3D Character Animation

Using Geometric Constraints

Khaled Abdelhay

A Thesis

in

The Department of Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements for the Degree of Master of

Computer Science at Concordia University

Montreal, Quebec, Canada

January 2008

Your file Votre référence
ISBN: 978-0-494-40935-0
Our file Notre référence
ISBN: 978-0-494-40935-0

Abstract

3D Character Animation Using Geometric Constraints

Khaled Abdelhay

3D character animation is a very challenging field as it requires understanding of two different paradigms: fine arts and computer science. Currently there are many techniques used for animation authoring such as key framing, motion capture, non-linear editing, and forward and inverse kinematics.

A limiting factor shared between current animation techniques is that they represent animation as a series of translation and rotation without considering other important aspects such as the rational of the motion, geometrical constraints between characters acting in the scene, or even between a character and his environment. For example, "standing" on floor indicates that both feet touch and parallel to floor.

In this thesis I studied the foundations of classical and computer character animation, and then proposed using 3D geometric constraints as another method to represent animation. The method aims at capturing character's posture by a set of geometrical constraints and satisfing them using numerical methods. Maya plug-in has been implemented to build and solve symbolic geometric constraints for simple skeleton. The thesis includes a discussion on the results, usability and limitations of the proposed solution.

Acknowledgement

I would like to take this opportunity to give grateful thanks to my supervisor, Professor Peter Grogono, for his supports and guidance throughout this research project.

Contents

List of Figures

List of Tables

Chapter One - Introduction

Computer animation is a multi-disciplinary topic where fine arts, computer science, management, and movie technologies form a homogenous environment to create the magic of animated movies. A scholar of 3D computer graphics and animation needs to have a good understanding of different artistic and technical aspects required to create 3D animation.

For example lots of production elements change dramatically with the realism level of the characters in a story, i.e. realistic or cartoony. The sophistication and modeling techniques of cartoony characters are usually easier than realistic ones, which means less time and money. Shading models change with the realism level of characters, and they might dictate which software to use for rendering. Finding the balance between time and image quality requires cooperation between artists, software engineers, and managers to find the level where artistic and cinematic specifics are satisfactory while meeting time and money constraints. Animation is also affected by the level of realism. For example motion-capture better suites human-like characters, whereas cartoony characters might have new expressions or movements and hence requires animation by hand. In both cases, different set of skills and software are required to meet the requirements of animation. From this example, it is very apparent that one artistic element can change lots of software and technologies used for the movie, as well as the budget and time needed.

This chapter represents a thorough overview of 3D animation production pipeline, as well as introducing thesis objective and reviewing related work.

1.1 3D Animated Films Production Pipeline

3D animated films production is broken up into four areas: development, pre-production, shot-production and post-production. Each one is dealing with a different phase of a typical production cycle.

1.1.1 Development

Development covers the initial planning stages, including story development, character design, art direction, and storyboarding.

- <u>Story</u>

  A good story is the foundation of every successful animated film. In the early 1980s, when computer graphics was in its infancy, audience was captivated because the new imagery was previously unimagined. However, in those days even the strongest presentation cannot save a poorly formed story idea.

- <u>Character design</u>

  It can be easily argued that all memorable films contain memorable characters. It's very important to create a resume or biography for each character that indicates physical, historical, social, and psychological specifics.

- <u>Art direction</u>

  The purpose of art direction is to create a unique visual experience, indicate setting specifics, and generate a particular tone and mood for the story. Realism level, color palettes, and lighting specifics are some examples of the factors that determine art direction of a movie.

- Storyboarding

  The first step in realizing the ideas of the story is to set them down on paper as storyboards panels. It's a very important step because it provides a first opportunity to begin working out the cinematic specifics of the film, camera angles, composition, point of view, and many others.

1.1.2 Pre-Production

In pre-production the digital elements of the film are planned, created, and assembled. These elements include: vocal tracks, 2D and 3D animatics, CG modeling, texturing and character setup.

- Planning

  A production plan is like a roadmap. It constructs the methodical pathway toward a destination. Financial analysis, time estimates, scheduling, and digital assets are some key elements that a production plan should cover.

- Vocal tracks

  Vocal tracks production includes recording, erasing background noise, changing speed, adjusting volume, adding effects and many other steps. Having vocal tracks are very important for animation timing.

- 2D Animatic

  2D animatic is defined as storyboards plus timing and necessary audio. This stage is mainly focused on story elements and aims to experiment with the structure and pacing of the story. To create a 2D animatic we need a piece of video editing software, and a method of digitizing the boards as well as the vocal tracks.

- 3D Animatic

  3D animatic represents a stage where rough 3D layouts and characters versions replace the held storyboard images that were assembled during the 2D animatic. The focus of a 3D animatic is basic positioning and movement of scene objects, characters and cameras.

- CG modeling

  Three main types of geometry are used in the modeling process: polygons, NURBS and subdivision surfaces. Some factors like organic or hard-surface objects, modeler's proficiency, and rendering time determine which geometry type to choose. For example, a polygonal mesh would render faster than similar NURBS mesh.

- Texture and materials

  A material is defined by attributes like ambient color, diffuse color, specularity, transparency, and reflectivity. Materials, as well as other factors, affect the final look and quality of CG models. Textures are often used to improve the visual quality and believability of the materials.

- Character setup

Character setup consists of rigging and binding. Rigging means building an internal skeleton hierarchy to animate characters. Binding, also known as skinning or enveloping, refers to the process of connecting a character's outer geometry to its skeleton structure so it bends and twists nicely when the internal hierarchy is posed and animated.

1.1.3 Shot Production

Shot production covers animation, lighting, rendering, FX and compositing.

- Animation

Animation involves the creation of convincing and believable movement for living and non-living objects, such as humans and rocks. Some defines animation as the art of movement.

- Lighting and Rendering

The process of lighting a scene requires virtually the same principles used in traditional media such as painting, theater, photography and film. Examples of light sources include: point light, spot light, directional and area light. Rendering is considered to be one of the most expensive stages in production, and a good balance between quality and speed is needed to satisfy the aesthetic sensibilities and deadlines.

- Visual Effects

Natural movement of complex geometry pieces or large groups of objects can be especially difficult and time consuming to animate by hand. Procedural animation and physical simulation can generate an illusion of these complex animations. For example, explosions could be generated easily by physical simulation.

- Compositing

Compositing is the last step of CG shot production. Compositing reassembles what was previously separated, allowing a great deal control over the visual specifies of the final imaginary. For example, combining real actors with computer generated shots is done at this stage. Another example is adding 2D rain to a 3D shot.

1.1.4 Post-Production

Post-Production covers sound effects, titles and credits, and marketing.

- Sound Effects and Music

They clarify and enhance the visual actions, realism level, and mood of the film. Obtaining or creating sound effects is a very challenging task as it requires a lot of creativity and patient.

- Titles and credits

Although it might sound less important, titles often represents the initial connection to the audience, and they are the first impression of the film than can help indicate the mood, attitude and style of what is to come. Credits represent an opportunity to recognize the persons who contributed to the creation of a 3D movie.

1.2 Research Objective

It is apparent that there are many research topics in 3D computer graphics and animation, for example, considering geometry only, we can propose some research ideas like: 3D morphing, clothes simulation, new modeling techniques, etc. As a matter of fact, each stage in the production could be optimized or done completely in a new and different way. A research topic in 3D graphics and animation may focus on improving an existing functionality, or finding new solutions to existing problems.

This thesis aims at reducing animation time needed in 3D film production. Physical simulation was initially used to achieve this goal; however due to a defect in the physics engine of Matlab the project was halted. The defect is reproducible on Matlab version 7.0.1 R14 and is summarized as having incorrect simulation results when motion-actuating a joint containing three prismatic primitives. The defect is reported and assigned defect ID 298984. Matlab website suggests a workaround to this issue, however I tried and it did not work (http://www.mathworks.com/support/bugreports/details.html ?rp=298984).

Due to the previous defect, another approach was proposed and used in this thesis. The approach is based on using geometric constraints to capture the main spatial features of a character posture and then satisfying them numerically to reconstruct the same posture on different characters and/or environments.

1.3 Related Work

Jianhui Zhao, Ling Li and Kwoh Chee Keong [4] proposed a method to reconstruct 3D human posture from monocular images which is a cheap alternative to motion capture. This method allows the creation of 3D human animation from the video sequences, scanned photographs, and historical shots and hence it could be used to produce large number of motion banks that could be used in animated films production.

Petros Faloutsos, Michiel van de Panne and Demetri Terzopoulos [6] proposed a framework for composing dynamics controllers in order to enhance the motor abilities of virtual actors such as walking and running. The general idea is to design a *super controller* that controls *specialized controllers*. Each *specialized controller* defines pre-conditions in which it can be invoked, and the *super controller* chooses the best controller to achieve the current goal.

Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey Hinton [7] used neural networks to create realistic animation. They replaced physics-based models by fast neural networks which automatically learn to produce similar motions by observing the models in action.

Depending on the model, the neural network emulator can yield physically realistic animation one or two orders of magnitude faster than conventional numerical simulation Mira Dontcheva, Gary Yngve, Zoran Popović [8] introduced a system that allows users to create and edit character animation by acting. The key contributions of their work lie in how the animator's motions are mapped to those of the character, and how animation can be built upon layer by layer.

C. Karen Liu, and Zoran Popović [9] presented a general method for rapid prototyping of realistic character motion by applying simpler dynamic constraints. The method is summarized by analyzing an animation sketch provided by an artist, and inferring environmental constraints on the character motion. Afterwards, theses constraints are used to synthesis other animation sequences or modify existing ones.

Chen Mao, Sheng Feng Qin, David K. Wright [10] presented a fast storyboarding interface, which enables users to sketch-out 3D virtual humans and 2D/3D animations. This technique allowed beginner to create virtual humans and animate them quickly. Their system is based on using 2D stick figures to describe main character poses, which are used to reconstruct 3D continuous motion. Their system also uses skeleton proportions and shape to create corresponding 3D models

C. M. Hoffmann and P. J. [11], proposed an analytical and numerical method to solve 3D spatial constraints. The analysis phase uses a graph to represent geometries as nodes and constraints as links, then tries to find a generic solution for the given problem. The numerical phase converts the given solution into equations and solves them. The drawback of this technique is that it uses only points and planes to represent geometries which cannot be used in character animation.

Glenn A. Kramer [12], proposed an elegant way to solve geometric constraints problem. His method is summarized by creating a database that describes different kinds of geometric constraints and how they could be satisfied analytically. The database is also categorized by the *Degree of Freedom (DOF)* available for each geometry in the system. For example, to satisfy a geometric constraint between geometry A and B, the database includes different algorithms for different *Degree of Freedom* for A and B:

- A and B have 6 *DOF*
- A has 6 *DOF* and B has 5 *DOF*

and so on. The drawback of this technique is that it requires a considerable versatile database and it does not scale well for generic characters animation. Also, for complex geometries and constraints system, e.g. character animation, the solution obtained by this technique becomes numerical rather analytical, which defeats the purpose of building constraints database in the first place.

Hiroshi Hosobe [13], wrote Chorus3D library which uses numerical optimization and genetic algorithms to solve 3D geometric constraints problem. A positive error function is associated with each constraint, and the *quasi-Newton* method is used to minimize the summation of the error functions. He also used weights to classify constraints as required, strong, medium, or weak. Genetic algorithms are used to search for global solutions.

Chapter Two - Background on Character Animation

2.1 A Brief History of Animation

People have used static images to suggest motion for a very long time. Over 35,000 years ago, humans were painting animals on cave walls, sometimes drawing four pair of legs to show motion. In 1600 BC the Egyptian Pharaoh Rameses II built a temple to the goddess Isis which had 110 columns. Ingeniously, each column had a painted figure of the goddess in a progressively changed position.

As far as we know [2, page 13], the first attempt to project drawings onto a wall was made in 1640 by Athonasius Kricher with his 'Magic Lantern'. Kricher drew each figure on separate pieces of glass which he placed in his apparatus and projected on a wall. Then he moved the glass with strings, from above. One of these showed a sleeping man's head and a mouse. The man opened and closed his mouth and when his mouth was open the mouse ran in.

Since then, there have been lots of experiments and work on animation. In 1928, Mickey Mouse took off with his appearance in *Steamboat Willie*— the first cartoon with synchronized sound. In 1932 Walt Disney produced the first full color cartoon *Flowers* *and Trees*. Then he followed it one year later with *Three Little Pigs*, and four years after that, Disney released *Snow White and the Seven Dwarfs.*

The tremendous financial and critical success of *Snow White and the Seven Dwarfs* became the foundation of Disney's output and gave birth to the "Golden Age" of Animation: *Pinocchio 1940, Dumbo 1941, Bambi 1942,* and *Fantasia* 1942 as well as Donald Duck and Mickey Mouse.

As more films were produced, knowledge was accumulated and animation started to be a science by its own. This knowledge applies to any style or medium, no matter what the advances in technology are. If drawn 'classical' animation is an extension of drawing, then computer animation can be seen as an extension of 'classical' animation.

2.2 Classical Animation Principles

This section discusses the most common principles of classical animation.

2.2.1 Squash and Stretch

Objects tend to have some degree of elasticity. When forces act upon them, they will deform appropriately depending on the nature, direction, and degree of those forces mixed with the physical properties of the objects themselves. For example, if we step on a tennis ball, it will squash vertically and stretch horizontally.

2.2.2 Anticipation

Anticipation is the setup before a main action, and it is often a physical movement in the opposite direction of the intended motion. They can also take on other forms like a growl before an attack or a deep breath before a bold statement. The rational behind anticipation is that a viewer's eye tends to lag behind by a few frames, so it is often necessary to announce that something is about to happen so the audience does not miss it. It is worth pointing out, although it is obvious, that anticipation and similar tricks are not what actually happens, however animators introduced these tricks to enhance animation and viewers accepted them.

2.2.3 Follow-Through

Follow-Through is the extension of the performed movement. For instance, if the character has any appendages, such as long ears or a tail, these parts continue to move after the rest of the figure has stopped.

2.2.4 Overlapping Action

Overlapping action is the concept that not all moving parts of a body will start and end at exactly the same time. For example, if you turn your head and point, your arm movement might begin before head finishes turning. Without overlapping, motions tend to look robotic.

