August 2011

Kiip - A Skeletal Motion Capture Library

Elliot Neil Borenstein
Worcester Polytechnic Institute

Kiip

A Skeletal Motion Capture Library

Department of Computer Science

A Major Qualifying Project Report
Submitted to the faculty of
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Bachelor of Science

By

Elliot Borenstein

Advised By

Robert Lindeman

August 24, 2011

Abstract

for the design and implementation of Kiip, a Skeletal Motion Capture Library

By

Elliot Borenstein

This report details the design and implementation of the Computer Science Major Qualifying Project (MQP) to create Kiip. Kiip is a library for the real-time conversion of optical motion capture data into hierarchical skeleton data. A large amount of research has been done previously on the topics of motion capture and retargeting, but there are few tools available to application and content developers. The tools that do exist are generally expensive closed systems. The Kiip library was designed to be free and easily integrated into external applications, where online conversion of the motion capture data is needed.

This report begins by discussing the theory necessary to perform motion capture. Various existing methods for capture and retargeting are presented and compared. Next, an overview of the hardware required for motion capture is given, including examples of the hardware used by and created for the MQP. This is followed by explanations of the mathematics used by the library, as well as for related and more complex facets of retargeting that were not added to the final library.

Following the theory, the report details the design and implementation of the Kiip library. The public API exists in a single *Kiip* class which handles all communication between an application and the various systems used to calculate the final motion. An application-definable *Reader* class handles the polling of data for use by the Kiip system. A *Mover* class then receives the data and processes it to find information about the movement of an actor in the real world. Finally, a *Retargeter* class attempts to apply that information to an application defined character, mapping the motion of an actor in the real world to a character in a digital world. Two applications of the library as discussed, one inside a real-time game engine, and another a plugin to an industry standard 3D animation software package.

Following the implementation, discussions of the schedule and outcomes of the project are presented. Testing of the accuracy of the system is detailed and analyzed, showing that the Kiip library produces reasonably accurate output given sufficiently accurate input. Unfinished areas of the library, particularly with respect to motion retargeting, are discussed, including the research conducted and the methods attempted. Suggestions for solutions to the problems encountered are also given. Finally, a small selection of topics for future work is given. These topics include alternate methods of calculating certain kinds of data, a proposed generic server application of the library for general use, and suggestions for improved input and output methods.

Contents

List Of Figures

1. Introduction

Character animation is the process of bringing movement and life to a digital character, and is used in many applications. Traditionally, these animations were created manually, but more recently animations based on motion capture data have become more prevalent. Even with motion capture data though, the animations are normally processed and applied to characters offline using expensive commercial software. The Kiip library described here aims to provide a method for performing the conversion from motion capture data to an animated skeleton in real-time. In this system, the motions of an actor being tracked by an optical motion capture system were to be applied to an application-defined skeleton for use in any application.

Two such applications were envisioned for the project. One intended to immerse the user fully within a 3D environment, allowing him to look and move around an environment. This type of application could be used for 3D gaming, virtual reality research, virtual meetings and performances, or similar applications. The other was real-time feedback of a motion capture session, allowing content creators to quickly preview animations on their models as an actor moved about.

Due to complications in the development of the project, the Kiip library does not yet properly calculate retargeted skeletons. As such, these applications were not fully completed for this MQP. However, the library does properly find the positions of joints in the world, and this functionality is visible in the applications.

1.1 Motion Capture Basics

Motion capture is a process of recording data about and object's movement, and using that data to control an animated character. In particular, Kiip uses captured data about the limbs of an actor, and uses it to calculate information about that actor's skeleton. There are several varieties of motion capture, such as optical, inertial, and magnetic, each with their own strengths and weaknesses. This project focuses on optical capture. Optical motion capture tracks fixed points, or markers, on an object from frame to frame, and provides their 3D coordinates. Unlike some other varieties, optical capture does not implicitly provide the orientation of the object being tracked. Some systems use passive markers, illuminating reflective spheres with external light sources that can be seen by cameras. More advanced systems use active markers, electronically modulating the rate and timing of LEDs to more accurately track each specific marker.

Tracking markers and interpreting the data is a complex problem. Finding the coordinates of a single marker requires aligning images from multiple cameras, each of which must be carefully calibrated to ensure accurate results. If too few cameras are able to see the marker in a frame, its position cannot be determined. If two markers on different objects pass in front of one another, some method must be used to determine which is which. The exact methods for solving these problems are beyond the scope of this project, and many commercial systems are available, providing both hardware and software aimed at solving these problems. Kiip assumes that the marker data, comprising the positions of all markers in a given frame, has already been processed by some external system.

2. Related Work

A great deal of previous work exists in the field of motion capture, including both research papers and commercial software that generate animations from optical marker data. However, most research topics only present solutions to specific portions of the motion capture pipeline, generally dividing capture of an actor's motion and retargeting of that motion into two distinct areas of study. The majority of papers discussing the calculation of the actor's motion also discuss various methods for capturing the base marker data, as previously described.

2.1 Commercial Tools

The industry standard tool for working with marker data is Autodesk MotionBuilder[1]. MotionBuilder is a character animation tool that is capable of calculating animations based on marker data, and retargeting those animations to a wide variety of skeletons. Many of the principles used in Kiip's design come from MotionBuilder, including the division of the body's limbs. While MotionBuilder is a powerful piece of software, it does have drawbacks. MotionBuilder is capable of retargeting animation based on real time marker data, but as it is a standalone application it cannot be integrated and shipped with a product. Also, MotionBuilder licenses are generally very expensive. While not as fully featured, Kiip was designed to be free and easily distributed with other applications, while also being easy to integrate into existing software.

2.2 Actor Capture

Three primary papers were used in designing Kiip. O'Brien et al.(2000) discuss finding joint locations using magnetic motion capture data. This method uses markers on limbs adjacent to each joint to find the position of the joint in world space. Unlike optical data, magnetic data has information about both the position and orientation of each marker, simplifying the calculations. Since optical data doesn't have this orientation data, it must be calculated. Ringer (2004) discusses the same technique as it is applied to optical data. Finally, Charalambous (2005), a senior thesis from Cambridge University, discusses methods for calculating the orientation data, as well as providing further explanation about this method of calculating the joint positions.

Xiao, Nait-charif, & Zhang (2009) use the same method as above, and describe a method for calculating the orientation of the actor's skeleton once the positions are known. Hornung & Kobbelt (2004) also use the above method, and discuss a way of rebuilding the position of the elbow (and similar joints) when it has been lost by using information about the two surrounding joints.

There are several other methods based on the same principle of using markers on adjacent limbs to find the joint positions. Kirk, O'Brien, & Forsyth (2005) and De Aguiar, Theobalt, & Seidel (2006) use alternate ways of calculating this data, while also using these calculations to automatically build a representation of the actor's skeleton. By comparing marker groups on every pair of limbs (instead of known pairs, as above), they determine which pairings are most likely to be about a common point of rotation, and assign that point to be a joint. Herda et al. (2000) and Herda et al. (2001) present similar

[1] http://www.autodesk.com/motionbuilder

techniques. Kiip assumes that the correspondence between markers, limbs and joints is known, and so the techniques for finding that information were not used.

Another common method for finding the joint positions is to use the skeleton's hierarchy. This method uses the fact that a joint should always remain at a fixed local position relative to its parent joint. It also relies on each marker below that joint moving about a sphere centered at the joint's location. Silaghi et al. (1998) use this method, but rely on a weighted average of the calculated centers of rotation for each marker to find the joint position. Gamage & Lasenby (2002), on the other hand, provide a closed form solution for finding the local joint positions using least squares estimation. The Gamage & Lasenby (2002) method is described further in Kwon (2010). Wen et al. (2006) use a similar technique to the ones in this category.

Finally, Zordan & Van Der Horst (2003) use a different approach to the ones above, applying forces and constraints to the marker data to calculate the skeletal animation.

Overall, most of the literature uses similar techniques to find the joint positions, though there are several different methods suggested for minimizing the error. As Kiip is designed for real-time processing, the solution found in the first three papers was used, providing reasonably accurate data using an algorithm that executes quickly. Many of these papers did not provide full solutions, omitting specific steps or only providing vague descriptions of the techniques used. Thus, information from multiple sources had to be combined, and some equations re-derived, to find the final solution.

2.3 Retargeting

A common element in papers about retargeting is the use of inverse kinematics (IK) to change an animation to do something new. For instance, Choi (2001) uses IK to adjust a skeleton to meet specific goal poses, as well as to enhance the captured motion. Unlike standard forward kinematics (FK), which defines poses by moving down the skeletal hierarchy and applying rotations, IK defines a final position for a joint and moves up the hierarchy solving for rotations. This mimics how humans instinctively move, but is more computationally expensive and harder to constrain.

Another common approach is to use constraints that rely on what the motion does over time, or spacetime constraints. Gleicher (1997), Gleicher (1998), and Gleicher & Litwinowicz (1998) all use a combination of constraints on the character and IK to adjust the motion. Monzani (2001) also uses *spacetime constraints*, but first builds an intermediate skeleton to aid in the transfer of motion from the actor to the output skeleton. The problem with using these constraint-based approaches is that they generally rely on knowledge of what is happening in the animation. In some cases, they simply rely on information from frames in the future to calculate the current frame, such as the method in Gleicher (1998) for maintaining foot planting. Other methods require that the user make note of when certain actions begin and end. These requirements make such approaches inappropriate for real time retargeting where the nature of current and future motions is unknown.

While Kiip is designed to apply the motion of a human actor onto a humanoid character, there has been research done on applying animations to arbitrary characters. One interesting example is Hecker (2008),

which discusses the methods used to animate user-generated characters in the video game *Spore*[2]. This approach allowed animators to flag each animation as being applicable to certain body parts in certain situations. A specialized IK solver was then used to apply these animations to the game characters. Another method for applying motion to an arbitrary skeleton is (Poirier & Paquette, 2009), which first builds a skeleton for the mesh based on its shape, and then maps input motion onto that skeleton.

[2] http://www.spore.com/

3. System Overview

The basic Kiip system is shown in Figure 1. First, the user defines the number of markers and the correspondence between each marker and one of several skeletal areas (e.g., forearm, leg, etc.). Next, a calibration motion is captured. From this motion, the position and orientation of the joints in the actor's skeleton are calculated. Once the base skeleton has been calculated, the system begins calculating the skeleton based on real-time marker data. Finally, the system attempts to retarget the calculated motion to fit a more complex skeleton provided by the user, though this portion is not complete.

