Technische Universität München

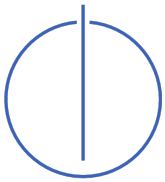
Department of Computer Science
Intelligent Autonomous Systems Group

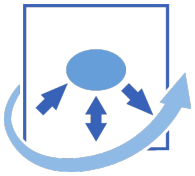
Diploma Thesis in Media Informatics

Physically Based Human Motion Tracking With Application In Manipulation Activities And Sport Analysis

---

Niels Kammerer





Technische Universität München

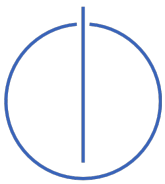
Department of Computer Science
Intelligent Autonomous Systems Group

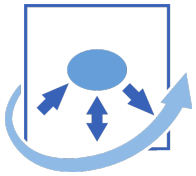
Diploma Thesis in Media Informatics

Physically Based Human Motion Tracking With Application in Manipulation Activities And Sport Analysis

Physik-basiertes Tracken menschlicher Bewegungen mit Anwendung in Alltagssituationen und Sportanalyse

Author:	Niels Kammerer
Supervisor:	Prof. Michael Beetz
Advisor:	David Weikersdorfer
Submission Date:	28. 2. 2012





I assure the single handed composition of this diploma thesis
only supported by declared resources.

Munich, February 28, 2012

.....

Abstract

This diploma thesis aims at improving a model based, visual tracking system, by means of introducing a physics engine as middle-ware. First, we give an overview of recent developments within the field of physics-based motion capturing, as well as providing a general basis of theoretical knowledge about model based visual tracking and particle filtering.
We discuss the way physics engines operate and how contemporary engines differ from one another in significant aspects. Being one of the most robust license-free engines available today, we introduce the Bullet Physics Library as our physics engine of choice.
Likewise, we outline the tracking framework at hand, giving insight to its limitations: Intersecting models, erratic, jerking motion artefacts and unrealistic slipping movements. The main point of the thesis is to counteract these limitations by implementing a physics based layer in the tracking framework, allowing us to check the scene for physical impossibilities like mesh collision or inexplicable force application. We use rigid-body dynamics to represent the models within the physics engine world and provide an interface to the physics layer that enables the tracker to recognize collisions between body parts of single or multiple models. In addition to collision detection, the physics interface allows to extract and visualize information of forces and torques acting on each of the models' body segments.
We tested force analysis in a custom environment with plausible results. Collision detection was tested within a custom built object manipulation task as well as genuine sports video recordings, bearing visibly positive results.

Zusammenfassung

Diese Diplomarbeit hat zum Ziel, ein visuelles Bewegungserkennungs-Framework durch das Integrieren einer Physik-Engine als Middle-Ware zu verbessern. Zuerst wollen wir einen Überblick über jüngste Forschungsprojekte auf dem Gebiet der physik-basierten Bewegungserkennung geben, gefolgt von einigen theoretischen Grundlagen zum Thema Modell-basierte visuelle Bewegungserkennung und der Partikelfilter-Methoden.
Wir diskutieren die Funktionsweise von Physik-Engines im Allgemeinen, wie auch die spezifischen Unterschiede zwischen gängigen Vertretern. Desweiteren stellen wir mit der Bullet Physics Library eine der robustesten, frei verfügbaren Physik-Engines vor, für die wir uns auch im Zuge dieser Arbeit entschieden haben.
Ebenso beschreiben wir das betroffene Bewegungserkennungs-Framework und beleuchten seine Einschränkungen: Modellüberschneidungen, ruckartige und unrealistische, rutschende Bewegungen.
Das Hauptaugenmerk dieser Arbeit wird auf die Bekämpfung dieser Einschränkungen mit Hilfe einer Physik-Ebene im Framework gelegt, welche es erlaubt, die Szene auf physikalische Fehler wie Kollisionen oder unerklärlich wirkende Kräfte zu überprüfen. Wir benutzen Festkörper-Physik um die Modelle in der Szene zu repräsentieren und bieten eine Schnittstelle zur Physik-Engine an, welche es ermöglicht Kollisionen zwischen Körperteilen einzelner oder mehrerer Modelle zu erkennen. Darüber hinaus können alle wirkenden Kräfte und Drehmomente berechnet und visualisiert werden.
Getestet wurde die Kraftanalyse in spezifischen Umgebungen mit plausiblen Resultaten. Kollisionserkennung wurde zum einen anhand von einem spezifisch generierten Objekt-Manipulationsfall, zum anderen anhand realer Sportaufnahmen getestet, was ebenfalls positive Resultate erzielte.

Contents

1 Introduction **1**
- 1.1 Overview . . . 1
- 1.2 Motion Capture Challenges . . . 3
- 1.3 Related Work . . . 4

2 Model Based Visual Tracking **7**
- 2.1 Video-Based Model Initialization . . . 7
- 2.2 Tracking and Pose Estimation . . . 9
- 2.3 Particle Filtering . . . 11

3 Physics Engines **15**
- 3.1 Functional Principle of Physics Engines . . . 15
- 3.2 Selecting A Physics Implementation . . . 16
- 3.3 Bullet Physics Library . . . 18

4 Initial Tracking Framework **21**
- 4.1 Tracker Components . . . 21
- 4.2 Tracker Limitations . . . 23

5 Tracking With Physics **25**
- 5.1 Physics Based Model Properties . . . 25
- 5.2 Model Setup in Blender . . . 34
- 5.3 Interface Implementation . . . 39
  - 5.3.1 Golem Physics: Collisions . . . 41
  - 5.3.2 Golem Physics: Forces . . . 44

6 Evaluation **49**
- 6.1 Force Calculation . . . 49
- 6.2 Collision Evaluation . . . 56

7 Conclusion **65**
- 7.1 Summary . . . 65
- 7.2 Limitations . . . 65
- 7.3 Future Research . . . 66

8 Appendix: Least Squares Fitting **67**

1 Introduction

We would like to begin the introduction to our work with a general view of the progresses in motion capturing technology and its state of the art applications in modern everyday life. Specific related research will be discussed immediately afterwards, in order to better place this work into scientific context.

1.1 Overview

The area of human motion tracking and analysis has enjoyed a constant increase of attention in the past three decades. Surveys have collected over 130 publications on the topic raging from the late eighties to the turn of 2000. As illustrated in Figure 1.1, the numbers of relevant research articles have increased in the following years to more than 350 related publications between 2000 and 2006 [31, 32].

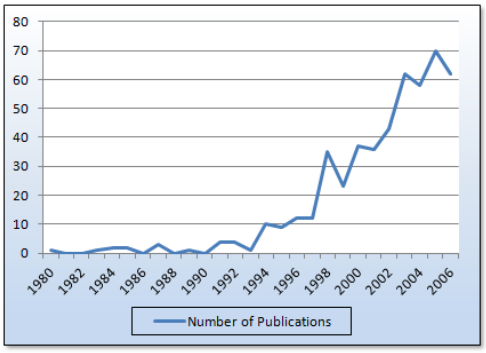


Figure 1.1: Number of publications on the topic of visual motion tracking and motion recognition since the year 1980 until 2006

Correctly tracking and analysing human motion, however, is a multi-problematic task. Moeslund et al. have legitimately broken the process down into four parts (see figure 1.2). The *model initialization* presents a match between the actor of the scene with a corresponding 3D-representation. For the actual *tracking task*, plenty of different methods have been developed to map the movement of the actor onto its virtual alter ego. Closely related to this step is the *pose estimation* for the model. Here, the model's underlying kinematic structure (typically an armature representing the skeletal body plan of the actor) is aligned to the appearance of the model. Eventually, in a step of indefinite complexity, the extracted motion trajectories can be analysed in a *recognition* step. This last step is the decisive reason for the increased, diversified research currently undertaken. A myriad of ideas and concepts have been sparked by the possibility of automatically extracting movement trajectories from our surroundings and more importantly, interpreting their semantic meaning.

Fields of interest for this sort of data have always been biological and medical areas. Different forms of locomotion were a discussed topic throughout recent history. In the late 19th to the early 20th century, Etienne Jules Marey was convinced that it was possible to create sophisticated, lifelike machinery by copying natural movement [30]. He built intricate machinery to observe and plot avian flight. Designing primitive forms of motion capturing technology, he might be considered a pioneer of the bionic field.
Having refined tracking technologies nowadays (see section 1.2), the focus shifted towards achieving economic improvements to leave expensive equipment behind. The option of

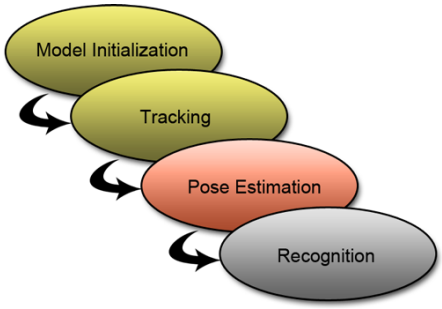


Figure 1.2: Four-step model of motion analysis

markerless and monocular tracking also opened up the field of computer aided sports analysis, where the athletes' feats are not only recorded but tracked to an extent that allows thorough analysis of technique and performance. The research in [40, 12] for instance aims to recognize and comprehend motion from low resolution video footage of track events and tennis matches.
Because recognition tasks for the mentioned examples is extremely error sensitive (especially in medical applications), this branch of motion capturing relies on good tracking and pose estimation. Best final results are still primarily achieved by human analysis. However, other options for motion tracking technology are more robust to recognition errors and can be classified as *surveillance* and *control* applications. Automated surveillance methods are applicable wherever human resources regularly fail due to lack of attention over time. Classic examples are crowd counting, flow and congestion control, where a large number of objects have to be tracked. More recognition reliant are systems that track and identify individual subjects and automatically reveal their actions or intentions. Areas of interest are security monitored parking lots, as well as shopping environments.

The most recognition robust application field is that of control applications. It involves the use of movement as a means of input for devices. The most distinctive example in recent years being motion control for home entertainment systems, Sony, Microsoft and Nintendo have introduced innovative, economic approaches to revolutionize human-machine-interaction. Commercially introducing the concept of motion-controlled games, Sony successfully paved the way for more sophisticated controlling devices with the release of EyeToy. Most notably, Microsoft's Kinect system is not only met by a substantial commercial success, it offers researches and hobby developers access to a cheap, stereoscopic video device. Selling more than 8 million units in just two months after public release, the Kinect was credited to be the "Fastest Selling Consumer Electronics Device" of its time [36], leaving any doubts about the concept's usability behind. Consequently, motion control solutions are being devised for office and home environments. They are considered especially useful for business conferencing and ubiquitous computing. Today, one of the most anticipated commercial releases of this kind is Apple's recently announced TV system. It is supposed to integrate motion as well as voice based interfaces and allow for cloud interaction between products of the family. Microsoft, too, has plans to provide Kinect for Television sets. Implementing this niche technology into mainstream devices is a first step towards present ubiquitous computing.
However, branches of visual motion tracking emerged that cannot be classified as one of the three mentioned categories. The automobile industry, for instance, is developing systems that combine aspects of different visual tracking and analysis techniques into security and convenience solutions. Examples being pedestrian tracking, lane following, line-of-sight based dashboard control and sleeping detection.

In order to provide a basis for all of these promising concepts, underlying structures and methods need to be reliable. As a very young field of information technology, motion capturing is still experimenting with different approaches to achieve best results. Some prominent examples for established tracking and pose estimation procedures will be discussed in the following sections and concluded with an overview of persistent limitations. Particularly for this work, we try to improve pose estimation by introducing a physics dynamics engine as middle-ware. The provided collision and forward dynamics methods help to decrease the rate of failure in the current tracking system and allow for a wider range of features and possibilities like pose-based force analysis. A general introduction to physics engines is given in section 3, followed by the specifics for Bullet Physics Library, the engine we chose to use. The tracker system we try to improve is outlined next and followed by our implementations of the physics based model. We evaluate our collision and force analysis framework after that and conclude the thesis with an overview of our accomplishments, discussion of its limitations and suggestions for possible future research on the topic.

1.2 Motion Capture Challenges

According to [23], popular motion capturing equipment can be categorized into two main branches. The first approach works with visual input of the person's movement and is further divided into active and passive systems. Passive systems use light-reflecting markers applied to the tracked subject's body, whereas active systems' markers are light emitters. A set of cameras circumfering the subject is then able to make out each marked segment and spatial trajectories can be reconstructed. Multiple cameras are of course needed to prevent occlusion.

A different method is to use electromagnetic sensors to capture the body segments' pose in space. Consisting of three perpendicular wire coils, the sensors are converted to electromagnets by applying low current. Measuring the induced current in the other coils allows to determine the exact position and orientation of the sensor relative to the transmitter. A detailed description of this sort of tracking technology is found in [39] and [35].
Unfortunately, both categories suffer from a number of disadvantages when it comes to whole-body tracking. The mentioned tracking procedures depend on highly orchestrated laboratory surroundings to be applicable. First and foremost, subjects have to wear sensors or markers to be tracked. Although they are usually worked into a body suit, these suits not only hinder the wearer's mobility, they impede the use of these tracking systems in a natural environment. For this task, a tracking system should be able to follow persons wearing any sort of clothing, and in all types of surroundings.

During tests with commercial tracking equipment, Richards et al. [23] have confirmed that systems relying on electromagnetic sensors generally perform fast and accurate. On the other hand, they are extremely susceptible to interferences in the environment such as caused by any metallic objects. Additionally, tracking accuracy is highly dependent on the distance between transmitter and sensors. On the other hand, magnetic tracking is not constricted to limited field of view and does not suffer from occlusion. This advantage led to electromagnetic tracking devices becoming a popular choice for adaptations in modern medical and surgery assistance equipment. Used in controlled, small-scale settings, the technology's shortcomings are very less pronounced. Most notably, it allows medical staff to follow the location and orientation of instruments during operations. Among various endoscopic techniques improved by this development, other beneficial uses have been proposed. In [17], the idea of tracking the surgeon's hands is presented in an effort to assess his technical performance. A value is automatically processed, which estimates the

physician's manual dexterity. Statistical analysis has proven this method of estimation to provide objective results that hold up with popular manual assessment techniques ([16]).

