Real-time full-body motion capture in Virtual Worlds

Final Project

Daan Nusman

June 28, 2006

Study: Computer Science, Human-Media Interaction group, University of Twente

Supervisors:
Dr. Ir. Job Zwiers, Human Machine Interaction group, University of Twente
Dr. Ir. Herman van der Kooij, Biomechanical Engineering group, University of Twente
Ir. Per Slycke, Chief Technical Officer, Xsens Motion Technologies

Abstract
This report details the integration of real-time motion-capture into VR physics-enabled environments. This report also treats of the basics of real-time rigid body dynamics, how these dynamics are used to integrate the motion capture into VR, and how to increase the stability of the physics simulation.

The report describes some points of the research and implementation that went into creating Lumo Scenario, a networked VR environment, developed at re-lion, that is used as a frame to run the motion capture simulation in.

The motion capture integration techniques are applied in particular to the kinematics generated by motion-capturing a full human body using inertial sensors.

A high-level overview is given of a demonstration application that shows off these technologies.

Table of Contents

1 Introduction

This is the report of my final project (thesis) of the Computer Science study of the University of Twente. It documents the research I have done for re-lion, a company active in the field of VR.

This research pertained the creation of a virtual, physics-enabled, multi-user, fully scripted virtual environment, and the integration, using rigid body dynamics, of motion-captured full-body avatars into this environment.

1.1 About the parties involved

Re-lion, formerly known as Keep IT Simple Software, is a high-tech company located in Enschede. It provides contract programming services, products and advice, mostly in the area of 3D graphics and Virtual Reality. I am a co-owner of re-lion.

Two of the re-lion products used in this project are Lumo, a 3D graphics engine, and Lumo Scenario, a networked dynamics product still in development and due for release in 2006.

Throughout the project, development versions of the Xsens motion capture system were used. Xsens, a company also located in Enschede, manufactures high-precision inertial measurement sensors and software. Ir. Per Slycke has supervised the project on behalf of Xsens.

Parts of the Dismounted Trainer software were commissioned by TNO Defense & Safety.

Parts of the Lumo Scenario software were developed during the Scomosi project, a scoot mobile driving simulator using the Lumo Scenario software as basis. The Scomosi project was a joint project by re-lion, Roessingh' R&D, and the University of Twente.

Dr. Ir. Job Zwiers was a supervisor of the project on behalf of the Human-Media Interaction Group of the University of Twente. He is working on Virtual Reality research and projects for HMI.

Dr. Ir. Herman van der Kooij, assistant professor at the Biomechanical Engineering group of the University of Twente was also a supervisor.

1.2 Goals of this research

The main goal of my final project is a multi-user scriptable VR environment, with a representation of a human body being motion captured in real-time, integrated into the 3D world. This representation should be as close as possible to the actual body position, but not necessarily the same: it needs to look good and not in violation of the physical rules of the virtual universe. These physical rules are dictated by a physics engine, in this case ODE (Open Dynamics Engine). An important aspect is how much of the physics engine used to run in the VR environment we can use to aid in integrating the motion capture into the VR world.

2 Overview

The main goal is to create a virtual multi-user environment, enabled with physics and scripts, with integrated full-body motion capture functionality.

First the current full-body motion hardware is reviewed (section 3.1.1). Because of the promising nature of inertial motion capture, and the availability of a pre-production version of an Xsens motion capture suit, **inertial motion capture** was chosen. The advantages of inertial motion capture compared to many other motion capture techniques are: 3DOF orientation capture (meaning all rotation axes are captured), precise captures, portable and low-power sensors. Of course, some brands of inertial sensors are more precise, portable, etc than others. The Xsens sensors are also wireless. A disadvantage of inertial motion capture is that only orientation can be reliably captured.

Next, the virtual world the avatar operates in is defined. This is a world that exists only in the state of a physics engine, on a single machine. This machine is called the **server**. The objects in this mathematically described world all have real-life equivalent properties, such as a position, speed, mass, center-of-mass, orientation, angular velocity, and a clearly defined shape. One can imagine that these objects can interact with each other, for example a sphere lying on the floor or stack of boxes collapsing in on each other. This is called an **interactive simulation**. 'Interactive' because a user can interact with the objects (for example, using a motion capture suit).

The software that makes all this possible is called a physics engine, or dynamics engine (chapter 5). The terms 'physics' and 'dynamics' are used interchangeably in this report. We assume all objects interacting with each other on the server are all rigid. Therefore, we simulate **rigid body dynamics**.

In this case, **ODE (Open Dynamics Engine)** was used. Is this because the engine is open-source, making it easier to tweak and understand. Also, it has a proven track record of many diverse applications.

The problems pervasive in dynamics engines are threefold. First, the simulation can be unstable, causing it to 'explode', meaning the objects fly off to infinity. Causes for this are too small time steps, too large forces being exerted on objects, using high-friction surfaces or mixing very small and very large masses in the same simulation. A second problem in dynamics engines is that the simulation might be too slow to use in interactive environments. Finally, a more insidious problem can be the sheer number of parameters to tweak. For example, a simulated car has many properties: the mass of the chassis and wheels, four joints keeping the wheels in place (each joint has a large number of parameters), controllers (motors) that drive the engine, the amount of down-force to generate at what speeds, the tire friction in the driving direction and in the tangential direction, air resistance, brake strength, etc. All these parameters have complex relationships can be make tweaking parameters a black art.

Now that all objects can interact with each other in a way that makes sense physically, we need to control what happens on a higher level. For example, what objects are created, what environments are loaded, how do objects respond to input, etc. For this a **scripting language** was chosen, in this case **Lua**. While integrating the physics and other systems successfully into a scripting engine is a considerable task in itself, it is not further discussed in this report.

The next thing to do in the server is creating a physical object that represents a human body (section . After all, the motion capture data acquired is from a human body. Simulating human bodies in physics engines has become commonplace in the games market for some years now. This is usually referred to as **ragdoll dynamics**. A set of limb-like objects is created in the physics engine and attached to each other with ball joints or hinge joints. Because no controlling forces are exerted, the system will collapse in a ragdoll-like fashion. This is often used to simulate enemies in games getting killed and falling down stairs, etc.

While sending limp ragdolls down sets of stairs is certainly a lot of fun, the next step is to **actuate** the ragdoll with the motion capture data. This means having the limbs of the physics engine roughly take the orientation of the motion capture data. This can be done in several ways. First, it is possible to directly set each limb in the correct orientation. The major downside of this approach is that it mostly disables natural physics interaction with the ragdoll and the rest of the environment. Second, another way is applying

forces to each limb to have it assume the desired position. While this works, stability problems arise from the fact that we are actually modeling springs to keep the limbs in place. A third option is to use motor controllers, this is called an 'angular motor' in ODE physics terms. However, there are still stability and usability issues left.

Another important aspect to the simulation system is the requirement that is should be **multi-user**. This is because professional simulations rarely use one rendering station only, and many simulations require the involvement of multiple persons. To reiterate, the server runs the physics simulation and scripting that controls the physics and the flow of the simulation. We still cannot actually see what is going on the the server. We need one or more **clients** for that. Each client is fed a constant stream of update packages from the server through a **network** link. It renders the positions of the objects and the static environment. It also generates the appropriate sounds and samples any input devices (such as keyboard, mouse, motion capture suit) and sends this data to the server for processing.

The server sends updates at a low frequency, 10-15 Hz depending on the simulation. This means that on the clients, the objects will jerkily move around at the same frequency. A solution to this problem is using interpolation and extrapolation to smooth movement. Some problems remain, such as objects extrapolating for too long.

Some techniques are used to reduce network bandwidth, such as **quaternion compression** in the case of motion capture data, which consists mostly of quaternions. Quaternions are a non-commutative extension of complex numbers and can, in unit form, describe three-dimensional rotations. Multiple threads and queues are used to optimize CPU usage of transferring and receiving data.

The motion capture data is sent from the client to server, processed in the physics engine on the server, then sent back to all clients for displaying. Because of this long path, the del€ays on the client between sampling and displaying movement can be significant. A **local feedback mode** was used to alleviate this problem, at the cost of the loss of some interaction with the environment (section 7.5).

All this technology was wrapped up in a demo (chapter 9), allowing two (or more) people, wearing motion capture suits, to visually interact with each other and the environment, and showing vehicle dynamics by allowing people to drive around in a tank. It employs most of the technologies described in this report.

In conclusion, it can be said that most of the supporting framework for successfully running interactive simulations is now firmly in place. However, the ragdoll interaction with the environment needs more attention. The two main problems to battle are the latency of such a complex system, and the stability and feasibility of actuating limbs with forces or motors. It is possible to alleviate these problems by indirectly interacting with the environment (by proxy) instead of directly.

3 Current technology

3.1 Kinematics and dynamics overview

We will take a look at the current research and technology in the kinematics and dynamics fields.

3.1.1 Motion capture (kinematics) overview

Motion capture is the technology of capturing some real-life motion into a computer, for later playback or analysis. Commercial motion capture has been around for two decades. Many kinds of technologies are available:

Optical motion capture

Optical motion capture systems work with one or more cameras. Usually the subject is equipped with reflective patches or spheres (called markers), indicating the position to the cameras. The markers themselves are tracked using software. By combining the same markers on multiple cameras (which all have a different position), a 3D position of a marker can be determined. The cameras are often infra-red and mounted to a rig or to the walls of a room.

The advantages:

- High precision
- Absolute position determination
- Can cope with a high number of markers

The disadvantages:

- Multiple cameras: a set-up can have a high cost
- Fixed location
- Limited reach
- Capturing rotation of limbs can be tricky. Sometimes, marker clusters (three or more markers fixed to a small frame) are used to capture a rotation. Because the software knows the relative positions of the markers in a cluster, it can calculate the orientation of the body attached the cluster.
- It's possible that some markers are (temporarily) obscured, heuristic algorithms have to be applied to determine where the marker went and what marker maps to what limb

Some companies that develop optical motion capture solutions are:

- Vicon Peak

  http://www.vicon.com/
- Motion Analysis

  http://www.motionanalysis.com
- Adaptive Optics

  http://www.aoainc.com/technologies/adaptiveandmicrooptics/wavescope.html
- Charnwood Dynamics

  http://www.charndyn.com/Products/Products_Hardware.html

Magnetic motion capture

Electro-magnetic motion capture use sensors which operate in a low-frequency electromagnetic field. The sensors report their movement and orientation based on that field.

Advantages:

- Absolute orientation as well as position are measured

Disadvantages:

- The motion captured subject cannot be near, or contain, metal
- Fixed location
- Limited reach
- Limited number of sensors

Some companies that develop magnetic motion capture solutions are:

- Polhemus

  http://www.polhemus.com/
- Ascention Techonolgy

  http://www.ascension-tech.com/

Mechanical motion capture

Mechanical motion capture uses exo-skeletal structures to measure relative joint angles.

Advantages are:

- Precise
- Portable
- Unlimited reach

Disadvantages:

- Captures joint rotations only
- Unwieldy exo-skeletons
- Can only capture (parts of) the human body

The leading company in mechanical motion capture is Animazoo (with their Gypsy4 product) (http://www.animazoo.com/products/gypsy4.htm).

Inertial motion capture

Inertial motion capture uses gyroscopes, sometimes combined with measuring the magnetic north and the gravity vector, to measure the 3DOF orientation of a sensor.

Advantages are:

- Precise
- Very portable
- Low-power
- Captures complete orientation

Disadvantages:

- Does not capture position

Some companies that develop inertial motion capture solutions are:

- Xsens (using their own sensors to develop to full-body motion capture suit)

  http://www.xsens.com

- Intersense

  http://www.isense.com/

- Animazoo (Gypsy Gyro-18, using intersense sensors)

  http://www.animazoo.com/products/gypsyGyro.htm

3.1.2 Dynamics today

A rigid body dynamics simulation (physics engine) is a library that simulates how objects would behave, based on Newtonian physics, using variables such as mass, friction, (angular) speed and position. Physics engines usually consist of a collision detection engine and a dynamics simulation engine. The collision detection engine obviously detects inter-penetrating bodies. This data is used to generate forces on the bodies, that are resolved in the dynamics simulation step.

There are a couple of important variables regarding different implementations of physics engines:

- Performance – efficiency of algorithms and implementation
- Stability – how easy it is for simulations to arrive in incorrect states (for example, objects flying at infinite speed or being stuck in each other)
- Precision – how much detail the simulation has and how much the objects in it behave like real-world objects
- Ease of use – an easy pitfall when creating physics engines is to introduce too many user-adjustable variables. This makes the simulation very hard to tune

A *real-time* physics engine sacrifices some precision to attain interactive speed. Stability can also be 'traded in' for precision. The revolution of real-time, low precision, realistic physics in simulations and especially games, is well underway.

A good and entertaining start on Newtonian physics is [feyn], a set of lectures (in book form) by Nobel-prize winner Richard Feynman and others. The first part of the book is a sufficient introduction; the physics emulated in rigid body dynamics are not very complicated.

A good place to learn the basics of rigid body dynamics is a series of four articles called *Physics, The Next Frontier* written by Chris Hecker of Game Developer Magazine [hecker96]. The series starts off with numerical integration, moves on to two-dimensional dynamics, and finishes with an introduction to three-dimensional dynamics.

Andrew Witkin and David Baraff also have created an excellent course called *Physically Based Modeling: Principles and Practice* for Siggraph '97 [witkin97], aimed at math-challenged computer graphics specialists. The course covers ordinary differential equations, implicit and explicit integrators, constrained dynamics, and unconstrained and constrained rigid body dynamics. Baraff and Witkin are oft-quoted researchers in the field of rigid-body dynamics.

The rigid body dynamics engine used in Lumo Scenario, ODE, is described in more detail in chapter 4.

Commercial real-time physics technologies include:

- Havok Physics 3 is a popular commercial physics engine.

  http://www.havok.com/content/view/17/30/

- AGEIA PhysX Technologies also supplies a physics engine, but takes physics processing one step

further with the addition of a physics processing unit (PPU, in the same vein as the graphics processing unit, GPU). The PPU has recently become commercially available.

http://www.ageia.com/

The trend in dynamics:

- Offloading work to other processing units, such as the GPU (ATI, NVidia) or to a specialized PPU (Ageia).
- Load-balancing physics processing to accommodate dual-core processors.

3.1.3 Kinematics and dynamics combined

A recent development is blending motion capture and dynamics using controllers. A leading paper on this subject is *Hybrid Control for Interactive Character Animation* by Ari Shapiro, Fred Pighin, and Petros Faloutsos [shapiro03]. This technique, which I will refer to as **hybrid control,** implies switching between pre-recorded sequences (kinematics) and run-time simulations (dynamics). The dynamics part is augmented by different types of controllers, such as rule-based controllers and even genetics-based AI controllers. The controllers emulate how a normal person would react to different situations. For example, a rag doll could execute a prerecorded kinematics walking sequence, until it reaches a tripwire, causing the processing to switch to dynamics mode, which uses controllers to extend the arms forward to try to maintain balance, like a real human would.

A great example of such a system is NaturalMotion endorphin 2.0 [end], a "dynamic motion synthesis" software that enables you to interactively set up stages for rag doll actors to play in.

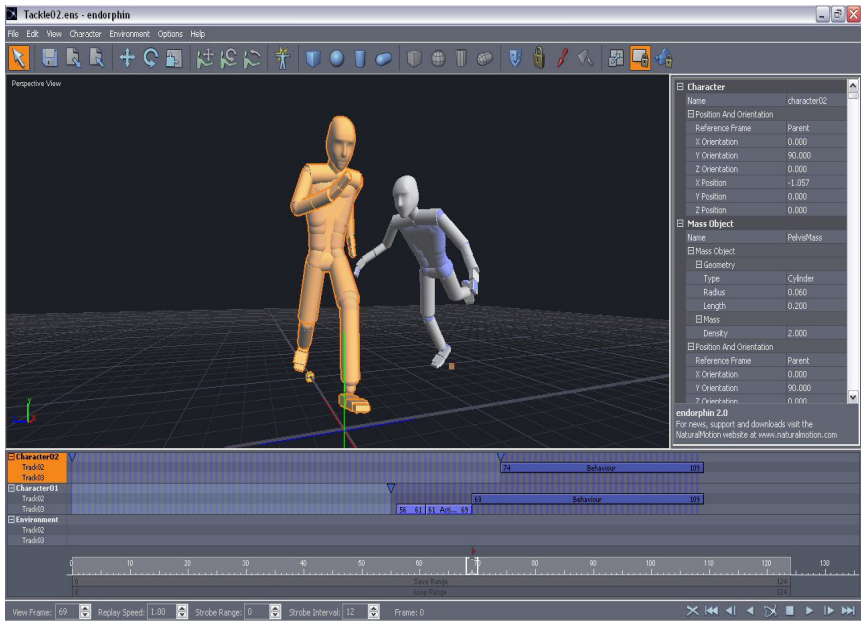


Illustration 1: NaturalMotion's endorphin 2.0 in action

It is important to note that, while hybrid control combines dynamics and kinematics, it does it in a different way than proposed in this research. Hybrid control is designed to create new behaviors using dynamics, extrapolated from post-processed motion captured or even hand-made kinematics. This in contrast to this thesis, which tries to correct raw motion capture data using dynamics.

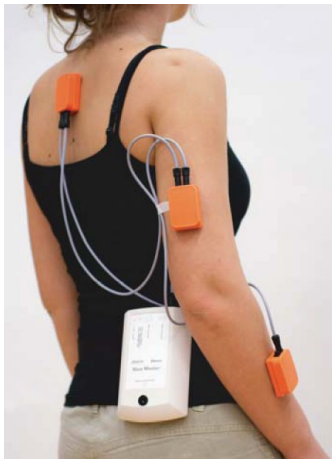
http://www.magix.ucla.edu/pacificgraphics2003/

http://www.ode.org/slides/igc02/s17.html

3.2 Existing technologies employed

While a solid theoretical basis is very important, a smoothly working framework to conduct tests and record results with is also invaluable. To this end, I have chosen to use the following technologies during this project.

3.2.1 Xsens Xbus Master system



The Xsens Xbus Master system is a portable, wireless bus system that can have up to fifteen Xsens motion trackers attached. [xsens01]

Each motion tracker can measure its own orientation in space. The reasons for choosing the MTx were:

- Re-lion already has experience with Xsens software and hardware
- Xsens is a local company, operating from the BTC-Twente, and re-lion has a good business relationship with it
- the device itself is very accurate and is suitable for real-time processing

3.2.2 Open Dynamics Engine

The Open Dynamics Engine (often referred to as ODE) is an open source rigid body dynamics library. [ode03]

It has the following features:

- Stable and fast; several types of integrators (steppers) are available
- Rigid bodies
- Advanced joint types
- Integrated collision detection
- Open source: I was able tweak the library to my liking

Using ODE has allowed me to concentrate on solving problems with a dynamics engine instead of spending most of my time creating and tweaking a dynamics engine myself.

3.2.3 Lumo SDK

The Lumo SDK is a full-blown VR visualization toolkit.

Features include:

- multi-platform: Microsoft Windows, GNU/Linux, MacOS X
- DirectX 6, 8 and 9, OpenGL renderer support
- Serializable scenegraph data structure
- Culling, resource management, etc. all done automatically
- VR-device support (such as the Xsens Xbus master system)

Lumo

The main reason I have chosen Lumo for visualization is that, of course, my own company produces the software. Another reason is that, just like using ODE, I did not have to worry about displaying worlds and avatars during the project, which allowed me to concentrate on developing the algorithms.

3.2.4 Lua

Lua is a scripting language [lua01]. Its most eye-catching features are:

- Really fast and small code, still full-featured
- Byte-code interpreted by register-based virtual machine
- Easily embeddable into existing programs
- Powerful language features
- ANSI C compliant open source software.

The Lua scripting was used to facilitate several tasks, such as loading BVH, worlds and configuration files, and creating events and dynamics controllers.