2.2.5 Slow-In and Slow-Out

Organic motion tends to accelerate and decelerate into and out of actions except when met with a force that causes an abrupt stop or direction change, such as a wall or a floor. The movement of an object will look mechanical if we don't apply this principle.

2.2.6 Arcs

When a wrist travels from point A to point B, it typically does so as the result of elbow and shoulder rotations. Therefore, the motion of the wrist will tend to describe an arc. Although this role seems to be very logical to scientists, it is surprising to learn that inexperienced animators make this mistake.

2.2.7 Secondary Action

There is usually a secondary motion to the main action— for instance, drumming your fingers on you knee while talking. Secondary actions often reveal emotional subtleties or hidden thoughts.

2.2.8 Timing

Varying the speed of a particular motion can indicate different weight, forces, and attitudes. A slow head turn might indicate a careful or casual search, while a quick head turn can indicate surprise. Fast walks can imply determination; slow walks can imply depression.

2.2.9 Exaggeration

Exaggeration is used to increase the readability of emotions and actions. For example, if a character was to be sad, make him sadder; bright, make him brighter.

2.2.10 Appeal

Appeal is the most subjective of all the principles. It simply means that a character or its performance is visually interesting.

2.3 Classical Animation Approaches

Before computer animation, animators depended mainly on two animation techniques [3, page 53]:

2.3.1 Pose to Pose

In the "Pose to Pose" approach, the animator plans his action, figures out just which drawings will be needed to animate the scene, makes the drawings, relating them to each other in size and action, and gives the scene to his assistant to draw the in-betweens. Such a scene is easy to follow and works well because the relationships have been carefully considered before the animator gets too far into the drawings.

One of the main benefits of this approach is the strength of poses an animator can display. If done correctly, the important beats of a scene are clearly defined.

Also this approach definitely lends itself to adjusting the timing of a shot as the character's poses are neatly contained at intervals along the shot.

2.3.2 Straight Ahead

In the "Straight Ahead" approach, the animator literally works straight ahead from his first drawing in the scene. He simply takes off, doing one drawing after the other, getting new ideas as he goes along, until he reaches the end of the scene.

This method has significant advantages. The spontaneity it generates can create some extremely successful results. The animator can feel his way through a scene and alter the conclusion if he desires. In addition, complex motions can be pared down to mini-actions that are easier to resolve.

2.4 Computer Animation Approaches

All classical animation principles are also applied to computer animation. This section introduces some common computer animation approaches.

2.4.1 Key Frame Animation

Key-frame animation is one of the oldest computer animation techniques. It resembles pose to pose technique in classical animation where each pose represents a key frame. Key frames specify time, location and pose of a character. The in-betweens are created by mathematical interpolation rather than human intervention.

Modern animation packages usually have graphical editors where animators can change the location and time of keys, as well as the interpolation method used. For example, figures 1 and 2 show the key frames of a falling ball before and after editing

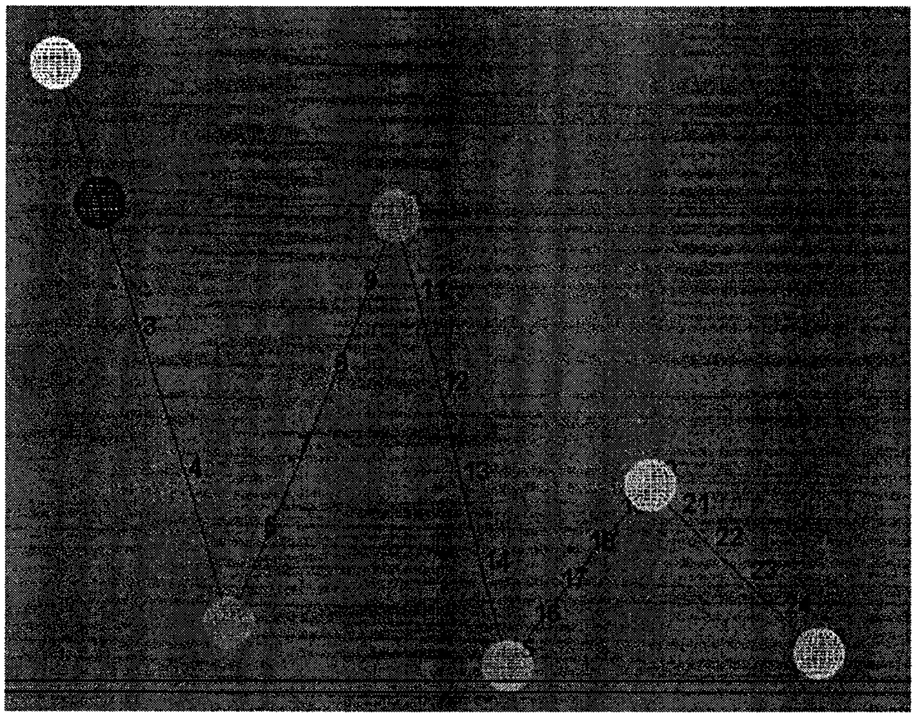


Figure 1-Key frames of a falling ball

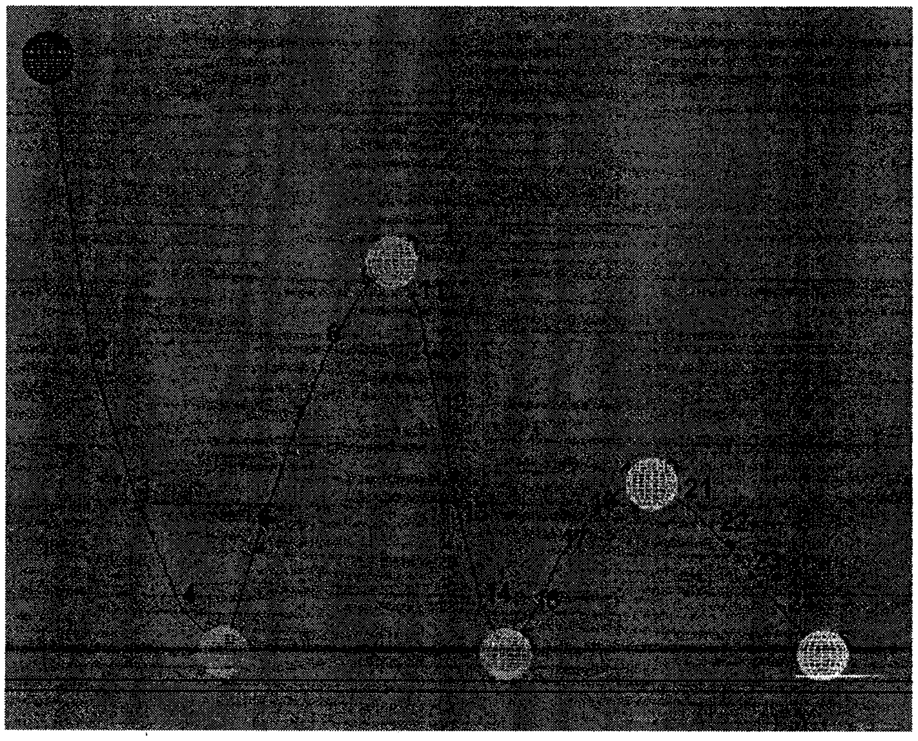


Figure 2-Enhanced falling ball key frames

2.4.2 Physical Simulation

Animating natural movement such as cloth, hair, smoke, water, snowflakes or a collection of falling rocks can be too complex to be done by key framing.
Even in cartoon-style animation, audiences except the movement of such elements to be believable; it is therefore necessary to simulate the effects of gravity and physics convincingly.

Most computer animation packages offer physical simulation solutions that animators can use to simulate natural movement of complex geometry or large groups of individual objects. Figure 3 and 4 show two examples where physical simulation is required.

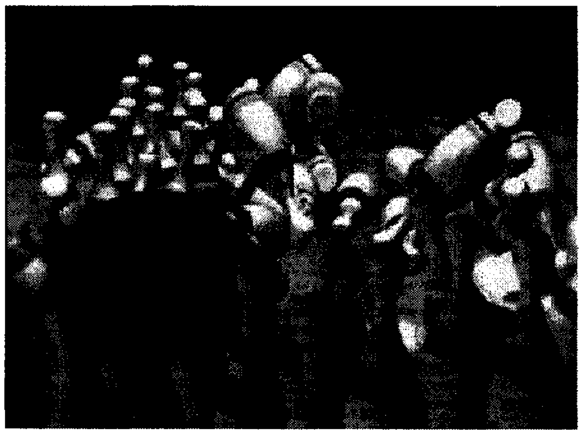


Figure 3- Physical simulation animation example 1

(Source: Maya documentation)

Figure 4-Physical simulation animation example 2

(Source: Maya documentation)

2.4.3 Motion Capture

In Motion Capture, a human performer wears markers near each joint to identify the motion by the positions or angles between the markers. Markers are used by a tracking system to reconstruct the performed motion; for example, an optical tracking system typically uses small markers attached to the body—either flashing LEDs or small reflecting dots—and a series of two or more cameras focused on the performance space. The motion capture and software picks out the markers in each camera's visual field and, by comparing the images, calculates the three-dimensional position of each marker through time.

Motion Capture provides an accurate digital representation of the motion. Figure 5 shows Tom Hanks using a motion capture system for his movie Polar Express

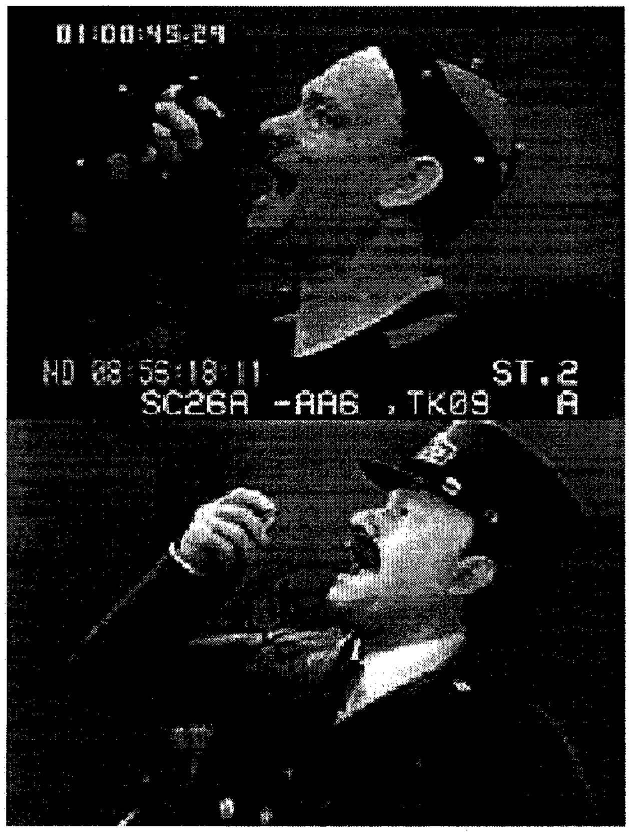


Figure 5-Motion capture example

A cheap alternative to motion capture is rotomatting, which means posing and animating a character on top of a filmed background plate. For example, a human performer will be filmed playing out a scene, and then a cartoon character will be drawn manually on the top of the human performer to imitate his motion.

Motion Capture could be used to create motion cycle, which is a movement that loops when played back multiple times. Animation cycles could be used as shortcuts for movements that repeat over time, for instance, walking could be represented as one animation cycle.

2.4.4 Non-Linear Animation

After animating a character with key frames or motion capture, we can collect its animation data into a single, editable sequence. This animation sequence is called an animation clip. Moving, manipulating, and blending regular clips to produce a smooth series of motions for a character is the basis of nonlinear animation.

Most animation packages provide high-level non-linear animation editors. Most editors provide many functions like selecting characters and their animation clips, layering and blending animation sequences, and synchronizing animation and audio clips.

2.4.5 Forward and Inverse Kinematics

Although it might be possible to animate simple character models by merely translating, rotating, scaling or reshaping their overall forms or individual parts, we generally need to turn these character models into digital "puppets" by adding internal skeleton to them. A digital skeleton is typically made up of a hierarchy of joints; each joint is represented by a transformation matrix. Each joint in a skeleton hierarchy may be both a *child joint* and a *parent joint*. A *parent joint* is any joint higher in a skeleton's hierarchy than any of the other joints that are influenced by that joint's actions. Joints below a *parent joint* in the skeleton hierarchy are called *child joints*. The joint that is the highest joint in a skeleton's hierarchy is called the *root joint*.

Most animation packages call the virtual line between any two consecutive joints a *bone*. Bones are only visual cues that illustrate the relationships between joints. Figures 6, 7, and 8 demonstrate these ideas.