A topic that received much more public attention while pushing the development of tracking equipment in the last decade, was that of motion capture animation (also known as *MoCap*). Often used for the purpose of entertainment or advertisement, one or more actors' whole bodies need to be tracked while performing specific tasks or enactments. Both electromagnetic as well as optical tracking technology find use in this field. Optical systems have been known to be slightly more accurate than magnetic ones at the price of longer processing times ([18]). Many directors value the option to see the tracking results in real time, which for some time was only possible with electromagnetic tracking, but is now a standard feature among commercial systems ([9]). A long standing advantage of visually based tracking in particular, was the lack of trailing cables restricting actor movements. With the emergence of wireless magnetic tracking gear, this is no longer an issue. The range and equipment restrictions albeit still are. Therefore, even less obtrusive systems have been in development for the last couple of years.

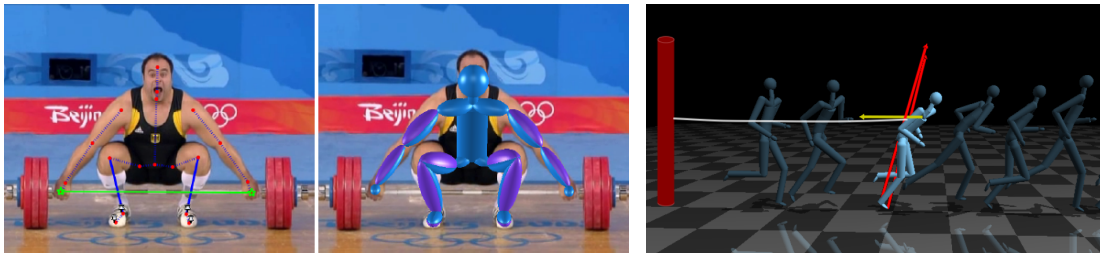
In order to allow motion tracking regardless of setting and preparations, researchers have been trying to overcome the following obstacles:
1. Necessity of carefully laid out laboratory setup. Modern motion capturing works well in studios specifically designed and equipped for the task. The scope of tracking tasks is therefore limited.
2. Actors' dependency on the system. As long as sensors or markers have to be applied to the actor in question, tracking tasks will have to be precisely planned and executed. Automatic tracking of people or objects, as intended by security or surveillance applications, needs independence of wearable gear.
3. System sensitivity to interferences. To improve the overall quality of motion tracking, improvements to robustness are constantly investigated. Interferences can be caused by magnetic fields, visual occlusion, moving backgrounds or similar events, depending on the technology used and the scope and surrounding for the tracking task.
4. Very high cost. The prices for complete optical or magnetic motion capturing equipment reach up to several thousand dollars.

1.3 Related Work

The research of unobtrusive visual motion tracking is a very active field at the moment, trying to address points two and four of the mentioned limitations. Markerless systems working only with visual camera input are among the most cost-efficient approaches and eagerly investigated. It is also clear that contributors are interested in creating robust tracking frameworks that are free of assumptions. In recent years, the focus has shifted to incorporating additional levels of information into tracking systems, expanding upon pure spatial data of former techniques. The roles of machine learning, dynamic simulations and particle filtering refinement is continuing to grow and improve the possibility to apply information to new sets of data without the need of manual preprocessing.

Visual tracking often utilizes statistical motion models to predict human movement. These models consist of functions describing the motion and the probability distribution of their parameters. They allow to generate or track specified movements, however suffering from clearly visible artefacts like unrealistic jerking movements or foot slipping (see section 4.2). Wei et al. have investigated the advantages of introducing physics based dynamics to motion models in [42]. The use of physically based frameworks allows for much smoother, more realistic looking movement to be extracted. Allowing reconstruction of Newtonian forces and torques, balanced movement as well as interaction with the environment can be

simulated. Working on monocular (i.e. single-view) video sequences, the modelling framework requires minimal user interaction. In a semi-automatic modelling step, the source video is annotated by the user, specifying joint locations on key frames. The 2D constraints used help the system to avoid common failures. Particularly contact constraints where feet touch the ground or distance constraints where a weightlifter clutches a handle with both hands (displayed as a green line in Figure 1.3), ensure that feet don't slip through the ground and hands maintain a certain distance from one another, as would realistically be expected.
Tracking then occurs by automatic 3D-keyframe interpolation. The model is not only described by its joints' positions, but also by joint velocity and acceleration, resulting in a Newtonian dynamics equation including inertia and gravity information. Due to the unavoidable restrictions provided by monocular methods (heavy occlusions, foreshortening of body-parts etc.), some manual adjustments may be needed to optimize the result. For this reason, refinement tools have been included, which allow the user to drag 3D joints to appropriate positions on an image layover - a very laudable feature from a usability standpoint. Wei et al. compared their results with those of a state-of-the-art commercial motion capturing system and found it to be almost identical in accuracy, all the while being much cheaper and nonrestrictive.



(a) **(b)**

Figure 1.3: (a) Manual setting of joint and physics constraints as described in [42]. (b) Generated running motion under environmental influence. A rubber band is holding the character back, which in response leans into the run to counteract the resistance realistically [43].

In [43], the authors have further refined the combination of statistical and physical motion models. The problem with physics based motion models is that a large number of poses may in fact be physically correct, yet simply look unnatural. Former research concentrated on filtering out the natural poses by applying a criterion called *minimal principle*, which assumes that a person would always move in a way that requires the least muscular effort. Wei et al. on the other hand extract the correct natural pose with the help of statistical priors. Therefore the idea is not only to introduce physics, but to take advantage of both statistical and physical motion models' strengths, resulting in natural looking synthesized motion that reacts to environmental parameters in a believable, realistic manner.
Like before, the subject is represented by a set of rigid bodies, which are in turn linked by joints specific to human anatomy and defined by relative position and orientation, velocity and acceleration. Furthermore, contact detection mechanics are applied that provide friction values and contact points. Checking that these are within reasonable constraints, the body can maintain a realistic balance. Complex muscle activity and therefore body movement can be modelled as force fields, which can then be used as a prediction method. Force field priors can be extracted from kinematic poses, so long as the joint forces can be generalized. This, however can not be done directly with current motion capture technology. A sampling method is used to estimate the velocity and acceleration from current,

previous and following frames, minimizing deviations for forces and applying aforementioned contact and friction constraints. A Gaussian process model is ultimately trained to learn the force field function from training poses. This function is able to predict generalized forces from joint velocities and accelerations.
Several experiments were conducted, comparing the synthesized motions with ground-truth sequences (i.e. unaltered movement input). The results are downright impressive, with the generated movements almost indistinguishable from ground-truth data. The inclusion of a physical model allows for correct responses to environmental forces like a rubber band attached to the body of a runner. The character begins to lean in appropriately, reacting to the elastic force resisting his forward movement (see Figure 1.3). Likely, classic statistical motion priors allow to synthesize different walking styles that are natural to us, yet have so far not been possible to generate due to the fact that they do not conform the *minimal principle*. A good example is walking stealthily on tip-toe, or throwing up legs like a marching soldier.

Another field of research that greatly benefits from physical motion composition is the mapping of natural movements onto (at least partially) humanoid robots. Albrecht et al. [6] have investigated a way to create motion models for reaching motions that can be mapped to the iCup, a highly articulated humanoid robot. Their goal was to find a novel cost function for imitation learning, using dynamics based optimization. Recorded trajectories are clustered and separated by motion type - reaching for a cupboard, a drawer or on top of a waist-high surface. Each cluster can be subjected to an optimization step, assuming the previously mentioned *minimal principal*, as well as other cost function hypotheses like *minimum jerk* and *minimum torque*. Each hypothesis is limited in its ability to describe movements, which makes it important to weight cost functions according to how appropriate they are for certain movement types. Calculating this weight vector was the main goal. Arm dynamics were modelled using two joints, one for the elbow and the other one for the shoulder, including all relevant muscle pairs. With this dynamic model, physically based constraints were established. After solving all cost minimization attributes, a reaching motion was successfully generated and executed by the robot. Figure 1.4 shows iCup imitating a human reaching for kitchen utensils.

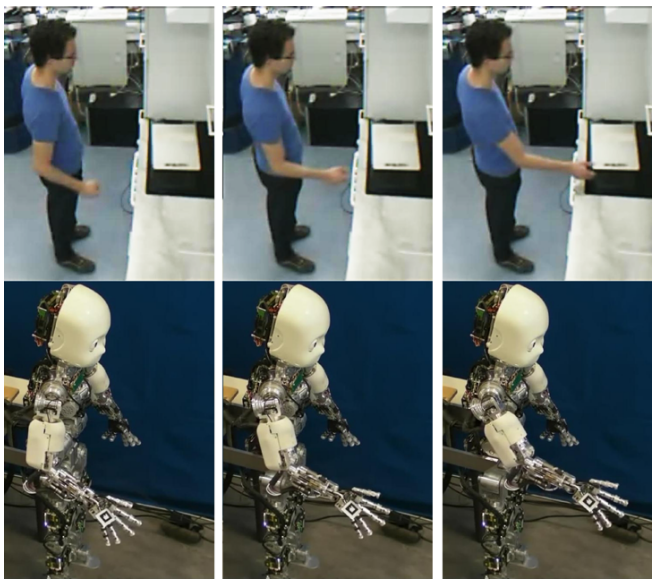


Figure 1.4: Humanoid robot iCup reaching for kitchen utensils. The Movement was generated using imitation learning techniques as presented in [6].

2 Model Based Visual Tracking

Including a section dedicated to the basics of model based visual tracking, we aim to provide basic explanation of fundamental techniques for visual tracking, as well as presenting important developments in the field, even though they may not be closely related to our own work. Poppe [37] provides an overview of some achievements in model initialization for tracking and pose estimation techniques. He also emphasizes the impact of combining different approaches in order to create more fleshed-out hybrid systems.
The move away from laboratory settings into open environments presents a set of challenges. New forms of interferences are introduced with varying lighting, shadows and moving backgrounds (for instance foliage or passants). In order to acquire movement trajectories, the subject first has to be identified in the provided video. Its body segments' degrees of freedom have to be identified and its pose estimated.

2.1 Video-Based Model Initialization

Usually, animated 3D-models consist of two entities, a skeleton that implies the character's kinematic structure, and a mesh, which presents its actual appearance. The structural aspect also contains constraints for joint movement and rotation, and is usually predefined to suit typical human features. An example of an ergonomics based generic human model is RAMSIS, which contains an inner structure with 63 degrees of freedom [7]. The human model we use is based on RAMSIS, and further outlined in section5.2.
The actual shape of the model is either constructed as a single surface mesh, which is deformed by the movement of the underlying structure, or consists of volumetric rigid body for each segment of the skeleton. These segments are usually 3D primitives such as cylinders and spheres, which can be described by just a few key attributes.
Because of the large variation among human physiques and shapes, a generic model is almost impossible to fit every potential human subject. If the model used does not align well with the tracked subject's shape, the quality of the pose estimation may suffer. It is therefore required to perform a model initialization, i.e. to make adjustments to the model, so that it best fits the subject in question. This is problematic, since it usually involves a lot of undesired manual preprocessing, and several methods of automatic model initialization have been developed.

In an attempt to allow free viewpoint rendering for athletes, Carranza et al. [10] initialize their model with the actor taking on a specific initialization pose. The pose is conceived for minimal occlusion and with slightly bent arms and knees, to facilitate the detection of elbow and knee locations, which are relevant for correct scaling of the model. Silhouettes are extracted for all available camera views and a series of iterations then approximates the model to the silhouette by computing the scaling and pose parameter, as well as non-linear scaling. At the end, the model's structure corresponds to the actor's skeleton and the mesh aligns well with the extracted silhouettes in all body parts, creating a digital likeness of the actor that can be used for tracking.

Since this method hinges on the subject assuming a specific pose for initialization, boundaries are again met. With the introduction of statistical priors, Grauman et al. [26, 25] developed a system that, once trained, is able to infer both shape and structure of a human model from video data. They use multiple cameras in order to reconstruct the subject's shape in a visual hull procedure. The shape can be determined by just the overlapping silhouettes from each view, whereby a high sensitivity to segmentation errors is observed. Much more robust results were achieved with the implementation of Bayesian inference, constructing a prior with PPCA techniques. The inference of related structure parameters then relies on training data, basically depending on the shapes recognized, without a

segmentation of the shape being necessary. See Figure 2.1 for a schematic overview.

For training, 20.000 human models were generated with randomly varying shapes and structural properties, which were then adjusted to assume a walking pose at a random point of time and facing a random direction. The models were rendered from several viewpoints, roughly at the same angles that are to be expected from the real camera setup eventually. The figure silhouettes are extracted from these images, alongside a fix number of structural reference points according to the position of particular joints. Each silhouette can be represented by a vector of contour-points as described in [25]. For multi-view data, the whole data set can be seen as concatenation of these vectors, with the tagged joint positions appended as well. The method was tested for a pedestrian walk cycle movement and both shape and structure could accurately be reconstructed, even with missing input views or high segmentation values, thanks to the inclusion of class-specific priors.

