mesh positions.

A developer can create a series of head model meshes representing many different facial expressions: smiling, frowning, surprise, fear, and others. These different meshes can be loaded into Cal3D as part of a standard character model and the developer can morph between them, blending them into a wide variety of different emotions.

Facial expressions can also be used to represent “phonemes”, the smallest units in a language that make up words. These are expressions that a person’s face makes when they speak different parts of words. For instance, phonemes like “mmm”, “aah”, and “ooo” are all visually represented by different facial expressions. These are often called “visemes”.

One open source project called Expression [48] has produced a graphics library suitable for real-time animation of faces using a muscle-based approach that tries to approximate the way real human faces work.

Speech Communication

Speech communication could also complement the addition of facial animation. To better facilitate human-to-human communication, a voice-over-IP solution could be integrated into the vjAvatar library. This would allow users in networked virtual environments to communicate directly with each other as if they were in the same room, rather than be limited to nonverbal kinds of communication.
Component Separation

Although the vjAvatar library currently depends on VR Juggler's modules to deal with the intricacies of complex virtual reality systems, there is no need for this on a simple desktop setup. A version of vjAvatar could be stripped down to remove any dependencies on VR Juggler and make it usable for developers who just want to integrate animated characters into their OpenGL-based applications (see Figure 16). Such a version of the library would be useful to developers not using VR Juggler, such as those making games.

By removing the dependence on VR Juggler, there is much less work involved in getting vjAvatar working and less overhead when running applications that use it. The library

Figure 16. Library component separation with glAvatar and vjAvatar
would still need to depend on GMTL for its 3D math, but this is possible because GMTL exists as standalone software with no dependencies on other libraries.

**Animation System Abstraction**

One noticeable aspect of various character animation SDKs is that they all support some common functionality. This includes things such as loading characters, triggering animations, altering bone transformations, and rendering. Although vjAvatar currently only supports Cal3D as its base skeletal animation library, it would be possible to abstract the common functionality of other SDKs like Kaydara, DI-Guy, and Jack to allow developers to choose the underlying library that suits them while providing a common interface.

With such an abstraction, developers could write code for vjAvatar that would work regardless of the underlying SDK. Say for instance, Bryan is developing a coffee-drinking simulator for Folgers and is using DI-Guy as the character animation library. His friend Jared, who works for Pepsi, is working on a similar Mountain Dew-drinking simulator, but is using Cal3D for character animation. Using the vjAvatar abstraction, they could write a single dynamic motion plug-in that makes a character shake and twitch randomly as if from a caffeine overdose. This plug-in would work on both of their characters, despite the fact that they're using different underlying animation libraries.
REFERENCES


[21] “AvatarNET Psychology Experiment”,

[22] “Template Applications Collection”,


[41] “Multi-Sensor Command and Control Aircraft (MC2A)”,


Addison-Wesley, 1999.


ACKNOWLEDGEMENTS

Iowa Air National Guard, 133rd Air Control Squadron
United State Air Force Research Lab
John Deere
Miller Fellowship
Carolina Cruz-Neira
Adrian Sannier
Yan-bin Jia
James Oliver
Bruno Heidelberger
Laurent Desmecht
Allen Bierbaum
Patrick Hartling
Ben Scott
Ben Vance
James Allen
David Kabala
Sarah Kabala
Allan Fischer
Chris Just
Moonho Park
Ronald Sidharta
Josh Brown
Café Beaudelaire