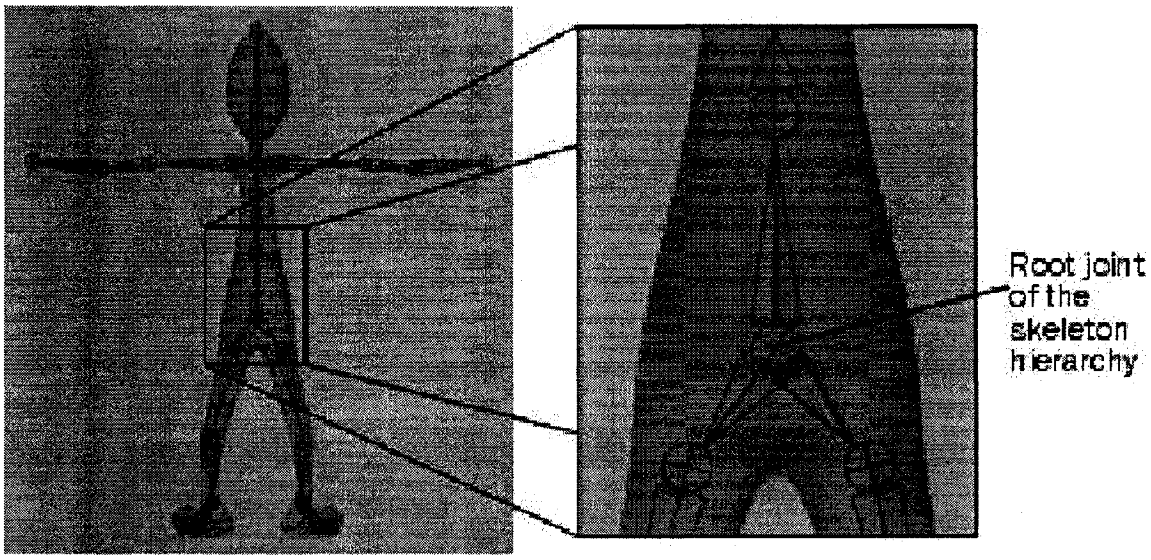


Figure 6-Animation skeleton and the root joint

(Source: Maya documentation)

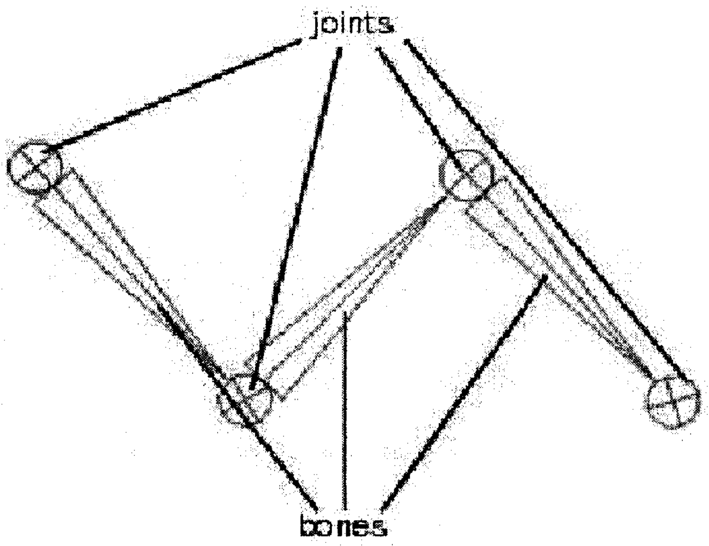


Figure 7-Joints and bones

(Source: Maya documentation)

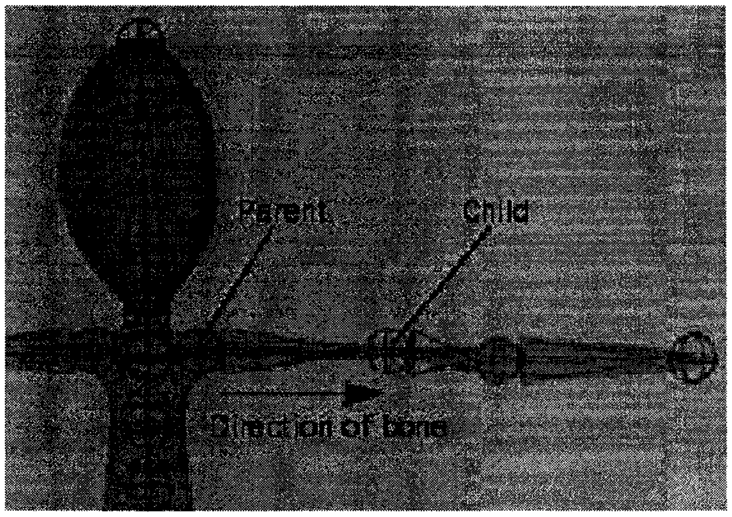


Figure 8-Parents and child joints

(Source: Maya documentation)

Characters with skeletons are typically posed simply by rotating some or most of their joints — for instance, turning character's head involves only his nick joints, while running or kicking a ball will require most of the joints to rotate. Top level root joints are generally translated and rotated for global positioning and orientation, but the actual pose of the body is reached by rotating each of the remaining joints.

Posing skeleton chains and hierarchies by rotating joints is known as *forward kinematics* or simply FK. For instance, bending an arm with forward kinematics typically involves rotating the shoulder and elbow joints independently to reach the desired pose.

An alternative method of posing joint structures is by using *inverse kinematics* or IK, in which the end joint of a skeleton chain is translated to a desired location and the individual joints in the hierarchy are computed to bend accordingly to compensate. The end joint that gets translated is sometimes known as an *end effector.*

Choosing between forward and inverse kinematics is a matter of preferences, however, some animators prefer IK for legs, and FK for arms.

Chapter Three – Design and Implementation

3.1 Problem Definition

From our review on classical and computer character animation, we can argue that a pose of a character is one of the corner stones in authoring animation. Pose description is an art form and a strong pose can tell volumes about a character. Adding the time element to subsequent poses creates the magic of animated characters, or as artists say: breath life into them. From this perspective, we can look at character animation as poses plus timing.

Currently poses are created by animation artists, or by motion capture which creates an entire animation sequence. Unfortunately, the way poses are represented is very limited in terms of reusability as they depend merely on storing rotation and translation of the joints that make up the character skeleton. Applying prerecorded rotations and translations on different characters or even the same character but in a different environment will fail, or at best will require human intervention to adjust the animation to the new character or environment.

3.2 Proposed Solution

To address the pose representation problem, this thesis proposes encoding geometric constraints into the pose description such that they capture main visual characteristics of the pose. Visual characteristics include: ratios between angles, ratios between distances, parallelism, orthogonality, etc.

A simple example would be describing a pose in a walk cycle. A walk cycle can be seen as pivoting one foot on the floor and swinging the other to the next step. For a character walking uphill, the pose shown on figure 9 and figure 10 can be described many geometric constraints, among them:

- The step size should be one fifth of the character height,
- The leg that is swinging should land on the floor,
- The feet that is pivoting should be touching and parallel to the floor

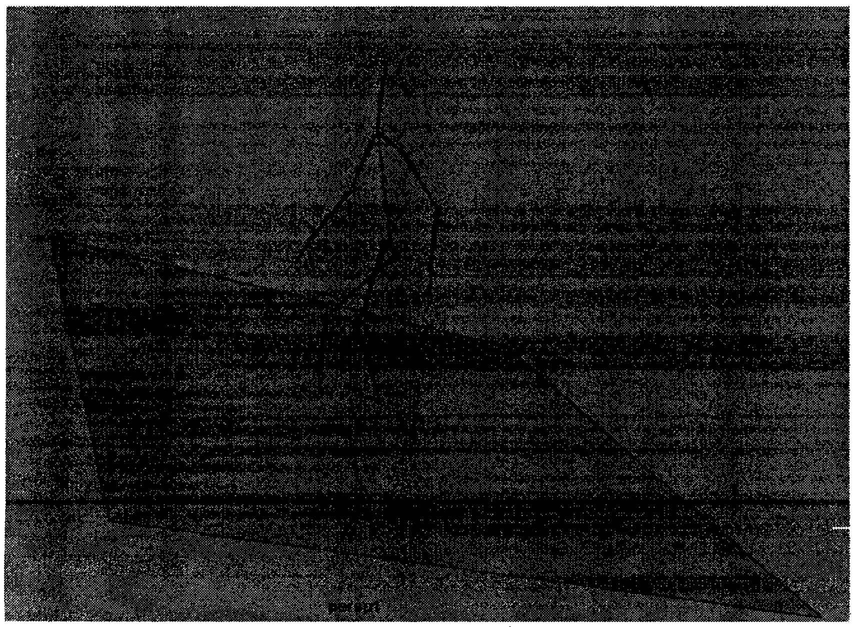


Figure 9 Charactering walking pose – Perspective projection

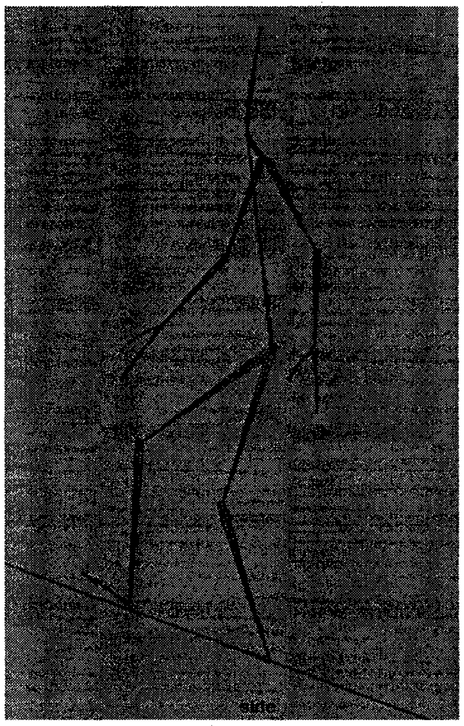


Figure 10 Character walking pose – Side projection

The most important aspect of these constraints is that they do not specify actual or absolute values; otherwise they will defeat the purpose of using geometric constraints. Instead, these constraints focus more on generic relations that will be always true in all characters regardless their dimensions or environments.

Geometric constraints are identified by pose creator and it is his job to provide a proper set of constraints that describe each pose. Augmenting each pose of an animation sequence with a corresponding set of constraints makes the animation generic and applicable on different characters and environments. The same idea is applicable also on animation clips created by motion capture.

However, we must state that, in this thesis, geometric constraints should not be used to synthesis an entire pose by itself. Instead, they should be used to adjust a given pose to its new environment or character because:

- Not all features have apparent geometric constraints
- The number of constraints required to describe a pose entirely is linear to the number of joints in a character which means an unaffordable number of constraints will be required. For example, for the character used in this thesis, 30 independent geometric constraints are required to fully describe the pose.

3.3 Design Overview

The goal of this thesis is to determine the feasibility of using geometric constraints as a method to reconstruct animation poses from constraints equations and an initial solution.

3.3.1 Geometric Constraints Satisfaction Problem

Geometric constraints could be satisfied either analytically or numerically:

- <u>Analytical methods</u>

  An example of analytical method is Degree of Freedom Analysis by Glenn A. Krammer [12]. In his analysis, he created a database that stores information about different kinds of constraints, algorithms to satisfy them, and affected Degree of Freedom. For example, there is an algorithm to describe how to rotate an object to be aligned a line passing through one of its vertices.

- Numerical methods

  Geometrical constraints are encoded as algebraic non-linear simultaneous equations and solved numerically using algorithms such as *Newton-Raphson*. It is also possible to convert geometric constraints problem into an optimization problem and solve it numerically [13].

This thesis uses numerical methods due their ability to solve a versatile range of geometric constraints without prior need to geometrical reasoning functionality or algorithms to solve individual constraints. On the other hands, numerical methods suffer from other problems like convergence speed, floating point rounding errors, local minimum, etc.

3.3.2 System Components

The solution proposed in this thesis contains four main components; the following is the summary of each component:

- Autodesk Maya

  Maya is used to do the following tasks:

  - Design and build simple character skeleton and environment. The skeleton has 16 joints with a total of 30 Degree of Freedom in 3D space. The environment contains only a floor and a group of markers in 3D space. A *marker* is a point in 3D space that could be easily queried to obtain its coordinates in world coordinate system.

    - Pose the skeleton to provide an initial solution to the rest of the system.

    - Display the results created by other components to the user.

- Symbolic engine

  The *symbolic engine* facilitates the process of building and evaluating mathematical expressions. In this thesis, *mathematical expressions* represent the geometric constraints to be satisfied. The design of the symbolic engine is tailored toward the thesis and aims at providing users with a tool to build and evaluate easily geometric constraint equations. The other alternative will be building equations manually which is error-prone and time consuming.

- Numerical solver

  The *numerical solver* solves simultaneous non-linear equations for the geometric constraints. The solver is based on a slightly modified version of Newton-Raphson algorithm [1, 14]. The version developed for the thesis uses the *Generalized Inverse Matrix* formulas for matrix inversion [1, 14]. This modification is required to cope with singular matrices created due the inequality between number of constraints and number of unknown variables in the system.

- A Maya plug-in

  The plug-in uses the symbolic engine and the numerical solver to read skeleton and environment description, build corresponding geometric constraints, invoke the numerical solver and update Maya scene with the result.

The symbolic engine and the numerical solver were written entirely, in C++, due to the lack of available free packages that can do the same job. The plug-in was written in C++ and uses Maya SDK to interface with Maya internal functions, such as reading skeleton dimensions, creating key-frames, etc.

Figure 11 shows a conceptual design of the main components and their interaction with each other.

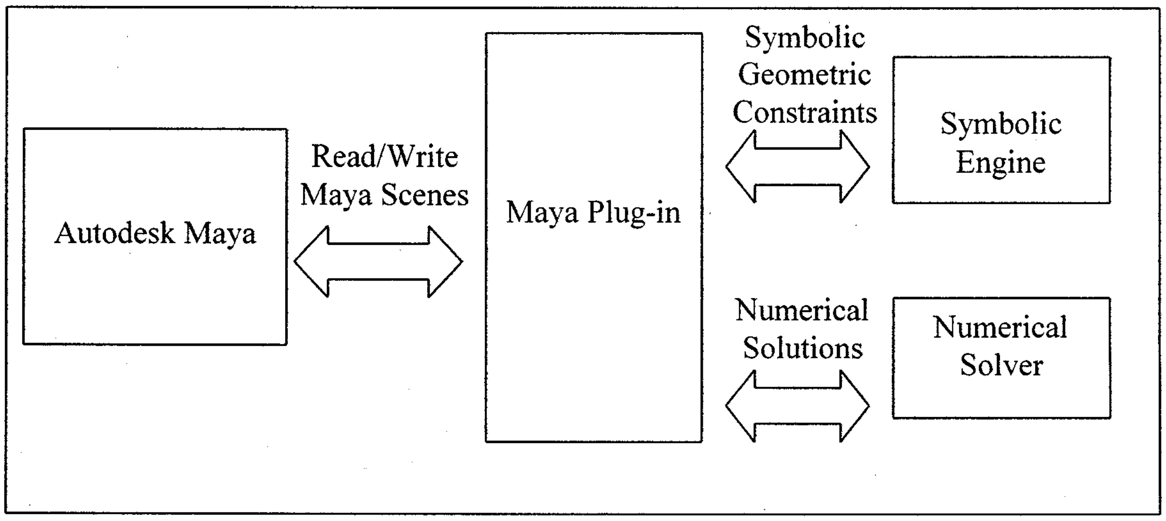


Figure 11 System Components

3.4 Character Model

The character is represented as a 33 joint skeleton in which 16 are used in geometric constraints formulation. Some joints are used for displaying purposes only, for example fingers, toes, and head joints. The unused joints help give a skeleton a human look as shown in figure 12, as well as help visualize the solution obtained.