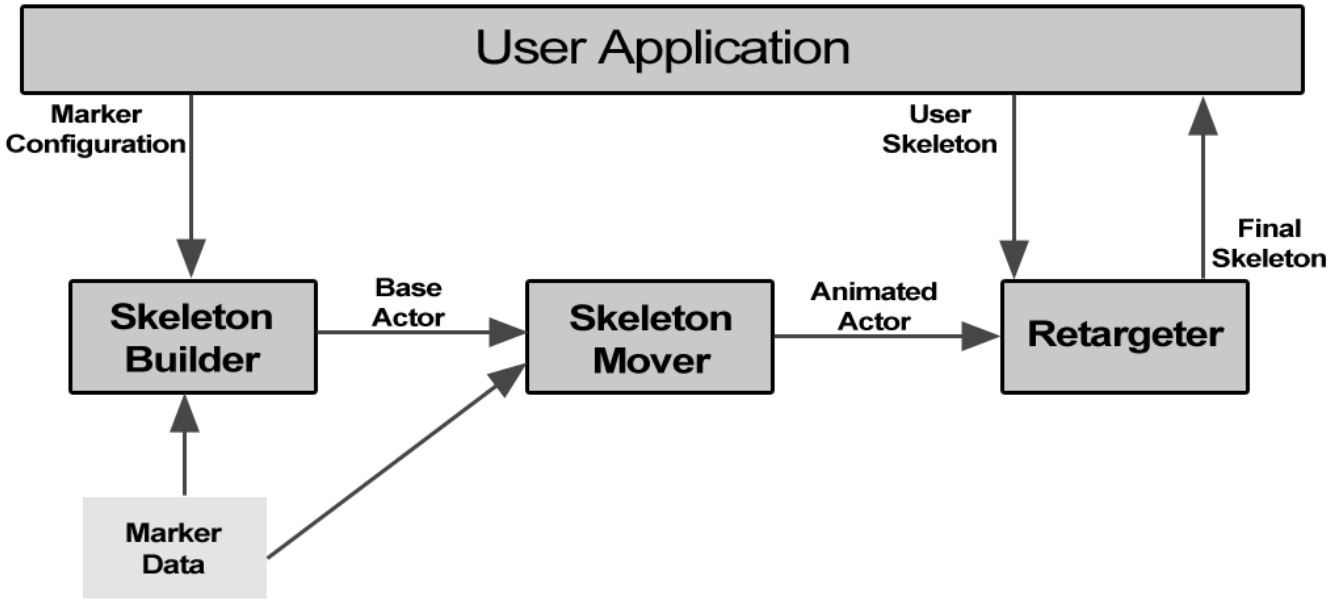

Figure 1 – Basic Kiip System

3.1 User Setup and Capture System

Kiip can be configured to accept input marker data from any source, however it was tested during creation using an eight-camera PhaseSpace[3] optical motion capture system. This system uses light-emitting diodes (LEDs) placed on the actor as markers, and then returns the 3D coordinates of these markers. By pulsing the LEDs at specific times, the system is able to accurately determine which marker it sees, thus avoiding any errors due to misidentified markers.

The user stands inside a ring of the eight cameras, wearing markers on each limb as indicated in Table 1. In general, each limb has four markers attached to it, though the feet, hands and clavicles each have three. These markers are semi-rigidly attached to the limbs to ensure that each marker maintains a constant distance from its controlling joint.

[3] http://www.phasespace.com/

Table 1 – Marker Counts

Limb	Marker Count
Lower Torso	4
Upper Torso	4
Clavicle	3
Upper Arm	4
Lower Arm	4
Hand	3
Head	4
Upper Leg	4
Lower Leg	4
Foot	3

The PhaseSpace system came with a full-body spandex suit pre-wired to accept markers. However, the suit was designed to hold far fewer markers on each limb than is outlined above. Also, it was difficult to take off and put on, and uncomfortable to wear. As such, a new modular suit was designed and created over the course of the project, and can be partially seen in Figure 2. Rather than use a single piece of material, the new suit is comprised of a separate piece for each limb, allowing the user to only wear as much of the suit as needed for a given application. Each piece is made of a Velcro-accepting fabric, allowing markers to be placed in arbitrary numbers and orientations on a given limb.

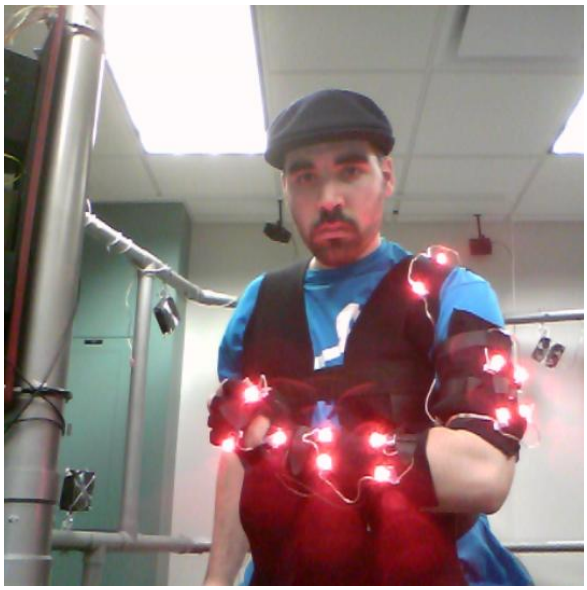

Figure 2 – Part of the New Suit

3.2 Marker Input

Before the system can work with input marker data, it needs some information about the markers. The user must define the number of markers that the system should expect to receive, as well as what limb of the body each marker is attached to. The possible limbs are the same as those listed in Table 1. To calculate the joint rotations, it is necessary to have at least three markers visible on each limb, with four being ideal for redundancy.

3.3 Building the Actor Skeleton

Now that the system knows how the markers are attached to the body, calculation of the actor's rest skeleton can be performed. First, the actor performs a *gym movement* which is recorded. The gym movement consists of rotating every major joint around each degree of freedom, which is important for the next step of calculating each joint's location.

The method for determining the joint location requires that the marker locations be in the local space of the joint rather than in world space. (Ringer, 2004) shows that the joint location can be solved from:

$$\boldsymbol{z_l^p(k) = R_l(k)e_l^p + t(k)} \qquad \textbf{Eq 1}$$

Where $z_l{}^p(k)$ is the world space position of marker *p* on limb *l* at frame *k*, $R_l(k)$ is the 3x3 matrix defining the orientation of the limb relative to the joint, $e_l{}^p$ is the vector from the joint to the marker *p*, and $t_l(k)$ is the world space position of the joint. The background of this equation is further explained in O'Brien, Bodenheimer Jr, Brostow, & Hodgins (2000).

3.3.1 Limb Rotations

Before the local marker positions can be found, $R_l(k)$ must be found. Charalambous (2005) proposes two methods for estimating this value, one for limbs with at least three visible markers, and a less accurate one for limbs with only two visible markers. Both methods use a reference frame and the current frame to calculate the rotation of the limb relative to the joint. Kiip uses the three-marker version, which is based on Singular Value Decomposition (SVD). This method is also derived in (Kwon, 2011).

Figure 3 shows two limbs, *X* and *Y*, which rotate about a joint. Limb *X* has three visible markers, while limb *Y* has four. The limbs are shown at the base frame (*k=1*) and at some later frame, *k>1*.

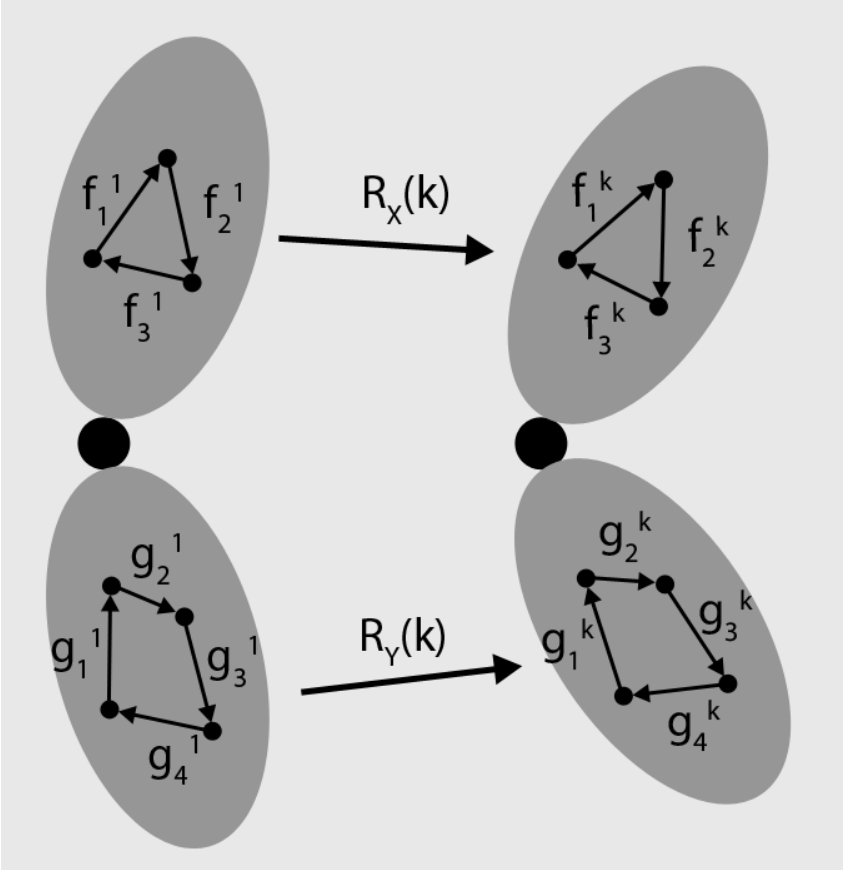

Figure 3 - Rotation of Limbs X and Y

$$M_1 = \begin{pmatrix} \uparrow & \uparrow & \uparrow & \uparrow \\ {g_1}^1 & {g_2}^1 & {g_3}^1 & {g_4}^1 \\ \downarrow & \downarrow & \downarrow & \downarrow \end{pmatrix}, M_k = \begin{pmatrix} \uparrow & \uparrow & \uparrow & \uparrow \\ {g_1}^k & {g_2}^k & {g_3}^k & {g_4}^k \\ \downarrow & \downarrow & \downarrow & \downarrow \end{pmatrix}$$

First, a pair of 3x*N* matrices is created, one at the base frame and one at the current frame, where *N* is the number of visible markers on the limb. *N* must be at least 3, but can be larger. Each column of the matrix is a vector from one of the markers to the next, going in a loop.

$$H = M_1 * {M_k}^T$$

Next, matrix *H* is calculated by multiplying the base frame matrix by the transpose of the current frame matrix.

$$H = U * W * V^T$$

Taking the SVD of *H* provides two orthogonal matrices, *U* and *V*, and a diagonal matrix, *W*.

$$R(k) = U * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & det(U * V^T) \end{bmatrix} * V^T$$

The limb rotation at the frame, *R(k)*, is then given by multiplying *U* by the transpose of *V*. However, Kwon (2011) notes that the resulting matrix may have a negative determinant, meaning that the rotation matrix is a reflection of the one desired. To fix this, the inner matrix shown above is inserted, which will correct the reflection if necessary.

