Roborace: Postmortem

Jacob Kuenzel, Devon Penney, Adhitya Chittur

16th May 2006

Project Vitals	
Developers	3
Length of Development	8 Weeks
Platforms	Windows
Hardware Used	
Software Used	
Notable Technologies	Motion Capture, Inverse Kinematics, Physical Simulation
Size	2,916 lines of code written for OPAL, 164 lines of Matlab code, 6,778 lines of standalone code written for Dojo, and a 332 line increase in the size of Dojo sources from modifications.

1 Original Goals

Our initial goal was to build a platform on which to research a number of new schemes for controlling 3D articulated bodies in the context of a physical simulation. Upon completion, we planned to use what we learned from the research to create a simple game that was both fun to play and appropriate for assessing the effectiveness of the various control schemes.

We chose four different articulated bodies which are loosely based on common robot designs and three different categories of control schemes with which to control these bodies. There was a four-legged hopper, a four-legged crawler, a three-wheeled robot, and a humanoid bipedal robot. The three categories of control schemes were Direct Control, Segmented Control, and Dimension Reduction-Based Control.

In a Direct Control Scheme, input from the user is directly connected with the position or angle to which joints are servoed. For example, a direct control scheme for a bipedal body might consist of setting the distance of one foot from the floor to be equal to the mouse's y-axis position.

In a Segmented Control Scheme, the user's screen space is split into a number of different segments, each of which corresponds to a different part of the articulated body. When the mouse enters are particular segment, its corresponding body part is activated in a certain way. For example, a segmented control scheme for a crawler robot might consist of splitting the screen in to quadrants, and activating a segment would cause the corresponding leg to take a step forward.

In a Dimension Reduction-Based Control Scheme, the idea is to treat the set of all possible poses that the body can assume as a vector space, and to then allow the user to control where in the space the body takes its current position from. For example, by applying Principle Component Analysis to a high dimensional set of motion capture data, a set of basis vectors for a space with far fewer dimensions is obtained. The user's input is then used to create linear combinations of these basis vectors and to obtain new poses.

2 Development Team

All three members of the team have known each other for the majority of their time at Brown, and with the exception of Devon and Adhi, have worked with one another on projects in past CS classes. When not slaving away on CS projects, the trio has been known to get together for a drink or ten.

Jacob and Adhi have known each other since freshmen year when they took CS17/18. They have worked on two other CS projects together: an autonomous trading agent which uses statistical models in CS141 and a sonar mapping robot in CS148. Jacob and Adhi are currently housemates.

Adhi and Jacob have known Devon since sophomore year when they took Abstract Algebra (MA153). Jacob and Devon have worked on one other CS project together: an extension to ray marching used to render physically realistic rainbows in photon mapped participating media.

3 Tools Used

3.1 ODE

Right from the start, it was clear that the project would require a fast, high-quality physics simulation to achieve the level of realism desired with the complex bodies being used. The Open Dynamics Engine (ODE) was frequently mentioned in papers describing previous research related to the project.

ODE is an open source library for simulating rigid body physics that has been under active development since 2001. It has a mature, full featured API and contains relatively few bugs. Its wide variety of joint types and integrated collision detection with friction made it an ideal library to use for the project's physics.

It was decided early on that whatever physics library was used would have to at least have feature parity with ODE. As it turns out, the platforms that were chosen were based on ODE, so this requirement was easily satisfied.

3.2 OPAL

A significant amount of time at the beginning of the project was devoted to finding a platform appropriate for the project. As all three group members have far more experience with C++ development under Linux than Windows, the ability of a platform to run on multiple operating systems was a major consideration. The platforms that were tested include Torque, Yake, ODEJava, OgreODE, and Open Physical Abstraction Layer (OPAL) with Ogre. OPAL was finally chosen as it appeared to have the most complete and functional API.

OPAL is a high-level open source C++ library for interfacing low-level physics engines with applications such as games and simulations. One of the most attractive features of OPAL is its concept of "blueprints"– XML files that describe every aspect of a physical system. We thought that these files sounded like a great way to specify robots. Although it supports the use of multiple physics libraries, OPAL was initially written as a wrapper for ODE. Because of this, it has a full object-oriented implementation of all ODE features that closely resemble the ODE API. In addition, code is available on the OPAL website that allows for (relatively) easy integration with the Ogre graphics engine.

After 2 weeks of development and nearly 2600 lines of code, OPAL was scrapped due to problems which will be discussed in section 5.

3.3 Dojo

Dojo was not initially considered as a development platform, both because of its platform-dependence and because of bad past experiences with G3D. At the urging of the course staff, however, and with no other immediately available options, we made the decision to use Dojo as the new platform once OPAL was scrapped. Usage of Dojo is also discussed in section 5.

3.4 Mocap

The dimension reduction control techniques require the use of motion capture data. The two most common motion capture formats are BVH and ASF. During our development with OPAL, we wrote an ASF file loader due to the availability of ASF files on mocap.cs.cmu.edu. However, this was scrapped once we moved over to Dojo. Also, we discovered that BVH files store the motion data in a matrix form where rows are character poses. This made computing principle component analysis on the frame data simple since PCA

requires data in a matrix form. Thus, we used BVH files provided by the course staff for the remainder of the project.

4 Things That Went Well

4.1 PCA Development

4.2 Crawler

The Crawler character was successful in a number ways–it ended up being the easiest character to control with the most working control schemes and also acted as catalyst for the addition of code that became more widely useful during the project.