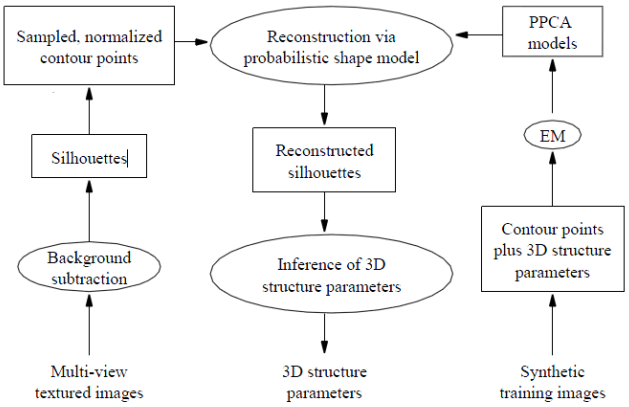


Figure 2.1: Steps for 3D shape reconstruction and structure inference [26].

A solution for visual hull reconstruction over time is proposed by Cheung et al. in [13]. They developed a procedure that uses multi view video sequences to accurately acquire the six degrees of freedom of a moving rigid item over several time steps. Using a ray based model, bounding edges are revealed that project from each camera to the edge of the extracted silhouette. Given the color information from the camera input, a stereo algorithm can be used to extract both the position and color of the touching points between bounding edge and subject, called Colored Surface Points. By projecting these points back into the images, the motion of the rigid object can be determined and compensated for. This allows the application of visual hull reconstruction algorithms like that mentioned above, in order to acquire an accurate shape model for the subject.
Because the shape reconstruction of an articulated object like the human body is a much more complex task, the procedure needs to be expanded to allow the extraction of humanoid models. As we already mentioned, such a model can be broken down into a constrained set of rigid objects, corresponding to human body parts like arms, legs, head and torso. This of course gives rise to the problem of correctly segmenting the silhouettes and thereafter applying the described method of temporal shape reconstruction. The authors propose an iterative algorithm to solve this problem. To start with, the Colored Surface Points (CSPs) are segmented into N Groups, where N is the number of expected segments. Afterwards, the iterative phase proceeds as follows. 1. Detect the motion for all segments. 2. Re-assign CSPs to the most suitable group. 3. Determine convergence of segmentation or conclude after maximum number of iterations.

Hereby step 2 is the pivotal calculation. To assign a CSP to one of the groups, the error

values for each possible allocation are calculated and compared. The CSP is then assigned to the group showing the least error. To improve robustness, two rules are implemented as well: one for spatial coherency, which operates in each frame and assures that no outliers are included in the wrong group. And a rule of temporal consistency, which assures that CSPs do not suddenly switch groups in between frames. With the separate motions for all segments determined, the joints can be estimated using a least square approximation to finish the model initialization.
Cheung et al. also included a successful test to track a human body in a motion capture environment, using the model generated by their procedure. Their work of temporal shape reconstruction begins to blur the boundary of the taxonomy between initialization and tracking.

2.2 Tracking and Pose Estimation

Because the tracking and pose estimation steps are related, some taxonomies prefer to combine them to a single *tracking* step, as for example in [5]. Simplified, this step is responsible for finding the best fitting model pose parameters for the observation, i.e. the video images. Pose estimation can occur either top-down or in a bottom-up fashion. Top-down approaches project a 3D model into the input image to find the pose that aligns best with it. Because of the large pose space provided by a human model with no less than 20 DOFs (in general, more are used for a fully articulated body), some methods like [24] include the manual setup of a starting pose. This first pose is aligned by hand in order to allow the use of gradient descent algorithms in subsequent video frames. Depending on whether the match is calculated due to a likelihood function or a cost function, local maxima or minima are searched, which eventually represent the closest fit between model and observation. Nevertheless, the dependency of a manual initialization step is a major drawback.

Another difficulty is the integrity of the kinematic hierarchy. Fundamentally, any movement in lower kinematic nodes such as the pelvis or torso of a model, will entail movement in it's upper nodes, i.e. the head and appendages. Vice versa, an error in the pose parameters for e.g. the head may race through the kinematic chain, propagating the error to other segments of the body. In [20], a countermeasure to error propagation throughout the kinematic chain is given with the introduction of constraints between connected body segments. A common method in computer animation, this restrains body-parts to realistic, ergonomic movement by limiting their maximum rotation and movement. A valid measure is from then on not only computed by minimizing the error of projection, but must also comply with the ergonomic constraints of the human body. This propagating process was able to track a human in real-time at 10 frames per second, using a setup of three cameras.
Bottom-up estimation on the other hand tries to make out specific body parts in the image plane and build a complete model from these. Prerequisites of bottom-up methods are templates - or detectors, for every individual body part, as well as fitting rules for spatial and temporal constraints to improve robustness and counteract interferences. However, a manual initialization step is not needed. This advantage led to bottom-up techniques being used as automatic initialization for top-down estimation. Navaram et al. [34] take advantage of the human kinematic structure to facilitate template based search. By finding an essential hierarchical node of the kinematic structure first (like the head or torso), and then following restrictions of body part proximity, the amount of templates needed to be searched is significantly decreased. This technique is therefore a type of *search space decomposition*, similar to that used by Gavrila et al. in [24].

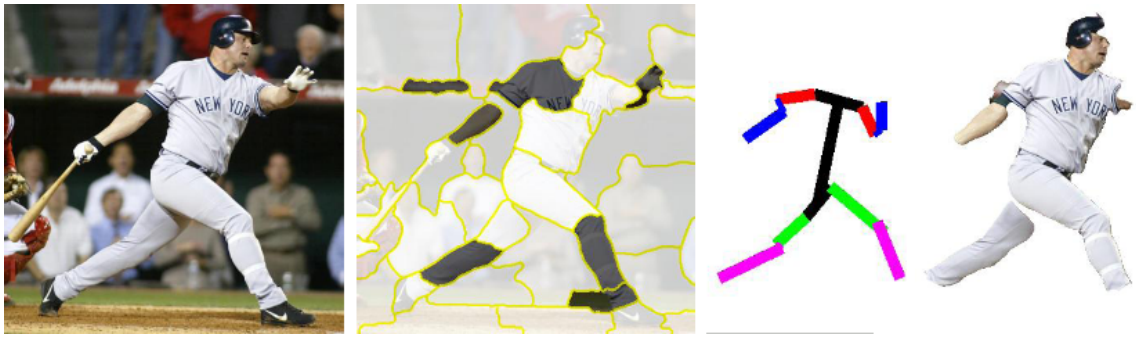


Figure 2.2: Exemplary stages for Mori's pose estimation technique. Left to right: Input image, super pixel map, extracted body structure, segmentation mask. [33]

For the task of finding specific body parts in images, a multitude of different approaches have been conceived. For instance, Mori et al. rely on sophisticated image processing algorithms in [33], breaking input images into *superpixel* segments and regions to identify *salient* features like half-limbs, head and torso (see Figure 2.2 for an example image). These are extracted by searching for a series of cues, which are typical to mentioned body parts. Both contour and shape are examined to determine likeness between a given segment and limbs. Another cue is the shading of the segment. Since limbs basically consist of a cylindrical shape, they often show a pronounced gradient of value in images. To further differentiate subject and background, the focus cue gives information on the region of interest in the image. Backgrounds are often out of focus and blurry, resulting in lower overall frequencies. Searching for torsos is slightly more challenging, since they are usually spread across multiple (adjoining) segments and do not provide a typical shading quality. The overall shape of the torso is extracted using a bounding box estimation in the shape cue, spanning multiple segments. Orientation of the torso is then recognized with the search for an appropriate head.
Once the salient segments have been identified, the whole body needs to be reassembled. The very large number of possible body configurations are limited by a set of constraints applied to each one. Among the remaining possible body configurations, scores are calculated with the help of the aforementioned cue values.

On a similar track, [22] describe the model as pictorial structure - an undirected graph, representing a series of parts with appropriate bilateral connections. A learning model is introduced based on a maximum likelihood function, which is solved to obtain an optimal configuration of body-parts.
Sigal et al. [41] expand this concept, allowing to infer the 3D location of the body-parts, as opposed to the 2D configurations discussed so far. Similar to [22], an undirected graph is constructed with each node being a parametrized primitive representing a body-part and soft constraints between these represented by edges connecting the respective nodes (see Figure 2.3. For every body-part, an individual maximum likelihood function is conceived and pose estimation then works by using belief propagation.

This would technically be the point to immerse in the last step of motion analysis - the recognition step. A fairly up-to-date overview of research on this task is provided by Poppe in [38]. However, the recognition task is beyond the scope of this thesis, since it is not relevant for the work presented. Instead, we would like to give a technical introduction into the prominent motion tracking method of *particle filtering*.

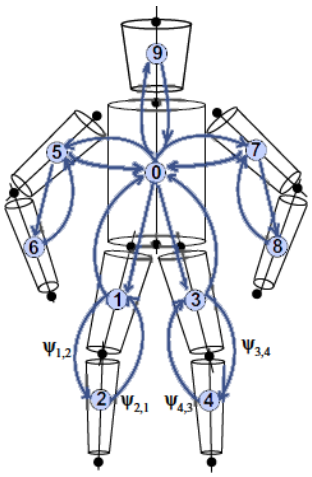


Figure 2.3: The graphical model for a human body. Nodes (light blue) represent body-parts and edges (dark blue) represent articulated joints between them.

2.3 Particle Filtering

After the introduction of pose estimation approaches, the next issue at hand is to recognize the pose's change over time (i.e. over a series of frames) and ensure what Poppe calls *temporal coherence* [37]. With the transition from Video to 3D space, noise is inevitably introduced to the system. In the old days, Kalman filters were used to minimize offsets in the model configurations between frames. Given the noisy input of a dynamic system, and provided a mathematical model of said system, a Kalman filter removes the error caused by the noise by predicting the future state of the system and consequently correcting the output. This cycle of prediction and correction allows the use of Kalman filters on continuous input streams. Its major drawback on the other hand, is that it only works within linear dynamic systems, and it assumes a Gaussian distribution in the error terms. Due to human body motion being non-linear in nature, utilization of Kalman filtering in vision based tracking tasks suffered from accumulated errors. A successor for the Kalman filter has been found in the *Sequential Monte Carlo Method* or *particle filter*.

Particle filtering (also known as *condensation* is a sample-based method with very much the same purpose as the Kalman filter - eliminating errors in the measurement of dynamic systems. A significant improvement over the Kalman filter are the facts that no assumptions of error distribution have to be made and system dynamics may be non-linear. *Particles* are defined as weighted samples from the state space. Hence for the estimation of a rigid body's position in 3D space, a single particle would include a set of possible variables for all six DOFs, as well as a weight value. In the estimation step, a *prior belief* is used to take a sample from all possible particles. This prior belief can be based on saved data or it can simply be a uniform distribution. Following that, the importance weights are calculated by updating with observations (see Figure 2.4).
Given the particles' importance, it is possible to reach an improved best guess by picking the highest rated particle, or continue the cycle by resampling all particles. This is done by selecting particles according to their importance, without altering the total number of samples. Doing so increases the density of particles in high probability regions, while discarding less likely particles.

Unfortunately, the condensation algorithm is not without its drawbacks, either. In visual tracking tasks, the model needs to be rendered for each particle, which is a considerable

computational effort. Also problematic is the possible clustering of particles in local maxima on multi-modal distributions. Much research has been conducted to improve particle filtering, in part specifically for tracking.

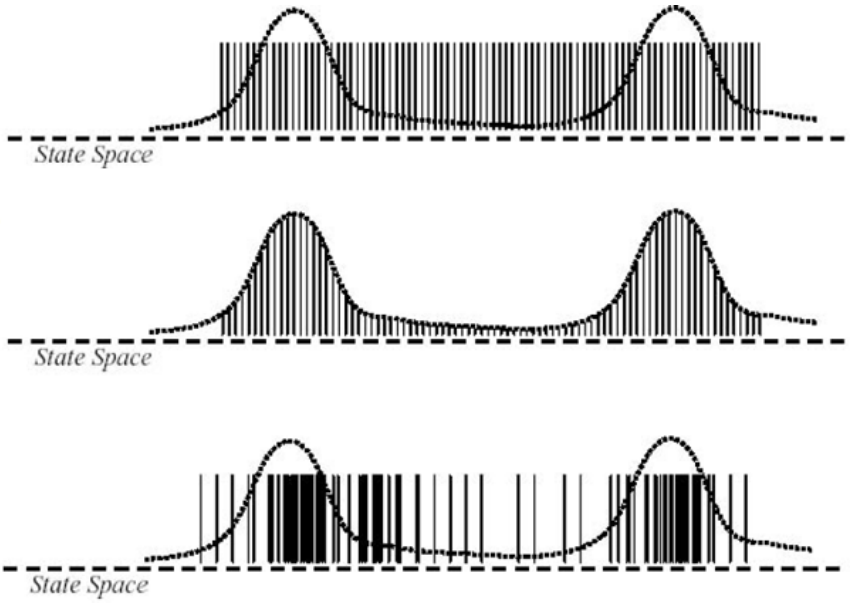


Figure 2.4: *Top:* initial particle distribution. *Center:* particles are weighted according to weight function (dotted line). *Bottom:* new particle distribution shows the clustering of particles around high probability regions. Graphic from [28].