This method provides a world-space rotation matrix of the limb relative to the base pose for that limb. This matrix is suitable for use in the following sections on finding joint positions, but does not work for finding the local rotation of a limb when retargeting.

3.3.2 Marker-to-Joint Vectors

With the rotations known, Charalambous (2005) and Ringer (2004) show that it is possible to solve for $e_l{}^p$. Looking at two different limbs, x and y, attached to a given joint produces the following:

$$\boldsymbol{z_x{}^i(k) = R_x(k) e_x{}^i + t(k)} \qquad \textbf{Eq 2}$$

$$\boldsymbol{z_y{}^j(k) = R_y(k) e_y{}^j + t(k)} \qquad \textbf{Eq 3}$$

where *i* is a marker on *x* and *j* is a marker on *y*. Rearranging these equations to solve for $t(k)$ produces:

$$\boldsymbol{t(k) = z_x{}^i(k) - R_x(k) e_x{}^i} \qquad \textbf{Eq 4}$$

$$\boldsymbol{t(k) = z_y{}^j(k) - R_y(k) e_y{}^j} \qquad \textbf{Eq 5}$$

which can be set equal to each other eliminating $t(k)$, and producing:

$$\boldsymbol{z_x{}^i(k) - R_x(k) e_x{}^i = z_y{}^j(k) - R_y(k) e_y{}^j} \qquad \textbf{Eq 6}$$

and finally:

$$\boldsymbol{b^{ij}(k) = z_x{}^i(k) - z_y{}^j(k) = R_x(k) e_x{}^i - R_y(k) e_y{}^j} \qquad \textbf{Eq 7}$$

This is roughly in the form $Am = B$ where,

$$A = [R_x(k) \quad -R_y(k)], 3x6$$

$$B = \left[z_x{}^i(k) - z_y{}^j(k)\right], 3x1$$

$$m = \begin{bmatrix} e_x{}^i \\ e_y{}^j \end{bmatrix}, 6x1$$

which, when solved, will provide estimations for $e_x{}^i$ and $e_y{}^j$.

3.3.3 Joint Positions

Finally, with all other variables known, the value of $t(k)$ can be found by solving Eq 1 for all p markers on a given limb, for all limbs attached to the joint. The average of the locations for all markers is the estimate. From this, the distance from each marker to its corresponding joints, as well as the limb length for each limb, can be calculated by averaging the values from each frame.

A rest pose is then chosen to be the base pose from which skeletal movement is defined. This pose is the first frame found in which all markers are visible, and is the same one used above to calculate limb rotations. Ideally, the actor will be in a pose similar to that of the skeleton to which motion is to be applied, as this creates less distortion later during retargeting. The world-space joint and marker positions in this pose are stored, along with the vectors from each joint to its adjacent markers. The final skeleton looks like the one in Figure 4.

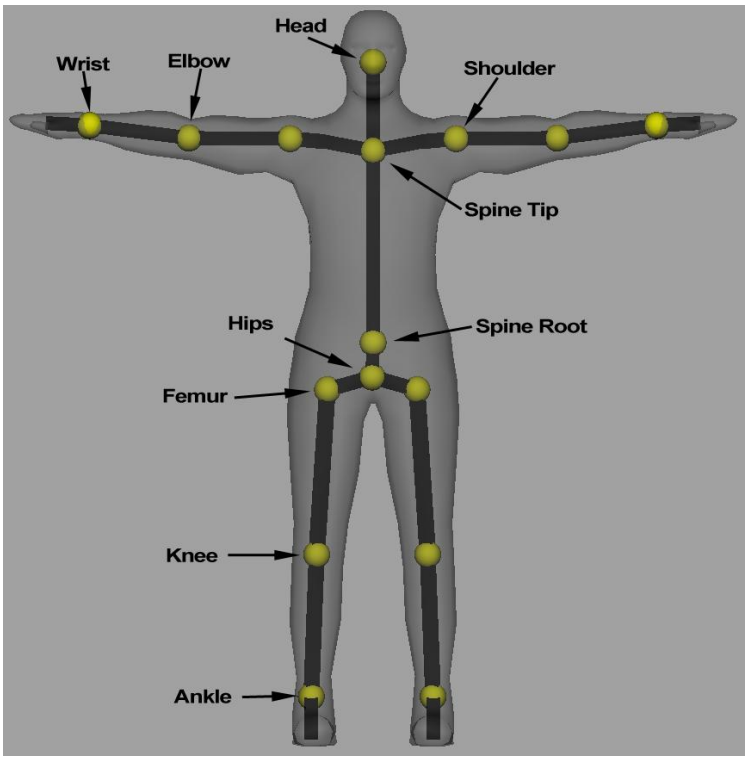

Figure 4 – Actor Joints

3.4 Skeletal Capture

At each frame during real-time capture, the position of the joints must be determined, again using Eq 1. As before, the rotation of each limb is calculated based on the world space positions of the markers on the limb. The $e_l{}^p$ vectors for each marker that were stored for the base pose can be multiplied by this rotation to find the vectors for the current frame. Finally, Eq 1 can be solved and averaged for all markers as before, providing base locations for the joints.

Once the joint locations are estimated, they are further refined by building the joint hierarchy outward from the root, which is assumed to be correct. Figure 5 shows how the positions for the children of the

root are calculated. A vector, v, is calculated from the root position, *R*, to the estimated child position, *C'*. Since the length of each limb, *L*, has been calculated, the position of the children joints must be constrained to that length. The final joint position, *C*, is placed at a distance of *L* along v. Ideally, the original estimation should be close to this point anyway, but this step ensures that the limbs of the skeleton maintain their rigid structure.

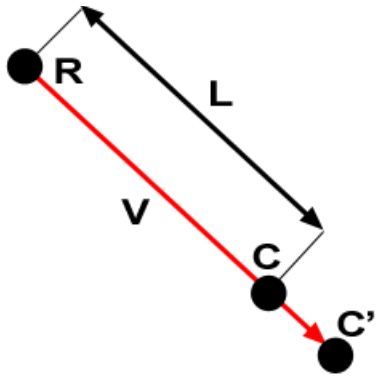

Figure 5 – Joint Location Correction

3.5 Skeletal Retargeting

3.5.1 Retargeting

Before retargeting can occur, the application must provide a character skeleton. Similar to marker input, the user must specify the type of each joint. Figure 6 shows the acceptable joint types that were intended to be handled by Kiip. Joints in yellow were to be required, while the others were optional. The red joints are roll joints that sit between two primary joints and help to smooth out the character's deformation. The user was to be able to specify multiple spine and neck joints, in order, if desired.

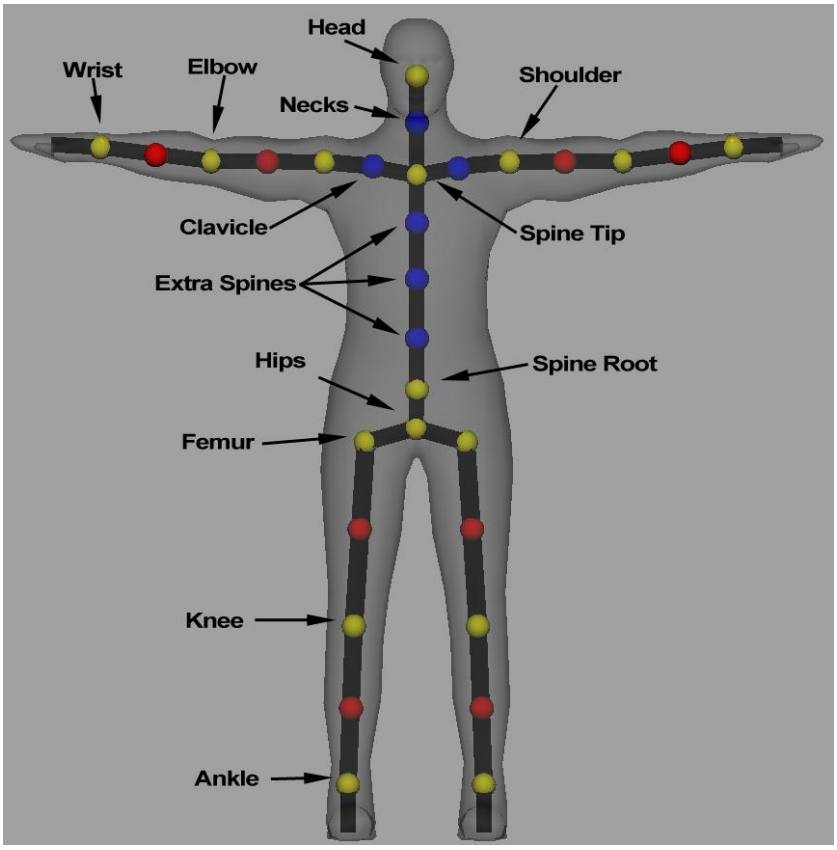

Figure 6 – Character Joints

As stated previously, the closer the provided skeleton's rest pose is to the actor's base pose, the more likely it is for retargeting to appear accurate. Monzani (2001) provides further examples and explanation as to why the base poses should be similar.

As little to no research was found on the topic of real-time retargeting, a method had to be devised for the project. The simplest way to retarget the motion is to copy all of the local rotational values onto the new joints. Given accurate rotation data, this is simple for most joints. However, for the spine and neck, this is more complicated as the actor skeleton only has locations for the start and end of the spine and the position of the head. Any intermediate joints in the user's skeleton would require that the single rotation be broken up and applied to multiple joints. Also, for the spine root, there are two output limbs from which to determine a world orientation for the skeleton, which must be resolved.

As of the end of the MQP, simple retargeting was implemented but unfinished. None of the more complex methods were implemented, but they are outlined in the next several sections for completeness.