After an attempt at defining a hopper robot using an ASF file proved the format to be insufficient for our needs (we needed body parts with shapes other than the capsule shape used for bones), it was decided that other robot characters should have their structure hard-coded.

Dojo provided a way to easily add ASF characters (after modification) and Quake 2 models to the world, but it did not provide a simple or straightforward way to build arbitrary articulated bodies. The only references we had to work with were the code that builds the articulated body from an ASF skeleton and a couple hard coded stick figures. The stick figures were particularly unhelpful as each piece of geometry is constructed over a number of calls which involve CoordinateFrames that, without documentation, made little sense while reverse engineering.

This was dealt with by abstracting geometry-creation code to methods in the Character class. By hiding the confusing underlying calls necessary to create a piece of geometry, we were able to reduce redundant and hard-to-read code and actually get to creating the character.

Once the geometry for the Crawler was created, a CrawlerController class was created for all control scheme implementations for the crawler to inherit from. This class contains references to all of the Crawler's servos, the definition and four instances of an inner LegController class for abstracting the task of positioning legs, and a large number of methods used to perform actions such as stepping, pushing, and kicking with these LegController instances.

In order to perform complex actions such as stepping, it is insufficient to simply servo a limb to any one pose - instead, a number of poses must be assumed with a small delay in between each to allow for the servos to converge so as to have better control over the limb's trajectory. Two possible ways of achieve this were considered. One possibility was to detect when the servos had ceased movement above a certain threshold before trying to assume the next pose. The other possibility was to simply wait a fixed amount of time before assuming the next pose. Because there might be situations in which the servo would take a very long time to stop moving, causing unresponsiveness in the controls, the fixed time delay solution was used.

Unlike conventional robot programming, in which one can achieve a delay simply by causing the control program to sleep for a moment, a physically simulated robot's physics do not run concurrently with its control program. In order to circumvent this, a scheduling system was implemented in the CharacterController class. The system works by keeping a queue of scheduled tasks and a corresponding list of delays. By decrementing the delay times by the size of the last simulation step, the system determines when the approximate delay time for a particular task has passed in the simulation, at which point it removes the task from the queue and executes it.

This scheduling system, coupled with the abstraction layers, provided a very useful base on which to implement controllers for the crawler. Writing the Direct and Segmented controllers was basically as simple as detecting when a key was pressed or a segment was activated and then calling the appropriate method from CrawlerController–significantly simpler than our experience with some of the other characters. The automated controller also involved minimal additional code, requiring little more than a simple layer of abstraction to allow stepping motion loops to be started and stopped.

Because it was completed in a relatively short amount of time, the crawler's controls received more attention and were able to be tweaked more, resulting in the most usable of the robots.

4.3 ODE

ODE proved to be one of the easier to use portions of our project. While there were bugs present that we mention through this paper, overall we had a good experience, primarily due to the extensive, accurate, and well-defined nature of the documentation. The ODE user guide was of special importance to the project. Additionally, the API to ODE was consistently structured, making debugging physics errors considerably easier for those of us without extensive experience working with it.

4.4 Matlab

Developing the PCA control schemes went smoothly. First, Matlab provided an excellent tool for the numerical algorithms involved with applying dimension reduction to motion capture data. It was simple to write BVH loaders to extract frame, compute PCA, and save the basis files. Integration with C++ was a matter of simply parsing data generated from matlab. Next, generating and servoing to poses was simple and easy to implement using mouse coordinates as coefficients for basis vectors lead to a plausible reconstruction of the original motion space.

5 Things That Went Wrong

5.1 OPAL

One of the first hurdles with using OPAL was finding a Linux computer for our third group member to use. We decided most of our development would not be taking place in the CIT, since collaboration at our places of residence proved much easier. A significant portion of time was devoted to installing Debian Linux onto a third computer, as some driver issues plagued the process.

Next, we wrote an ASF and AMC parser to use motion capture data within OPAL. This was not particularly difficult code to write, but it was time consuming and took away from more exotic and interesting areas of the project. Lastly, there were many of issues with our humanoid character simply exploding due to bone intersections and joint gains being set improperly. This was eventually fixed shortly after we decided to switch to using Dojo.

5.2 Dojo

We had to modify the Dojo game engine significantly to suit our needs.

Our first need was for an arbitrary ASF parser. Dojo included one, but it was specialized to the Humanoid character. We chose to refactor this parser into an ASFCharacter and generalize it, as this made much more sense than rewriting the parser for every arbitrary ASF character we wished to create.

Our second need was for sliding joints so that we could implement the pogo-stick hopper. This entailed some extensive modifications to both the Entity code to support ODE sliding joints, as well as reverse engineering and modifying the ASF parser to support sliding joints. The parser code was unfortunately not abstracted to a very high level, possibly due to the limitations of the ASF specification, and thus the the process of finding all the appropriate places to add support and subsequent debugging took much time away from actual game development.

The third need we had was for servo'ed joints. To accomplish this, we modified the Entity class to include the Servo and SlidingServo nested classes. Unfortunately, we had to track down a significant bug in the ODE physics engine: When joints went completely stationary, they ended up locking in place, and it became impossible to move them again without some artificial secondary force. The tracking took quite a bit of time, and the solution, to always keep some minimum force applied to each joint, had negative effects on some controllers.