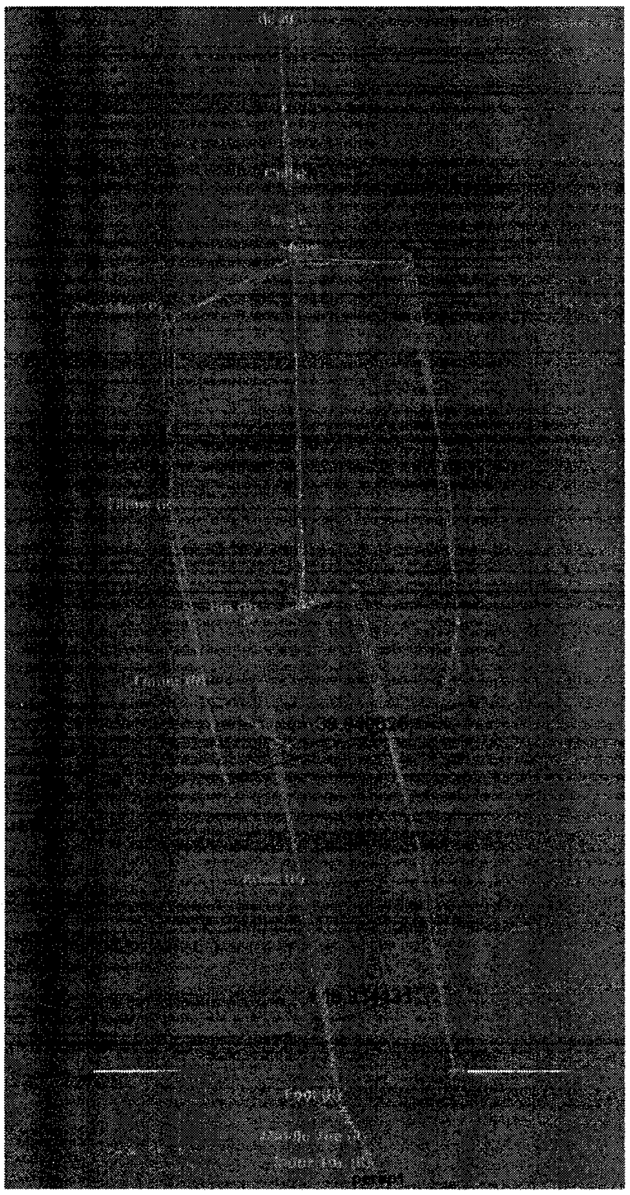


Figure 12-Character Skeleton

In Maya, the skeleton is represented as hierarchical transformations chains. The root joint represents the first transformation and the rest of the skeleton is built upon it. As in standard mathematics, in Maya each joint is defined as a sequence of translation, rotation and scaling. However, in this thesis, I used only translation and rotation to simplify the equations created. Figure 13 shows how the skeleton is represented internally in Maya.

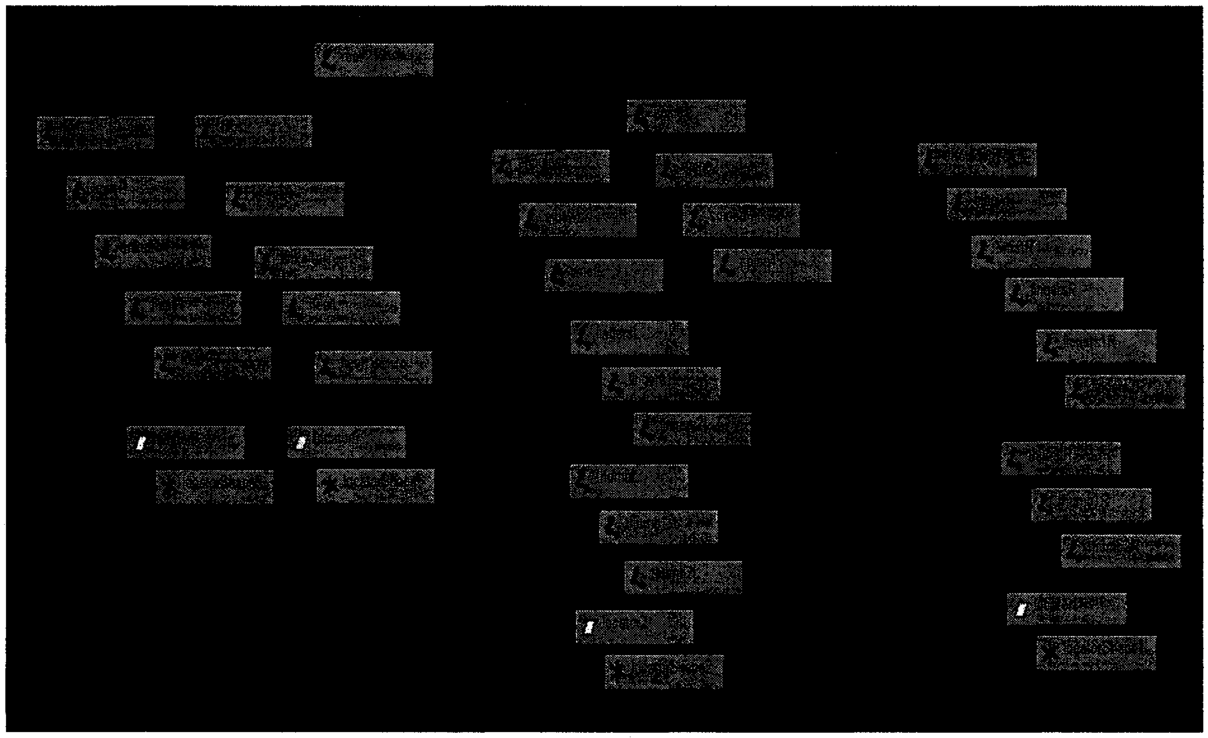


Figure 13 Maya Skeleton Representation

In the figure above, "root" transformation matrix is defined with respect to world coordinate system; "hipsL" transformation matrix is defined with respect to the "root" and so on.

Note that each joint is given a unique name which helps map them between Maya and the symbolic engine. The locators shown at the bottom part of the graph are used for visualization purposes only, for example “locator6” and “locator5” measure the distance between “ankleL” and “ankleR” and print the result on screen.

3.5 Environment Model

Environment is modeled as a titled floor and a box. Instead of providing high-fidelity description of the environment and the box, markers are used to describe important features in the environment. A *marker* is a point in 3D space as shown in figure 14, and it can be manipulated easily using Maya interface.

By using markers we can describe the floor as one marker that lies inside it, and two markers to indicate the floor normal. Although Maya provides a rich set of API to describe complex geometries, using markers helped implement a simple environment quickly. The box is described as two markers to indicate its width.

Markers are also used as short-cut to more complex computation, for example two markers could be used to specify a vector that lies inside a plane instead of generating one mathematically. Another example is if a foot has to step on the floor, a marker will be created on the floor to indicate the point where the foot should touch instead of projecting the heel and calculate the location of this point.

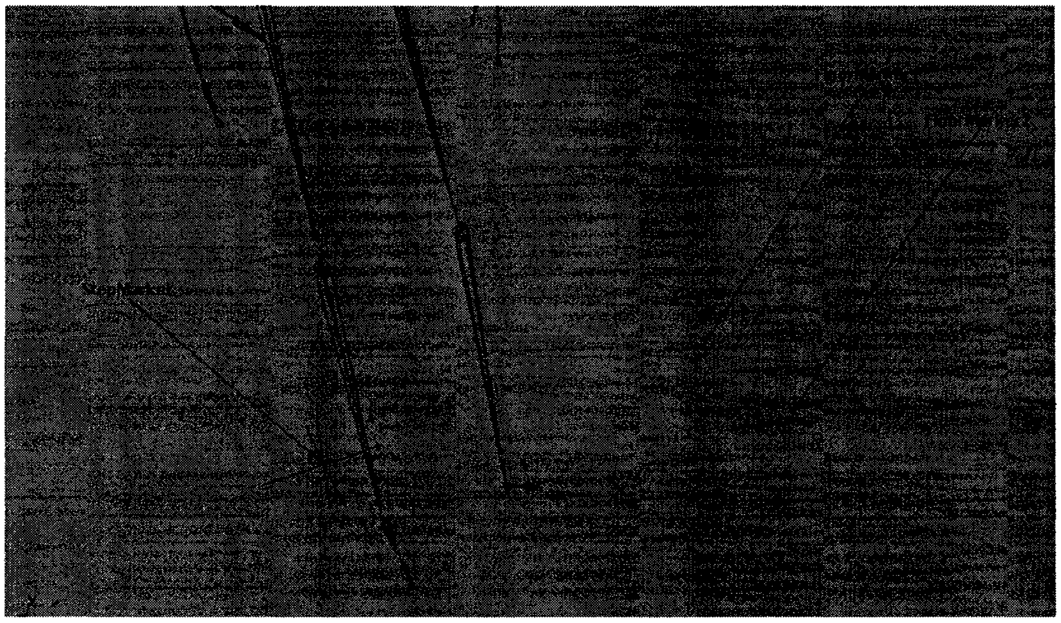


Figure 14 Environment markers

3.6 Symbolic Engine

The main building blocks of the symbolic engine are *symbols*, *symbols table*, and *expressions*.

3.6.1 Symbols

A *symbol* is used to represent any physical entity in Maya scene. For example, joint "root" in the skeleton is represented by the following six symbols:

"root.tx" for translation in X direction

"root.ty" for translation in Y direction

"root.tz" for translation in Z direction

"root.rx" for rotation around X axis

"root.ry" for rotation around Y axis

"root.rz" for rotation around Z axis

Each symbol has a unique ID that identifies in internally.

3.6.2 Symbols Table

The symbolic engine maintains a *symbols table* that contains the definition of each symbol used in the system as shown in table 1. Each symbol has a unique ID (or *Symbol ID*), *type*, *value* and *representation string*. The following shows a segment of the symbol table. The symbols table is stored as hash-table where the symbol ID is used as the table key, which ensures unique symbols ID.

Symbol ID (unsigned int)	*Symbol Type* (enumeration)	*Representation String* (string)	*Value* (double)
1	Power	^	N/A
2	Add	+	N/A
3	multiply	*	N/A
4	Cosine	cos	N/A
5	Variable	x	0.0
6	Variable	y	1.2
7	Constant	PI	3.14
8	Constant	const_1	45
..	..	..	..

Table 1 Symbol Table

- *Symbols IDs* are allocated dynamically when the program starts and during the course of its execution.

- *Symbol Type* is defined as enumeration and is used to determine how to evaluate and display symbols. For example if the type is "cosine", then the symbolic engine evaluates the symbol by calculating the cosine value of its argument. For displaying purposes, the symbolic engine uses the representation string associated with type "cosine" to print on screen, in this case the engine prints "cos". As a general role, if the symbol type is not "constant", the representation string will be used to print the symbol.

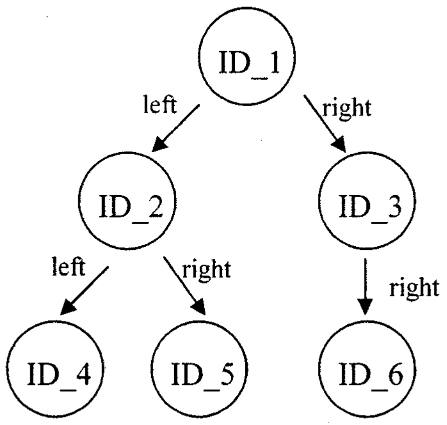
  On the other hand, if the type is "constant" then the symbolic engine uses the associated *value* field to evaluate and print the symbol. Note that symbols of type "constant" could be created by user or by the system when it optimizes expressions.

  Finally, *Symbol type* contains the following symbols: *constant, variable, add, sub, multiplication, division, power, sine, cosine,* and *tan.*

- *Representation strings* are used to print symbols on screen. Representation strings are provided by the user when he creates symbols, or generated automatically for system created symbols. The only exception for using representation strings for printing is symbols of type "constant" as they are printed using their associated values.

3.6.3 Expressions

An *expression* is a C++ class that owns a binary tree. Binary trees are used to represent mathematical expressions, such that each tree holds an entire expression. A sample tree is show figure 15 below. For demonstration purposes, a sample *symbols table* is shown on the right hand side of the tree to show the association between symbol IDS and their definitions in the symbols table.



Symbol ID	Symbol Type	Representation String	Value
ID_1	add	+	N/A
ID_2	multiplication	*	N/A
ID_3	cosine	cos	N/A
ID_4	variable	Y	23.4
ID_5	constant	N/A	33
ID_6	variable	X	1.4

Figure 15 Expression binary tree

The equivalent expression of the binary tree above is (Y*33)+(cos(X)).

Nodes are allocated and deallocated dynamically upon request. Each node stores two pointers to its sons, as well as *Symbol ID* which is used to interpret the meaning of the node using the *Symbol Table.*

Expression class defines a set of mathematical operators to manipulate its associated binary tree. These operators corresponds the types defined in the symbol table and they include: addition, subtraction, multiplication, cosine, sine, power, and equality. Complex expressions are built by aggregating smaller expressions using the mathematical operators.

Expression class also defines a set of helper functions that are needed for symbolic processing, such as:

- *Expressions* simplification: will be discussed in details in section. 4.3.2 on page 66.
- Partial differential: uses standard mathematics roles to differentiate expressions tress, for example $d(x * y + y) / \partial x = y + 0$.
- *Expressions* evaluation
- *Expressions* Printing
- *Expression trees* copying and deleting

3.7 Numerical Solver

The solver implements *Newtown-Raphson* algorithm and *Moore-Penrose Generalized Inversed Matrix* to solve a system of non-linear simultaneous equations.