3.5.2 Simple Retargeting

$$C_L = C_W * {P_W}^{-1} \qquad \textbf{Eq 8}$$

Eq 8 shows how to calculate the local rotation of a joint, C_L, given its world-space rotation, C_W, and the inverse of its parent's world-space rotation, ${P_W}^{-1}$, using simple matrix math. This equation assumes column-vector matrices. The actor skeleton is not stored in a hierarchical relationship, but those relationships can still be used here (the shoulder is the parent of the elbow, for instance).

$$T_F = C_L * T_B \qquad \textbf{Eq 9}$$

Eq 9 shows how the user joint's final transform, T_F, is calculated based on the previously found local rotation offset and the base transform, T_B, for that same user joint.

3.5.3 Spine and Neck Retargeting

Figure 7 shows a side-view of the actor spine (left) and a possible character spine (right). To retarget the movement of the spine, the start of the two spines must be aligned. Then, the end of the spines must also be aligned by calculating rotations for all of the intermediary spine joints along the way. The end points are not actually the same world space position, since the user's skeleton is likely of a different scale than the actor. To address this, an offset must be calculated based on the base poses for both skeletons, and used to calculate the actual end point of the user spine. To calculate the rotations of the various spine joints, an IK chain can be used. The resulting rotations may then be assigned to the spine of the character's skeleton.

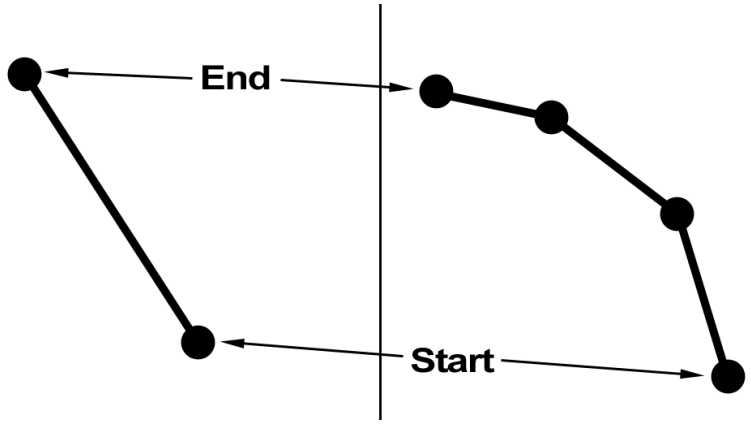

Figure 7 – Spine Retargeting

A similar method can be used for calculating the rotation of any neck joints, with the end of the spine acting as the start of the IK chain, and the head acting as the end. Once again, a scale factor must be calculated from the base skeleton to find the position of the head on the user's skeleton based on the head of the actor.

3.5.4 Clavicle

The clavicle joints can sit somewhere between the tip of the spine and the shoulder, connecting the two. While the actor moves, the position of the shoulder joint may also move relative to the spine. Without a clavicle, this movement is not possible as rotating the spine joint would cause other joints to be in the wrong positions; thus, if no clavicle is provided, the movement is lost. If a clavicle is provided, its rotation can be set such that it points to the location where the shoulder should be placed, preserving the shoulder movement.

3.5.5 Roll Joints

Figure 6 shows upper and lower roll joints for the arms and legs in red. These joints sit between the main captured joints (shoulder, elbow, wrist for the arm; femur, knee, ankle for the leg) and distribute the twisting rotations to provide more realistic deformation of the muscles and bones in those areas. In particular, the upper roll joint distributes the twisting of the shoulder, and the lower joint distributes the twisting of the wrist; the elbow ideally acts as a hinge joint and so should not have any twisting to distribute. Retargeting to the roll joints involves calculating the twist along the limb's axis, and then dividing that twist so that a portion of it is in the rotation of the primary joint and the other portion is in the roll joint.

3.5.6 Foot Planting

The final type of retargeting for the system is adjustment of leg movement to ensure that the feet do not penetrate the ground. Since the user's skeleton is of a different scale than the actor skeleton, simply applying the rotation of the actor's leg joints to the user's skeleton may result in the feet floating above or penetrating through the floor. To fix this, IK chains can be attached to the legs and used whenever it is detected that the feet have penetrated a user-defined plane. The IK handles will then adjust the rotation of the femur and knee joints to pull the feet back above the ground plane.

4. Implementation

4.1 System Overview

The Kiip system is a C++ static library that can be linked into other applications. The system is comprised of one main class, "`Kiip`", which creates, and interacts with, various subsystems as needed. Figure 8 shows a high level overview of the system. For each actor being tracked, the user instantiates a `Kiip` object which acts as an interface between the user and the system. Based on user requests, the `Kiip` object interacts with the subsystems, sending and receiving data between them as needed and storing the most recent data to be provided as the user requests it.

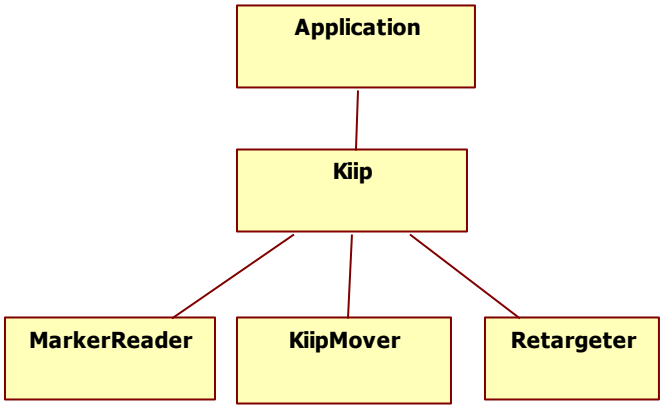

Figure 8 – Kiip Implementation Overview

On instantiation, the `Kiip` object spawns a thread which continually loops until the object's destruction. As the `Kiip` object must not block when user code requests data, a copy of the most recent data is always held by the `Kiip` object. The looping thread polls the various subsystems for new data, and stores its own copies of this data. The `Kiip` object also maintains a single current task, explained in Table 2. The current task is changed by various user calls, moving through the various initialization stages until finally beginning general capture and retargeting.

Table 2 - Kiip Task States

`kKiipNoTask`	`Kiip` is idle
`kKiipFindBasePoseTask`	Searching for a frame where all markers are visible
`kKiipCaptureGymMotionTask`	Capturing and storing frames of the gym motion
`kKiipProcessGymMotionTask`	Waiting for the Mover to process the gym motion
`kKiipProcessMotionTask`	Standard data processing from frame to frame

4.2 Marker Input

Markers are the main data source for Kiip, and up-to-date marker data is required for all of Kiip's computations. Kiip does not handle the tracking of markers, but rather reads in this data from another source.

4.2.1 Marker Reader

A `MarkerReader` object is responsible for reading in frames of marker data from a source. Figure 9 shows the base class for Marker Readers, as well as the subclass used to read from the PhaseSpace system. The application instantiates a `MarkerReader` subclass, and passes it to the `Kiip` object, along with a list of marker correspondences, as described in Section 3.2. The application must be the one to instantiate this instance because each subclass may have specific initialization information that only the application knows, such as a filename to read from or a server address to connect to.

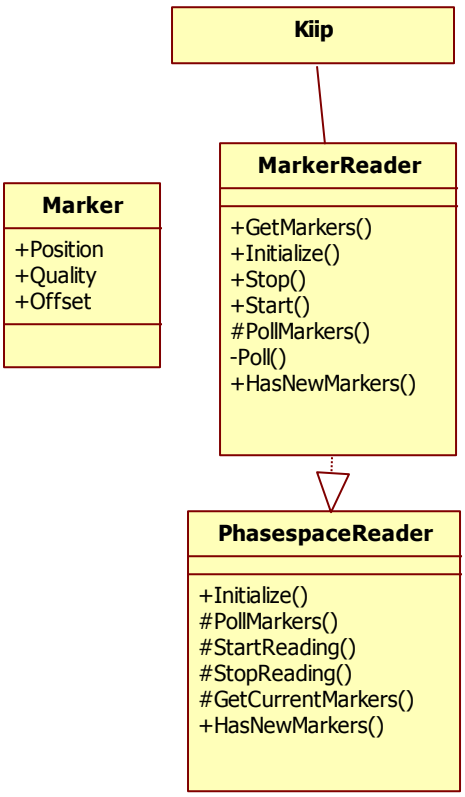

Figure 9 – Marker Input Overview

When the application tells Kiip to begin collecting marker data, the `Kiip` object will tell the `MarkerReader` to begin polling using `Start()`. The `MarkerReader` super class will begin calling the subclass's `PollMarkers()` function in a new thread, which will continuously poll for marker data each

frame until `Stop()` is called. The `Kiip` object can access the most recent frame of marker data by calling `GetMarkers()`.

4.2.1.1 PhaseSpace Reader

The `PhasespaceReader` is able to connect to a desired PhaseSpace server and capture live marker data. It has various options including server address, capture rate, data interpolation, and capture mode, as defined by the PhaseSpace API.

4.2.2 Markers

The marker data being passed around consists of three values per marker: the 3D position, a quality value, and later an offset from the joints it rotates about. The quality value tells other parts of the Kiip system how that marker should be interpreted. For instance, when a marker position is known, it is assigned a quality of 'good', while if the position is unknown it is assigned 'bad'. When the existence of the maker is unknown, it is marked as 'unknown'.

4.2.3 Marker Info

In addition to a `MarkerReader` implementation, the user must pass an array of `MarkerInfo` objects to Kiip. These allow the user to define which limb each marker is attached to. This data is then passed on to the `MarkerReader`, and as such can be subclassed to include additional info for the specific reader. For example, the PhaseSpace Info subclass can assign a PhaseSpace marker ID that is different from the ID used locally by Kiip.

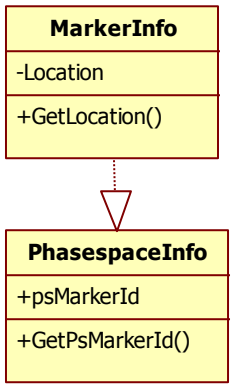

Figure 10 - Marker Info Class

4.3 Building the Actor Skeleton

Before capture can proceed, Kiip must determine and store various pieces of information about the current actor. This data is used in later steps as both a base and timesaver for computations.

4.3.1 Finding the Base Pose

The first step in capturing a skeleton is to capture a base pose for the actor. This pose should be as close as possible to the skeleton the motion is to be applied to for best results. The application tells Kiip to begin searching for a suitable base pose with the call "`kiip->findBasePose()`". This causes Kiip to begin checking every new frame of data until it finds one where all defined markers are visible. This becomes the base pose for the current session.