One problem we experienced with Dojo was that some of the code we were introduced to in the assignments was not present in the CVS version, and when trying to import this code, we found portions of it were incompatible with the Dojo version we had been working with. One example of this was the Trigger code: the actual function calls generated compile errors, and once those were resolved, the semantics of the trigger physics were causing ODE to crash.

Another problem we experienced with using Dojo was improper usage of the data structures used by G3D. On multiple occasions, we found that the way Dojo was iterating through lists, by using the overloaded [] operator and iterating as if it were a regular array, was actually causing crashes in Entity and the ASF parser after we made our modifications to the base Dojo code. This is not the recommended way to iterate through abstracted lists with STL-style iterators - in fact, changing this code to use the G3D ConstIterator fixed several strange bugs, though why this was the case we do not know.

A third problem experienced was with the Dojo documentation. The API for Dojo changed between assignments and differed both from the CVS version and the documentation present online. Thus, when we tried to study the Dojo documentation and assignment documentation, we were often met with much frustration at trying to match up specification with implementation. We were told to reverse engineer these problems, and this caused much delay in the development process.

5.3 Development Environment

The switch to using Dojo necessitated that we switch our development environment from Eclipse on Linux to Microsoft Visual Studio on Windows. This presented several challenges to the group. First, only one group member has had extensive experience working with Visual Studio in the Windows environment, whereas all three have had extensive experience working with Eclipse in a Linux environment. This presented a major issue, as only one group member, instead of three, had the IT and industrial development experience to fix the myriad of eccentric problems present in Visual Studio and Windows which were experienced during development.

The first major problem we encountered was in setting up our source control system. We chose CVS on the CS filesystem for this, as it was the only option available at the time of the switch. Our first problem was the lengthy process of setting up a Windows CVS client to work remotely. Several options were explored, including command-line CVS, WinCVS, and TortoiseCVS. TortoiseCVS was settled on, as it was the first one we were able to successfully SSH into the department and connect to the CVS server with. However, as the documentation on using CVS remotely for the department was out of date and lacking in detail, it took an entire day's worth of development (about 8-10 man-hours) to simply figure out the correct connection settings to use department facilities remotely - and our group is not exactly inexperienced with either the CS systems nor their remote access capabilities.

The second major problem we encountered with CVS was that because the cvs init had (unavoidably) been done via command line on the Linux machines, the correct differentiation between binary and text file formats was not set. Windows executables and libraries were all set in text mode, which we could not notice until we actually started trying to build our code, after we had started basic development. We had two choices: either hose our current CVS repository and start again from scratch, or try to fix the problem in-flight. We attempted the latter solution, as we estimated fewer man-hours would be lost; finding the solution, an obscure setting in Tortoise to use Unix line endings, took away approximately another day of development.

It should be noted that for real development work on Visual Studio, CVS is not the version control system of choice. Microsoft Visual SourceSafe support is integrated with Visual Studio, and either a Visual SourceSafe server or one of its compatible alternatives, such as SourceGear Vault, would have virtually eliminated the problems we experienced. Development on an end-to-end Windows platform would have saved much time and frustration.

Unfamiliarity with the intricacies of Visual Studio also proved a problem for development for some group members - while from a user interface perspective Visual Studio and Eclipse are very similar, how they operate under the hood is quite different. The functionality of project files, solution files, and IntelliSense compilation files will not be discussed in detail here, but they posed problems with regards to the version control system. The end solution was that each group member ended up maintaining their own set of configuration files, to be manually edited whenever anything in the project changed. Figuring out that some of our build problems were due to this wasted another day of development, but the end solution was luckily not too time-consuming.

Working with Windows also proved a major problem. Please note that one group member was recently an experienced information technology consultant with a specialty with these specific problems, yet we still ran into the security problems endemic with Windows, namely the uncanny ability of adware/spyware to

infect supposedly secured machines. One group member's computer had become infected with spyware near the end of the development process so badly that it necessitated a full format and reinstall of Windows. The other group member's machines were infected by this offender since we were all on a local network, but luckily not quite to the extent of a necessary format. Approximately another day's worth of man-hours was lost to fixing these problems.

In sum, we lost approximately a week of development time due to environment problems which we would not have experienced on another platform.

5.4 Implementing IK

A significant amount of time was put into implementing two different and complementary IK algorithms. While these algorithms worked well, they ended up not being incorporated into the project and therefore represent one of the largest mistakes made during development.

After some initial experimentation with direct activation of the humanoid character using simple mappings between mouse axes and joint angles, we determined that if a user had any hope of balancing the humanoid they would need a more intuitive way of controlling the positioning of the legs and the feet. IK naturally came to mind as an intuitive way of positioning a limb in 3D, so we set out to implement it. Initially only the Jacobian Transpose method was to be implemented, but after reading about the difficulties it has with singularities we decided to also implement Cyclic Coordinate Descent, a complementary method that does not exhibit such problems. Our plan was to test both methods and use whichever one felt better.

Implementation of the algorithms took approximately a week, and a significant portion of another week was spent experimenting with different ways of using the IK, the most successful of which are described in section 6.2.4. There were still problems balancing the character, and two main solutions were proposed, neither of which was actually put in motion.