A full description of the *Newtown-Raphson* and the *Moore-Penrose Generalized Inverse Matrix* can be found in [1], [14] and [18]. The following is the summary of the algorithm:

- Input:
    - System of symbolic constraint equations, or SCE
    - Initial solution vector, or IS
    - Maximum number of Iteration, or MaxItr
- Output: solution vector
- Algorithm:

```
1-Iteration ← 1
2-CS ← IS (CS stands for Current Solution)
3-IE  ← SCE ( IE stands for Input Equations)
4-SJM ← calculate Symbolic Jacobian Matrix of IE
5-EJM ← Evaluate SJM, i.e. substitute CS into SJM and evaluate
6-GIM ← calculate the Generalized Inverse Matrix of EJM
7-DX   ← GIM * CS (DX stands for Delta X)
8-CS ← DX + CS
9-Iteration ← Iteration + 1
10-if Iteration <  MaxItr Jump to 5, else print CS and quit
```

The exit condition of the algorithm is not optimized. A right implementation will use a heuristic function to measure the deviation of the consecutive solutions or/and satisfiability of the constraint equations to determine when to stop. However, due to time constraints I used the above implementation and visualized solution progression using

Maya. Chapter 4, i.e. Testing and Results, elaborates more on the conversion speed of the solution.

3.8 Usage Model

This section demonstrates how to use the system from user's prospective. The following scenario highlights the required steps to use the system:

1. Design character and environment using Maya
2. Pose the character and deduce a group of geometric constraints that collectively capture the characteristics of the pose
3. Use the symbolic engine to create corresponding constraints equations
4. Invoke the numerical solver
5. Apply the result on the character
6. If the result is not satisfactory, refine the geometric constraints and try again.

Chapter 4 demonstrates different testing cases and their results

Chapter Four - Testing and Results

4.1 Introduction

The testing process includes 10 equations that represent 6 spatial geometrical constraints. As mentioned earlier, the skeleton has 33 joints, in which 16 are used and create a total of 30 Degree of Freedom in 3D space. The testing aims at addressing the following points:

- Whether a solution could be found if environment description or character dimensions or pose have been changed
- How fast could the solution be found
- What is the memory requirement for the system

The first point has been tested against individual constraints as well as all constraints combined. The numerical solver succeeded in finding solution for each geometric constraint individually, however 5 equations out of 6 can be satisfied simultaneously. Section 4.2.6 discusses this problem in details.

The second and third points were tested against the 6 geometrical constraints combined together as performance issues were handled at the end of the research. In general, the symbolic engine performance needs to be optimized as it's considered the bottle-neck of the system. Section 4.3 discusses these points in details.

4.1.1 Templates and Notation Used in Testing Process

In order to facilitate the description of the geometric constraints, the following notation is used:

- (Joint) = Spatial position, i.e. xyz, of joint "Joint" in world coordinate system.
- (Joint2-Joint1)= Vector from Joint1 to Joint2 defined in world coordinate system
- (dot)= Dot product between two vectors, e.g. vector1 (dot) vector2
- distance(Joint2, Joint1)=Square of the distance between vector (Joint2-Joint1)
- normalize(Joint2, Joint1)= Normalized vector (Joint2-Joint1)
- length(Joint2, joint2)= Length of vector (Joint2-Joint1)

A constraint is written such that when it is satisfied it equals to zero. For example, if we want to say that position of Joint1 and Joint2 are the same, we define the constraint as: (Joint1-Joint2)

Also the following template is used to describe geometric constraint test cases:

Description: The purpose of the constraint or what it should achieve

Markers Used: Markers used to facilitate writing constraints equations

Helper Equations: Equations used to formulate geometrical constraints

Constraints: Equations passed to the numerical solver to be solved.

Notes: If required, notes are added here

Figure: Figure number associated with the geometric constraint

4.1.2 Test Cases Summary

Test cases have been designed to address a pose in walk cycle as shown in the figure 16 below where a character is walking on a tilted floor and carrying a box. The box is not shown; however the character hands have to maintain a certain distance apart simulating the width of the box. In general, during a walk cycle, one leg is swinging and the other is pivoting on the floor. In the following discussion, “swinging” and “pivoting” will be used to refer to these states respectively.

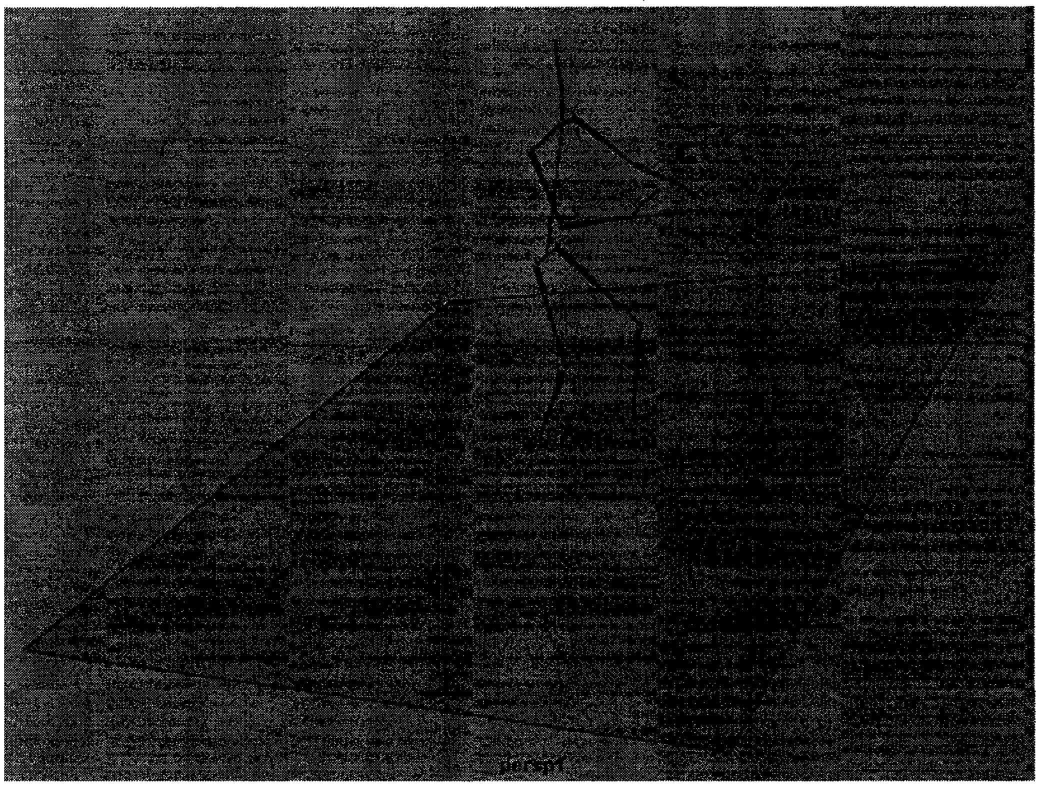


Figure 16 Character walking pose

The test cases below handle each geometric constraint individually, i.e. each test case addresses one geometric constraint. Note that a single geometric constraint might be realized by multiple equations, for example imposing parallelism between two vectors in 3D requires two equations to be satisfied.

Finally, all geometric constraints are tested as a group and section 4.3 discusses results in details. In summary, the system requires 3 to 5 iterations to find a solution for the all geometric constraints combined if the initial pose is close to the final solution. If the initial pose is not close to the solution, the system takes around 20 iterations to find a solution and although the solution is mathematically correct, the pose created might not look a possible or not a human pose as seen in figure 17.

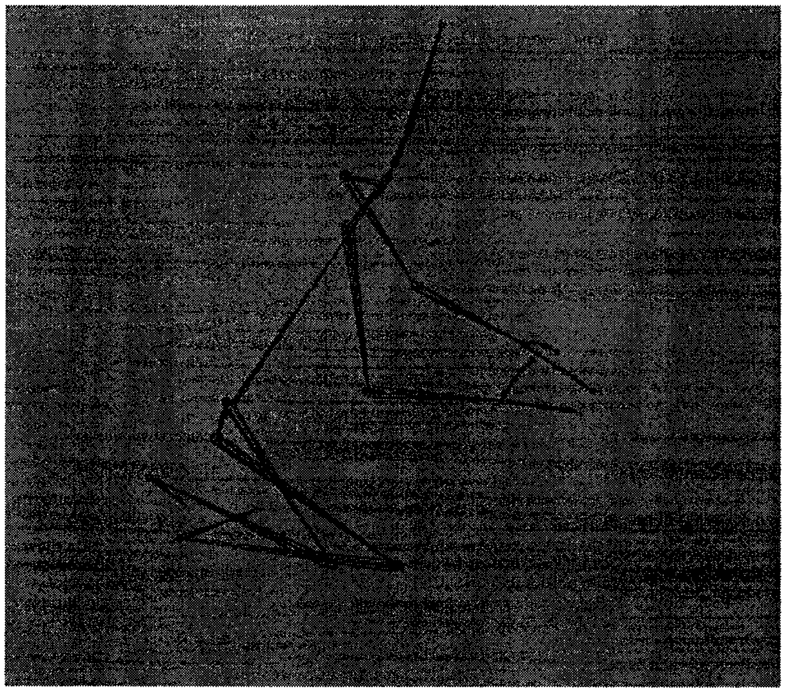


Figure 17 Faulty pose

4.2 Test Cases

4.2.1 Test Case 1

Description: Foot(R) touches the floor

Markers used: “StepMarker” is used to indicate the exact point where Foot(R) should touch the floor

Constraint(s): Foot(R)-StepMarker

Figure: 18

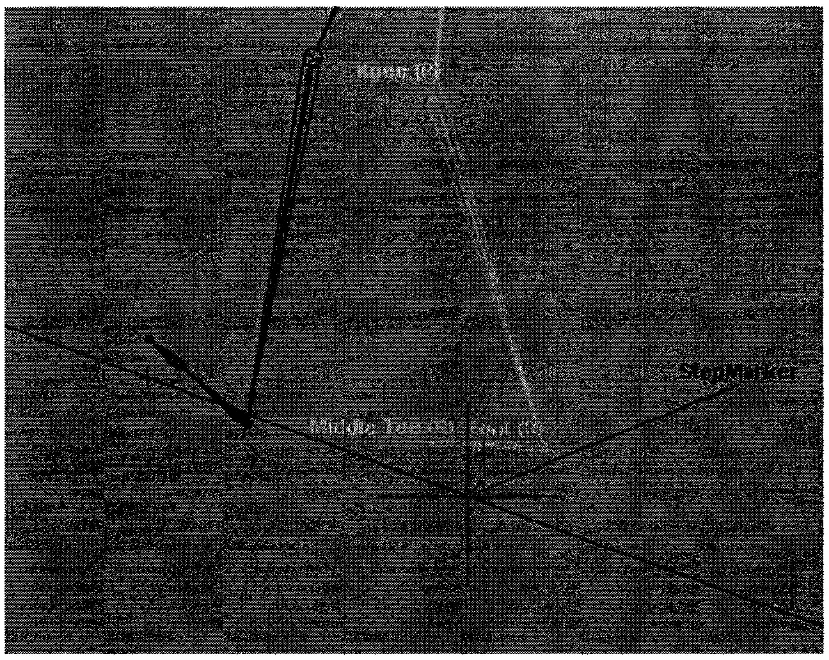


Figure 18 Test case 1

4.2.2 Test Case 2

Description: Foot(R) bottom is parallel to a vector in floor simulating standing on it

Markers used: "FloorMaker1" and "FloorMarker2" represent a vector in floor

Constraint(s): (Middle Toe(R)-Foot(R)) (dot) normalize(FloorMarker2-FloorMarker1) – length((Middle Toe(R)-Foot(R))

Note : One way to calculate the location of "FloorMarker1" and "FloorMarker2" is by projecting Foot(R) and Middle Toe(R) on the floor respectively.

Figure: 19

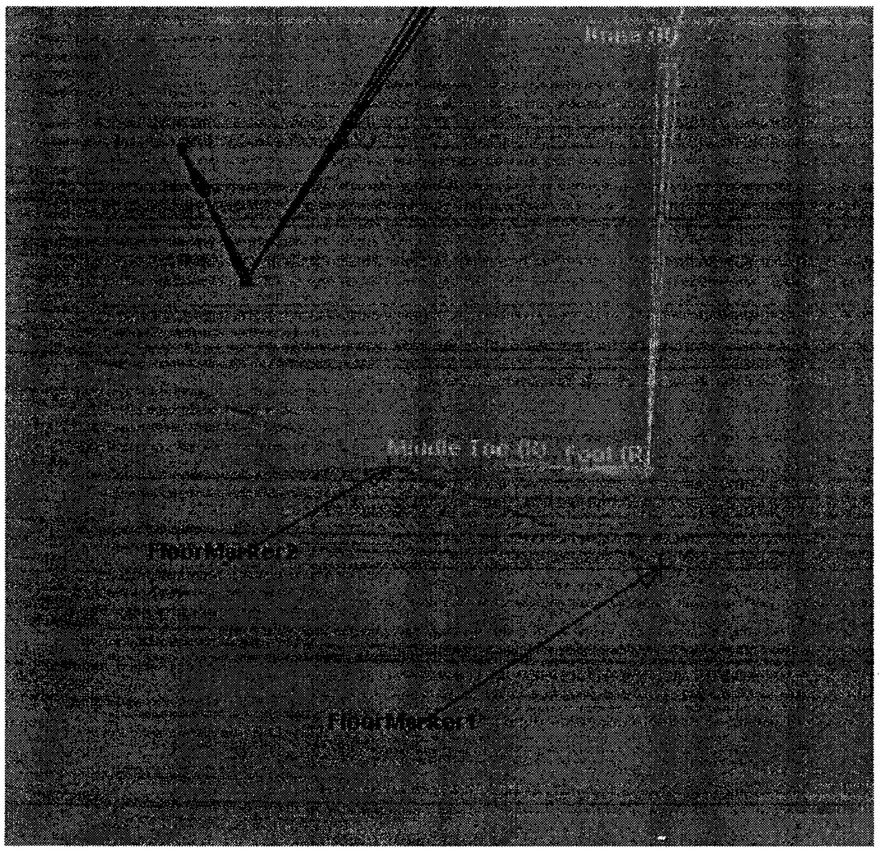


Figure 19 Test case 2

4.2.3 Test Case 3

Description: Distance between Wrist(L) and Wrist(R) equals a numerical value. This numerical value represents a width of a box that the character is holding