4.3.2 The Gym Motion

After finding a base pose, the application must call "`kiip->captureGymMotion(time, fps)` ". This will cause Kiip to begin recording frames of marker data into a buffer for '*time*' seconds at a rate of '*fps*' frames per second. While this data is being recorded, the actor should be moving his various limbs through a routine that moves them through their normal range of motion. Once this capture is complete, the application can call "`kiip->processGymMotion()`", which will send the base pose and the captured frames to the `KiipMover`.

The `KiipMover`, as shown in Figure 11, is responsible for calculating the positions of the joints and the orientations of the limbs, as well as processing the gym motion. The first step in this processing is to determine the orientation of each limb for each frame. Using the method described in Section 3.3.1, the rotation is calculated for all limbs before moving on to the next step. Once the limb rotations are known, the marker offset vectors and joint positions can be calculated for each frame. Some joints may not be calculated on certain frames if their associated limb rotations could not be found in the previous step. With the data for each frame known, the `KiipMover` proceeds to average the values it has calculated for each frame, storing the average offset vectors for each marker, as well as an average length for each limb. The former allows for quicker calculations later in the process, while the later allows for correcting the joint positions that are found.

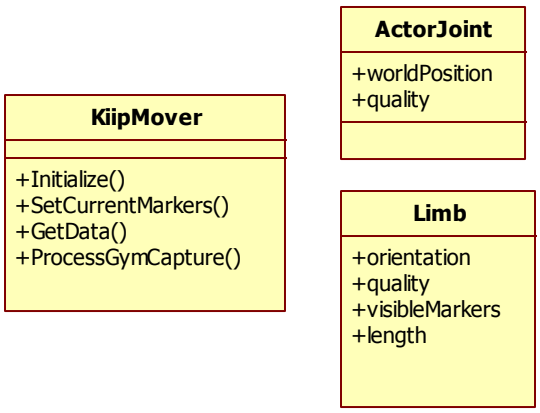

Figure 11 - Actor Mover

4.4 Skeletal Capture

Once the `KiipMover` finishes processing the gym motion, the main `Kiip` loop automatically switches to skeletal capture. This process also takes place within the `KiipMover`. Using all of the pre-calculated data, along with new marker data, the positions and orientations of visible joints and limbs are found and stored for later access.

4.4.1 Frame Processing

When the `MarkerReader` has a new frame of marker data ready, and the `KiipMover` is not busy, `Kiip` will pass the new marker data into the Mover. This spawns a new thread responsible for processing of the frame, using the method from Section 3.4. First each limb's rotation is calculated based on the base pose and the current frame, as before. Then, for each visible marker, a position is found for the joint by multiplying the stored offset for that marker by the limb's rotation, and subtracting this value from the marker's current position. All of these rough position values are averaged across all limbs attached to a given joint, providing the final joint position. Finally, the joint positions are adjusted based on the stored limb lengths found earlier.

By pre-calculating the marker offsets from the gym motion, the process above can remove a step from the calculations, providing a speed increase. More importantly, calculating the offsets requires that two limbs around each joint have enough visible markers to calculate their orientations. By pre-calculating the offsets, joints can often still be found even if one of their attached limbs is missing.

4.4.2 ActorJoint

The `KiipMover` provides an `ActorJoint` for each requested joint in the system. These store both a world-space joint position, and a quality value. Similar to `Marker`s, the quality specifies if the joint position for the frame is calculated, unable to be calculated, or if the joint cannot exist given the available markers.

4.4.3 Limb

The `KiipMover` also provides a `Limb` for each limb in the system. These store a world orientation, a quality value, the number of markers visible on the given limb, and the constant length of the limb.

4.5 Skeletal Retargeting

The `KiipRetargeter` subsystem, when enabled by the application, is designed to apply the animation of the actor to a skeleton provided by the application as described in Section 3.5. The user passes local-space transform matrices of a skeleton's base pose to `Kiip`, which activates that joint in the `KiipRetargeter` and stores the matrix. Once a frame has been processed by the `KiipMover`, the `ActorJoint` and `Limb` data are passed to the `KiipRetargeter` for final processing. The final outcome is a series of transform matrices which apply new transforms to the base matrices passed in by the user.

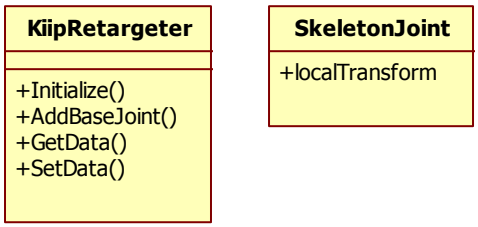

Figure 12 - Kiip Retargeter

With the exception of the neck, spine, and clavicles, all of the joints are handled by a single function. This function is intended to calculate local rotations for each limb, and apply them to the base matrices, as described in Section 3.5.2. Unfortunately, this portion of the project is incomplete, as will be addressed further in the evaluation section.

4.6 Input/Output

4.6.1 Data

Internally, Kiip uses a linear algebra library to store vectors and matrices. However, applications using Kiip will most likely have their own data structures for these elements. To avoid complicating data access, all previously defined structures (Markers, Actor/Skeleton Joints, Limbs) have a 'user' version where vectors and matrices are stored as simple arrays.

4.6.2 Logging

Kiip uses a simple global logging class with multiple levels of debug logging, as well as error, warning, and general info. The desired verbosity can be changed at both compile time, and at run time. By default, the log prints to a file called "kiip.log" in the application's working directory, but this can also be changed by the user at runtime.

4.7 Libraries

Kiip uses two external, cross-platform libraries for linear algebra, threading and other tasks.

4.7.1 Eigen

Eigen[4] is a free C++ template library providing fast implementations of standard linear algebra concepts, including those specific to 3D animation such as 4x4 transformation matrices and quaternions. Eigen also includes modules for SVD and least-squares calculations, which are both at the core of the motion-capture process described here. Kiip was developed using Eigen version 3.

4.7.2 Boost

Boost[5] is a set of free, cross-platform peer-reviewed C++ libraries. Kiip uses the Boost Thread[6] library to handle threading. Kiip also uses a handful of other Boost components, including the Date/Time[7] library for accurate timers. Kiip was developed using Boost version 1.45.0.

4.7.3 PhaseSpace API

Kiip also uses a library provided by PhaseSpace to communicate with the motion capture equipment used for testing.

[4] http://eigen.tuxfamily.org/index.php?title=Main_Page
[5] http://www.boost.org
[6] http://www.boost.org/doc/libs/1_45_0/doc/html/thread.html
[7] http://www.boost.org/doc/libs/1_45_0/doc/html/date_time.html

5. Applications

Despite the fact that the retargeting system is incomplete, Kiip is still able to calculate joint positions. To demonstrate that Kiip is application independent, it has been integrated into two different applications. Both of these applications use similar code to integrate Kiip, and a generic overview of this integration code is provided in Appendix A - Integration.

5.1 C4 Game Engine

The first application uses the *C4 Game Engine*[8], and was built alongside the Kiip library. When run, it generates spheres representing marker and joint positions of the actor. C4 uses a z-up orientation, with row-vector matrices, while Kiip uses y-up and column-vector matrices. The C4 implementation includes an interface which converts positions and matrices between the two systems. Once a frame, the interface polls Kiip for the newest data, and stores it for access by other game-world elements. Figure 13 shows an example of the C4 application in use. The small spheres represent marker locations, and change color between green and red depending on the quality of the data. The larger spheres are calculated joint positions. The grey shapes near the bottom represent the joints of an arm.

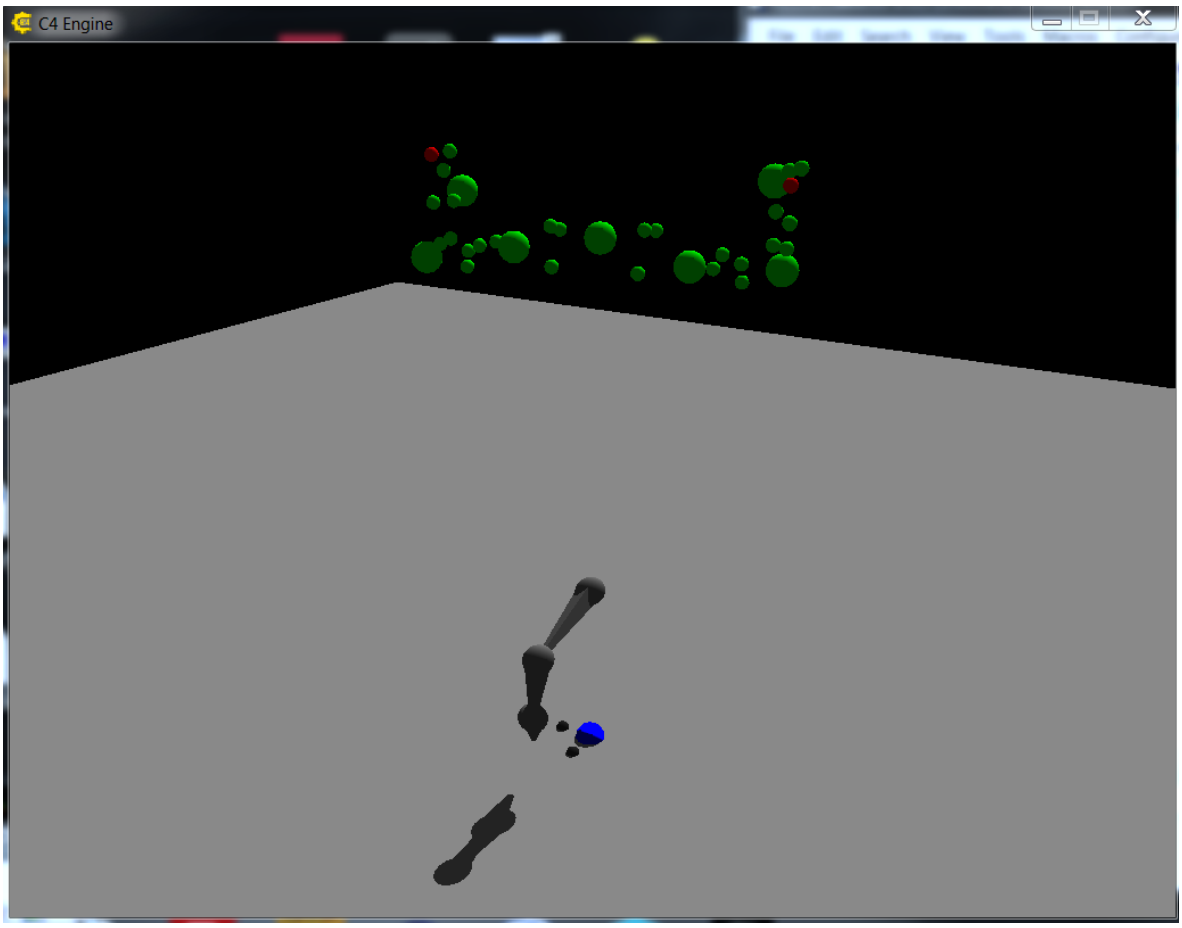

Figure 13 – C4 Application

[8] http://www.terathon.com/c4engine/index.php

5.2 Maya Plugin

The second application is a plugin for displaying the data in Autodesk's *Maya*[9], a standard application for creating 3D content. The original goal of this plugin was to allow producers to get a visual representation of how motion capture will look on a character while the capture session is taking place. Normally, only a view of the markers is available, hindering the producer's ability to know what the final animation will look like. As the retargeting is not complete, this plugin is essentially used to show that the library is application independent. The plugin currently performs broken retargeting onto a skeleton in Maya, showing that the application is sound even if the internals of the library are not. While the C4 implementation was created alongside Kiip, the Maya plugin was created once the API was set in place as a way of testing ease of integration.