The first solution was to use two mice instead of one, a solution without a problem that we had envisioned at the beginning of the project. We found that it was fairly easy to get one leg to go where we wanted using an IK target controlled by the mouse, but that controlling the other leg at the same time was unreasonably difficult. We surmised that by adding a second mouse-controlled IK target, the user would have a control scheme that maps much more readily to their body, allowing for much more intuitive control. The main problem with this solution was that using multiple mice on Windows requires using a raw device driver and manually decoding mouse protocol packets, something we did not have the time to learn.

The second solution was to take advantage of the redundancy of our IK chains and use one of the existing Jacobian null-space approaches in the IK solver to also optimize character balance. We began to wonder if all the work we were putting in to IK was really worth it. When we were told after an in-class presentation that IK has already been done and that we should move on to something else, we were more than happy to oblige. In retrospect, however, we feel that what we were trying to do with IK had some novel aspects to it and that had we chosen to put more time into it we could have ended up with an interesting and useful control scheme. Unfortunately we did not feel it was worth the risk at the time, making IK one of the biggest time-wasters of this project.

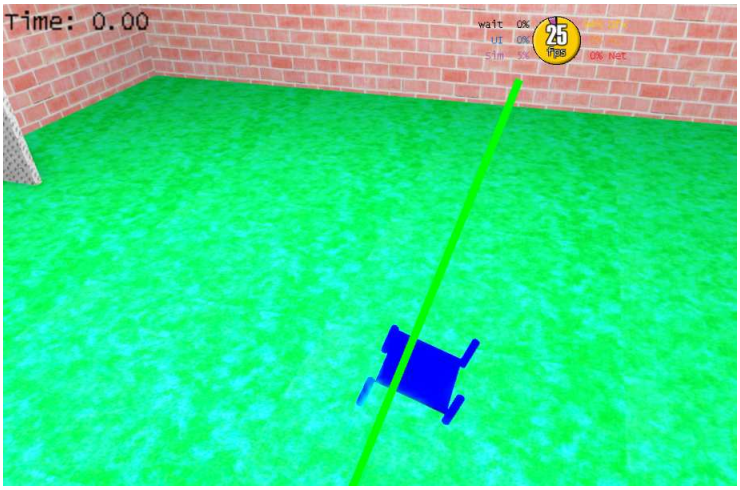
5.5 Control Schemes That Didn't Work Well

One of the biggest downfalls and disappointing aspects of this project resided in the fact that none of the control schemes worked very well at all. While in most cases it was possible to obtain some level of locomotion with our control schemes, they were not robust enough to work in a game setting.

A major shortcoming of our project was the soccer game. None of the implemented control schemes were able to play soccer well, primarily for the reason that physically accurate kicking of the ball is a very difficult task. To send the ball on an appropriate trajectory, there is a very small range of points on the ball that can be collided with and a very small range of direction vectors for the force. Real soccer games, such as FIFA, do not use such a model for ball physics - they instead use precanned motion capture movements and artificial force vectors to send the ball in the intended direction from a controlled kick. Additionally, controlling the physical act of kicking for a humanoid character is particularly difficult due to the many degrees of freedom for the character to change posture in the pre-kick movements. The fine grained control our characters provided for self-movement did not translate well into manipulation of other physical objects.

6 Research Results

6.1 Hopper



6.1.1 Direct

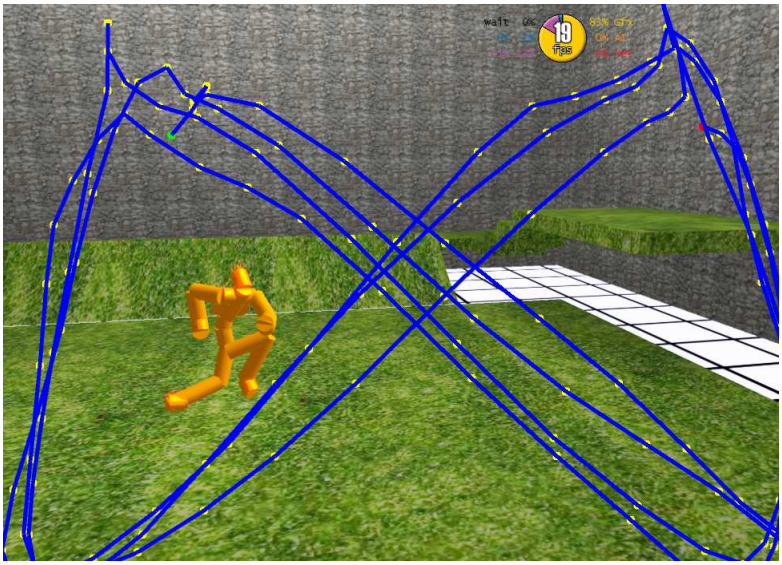
The direct hopper controller was based solely around keyboard input. The keys were broken into two quartets, the inner ring of {H,J,B,N} and the outer ring of {G,K,V,M}, the keys controlling each pogo leg of the hopper respectively. The inner ring actuated the prismatic joints in the "upward" (or original resting) position, while the outer ring actuated the joints in the opposite direction.

This controller was fairly difficult to use. The primary problem was with an ODE simulation bug, which prevented the movement of joints once the character came to stand in a static position. This required the hack of always maintaining some force on each of these joints. Additionally, another ODE bug was present for slider joints: the API call to set the joint velocity was faulty (it had no effect), so the only way to actuate the joint was to apply an impulse force, which is the time derivative of the velocity and hence provides less precise control over the joint's position. These two hacks made it nearly impossible to have any useful actuation or locomotion from the character.