One possibility of reducing the initial search space, is the introduction of priors. Although the range of movement a human body can perform is staggering, most actions can be generalized to specific motions. A classic example is the human walk cycle. Even though each person has a unique way of walking, the motion involved is always essentially the same. These types of movement can be reduced to generic motion models, which are learned from training data. Disregarding the effort of training the system, using motion priors facilitates tracking for given motion types on one hand, but effectively limits the variety of movements that can be recognized on the other. It is nevertheless an excellent method of improving tracking capabilities for specific tasks.
As an example, Hamer et al. have achieved to create motion priors from very limited training data of human hands grasping scene objects. The quality of a prior is generally assumed to be dependent on the amount of available training data, which is why the hand pose estimation method presented in [27] is so intriguing. Reducing the amount of necessary training data to a single instance, successful tracking is achieved. However, the authors do take into account an improvement in accuracy with the inclusion of additional training samples.
Construction of the prior works by extracting position and orientation (six DOF) for each hand segment, as well as their contact points on the object. The prior is specific to that object class, being modeled as a spatial distribution. Warping algorithms allow to map the prior to any unique hand grasping any instance of the given object class. Unfortunately, heavy occlusions may compel the need to label some pose parameters for the training data manually. During the tracking itself, no manual interaction is needed. This allows for on-the-fly synthesis of grasping motions for unknown models, which is especially useful for animation and robotics tasks.

Considering the problem of particle clustering in local maxima, Deutscher and Reid [19]

have proposed a variation of regular particle filtering, based on simulated annealing. Annealing basically introduces random factors into a search space, so that solutions may be found anywhere in the distribution. The consequence is the finding of a global best solution, instead of spiraling into local maxima. The idea and the term are borrowed from the metallurgical technique of heating metal and letting it cool, allowing the crystallization to form structures of minimal energy [4]. Using this method - labeled *annealed particle filtering*, allows for significantly more accurate tracking results than default condensation algorithms.
Tracking performance can be further improved by decomposing the pose space as proposed by MacCormick et al. in [29]. Since the human body is generally modelled as a hierarchical structure, it is possible to use this *partitioned sampling* approach with whole body tracking. The basic idea is to reduce computation cost by first estimating a pose for a hierarchical root, for instance the torso, before starting estimation of the depending hierarchies of arms, legs and head. Bandouch et al. [7] call this technique *hierarchical particle filtering*, and have conducted successful tracking experiments with highly articulated models containing 51 DOFs.

3 Physics Engines

The described research tries to emulate realistic dynamics and constraints in the form of models and formulas based on real world physics. In general, researchers have selected specific traits they want to incorporate into their models, for instance the use of joint torques and velocities to create motion models. Complex equations were constructed for specific purposes only. This thesis, on the other hand, aims to incorporate existing physics engines as middle-ware to a sophisticated model based visual tracking system. The following section is divided into an introduction to the workings of a physics engine, followed by a detailed comparison of common physics libraries and a reference of our chosen candidate, the *Bullet Physics Library*.

3.1 Functional Principle of Physics Engines

A physics engine can be defined as a piece of software that allows calculations for realistic movement of dynamic objects. Its main objective thereby is to solve forward dynamics. This means that the engine calculates the movement of a system driven by applied forces [8]. The relevant groups of dynamics being fluid dynamics, soft-body dynamics and rigid-body dynamics. Fluid dynamics deal with particle flow and behaviour to simulate bodies of liquid substance, whereas soft- and rigid-body dynamics handle solid forms. Soft-body dynamics further differentiates itself from rigid-body dynamics by including complex mesh deformations in respect to applied forces and collisions. Although in reality almost all materials are subject to deformations (including flesh and skin), most simulations settle for rigid bodies to model subjects. This is because soft-body calculations are of considerably higher computational cost, which until recently prohibited their use in real-time applications. Also, soft bodies introduce additional parameters for the dynamics engine that, when not configured properly, lead to unrealistic behaviour. Since we are modelling all dynamic subjects as rigid bodies in our simulation, this section will focus on rigid-body dynamics only.

Rigid-body dynamics predominantly take place in the physical system of classical mechanics (Newtonian mechanics). This field describes real world objects as particles, points of negligible size, which are primarily defined by their mass, their position in space, and the forces applied to them. Knowing the forces acting on an object is sufficient information to calculate the object's position according to Newton's second law:

$$\sum_{i=1}^{N} f_i = \frac{d\,(mv)}{dt} \tag{1}$$

Rigid-body dynamics expands this system by describing objects as rigid bodies of a defined shape. This entails the inclusion of other crucial attributes like center of mass, moment of inertia and orientation, because, unlike particles, rigid bodies occupy a certain amount of space. The inclusion of constraints is also considered part of rigid-body dynamics. Constraints basically define limitations of transformation for bodies. For instance, a rigid body's position might be constrained to a specific point in space, leaving it's rotational parameters to be described by other dynamics. A hierarchical chain is achieved by defining constraints that attach the position of one body to a point on the shape of a parent-body. Satisfying constraints during simulations is a mathematical problem that is addressed by implementing a *constraint solver*. One possible approach to do this is described below in section 3.3.

Probably the most used feature of physics engines is collision detection. The term includes

both the detection of intersecting (i.e. colliding) objects and the calculation of an appropriate collision response. Collisions occur in all forms of dynamics. However, rigid bodies are the most efficient to determine and resolve. Collision detection can either be performed discrete (*a posteriori*, i.e. after the collision) or continuous (*a priori*, i.e. before collision occurs). Discrete detection is the more efficient, yet inaccurate method to retrieve collision data. For each time step, the spaces of all bodies are checked for pairwise intersections. If an intersection is found, some form of reaction can be implemented to resolve it. Very problematic are fast moving bodies. If one object is moving towards another quickly, it happens that the resting body is skipped in one step, because it fits into the space that has been traversed in a single step by the travelling body. Continuous detection on the other hand uses many pieces of information in order to predict the trajectories of the bodies and calculate any possible collisions before they actually happen. This way a more accurate collision resolve can be achieved, because no actual interpenetration occurs. The downside of course is higher computational cost and an inseparability of dynamics and collision detection algorithms, due to the resulting dependencies for *a priori* estimations. In continuation is an introduction and comparison of different physics implementations, as well as a detailed round-up of the Bullet Physics Library, the engine used in our work.

3.2 Selecting A Physics Implementation

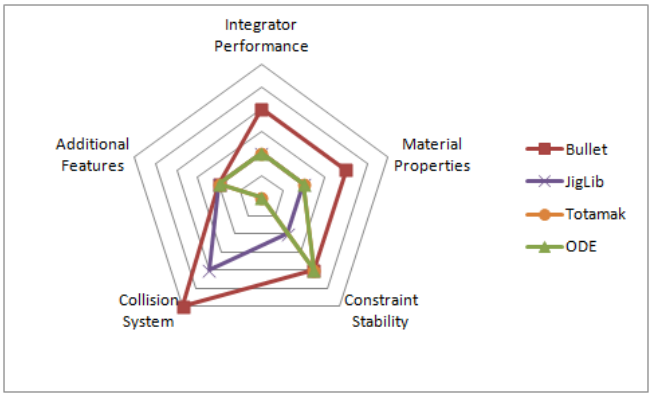
Among the first applications for physics engines were ballistic calculations for military artillery, which required very high-precision calculations. Simulation engineers today have demands for these high-precision engines in all sorts of industrial design, especially aerodynamics and automobile design. On the other hand, the increased use of physics engines in the entertainment sector has given rise to new, quickly processing physics engines. This development is in part also driven by the possibility to perform the needed calculations on graphic processor hardware.
Although the engines used for real-time physics simulation are less precise, they allow for interactive simulations, which are especially interesting for game developers. With increasing demand for these fast processing physics engines, a plethora of different libraries have been created during the last couple of years. Besides successful commercial licenses like Ageia PhysX, numerous open source solutions are available. All of which vary to some degree in their implementation and available features. See Figure 3.1 for a round-up of available physics engines, their implemented paradigm and license restrictions.

Physics Engine	Paradigm	License
AGEIA PhysX	Unknown	EULA
Bullet	Impulse/Constraint	Open, Zlib
ODE	Constraint	Open, LGPL/BSD
JigLib	Impulse	Open
Newton	Uknown	EULA
Tokamak	Impulse	Open, BSD
True Axis	Unknown	EULA

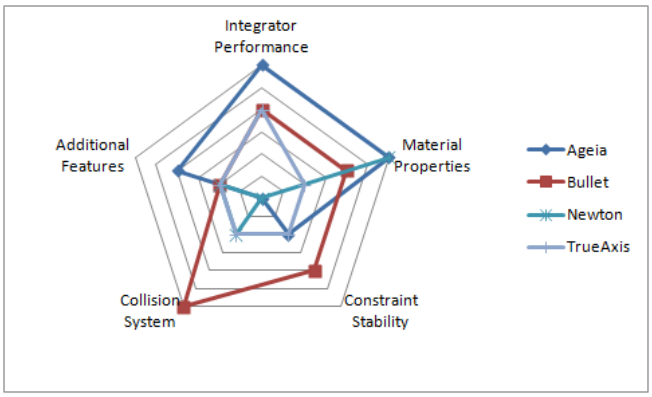
Figure 3.1: Overview of the most prominent real-time physics engines. The paradigm followed by closed-source engines is unknown, as opposed to the open source candidates. Engines enforcing an end user license agreement are not free to use.

Due to the high diversion of features and performance among the available engines, it is necessary to chose the engine that best suits the respective project. A comparing survey by Böing et al. [8] provides a reference for the strengths and weaknesses of the most

commonly used physics engines, both commercial and free. Bullet represents a relatively new contestant among physics engines. However, because of its open source nature and relatively good support, it has quickly become a favorite among consumers. [8] has tested performances for crucial elements shared by all engines. The relevant results are summed up in Figure 3.2.



(a)



(b)

Figure 3.2: Results for each of the major engines evaluated in [8]. Exact results are omitted for the sake of a general placement analysis. Engine performances were given a relative score estimate between 0 (poor) and 3 (superior). Bullet Physics Library is compared to other open source engines (a) and commercial engines (b).

There are several important factors determining the performance of a physics library. [8] have explicitly tested the following criteria: 1. *Integrator Performance* is the engine's accuracy when calculating the movement of dynamic bodies subject to forces. 2. *Material Properties* represent the engine's accuracy when determining the effects of friction and restitution on bodies of varying material. 3. *Constraint Stability* determines the lack of numerical errors during the constraint solving phase. This is reflects the realism and accuracy of dynamics bound by constraints. 4. The engine's *Collision System* is its ability to track and resolve collision between objects correctly.

Although unimportant for our own project, it should be mentioned that some engines are more fully featured than others. Especially the commercial AGEIA PhsX Engine stands out as a versatile engine, including features like fluid simulation, character controllers and numerous special constraints. This engine is notably also the only one to support dynamic triangle meshes.

The authors also performed *stacking tests* in order to make out the computational effort of stacking numerous objects on top of each other. The Tokomak engine fared best in this regard. However, this feature can be considered a niche criterion and is not included in the figure above .
The underlying *simulator paradigm* is also influential in the accuracy of different simulations (see figure 3.1). Since the implementation of closed engines cannot be inspected, it is not apparent which paradigm these engines use.
Results for the performed tests are very diversified, showing the complexity of determining an optimal physics engine for a certain project. While all investigated engines have their strong and weak points, the authors could verify that Bullet performed significantly well in most aspects, making a well-rounded impression. It was especially successful in solving rigid-body collisions. This was one of the main criteria for our research, which, alongside open source code and a quickly growing good reputation, led to Bullet being the first pick among the available engines.

3.3 Bullet Physics Library

We now want to provide an overview of Bullet's general structure, implementation details as well as core mechanics of the engine.

Overview Bullet is an open source physics engine started and maintained by Erwin Coumans. It is written in C++ and is currently on version 2.79. Its prominent features, according to [15], are integration in popular 3D-Editors like Blender and Maya, soft-body dynamics, a "fast and stable [..] constraint solver" and a carefully phrased promise of discrete and continuous collision detection between both convex and concave meshes. Concave meshes turn out to be only possible as static scene décor, however, which will be addressed later in this thesis. Also considered a major feature is the fact that Bullet is available free of charge, even for commercial use, under zlib license. This basically means that the use and alteration of licensed library has to be formally declared as such.

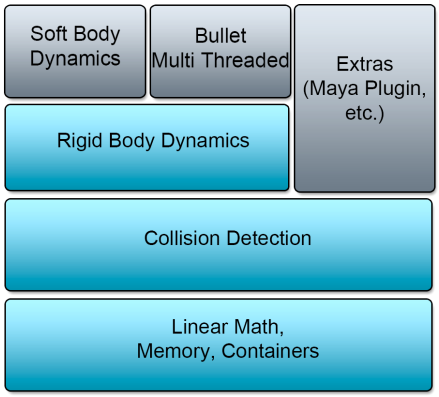


Figure 3.3: Bullet Physics Library's modular component structure [15] . If not necessary, components on top of the hierarchy can be left out of the simulation. Higher saturated components are the ones used in our project.

Bullet is composed of modular components, as illustrated in Figure 3.3. Each of the components may be isolated, bearing dependencies from top to bottom. Therefore, it is possible to use Bullet only for collision detection, without including dynamics behaviour. As it is in our case, we have no need for soft-body dynamics, and consequently use only

the libraries *LinearMath*, *BulletCollision* and *BulletDynamics*. Neither did we have need for any integration extras, since all our model data was structured in .tsv files (see 5.2 for more details).

Dynamic Steps Each simulation in Bullet is defined by a "world", in which objects may be placed. The default world is *btDiscreteDynamicsWorld*. Figure 3.4 gives an overview of a single simulated time step. Although the world is not directly responsible for all the tasks depicted, it controls the simulation flow by determining step sizes and initializing dynamic processing iterations when calling *stepSimulation()*. The pipeline is started at the left side, applying the gravitational force. This has a direct influence on the velocities of all objects (represented as a red arrow in the graphic). Gravitation in Bullet is set per body. Thus, each body may have a different force acting upon it in this step. This may be of interest for game developers, but does not occur in our simulation. Next, the position and orientation for each body are predicted to allow for continuous collision detection. In many 3D applications, position and rotation of a frame are defined as a 4x4 transformation matrix. Bullet uses the class *btTransform* to store the frame's origin as vector and its rotation as quaternion, which can conveniently be transformed to a 3x3 rotation matrix.