Markers used: “BoxMarker1” and “BoxMarker2” used to indicate the box width

Constraint(s): Distance(Wrist(L)-Wrist(R))- Distance(BoxMarker1-BoxMarker2)

Figure: 20

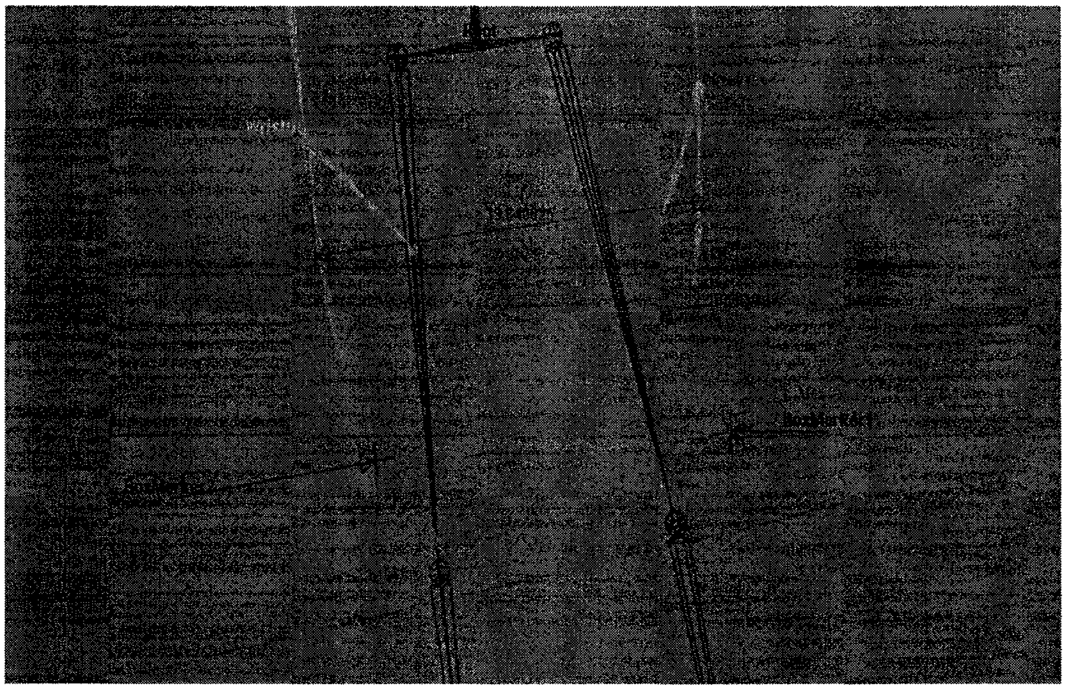


Figure 20 Test case 3

4.2.4 Test Case 4

Description: Swinging leg lands on the floor and is a 40 unit apart from the pivoting feet

Markers used: The floor is defined as a point "StepMarker" and normal vector "floorNormal1" and "floorNormal2"

Constraint(s):

Distance(Foot(L)-Foot(R))- (40^2) [Note, Distance calculate square distance]

(Foot(L)-StepMarker) Dot (FloorNormal2-FloorNormal1)

Figure: 21

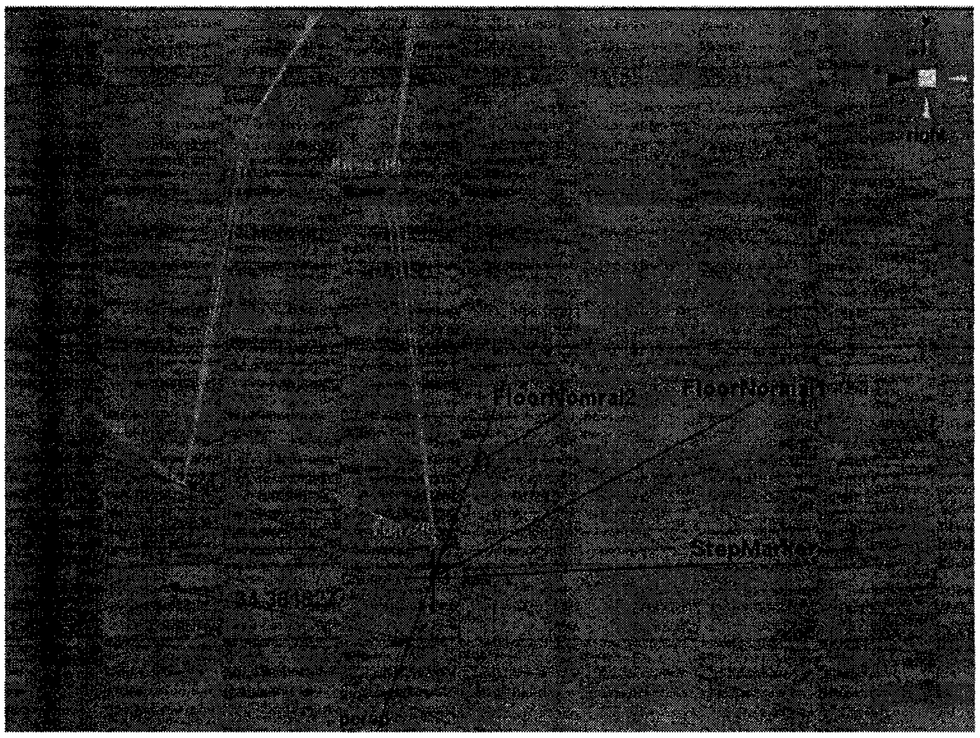


Figure 21 Test case 4

4.2.5 Test Case 5

Description: "Root" joint projection on the line connecting two feet splits this line into two equal halves

Markers used: None

Helper Equations:

AnklesMidPoint ← (AnkleL-AnkleR) * 0.5 + AnkleR

RootToAnklesMidPoint ← (Root-AnklesMidPoint)

Constraint(s): RootToAnklesMidPoint (Dot) (AnkleL-AnkleR)

Figure: 22 shows the problem, 23 shows the result

Note:

(i) It might be argued that constraint equation does translate the description section correctly. In general, there are many ways to write constraint equations to satisfy the description above, and all should give the same the result.

(ii) The constraint is intended to have "Root" joint between legs which should fix the problem seen in figure 22. After satisfying the constraint the skeleton becomes as seen in figure 23.

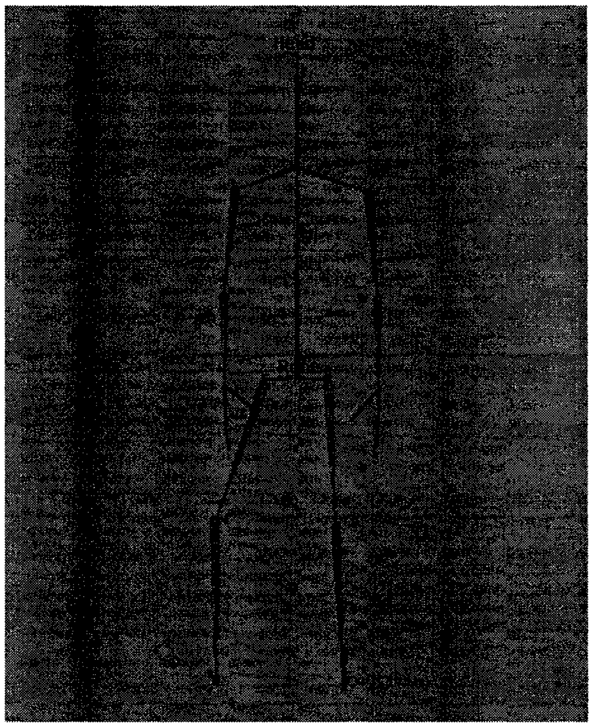


Figure 22 Test case 5 - A

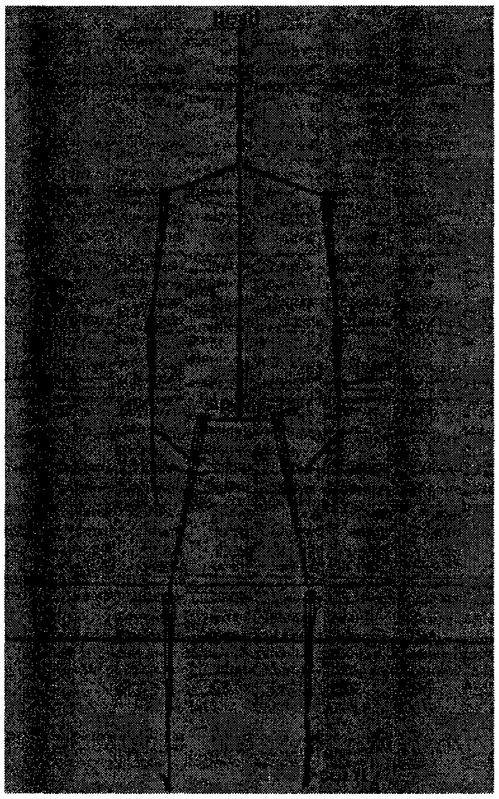


Figure 23 Test case 5 – B

4.2.6 Test Case 6

Description: Backbone, defined as (Spine-Root), is parallel to the line from “Root” joint to the middle point between two feet.

Helper Equations:

Backbone ← Spine-Root

AnklesMidPoint ← (AnkleL-AnkleR) * 0.5 + AnkleR

rootToAnklesMidPoint ← AnklesMidPoint-Root

Constraint(s):

(Backbone[0] * rootToAnklesMidPoint [1])-(Backbone[1] * rootToAnklesMidPoint [0])

(Backbone[1] * rootToAnklesMidPoint [2])-(Backbone[2] * rootToAnklesMidPoint [1])

Notes:

(i) This test case is another way to address the problem described in test case 5, however it produces mathematically satisfied yet incorrect pose as shown in figure 24.

(ii) Combining this constraint with the one described in test case 5 makes the numerical system unstable, and no solution could be found. Further research is required to determine the cause of this result; however my assumption is that the constraints combined should be satisfied, however they might create many local minimums such that it becomes hard to find the solution

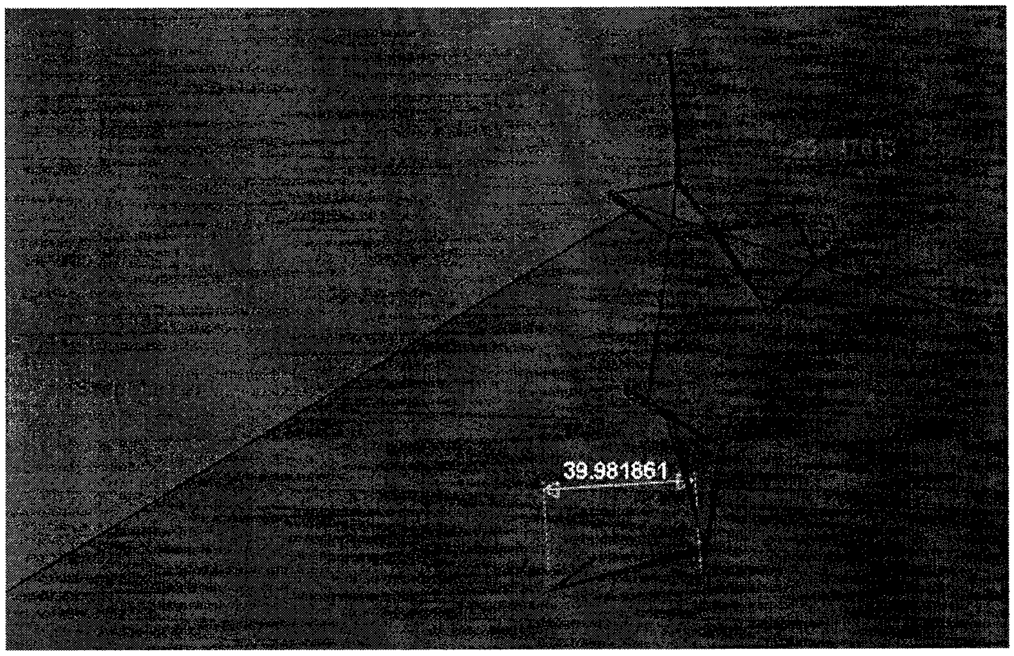


Figure 24 Test case 6

4.3 Results

4.3.1 Numerical Solver Issues and Workarounds

In general, the numerical solver performance is satisfactory and it requires around 0.2 second to solve a system with 30 variables and 12 constraints equations. Memory requirement is proportional to the size of matrices used. For example using 32 bit floating point precision, 12 equations and 30 variables will need 12 * 30 * 4 = 1,440 byte for the Jacobian matrix which is the largest matrix required in the solver.

The most challenging problem in the numerical solver is to find the right set of constraints that describe what the user needs to realize. The following list summarizes the issues encountered during designing and using the numerical solver.

- Running the numerical solver for many iterations does not always guarantee better solutions. For example, the solver might find a solution vector that is close to the optimum solution, however in order to find a better or closer solution, it will keep running, deviates and then finds a more accurate solution that does not look natural or human-like. To solve this problem, either:
    - Create a metric that defines acceptable solutions, or human poses, and use it to select a good solution. For example, the metric might include conditions like: angle between two feet should be between -45 to 45 degree.

  - Create multiple solutions and ask a human animator to choose the best one
  - Impose more constraints to eliminate unacceptable solutions; however it is a complex trial and error process.

- Initial solution influences the final result. The problem with that is any irregularity or deformation in the skeleton will be magnified in the final solution. This problem is inherited from the numerical nature of the system expressed as using the initial guess to find the solution. There are two ways to solve this problem:
  - Impose more geometric constraint to eliminate any deformation in the final solution
  - Adjust the initial solution to eliminate any irregularity. However, it would be hard to pinpoint these irregularities by the naked eye, and even worse it would be hard to fix them without using precise geometric processing.