Unlike C4, Maya uses the same y-up column-vector system as Kiip, so no translation of data was needed. However, Maya is not designed to stream data in for playback; it is built on a node-based architecture, and does not work well with data that changes independently of its internal timing. Also, Maya does not allow transformation matrices to be set directly through its node graph, but instead wants the decomposed translation, rotation, and scale values only, along with some internal undocumented offsets. To overcome this, the plugin takes advantage of the ability to set a callback to run every time the current time is set in Maya. Pressing the 'Play' button, normally intended to playback keyframed animation, causes a special command to be run that queries a Kiip object attached to a Maya node for new data. This data can then be set directly to a transform in the scene, without first having to decompose it. Figure 14 shows an example of the Maya plugin. The green symbols near the top represent marker locations, while the structure at the bottom are the joints of two arms.

[9] http://www.autodesk.com/maya

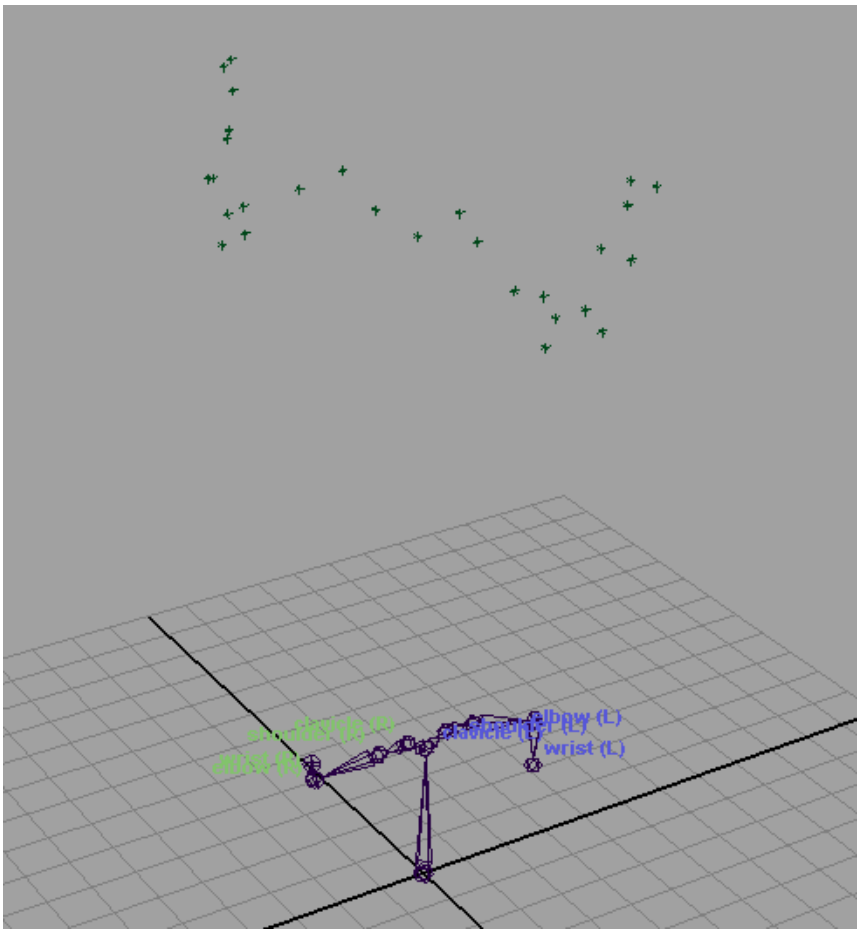

Figure 14 – Maya Plugin

An alternate method for such a plugin, and also for generic use of Kiip, would be to build a local server that passes data between Kiip and Maya over sockets. Maya does have a method for sending network data to its node-graph, but it is fairly limited. This would potentially also sidestep the need to 'Play' the timeline to force updates. Again, this server-based method could also be useful for other integrations, especially when the destination software is not written in C++.

6. Schedule

This project began in B-term of 2010 and was completed in E-term of 2011. A division of work by term follows.

6.1 B-Term

B-term was spent researching background information for the MQP. The majority of the necessary research was compiled over this term, and the fundamentals of the math needed for the process were determined.

6.3 C-Term

C-term was primarily used for setting up the basic Kiip framework, and implementing a proof-of-concept for the calculation of joint positions. This term also included:

- prototyping of the improved capture suit
- debugging of the PhaseSpace system
- sourcing of parts for the new suit's wiring and camera mounts
- determining and implementing a plan for mounting the eight capture cameras to provide a stable and large capture volume

6.4 Beyond C-Term

Due to external complications, the project was put on hold over the course of D-term. Work resumed in E-term, and included:

- generalization of the joint position algorithm to work with an arbitrary set of limbs
- creation of cabling and pieces for the new suit
- creation of the Retargeter sub-system, which involved testing many different methods for calculating the final transforms to be output
- Reworking the C4 implementation to be a better integration example, along with implementation of Maya plugin implementation

7. Discussion

7.1 Joint Position Accuracy Testing

7.1.1 Methodology

The main data output that Kiip successfully produces are the positions of the actor's joints. To test the accuracy of this data, a simple jig was setup in Maya. The Maya jig was chosen, as using data from a real actor would introduce several forms of error into the testing. One major source of error comes from the fact that the markers attached to the actor are not completely fixed relative to the actor's joints, and can move slightly along the actor's limb. Another source comes from the inability to accurately measure the real position of each joint. It would be possible to construct a physical jig in place of the virtual one used in Maya, but no sources of easily measurable and usable joint-like devices could be found, and even these could potentially have error. Using a virtual input from Maya ensures consistent data quality, and removes any outside variables This allows for testing to focus solely on the accuracy of the algorithms in Kiip.

The Maya Jig consisted of four joints, mimicking a human arm (clavicle, shoulder, elbow, wrist). Only the last three had positions calculated, with the clavicle used to provide the second limb needed to find the shoulder. Each limb had exactly three points (called *locators* in Maya) parented rigidly to it. Five tests were performed, with the rotation of all joints changed for each test. After the first 3 tests, the base pose was changed.

7.1.2 Results

Table 3 shows the results of the five tests, separated by joint. For each test, both the standard SVD rotation and an alternate rotation (discussed in Section 8.1) were calculated, and were individually used to calculate the joint's position as described previously. The first column shows the difference between the position calculated using the SVD method, and the one calculated using the alternate (triangle) method. The second column shows the distance between the real joint position, and the position found using the SVD method. The final column shows the same, but using the position from the triangle method. All units are in centimeters.

Table 3 - Test Data (units are in cm)

Limb	Method Difference	SVD Error	Triangle Error
Test 1			
Shoulder	0.069	0.505	0.442
Elbow	< 0.001	0.757	0.757
Wrist	< 0.001	0.984	0.984
Test 2			
Shoulder	0.417	0.892	0.851
Elbow	0.441	1.354	1.215
Wrist	0.231	1.194	1.092
Test 3			
Shoulder	0.204	0.425	0.416
Elbow	0.264	0.903	0.700
Wrist	0.161	1.210	1.318
Test 4			
Shoulder	0.237	0.425	0.586
Elbow	0.206	0.903	0.899
Wrist	0.263	1.210	1.345
Test 5			
Shoulder	0.217	0.661	0.538
Elbow	0.548	1.082	1.174
Wrist	0.169	0.999	1.113

Table 4 shows the average values across all tests, again divided by joint. Table 5 shows the maximum value for each calculation.

Table 4 – Averages (units are in cm)

Limb	Method Difference	SVD Error	Triangle Error
Shoulder	0.229	0.582	0.567
Elbow	0.292	1.000	0.949
Wrist	0.165	1.120	1.170

Table 5 – Maxima (units are in cm)

Limb	Method Difference	SVD Error	Triangle Error
Shoulder	0.417	0.892	0.853
Elbow	0.548	1.354	1.215
Wrist	0.263	1.210	1.345

7.1.3 Conclusions

The first conclusion that can be drawn from the data is as regards the accuracy of the two methods for calculating the orientation of a limb. On average, the distance between the two calculated positions was less than a third of a centimeter, with the largest difference seen equal to 0.55 centimeters. Comparing the average error from the actual position shows an even smaller difference (ranging from 0.014cm to

0.051cm). These results seem to indicate that both methods are roughly comparable in accuracy, and thus the choice of which to use should be based on which method would be better suited for retargeting calculations later in the process.

The other conclusion to be drawn is the accuracy of the system's ability to find actor joint locations. The highest error seen was 1.354cm, with the overall average at 0.898cm. This data shows that, while not exact, Kiip is able to fairly accurately calculate the position of the actor's joints. This level of accuracy is sufficient for the intended uses of the library, where the overall movement of the actor is more important than absolute accuracy. Also, it is likely that retargeting to skeletons with proportions different from the actor's would cause larger error than seen here.

Overall, the data appears to show that, in the absence of external error, the algorithms used by the system are sufficiently accurate for calculation of joint positions. As previously stated, capture of a physical actor has the potential to introduce additional error into the calculations that would make the calculated positions less accurate, though, observation has shown that the general movement is still correct, and the error introduced by retargeting will likely outweigh any error from the calculation of the joint positions.