3.3 New technology developed

3.3.1 Lumo Scenario

Many of the libraries and products described above are being integrated into a new product called Lumo Scenario. Lumo Scenario is currently being developed at re-lion, mostly in tandem and sometimes as a part of my final project. It is designed to enable our customers to more easily create full-blown VR simulations. Its features will include:

- Distributed client/server architecture
- All popular VR input devices supported
- Passive and active stereo supported, active stereo on a single render station or rendering each eye on separate stations
- Multiple participants, using any kind of input/output combination
- Realistic dynamics simulation
- Full scripting support, both server-side and client-side
- Full world-building support though Lumo Editor, using ready-made building blocks
- Full integration with the Lumo 3D engine

Many of the techniques described above and developed during my final project are used in Lumo Scenario.

3.3.2 Dismounted Trainer

The dismounted trainer (DT) is a project whose first phase was developed by re-lion for TNO Defense, Security & Safety, commissioned by the Royal Dutch Army. The intent of the DT is to train soldiers for combat on foot (dismounted combat).

Users are completely immersed in their environment. They wear a HMD and motion capture suit. The HMD shows the surroundings and the virtual body of the user.

One can replay a training from the start (after-action review), from many camera positions. It is possible to record to movie files (AVI format) for on-line fixed-camera reviewing without the simulation software present.

The DT is still in a prototype phase, but future training goals include:

- Squad-based training
- Mission rehearsal
- Reconnaissance - train in a building or urban environment prior to a real operation

The dismounted trainer from a hardware point of view

For an graphical overview of all hardware involved, see diagram 1 below.

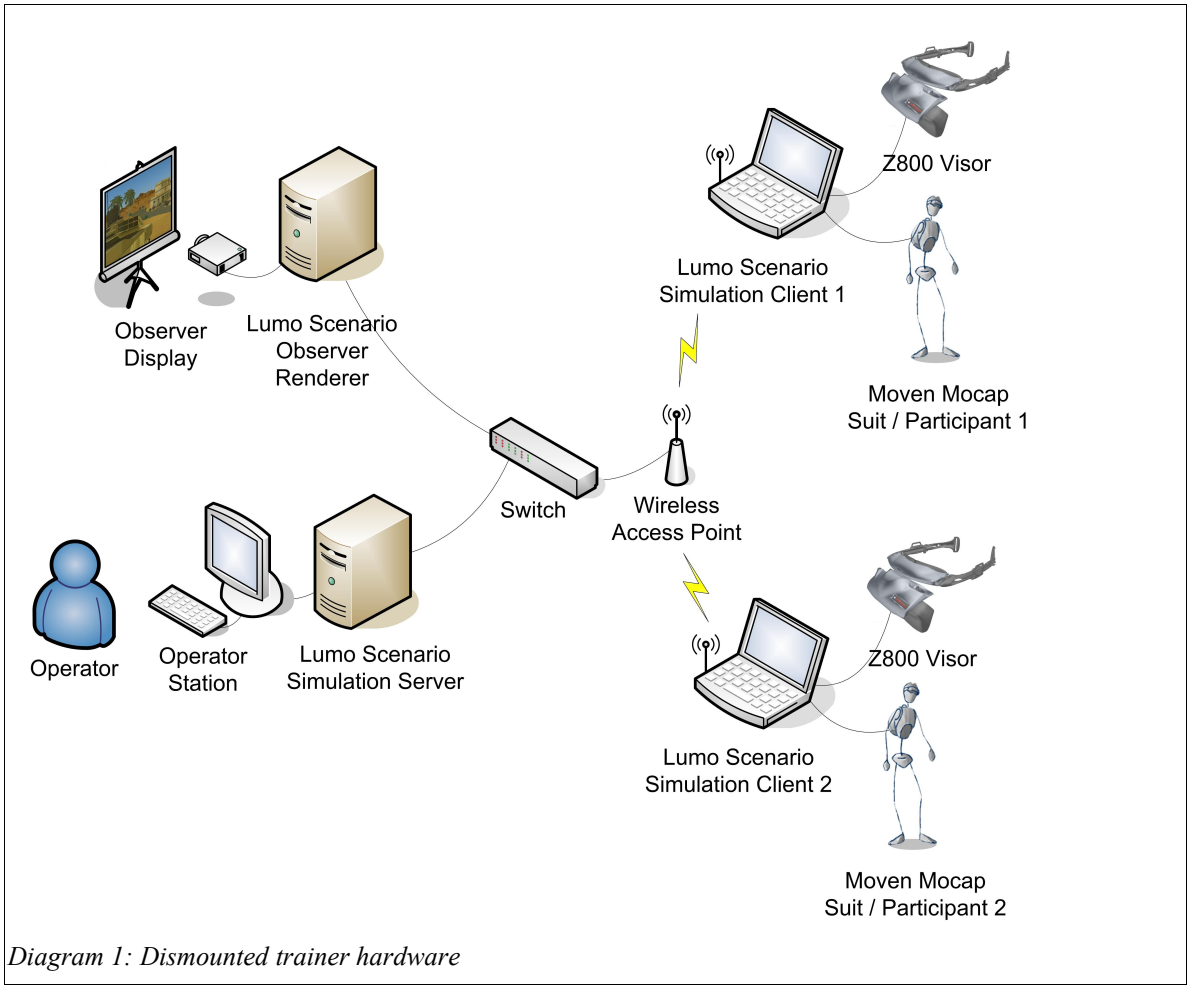


Diagram 1: Dismounted trainer hardware

Each actively participating user carries the following hardware.

- A wired Xsens motion capture suit.
- A wired head-mounted display, in this case a low-cost, light-weight eMagin Z800 visor (www.emagin.com).
- A backpack, carrying a laptop. The motion capture suit and HMD are connected to the laptop. The laptop uses a standard 802.11g wireless LAN connection to connect to a wireless Access Point. The laptops have capable real-time graphics performance (e.g., an NVidia GeForce Go or ATI X600).

Furthermore, a server computer (a standard PC) and an observer rendering computer are connected to the same network as the Access Point.

The dismounted trainer from a software point of view

From a software point of view, things look a lot simpler: see diagram 2 below for a high-level overview of separate computers (boxes), communication lines (arrows), and the database (cylinder).

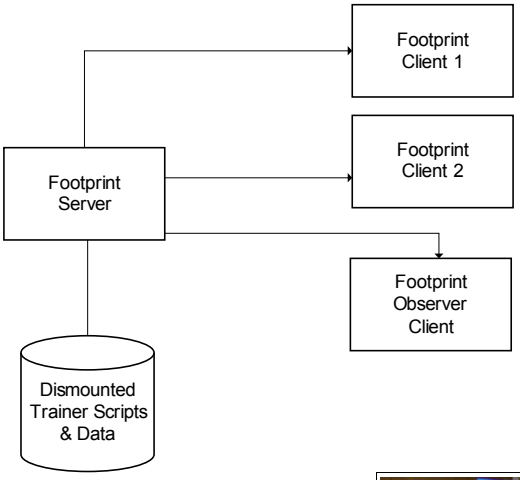


Diagram 2: Dismounted trainer softwa

The server runs the physics simulation, guided by Lua scripts. It communicates with a number of clients. All user input the clients gather are sent to the server, and the server sends the current VR world state to each client.

Each client can be a participant or an observer. The output of a client is always vision (taken care of by the Lumo 3D engine) and sound (a 3rd-party 3D sound engine), controlled by the network input. Optionally other VR output devices can be used, such as force-feedback platforms and other real-world actuators. The input for the clients are the usual input devices (keyboard and mouse), and VR input devices. In the case of the DT, the Xsens motion capture suit is the VR input device for the participating clients.

All simulation related data, such as 3D models, scripts, textures, etc., are stored in a network file share on the server. Server and clients use this file share to load data from.

Illustration 2: Real-time motion capture for the entertainment industry: Dance 4 Life festival

3.3.3 Motion capture integration

The idea is to virtually actuate a 'rag doll' simulated physics object with on- or off line motion capture data. This enables the interaction of the rag doll with its environment:

- Collision detection and response with the world – for example, will our rag doll be able to walk into a wall, or up a flight of stairs?
- Ice-skating prevention – because the sensors only measure orientation, and the root (origin) of the skeletal model (rag doll) is its pelvis or torso, the feet will not have any meaningful contact with the floor, even assuming it is flat. There are many seemingly viable solutions or workarounds to this problem:
  - Using Global Positioning System to determine the global position. This is probably not very precise. I will not pursue this technique in this thesis.
  - Using sensors in the shoes, detecting whether or not a shoe is on the ground. One can then use skeletal re-rooting or dynamics constraints (joints) to fix one or two feet to the ground. This technique looks very promising, but needs modification to the hardware.
  - Using simple position determination (linear algebra) to check what the horizontal positions of the feet are. The lowest foot is probable to be on the ground. Next, you could use the same techniques as sensors in the shoes to lock one or two feet to the ground. See section 8.4.1.
  - Using the physics engine itself: if one can keep the rag doll upright, using a pendulum weight or angular motor, the contact joints generated by the feet touching the ground might result in a realistic motion. See chapter 8.4.3.

3.4 Inertial sensor calibration and correction

This thesis is not about sensor calibration, correction or real-world precision validation. A lot of research has been done and is currently being done on this subject.

Of course, the more precise the input is, the better the quality of the final motion will be. So, rather than replacing existing sensor calibration algorithms, the algorithms described in this thesis can be applied to the output of the calibration algorithms.

3.5 Why real-time?

I have chosen to create algorithms that run in real-time, responding to real-time or recorded captured motion data. This has some important consequences.

- First, the amount of computation that can be done per frame is limited. This is why algorithms and implementations will also have to be analyzed with regard to their efficiency as well as the other criteria. However, we have found that careful programming can keep processing time well within the real-time time frame. Most of the problems arise when multiple rag dolls must be calculated, or other simulations have to be run on the same computer.
- Second, some of the more advanced algorithms could benefit from a certain amount of foresight determining what the most probable pose is. This is called 'causality'.

But the advantages are also evident.

- Most importantly for the Dismounted Trainer: the Dismounted Trainer is a real-time simulator, just like a flight simulator or other “mounted” simulator. Real-time, low-latency feedback is of vital importance to the user experience.
- We have noticed that real-time feedback saves valuable time during motion capture sessions. For example, sensors can malfunctioning or other errors can creep in. Motion capture is still a process of trial and error; the earlier the errors in acting and setup are caught, the less money and time

have to be spent on doing re-takes, or even worse manually correcting animations later on.

- It also enables live-feedback entertainment purposes. For example, re-lion has demonstrated an early version of the Xsens motion capture system and re-lion software during the “Dance 4 Life” festival, showing an Elvis (modeled by 2morrow, http://www.2morrow.nl) mimicking the dancing of someone picked from the audience in an Xsens motion capture suit: see the illustration to the right.
- In simulations and games, NPC's (non-player characters) can also be driven using the physics engine, resulting in more realistic interactions with the environment. Using 'hard' animations often results in the character walking through objects, or sticking limbs trough the floor or walls.

4 Architecture

4.1 Client/server architecture

The simulations run in a client/server network. The server runs scripting and physics, the client renders scenes and processes input.

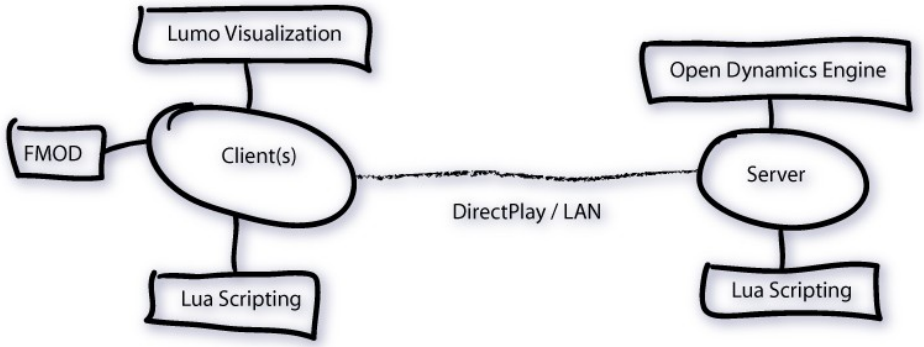


Diagram 3: Global architecture (image taken from Footprint documentation)

There are a couple of reasons for this division.

- The fixed time stepping required for stable physics requires the physics calculations to be decoupled from the rendering loop. There isn't a more drastic way of doing this than moving it to another process, optionally on another PC. More on why this is necessary in section 5.
- It enables multi-user interactive training and entertainment environments: each player has its own client station that gathers input and renders the simulation.
- It enables complex multi-display VR-setups, such as multiple projectors, CAVE VR systems [cave] or passive stereo setups: each display has its own client station that renders the simulation, and the server or a specific client station gathers input.
- Computing power can be distributed. For example, when running a single-PC client/server setup, processing the physics at the server could become too intensive. You can then move the server to another PC, drastically increasing available processing time for both server and client.
- Network feeds from the server to the client can easily be recorded. This allows sessions to be reviewed at a later time, or converted to an .mpeg or .avi movie, for example.

The big downside is, of course, the communication between the clients(s) and the server, which leads to:

- Latency and timing problems: packets can arrive too late or out of order. Packets arriving too late results in a sluggish simulation.
- Bandwidth and flow control problems: the data stream from the server to client can become too great for the client or connection to handle.
- Complexity of code: managing the synchronization of states is a difficult job, involving a lot of timing issues and network messages. This complicates development considerably.

More information on the network issues in chapter 5.

4.2 Physics world vs. mesh world

Lumo Scenario has three visualization 'worlds' you can turn on or off. Each world has its own special uses.

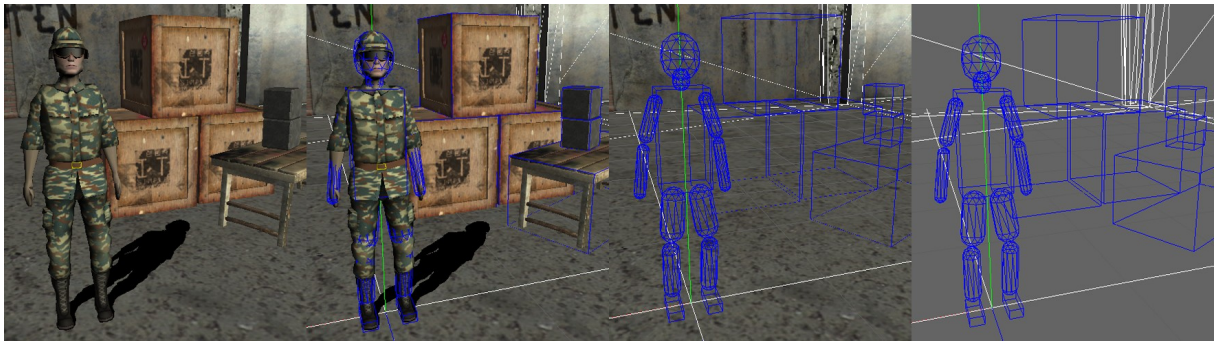


Illustration 3: Left to right: Mesh and static world, all worlds, physics and static world, physics world only

4.2.1 Mesh world

The “mesh world” contains the final appearance of all dynamic objects. Every server-side PhysicsEntity object is represented by a client-side Visualizer class, which loads the appropriate meshes and decompresses the network stream for specific objects into graphical effects (position changes, rotating elements, etc). This is also chiefly where interpolation occurs, see section 7.1.

Keeping this world in sync with the server is a great challenge and a strain on even broadband connections.

4.2.2 Static world

The static world is the visual representation of the non-changing environment in which the dynamic objects move around. The static world is usually quite large, and thus rendering it at interactive speeds poses a special challenge.

Because it is, by definition, an unchanging world, some optimizations can be used to speed up rendering, all of which are some form of pre-computation.

- Potentially visible sets: divide the world into cells, and for each cell pre-compute what other cells are visible.
- Binary Space Partition trees: can be used to do quick front-to-back ordering and are a useful partition.
- Portals: the world is divided into cells. The area where two cells are joined, for example, a door, is called a “portal”. Any rendering of a portal triggers the rendering of the cell behind that portal. This cell can then be recursively rendered, with a smaller view frustum.

Combining these techniques yields sufficiently fast world rendering for most indoor environments. Outdoor environments are more difficult There are many other techniques, using both pre-computation and run-time processing.

Because there are no moving parts in the static world, no network traffic is required, other than a few messages when a new static world should be loaded.

4.2.3 Physics world

The “physics world” shows the direct state of the physics engine using wireframe primitives. The physics world is used for physics engine- and simulation state debugging purposes. Because of this, there is no network optimization, network interpolation or rendering optimization done for this world.

The physics world is used to:

- check positions of objects present in the physics engine ('bodies') and the shape these objects take ('geoms', see next chapter),
- check interpolation network performance (see chapter 7),
- check simulation logic, such as object scripting states and triggers.

5 ODE physics

As stated before, Lumo Scenario uses the Open Dynamics Engine (ODE) for its physics. ODE has a structure that is fairly typical to all physics engines, which is outlined below. For clarity, a simplified model of the ODE code will be presented. A lot of members and classes are left out.

The main concept in a rigid body physics simulation is, of course, the rigid body. In ODE, this is the dBody class.

A body has a position, orientation, velocity and angular velocity that changes over time. Some properties of bodies are the mass and center of mass. These properties are enough to move ('step') the body over time and have forces act on it.

The dBody is tightly coupled, in a one-to-one relation, to a dGeom. The reason the dBody and dGeom are not a single class is because the physical behavior and physical shape of an object are disparate in ODE.

Forces that act on the body can be constant forces, such as gravity. They also can be forces resulting from contact with other bodies. But note that the physical *appearance* of the body isn't one of the properties of a body, so if we only had the bodies, we could not actually know if bodies are in contact. For collision detection the shape of the body is needed, 'geometry objects' or geoms for short. These objects are for example spheres, rectangles, (capped) cylinders, or meshes.

The dBody has all the relevant physical properties and the dGeom contains information about the shape of the object.

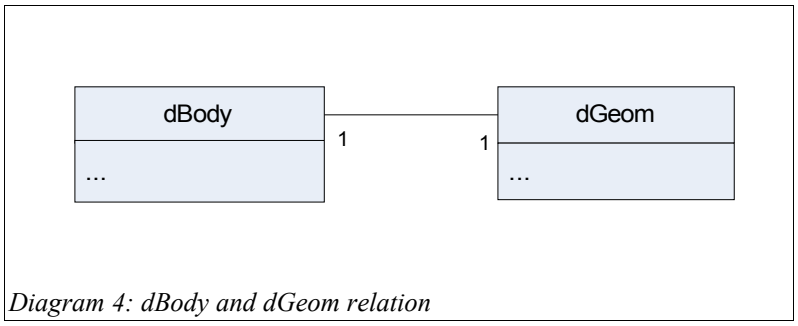


Diagram 4: dBody and dGeom relation

This is reflected in the entire structure of ODE. The integrator uses properties from the dBodies to step the world. Constraints make sure the next state of the world can only be in certain states. For example, joints are constraints.

5.1.1 Bodies

The bodies define the Newtonian physical properties of a rigid body. A body is optionally associated with a geom, which relates to collision detection and will be described in section 5.1.4.

The 'mass' property is the simulated mass of an object, represented by a dMass type (more on that later). The orientation of the body is represented by a quaternion, and a 3x3 rotation matrix. Both represent the same orientation, and are kept in sync for efficiency reasons. The current linear velocity of the body is represented by 'linearVel', the current angular velocity is represented by 'angularVel'.