6.1.2 Segmented

The segmented hopper controller was based partly on keyboard input and partly on mouse input. The controller operated by reading in the coordinates of the mouse position, calculating the scaled Cartesian distance from each corner of the game window, and then actuating the hopper joints based on these four distance scalars. Each corner of the screen space represented the zero point of its respective joint. The G and H keys on the keyboard were used to actually actuate the joints in the up or down direction. This controller suffered from the same usability issues as the Direct Hopper Controller, for the same reasons as stated above.

6.2 PCA Humanoid



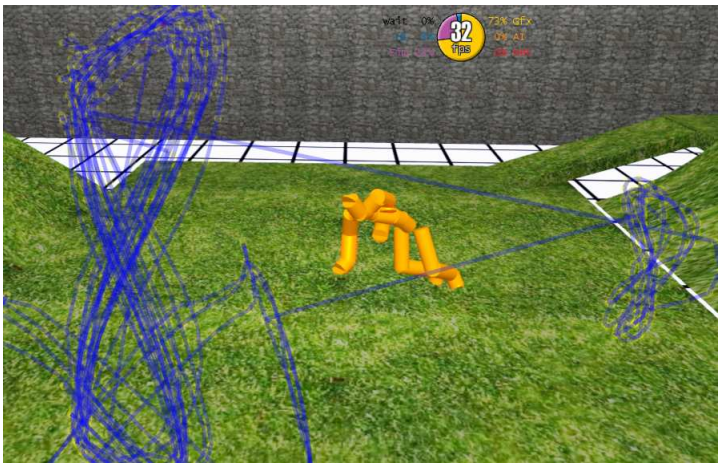
6.2.1 Introduction

PCA has been used extensively for years in many areas of computer science for things such as computer vision. However, there has been little to no research on using it for a user control interface for humanoid characters. We had mixed results from our work, but it showed that this is an area ripe for innovation. Please visit our website for videos pertaining to the information below.

6.2.2 Structure of Motion

All motion capture sequences have a very unique structure when projected onto a 2D plane. In the case with a human walk, the sequence appears as a figure eight, which exemplifies the cyclic nature of the human gait. This has a fairly intuitive control structure associated with it. By moving the mouse along the figure 8, it is possible to obtain a walking motion. However, it was often times difficult to follow the figure eight path with the mouse, since any deviation merely cause the character to make spastic and unpredictable movements. With motions other than a walk, the path in 2D space was often indescernable. With these types of motions, it was clear that a logical user control scheme is essentially impossible to obtain with the current algorithms.

6.2.3 Embedding Multiple Motions



One of the interesting applications of the PCA based control schemes was embedding multiple motions in the screen space plane. For example, we embedded a walk and a crawl sequence. PCA inherently produces embeddings for the two motions in different locations in the plane. This allows the user to switch between the different motions by merely moving the mouse from the space of one to another, hence transitions between motions are handled by the structure of the embedded space. While this worked to some degree, it did not produce a usable control scheme as the character would jump around between the two spaces in an illogical way. However, it is interesting to note that the user was able to transition between different motion sequences by merely moving the mouse from the embedding of one to another.

6.2.4 Servoing Issues

Some of the problems that rendered PCA based control systems unusable are related to the servoing of the humanoid character. First of all, the gains on the joints would need significant tweaking since. This is because the motions recovered through PCA often produced harsh transitions between frames, which resulted in exaggeration of the movement. Next, it was difficult to change the orientation of the humanoid character outside of the motion capture replay. This meant that it was difficult to develop a generalized control scheme that allowed for a full range of motion rather than simply a forward walk.

6.2.5 Beyond PCA

PCA is only one dimension reduction technique. There are many other methods that can have significant changes on our algorithms for dimension reduction based control. Isomaps is an interesting area to explore, which was obmitted from our project for several reasons. First, we focused more on developing a wide range of techniques such as IK based schemes. Second, isomaps would not change the general idea of the control scheme, but merely change the topology of the screen space projection. However, isomaps are often used to capture more salient information about datasets since it preserves n dimensional geodesic distance from the original dataset. However, even if isomaps were used, it would use the same control scheme of exploring the embedded space, which would result in unpredictable results when venturing out of the motion path.

6.2.6 Conclusions

Simply following the path of motion data in a 2 dimensional space is essentially equivalent to playing back a motion capture sequence at a lower resolution. Also, when one deviates from this path, it results in unpredicable poses, which translates into an unintuitive control scheme. That being said, this is still an area that is ripe for innovation. Embedding multiple motions in the plane has potential to be useful for cases where a control scheme requires moving from one action to another quickly. Also, other dimension reduction techniques could provide interesting insight into the problem, but this was out of the scope of our project. The best avenue for innovation is combining an IK based control system with a probabilistic search over a reduced dimensional space of all poses. Such a system would be very similar to StyleIK, except used in a physics environment. However, as we found, any humanoid based control scheme requires huge amounts of tweaking for joint gains, gravity, levitation, balancing, etc. This is a huge stumbling block for development, which leads to unpredictable and unrealistic goals. However, the use of dimension reduction for controlling physics based characters is an area of character animation of human computer interfaces that has several distinct and interesting avenues for research.