- Determining the Degree of Freedom for each joint might not be a straight forward. The user of the system must be aware of how many Degrees of Freedom he grants each joint; otherwise the result might be unpredictable. For example:
  - My basic assumption was to give the root joint six degrees of freedom, however testing this setup shows that since the root joint orients the whole skeleton, the pose generated might be rotated in uncommon way. The case is specific to my skeleton and might not a problem on other skeletons. Even worse, if the root joint has no degree of freedom, the solution will

not be found although it is possible. After testing, I realized that the best configuration for the root joint is to have three translation degree of freedom

    - Some joints are structural joints, meaning they are not intended to move or rotate. For example hips joints have constant relationship between the root joint and they should not move.

- Solving geometric constraint equations simultaneously means that all equations have to be in the same transformation space, e.g. world space, which increases the complexity of the equations. For example if a geometric constraint is required between elbow and wrist joint, the constraint has be described with respect to root joint space, although the same constraint will be easier written and solved if written with respect elbow joint space. Possible solutions are:
    - Write an algorithm to determine independent sets of constraints, for example the algorithm will process the input constraint equations and deduce that there are two independent sets A and B: A has 5 equations and B has 6 equations. However, writing this algorithm will be tricky as equations are non-linear, and for each initial pose there might be different set of independent equations.
    - Rely on the user to create the sets of independent equations

- Finding the right set of geometric constraints that capture what the user wants to express is a very challenging process. The main reason is that number of constraints is usually less than number of degree of freedom of the skeleton, which means that there is endless number of possible solutions. For example, test case 5 which is used to keep the root joint between the two feet produced the following pose as shown in figure 25:

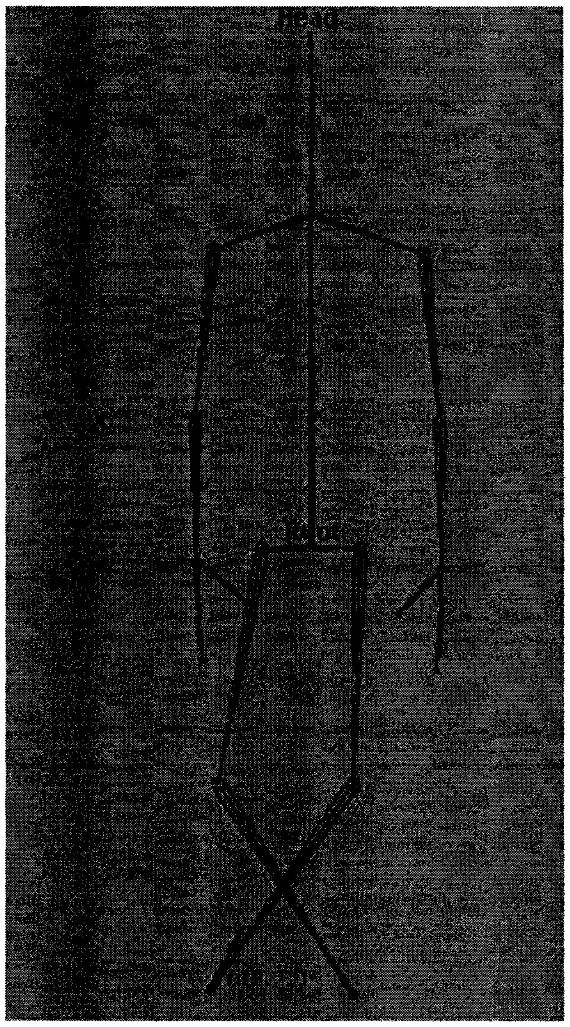


Figure 25 Unexpected pose

In general, since the constraints need to be generic, solution domain is versatile and the user might get an unexpected result. The best way to overcome this problem is to try different constraints and develop a sense of which group will get the required result.

- False solutions. The numerical system might oscillate due to local minimum before it finds the root of the equation as shown figure 26. The problem manifested itself as one leg is swinging between two different locations; however after around 24 iterations the system finds the real solution. To solve this problem, we might:
    - Monitor F(X) and if oscillation is detected we can reposition X away from the oscillation area and try again. Also we should use a heuristic function that records where X experiences oscillations and avoid these areas.

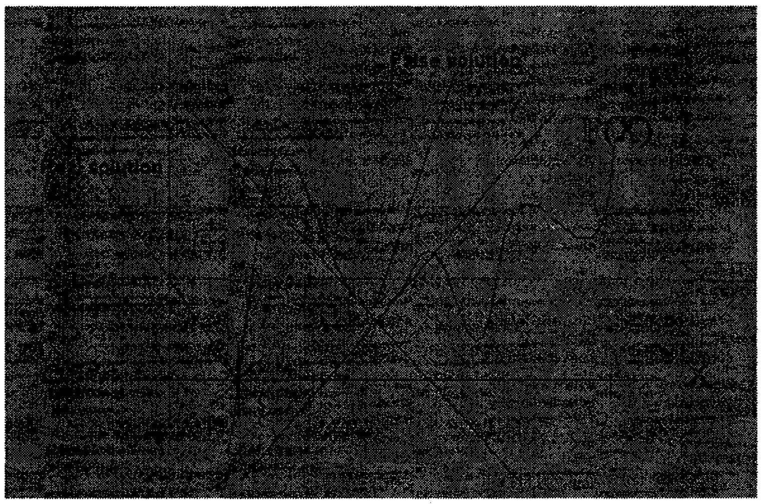


Figure 26 False solution and local minimum

- Floating point rounding errors. Initially the expressions created contained many constant terms with order less 10^{-5} which caused lots of unnecessary operations. The existence of these numbers depends on how the hardware represents floating point numbers internally. Further testing revealed that multiplying a simple expression like

1.0 * 0.0 might create such results. To solve this problem, a check to eliminate any constant values that have values less than 10^{-5} is implemented.

4.3.2 Symbolic Engine Issues and Workarounds

After combining the six geometric constraints and limiting kinematics chains to only 4 consecutive joints, the application required around 2 minutes on Pentium dual core 2.0 GHz and 1.5 GB to finish building the symbolic Jacobian matrix corresponding to the input equations. Even worse, if the kinematics chains have 5 joints, the system runs out of memory.

The first step to solve this problem is to find its causes. In general, most of the issues are related to simplifying and storing the mathematical expressions represented by binary trees. More research is required to find a better way to simplify and store binary trees in memory. The following summarizes the problems related to the way I implemented and used binary trees:

- Excessive number of constants that can be aggregated as shown in figure 27. The problem arises as transformation matrices contain many zeros and ones by nature.

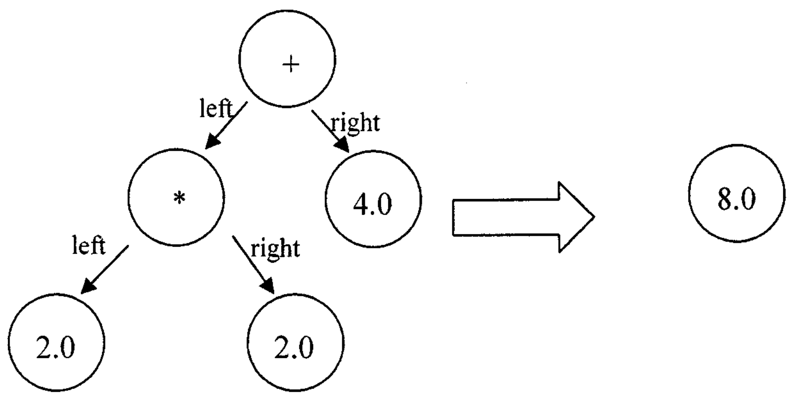


Figure 27 Expression binary tree simplification - A

To solve this problem, expressions operators are modified as the following:

Constant_1 + Constant_2 → Constant_3

Constant_1 * Constant_2 → Constant_3

Zero + Expression_1 → Expression_1

Zero * Expression_1 → Zero

One * Expression_1 → Expression_1

However, the above modification does not simplify a binary tree that has a variable as one of its operand. For example the tree on the left hand side of figure 28 has symbol x of type *variable* and hence will not be reduced to the tree shown on the right hand side.

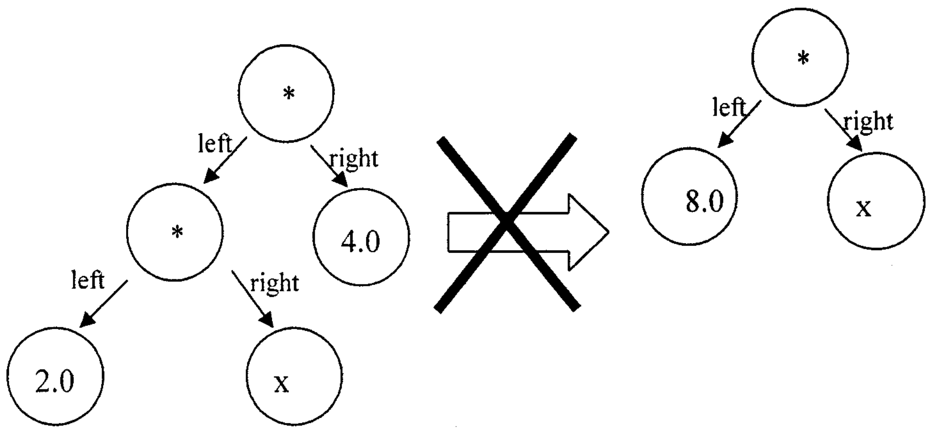


Figure 28 Expression binary tree simplification - B

- Excessive number of recursive calls due to using unbalanced binary trees. The problem is related to operator precedence in C++ which affects how expressions are being built. In C++ if all operators have the same precedence, then the compiler will execute them from left to right. For example, assume A, B, C, D and F are of type expression class. If we construct another expression G=A + B + C + D + F, then G will have a binary tree similar to figure 29:

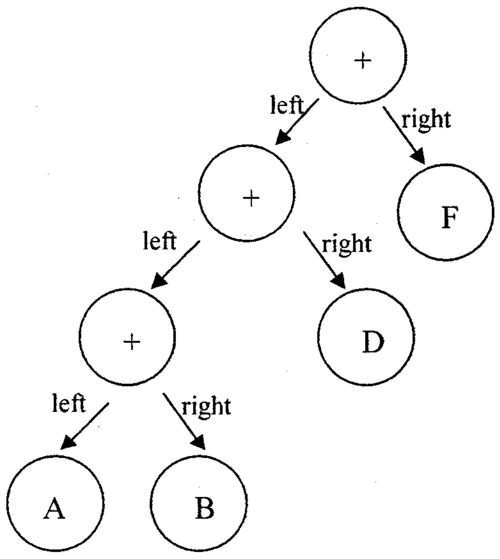
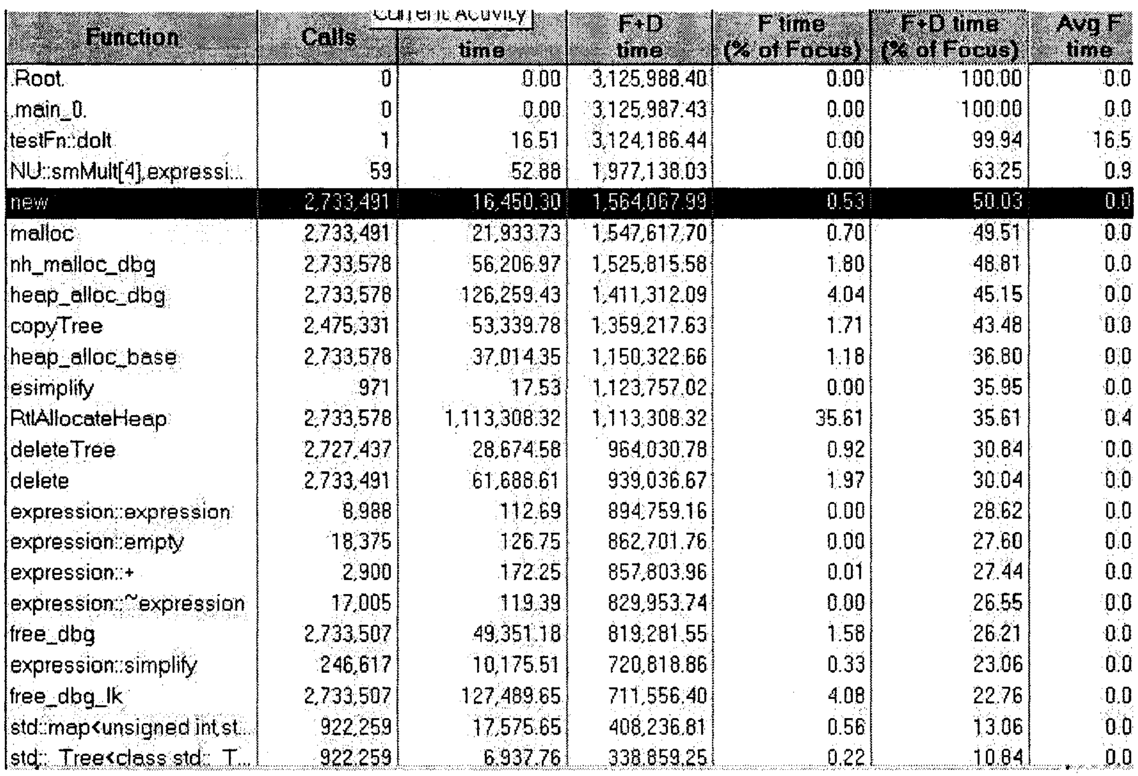


Figure 29 Unbalanced binary tree

In the unbalanced tree above, each level has two nodes, one operand and one operator — the only exception is root and leaf levels. This means that for *n* nodes we need *n* level. If the tree is balanced, then for *n* nodes we will need approximately *ln(n)/ln(2)* level which should reduce the depth of the tree significantly. More research is required to find an efficient way to balance the trees used.