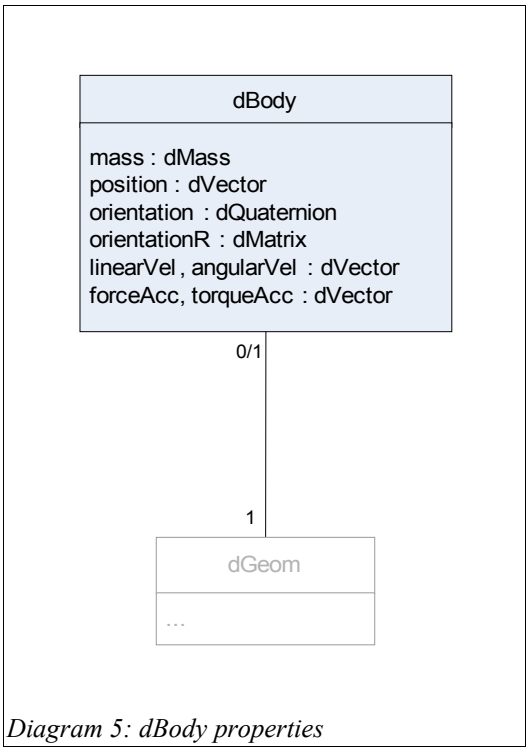


Diagram 5: dBody properties

To formalize:

Name	**Symbol**	**Properties**
Body position	$\vec{p}$	The position of the center of the body in $\mathbb{R}^3$ Cartesian space. $$\vec{p}=\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$
Body orientation as quaternion	q	The quaternion q is defined as $$q=(q_0, q_1, q_2, q_3)\in\mathbb{R}^4$$ Or, more refined $$q=(\cos(\Theta/2), \vec{u}*\sin(\Theta/2))$$ where $\vec{u}$ is a rotation axis of unit length in $\mathbb{R}^3$ Cartesian space, and Θ is the angle the object is rotated along $\vec{u}$. This means that logically $$\sqrt{q_0^2+q_1^2+q_2^2+q_3^2}=1$$ making q a unit quaternion, rotating about axis $\vec{u}$. In other words, unit quaternions live on the unit hypersphere.

Name	Symbol	Properties
Body orientation as 3x3 matrix	R	The 3x3 rotation matrix R is defined as $$R=\begin{bmatrix} lx_x & ly_x & lz_x \\ lx_y & ly_y & lz_y \\ lx_z & ly_z & lz_z \end{bmatrix}$$ The vectors *lx, ly* and *lz* are all of length 1, and represent the body-local x, y, and z axes of the object in global space. Note that you can rotate a vector $\vec{l} \in \mathbb{R}^3$ from local space by global space by multiplying it with R: $$\vec{l}\,' = R\vec{l}$$ where l' is the global vector. This means that $$\vec{k}\,' = R\vec{k} + \vec{p}$$ yields the global position $\vec{k}\,' \in \mathbb{R}^3$ of a point $\vec{k} \in \mathbb{R}^3$.
Body velocity	$\vec{v}$	The current velocity (speed) of the center of the body in $\mathbb{R}^3$ Cartesian space. $$\vec{v}=\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$
Body angular velocity	$\vec{\omega}$	The angular velocity $$\vec{\omega}=\begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$ specifies the rate of rotation of the body. You can look at the $\vec{\omega}$ as a vector from the origin of the body. The body rotates about this vector. The length of the vector specifies how fast the body rotates. The $\vec{\omega}$ is defined in the global space. More specific, if $\vec{l}$ is a vector in $\mathbb{R}^3$, in the global space, indicating the position of a point (any point) relative to the center of the body ($\vec{p}$), the speed (time-derivative) of $\vec{l}$ is $$\dot{\vec{l}} = \omega \times \vec{l}$$

Name	Symbol	Properties
Body force accumulator		The body force accumulator is a global space vector that keeps track of all forces on an object. The force accumulators are cleared every physics step. Gravity, user forces and LCP forces (see section 5.1.6) are all added to the force accumulator. The accumulator is then used in the step function itself.
Body torque accumulator		The body torque accumulator does the same thing as the force accumulator, only for rotations.

For more information about these properties, see [ode02].

These properties are sufficient to integrate object positions over time: the user adds forces to the force accumulators and the bodies will fly around correctly. However, they will fly through each other and it is not possible to attach two bodies together in a meaningful way. So to complete the definition of the physics world, we need joints.

5.1.2 Joints

Joints make sure two bodies can only move in some regard relative to each other; in other words, they remove one or more degrees of freedom from the simulation.

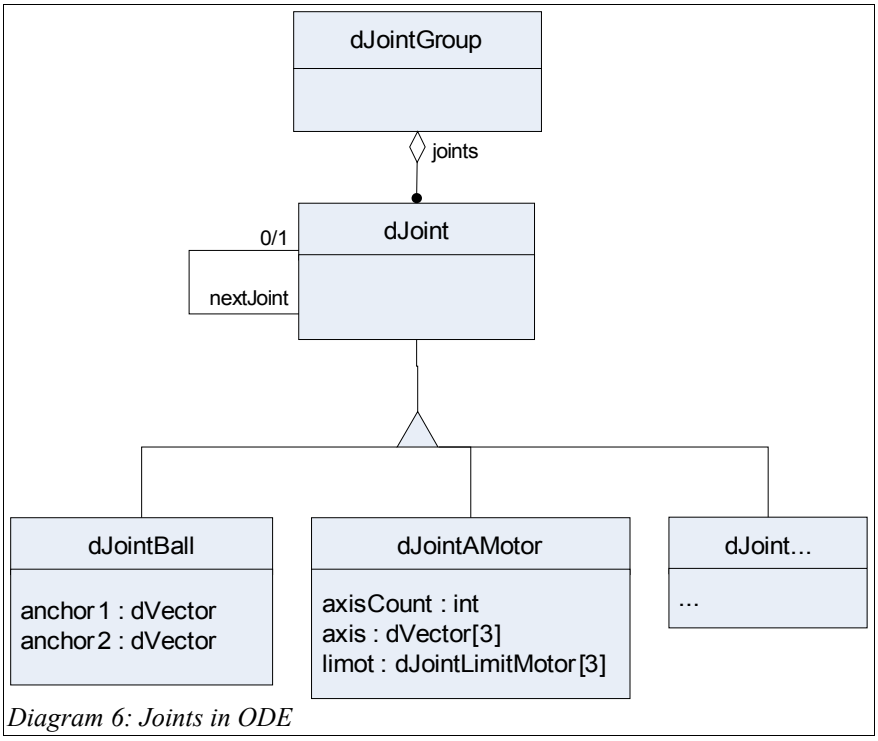


Diagram 6: Joints in ODE

Here is a simplified UML diagram of the joint implementation in ODE.

Some joints are in a dJointGroup. This allows efficient addition and removal of many joints at a time, which is convenient for reasons that will later become apparent (contact joints).

The dJointBall (ball joint) and dJointAMotor (angular motor joint) are two examples of joints. There are many more joint types, such as hinge joints, universal joints, slider joints, and, important for collision

detection, contact joints.

The ball and angular motor joints are mentioned here because they play in important role in rag doll physics. The ball joint, obviously, keeps tho bodies pivoting around a shared point. However, it does not constrain the movement in any other way. This means two bodies can rotate freely about, or even into themselves.

Compare this to a hinge joint, which restricts relative body motion to a single rotational axis and has stops on this axis (called low and high stops) that restrict the range of motion along the axis,

5.1.3 Worlds, bodies and joints

The world keeps track of all the joints and bodies. We will now combine the above two UML diagrams into one and add the world.

Each joint has zero, one or two bodies associated with it. These are the bodies it is constraining.

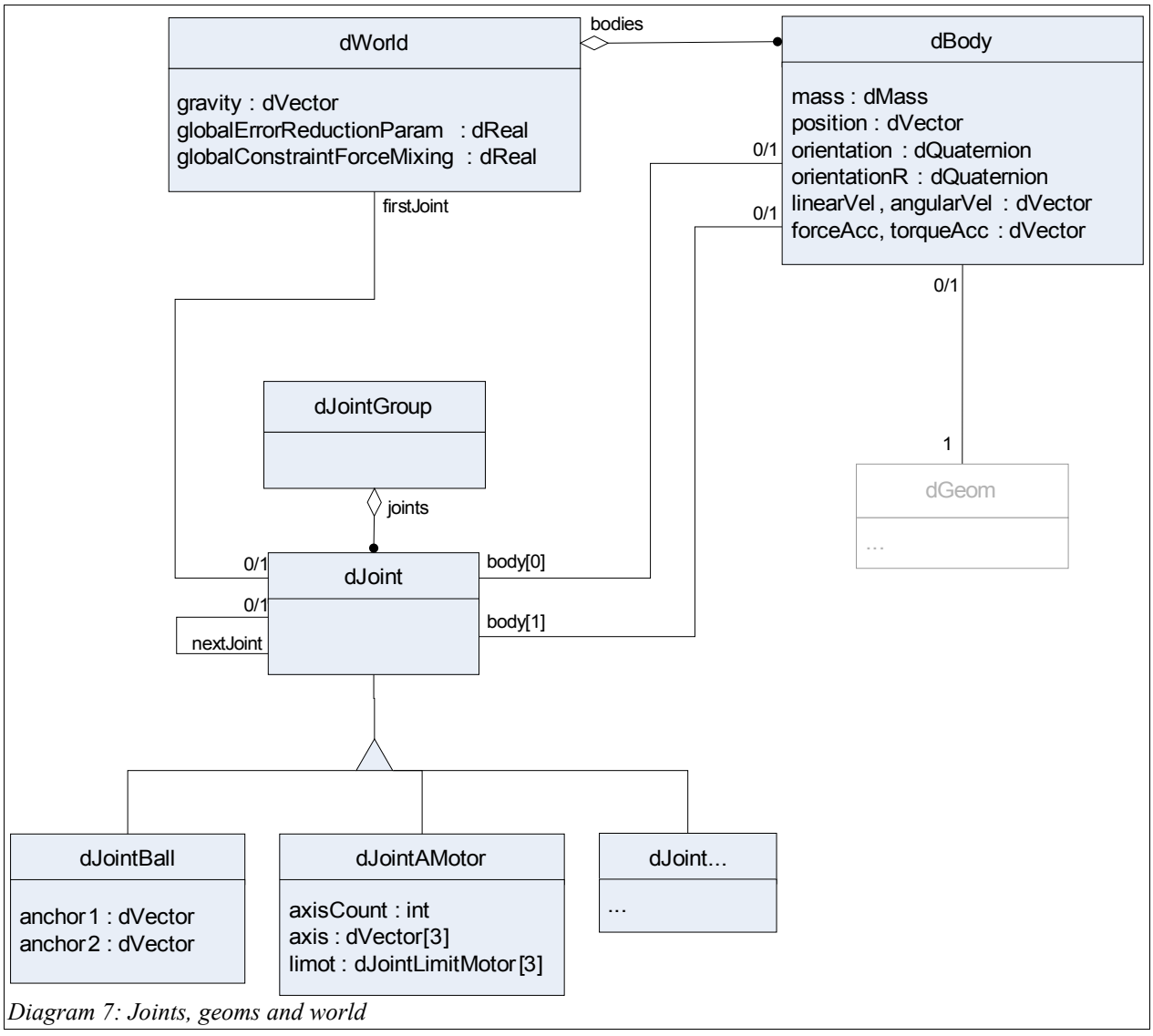


Diagram 7: Joints, geoms and world

5.1.4 Geoms

The geoms determine the physical 'appearance' of bodies. Because ODE is a physics engine (and not a graphics engine) a mathematical description of the appearance of bodies will often suffice. Collision detection generates contact joints when bodies intersect. So the entire goal of the complete dGeom structure is generating all contact joints fast enough for real-time calculations.

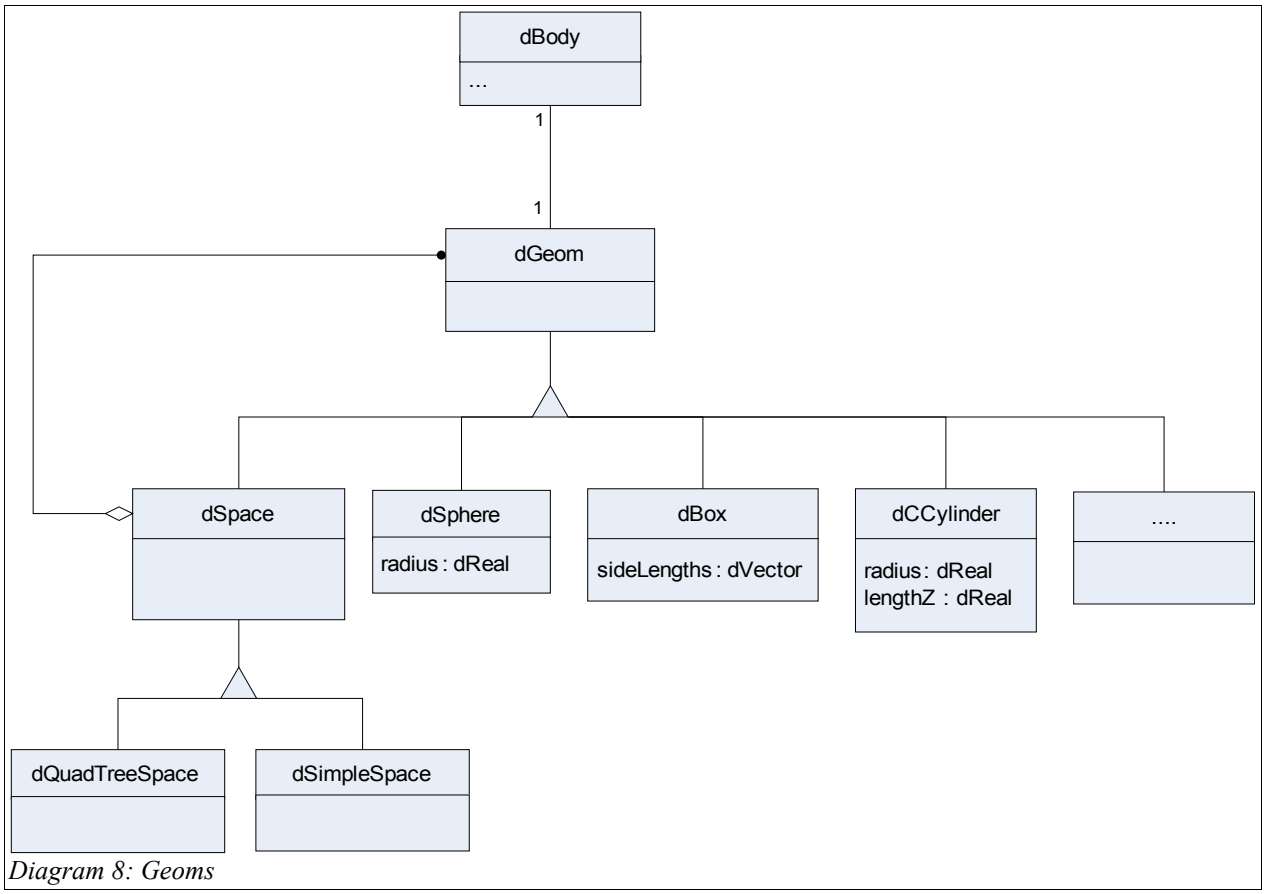


Diagram 8: Geoms

You can recognize the Composite pattern ([gamma95], page 163) here. Lumo Scenario uses one main collision space, an instance of dSimpleSpace. All other spaces and geoms are put into this space. The position and orientation of a geom is linked to the position and orientation of the body.

Static Geoms

If a geom has no body, it is considered static. In this case, the geom has its own position and orientation (contrary to the diagram above). Without a body, it cannot move in response to impulses. This is why its called static. Usually these kinds of objects are used for the world the dynamic objects are in (if they respond to collisions by generating contact joints), or sensors (if they respond to collisions by triggering some application-specific sensor event).

5.1.5 Collisions and contact points

The output of collision detection is a list of points, indicating the intersections between all intersecting

points. These are called 'contacts'. This list is regenerated each frame. ODE does not simulate contact surfaces, as this is generally considered too computationally intensive for interactive simulations. These contact points are converted to contact joints and added into the joint list. This is the bridge between collision detection and the integrator.

5.1.6 Linear Complementary Problem

So, now we have a list of bodies, with all the properties mentioned above, and a list of constraints of where those objects can and cannot go.

Each violated constraint generates a correction force that would resolve any constraint violations. If these correction forces were all applied in order without regard to each other, there would only be one-to-one interactions between objects.

Instead, we need to find forces on all the objects that satisfy the constraints. This means solving the constraints as a linear system. Solving the system using standard linear algebra techniques does not work. This has two major reasons. The first is because this approach will sometimes yield negative forces, which are physically meaningless. The second is that the system is unsolvable if constraints do not provide enough information to arrive at *one* correct state.

If we require that all forces be positive, we have a linear system with some extra constraints, also known as a linear complementary problem. This is an iterative method of solving that can provide a solution where all forces are positive (or zero). The most basic example of an LCP solver the the simplex method. More information on the basics of LCP can be found in [winston93]. ODE uses the Dantzig LCP solver described by Baraff in [baraff97].

5.1.7 Time stepping

Physics engines work best when each integrator step moves the simulation forward by the same amount of time. To illustrate this, imagine what would happen if we couple the rendering loop with the physics loop and step the simulation with the last frame time. This would mean that with a fast frame rate the physics are very smooth and will generally behave correctly. However, if for any reason the framerate is lower than expected, the stability will decrease to a point where the simulation might explode. It might even explode only on a particular sequence of frame times, even if those frame times are very small (=high frame rates). The upshot of all this is that you cannot be sure that the simulation is stable when you have an varying frame time.

To remedy this problem, the physics loop is decoupled from the rendering loop. It runs in its own thread, and, this in case, even in its own process: the server executable. The server does its physics and other processing, then measures the time left in the frame, and yields the thread to other processes and threads (such as rendering) until it's time to start the next physics frame.

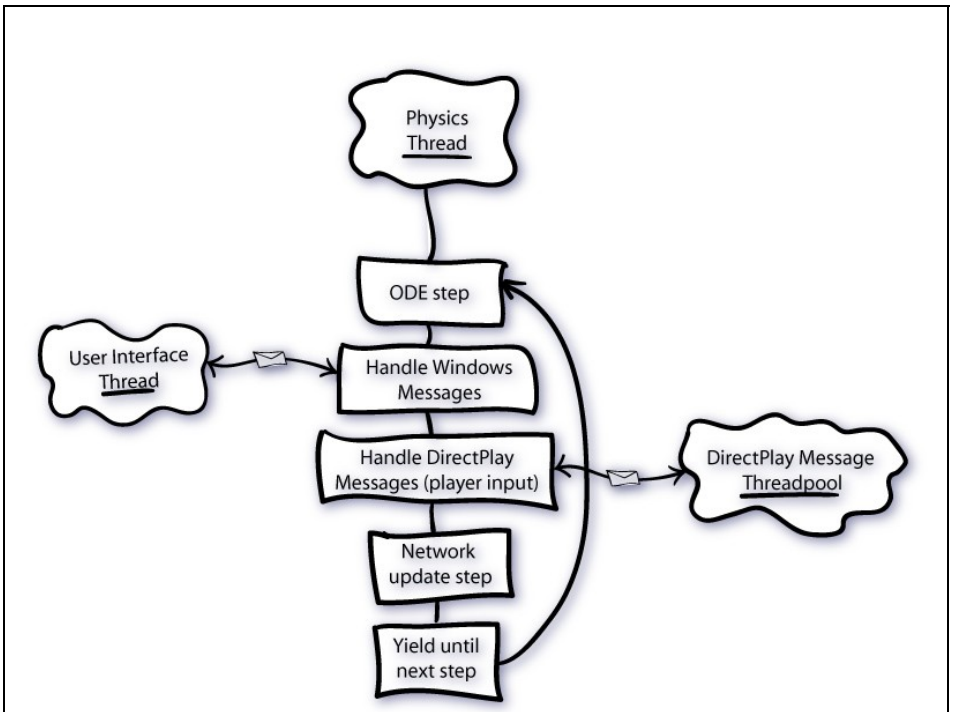


Diagram 9: Tasks of the physics thread on the server

This makes sure the physics run at a steady frame rate, but also puts a hard limit on how much physics processing you can do in a single physics frame. If the physics calculations (and scripting and other tasks described in the illustration above) take too long, the physics thread will not sleep at all. However, the physics time steps will remain the same, so the simulation will run slower than real-time (meaning, the time in the physics world will run slower than actual time).

Another big advantage of using a stable physics frame time is that the simulation becomes deterministic. If you set the simulation of a particular state and step a couple of times, you will end up with the same state over and over, each time you try this. With a variable frame time there is no telling where your simulation might end up. There are some caveats when relying on ODE being deterministic:

- random generators are used to randomize constraint orders in the LCP solving iteration (see section 5.1.6). This can be solved by resetting the random generator's seed before starting a simulation.
- Different CPU architectures and even CPU brands vary slightly in floating point precision. There are no viable solutions to this, other than using the same CPU when replaying a simulation.