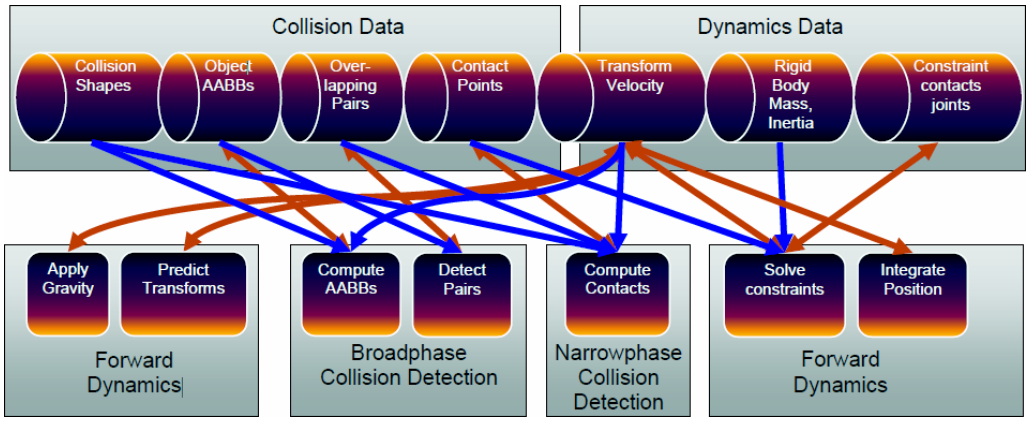


Figure 3.4: Bullet Pipeline as described in [15]. The elements in the upper row represent data structures, the lower row are processes that either depend upon data (blue arrow) or affect data (red arrow).

Like all physics engines, Bullet splits collision detection up into two steps. Broadphase collision detection first computes an axis-aligned bounding box (AABB, see Figure 3.5) for each rigid body, using the information provided by its collision shape (indicated by blue and red arrows in the pipeline, respectively). Note that Bullet separates the rigid body from its collision shape. This is done for performance reasons, since it allows the same collision shape information to be used for many instances of rigid bodies (e.g. a bowling simulation would only require three shapes, one for the ball, one for the lane and one for the pins). This means that a collision shape like *btCapsuleShape* holds only the geometric data relevant for collision detection. All the data specific to forward dynamics - mass, inertia and positioning, are stored in the *btRigidBody* object. Finding pairs of AABBs that intersect is an easy enough task for computational geometry algorithms, and can be performed with little computational effort.
So far, an inaccurate guess of intersecting bodies could be made. Real collisions are then predicted in the second step, also known as narrowphase collision detection. All calculated data (again indicated by blue arrows) is evaluated to determine whether or not each overlapping pair actually collides. For all pairs, an appropriate collision algorithm is chosen

depending on the type of collision shapes within the intersecting bodies, resulting in a set of contact points. These are stored in point manifolds (by default *btPersistentManifold*), providing easy access to critical information such as the point of greatest penetration depth.

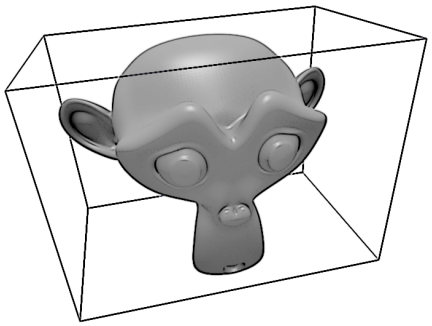


Figure 3.5: This rendering demonstrates a 3D model's AABB. Note that the box is aligned to the object's local coordinate frame, not the global space frame.

Contact points are passed to the constraint solver along with the bodies' dynamic properties. The problem a constraint solver needs to address, is to keep an object attached to its parent - e.g. an arm attached to the body. This is done by applying just the right amount of forces or impulse to the body in question. By default, Bullet uses a projected Gauss-Seidel algorithm for iterative constraint solving, also known as *Sequential Impulses*, because it sequentially applies impulses for all constraint that are not satisfied. A complete and detailed explanation of this solver approach can be found in [21, p. 136-162] or - focusing on game dynamics, in [11]. Eventually, the velocities and positions for all bodies can be calculated for the next time frame, where the pipeline is reiterated.

For reference purposes, Figure 3.6 gives a quick overview of Bullet Physics classes and their role in the dynamics processing.

Bullet Class	Function
btDynamicsWorld	Rigid body container; Dynamics processing pipeline
btTransform	Transformation Matrix; position and orientation of frame
btVector3	Vector used to describe frame origin (x, y, z)
btQuaternion	Quaternion used to describe frame rotation (w, x, y, z)
btRigidBody	Rigid body with physics properties mass, inertia, transform
btCollisionShape	Shape information. Can be used for many rigid bodies
btPersistentManifold	Collection of contact points for a given pair of bodies
btHingeConstraint	Constraint simulating hinge joint
btConetwistConstraint	Constraint simulating ball and socket joint (e.g. human shoulder)
btDefaultMotionState	Control interface for rigid body; Convenience asset

Figure 3.6: Some of the basic classes encountered in Bullet Physics simulations.

Being an open source project, Bullet documentation is relatively sparse besides a rough online quickstart wiki. The developers suggest to study the accompanying demos, instead. These provide simple examples of most relevant features in Bullet, set in an OpenGL user interface that allows for mouse and keyboard interaction with the simulation. We chose to use the Bullet ragdoll demo as a basis for our physics framework.

4 Initial Tracking Framework

The main goal of this thesis was to create a physics engine interface that provides physics simulation support for a condensation based motion tracking framework. Chapter 4.1 gives an introduction to the tracking system we wanted to improve. Chapter 4.2 addresses the limitations found in this system and briefly describes how we planned to remedy them.

4.1 Tracker Components

The tracking system we want to modify consists of three major components, each of which is implemented as a separate library and can be further partitioned into various subcomponents. Figure 4.1 presents a top-level view of the system composition. On this level, we have a pre-initialized *model data structure*, which is used to describe a *model component state space*, leading to a library of *optimization and tracking* algorithms applied to the data.

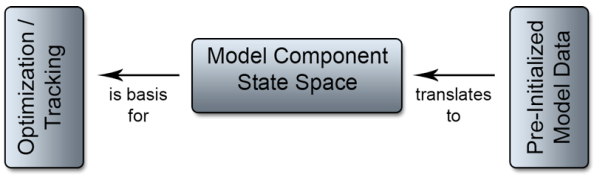


Figure 4.1: Top level view of our tracking system components. Each is is implemented in its own library, dependencies running from left to right. At a basic level, the pre-initialized model properties are stored. These help defining the tracker's state space. The middle component also includes various higher level tracking algorithms and configuration options. Optimization and the actual tracking tasks are performed by a final component.

We model objects for the tracking tasks in a *mesh and structure* form. This means that the object is assumed to have an inherent structure consisting of articulated bones. These bones form a hierarchy that can be exploited to store the position and orientation of each bone in a simple fashion, thus creating local poses. The mesh contains the geometry and a web of weights for each vertex. These weights define which parts of the geometry will move together with each underlying bone. Geometry can be stored and accessed in three ways: As vertices, as triangles and as triangle corners. Furthermore, the model's appearance may be defined by a color or texture map. At the end of the day, the model is usually a fully rigged, digital character ready for animation.

The model data library (see figure 4.2, fondly referred to as *Golem*, also includes several utility classes. Among others, a limited rendering feature is available, as well as functions to facilitate the input / output of data. All information necessary to reconstruct both hierarchy and mesh are stored and loaded from in a very simple form as tab-separated-value files (structure.tsv and mesh.tsv, respectively). Chapter 5.2 explains the details of exporting mesh and structure information from a 3D editor.

For tracking tasks, the relevant pieces of information are the models' poses and *components*. Components in our case can be all sorts of predefined constraints or options for parts of the model's structure. In particular, each articulated joint is defined by such a component. Possible types of component include hinge and ball-and-socket constraints (with one or three DOFs, respectively), but also for instance scaling. The type of each component is again read from an external components.tsv file, along with its respective

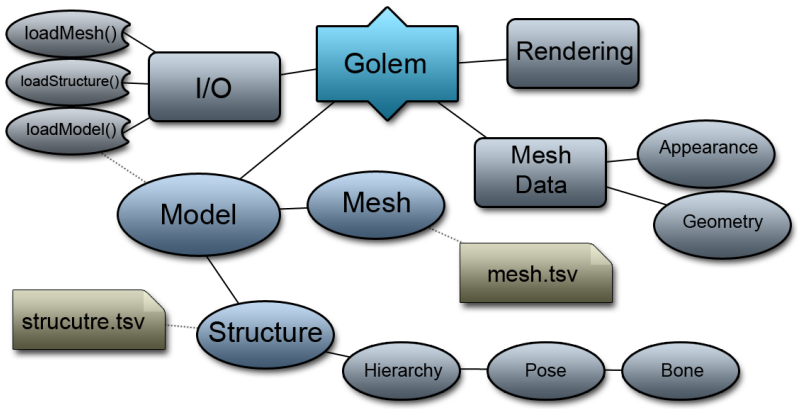


Figure 4.2: Overview of the model data library called *Golem*. Model properties are loaded from two separate files: mesh.tsv and structure.tsv, one defining the model's geometry and the other its bone hierarchy. The library also includes several classes for utility, input / output and rendering purposes.

parameters. Components and actual parts of the structure are then bound by a specific layout. A layout.tsv file contains the correct mappings used to fit a component to one or more body parts. A joint, for instance always requires two bones to be attached, and further specifies a relation like forward or inverse kinematic, or a movement copy.

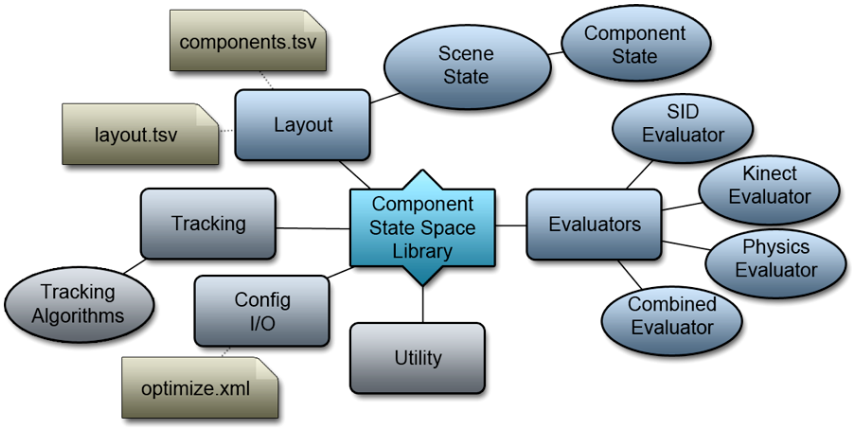


Figure 4.3: The library defining the component state space. It can be broken down to five main aspects: high level tracking algorithms, configuration, model component layout, utility tools and evaluators. The newly introduced Physics Evaluator is where the physics engine will be latched on (see section 5.3).

Model data is further processed into *component states*, which are in turn aggregated in *scene states*. These will be usable by the optimization component later on. But beforehand, a configuration file (optimize.xml) is loaded that specifies the model files to be loaded and features to be evaluated. The tracker is build so as to be extendible by custom evaluators. Figure 4.3 provides a reference diagram for our component state library. There are currently 4 evaluators available: A Kinect Evaluator for use with Microsoft's Kinect hardware. A State Evaluator that uses silhouettes as image descriptor to compute a score for each particle. The Physics Collision Evaluator developed for this thesis (see chapter 5.3.1) and a Combined Evaluator, which calculates a score from multiple evaluators, using the equation $s^* = (s_1 * .. * s_n)^{\frac{1}{n}}$. Among other tools, some tracking algorithms for

specific cases are also included in this library, as well as an access point to the graphical user interface.

The abstract tracking implementation can be found in the third component, though. This is were optimization and tracking according to given configurations take place. Optimization amounts to calculating scores for the particles using the chosen evaluators, resulting in a best state for the current frame. This basically means the model is posed in some fashion, referred to as an *hypothesis*, which is evaluated and given a score between zero (worst case) and one (best fit).
The whole sequence is thus computed frame by frame, resulting in a tracked 3D representation of the action. As is usually the case though, there are certain drawbacks in the system. Thus we want to outline the problems of the tracking system as it was, and how we think the implementation of a physics engine will mark an improvement.

4.2 Tracker Limitations

The outlined tracking framework presents good results as presented in [6], where the same system had been used. However, a considerable amount of pre- and postprocessing are required to obtain this level of accuracy.

To begin with, the initial model setup requires several handmade adjustments to joint limits and constraints, like we already explained in the previous chapter. These parameters are scattered throughout multiple separated files. Structure information could be exported from a 3D environment, then components had to be described for each joint, and ultimately a layout file had to match these two lists up. Although this might seem to be a superficial issue, it was a source of error nonetheless.

In order to reproduce realistic human ergonomics, it is necessary to quantify the joints' allowed degrees of freedom as precisely as possible. Having to handle multiple sources of information that are not specifically meant to be human readable was a burden. Moreover, the lack of a visual representation of the components made it all the more difficult to find the right setting for each joint. The layout description was thus prone to errors, meaning that individualization of models for different tracking tasks was cumbersome and unreliable, influencing the quality of the tracking result in a negative way. To facilitate the model initialization, we developed an extension for the open source 3D editor *Blender*, which allows mesh and hierarchy modelling as well as components setup within a single view. An introduction to the model setup with Blender is given in chaper 5.2.