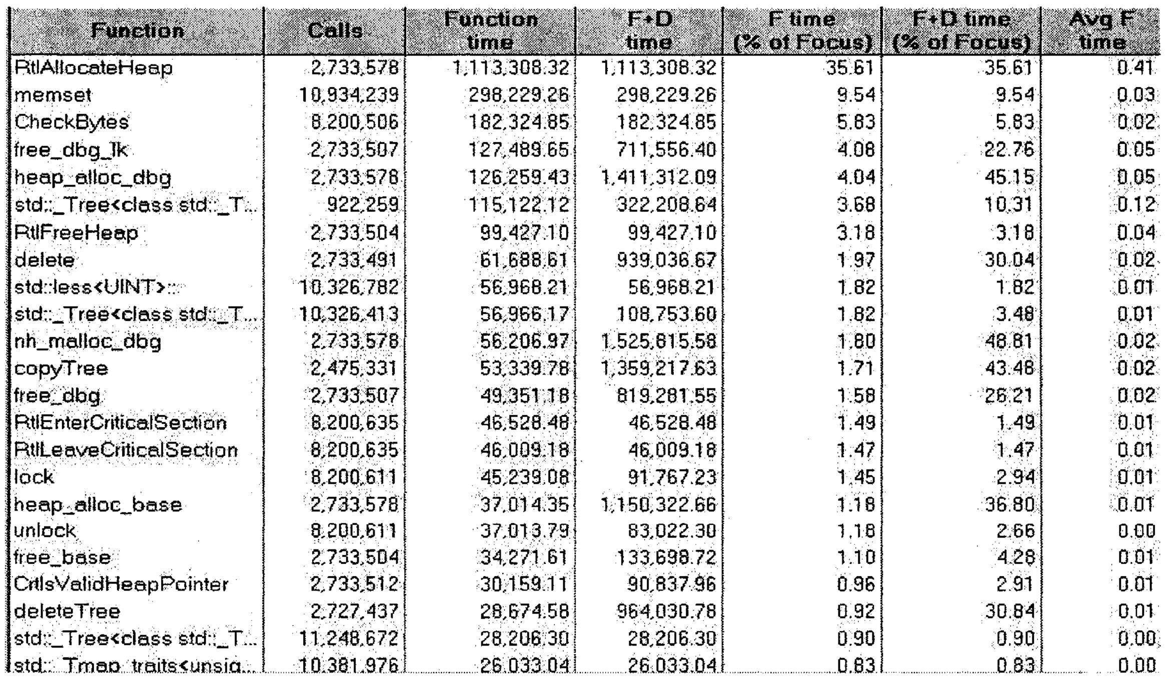
- Excessive number of dynamic memory allocation and deallocation due to allocating and deallocting nodes individually upon request. A better solution will be allocating a chunk of memory, however then another algorithm to manage the memory will be required.

To have a rough estimation on the cost of dynamic memory allocation and deallocation, IBM *quantify* profiler has been used to measure the amount of time spent on memory functions as shown in table 2 which is sorted by the percentage of time used by a function and its descendants, and table 3 which is sorted by the percentage of time used by the function only. From table 2, we can observe that C++ *new* function consumes 50% of the program execution time



Function	Calls	Current Activity time	F+D time	F time (% of Focus)	F+D time (% of Focus)	Avg F time
.Root	0	0.00	3,125,988.40	0.00	100.00	0.0
.main_0.	0	0.00	3,125,987.43	0.00	100.00	0.0
testFn::doIt	1	16.51	3,124,186.44	0.00	99.94	16.5
NU::smMult[4],expressi...	59	52.88	1,977,138.03	0.00	63.25	0.9
new	2,733,491	16,450.30	1,564,067.99	0.53	50.03	0.0
malloc	2,733,491	21,933.73	1,547,617.70	0.70	49.51	0.0
nh_malloc_dbg	2,733,578	56,206.97	1,525,815.58	1.80	48.81	0.0
heap_alloc_dbg	2,733,578	126,259.43	1,411,312.09	4.04	45.15	0.0
copyTree	2,475,331	53,339.78	1,359,217.63	1.71	43.48	0.0
heap_alloc_base	2,733,578	37,014.35	1,150,322.66	1.18	36.80	0.0
esimplify	971	17.53	1,123,757.02	0.00	35.95	0.0
RtlAllocateHeap	2,733,578	1,113,308.32	1,113,308.32	35.61	35.61	0.4
deleteTree	2,727,437	28,674.58	964,030.78	0.92	30.84	0.0
delete	2,733,491	61,688.61	939,036.67	1.97	30.04	0.0
expression::expression	8,988	112.69	894,759.16	0.00	28.62	0.0
expression::empty	18,375	126.75	862,701.76	0.00	27.60	0.0
expression::+	2,900	172.25	857,803.96	0.01	27.44	0.0
expression::~expression	17,005	119.39	829,953.74	0.00	26.55	0.0
free_dbg	2,733,507	49,351.18	819,281.55	1.58	26.21	0.0
expression::simplify	246,617	10,175.51	720,818.86	0.33	23.06	0.0
free_dbg_lk	2,733,507	127,489.65	711,556.40	4.08	22.76	0.0
std::map<unsigned int,st...	922,259	17,575.65	408,236.81	0.56	13.06	0.0
std::_Tree<class std::_T...	922,259	6,937.76	338,859.25	0.22	10.84	0.0

Table 2 System performance sorted by F+D %



Function	Calls	Function time	F+D time	F time (% of Focus)	F+D time (% of Focus)	Avg F time
RtlAllocateHeap	2,733,578	1,113,308.32	1,113,308.32	35.61	35.61	0.41
memset	10,934,239	298,229.26	298,229.26	9.54	9.54	0.03
CheckBytes	8,200,506	182,324.85	182,324.85	5.83	5.83	0.02
free_dbg_lk	2,733,507	127,489.65	711,556.40	4.08	22.76	0.05
heap_alloc_dbg	2,733,578	126,259.43	1,411,312.09	4.04	45.15	0.05
std::_Tree<class std::_T...	922,259	115,122.12	322,208.64	3.68	10.31	0.12
RtlFreeHeap	2,733,504	99,427.10	99,427.10	3.18	3.18	0.04
delete	2,733,491	61,688.61	939,036.67	1.97	30.04	0.02
std::less<UINT>::	10,326,782	56,968.21	56,968.21	1.82	1.82	0.01
std::_Tree<class std::_T...	10,326,413	56,966.17	108,753.60	1.82	3.48	0.01
nh_malloc_dbg	2,733,578	56,206.97	1,525,815.58	1.80	48.81	0.02
copyTree	2,475,331	53,339.78	1,359,217.63	1.71	43.48	0.02
free_dbg	2,733,507	49,351.18	819,281.55	1.58	26.21	0.02
RtlEnterCriticalSection	8,200,635	46,528.48	46,528.48	1.49	1.49	0.01
RtlLeaveCriticalSection	8,200,635	46,009.18	46,009.18	1.47	1.47	0.01
lock	8,200,611	45,239.08	91,767.23	1.45	2.94	0.01
heap_alloc_base	2,733,578	37,014.35	1,150,322.66	1.18	36.80	0.01
unlock	8,200,611	37,013.79	83,022.30	1.18	2.66	0.00
free_base	2,733,504	34,271.61	133,698.72	1.10	4.28	0.01
CrtIsValidHeapPointer	2,733,512	30,159.11	90,837.96	0.96	2.91	0.01
deleteTree	2,727,437	28,674.58	964,030.78	0.92	30.84	0.01
std::_Tree<class std::_T...	11,248,672	28,206.30	28,206.30	0.90	0.90	0.00
std::_Tmap_traits<unsig...	10,381,976	26,033.04	26,033.04	0.83	0.83	0.00

Table 3 System performance sorted by F%

- In order to reduce the effect of the above problems, I enhanced the symbolic engine in two different ways:
    - Modified expression operators, e.g. + and *, to simplify the expression if it contains zero or one as mentioned earlier in section 4.3.2 on page 59.
    - Implemented *simplify* function which evaluates the tree and tries to remove any expressions involving negligible numbers generated due to rounding errors. Rounding errors problem has been discussed in the numerical solver section, i.e. section 4.3.1, and it is summarized as having many negligible numbers, order of 10^{-5}, that impact the symbolic engine performance. The threshold for these the negligible number is chosen as 10^{-5} based on observing the numbers generated by the system.

      The idea behind *simplify* is to parse the tree and propagate any constants up the tree. The following algorithm explains how simplify works:

```
Simplify(node)

    If node->left,  simplify(node->left)
    If node->right, simplify(node->right)

    If node->operator equals cosine or sine, and
      Node->left is constant, //note: unary operator operand store in left
    Then evaluate cosine or sine and return the value
```

```
If node->operator equals multiplication, and
   Node->left or node->right equal zero
Then return zero


If node->operator equals multiplication,  and
   Node->left or node->right equal one
Then return none-one node


If node->operator equals addition, and
  node->Left or node->right operand equals zero,
then return the non-zero node
```

After implementing the above two enhancement, the same system of equations takes approximately 12 seconds and around 15 MB to build the Jacobian matrix as opposed to 120 seconds and 1.5 GB.

Chapter Five - Conclusion and Future Work

5.1 Summary and Conclusion

This research studies the foundations of character animation and suggests that current animation techniques representation of animation is limiting. The limitation is due to storing spatial translation and rotation and ignoring other important aspects such as apparent geometric constraints dedicated by animated characters' interaction with their environment and with themselves.

To overcome this limitation, spatial geometric constraints are used to encode more information and describe character poses. Augmenting this information helps reuse and reconstruct poses for different characters and environments.

However, we must state that the purpose of this system is not to synthesis a pose from scratch; instead it adjusts a close pose to meet new characters' dimension or different environment.

The system succeeds to achieve its purpose, however it comes with limitations. The obvious limitation is that it requires an initial solution, i.e. initial pose, otherwise the numerical solution might not converge or converge but produce a not human-like pose. Performance still needs to be addressed in this system in order to build more complex constraints and minimize the time and memory required.

5.2 Future work

- Improve the performance of the symbolic engine. This includes:
    - Reevaluating the current way binary trees are implemented to minimize the number of dynamic memory allocation and deallocation and the depth of the trees created.
    - Simplify expressions built such that constants are evaluated and combined together, instead of being spread over the tree.
    - Implement a smart simplify function that takes into account mathematical roles, for example $x * x * x$ becomes x^3, or trigonometry identities.

- The numerical solve could be improved as the following:
    - Integrate numerical optimization algorithms into the numerical solver. Currently the solver finds the root of simultaneous non-linear equations which help describe certain amount of geometric constraints. However, it will be helpful if we can ask the solver to maximize or minimize a vector, for example a standing pose could be described by pinning feet to floor and maximizing the head height.
    - Allow variables to be in a certain range, or define the domain of the problem. For example, it will be helpful to state that x should be in [10,20] range.
    - Add weights to constraints; however this will be useful if we solve over-constrainted problem which is hard to encounter in character animation. For example, for the simple skeleton we have in this thesis, we have 30

variables which means that the user needs to specify more 30 constraints to make the system over-constrained which is not probable.

- Create a graphical user interface that can be used to describe the geometric constraints visually
- Introduce constraints between consecutive poses, or even on the speed or acceleration of the skeleton.

References

1. Parviz E. Nikravesh, "Computer-Aided Analysis of Mechanical Systems", 1988 by Prentice Hall.

2. Richard Williams, "The Animator's Survival Kit, A manual of methods, principles and formulas for classical, computer, games, stop motion and internet animators", 2001 by Faber and Faber (FF).

3. Kyle Clark, "Inspired 3D Character Animation", 2002 by Premier Press.

4. Jianhui Zhao, Ling Li and Kwoh Chee Keong, "Human Posture Reconstruction and Animation From Monocular Images Based On Genetic Algorithms", International Journal of Image and Graphics Vol. 5, No. 2 (2005) 371–395.

5. Edmund Y. S. Chao, "Graphic-based musculoskeletal model for biomechanical analyses and animation", Medical Engineering & Physics Volume 25, Issue 3 , April 2003, pp. 201-212.

6. Petros Faloutsos, Michiel van de Panne and Demetri Terzopoulos, "Composable Controllers for Physics-Based Character Animation", ACM SIGGRAPH 2001, pp. 12-17 August 2001.

7. Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey Hinton, "NeuroAnimator: fast neural network emulation and control of physics-based models", Proceedings of SIGGRAPH 98 (Orlando, FL, July19–24, 1998), In Computer Graphics Proceedings, Annual Conference Series, 1998, ACM SIGGRAPH, pp. 9–20

8. Mira Dontcheva, Gary Yngve, Zoran Popović, "Layered acting for character animation", ACM Transactions on Graphics (TOG), ACM SIGGRAPH 2003 Papers SIGGRAPH '03, pp. 409-416.

9. C. Karen Liu and Zoran Popović, "Synthesis of complex dynamic character motion from simple animations", ACM Transactions on Graphics (TOG), Proceedings of the 29th annual conference on Computer graphics and interactive techniques SIGGRAPH '02, Volume 21 Issue 3.

10. Chen Mao, Sheng Feng Qin, and David K. Wright, "Computer graphics: Sketching-out virtual humans: from 2D storyboarding to immediate 3D character animation", Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology ACE '06.

11. C. M. Hoffmann and P. J. Vermeer, "A SPATIAL CONSTRAINT PROBLEM" Computational kinematics '95, Proceedings of the 2nd Workshop, Sophia Antipolis, France; NETHERLANDS; 4-6 Sept. 1995. pp. 83-92. 1995

12. Glenn A. Kramer, "Solving Geometric Constraints Systems", MIT Press, 1992.

13. Hiroshi Hosobe, "A Geometric Constraint Library for 3D Graphical Applications", ACM International Conference Proceeding Series; Vol. 24, Proceedings of the 2nd international symposium on Smart graphics, pp. 94 – 101.

14. Robert J. Schilling, "Fundamentals of Robotics Analysis and Control", Prentice Hall, 1st edition (January 2, 1990).

15. Margareta Nordin, Victor Hirsch Frankel, "Basic Biomechanics of the Musculoskeletal System", Lippincott Williams & Wilkins Press (2001).

16. Peter M. McGinnis, "Biomechanics of Sport and Exercise", Second Edition, State University of New York Press (2005).

17. B. Bruderlin, D. Roller, "Geometric Constraint Solving and Applications", Springer Germany (1998).

18. Peter A. Stark, "Introduction to Numerical Methods", Queensborough Community College of the City University of New York Press (1970).

19. Winnie Tsang: "Helping Hand: An Anatomically Accurate Inverse Dynamics Solution For Unconstrained Hand Motion", Symposium on Computer Animation archive

Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation, pp. 319 – 328.