6 Sampling and displaying rag dolls

This chapter describes how the motion capture data is sampled at a client, sent to the server for processing, then sent back to all clients for displaying. The illustration below shows the path motion captured data takes in Lumo Scenario. Because of this long path, the delays on the client between sampling and displaying movement can be significant. A local feedback mode was used to alleviate this problem, see section 7.5.

A diagram of the flow of rag doll orientation data, from sampling to displaying, is shown below.

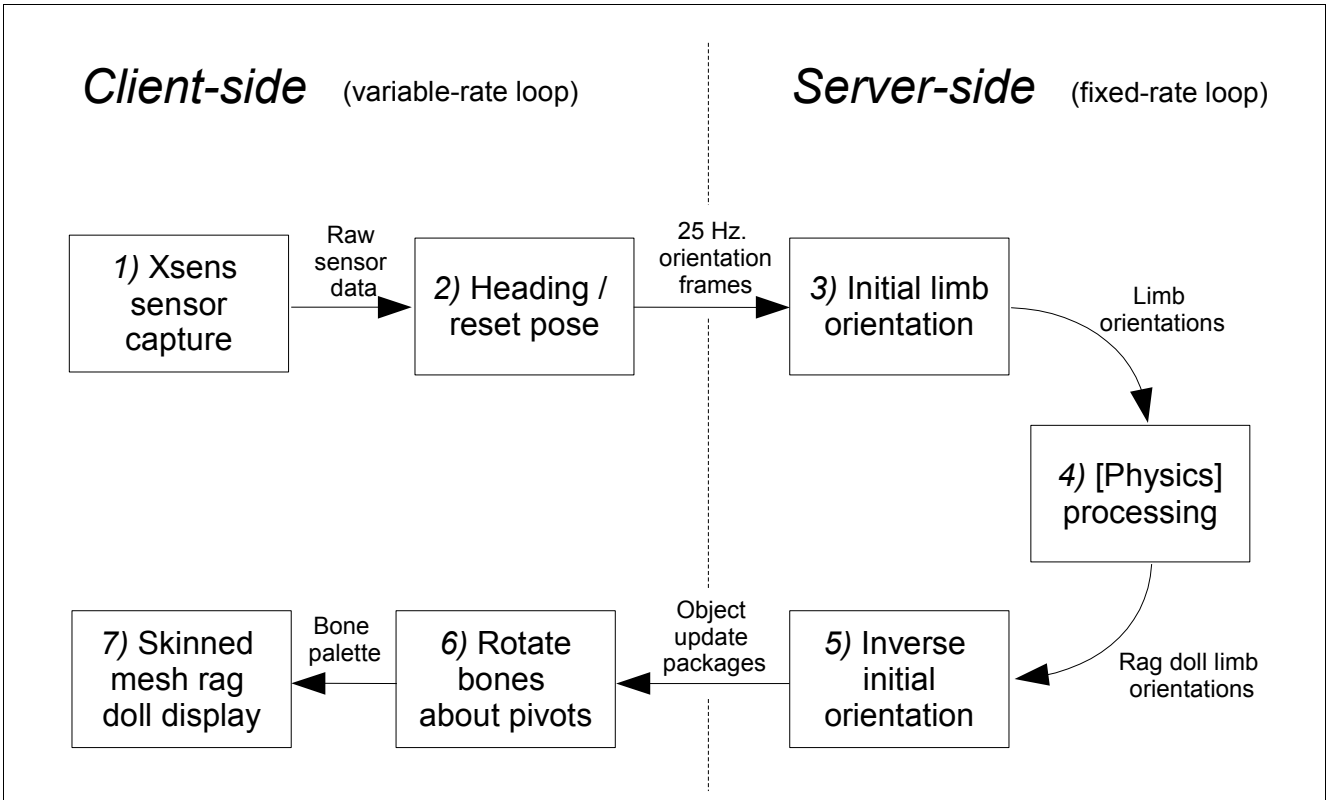


Diagram 10: The path a motion capture sample travels, from client to server to client

6.1 Client-side sampling

It all starts out with sampling each inertial sensor at the client, box *1)* in diagram 10 above. The Xsens quaternion data is in a right-handed coordinate system, as shown in the illustration to the right.

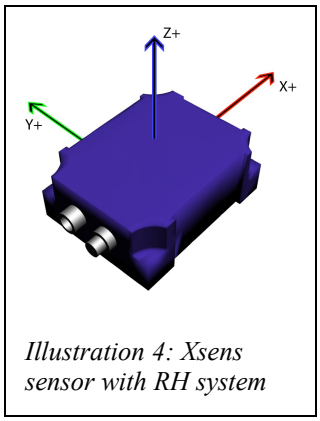


Illustration 4: Xsens sensor with RH system

The identity quaternion represents a sensor with its Z+ axis pointing up, and its X+ axis facing the magnetic North. This means each sensor reports its orientation relative to the North.

All orientations should be relative to the 'reset pose' of the model. This process is called resetting and is done as follows. The person in the motion capture suit mimics the reset pose as much as possible. These orientations are inverted and stored:

$$q_i' = q_i^{-1}$$

Where q_i is the sampled orientation of limb *i* and q_i' the calibration quaternion.

In client-side Lumo Scenario code:

```
local calibration


-- Reset the Xsens motion capture pose. Called from the server
-- when the 'Reste Pose' button is clicked.
-- sensorList: list of all Xsens sensors' quaternions
function resetXsensPose(sensorList)
  calibration = {}
  for sensorIdx,q in ipairs(sensorList) do
     calibration[sensorIdx] = q:inversed()
  end
end
```

Then, as the sensors are sampled, the samples are calibrated:

$$s_i' = q_i' s_i$$

Where s_i is the sampled orientation of limb *i* and s_i' the calibrated quaternion.

```
-- Called after sampling the Xsens sensors. Returns
-- calibrated orientations.
function calibrateXsens(sensorList)
  local orientations = {}
  for sensorIdx,q in ipairs(sensorList) do
     orientations[sensorIdx] = q * calibration[sensorIdx]
  end
end
```

This data then is sent off to the server at a fixed rate. This way, each client connected to the server is free to have one (or more) full-body motion captures. This means it is possible for each client to control a rag doll in the same virtual space.

6.2 Server-side transformations

So, the server receives a set of quaternions, $r_i = (q_{0,} q_{1,} q_{2,} q_3) \in \mathbb{R}^4, i = 1..n$, with n = number of limbs, indicating the limb movement relative to the reset pose. We still cannot use these to set the limb orientations; all limbs would be in their unrotated positions. We want them to be in the reset pose.

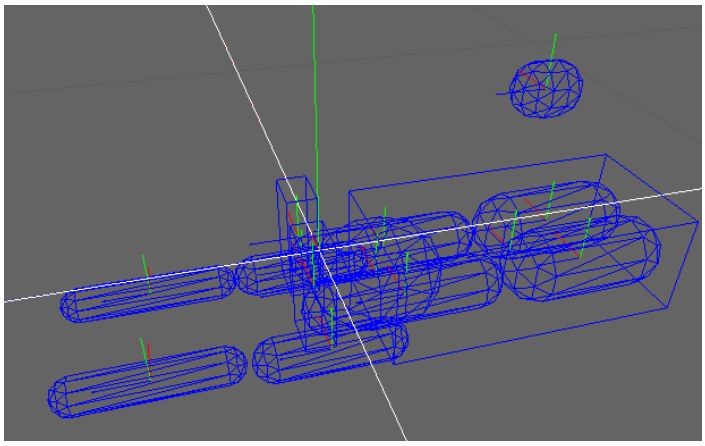


Diagram 11: Reset pose directly set in limbs: all limbs unrotated

Multiplying the quaternions with the quaternions representing orientations of the limbs corresponding to the reset pose solves this. That is why every rag doll has an initial orientation list $i_i'=(q_0, q_1, q_2, q_3)\in\mathbb{R}^4, i=1..n$. These orientations and other information are in a Lua data file:

```
geoms =
{
   -- Static Geom 'CM_Head'
   [1] = { type = "pill", x = 0.002308, y = 1.629900, z = 0.023619, rx = 0.000000, ry = 0.660030,
rz = -0.751239, rw = 0.000000, radius = 0.111881, length = 0.051264 },
   -- Static Geom 'CM_Spine'
   [2] = { type = "box", x = 0.006147, y = 1.140748, z = 0.011621, rx = -0.000000, ry = 0.707107,
rz = 0.707107, rw = 0.000000, sizex = 0.306296, sizey = 0.216449, sizez = 0.621202 },
   -- Static Geom 'CM_LeftUpperArm'
   [3] = { type = "pill", x = -0.176760, y = 1.269023, z = -0.024673, rx = 0.093603, ry =
0.694184, rz = -0.707756, rw = -0.091808, radius = 0.042317, length = 0.216325 },

... (and so on for 13 items) ...

   -- Static Geom 'CM_RightFoot'
   [13] = { type = "box", x = 0.119015, y = 0.030124, z = 0.055110, rx = -0.000000, ry = 0.707107,
rz = 0.707107, rw = 0.000000, sizex = 0.093834, sizey = 0.222733, sizez = 0.060007 },
},
```

Now,

$$q_i=r_i i_i$$

makes q_i the orientation to set the limb to. We are now ready to set the orientation directly or do other processing. The orientations in the above Lua file shown in the rag doll:

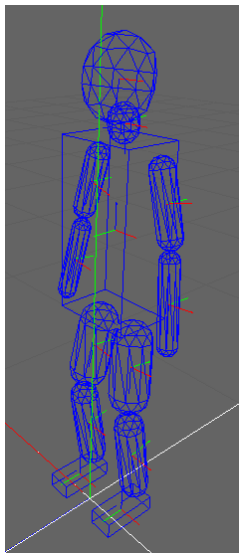


Diagram 12: Rag doll initial limb positions

After physics processing, the resulting orientations and positions of the limbs are sent to each client that potentially has the rag doll in view. Algorithms to determine what is potentially in view for each client can include frustum-culling and/or potentially visible sets (PVS), see 4.2.2.

6.3 Client-side transformations

When all matrices in the bone palette are the identity matrix *Id*, the skinned mesh takes its modeled position (the position in the `position : float3` entries in the vertices). In this case this is the position shown below in illustration 5.

The origin of the skinned mesh rag doll is, by re-lion convention, at the base the of model, near the feet. The diagram below shows this:

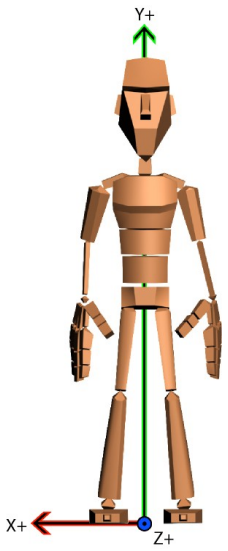


Diagram 13: Schematic rag doll, all bones = identity matrix

Also by re-lion convention, the rag doll looks along the Z axis. Because the rag doll is shown from the front, the X axis points the other way it normally does. And because we use a left-handed coordinate system, the Z axis points out of your paper/screen.

As shown later on, the physics engine calculates the positions and orientations of the limbs. Because a physics engine outputs these positions and orientations exclusively in global space (world space), the data for bone position and orientation is sent to the client using global space position/quaternion pairs. This means we do not need the rag doll to be hierarchical.

Now, when we for example rotate the bone associated with the head, using the Lumo graphics engine, it will rotate around the origin of the rag doll like this:

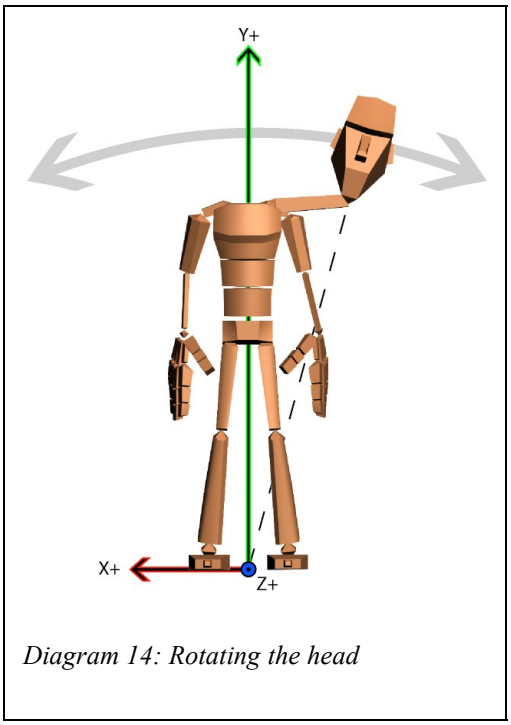


Diagram 14: Rotating the head

The physics engine (ODE) always uses the center of a limb as center of mass and reference point. Meaning, the position vector $\vec{p}$ of a body indicates the center of a limb. This means we will have to rotate the head around this pivot. Note that $\vec{p}$ changes position as the head rotates.

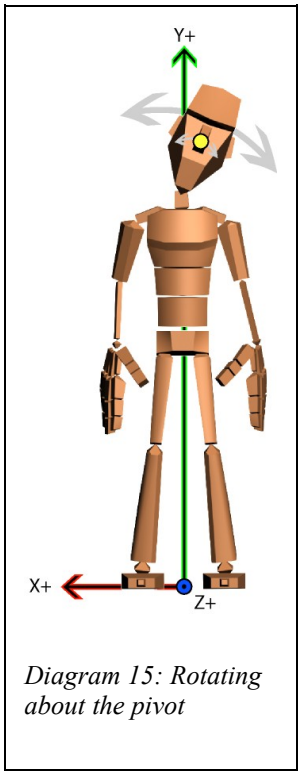


Diagram 15: Rotating about the pivot

The pivot is the yellow dot. The 'mustache' is the movement of the pivot. This is the limb position sent by the physics engine. One might expect the head pivot to be at the joint location, that is, the neck. This is not the case because ODE geoms and bodies are represented by their center positions (see 5.1.1).

So, we need the bones to rotate about the pivot instead of the model origin.

First, let's define $M(x)$ as the 4x4 geometrical transform matrix of x, where x can be a vector or quaternion.

> As an aside:
>
> Converting a vector $\vec{t} \in \mathbb{R}^3$ to a matrix:
>
> $$M_t(\vec{t}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \vec{t}_x & \vec{t}_y & \vec{t}_z & 1 \end{bmatrix}$$
>
> Converting a quaternion $\vec{q} = (q_0, q_1, q_2, q_3) \in \mathbb{R}^4$ to a 3x3 rotation matrix is a bit more involved, and is described in [kuipers98], section 5.14.

With this we can create a formula to rotate a bone about a pivot and translate it:

$$M' = M(\vec{p}) M(\vec{t}) M(q) M(-\vec{t}) \qquad \text{Eq. 1}$$

With

- $\vec{p} \in \mathbb{R}^3$ is the position of the limb pivot in global space, as produced by the physics engine;
- $\vec{t} \in \mathbb{R}^3$ is the position of the pivot in the untransformed rag doll local space (object space);
- $q \in \mathbb{R}^4$ is the orientation of the limb in global space, as produced by the physics engine;
- M' is the bone matrix associated with the limb.

There are a lot of 4x4 matrix multiplications and conversions involved. A faster way might be to directly set the quaternion and position into the bone matrix. The orientation then matches instantly (observe that eq. 1 only has $M(q)$ actually rotating the object); we only need to figure out how to move the object around $\vec{t}$ instead of the origin $\vec{o} = (0,0,0)$, without resorting to the matrix math described above.

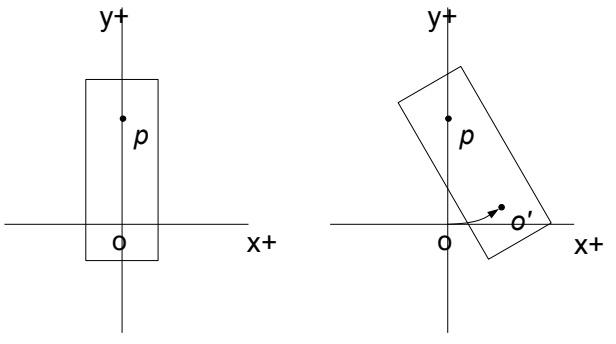


Diagram 16: Rotating an object about point p

Rotating *o* about $\vec{p}$ yields the correct offset:

$$\vec{o}' = (0 - \vec{t})\, q + \vec{t} \qquad \text{Eq. 2}$$

> As an aside:
>
> Here we are multiplying *-p* by quaternion *q*. This commonly means transforming (rotating) the vector by the quaternion. To compensate for the rotation, we need to translate (move) the object with *o'*.
>
> Rotating a vector directly by a quaternion is done elegantly like this:
>
> $$\vec{v}' = q\, \vec{v}\, q^* \qquad \text{Eq. 3}$$
>
> Where $q \in \mathbb{R}^4$ is a unit quaternion, and $\vec{v} \in \mathbb{R}^3$ the vector to be rotated. $\vec{v}' \in \mathbb{R}^3$ is the rotated vector. Note that multiplying a vector by a quaternion is done by treating it as a quaternion $q \in \mathbb{R}^4$ with $q_w = 0$, meaning the real part of the quaternion is zero. This is called a pure quaternion. For more information see chapter 5.8 of [kuipers98].
>
> Another, slower, way of rotating a vector by a quaternion is converting the quaternion to a matrix and transforming the vector by this matrix.

Combining eq. 2 and 3 yields:

$$\vec{o}' = q(-\vec{t})q^* + \vec{t}$$

Adding the physics engine position of the pivot in global space:

$$\vec{o}' = q(-\vec{t})q^* + \vec{t} + \vec{p}$$

Thus, setting the limb position to the $\vec{o}' \in \mathbb{R}^3$ and the limb orientation directly to $q \in \mathbb{R}^4$ yields correct results.

6.4 The skinned mesh

On the client, a skinned mesh is used to represent the rag doll. This skinned mesh uses a set of 3x3 rotation matrices, representing the position of each limb. This list of matrices is commonly called the *bone palette*. Each vertex of the mesh has a small list (two to four items) of *bone indices* and *weights*. The indices refer to the bone palette, the weights to how much the indexed bone influences the vertex. The weights should add up to 1.