7.2 Retargeting

7.2.1 Research

A significant portion of E-term was spent attempting to perfect algorithms for retargeting the motion. While researching in B-term, very little useful information was found regarding methods for retargeting. Almost every paper seemed to discuss offline techniques that required either knowledge of the movements at a given time, or knowledge of future movements. These methods are not applicable to Kiip's design, as all calculations must work for any motion without knowledge of the future. It would be possible to buffer the data by several frames, allowing for knowledge of future frames, but this would cause a delay in output. For applications like the Maya plugin, which would be used solely for observing the results of movement as applied to a character, a small delay could be acceptable. However, for applications that are designed to immerse a user within a virtual world, any delay between a user's movement and his avatar's movement could break the immersion completely. In particular, if a game using the system required fast reactions, a delay would ruin the gameplay experience.

7.2.2 Proposed Methods

The inability to find research on the topic meant that methods had to be devised from scratch. The first several methods tested used the vectors between joints to calculate the appropriate angles. By taking the vector between two joints at the base pose and at the current frame, a cross product and a dot product of those vectors will provide an axis and an angle respectively. A rotation matrix that rotates about this axis by the angle will also rotate one of the limb vectors to the other. This simple method, working limb by limb, would retarget the skeleton such that the joints would be in the right positions, but would have improper rotations and deformations. For instance, using this method on the shoulder, a ball joint, will point the shoulder in the right direction, but will not take twisting into account. Then, using this method

on the elbow, a hinge joint with one degree of freedom, will 'break' the elbow to compensate for the lack of twist, causing it to move with too many degrees of freedom.

The same concept of calculating the rotation between two vectors can be applied to other possible methods. Another method attempted to work backwards, and handled entire portions of the body, as arms and legs have the same general setup (three joints, with a hinge joint in the middle). This method first attempted to rotate the current joints about the shoulder, to align with the base shoulder, the inverse of above. Then, the shoulder was twisted to align the forearm to the same plane as the base pose. Finally, the elbow was aligned to the base pose. Since the shoulder's twist was already taken into account, the theory was that the elbow rotation calculated would only have one degree of freedom. In practice, this method didn't work at all, and was quickly dropped in favor of the method described in Section 3.5.2.

One final method that would likely work well is to use an IK solver to calculate the joint orientations. It was intended that IK would be used in some parts of the body, but the delay in finding a solution for the other parts of the body meant that there wasn't enough time to research a suitable IK solver. Unfortunately, a non-IK solution was needed for the leaf joints (the head, hands, and feet). These joints have no children, and so cannot use IK or the other methods suggested above to find their orientation, save for the method in Section 3.5.2.

7.2.3 Complications

One major complication in the processes above is factoring out the overall transformation of the actor. For instance, if the actor simply bends his elbow, the angle between the base forearm and the current forearm can be simply calculated as above. If the actor then rotates his shoulder by some amount, then this same method does not work, as the new forearm vector will include the rotation of the arm and the rotation of the shoulder. Calculating the angle now will provide an inaccurate result. If the actor then rotates his entire body, the result will be even more incorrect. A large amount of time was spent trying to find the best way to factor out these other rotations so as to find only the local rotation for each joint.

Another complication that hindered progress was a misunderstanding of how the SVD method of finding a limb's rotation worked. Simple tests with a jig in Maya seemed to indicate that it provided a simple offset rotation between the current frame and the base frame. When the parent joints didn't move, the SVD method produced a roughly accurate result, as compared to the actual rotation taking place, and factoring in the rotation of the parent joint produced the same result as having a static parent joint. However, the world orientation of the system did greatly affect the output rotation value. Due to an oversight, the tests in Maya were performed such that the primary joints were always pointed into one of the positive Z quadrants. As they moved closer to aligning with the Z axis, the calculated rotations became less accurate, to the point of flipping certain axes when pointing into negative Z quadrants. Meanwhile, the test data being used was of an arm whose base pose was pointing into the negative Z quadrant. Much time was spent trying to find errors in the code and trying alternate algorithms for retargeting due to this oversight.

Not all of that time was wasted though, as it caused the discovery of another problem. As previously mentioned, Kiip uses a y-up coordinate system with column-vector matrices, while C4 uses z-up. This was known, and was thought to be corrected by a simple rotation when passing data between the two. However, eventually it was discovered that C4 uses row-vector matrices. Compounding this problem, it was discovered that Eigen also uses row-vector matrices for its built-in transformations and quaternion conversions. This meant that the column-vector rotations were being incorrectly used with the input row-vector matrices. It also meant that matrix multiplications were being performed in the wrong order, as column-vector math was assumed. Once this was discovered, the Eigen transformation classes were replaced with normal Eigen matrices, and the transformation math was instead done by hand. Matrices moving between Kiip and C4 now must be both rotated and transposed before being used.

7.2.4 Conclusions

The lack of concrete methods for retargeting was the biggest challenge in the project. Ideally, more time could have been spent finding existing research on the topic of real-time retargeting, however the limited time in the overall schedule precluded it. Instead, a basic method that seemed feasible was proposed in B-term, which turned out not to work. In hindsight, some of the time spent trying unworkable solutions could have been spent on additional research, but at the time the solutions seemed like they would probably work.

Also, the problem of factoring in the overall transformations to find local joint rotations was underestimated. Had the difficulty of this problem been discovered sooner, the best course of action would have been to find an expert in 3D math to consult on the problem.

Finally, it is possible that some of the earlier methods could work to find the rotations of limbs connected to two joints. These were attempted before the row/column-vector problem was discovered, and also didn't properly factor in the overall actor transformation properly. Given more time, it is possible that one of these could produce a suitable outcome, however some other method remains to be found for handling the leaf joints.

8. Future Work

8.1 Alternate Limb Rotations

The method for finding limb rotations does not provide a rotation for the base pose, and is not always accurate. As finding the joint positions uses these rotations, an alternate algorithm for finding the rotations may improve calculation of the joint positions and retargeting. In addition, having access to the orientation at the base pose may allow for the creation of alternate retargeting methods that use this data.

One alternate way of calculating the limb rotations is to find a set of three markers on a limb and treat those as a triangle. The base pose is required to have all markers visible, but that is not the case from frame to frame, so the three markers used may be different each frame. For that reason, the base pose rotation must always be re-calculated each frame using the same markers as the current rotation.

Calculating a rotation requires three orthogonal vectors. The first two of these can come from two of the edges of the triangle. If the three vertices of the triangle are *A*, *B*, and *C*, then the *X* and *Y* vectors of the matrix can be calculated as:

$$X = {B - A}/{\|B - A\|}$$

$$Y = {C - A}/{\|C - A\|}$$

As indicated, the vectors must be normalized. The third vector can be found by taking the cross product of the *X* and *Y* vectors, producing the triangle normal, *Z*:

$$Z = {X \times Y}/{\|X \times Y\|}$$

Finally, these can be placed into a matrix, *R*, with each vector representing a column:

$$R = \begin{bmatrix} \uparrow & \uparrow & \uparrow \\ X & Y & Z \\ \downarrow & \downarrow & \downarrow \end{bmatrix}$$

As before, the current matrix will still need to have the overall transformation of the actor factored out to be useful. Also, there may be other, more useful, methods for calculating the orientation that can be used in place of either this or the original SVD methods.

This method of calculating the orientation has been implemented in Kiip, and the current and base orientations for each frame are passed along in the Limb structures. Tests with arbitrary triangles in Maya suggest that this method finds the rotation very accurately when there is no other rotation being added in, but still succumbs to the same retargeting issues as the SVD method, as described in Section 7.2.3. The accuracy of this method with regard to calculating joint positions was discussed in Section 7.1.

8.2 Completed Retargeting

One major piece that future work should focus on is completing the retargeting subsystem. At the most basic, just finding a method that solves all of the previously listed problems would be beneficial. Beyond those, implementing IK for use in the spine and neck, as well as foot planting, is a possible avenue, as discussed in Section 3.5.

More complex still would be to perform adjustments based on the difference in scale between the actor and the digital character. The suggested methods only copy rotations 1:1 between the two. If there is a significant discrepancy in size, for instance one is a tall adult and the other a small child, the animations will look incorrect; the stride-length will be wrong, one may have to reach to move to the same position as the other, etc. These are some of the more powerful features of commercial software like MotionBuilder, and would be a significant undertaking to implement well.

8.3 Generic Server Application

The end of Section 5.2 discusses the idea of using a client-server model to pass data between Kiip and an application. This would be useful for situations such as the Maya plugin, where streamed in data can potentially be easier to apply than going through the node-graph. In addition, creating a generic server application to pass Kiip messages would allow the library to be used with any software capable of socket-based communication. This may be necessary for integration with an engine such as *Unity3D*[10], which uses C# in place of C++, or to make the data available to scripted interfaces rather than fully integrating Kiip with the application.

8.4 Data Input/Output

Currently, all Kiip data is only available at runtime, and is not saved out to a file. Creation of a generic file format for marker data may be useful, for instance, to allow sessions to be run using the PhaseSpace equipment, and later played back at a remote location. When played back, the data would be interpreted by Kiip and output data would be produced just as it was during the original capture session. This would require the creation of a new MarkerReader implementation to read in the data, as well as an application to write out the data to a file. These tasks could be done without changing the core Kiip software. A more involved approach could integrate the output into the Kiip loop, writing out marker data as it is received. The base pose could also be saved out, and a new way of setting the base pose provided to ensure that the data is always interpreted the same way from run to run.

Another potentially useful output application would save out the data produced by Kiip (Actor Joints/Limbs and eventually retargeted joints). This would then allow Kiip to be used as the first step in an offline animation process, potentially reducing the need for MotionBuilder for the production of simple animations.

[10] http://unity3d.com/

9. Conclusion

The development of Kiip has involved both success and failure. The retargeting components do not work, but the actor data is fully calculated and available in real-time. Despite gaps in the explanations of many of the cited references, much of the missing information was determined over the course of the project, and has been documented here for use in future research. Even though the retargeting is not complete, the framework is setup and future work could fill in the details of retargeting specific joints; changing which joints can exist and how each one is retargeted is very simple, and all of the data required to perform the calculations is available to functions within the Retargeter.

This project was designed to try and provide a small subset of the features provided by expensive commercial software such as MotionBuilder in a free and easily integrated library. That parts of the project were only partially complete is a testament to the complexity of such software and the effort put into its creation. Still, despite these partial failings, the project still succeeded in providing a free implementation of some of the functionality of this software. This makes it, at least in part, a success.

References

Charalambous, Themistoklis. 2005. Interacting with a virtual environment. http://themistoklis.org/mengthesis.pdf.