In addition to the inconveniences during preprocessing steps, the tracking results themselves were foremost limited by the erratic motion artifacts described in chapter 4.2. Unrealistic and unlikely movements of single body parts can occur, like sudden, jerky glitches of limbs and head. Apart from being awkward, these artifacts cause the resulting trajectories to run untrue to the tracked motion and may as well ruin the tracking objective. For instance, motion analysis is hindered by the loss of critical information during these faulty sequences. Our plan was to exploit forward dynamics simulation to spot highly unlikely shifts of acting forces. By estimating the force required to perform a motion from one frame to the next, unexplainable, jerky motion should be recognized and avoided.

Another issue frequently encountered are intersecting meshes. This applies to both self-collision within a mesh and inter-penetration of different objects. Being based primarily on the silhouette image descriptor evaluator, the optimization sometimes yields dubious results of a human impaling his chest or head with his arm.
Figure 4.4 illustrates how a valid silhouette may actually describe an impossible pose. The latter, inter-penetration of two objects, occurs when there are environmental assets

in the scene, with which the tracked model interacts. Visible occlusion through overlapping meshes was encountered frequently (see figure 4.5). We wanted to address both of these flaws by exploiting Bullet's collision system.

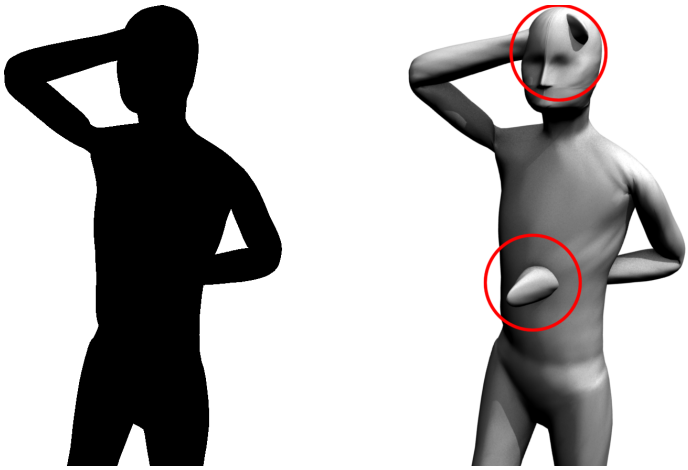


Figure 4.4: Left: Silhouette image descriptor of a human model. The pose looks valid from this point of view. Seeing the actual model (right) though, it is clear that the pose is impossible for a human to assume, since both arms penetrate other parts of the body.

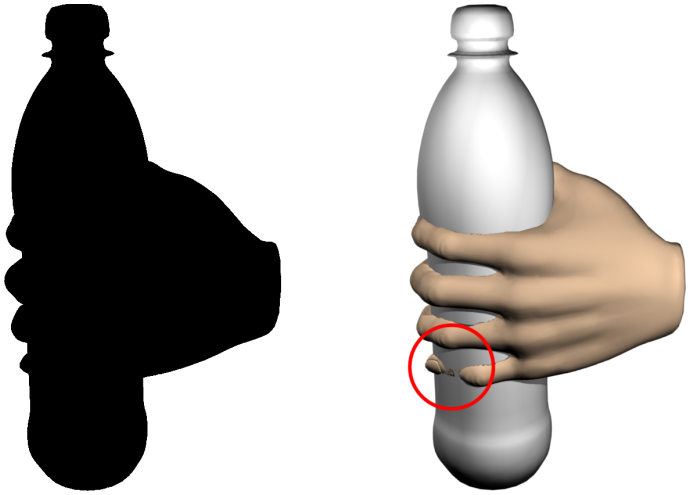


Figure 4.5: Hand model holding a bottle. Again, interpenetration is omitted in the silhouette (left). The models' collision can be clearly seen in the actual rendering to the right.

5 Tracking With Physics

The inclusion of the rigid-body dynamics in the Bullet Physics Library as middle-ware offers several options for improvements:
1. Collision detection routines enable prevention of poses containing inter-penetration of body parts or solid objects.
2. Force and torque calculations make forces analysis possible, which can be exploited for more consistent particle evaluation.
3. Dynamic simulation possibilities can prove useful for multi-object interaction in tracked scenes.
We now want to present the additional properties that have to be taken into consideration with the inclusion of physics, and how we facilitate the model initialization. Afterward, we showcase the structural interface between the physics engine and each of the tracker components discussed in chapter 4.1. Eventually, the physics contact evaluator will be discussed in detail.

5.1 Physics Based Model Properties

There are a few acknowledged rules of thumb when working with physics engines, which we would like to address. To begin with, all dynamic objects should be modelled as convex meshes, or even as convex 3D primitives (boxes or spheres). Collisions between concave objects - or between a convex and a concave one for that matter, will generally require extraordinarily more computation time than convex on convex collision. This has one meaningful consequence for the tracking system as it was. We described our model to be a combination of structural skeleton and mesh skin. The mesh was a single instance that moved according to the underlying bones. This type of model conflicts with the physics engine, since it is very much concave while also needing to be modelled as a soft body instead of a rigid body, to account for deformations. Apart from that, it would mean to recalculate the model's inertia after every change of pose. All in all it would require to go far out of ones way and in the end would probably not yield satisfying results. Instead, articulated bodies in physics engine environments are generally modelled as sets of linked rigid bodies called *ragdolls*. This technique has already been mentioned in chapter 2. The character's body is divided into segments that can each be interpreted as a single convex rigid object. For the simplest interpretation of a human body this would result in an amalgamation of head, torso, upper and lowers arms and upper and lower legs. The exact segmentation can be varied in our system and will be presented in the upcoming chapter.

Rather than replacing the skin and structure setup in place in the tracker, we extend the model to include all relevant physics information in a separate file. This information includes only two individual physics parameters for every bone in the hierarchy: the mass of the corresponding rigid body and the shape to be used as a representation. The remainder of the required attributes can be derived from either structure, mesh or components data available within the Golem Model. From this input, we create a Bullet specific representation of our model, which we simply call the *Bullet Ragdoll*. Each of these ragdolls includes the mesh and structure information contained in a Golem Model in addition to the physical attributes, which are stored within the ragdoll as its own *Physics Model*. It contains a list for bone masses, preferred collision shapes, constraint types, axes and limits, respectively.

Algorithm 1 describes the process of transforming the raw data into a dynamic Bullet Ragdoll. The relevant sub-steps and parameters are described in more detail below. For the sake of clarifying our denomination, we would like to mention that there are always only two representations of the model. One is the armature structure that is used in

Golem, the parts of which we generally refer to as *bones*. The other is the physical ragdoll representation of the model in the Bullet physics engine, which consists of *(rigid) bodies*.

Masses Each segment of the ragdoll is assigned a mass value. In order to solve dynamic forces on an object, we also need to calculate its local inertia. Bullet includes custom algorithms to calculate inertia for each type of convex shape. The result is always an *inertia tensor*, which basically describes the distribution of mass *inside* the given rigid body. More precisely, the inertia tensor is during rotational movement equivalent to *mass* in linear acceleration. The heavier an object is, the greater the force needed to accelerate it. An inertia tensor applies the same principle to accelerating an object's rotation. This applies to all three axes, thus the tensor is generally a 3x3 matrix.
Although the physical mass attribute itself is self-explanatory, there are some pitfalls involved regarding physics engines. It is generally advised to keep the difference of masses of interacting bodies to a minimum, or else extremely unrealistic contact resolutions may occur. This often ends with one or both items flying through the air at tremendous speeds, spoiling any illusion of realism in the simulation. Moreover, most constraint solvers do not function well under these circumstances, leading constraints to break or jitter. This limitation is a generally known, typical problem of physics engines integrators and solvers. The overall mass distribution for our model are also stated in the upcoming chapter.

Preferred Collision Shapes We initially intended to use only capsule and box primitives as approximations of body parts. In 3D computer graphics, the term *primitive* describes a three-dimensional object that can easily be parametrized. 3D primitives are boxes, spheres, planes, capsules, cylinders, pyramids etc. For instance a sphere is perfectly described by its central point and its radius. Further parameters may define its number of faces to provide control over the level of detail in the geometry. The point is that the exact position of the required vertices can be extrapolated from the given parameters. Because of this feature, most primitives are extremely favourable when using computational geometry algorithms.
Using only primitives proved to be an unexpectedly tedious endeavour, because of the way they are implemented in Bullet. The Golem library stores only the position of the head of each bone and its orientations - both relative to it's parent bone, and its length. The mentioned primitives on the other hand have a center world coordinate alongside their construction parameters: radius and height. Projecting the Golem structure to rigid body parameters was easy enough. Unfortunately, it entailed unexpected complications when setting the necessary constraints and especially when trying to take up a *pose* provided by the Golem library. Problematic was the fact that a pose in Golem is basically defined by the rotations of the bones, which are centred around their heads, whereas capsule, sphere and box shapes are not. They rotate around their center, resulting in different poses when performing the exact same rotations as specified by the pose. See figure 5.5 for an example of the problem. This illustration, as well as figure 3.5 also give an idea of how inaccurate primitives represent the model. Nevertheless, we kept the option to chose either a box or capsule as preferred collision shape for each segment.

Eventually, we use a shape that creates the convex hull of the body part in question and has its frame centred on its head-end, similar to the original bone data structure. A convex hull of a number of points in space is the one shape of minimum volume that encapsulates all points of the cloud. Figure 5.1 provides a visual example of the convex hull for Blender's mascot. For every bone's highest weighted vertices in the mesh, a convex hull is created. To clarify, a mesh and its armature are related to each other by a set of *vertex*

Algorithm 1 BulletRagdoll Creation

```
load Golem Model <- (mesh.tsv, structure.tsv)
if model is rigid then
  load model mass <- (physics.tsv)
  set collision shape to triangle mesh
  set components to empty
  get vertex groups
  calculate AABBs
  create triangle mesh
  set start transformation to world origin
  create rigid bodies for use in Bullet dynamics world
  set custom rigid body parameters
else if model is articulated then
  load annotations <- (physics.tsv, components.tsv)
  find global parameters for rest pose
  // compute active vertex groups:
  for all bones in armature do
    if bone has constraint type "none" then
      while bone's parent is neither constraint nor root bone do
        move vertices from this bonegroup to parent bonegroup
      end while
    else
      continue
    end if
  end for
  compute AABBs
  create collision shapes (Algorithm 2)
  for all bones in armature do
    calculate collision shape center offset
  end for
  // compute start transformations:
  for all bones in armature do
    if bone uses convex hull shape then
      assign start transformation directly from rest pose
    else if bone uses box or capsule shape then
      add collision shape center offset to start transformation
    end if
  end for
  // create rigid body objects
  for all bones in armature do
    create default motion state at bone's start transformation
    calculate local inertia for bone's collision shape
    assign motion state, bone mass, collision shape and inertia
  end for
  set constraints (Algorithm 3)
  set custom rigid body parameters
end if
```

Algorithm 2 BulletRagdoll Collision Shape Setup

```
for all bones in armature do
  box shape = new Box with AABB extents
  capsule radius = minimum( AABB width, AABB depth)
  capsule height = 2 * AABB height - 2* capsule radius
  capsule shape = new Capsule with height and radius
  for all vertices in bonegroup do
    transform vertex coordinates from global to local space
    add vertex to shape
  end for
  convex hull shape = build convex hull of bonegroup
  if preferred collision shape = "box" then
    use box shape for this bone
  else if preferred collision shape = "capsule" then
    use capsule shape for this bone
  else if preferred collision shape = "chs" then
    use convex hull shape for this bone
  end if
  set collision margin to 0.0
end for
```

Algorithm 3 BulletRagdoll Constraints Setup

```
for all bones in armature do
  if bone is root or has no constraint then
    continue with next bone
  else
    if bone uses convex hull shape then
      set constraint anchor to bone origin
      while ancestor bone has no collision shape do
        get immediate ancestor bone
      end while
      calculate relative transformation from closest existing ancestor to current bone
      origin
      set ancestor constraint anchor to that point
    else if bone uses box or capsule shape then
      set constraint anchor to -collisionshapecenteroffset
      while ancestor bone has no collision shape do
        get immediate ancestor bone
        add ancestor offset to current offset
      end while
      set ancestor constraint anchor to -ancestorcollisionshapecenteroffset +
      sumofoffsets + currentbonecollisionshapecenteroffset
    end if
  end if
end for
```

groups. Each of the mesh's vertices is assigned to one or more vertex groups (or none if there are parts of the mesh that cannot be moved) and given a *weight* in those groups. Each group on the other hand is tied to a single bone in the armature. Whenever this bone is transformed, the transformation carries over to the vertex group, being subdued for each vertex by its weight in the respective group. This process is known as character *rigging*, and we assume it to be finished on the model.