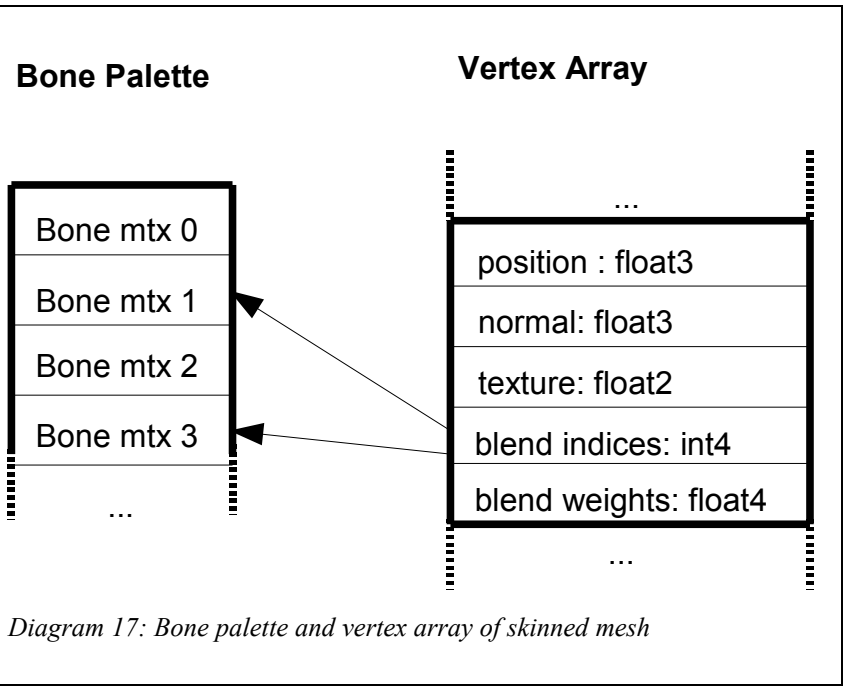


Diagram 17: Bone palette and vertex array of skinned mesh

Assuming these are vertices and the input parameters of a vertex shader:

```
struct VS_INPUT
{
   float3 Pos;
   float3 Normal;
   float2 Tex;
   float4 BlendWeights;
   float4 BlendIndices;
};

VS_INPUT SkinnedMeshVertex;
float4x4 WorldViewProjection;
float4x3 BonePaletteArray[26];
```

The positions of the vertices can then calculated in the vertex shader:

```
// The number of bones per vertex in the mesh
static int BoneCount = 4;

// Iterate over all bones except the final bone,
// add them to ObjectSpacePos
float LastWeight = 1;
float3 ObjectSpacePos = 0;
for(int b = 0; b < BoneCount-1; b++)
{
   LastWeight = LastWeight - SkinnedMeshVertex.BlendWeights[b];
   ObjectSpacePos += mul(float4(SkinnedMeshVertex.Pos, 1),
     BonePaletteArray[SkinnedMeshVertex.BlendIndices[b]]) *
     SkinnedMeshVertex.BlendWeights[b];
}

// Add in final bone (this makes sure the weights always add up to 1)
ObjectSpacePos += mul(float4(SkinnedMeshVertex.Pos, 1),
```

```
  BonePaletteArray[IndexArray[BoneCount-1]]) * LastWeight;

// Transform to view space and we're done
Viewspace_Position = mul(float4(ObjectSpacePos, 1), WorldViewProjection);
```

Combined with the talents of the re-lion graphics department, this results in a skinned mesh looking like this:

Illustration 5: Skinned mesh soldier

Now we have come full circle, from the client sampling the inertial sensors, to the server doing the processing, and back again to the client displaying the rag doll.

7 The network

This section describes some issues encountered while constructing a network protocol that could handle the one-way synchronization of the VR world in the physics engine to each connected client.

Interpolation and extrapolation are described because they are fundamental to creating fluid movement on clients. Some bandwidth-reducing techniques are discussed, and the problem of network delays is addressed.

7.1 Interpolation and extrapolation

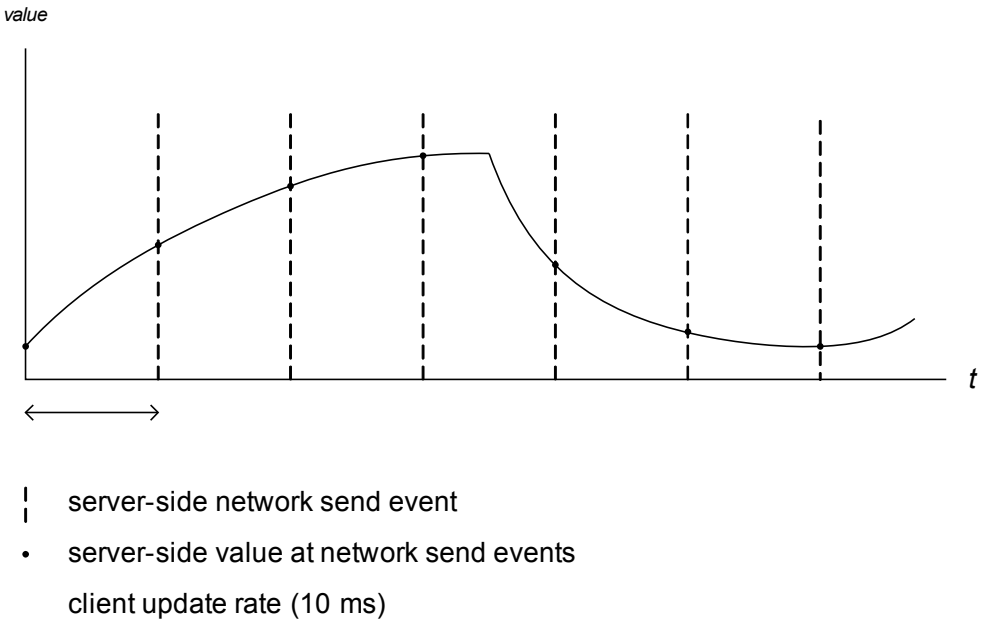


Diagram 18: Example X,Y,Z or rotational value component of a server-side object

Let's make up a server-side value that we want all clients to follow as closely as possible. For the purposes of this discussion, we will treat this as a generic value, from the state of a physics body. This means the value is a component of $\vec{p}$ (position) or a complete q (orientation). See 5.1.1 for more information about physics bodies.

The value makes a sharp nudge in the middle of the graphic, caused, for example, by a bounce or other sudden impulse to the body.

The vertical stippled lines represent the moments the server would like to send all moving objects to the client (the 'mesh world', see 4.2.1). The dots represent the value at these moments.

The server does three physics steps before sending the state data to the clients, as shown below in diagram 19. This diagram shows the highest definition the value actually has.

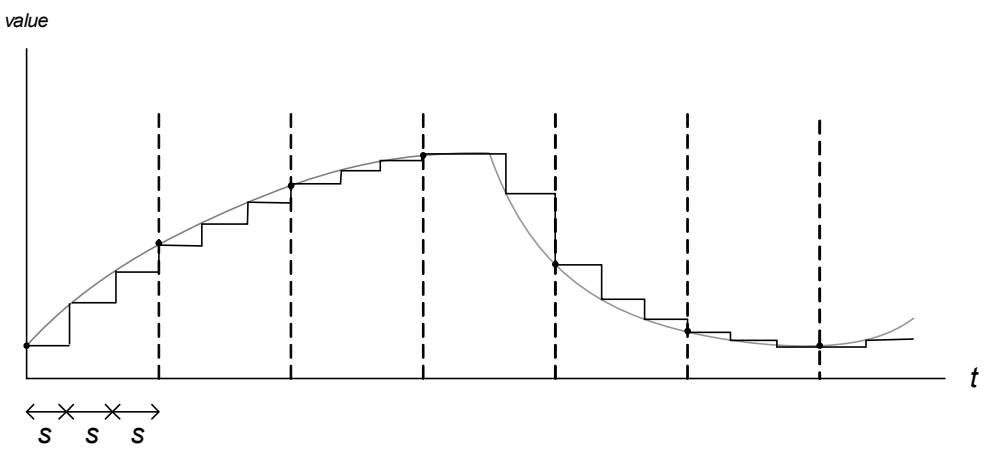


Diagram 19: Actual value through server ticks

A first implementation might set the value on the server as it is received, as shown below in diagram 20.

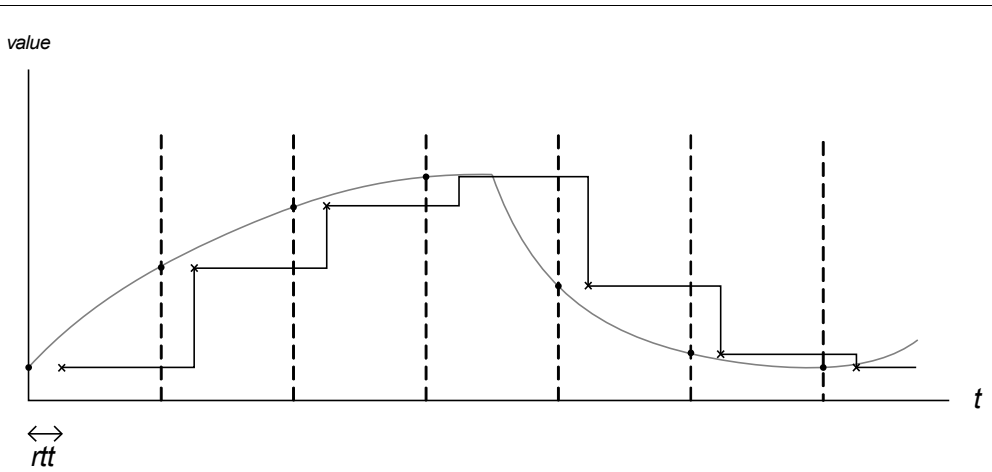


Diagram 20: Naive approach: set value as it is received

Of course there is a delay while the updates are being sent from the server to the client, called the half round-trip time. In this illustrative case, it is about 25 milliseconds. Usual round trip times, sometimes inaccurately called 'ping' times, are about 2 to 20 milliseconds on a local area network, making the half round-trip time about 1 to 10 milliseconds. The little x's mark the time when a sent value (a dot) is actually received by the client. It should be noted that round-trip times can vary during a session, which is not shown here.

Diagram 20 shows that objects will jerk around a lot. A straightforward solution to this is decreasing the

number of physics steps between sending data. This increases the definition of animation, but generates a proportionally higher network bandwidth. Detrimental effects of this are:

- more CPU time (on both server and clients) devoted to handling this bandwidth
- send and receive buffers (on both server and clients) get clogged with data, increasing network delay
- less clients can be serviced

During testing at an update rate of one-to-one to the server rate, which is 30 Hz, sometimes server-side send buffers got so clogged that the system froze up completely. This happens when more memory allocation requests (for increasing the send buffer size) are issued than the Windows kernel can handle.

The next step is to interpolate (at the client) the values that are received. For interpolation to work, we need something to interpolate to. That is why we need to wait one frame before interpolation can take place, see diagram 21 below.

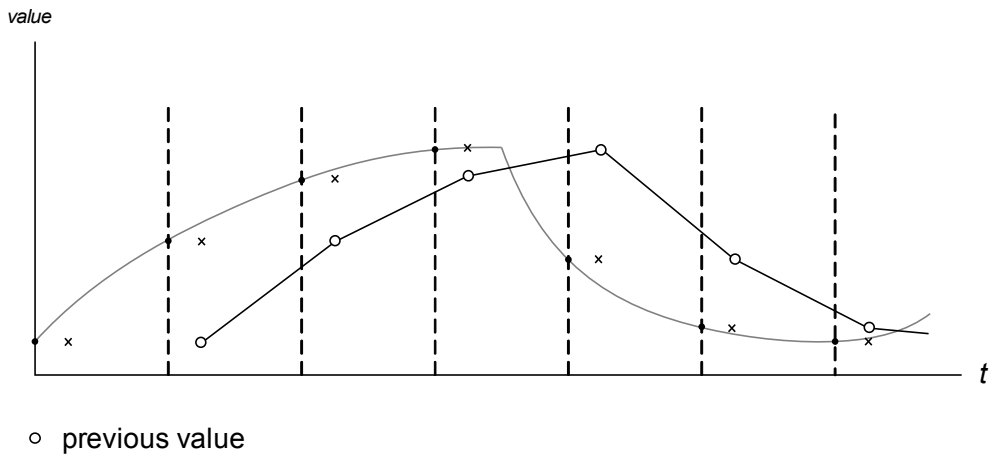


Diagram 21: Interpolation

While producing smooth movement, a disadvantage of client-side interpolation is that object updates are consistently lagging behind for the amount of an update frame, plus the half round-trip time. In this case, this amounts to 125 milliseconds.

To counter this, the current derivative (speed) of the value is considered. This data is part of the state of a physics body. This way, it's possible to extrapolate where, approximately, the value will be when the client receives the *next* update.

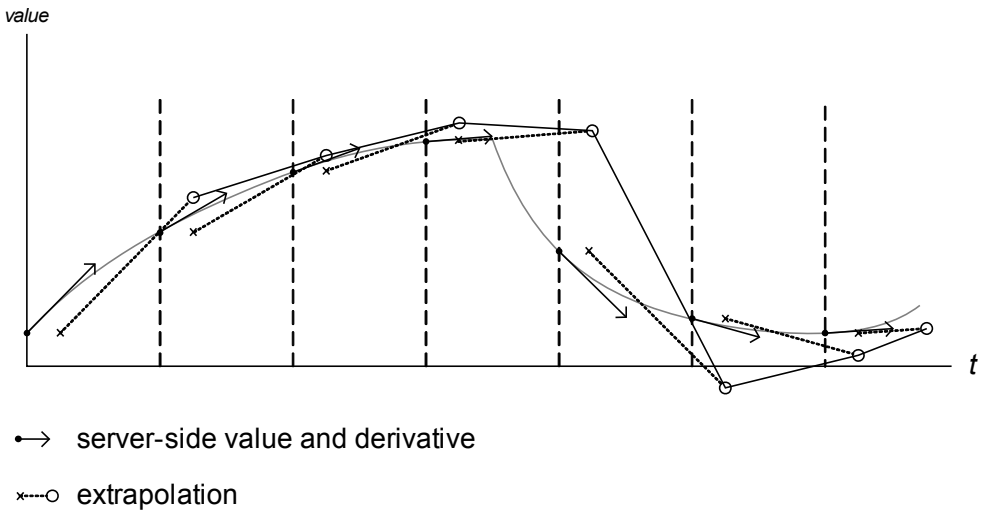


Diagram 22: Extrapolation

The arrows indicate the derivative. The dotted lines represent the extrapolation. Each time the client receives an update, the current client position is used to interpolate the extrapolated value sent.A cleaned-up version is presented below.

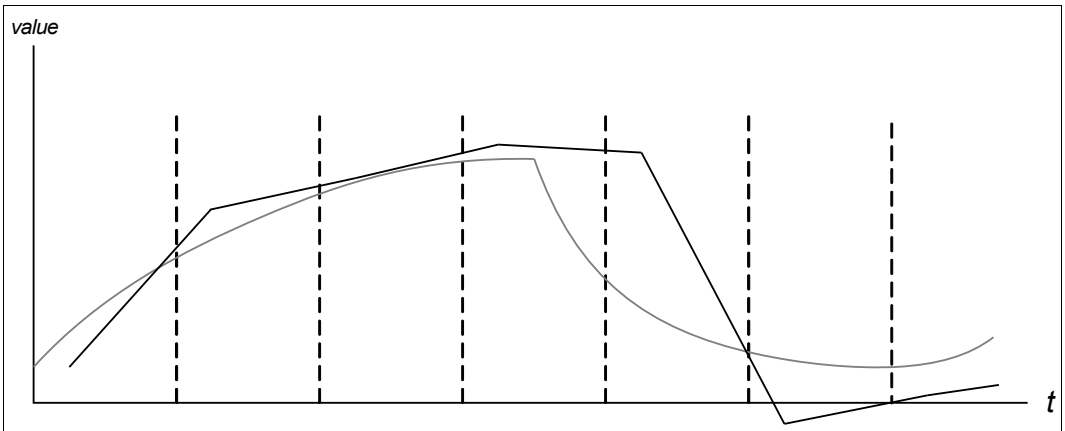


Diagram 23: Cleaned up extrapolation

This technique results in more accurate movement along smooth movement, but, like normal interpolation, fails at sudden changes of direction. This is the current technique implemented in Lumo Scenario.

Possible improvements include:

- also including second derivative (acceleration)
- while stepping the physics engine, it is possible to detect sharp jolts. This can be done by checking the position/speed vector, or by checking if the total force (multiplied by the mass) on an object. If this value is larger than some threshold, send an 'intervention' packet immediately, telling the client the new value a bit earlier.
- one could also run a secondary physics simulation on the client. The client is only corrected by the server is the simulation went wrong. This is called client-side prediction. This has always been a very popular technique in first person shooter games. However, because the arrival of

physics engines make this client-side prediction exceedingly difficult, and because the introduction of broadband networks for the masses reduced the average latency, client-side prediction is not often used anymore in modern simulations and games.

7.2 Data rates and threading

As shown above, using less bandwidth is a good thing. There are many techniques to further reduce network bandwidth:

- only sending updates when an object actually moves or rotates. This couples nicely with the 'auto disable' feature of the physics engine. This feature disables the physics processing part of a body when body movement and rotation are below an adjustable threshold for an adjustable amount of time. Note that only the body is disabled, but not the geometry part used for collision detection, see sections 5.1.1 and 5.1.4. When a body is hit by another body, it is re-enabled again. It is easy to check for disabled bodies when sending updates.
- using knowledge about what kind of data is sent. For example, positions can be reduced in dimensionality or precision (number of bits used) in some specific cases. Also, unit quaternions can be compressed in several ways (see 7.3 below).
- letting the client extrapolate further by not sending updates, if the velocity of the value is constant. This is called dead reckoning, see [aronson97].
- using server-side potentially visible sets (see section 4.2.2) or other occlusion detection techniques to determine line-of-sight for each client vs. each object. Only send the updates for the potentially visible objects.
- using dictionary-based data compression techniques, such as zip or bz2 compression. It is a good idea to use a single dictionary over all network messages.

The last two techniques are not researched or implemented in Lumo Scenario at the moment, the others are.

7.3 Quaternion compression

The implementation of quaternion compression is described in here. This is because in the dismounted trainer, most of the rag doll data consist of a set of 15 to 20 quaternions. The number depends on the number of limbs in the rag doll and the number of sensors in the motion capture suit. See sections 6.1 through 6.4 above.

A rotation quaternion is defined as (see 5.1.1):

$$q = (\cos(\Theta/2), \vec{u} * \sin(\Theta/2))$$

where $\vec{u}$ is a rotation axis of unit length, and Θ is the angle the object is rotated along $\vec{u}$.

Define q as

$$q = (q_0, q_1, q_2, q_3) \in \mathbb{R}^4$$

a property of q (as long as it is a unit quaternion) is then:

$$\sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = 1 \qquad \text{Eq. 1}$$

making $\vec{q}$ a unit quaternion, rotating about axis $\vec{u}$.

So, when using single-precision (32-bit) IEEE Standard 754 floating point [wiki2006] numbers, sending one quaternion over uses 16 bytes in the data stream.

Although quaternions have much less redundant data than rotation matrices, you can still create non-unit

quaternions by scaling a quaternion linearly. You can remove this redundant information by removing a component. E.g. when we solve Eq. 1 for q_0 :

$$q_0 = \pm\sqrt{1 - q_1^2 - q_2^2 - q_3^2}$$

it is clear that we can reconstruct one component from the three others, except the sign of q_0 . But we can solve this when we realize that component-wise negated quaternion $(-q_0, -q_1, -q_2, -q_3)$ actually describes the same rotation. This makes sense: when the axis an object rotates about is inverted together with the rotation direction, you end up with the same rotation.

So, if q_0 turns out to be negative, the whole quaternion is negated first. So, when decompressing, q_0 must always be positive:

$$q_0 = \sqrt{1 - q_1^2 - q_2^2 - q_3^2} \qquad \text{Eq. 2}$$

During implementation, we noticed that sometimes objects would disappear randomly. This turned out to be due to precision errors in the subtraction under the square-root sign of Eq. 2, leading to a negative square root, leading to an indefinite value for q_0 . These precision errors are amplified by the fact that sometimes numbers very close to each other are subtracted [forsyth06].

So, we can reduce the number of bytes sent per quaternion to 12. Despite the precision problems described above, this can be done without losing much precision.

As stated above, any of the three components can be removed.

If we care less about precision, it is possible to reduce the number of bytes to 4. This is done by converting the resulting direction vector (three components) into polar coordinates (pitch, yaw) and length. These allow for distributing the bits for yaw and pitch values according to their values. For example, when the pitch points straight up, the yaw value does not have to be very refined. On the other hand, a pitch of zero (pointing along the equator) requires more precision in the pitch and much less in the yaw.

The polar solution was tested, and deemed to imprecise for the dismounted trainer: it introduced a noticeable jitter in the movement of extremities (hands and feet). Because the dismounted trainer application is expected to run on a local network, the trade-off decision between precision and data rate was made in favor of precision. Nevertheless, for simulations running on wide-area networks or the Internet, this is a very viable form of compression.

7.4 Network delay

Because of the long path taken by the sampling data, the delays on the client between sampling and displaying that sample can be significant, up to 100 ms. Most importantly for the Dismounted Trainer, real-time training using full-body motion capture needs immediate real-time feedback to the user. The delays incurred in the system described so far were measured.

7.4.1 Delay measurement method

During each Xsens suit sample (box 1 in diagram 10 on page 29), two extra items of data were added to the sample:

- a unique sample ID: the serial number of the sample. This serial number is increased after every successful sample.
- the local time of the sample.