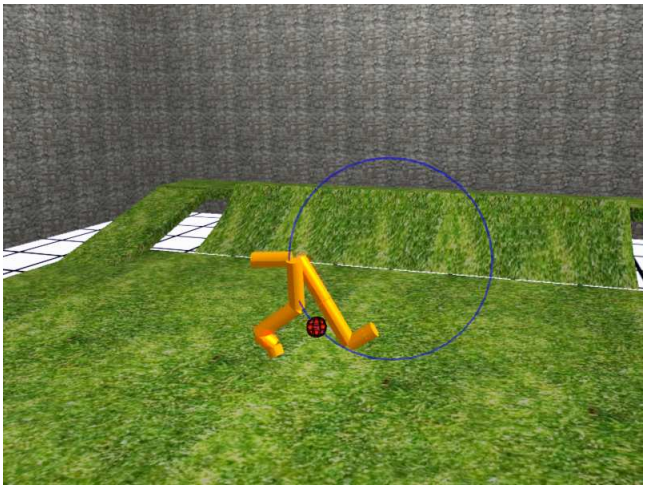
6.3 Other Humanoid Control Schemes

6.3.1 Segmented

The segmented humanoid controller was based partly on keyboard input and partly on mouse input. The mouse space was segmented into two equal areas divided by the vertical center of the game window, the left and right sides bound to their respective legs. The Cartesian mouse position within each segment was calculated with respect to the center of the segment and used to actuate the hip and knee joints accordingly when the space bar on the keyboard was hit. The G and J keys straightened the left and right knee joints, respectively. This method of control proved useful for walking in a straight line, both forwards and backwards, especially with the cheat addition of the Y and H keys for impulse forces forwards and backwards.

The major fault of this controller is that physically accurate turning was very difficult. The problem was twofold: the friction calculations used in ODE made it nearly impossible for a foot touching the ground to rotate and the joint constraints for the upper body kept servoing the hips back to their original position, instead of the hips causing the body to turn. Also, the actual physical act of a humanoid turning is a very complex motion, and no scheme was found to discretize these motions in such a way as to provide for useful turning capability. This controller proved useful for the straight-line race, with race times as good as the crawler character, but was entirely deficient for playing soccer.

6.3.2 Direct control with IK



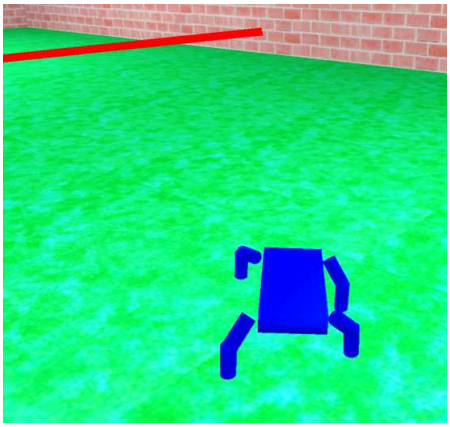
As described in section 5.4, we decided to implement IK for the humanoid after a little bit of experimentation. We figured that by making the humanoid's feet end effectors and finding an intelligent way to control the effector targets, we could ease the task of achieving a walk. We came up with a couple different ways of controlling the effector targets.

First we tried simply moving the target around in world space using the mouse and keyboard. We were able to get some walking type movement from the character, but as the character moved further away from the world's origin it became harder to keep the target very close to the character.

The next thing we tried was moving the target around in the coordinate frame of the character's feet and legs. This kept the target close to the character, but it was still difficult to control the height of the foot from the ground. Also, there were stability issues resulting from the fact that the position of the target would effect the coordinate frame of the foot, which would in turn effect the position of the target, ad infinitum.

Finally we tried defining a circular trajectory for the target to follow. By moving the mouse, the user could specify where on the circle the target should be. The circular shape allowed much better control of the height of the feet. The trajectory was put in the coordinate frame of the character's hip which minimized the influence of the effector target and got rid of the stability problems. Unfortunately there were still balance issues with this scheme that ultimately led to it being abandoned.

6.4 Crawler



6.4.1 Direct

The direct crawler controller has a steep learning curve, but is one of the most effective controllers in our demo once mastered. The keys J,K,N, and M are layed out in a square in the keyboard wich corresponds to the layout of the crawler's legs. When pressed, each of these keys causes the corresponding leg to step forward by lifting up, rotating forward, and then lowering back to the ground. When released, the corresponding leg is then rotated backward, causing it to push against the ground. The keys to the immediate left and right of this square, H, L, B, and < are used to step backward instead of forward, and the I and O keys are used to cause the front left and right legs, respectively, to kick out in front of the robot.

There are two ways to achieve forward or backward motion with this control scheme. The first resembles a four-legged animal trotting–the front right and back left legs are in sync, and the front left and back right legs are in sync. By timing the steps of these pairs of legs correctly, the character can be made to walk in a straight line. The second way resembles a gallop–the front legs are in sync and the back legs are in sync. When the front legs push backward, the back legs are stepping forward and vice versa. This way is much more difficult to time correctly and is also much more difficult to control, but appears to have the potential to achieve higher speeds.

The character can also be turned two different ways. The first involves simply altering the speed with which each side of the character steps forward. If one side steps forward more slowly than the other, the character will turn toward that side. This method is extremely difficult to use, however, and produces unpredictable turns. The second way uses the forward step buttons on one side and the backward step buttons on the other side. This results in a quick turn about the center of the character and is fairly accurate and easy to control.