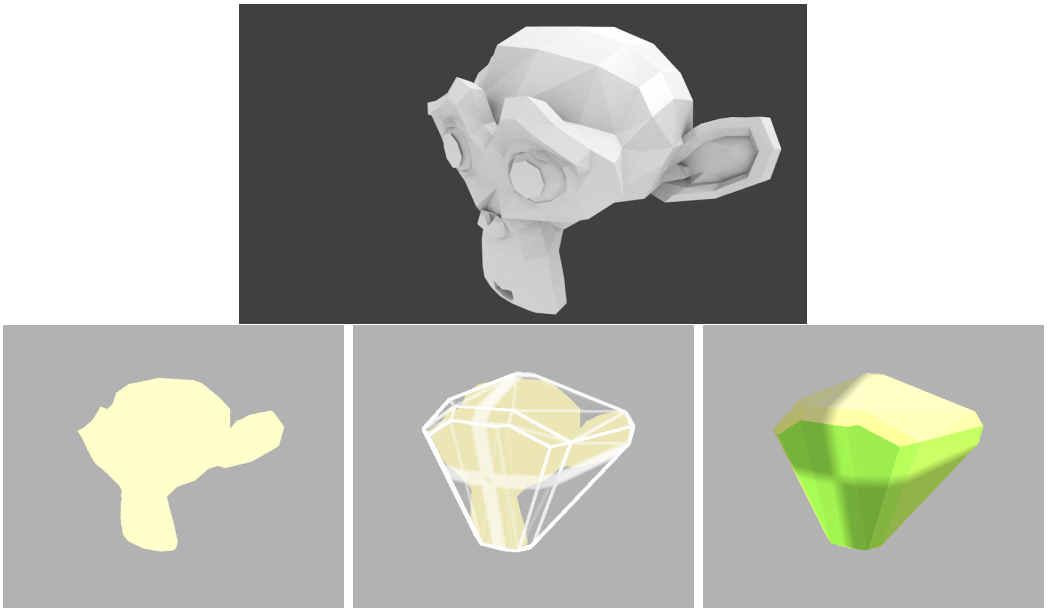


Figure 5.1: Blender's mascot monkey. The top picture is a render of the actual mesh (507 vertices). The bottom row depicts the triangle mesh rigid Bullet Ragdoll (*left*), the corresponding convex hull of the monkey as a dynamic rigid body (*right*) and an edited picture of the triangle mesh shape with overlaid convex hull (*center*).

When creating the convex hulls, the only structural information needed is the vertex groups and weights. We iterate over all vertices and remove them from all vertex groups but the one they have the highest weight in. In a step that will be further clarified in the next section, we then detect the vertex groups that will not have a ragdoll body. They are moved to the next higher hierarchy level, effectively increasing the size of the target shape. Bullet provides a convenient convex hull collision shape, which creates the convex hull of a given list of vertices. We transform the vertices' positions so as to be defined relative to the corresponding bone origin. The benefits of using convex hull shapes are that the rigid bodies of our Bullet Ragdoll have the same alignment as the bones in the Golem model, and they are much more accurate representations of the mesh. The only downside is an increase in computational effort especially during narrowphase collision detection.

We actually use the convex hull shapes to create the primitives, as well. All collision shapes in Bullet have the option to calculate their *axis-aligned bounding-box* (AABB, see figure 3.5). We base the parameters for the primitive shapes on this box template. These shapes are however constructed from the model's mesh, utilizing the vertex group data as opposed to structure attributes (see figure 5.2 for a schematic overview of weight distributions). The difference being that the rigid bodies are neither exactly the same length as the original bones, nor are they aligned with the bones like the convex hull shapes are. To correctly reassemble the model, we do not use the bone length provided in the structure data, but calculate the actual size of the body based on its AABB. This is especially

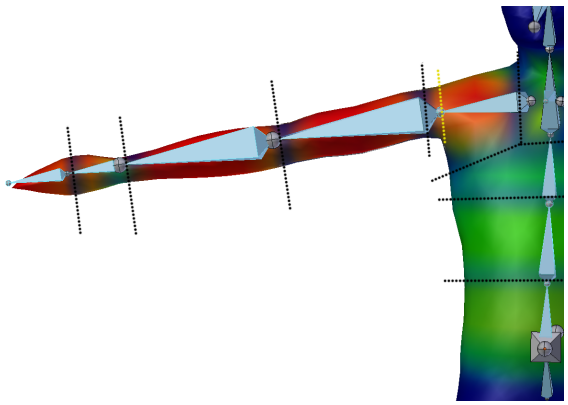
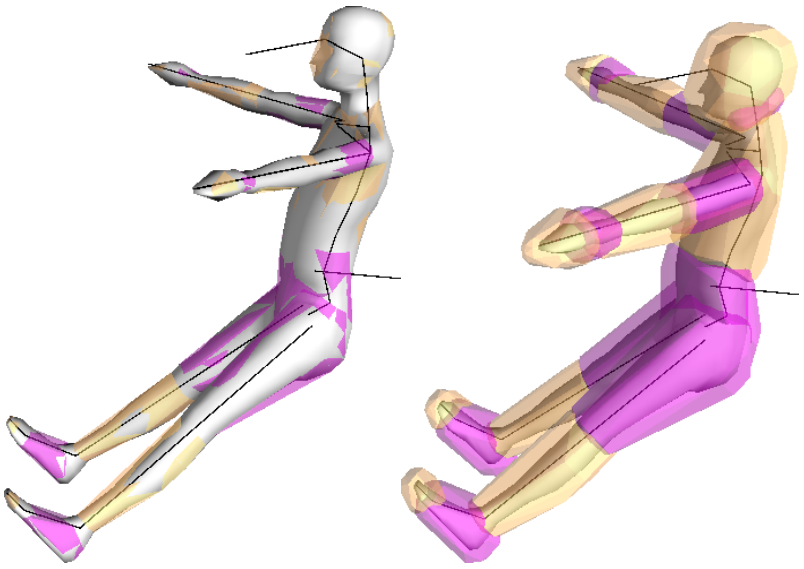


Figure 5.2: Human model armature and its corresponding weights. The black lines were added to make the individual vertex groups better distinguishable. The torso is split into various vertex groups, with some bones affecting only the front of the mesh, and other only the back. This makes it pointless to create a rigid body for each vertex group. The yellow line indicates an example where the length of the bone does not match the length of the respective vertices' convex hull.

important when the structure contains bones that have very sparse weight fields, i.e. only few vertices assigned to them.
Furthermore, models usually include root bones that act as handle to move and rotate the complete model, but do not cause mesh deformation. These bones are ignored in the Bullet Ragdoll. Then again, some bones that do have influence on the mesh may not be suitable for segmented models. For instance, the clavicle bones that control human shoulder regions are necessary for realistic mesh deformation, but these areas would more correctly be part of a single upper torso body when using segmentation. For this reason, we have the option to include the vertices assigned to multiple hierarchically sequential bones to a single rigid body shape, without altering the original model to suit the segmentation. This is achieved by only modelling rigid bodies for bones that are assigned to a constraint. If a bone does not define a constraint, its assigned vertices are recursively projected onto its parent bone, until either a moving (i.e. constrained) or the root bone is found.

Bullet allows to set collision margins for rigid bodies. These are set to a small number by default, causing the Bullet model to not line up properly with the actual mesh. The margins basically "inflate" the shape like a balloon, increasing its size noticeably. Even though it is recommended to maintain a small margin, we did not encounter any problems after removing it. See figure 5.3 for a comparison between the model with and without default collision margin. It is easy to see why the collision margin has a dramatic effect on collision scores.

Start Pose With the collision shapes set up, Bullet only needs the mass and start transformations to create the rigid body parts of the ragdoll. Masses are predefined by hand (see section 5.2) for every segment. The start transformations are calculated from the model structure, which upon loading describes an initial rest pose. At this point we need to distinguish between the segments using primitive shapes and those using convex hulls. Because the convex hulls have their frame origin aligned with that of the Golem structure, we simply have to use the bones' global coordinates as starting points for this type of collision shape. Due to their centric layout, starting positions for segments using



(a) (b)

Figure 5.3: Original skin mesh with Bullet Ragdoll overlay. The ragdoll's segments are coloured in alternating yellow and purple to be better distinguishable. (a) shows the ragdoll without collision margin, (b) has the default collision margin enabled.

primitive shapes have to be corrected. For this matter we calculate the *bone offsets* for every shape by finding the distance between the shape's AABB's center and its bottom on the Y-axis. The body is moved upwards by this amount to settle in a position that conforms the actual position of the bone.
Once the initial position for every segment in the rest pose is determined, we create a Bullet rigid body object for each one. It includes the respective collision shape, mass and a utility handle for object movement called a *motion state*. Bullet assigns a motion state to all dynamic objects in the simulation. It is suggested that all specific movement should be handled through these motion states. They basically provide an interface to control objects in the simulation within a running game engine. Initializing the motion state with a non-identity transformation entails the rigid body to start at a specified point, once it is added to the world. Interestingly, the rigid body itself also contains attribute fields for its current position and rotation. We discovered that to actually move a rigid body to a new position, *both* motion state and rigid body need to be reset. Due to this inconsistent behaviour, attempts to create a custom motion state class that would handle both convex hulls and primitive shapes similarly failed, so we stuck to the default motion state provided by Bullet.

Constraints So far, masses and shapes are assigned to rigid bodies, which have been moved in place to correspond with the specific starting pose for the model. The next and last step in initializing the ragdoll is establishing constraints. In Bullet, constraints are handled separate from the respective bodies. They are created by choosing a constraint type and assigning a pair of constrained members, in our case rigid bodies. Anchor points are defined in each body's local frame, which specify at what spatial point the constraint is to be applied. Rotational limits will eventually be measured with this point as pivot. We need to differentiate again between rigid bodies that contain a convex hull shape and those with primitives. The former can be specified without much effort, since the pivot point for the bone is its frame origin and the anchor on the parent bone is the relative bone offset, which is directly accessible through Golem. Primitive shapes need to be re-aligned

again because of their centric layout. Figure 5.5 illustrates the process and highlights the differences between the use of convex hull shapes and primitive shapes.
Constraint setup in our case grows more complicated with larger segmentation steps. Like we explained earlier, it is often sensible to reduce the number of hierarchical components for the Bullet Ragdoll. For instance, most of the vertebrate bones will be condensed to a single "thorax" rigid body. The Golem data we read from, however, still contains all of the small offsets from bone to bone. A simple summation of these offsets may accumulate small errors. For this reason we take advantage of the fact that the rigid bodies are already positioned in the start pose. We use the rigid bodies' motion states to calculate the offset from the closest active ancestor segment (i.e. the next bone that is either a constraint or the root) to the current active segment. In the case of primitive collision shapes, the priorly computed offsets are again used to compensate for their centric frame.
By default, Bullet filters constrained pairs when determining collisions, so the constraints need to be well-defined if the body shapes are not to penetrate each other. The process of setting constraint limits and choosing a proper initial rest pose for the model is outlined in detail in the next section.

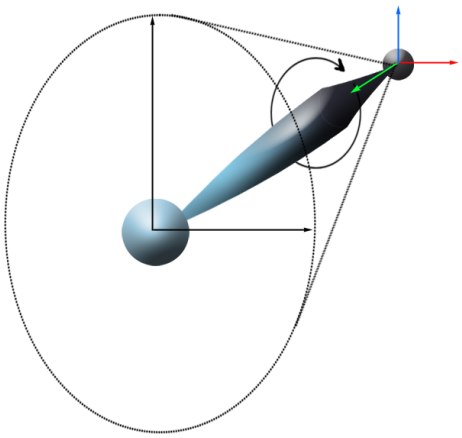


Figure 5.4: Stylized example of a bone with cone-twist constraint, also sometimes referred to as ball-and-socket constraint. The far end shows the origin frame for the bone (in red, green and blue). The cone displays the freedom of movement for the bone. It is described by limited rotations around the bone's local X and Z axes (red and blue). Rotation around the Y axis (green) is the so-called twist and may be limited as well. This type of constraint is based on biological spheroidal joints, like shoulder or hip articulations.

Although many unique types of constraints are available in Bullet, the ones that correspond to our components types are *btHingeConstraint* and *btConeTwistConstraint*. A hinge constraint is best described as an analogue to a simple door hinge. The door may swing open and shut, but is restrained to a single degree of freedom (DOF): the rotation around the hinge axis. Hinge constraints in Bullet are given a minimum and maximum rotation angle, as well the axis that defines its single DOF. The axis is typically one of the basis axes in the appropriate body's local space. Cone-twist constraints on the other hand have three DOFs, allowing rotation on all three axes. Bullet's cone-twist constraints require three parameters, one defining the Y-rotation or "twist" of the body and two angle parameters that span the cone in which the body may rotate. This type of constraint is especially well suited to describe the movement range of the human shoulder articulation. See figure 5.4 for a visual example. In order to be compatible with the tracker's components list, the two angles defining the swing limits need to be equal, so as to describe a round cone. An

in-depth description of component initialization is provided in the following section.

Rigid Model Ragdoll Inarticulate models can be loaded by simply supplying a structure of a single or even no bone at all. The Bullet Ragdoll will then be initialized as a single rigid body, having a triangle-based, concave collision shape. Golem provides the triangle mesh informations needed and Bullet has a ready-to-use collision shape implementation for triangle meshes. Although this way, any type of object may be represented in the dynamics world, Bullet neither contains algorithms to determine inertia of triangle meshes nor for detecting collisions between two concave shapes. According to comments in the source code, triangle meshes should only be used as static scene assets. Compound Shapes are suggested instead, which are basically containers for convex shapes that can be solved iteratively. This is very unfortunate, because aggregating a concave form with convex components is not easily automated. Furthermore, in our experience, inertia calculation for these compound objects is also dubious at best. This issue will be discussed further in section 7.2: Limitations.