Then, this information is carried over from the client to the server (boxes 2 and 3 in diagram 10), which stores it, and after the physics processing tick (box 4), sends it back to the client along with the object (rag doll) update packet (boxes 5 and 6). At each step, the time is recorded, up until the rendering of the rag

doll.

The delays were measured on the re-lion standard single-user dismounted trainer setup, which is:

- Server: Intel P4, 3.2 GHz, 1024 MB memory
- Client: Dell Dimension 8400, 3.4 GHz, 512 MB memory, NVidia GeForce 6800, 256 MB memory
- Network: 1 Gbit direct crosslink cable

Note that during official dismounted trainer sessions, a wireless network is used. These delays were not measured, but have been observed to be noticeably larger than a LAN connection.

7.4.2 Hypothetical delay

Based on known delays, one can create a worst- and best case scenario:

Phase number	**Phase description**	**Delay (msec)**	**Total best - worst case (msec)**
1	Xsens sensor sampling	0	0
2	Frame delay polling (assumed 50 Hz frame rate)	0-20	0-20
3	Net send to server (LAN)	2-17	2-37
4	Server processing (30 Hz fixed rate)	0-33	2-70
5	Net sent to client (LAN)	2-17	4-87
6	Frame delay on client for rendering (assuming a 50 Hz frame rate)	0-20	4-107
7	1-2 potentially queued frames on client (assuming a 50 Hz frame rate) -- see below	0-40	4-147

The best case is very hypothetical: each loop in the system (network polling loops, server tick loop, client rendering loop) will have to be perfectly synchronized to make this happen.

It should be noted in phase 1 that the sample delay of the Xsens sensors themselves was not measured.

Because the client checks for availability of new Xsens orientation data once every frame, the potential delay suffered here is one frame, which is 20 milliseconds. A potential optimization here is to have a separate thread sample the sensors, which immediately independently from the main thread, sends the new orientations to the server. This was not implemented in the dismounted trainer.

The send delay from and to the server (phases 3 and 5) are always between 2-17 milliseconds. This is a real-life measurement.

The rendering frame rendering (phases 6) is potentially lost if the network message arrives just after polling the incoming network queue. The message will be processed one frame later, and thus the updated rag doll position will be drawn that frame.

Unfortunately, due to hardware render command queuing and frame buffer flipping, some graphics cards render as much as two frames ahead of what is actually showing on the screen (phase 7). The makes for two more potential frames lost. At an assumed framerate of 50 Hz (the current operating frame rate of the ITEC demo), this is another 60 milliseconds potentially lost.

7.4.3 Measurement results

A delay measurement was done during a dismounted trainer session. The results were measured over 3524 frames. The total a duration of the sample is 67.4 seconds, making the average frame rate 51.2 frames per second (19,5 milliseconds).

The mean delay was 50.7 milliseconds, with a standard deviation of 13.7 milliseconds. This quite large standard deviation signifies that the system's different loops are not running in sync at all. On the up side, it is always much less than the worst case of 147 milliseconds.

Below a graph is showing the bell curve that emerges when assessing delay vs. frequency of that delay.

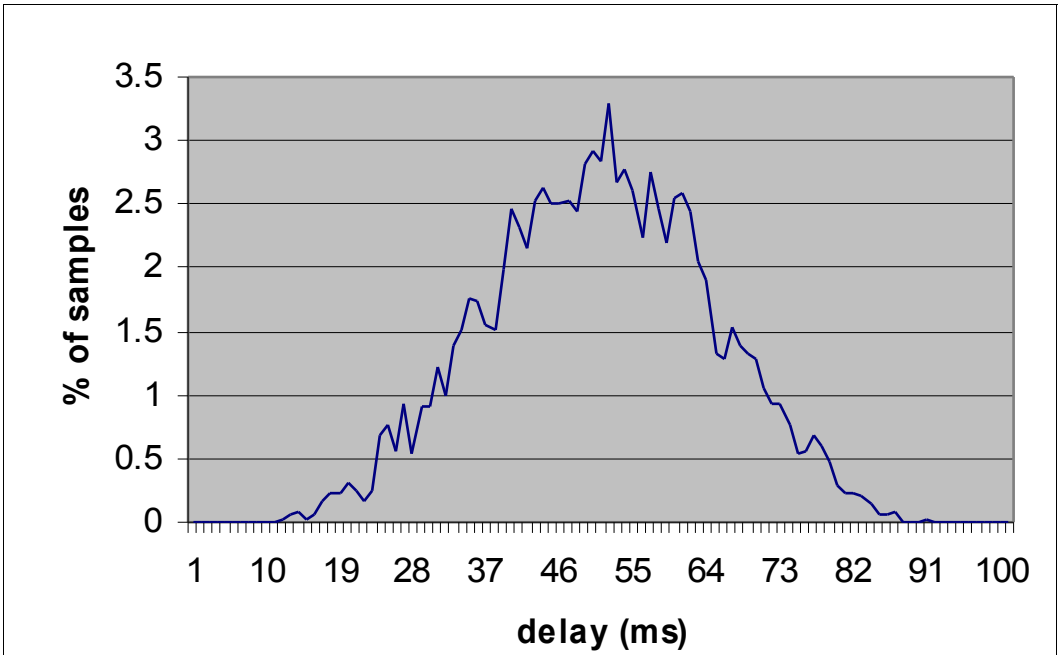


Diagram 24: Percentage of samples with a specific delay

7.5 Local feedback mode

As demonstrated above, the network delay is about 50 milliseconds. Because of the many steps a measurement takes, it is quite complex to reduce this delay: you would have to individually optimize each step.

The delay is most noticeable on the client that samples the motion capture suit. A typical dismounted trainer configuration means the participant wears the motion capture suit and a HMD to look around. This means the participant can see his/her own hands and limbs moving, delayed about 80 milliseconds. This breaks the feel of immersion and makes manipulating objects exceedingly difficult.

A way around this, at least on the sampling client, is implementing a local feedback mode. In short, the client will locally make the movements it expects the server to send.

The major downside is that, if the client-side movements turn out to be wrong, the rag doll limb orientations are corrected to the server-dictated orientations when the server update arrives in the client. Even though interpolation was used to smoothly correct the orientations, this still lead to jittery movement. Therefore, this method is not suitable for detailed interaction with the VR world.

However, it is suitable for the dismounted trainer. This is because in the dismounted trainer gestures and

body positions must be conveyed to other participants and interactions are mostly indirect, for example through weapons fire and explosions.

8 Rag doll actuation

'Rag doll actuation' for the purposes of this report means having a rag doll mimic the pre-animated real-time data that was captured from a client.

8.1 Direct-set method

The most straightforward way of integrating a pre-animated rag doll into a physics engine is brutally resetting the position of each limb to the position the kinematics dictate, before starting each physics frame.

8.1.1 Theory

Much of the theory has already been outlined in section 6.2, since we already have the limb orientations as quaternions we can just set them directly to the limbs of the rag doll.

There are some adjustments that need to be made for this to work:

1. the physics engine must not do any active physics processing on the limbs;
2. the positions of the limbs must be set (we only have the orientations now).

Placing a geom into the world without an associated body makes it a static geom. Forces on static geoms have no effect on it, effectively making it immovable, part of the static world. So to achieve the desired direct-set effect, the bodies associated with the rag doll are removed.

The second point (setting the limb positions correctly) is done by linearly traversing a list of joints and moving each limb so that the joint anchor points on both bodies match up.

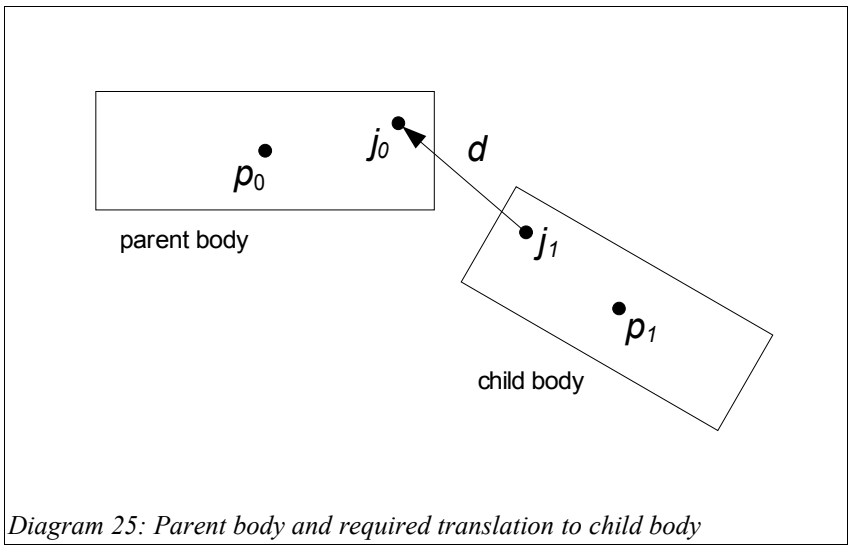


Diagram 25: Parent body and required translation to child body

Lets define the parent body as R_0 (rotation matrix) and $\vec{p}_0$ (position). To transform from local space to global space:

$$\vec{p}\,'_0 = R_0 \vec{l} + \vec{p}_0 \qquad \text{Eq. 1}$$

The child body is defined as R_1 and $\vec{p}_1$. $\vec{j}_0$ and $\vec{j}_1$ are the joint positions in **local body space**. This means they are local to the bodies. To transform them into global space we use eq 1:

$$\vec{j}'_i = R_i \vec{j}_i + \vec{p}_i \quad i = 0,1$$ Eq. 2

Now that both $\vec{j}'_0$ and $\vec{j}'_1$ are in the same coordinate space, we can safely subtract them to get vector $\vec{d}$:

$$\vec{d} = \vec{j}'_0 - \vec{j}'_1$$ Eq. 3

We need to move the child body with d. This results in the following transformation on the child body's position:

$$\vec{p}'_1 = \vec{p}_1 + \vec{d}$$ Eq. 4

Where $\vec{p}'_1$ is the new position of the body. Combining Eq. 2, 3 and 4 and simplifying yields:

$$\vec{p}'_1 = \vec{p}_0 + R_0 \vec{j}_0 - \vec{R}_1 j_1$$

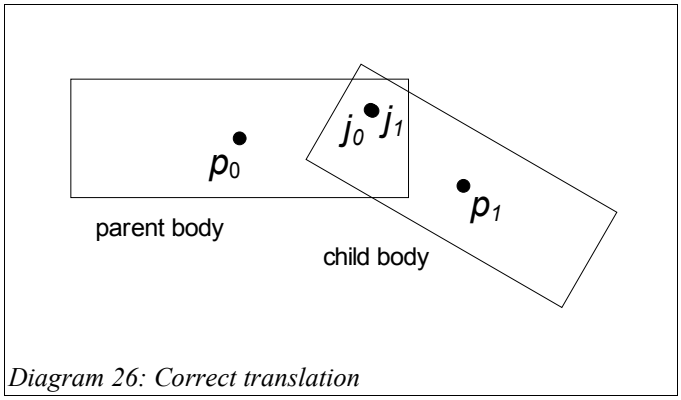


Diagram 26: Correct translation

Note that we should traverse the list of joints in a logical order. All lower-level bodies should be below higher-level bodies. For example, an upper arm should be processed before a lower arm. This is because the upper arm will function as a parent for the lower arm.

Recursion and a tree of joints can be used to enforce this, but a linear list was chosen for convenience of data notation and implementation.

8.1.2 Implementation results

Of course, one cannot expect any correcting behavior from the physics engine when there are no dynamics involved in the rag doll. The movements are exactly the same as without a physics engine. You can still look at the response the rag doll has on the physics world, as shown by the movie in illustration 6 below. A rudimentary walking algorithm is implemented here, described in section 8.4.1.

You can see the objects are perturbed somewhat by the rag doll. When the limb bodies are set directly into the dynamic bodies (in this case, the crates), ODE creates special non-penetration constraints (also known as joints – the terms joints and constraints are often used interchangeably in dynamics). This type of joint is called a contact joint.

The contact joint will be in violation of the constraint it should hold. The constraint it should hold is to keep the two bodies, the rag doll limb and the crate, in a non-penetrating state. The mechanism used in most dynamics engines to resolve these joint errors is introducing an error reduction parameter, ERP for short. The ERP controls how much force is exerted on the bodies to correct the joint error. Setting the ERP to 1.0 will fix the joint error as much as possible, within round-off errors and other internal approximations. Setting the ERP to 0 does not correct joint errors at all and will cause bodies to drift apart. The movie was made with an ERP of 0.2. The forces generated by the ERP are not enough to resolve the joint violations quickly enough to resemble natural movement. Setting the ERP to 0.9 perturbs

the crates more, but not enough to even push them aside.

Illustration 6: Direct-set method filmstrip. The rag doll walks through a set of crates, slightly perturbing them.

8.1.3 Conclusion

Directly setting the orientations of the limbs of a rag doll is a nice way of testing if the whole chain described in chapter 4 works, but of course does not actually use the physics engine in any meaningful way. Any collision response of the rag doll itself is absent and the collision response of other objects (should the rag doll geoms penetrate them) is inadequate.

8.2 Converting animations into forces

A controller is used to compare the current state of the rag doll to the state the kinematics dictate. Forces are applied to the rag doll to nudge it into the desired position.

8.2.1 Theory

Given a motion-captured animation set, how does one drive the physics engine so the resulting animation will be give same result?

What complicates the matter, of course, is that the forces calculated here are not the only forces working on the physics model. First, there's gravity and the normal forces from the ground, and, if the rag doll walks into a wall or otherwise collides with the worlds, the results can be unexpected indeed. So for the first iteration, we will disable these forces by disabling the gravity and having no objects in the rag doll its way.

To convert a kinematics frame to a dynamics forces and torques, we use this procedure, adopted from [ode01]. For each body of the current model, measure the angle relative to its parent in the animation hierarchy. Compare this to the same relative angle of the animation. This results in a relative angle error of the body. Calculate and apply the torque necessary to move the body to the angle relative angle of the animation.

Let's define k as the kinematic quaternion of some limb and q as the dynamic quaternion, that is that current orientation of the body representing the limb. The error quaternion between these two is:

$$e = q\,k^{-1}$$

To repeat, the definition of quaternions,

$$e = (\cos(\Theta/2), \vec{u} * \sin(\Theta/2))$$

where $\vec{u}$ is a rotation axis of unit length in $\mathbb{R}^3$ Cartesian space, and Θ is the angle the object is rotated along $\vec{u}$.

This means that $\vec{u}$ is the axis to rotate the body around to get it into orientation *k.* We can feed this value into the angular force accumulator, scaled by the angular distance to travel and a spring constant *k*:

$$\vec{\omega}_f = |\vec{u}| \Theta k$$

The value of the spring constant determines the strength of the spring, and thus the resulting animation and stability. A large spring constant value causes the rag doll to rigidly follow the animation and exert a large force if stopped (by another body or wall, for example). However, a large spring constant can cause instability.

Another problem with this method is that, when some rotations achieve a constant speed, some bodies may be off for many steps. A solution to this is to introduce an error accumulator, in the form of a leaky integrator[1]. The force added to the body is multiplied by the result of the error accumulator. The following equation can be used for a leaky integrator:

$$\frac{dx}{dt} = i - ax$$

- $x = x(t)$ is the angle error over time
- i is the input angle error
- a, a constant, represents the decay rate

In this case, we can use the Θ as angle error and compensate by multiplying $\vec{\omega}_f$ by the result of the leaky integrator.

8.2.2 Conclusion

One of the major problems with the forces method is that we have effectively created a spring, pushing the rag doll limbs into the animated position. Any time a spring is used, the programmer should be wary of the stability and usability of the simulation.

The first problem is that, even with a leaky integrator, it will take some simulation steps before the body is in the desired position, introducing (more) lag to the simulation.

The second problem is that the spring constant *k* and leaky integrator decay rate *a* from the discussion above must be chosen carefully. If it is too low the body will never reach its final orientation. If it is too high the body will take too large steps (overcompensating), and the simulation could become unstable and 'explode'.

The third and final problem is that, if other forces (such as gravity) are applied to the body, the body may never reach its final orientation. In some cases this can be beneficial for integrating the rag doll with the world, as described in the section above.

8.3 Angular motor

There are some big problems with the forces approach (see the section above). These problems are similar to the problems solved by motors and joints in a physics engine. These use implicit first-order integration to achieve similar effects, but without the downsides of joints (tweaking, instability, more steps to achieve desired state).

1A leaky integrator model is an integrator thats 'leaks' at a rate proportional to its value.

8.3.1 Theory

For this to work, we will need two sets of constraints on each rag doll joint. This means two joints are attached for each rag doll joint, one ball joint and one angular motor.

The first type of joint, the ball joint, keeps the bodies together, rotating around a specified joint position. Ball joints do not constrain any angular degrees of freedom; in other words, the bodies can freely rotate about the ball joint positions.

The second type of joint, the angular motor, controls the relative angular velocities between two bodies. Let's assume we have angular motors attached to all joints of a rag doll. If we tell the motor to keep the speed at zero, we will effectively have implemented dry (Coulomb) friction in the joints.

So now, instead of a force (which is an acceleration times mass), we can supply a speed to control the animation. We still need the spring, but now it is in the speed domain instead of the acceleration domain, making it more stable.

8.3.2 Conclusion

The angular motor is more stable than the springs/force method. However, the movement of the rag doll is still somewhat sluggish: if the motion capture feed moves too fast, the rag doll will still become unstable.

8.4 Walking

Many types of motion capture data do not have an absolute position recorded with them (see section 3.1.1). If the rag doll is modeled accurately enough, we can try to use the collision and dynamics responses from the physics engine to make the model walk around. Correct friction simulation is an important part of this process.

8.4.1 Lowest foot

In a nutshell, the lowest foot method simply checks which foot is lowest and then anchors the model around that point.

Theory

Let $\vec{c}_i, i=1..\mathrm{n}$ be a number of sample points near the feet. Below is a diagram showing where these coordinates are. They are stored as local coordinates to the left and right foot dBodies, respectively. Increasing the number of sample points (n) increases the definition of the foot contact algorithm, but for the purposes of this test x were chosen.

The reason the points at the front of the feet are not at the tip of the shoes is because this allows the feet to pivot at these points while walking, emulating the bending of the toes. This has the effect that the toes disappear a small amount into the ground, because they are part the the foot bone and not a separate bone or limb.

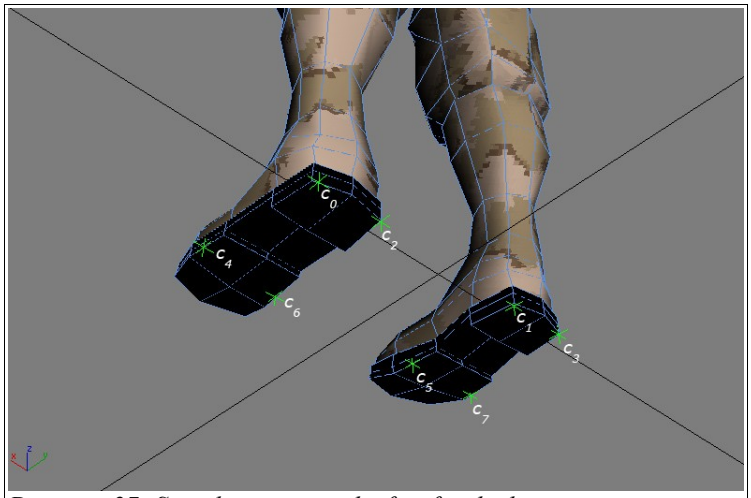


Diagram 27: Sample points on the feet for the lowest point measurement (feet seen from below ground position)

At run-time, these coordinates are translated from this local space of the foot limb they belong to global coordinates. The resulting global coordinate with the lowest (smallest) y axis is then designated as lowest point, thus the point that should be constrained to the ground.

Let's designate *j* as the index of the lowest point, as determined by the 'algorithm' defined above. This makes $\vec{c}_j$ the lowest point. To keep point $\vec{c}_j$ constant while the rest of the body moves, simply subtract the $\vec{c}_j$ from the positions of all limb of the rag doll for each frame.

$$\vec{p}\,' = \vec{p} - \vec{c}_j$$

Here $\vec{p}$ is the current position of the limb and $\vec{p}\,'$ the position to use for physics processing. Note that this calculation is done each frame and that $\vec{p}$ (the state) is unchanged.