Overall, the direct crawler controller is interesting to play with, but doesn't seem to have much of a place in video games. It takes some practice to learn, and is tedious to use even for those experienced with it. If it were possible to define some sort of "macro" in which a series of keystrokes were represented by a single keystroke, it might improve the controller significantly. Users would still have full control over the character's motions, but wouldn't be forced to result to incredibly tedious button mashing.

6.4.2 Segmented

The segmented controller is easy to learn and use. The screen is split into four segments and has a neutral circle in the middle. The segments correspond to the crawler's legs and activate the forward stepping motion whenever the mouse is detected within the segment. The neutral circle exists to allow the user to avoid activating a limb when necessary. A walking motion can be achieved using the same idea as a trot with the direct controller–by synchronizing the motion of diagonally separated legs. In the segmented controller this is achieved by making figure-eights whose center is inside the neutral circle. The walk achieved by this is just as effective as the trot in the direct controller, and arguably easier. Most users have noticed some wrist fatigue after using it for more than a few minutes, however.

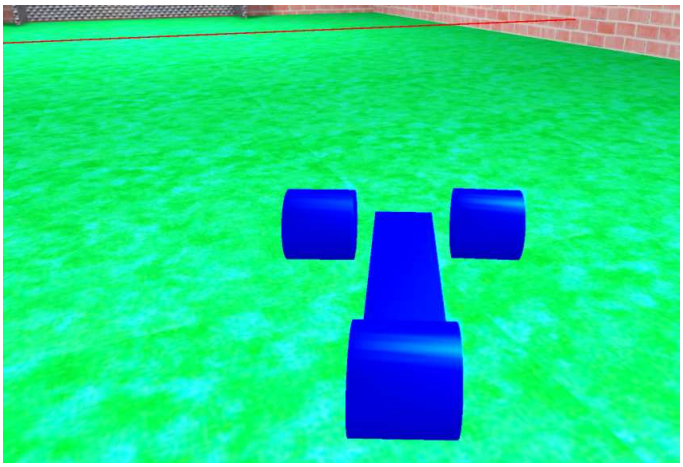
Incidentally turning also works in the segmented controller. By making circular motions with the mouse that pass through all the segments in sequence, the character can be made to turn in the direction opposite to the circular motions. Intuitively this shouldn't work, but it does for some reason.

The segmented controller is fairly rudimentary in our demo, but we feel that of all the control schemes we experimented with it has the most potential. It is interesting to note that the trajectories of walking sequences of motion capture data visualized with the PCA controller closely mirror the trajectory of the mouse across the screen necessary to achieve a walk. The segmented controller could be extended to have multiple bands of segments at different radii and could use the mouse's position and/or speed in the segment to modulate the action being performed, allowing more complicated motions.

6.4.3 Auto

While not completely relevant to our project, it is worth noting that a completely automated controller was implemented. The controller has a target position attribute and it is constantly attempting to move the crawler to this position. Because turning on the crawler is fairly inaccurate, the controller has two cases. If the angle between its heading and the heading to the target is greater than 45 degrees the controller turns the robot about its center to try to reduce this angle. When the angle is less than 45 degrees, the controller walks in a straight line, forward or backward, toward the target. Although it is extremely simple, the automated controller is surprisingly good at getting the crawler to where it's supposed to be.

6.5 Wheeled



6.5.1 Direct

The wheeled robot was created in the hopes of an infallible and easy to use control scheme. The basic idea was to control the rotation of the wheels independently so turns, forward and backward motion could be created. However, with user control, there were problems with guiding the robot on an accurate trajectory. This is because of problems with modeling the friction between the wheels and the ground. The result was the robot would make erratic and unpredictable turns. This being said, it was still possible to guide the wheeled robot to desired locations, albeit with a certain amount of frustration.

6.5.2 Auto

Automated control for the wheeled robot provided interesting results. The robot was coded to reach a specified goal point. This was done by calculating the angle and distance between the current and goal location. This method for automated control compensates for the fricion problems, and the robot is quite efficient at obtaining its goal location.

6.6 User Testing

6.6.1 Introduction

We did a user study on the three most effective and well-developed control schemes. The goal of this study is to determine the effectiveness of the control schemes by studying user behavior. We used a race setting to test the crawler with both direct and segmented control and the wheeled robot. The race involved guiding a character from a starting line to the end. There were no obstacles in the race course, as it is intended to test forward motion exclusively.

6.6.2 Setup

There were six college student participants in the study. A standard desktop machine with a keyboard and mouse was used in the study. Three of them were the creators of the project, and the other three were friends unfamiliar with the project. Each person was given a training period where they experimented with the control schemes. This was to ensure they were able to control the character with proficiency. Each person used the control setups in a race application where we recorded three consecutive race times after the afformentioned training period.

6.6.3 Results

Direct Control Crawler

Crawler Direct	Race 1	Race 2	Race 3	Average
Participant 1	54.27	50.11	51.21	51.8633
Participant 2	38.44	34.12	32.15	34.9033
Participant 3	102.55	99.23	106.43	102.737
Participant 4	70.32	72.31	68.53	70.3867
Participant 5	120.32	123.87	123.12	122.437
Participant 6	95.21	96.87	96.21	96.0967
Average				79.7372

The participants took a while to master this control scheme, but found it fairly intuitive for controlling the crawler. They found the pattern easy to master, but hard to get the crawler going fast. There was a huge amount of variance in the data due to several factors. One participant (namely the one who created this control scheme) was particularly adept at using it. Also, a few people never could quite get the hang of it and were forced to go very slowly, thus never developing a good rhythm to control the gait.