Choi, Kwang-Jin. 2001. Online motion retargetting. *The Journal of Visualization and Computer Animation* 61, no. 5 (December): 260-235. doi:10.1002/1099-1778(200012)11:5<223::AID-VIS236>3.0.CO;2-5.

De Aguiar, E., Christian Theobalt, and H.P. Seidel. 2006. Automatic learning of articulated skeletons from 3D marker trajectories. *Advances in Visual Computing*: 485–494. http://www.springerlink.com/index/26760v071m0962m5.pdf.

Gamage, Sahan S Hiniduma Udugama, and Joan Lasenby. 2002. New least squares solutions for estimating the average centre of rotation and the axis of rotation. *Journal of biomechanics* 35, no. 1 (January): 87-93. http://www.ncbi.nlm.nih.gov/pubmed/11747887.

Gleicher, Michael. 1997. Motion editing with spacetime constraints. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, 139. ACM. http://portal.acm.org/citation.cfm?id=253321.

Gleicher, Michael. 1998. Retargetting motion to new characters. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 33-42. ACM. doi:http://doi.acm.org/10.1145/280814.280820. http://portal.acm.org/citation.cfm?id=280814.280820.

Gleicher, Michael, and Peter Litwinowicz. 1998. Constraint-based motion adaptation. *The journal of visualization and computer animation* 9, no. 2: 65–94. http://www3.interscience.wiley.com/journal/5507/abstract.

Hecker, Chris. 2008. Real-time motion retargeting to highly varied user-created morphologies. *ACM Transactions on Graphics* 27, no. 3 (August): 1. doi:10.1145/1360612.1360626.

Herda, L, P Fua, R. Plankers, R Boulic, and D Thalmann. 2000. Skeleton-based motion capture for robust reconstruction of human motion. *ca*: 77. http://www.computer.org/portal/web/csdl/doi/10.1109/CA.2000.889046.

Herda, L, P Fua, R Plänkers, R Boulic, and D Thalmann. 2001. Using skeleton-based tracking to increase the reliability of optical motion capture. *Human movement science* 20, no. 3 (June): 313-41. http://www.ncbi.nlm.nih.gov/pubmed/11517674.

Hornung, Alexander, and Leif Kobbelt. 2004. Robust and Automatic Optical Motion Tracking. *ahornung.net*. Workshop Virtuelle und Erweiterte Realität der GI-Fachgruppe VR/AR. http://www.ahornung.net/files/pub/Hornung_VRAR04.pdf.

Kirk, Adam G., James F. O'Brien, and David A. Forsyth. 2005. Skeletal Parameter Estimation from Optical Motion Capture Data. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR) 2005*, 782–788. June. http://graphics.cs.berkeley.edu/papers/Kirk-SPE-2005-06/.

Kwon, Young-Hoo. 2010. Joint Center: Functional Method. http://www.kwon3d.com/theory/jkinem/jcent.html.

Kwon, Young-Hoo. 2011. Computation of the Rotation Matrix. http://www.kwon3d.com/theory/jkinem/rotmat.html.

Monzani, Jean-S. 2001. Using an Intermediate Skeleton and Inverse Kinematics for Motion Retargeting. *Computer Graphics Forum* 19, no. 3 (December): 11-19. doi:10.1111/1467-8659.00393.

O'Brien, J.F., R.E. Bodenheimer Jr, G.J. Brostow, and J.K. Hodgins. 2000. Automatic joint parameter estimation from magnetic motion capture data. *Proceedings of Graphics Interface 2000*, no. May: 53--60. http://graphics.cs.berkeley.edu/papers/Obrien-AJP-2000-05/.

Poirier, Martin, and E. Paquette. 2009. Rig retargeting for 3D animation. In *Proceedings of Graphics Interface 2009*, 103–110. Canadian Information Processing Society. http://portal.acm.org/citation.cfm?id=1555907.

Ringer, M. 2004. A procedure for automatically estimating model parameters in optical motion capture. *Image and Vision Computing* 22, no. 10 (September): 843-850. doi:10.1016/j.imavis.2004.02.011.

Silaghi, M C, R Plankers, R Boulic, P Fua, and D Thalmann. 1998. Local and global skeleton fitting techniques for optical motion capture. In *Proceedings {CAPTECH}'98*, ed. N Magnenat and D Thalmann, 26-40. Springer.

Wen, G., Zhaoqi Wang, S. Xia, and D. Zhu. 2006. From motion capture data to character animation. In *Proceedings of the ACM symposium on Virtual reality software and technology*, 165–168. ACM. http://portal.acm.org/citation.cfm?id=1180495.1180528.

Xiao, By Zhidong, Hammadi Nait-charif, and Jian J Zhang. 2009. Real time automatic skeleton and motion 20, no. September: 523-531. doi:10.1002/cav.

Zordan, V.B., and N.C. Van Der Horst. 2003. Mapping optical motion capture data to skeletal motion using a physical model. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 245–250. Eurographics Association. http://portal.acm.org/citation.cfm?id=846311.

Appendix A - Integration

Dependencies

There are three dependencies that must be included to compile Kiip, four when using PhaseSpace as a source of marker data. Includes for the external libraries is located in */Kiip/Code/Libraries/*. These are:

- *eigen3*
- *boost_1_45_0*
- *phasespace\include*

The headers for Kiip itself are located in */Kiip/Code/KiipCode/*. For compiled libraries, the folder */Kiip/Code/lib/* should be included.

Headers

`KiipCore.h` and `Kiip.h` are the two headers required to include `Kiip`. `KiipCore.h` stores definitions for the data structures that are used to pass data between `Kiip` and an application, as well as enumerations used to specify what limb/joint/etc is being used in a given function call. `Kiip.h` stores the definition for the Kiip class, which defines the public API available to applications.

`PhasespaceReader.h` is needed when using PhaseSpace as a marker source.

Defining Markers

On construction, the application must pass an array of pointers to `MarkerInfo` structures, defined in `KiipCore.h`. Each struct stores a `MarkerLocation`, which defines on which limb the marker resides. The order of the structs in the array defines the index numbers used to refer to the markers in the future. The application owns the array, and is responsible for deallocating it when no longer in use. This deallocation can occur anytime after the `Kiip` constructor has returned.

The array of structs is passed to the `MarkerReader` subclass currently in use. The `MarkerInfo` struct accepts a value of type long on construction, which acts as a type identifier for the struct. Subclasses of `MarkerInfo` can be defined and used, allowing for extra information to be passed to the reader. For instance, the `PhasespaceReader` defines a `PhasespaceMarkerInfo` which includes an extra field for the PhaseSpace ID value of the marker, allowing it to differ from the value `Kiip` assigns it. The `MarkerReader` subclass can determine if a given `MarkerInfo` object is of the proper type by calling `getInfoType()` on it.

Constructing Kiip

The constructor for `Kiip` takes in 3 values: a pointer to a `MarkerReader` implementation, a pointer to the array of `MarkerInfo` pointers, and the number of elements in that array (the number of markers being tracked). Once passed to the constructor, `Kiip` owns the `MarkerReader` and will handle its destruction at the appropriate time.

Input Joints

Only joints that are activated by the application are eligible for retargeting. Joints are activated by calling `addSkeletonJoint()` on the `Kiip` object. This function takes in a `SkeletonJointLoc` defining which joint to activate, and a 4x4 array of floats representing the column-vector transformation matrix of the joint in its base pose.

Initialization

A series of steps must be performed to allow `Kiip` to begin processing of incoming frames, including finding the base pose, capturing the gym motion, and finally processing that gym motion.

Base Pose

Initially, `Kiip` has no base pose. To tell `Kiip` to begin searching for one, the `findBasePose()` function must be called. This will cause `Kiip` to begin searching each new frame of marker data until it finds one in which all markers are visible. While searching, calls to `getCurrentTask()` will return `kKiipFindBasePoseTask`. Once the pose is found, `hasBasePose()` will return true. Calling `findBasePose()` and passing in a value of false will cause `Kiip` to stop looking for a base pose.

Gym Capture

A call to `captureGymMotion()` will cause `Kiip` to begin storing frames of data in a list. The first argument is a long representing the number of seconds that the capture should last. The second argument is an integer representing the number of frames to capture each second. For instance, the arguments (2, 10) will result in 20 frames being captured over 2 seconds. While capturing, a call to `getCurrentTask()` will return `kKiipCaptureGymMotionTask`, and `kKiipNoTask` when the capture has completed. Calls to `getCaptureLength()` will return the number of frames that have been captured thus far, or -1 if the capture has not been started.

Gym Processing

Once the motion has been captured, it must be processed by calling `processGymMotion()`. While this is taking place, calls to `getCurrentTask()` will return `kKiipProcessGymMotionTask`. Once processing is complete, calls to `isGymMotionProcessed()` will return true, and `Kiip`'s task will automatically switch to `kKiipProcessMotionTask`. From this point on, `Kiip` will process incoming marker data and store the calculated joint and limb information.

Accessing Data

`Kiip` will run an internal loop, storing new marker data, passing it between the various subsystems, and updating flags to indicate this new data. There are four flags, one for each type of data that can be retrieved. Retrieving the corresponding data will reset the flag to false until new data is available:

- `hasNewMarkers()` indicates if new marker data is available. Call `getMarkers()` to retrieve them, passing in a pointer to an array of `uMarker` structs, and the number of markers to get
- `hasNewLimbs()` indicates if new limb data is available. Call `getLimbs()`, passing in a pointer to `kActorLimbCount uLimb` structs to be filled.

- `hasNewActorJoints()` indicates that new joint position data for the actor is available. Call `getActorJoint()`, passing a pointer to `kActorJointCount uActorJoint` structs.
- `hasNewSkeletonJoints()` indicates that new joint transform data for the retargeted skeleton is available. Call `getSkeletonJoints()`, passing a pointer to `kNumSkeletonJoints uSkeletonJoint` structs.

Restarting

A call to `getKiipStatus()` should always return `kKiipOk`. If it does not, then `Kiip` is not running correctly and should be restarted by deleting the existing `Kiip` object, and repeating all of the previous steps. The same `MarkerInfo` array can be reused, if not already freed. In practice, the only time the status changes is when the `PhasespaceReader` has a server error. This could come from one of two places. First, if the reader is unable to connect to the PhaseSpace server for some reason, it will fail. More often, though, is the case where a pre-recorded capture session is being streamed in place of live data. When the recorded data ends, the server stops streaming and sends an error that causes `Kiip` to stop running. In general, when not using pre-recorded data, `Kiip` should never stop running until deleted.