Now the change from one contact point to another we need to be handled. If nothing is done, the rag doll will 'jump around' each contact point, each contact point being the center of the rag doll. If *j* changes from one frame to the next, it is clear that a new contact point to the lowest. Here, *k* is the old contact point index.

$$\vec{p} = \vec{p} + \vec{c}_j - c_k$$

In this case, the state of the bodies is modified to incorporate the offset between the old and new lowest point.

Evaluation

The main advantage of the lowest foot method is that it is simple to implement and looks reasonable. However, disadvantages are manifold:

- The algorithms only works for persons standing on their feet. Crouching, lying down and standing upside-down do not work, because only the feet contact points are checked.
- The technique (as described here) only works on a flat surface. One could ray-cast the points to collide with the rest of the virtual world and adjust the height of the rag doll accordingly. Of course ray-casting leaves open the possibility of missing important geometry with the ray cast. A better solution along these lines is presented in the next section.
- The wrong foot might the chosen as lowest foot, when the lowest points of the two feet are very close together and the motion-captured person is still moving. For example, this can happen, when he/she drags his/her feet. The result of this is incorrect movement of the rag doll.
- Walking does not look very natural; the character bounces up and down too much. This is due to

the tips of the toes not bending. It is possible to incorporate this bending into the algorithm.

8.4.2 ODE collision detection

The lowest point is the 'direct set method' of walking: it does not use the physics engine to generate contact points or more the rag doll. A first step is using the contact point generation algorithms to:

- enable walking over uneven terrain
- increase the definition of collision detection: because ODE uses solid collision primitives, the rag doll will not fall through uneven terrain

A detail of the collision mesh illustrates the concept in diagram 28 below. The blue wireframe represents the collision geom primitives. The green wireframe is the 'mesh world' mesh. Unrelated to this discussion, the red wireframe represents the bone structure.

The physics engine (ODE) can now generate the contact points using the collision detection. Normally these are used to generate contact joints (see chapter 5). Using these contact points results in a variable number of contact points each frame (n), which can then be processed in the same manner detailed in the section above.

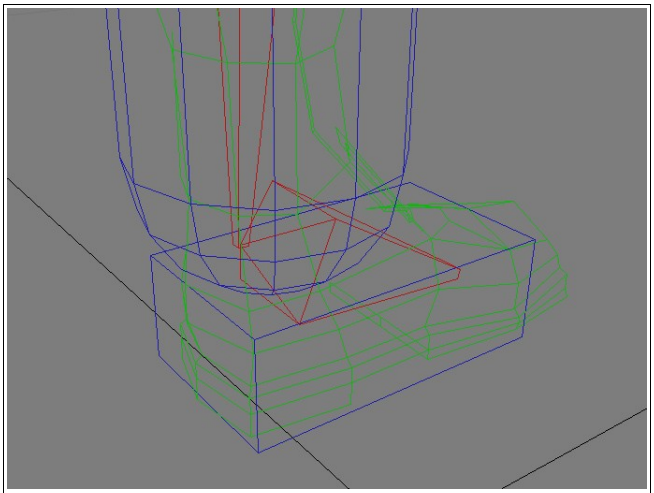


Diagram 28: Left foot of soldier, with collision primitives and bones

8.4.3 Invisible pendulum model

When physics are enabled using the force/spring method (see 8.2) or angular motors (see 8.3), it is possible to directly move the rag doll by letting the physics do their job. A problem is that, unless the animation is impossibly perfect, the rag doll will fall over almost immediately. A simple, yet inaccurate solution is attaching a body with the center of mass to way outside and below the rag doll. This weight works as a pendulum, keeping the rag doll upright.

A disadvantage to the invisible pendulum method is that it further increases the amount of force both the force/spring method and angular motor method need to generate, and thus make them (even) less stable.

Another disadvantage is that the distribution of weight is altered, which results in not very natural movement.

8.4.4 Self-righting constraints

Instead of using a pendulum-like construction, one could also attach an extra angular motor on each limb, or on a set of major limbs, such as the pelvis and torso, to keep the rag doll upright. The angular motors are also attached to the static world. The error angle is measured and feedback force is supplied to keep the torso upright. This method should have as an advantage that the pendulum will never 'sway'. This method has not been implemented.

9 ITEC/Dismounted Trainer demo

The ITEC/dismounted trainer demo is an application that allows two (or more) people, wearing motion capture suits, to visually interact with each other and the environment. It shows vehicle dynamics by allowing people to drive around in a tank.

The demo demonstrates Lumo Scenario physics, scripting and multi-user capabilities.

9.1 Tank physics

The tank uses a set of ten spheres as wheels. The reason for this is that this configuration provides excellent stability and tank-like vehicle behavior at a modest performance cost.

Alternative configurations are:

- Individually modeling tank tread links. Far too costly in CPU terms (the number of contact points and the physics engine's LCP matrix would become huge) and very unstable.
- Using ray-casting and a feedback loop to model a “hovertank”, emulate tread contacts with the ground. This interesting approach takes very little CPU time. However, it is more complex to implement and it remains to be seen if tank-like behavior is possible. This is why an implementation was not attempted. Certainly an option if many tanks need to be simulated. See [watte06].
- One can create rigid bodies shaped like tank treads. The contact points these bodies generate could be modified to transport the bodies forward, much like an up-side-down conveyor belt. This is quite a low performance overhead technique, but creating adequate suspension is a challenge.

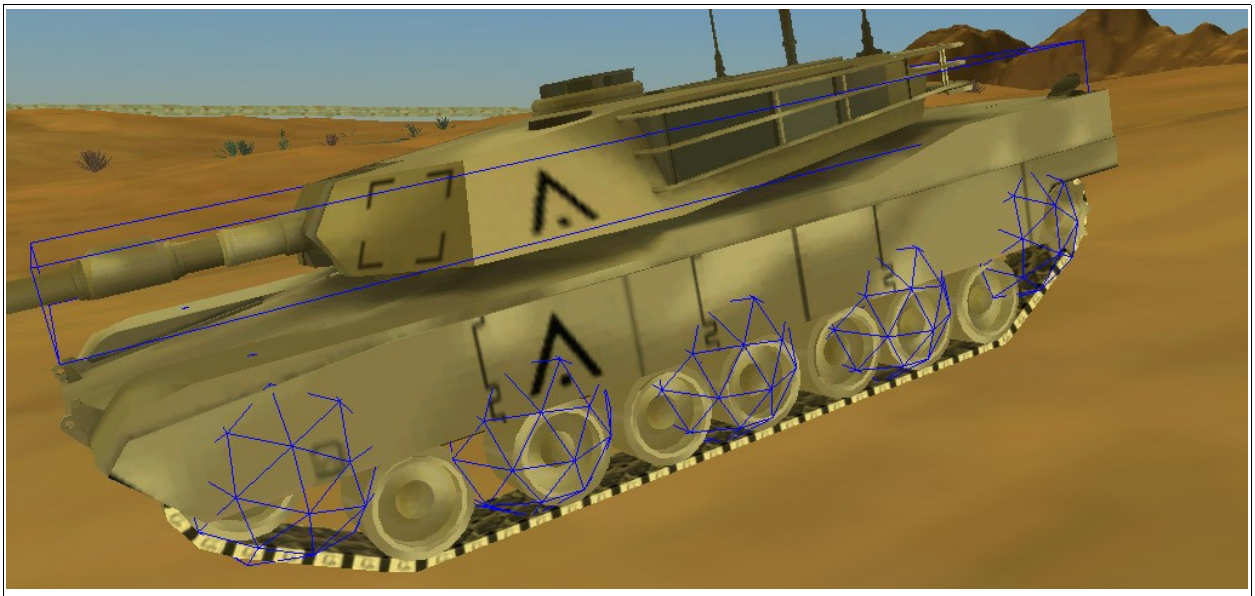


Illustration 7: Tank with sphere 'treads'

We believe using spheres a collision primitives results in an acceptable trade-off between physics quality, CPU resource usage and implementation time.

The spheres are connected to the main body using dampened slider joints. This means they can move up and down individually. An angular motor joint on each sphere controls the angular velocity of each sphere, and with that the forward and backward movement of the tank.

The contact joints generated by the inner 'tread' spheres allow for considerable sideways slipping. This is

because tanks corner by letting one 'tread' move faster than the other, effectively slipping around corners. If slipping in the contacts joints is disallowed, the tank cannot turn at all!

The front and back two spheres are elevated somewhat and do not normally touch the ground when driving around. They do help when climbing obstacles and maneuvering hilly terrain. These spheres slip considerably less than the others, to allow for easier climbing of steep obstacles.

The tank also includes a special client-side visualizer. A visualizer is an object that interprets the object update messages from the server into objects to render. The tank visualizer includes the following special effects:

- rotating turret
- exhaust valves that bounce according to engine rev
- diesel fume particle systems from the exhausts, taking into account the engine rev
- dust clouds from the tank treads ground-contact points, taking into account the tank velocity
- tank commander hatch opens and closes
- the tank tread meshes are separate meshes, rotated and translated according to the mean position of eight sphere wheels
- tank tread textures in these meshes are animated. These animations are driven by the individual speed of each tread (left or right)
- the sixteen mesh wheels and two cogs (at the end of the tank) are animated. These animations are also driven by the individual speed of each tread (left or right), and each cog meshes with the treads in the texture

This shows the power of using specialized network message handlers of each type of object: special effects can be programmed at client-side, without increasing network bandwidth.

The particles effects are created using a custom particle system described below.

9.2 Particle dynamics

Newly developed for the dismounted trainer/ITEC demo is a particle system, called Subatomic. Subatomic plugs into Lumo Scenario (and other Lumo applications).

Particle systems, in a real-time VR simulation context, are usually a cloud of many facing quads or triangles. “Facing”, in this context, means that the particle always faces the viewer.

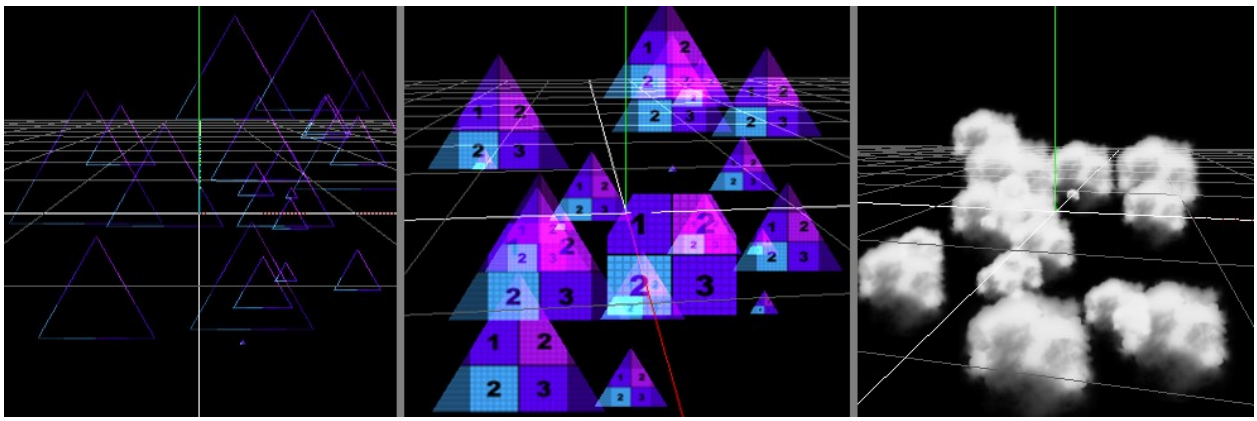


Illustration 8: Particle wireframe, textured saturated triangle particles, textured alpha-blended quad particles

Each particle has some properties that determine its appearance:

- Texture
- Color
- Size
- 3D position
- Rotation
- Topology - quad or triangle
- Blend mode - alpha blending or saturated blending. Alpha blending uses a mask to blend different parts of the particle differently with the background. Saturated blending adds the particle to the background.

Each particle also has some other characteristics, that are dependent on what type of effect the particle system is supposed to model. These can include:

- Speed
- Weight
- Time-to-live
- Behavior

An interesting feature of this particle system is that these characteristics are modeled using an approach similar to vertex shaders. The rendering of the particle is done in the base system. The logic of the particle is separated into a .c file, which is compiled on the fly using a modified version of TCC [bellard05]. TCC (TinyCC) is a small, open-source x86 compiler, used here as a back-end for run-time code generation.

The particle system (called Subatomic) compiles the .C code into machine code at run-time. This has the following advantages:

- Speed: much faster than scripting.
- Flexibility: the system can re-load and re-compile a particle C file. This means that you can immediately check the results of your changes, without restarting the simulation.
- Ease of use: isolating the particle behavior in a single file enables non-programmers to create and tune their own particle effects, easing the burden on programmers.

Disadvantages:

- Because particle C files are machine code (and not interpreted scripts or byte-codes, such as Lua), it is possible to crash a process using an incorrect particle C file.
- TCC does not yet generate SIMD (Single-instruction, multiple data) code, which would be ideally suited for these kind of calculations.

An example .pc (particle C) file:

```
typedef struct
{
  float position[3];
  float size;
  float speed[3];
  float life;
} my_particle_t;
```

```
const char* texture = "flare2.bmp";
int count = 300;
BLENDTYPE blendtype = BLENDTYPE_SATURATED;
PARTICLETYPE particletype =
PARTICLETYPE_TRI_UNIQUESIZE_UNROTATED_SINGLECOLOR;
unsigned long constant_color = 0x8f3403;
int particledatasize = sizeof(my_particle_t);
int emitterdatasize = sizeof(EmitterData);

void init(EmitterData* emitter, my_particle_t* particle)
{
  particle->position[0] = rnd_getrange(-1, 1);
  particle->position[1] = 0;
  particle->position[2] = rnd_getrange(-1, 1);
  particle->speed[0] = rnd_getrange(-2, -31);
  particle->speed[1] = rnd_getrange(18, 20);
  particle->speed[2] = rnd_getrange(-15, 15);
  particle->size = rnd_getrange(1,2);
  particle->life = rnd_getrange(0,.6);
}

int update(EmitterData* emitter, my_particle_t* particle)
{
  particle->position[0] += particle->speed[0] * emitter->timepast;
  particle->position[1] += particle->speed[1] * emitter->timepast;
  particle->position[2] += particle->speed[2] * emitter->timepast;

  particle->speed[1] -= 40 * emitter->timepast;
  particle->size -= emitter->timepast*0.5;

  particle->life -= emitter->timepast;
  if(particle->life < 0) init(emitter, particle);

  if(particle->position[1] < 0) particle->speed[1]*=-1;

  return 0;
}
```

A small application, called Subatomic Studio, was created. It immediately recompiles a particle C file after every key press. Because TCC compiles such a small file in manner of microseconds, results (including error messages) are immediate.

A screen shot of Subatomic Studio in action:

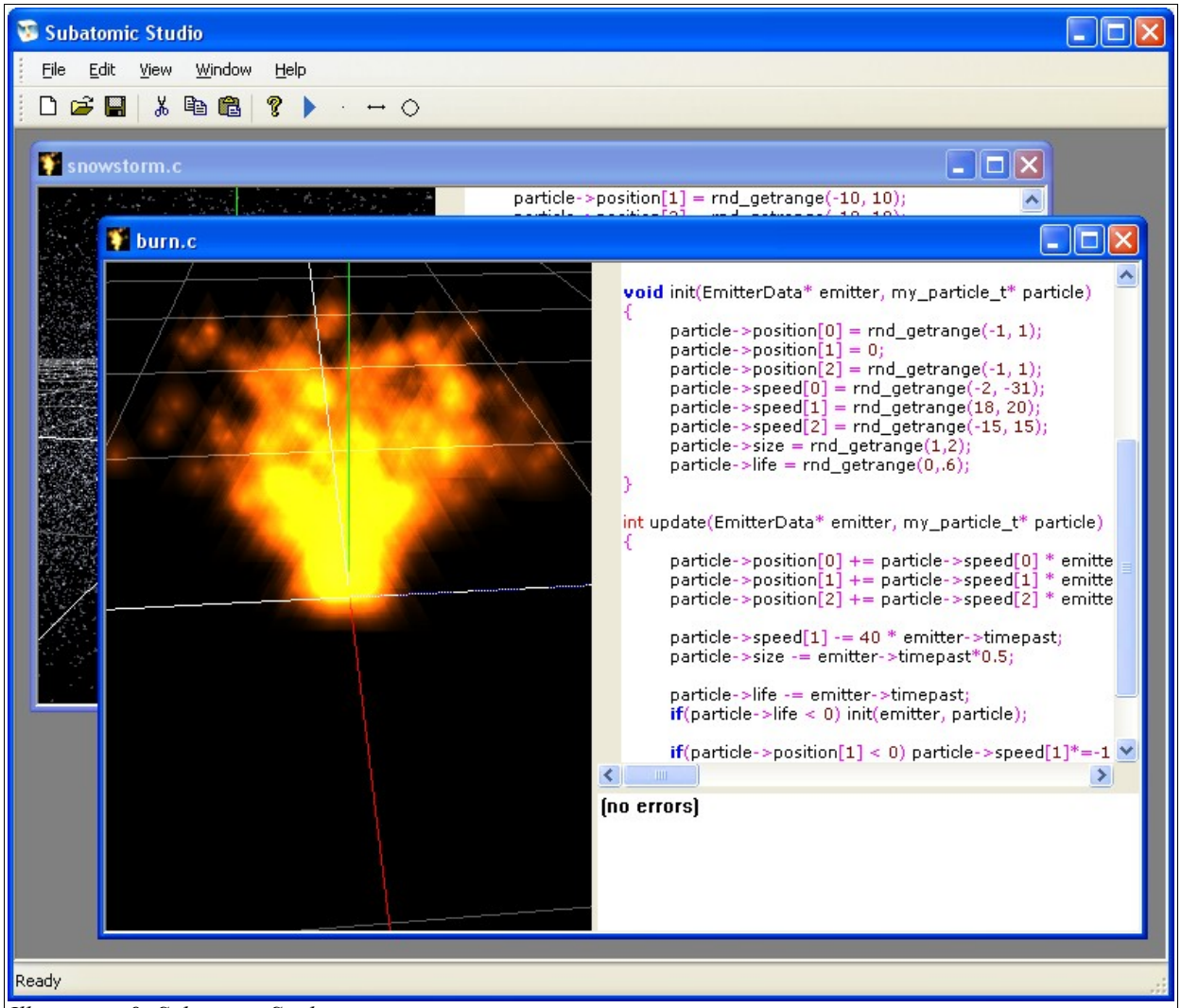


Illustration 9: Subatomic Studio

9.3 Demo screenshots

All models and textures in the screenshots are made by the re-lion graphics department, being Edwin van het Bolscher and Bart Wttewaall.

The main areas in the ITEC demo world are a desert, a desert town, and a hangar. The desert has moderate bumps and hills to show off tank physics. The town was made to demonstrate tank behavior in more confined spaces, and to show the city- and road-building capabilities of Lumo Builder, our VR world creation tool. The hangar was integrated from a previous demo as an area to experiment with the dismounted trainer. This factory-like structure contains crates and other objects to interact with.

Illustration 10: The avatar sitting in a tank hatch. One user drives the tank, the other the ragdoll in the tank hatch

The actuated rag doll was, in this case, put into a tank hatch to underline the fact that both the tank operator and the motion captured person were in the same world.

The next screen shot shows the rag doll and a part of the hangar most of the dismounted trainer was developed in.

Illustration 11: The dismounted trainer avatar on its own. The tank is driving around elsewhere

10 Conclusion

10.1 The good and the bad

The client/server architecture has proved itself to be very extendable and maintainable. Because functionality is so strictly separated, computing power is efficiently distributed between clients and servers. The server calculates all simulation logic and physics, the clients do the graphics rendering. This scales very well, as adding more users entails connecting more clients to the server, meaning more computing power in the total system.