Segmented Control Crawler

Crawler Segmented	Race 1	Race 2	Race 3	Average
Participant 1	78.2	79.55	78.87	78.8733
Participant 2	79.84	80.12	78.03	79.33
Participant 3	99.31	102.33	100.47	100.703
Participant 4	82.43	82.83	85.31	83.5233
Participant 5	128.31	127.44	130.58	128.777
Participant 6	99.21	99.57	100.45	99.7433
Average				95.1583

Participants found this to be the easiest to learn, but the most tiring to use. The excess of mouse movement made it difficult to keep up a good pace for the entire race. Hence, it turned into a mouse hand endurance task more than anything. It is interesting to note that people discovered you could turn the robot with

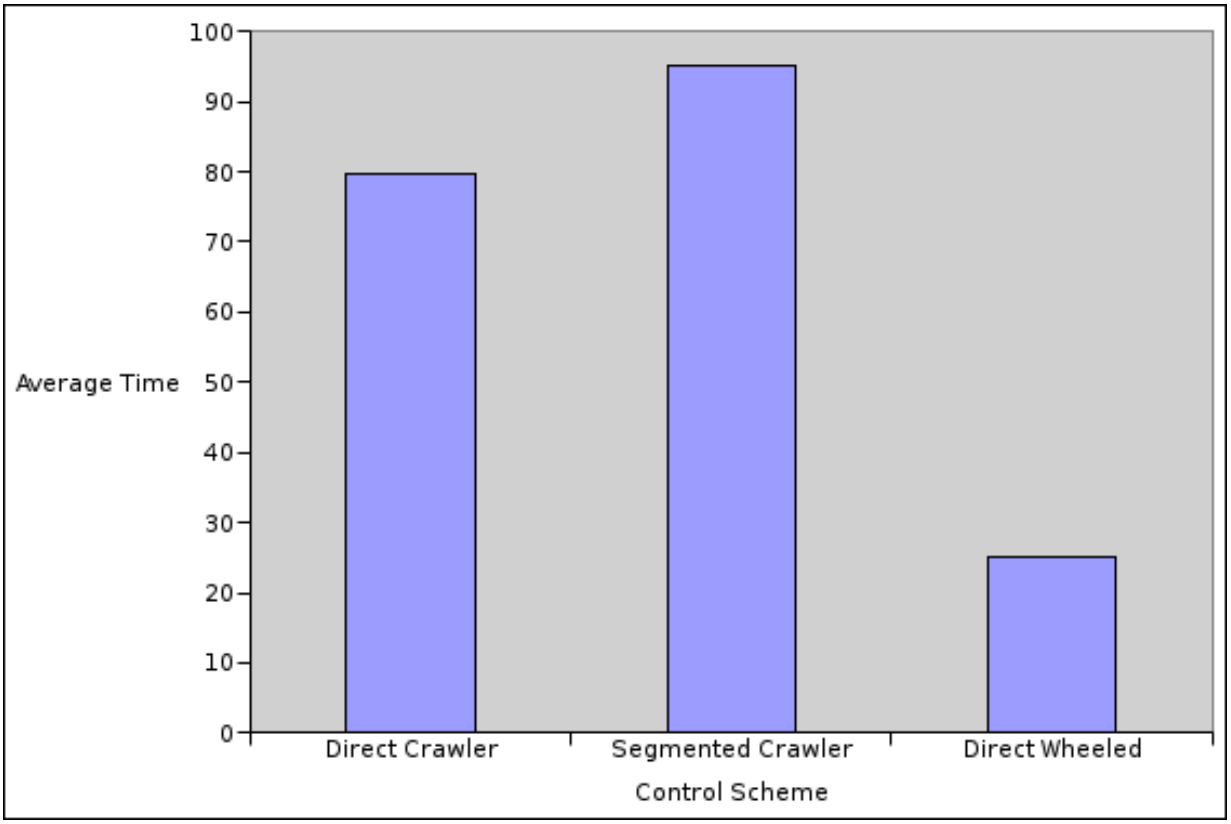
different mouse gestures.

Direct Control Wheeled

Wheeled Direct	Race 1	Race 2	Race 3	Average
Participant 1	20.97	22.56	23.96	22.4967
Participant 2	13.96	10.32	9.9	11.3933
Participant 3	33.2	29.12	30.54	30.9533
Participant 4	29.32	25.43	32.83	29.1933
Participant 5	32.91	35.31	30.76	32.9933
Participant 6	21.51	25.38	23.78	23.5567
Average				25.0978

The wheeled robot had the best overall time by far. However, there was a resounding negative feeling about the control scheme, and most participants found it annoying and difficult to use. This is partially due to deficiencies with the friction models with ODE, which rendered wheeled control difficult to mimic. That being said, participant 2 found a particularly good way of maintaining a forward heading by staying pressed against a wall.

6.6.4 Discussion



Strictly by the numbers, the best control scheme is clearly the wheeled robot. As we can see, the average time per trial was significantly lower than the two other control schemes. Between the two control schemes for the crawler, it is clear the direct control is better. However, in an informal side test, people found it much easier to turn with segmented control since it translates into a circular motion through the screen. Turning with the direct controller is much more difficult since it requires an unintuitive pattern of keystrokes. Hence,

it would be interesting to test control on a course which requires turning since it is quite likely the segmented controller would perform much better.