Using extrapolation for object updates greatly increases smoothness of movement, and increases the number of objects that can be sent to a server, because it greatly decreases necessary network bandwidth. Network and processing delays remain a problem but can be reduced, at significant engineering effort, by client-side prediction and processing.

Using physics engines for full-body motion capture integration in 3D worlds was not that fruitful. While it should be possible, in theory, to actuate a rag doll instead of using the direct-set method, some problems remain:

- stability problems: the rag doll (virtual avatar) needs springs or motor forces to assume the correct position. Tuning these motors and springs is exceedingly difficult.
- precision problems: despite usage of leaky integrators and other stop-gap measures, limbs tend to be out of sync with the motion captured source more than is generally acceptable.

Another, more general problems identified was if one was to walk a distance using the dismounted trainer, you could bump into a real wall! The real environment will usually differ from the virtual environment.

One of the easier solutions to use in production-level simulators would be not using a completely articulated rag doll at all, but an approximation, such as a capped cylinder (a cylinder with semi-spheres as caps), standing on its side, moving around. This is an approach common in current-generation games (Quake 4, Half Life 2, etc). Interaction is usually done by proxy. 'Proxy' in this case usually means weapons to blow stuff up and shoot crates, monsters, etc. Half Life 2 has shown interesting indirect manipulation possibilities by way of a 'gravity gun'.

The full-body motion capture then, seems more suitable for the following tasks:

- visual interaction with team mates
- exact replay of body movement for later analysis
- interaction with the environment, mainly through weapons and other tools

The sheer volume of the system required to start experimenting with real person-to-person interactions has limited the amount of time available for more creative experimenting, especially in the area of integrating the rag doll motion capture into the physics engine.

10.2 Near-future products

The base multi-user system is now up and running, and there are already next-generation simulations being developed with it. A simulator also developed on the Footprint platform during the time frame of this project, besides the ITEC/Dismounted Trainer demo presented in chapter 9, is a scootmobile simulator. This simulator has been written for a different project, a co-operation between Roesingh'

Research & Development, the University of Twente and re-lion.

Also in developed at re-lion during the same time, but not by this author, is a content creation tool that can drastically speed up creation of VR worlds using a point-and-click interface. This product is called Lumo Builder. re-lion hopes that more useful simulations based on Lumo Scenario and/or Builder will be developed in the future.

Acknowledgments

Many thanks to Per Slycke of Xsens and Koen Tan of TNO, for providing access to their motion capture suits every now and then. Also my thanks to Job Zwiers and Herman van der Kooij.

While creating a good graphics engine is nice, it can only shine with the help of expert content creators. Many of the screenshots in this report contain models and textures made by graphics wizards Edwin van het Bolscher and Bart Wttewaall, the re-lion graphics dept.

Also my gratitude to Chris and Steven, for putting up with me frequently being unavailable for other projects. To Alex for input and work on Footprint Scenario, and the rest of re-lion for being a generally awesome bunch of people.

Many kudos to Herbert for proof-reading.

Last but not least, many thanks to my family and friends.

11 Appendices

11.1 References

[aronson97] Jesse Aronson, *Dead Reckoning: Latency Hiding for Networked Games*, http://www.gamasutra.com/features/19970919/aronson_01.htm

[baraff97] David Baraff and Andrew Witkin, *Physically Based Modeling: Principles and Practice,* SIGGRAPH 1997 Course Notes, http://www-2.cs.cmu.edu/~baraff/sigcourse/index.html

[baraff95] David Baraff, *Coping with Friction for Non-penetrating Rigid Body Simulation,* Computer Graphics, 25(4), p. 31

[bellard05] Fabrice Bellard, *TinyCC,* http://fabrice.bellard.free.fr/tcc/

[bew] *Dictaat Bewegingssturing*, 2003-2004, CTW, Biomechanische Wetenschappen, Universiteit Twente

[cave] Dave Pape, *The CAVE Virtual Reality System,* http://www.evl.uic.edu/pape/CAVE/

[end] Natural Motion endorphin http://www.naturalmotion.com/

[feyn] Feynman, Leighton, and Sands, *The Feynman Lectures On Physics, Volume 1*, , Addison Wesley, ISBN 0805390464

[forsyth06] Tom Forysth, *A matter of precision*, http://home.comcast.net/~tom_forsyth/blog.wiki.html#%5B%5BA%20matter%20of%20precision%5D%5D

[gamma95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Object-Oriented Software,* ISBN 0-201-63361-2

[gleicher99] Michael Gleicher, *Biovision BVH,* http://www.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/BVH.html

[hecker96] Chris Hecker, *Physics, part 1-4: The New Frontier, Angular Effects, Collision Response and The Third Dimension,* http://www.d6.com/users/checker

[kirmse04] Andrew Kirmse, *Game Programming Gems 4,* ISBN 1-58450-295-9

[kuiper98] Jack B. Kuipers, *Quaternions and Rotation Sequences : A Primer with Applications to Orbits, Aerospace and Virtual Reality,* Princeton University Press, ISBN 0691102988

[lua01] The Programming Language Lua http://www.lua.org

[lua02] Roberto Ierusalimschy, *Programming in Lua,* ISBN 8590379817, http://www.lua.org/pil

[ode01] http://ode.org/cgi-bin/wiki.pl?PhysicalAnimation

[ode02] Russel Smith, *How to make new joints in ODE,* http://www.ode.org/joints.pdf

[ode03] Russel Smith, *ODE User Guide,* http://www.ode.org/ode-0.5-userguide.html

[shapiro03] Ari Shapiro, Fred Pighin, Petros Faloutsos. *Hybrid Control for Interactive Character Animation*, p. 455

[treglia02] Dante Treglia (editor), Mark Zarb-Adami (author), *Game Programming Gems 4,* ISBN 1584502339, p. 187

[watte06] Jon Watte, *Raycar example*, http://www.mindcontrol.org/~hplus/raycar/

[wiki2006] Wikipedia, *IEEE floating-point standard*, http://en.wikipedia.org/wiki/IEEE_floating-point_standard

[winston93] Wayne L. Winston, *Operations research: applications and algorithms*, Internation Thomson Publishing, ISBN 0-534-20971-8

[witkin97] Andrew Witkin and David Baraff, *Physically Based Modeling: Principles and Practice* http://www.cs.cmu.edu/~baraff/sigcourse/index.html

[xsens01] Xsens Xbus Master System product description, http://www.xsens.com/index.php?mainmenu=products&submenu=human_motion&subsubmenu=Xbus_Master

11.2 Diagram and illustration index

Diagram index

Illustration index

11.3 Reading BVH files into a Lua table

The .bvh format [gleicher99] is a text-based motion capture format. The transformations described in the .bvh file can sometimes be a bit tricky, and a lot of text parsing is required to read the format. Because text parsing is best done in scripts, I have created a Lua script that:

- Reads and validates the .bvh file
- Executes the relative transforms to generate world-space quaternions. We need worldspace quaternions because the physics engine does not use hierarchies

Each segment has an 'offset' vector, a 'rotation' list (containing one non-relative (worldspace) quaternion for each frame of animation), and an optional 'segments' list, which contains sub-segments.

The .bvh format was used to store and play back motion captures, because most of the time access to the available motion capture suits were limited, and working in the motion capture suit while at the same time programming a simulator is very tiring.

First the script is given, then an example .bvh file and the resulting Lua table.

```
--[[
   Parses a .bvh (motion capture) file into Lua table format.
--]]


-- A list of valid BVH transform tokens and their
-- quaternion sequence equivalents.
local xform_tokens =
{
   Xrotation = "rx",
   Yrotation = "ry",
   Zrotation = "rz",
   Xposition = "tx",
   Yposition = "ty",
   Zposition = "tz"
}

-- Main Function
BVH2Table = function(bhvfilename)
   local bhvfile, linenumber, bvhtable, levels_temp_info, line, rest, _ =
         assert(io.open(bhvfilename)), 0, {}, {}

   local function readline()
       repeat
           linenumber = linenumber + 1
           line = bhvfile:read("*line")
           -- trim
           line = line and string.gsub(string.gsub(line, "^[ \t]*", ""), "[ \t\n]*$", "") or nil
           -- enable comments and while lines
       until not line or (line ~= "" and string.sub(line, 1, 1) ~= "#")
       _,_,token = string.find(line or "", "^([%u%l{}:]+)")
       -- rest of line, stripped
       _,_,rest = string.find(line or "", "^[%u%l{}:]+[ \t]([%a%d%s%c%p]*)")
   end

   local function parse_error(err, ...)
       error(string.format("%s:%d: %s", bhvfilename, linenumber, string.format(err, unpack(arg))),
0)
   end

   local function checktoken(ctoken)
       if ctoken ~= token then
           parse_error("token %s expected, got %s", ctoken, token)
       end
   end

   bvhtable.root_segment = {}

   local function read_joint_level(level)
       local level_temp_info = { rotation_seq = {}, level = level }
       -- stored info for use at animation frame decoding time
       table.insert(levels_temp_info, level_temp_info)
       level.rotation = {}  -- one for each animation frame
```

```
        checktoken("{") readline()
        while token ~= "}" do
            if token == "OFFSET" then
                local ox, oy, oz
                _, _, ox, oy, oz = string.find(rest, "[ \t]*([%-%+%d%.]+)[ \t]+([%-%+%d%.]+)[
\t]+([%-%+%d%.]+)")
                level.offset = vec4(ox, oy, oz)
            elseif token == "CHANNELS" then
                local _, e, channelcount = string.find(rest, "([%d]+)[ \t]*")
                for i=1, channelcount do
                    local chname
                    -- extract channel name (such as "Xrotation")
                    _, e, chname = string.find(rest, "([%w]+)[ \t]*", e)
                    local token = xform_tokens[chname] or
                        parse_error("unknown channel type %s", chname or "(nil)")

                    if string.sub(token, 1,1) ~= "t" then  -- ignore translation tokens.
                        table.insert(level_temp_info.rotation_seq, token)   -- "rx", "ry", or "rz"
                        table.insert(level_temp_info.rotation_seq, 0)       -- spot to put the
angle
                    end
                end
            elseif token == "JOINT" or token == "End" then
                local newlevel = {}
                level.segments = level.segments or {} -- make sure we have a table there
                level.segments[(token == "End") and "EndSite" or rest] = newlevel  -- new level
                readline()
                read_joint_level(newlevel)
            elseif token ~= "}" then
                parse_error("illegal/unknown token '%s'", token)
            end
            readline()
        end
    end

    -- Read hierarchy
    readline()
    checktoken("HIERARCHY")  readline()
    checktoken("ROOT")  readline()
    read_joint_level(bvhtable.root_segment)
    readline()

    -- Read frames (insert them into the right places of the hierarchy)
    checktoken("MOTION")
    readline()

    checktoken("Frames:")
    _,_,bvhtable.framecount = string.find(rest, "([%d]+)")
    readline()

    checktoken("Frame")
    _,_,bvhtable.frametime = string.find(rest, "([%d%.]+)")
    readline()

    for fr = 1, bvhtable.framecount do
        if token or not line then parse_error("frame data expected") end
        local e = 1

        -- Skip global translation - we don't need it
        for l = 1, 3 do
            local n
            _,e,n = string.find(line, "([%-%+%d%.]+)[\t ]*", e)
        end

        -- fill in the rotation sequence in rotation_seq with
        -- the right angles, turn it into a quaternion
        for l = 1, table.getn(levels_temp_info) do
            local rs = levels_temp_info[l].rotation_seq
            for c = 2, table.getn(rs), 2 do
                local n
                _,e,n = string.find(line, "([%-%+%d%.]+)[\t ]*", e)  -- capture number
                rs[c] = math.rad(tonumber(n)) -- insert right number into rotation sequence
            end
            -- constructs quad from rotation sequence
            levels_temp_info[l].level.rotation[fr] = quat(unpack(rs))
        end
        readline()
    end

    return bvhtable
```

```
end
```

This is one of the .bvh files tested. This BVH file has only two frames of animation:

```
HIERARCHY
ROOT Hips
{
	OFFSET 0.00	0.00	0.00
	CHANNELS 6 Xposition Yposition Zposition Zrotation Xrotation Yrotation
	JOINT Chest
	{
		OFFSET	0.00	5.21	0.00
		CHANNELS 3 Zrotation Xrotation Yrotation
		JOINT Neck
		{
			OFFSET	0.00	18.65	0.00
			CHANNELS 3 Zrotation Xrotation Yrotation
			JOINT Head
			{
				OFFSET	0.00	5.45	0.00
				CHANNELS 3 Zrotation Xrotation Yrotation
				End Site
				{
					OFFSET	0.00	3.87	0.00
				}
			}
		}
		JOINT LeftCollar
		{
			OFFSET	1.12	16.23	1.87
			CHANNELS 3 Zrotation Xrotation Yrotation
			JOINT LeftUpArm
			{
				OFFSET	5.54	0.00	0.00
				CHANNELS 3 Zrotation Xrotation Yrotation
				JOINT LeftLowArm
				{
					OFFSET	0.00	-11.96	0.00
					CHANNELS 3 Zrotation Xrotation Yrotation
					JOINT LeftHand
					{
						OFFSET	0.00	-9.93	0.00
						CHANNELS 3 Zrotation Xrotation Yrotation
						End Site
						{
							OFFSET	0.00	-7.00	0.00
						}
					}
				}
			}
		}
		JOINT RightCollar
		{
			OFFSET -1.12	16.23	1.87
			CHANNELS 3 Zrotation Xrotation Yrotation
			JOINT RightUpArm
			{
				OFFSET -6.07	0.00	0.00
				CHANNELS 3 Zrotation Xrotation Yrotation
				JOINT RightLowArm
				{
					OFFSET	0.00	-11.82	0.00
					CHANNELS 3 Zrotation Xrotation Yrotation
					JOINT RightHand
					{
						OFFSET	0.00	-10.65	0.00
						CHANNELS 3 Zrotation Xrotation Yrotation
						End Site
						{
```

```
                                        OFFSET  0.00  -7.00   0.00
                                    }
                                }
                            }
                        }
                }
        }
        JOINT LeftUpLeg
        {
                OFFSET  3.91   0.00   0.00
                CHANNELS 3 Zrotation Xrotation Yrotation
                JOINT LeftLowLeg
                {
                        OFFSET  0.00  -18.34  0.00
                        CHANNELS 3 Zrotation Xrotation Yrotation
                        JOINT LeftFoot
                        {
                                OFFSET  0.00  -17.37  0.00
                                CHANNELS 3 Zrotation Xrotation Yrotation
                                End Site
                                {
                                        OFFSET  0.00  -3.46   0.00
                                }
                        }
                }
        }
        JOINT RightUpLeg
        {
                OFFSET -3.91   0.00   0.00
                CHANNELS 3 Zrotation Xrotation Yrotation
                JOINT RightLowLeg
                {
                        OFFSET  0.00  -17.63  0.00
                        CHANNELS 3 Zrotation Xrotation Yrotation
                        JOINT RightFoot
                        {
                                OFFSET  0.00  -17.14  0.00
                                CHANNELS 3 Zrotation Xrotation Yrotation
                                End Site
                                {
                                        OFFSET  0.00  -3.75   0.00
                                }
                        }
                }
        }
}
MOTION
Frames:    2
Frame Time: 0.033333
 8.03   35.01  88.36 -3.41   14.78 -164.35       13.09  40.30 -24.60  7.88   43.80  0.00  -
3.61   -41.45  5.82   10.08  0.00   10.21  97.95 -23.53 -2.14  -101.86       -80.77 -98.91
0.69    0.03   0.00  -14.04  0.00  -10.50 -85.52 -13.72 -102.93        61.91 -61.18  65.18 -
1.57    0.69   0.02   15.00  22.78 -5.92   14.93  49.99  6.60   0.00  -1.14   0.00  -16.58 -
10.51  -3.11   15.38  52.66 -21.80  0.00  -23.95  0.00

 7.81   35.10  86.47 -3.78   12.94 -166.97       12.64  42.57 -22.34  7.67   43.61  0.00  -
4.23   -41.41  4.89   19.10  0.00   4.16   93.12 -9.69  -9.43   132.67       -81.86  136.80
        0.70   0.37   0.00  -8.62   0.00  -21.82 -87.31 -27.57 -100.09        56.17 -61.56
58.72  -1.63   0.95   0.03   13.16  15.44 -3.56   7.97   59.29  4.97   0.00   1.64   0.00  -
17.18  -10.02 -3.08   13.56  53.38 -18.07  0.00  -25.93  0.00
```

Calling `BVH2Table("Example1.bvh")` results in this Lua table (truncated for brevity):

```
framecount = 2,
frametime = 0.033333,
root_segment =
{
   offset = vec4 (0, 0, 0, 0),
   rotation =
   {
```

```
        1 = quat (-0.11577000161481, 0.98634074464164, 0.1308934246975, 0.11462942471842),
        2 = quat (-0.092123745446054, 0.99079162002462, 0.11519387419897, 0.096296434110259)
    },
    segments =
    {
        Chest =
        {
            offset = vec4 (0, 5.21, 0, 0),
            rotation =
            {
                1 = quat (-0.67846116148714, 0.12346290260154, -0.018118578146337,
0.80173868330825),
                2 = quat (-0.71002633530165, 0.10297149585287, -0.016228624073583,
0.78534731626565)
            },
            segments =
            {
                LeftCollar =
                {
                    offset = vec4 (1.12, 16.23, 1.87, 0),
                    rotation =
                    {
                        1 = quat (0.0078171064286399, -0.088637181342426, -0.087502717571974,
0.99218225006713),
                        2 = quat (0.006021618823607, -0.035791871462232, -0.16579892649827,
0.98549144999639)
                    },
                    segments =
                    {
                        LeftUpArm =
                        {
                            offset = vec4 (5.54, 0, 0, 0),
                            rotation =
                            {
                                1 = quat (0.25104554088208, 0.31639235427109, -0.70958017540797,
0.61232540952431),
                                2 = quat (0.056273758461838, 0.17854386715035, -0.72064600783116,
0.67287313254332)
                            },
                            segments =
                            {
                                LeftLowArm =
                                {
                                    offset = vec4 (0, -11.96, 0, 0),
                                    rotation =
                                    {
                                        1 = quat (1.0258909101019, -0.3453428852374,
-0.25181944097625, 0.42973722992851),
                                        2 = quat (0.93886074109298, 0.39227367663586,
0.38287494672031, 0.49726498409251)
                                    },
                                    segments =
                                    {
                                        LeftHand =
                                        {
                                            offset = vec4 (0, -9.93, 0, 0),
                                            rotation =
                                            {
                                                1 = quat (-0.00052358926859083, -3.1527711529578e-
006, -0.006021348914081, 0.99998176870407),
                                                2 = quat (-0.0064575696942592, -3.9447539165837e-
005, -0.0061085188626025, 0.9999657042768)
                                            },
                                            segments =
                                            {
                                                EndSite =
                                                {
                                                    offset = vec4 (0, -7, 0, 0),
                                                    rotation =
                                                    {
                                                        1 = quat (0, 0, 0, 1),
                                                        2 = quat (0, 0, 0, 1)
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
```

```
                },
                Neck =
                {
                    offset = vec4 (0, 18.65, 0, 0),
                    rotation =
                    {
                        1 = quat (-0.71735991013929, -0.049407855716724, -0.054194112699621,
0.78685227788826),
                        2 = quat (-0.71471783287338, -0.04791004629929, -0.052869941930909,
0.78870911718489)
                    },
                    segments =
                    {
                        Head =
                        {
                            offset = vec4 (0, 5.45, 0, 0),
                            rotation =
                            {
                                1 = quat (0.68257456765686, -0.069016051683096, 0.043433909627014,
0.80788328327002),
                                2 = quat (0.68201735316684, -0.065119491426392, 0.044949626769842,
0.80845129359348)
                            },
                            segments =
                            {
                                EndSite =
                                {
                                    offset = vec4 (0, 3.87, 0, 0),
                                    rotation =
                                    {
                                        1 = quat (0, 0, 0, 1),
                                        2 = quat (0, 0, 0, 1)
                                    }
                                }
                            }
                        }
                    }
                },
                    ... etc ...
```