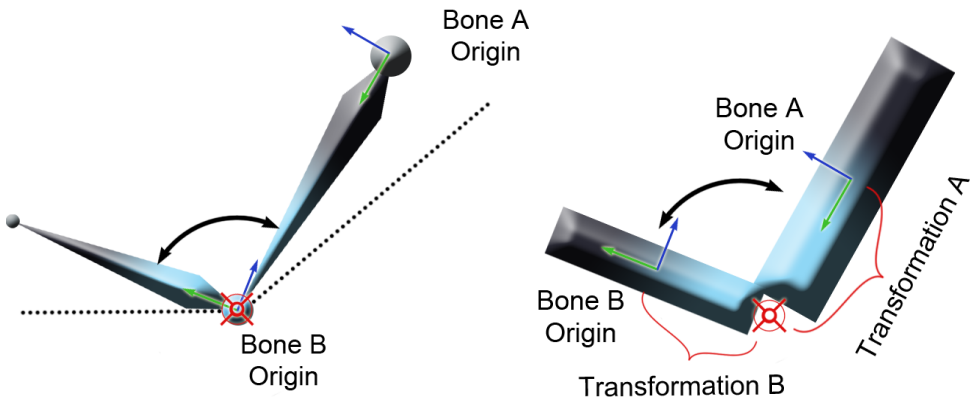


Figure 5.5: A structural hinge constraint between two bones as described by either convex hulls (left) or box primitives (right). To simulate an elbow articulation, we define a hinge constraint with a certain limit of torsion (black dotted lines) and one transformation matrix for each body. Matrix A describes a transformation from Bone A's origin to the red reticule, because that is where we want the child bone's center of rotation to be. Transformation B does the same for Bone B's origin. When using convex hull shapes, the bodies' frames will be aligned with the head of each corresponding bone, enabling us to simply use the bone offset information available in the initial model structure. The child transformation is not needed since the constraint is placed directly on top of the bone's natural rotation center. Using box or capsule primitives, more calculations have to be performed, because the bodies' frames originate in the center of their respective shape. Depending on the amount of active offsets for the child bone, a whole series of transformations is needed to correctly establish the constraint center.

Physics Demo Having adjusted the former model structure to work with physics parameters and a segmented ragdoll representation, we extended one of the original Bullet demos to provide a testing and debugging environment for our ragdolls. The features we implemented are:

1. Projecting a ragdoll onto a *pose* as used by the Golem library. This is achieved by

applying the same procedure from the starting pose initialization.
2. Changing single parts of the model to an unmovable state to simulate the body holding onto the environment. Bullet uses *Collision Flags* to describe an object's behaviour in the simulation. The three most common flags are *static*, *dynamic* and *kinematic*. Static objects are assigned a mass of 0 and are basically unmovable. Dynamic objects will collide with them, but they are not subject to any forces like dynamic objects are. Kinematic objects behave similar to static one, but can be moved explicitly. The purpose of kinematic objects is in being user-controlled objects that still interact with the dynamic world. Single rigid bodies can be switched from dynamic to kinematic objects and back at any time during the simulation, effectively freezing them in pose.
3. Test constraint behaviour. Bullets accompanying demos provide a visual OpenGL rendering of the dynamics world along with basic control input. The view can be panned, rotated and zoomed at will, and several shading options can be toggled. More importantly though, a mouse-click on any dynamic object in the view will create a temporary constraint between the clicked body and the current mouse position (projected into 3D space). This basically means that we are able to drag the ragdolls across the screen, grabbing them at any point of the model hierarchy. It is an easy way to estimate how constraints are set up and how their behaviour coincides with our expectations.
4. Perform both self and foreign body collision tests that yield a penetration depth. See section 5.3.1 for the full details of the collision detection.
5. Calculate linear and angular forces acting on each body. See section 5.3.2 for the full details of the collision detection.

5.2 Model Setup in Blender

We now want to show a brief explanation on how to set up a model for physics based tracking in the 3D editor *Blender*. Blender is an open source tool to model, animate and render 3D assets. There is an existing Bullet integration in Blender, which is mainly used in conjunction with Blender's game engine. Note that we do not make use of this particular Blender feature, since it does not provide the full freedom of working with the full Bullet source.
As a human prototype model, we use the RAMSIS model described in [7]. We say prototype because this is not the final model to be used in the tracking system. Instead, a 3D projection of the actual person being tracked is used to parametrize the prototype so that it resembles the tracked person as closely as possible. However, the model preinitialization still requires the setting of desired limb components, starting pose and all the physical parameters like mass and desired collision shape. Aside from the mesh that reflects the human, RAMSIS contains an armature based on ergonomic human skeletal features.
A python script previously only extracted the vertex and edge information of the mesh in one file, and the armature structure in another, leaving the necessary component file to be written manually. Blender does however have an own integrated constraint system for character rigging. We make use of that to allow the export of the most common components parameters: hinge and ball-and-socket constraints. To set up a constraint, Blender offers a dedicated constraints tab with plenty of options when selecting part of an armature. The only one needed to simulate a hinge or ball-and-socket component is the *Limit Rotation Constraint*. With it we can define limits for the three rotational DOFs of the bone. The export script will then detect whether or not a constraint has been set and check for the number of axes that are limited to 0°. If more than one axis has a move range higher than 0°, the component is considered ball-and-socket. This type of constraint is described by its cone and twist axes, which are automatically calculated,

and the maximum rotation values around either axis. For a hinge constraint, the axis of movement is stored besides a minimum and maximum rotation value. The effects of the constraints can further be directly evaluated when accessing the armatures pose mode and simply right-clicking and moving the bone in question. It will not be movable outside of its constraint limits. This allows for much quicker and more reliable components definition.

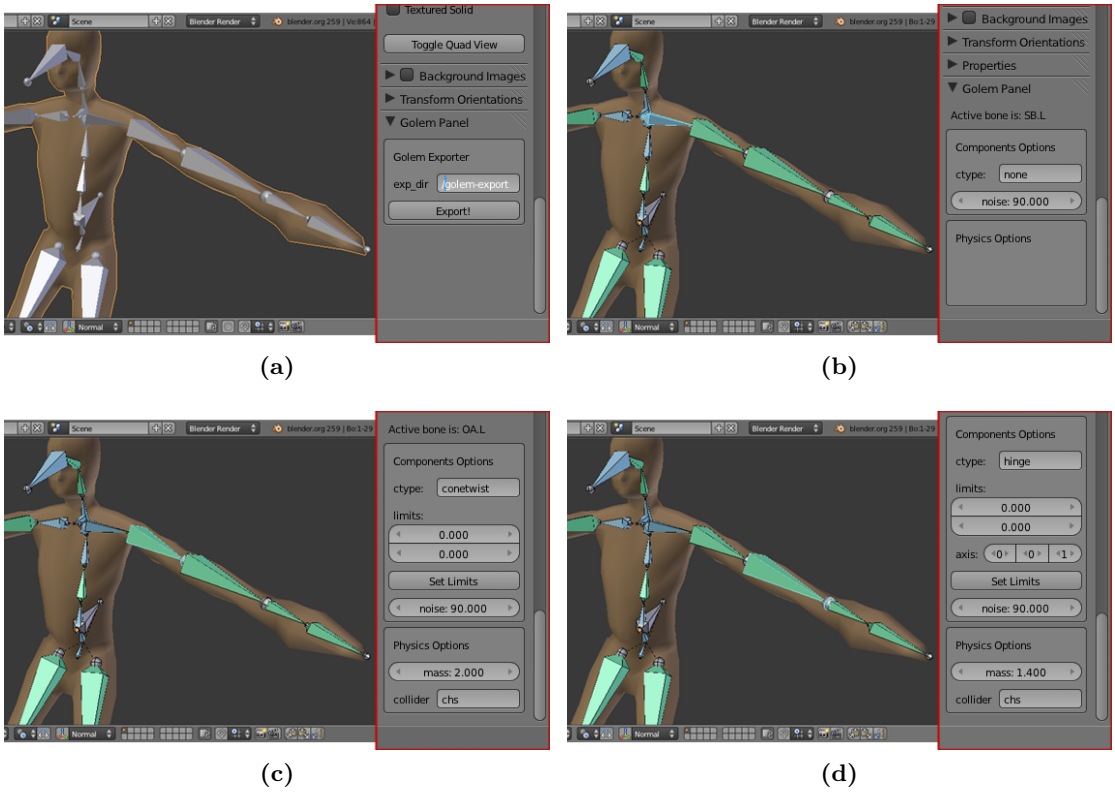
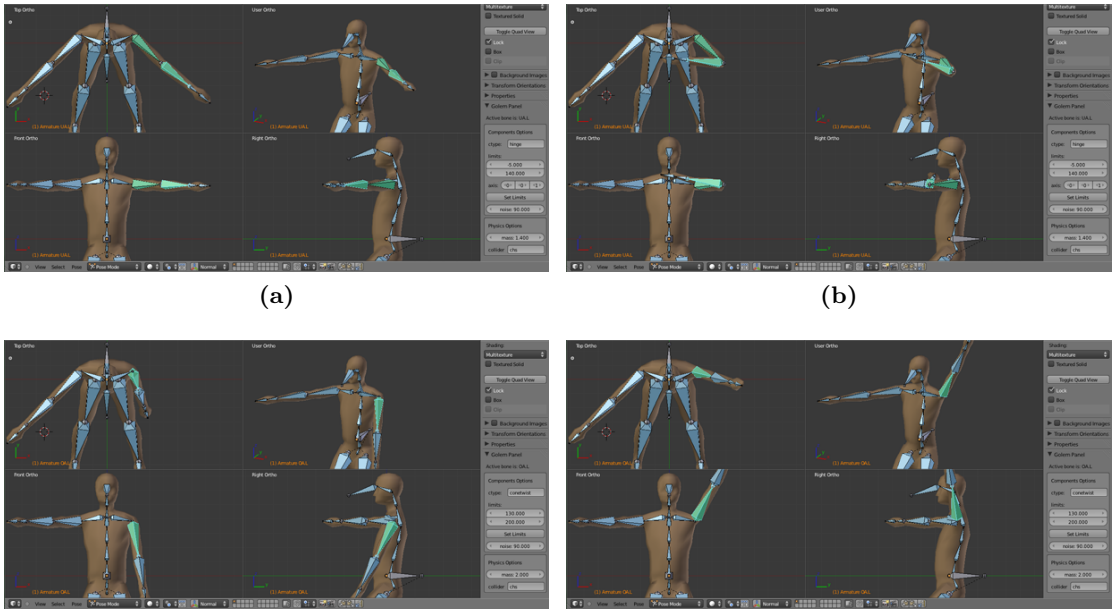


Figure 5.6: (a) Mesh is selected, panel shows export directory and export button. (b) Inactive bone is selected, panel hides constraint and physics fields. (c) Bone with cone-twist constraint is selected, panel shows twist limit (top) and swing limit (bottom) fields as well as physics options. (d) Bone with hinge constraint is selected, panel shows limits and rotation axis fields as well as physics options.

Since Blender is easily accessible through Python and has a uniquely customizable user interface, we have written a second script that facilitates the constraint setup for being closer to the Golem components structure. Algorithm 4 describes how a new UI panel is initialized for that matter. Because the script works on the editor's selected context, it will only run if a valid bone is currently selected. This suggests that the armature must be set to edit or pose mode, otherwise it is not possible to select single bones. Note that Blender uses two parallel implementations for each bone. One contains all the structural and geometrical parameters, while the other is used to determine pose information. We utilize the latter, a so-called *pose-bone* object. First, all pose-bones of the selected armature are scanned for custom properties. Blender allows to set custom properties for any object, in order to increase customizability. We make use of Blender's ID-properties, because this allows the information to be saved within the .blend file, as opposed to simple custom properties, which are lost upon reloading the file.

Since our needed component and physics attributes are not naturally part of Blender, we



(a) **(b)**

(c) **(d)**

Figure 5.7: Examples of how upper and lower arm are restricted by constraints. Pictures (a) and (b) show minimum and maximum angle for the elbow hinge articulation as seen by top, front, side and perspective view inside the Blender 3D editor. Pictures (c) and (d) likewise demonstrate minimum and maximum values for the shoulder joint's X-rotation.

define these as custom properties for each bone: the constraint/component type, exactly two limit values for its rotation, and a noise value to define the components. Specifically for the physics simulation, all relevant bones also have a mass associated with them. We also included the option to change the desired collision shape to be used in the simulation, even though primitive shapes may cause problems within the constraint setup as described later on. Influence and meaning of these custom properties may vary for individual component types, and is described below, and illustrated in figure 5.6.

The type field expects an input of either *none*, *hinge*, or *conetwist*. None entails that the body has no DOF in relation to its parent, and will not be part of any component. In Bullet, these bones will not be modelled at all, but rather have their assigned vertices integrated into their parent bone's shape. Consequently, a bone can not be assigned primitive collision shapes if one or more of its child bones are not assigned component types. More importantly, the bone may not have a mass value. This needs to be considered when determining mass distribution among the bone hierarchy. We suggest that the mass of a bone, which is not part of any component, is to be included in its parent bone by simply adding the values.
A bone designated hinge will allow to set a mass and preferred collision shape, as well as the input of the minimum and maximum rotation and the rotational axis to be used. This axis is limited to one of the basis vectors X, Y or Z, because of the way it is implemented in Bullet. It is very easy to misjudge the bones' local frame orientation, thus we suggest to test the rotation in the 3D-view after setting the limits.
If designated conetwist, the axis field disappears from the panel while the two limit fields stay there. These do however not define an upper and lower limit any more. Instead they represent the allowed swing (top field) and twist (bottom field) angle. Figure 5.4 illustrated the rather abstract concept of swing and twist rotation. Components setup

does not allow the swing range on the X axis to differ from that on the Z axis, as it would when describing a non-round ellipsoid as cone base. Also, the minimum and maximum swing values are always of equal magnitude, meaning that a cone-twist constraint rotating from -30° to +140° can not be modelled. We suggest to keep this restriction in mind when deciding upon an initial rest pose for the model. To remedy this limitation, the character can be posed so that all body parts bound by cone-twist constraints are centred within their range of movement. Like before, the setup for conetwist constraints also allows for the input of component noise, a relative mass value, as well as the setting of the desired collision shape. Figures 5.6 and 5.7 present examples and explanation for all possible adjustments.

Algorithm 4 Golem Panel Initialization

```
1: if armature selected & edit or pose mode active then
2:    for all bones in armature do
3:       check bone for custom properties
4:       initialize missing custom properties
5:    end for
6:    register SetLimitsOperator
7:    initialize components fields
8:    initialize physics fields
9:    register GolemExportOperator
10:   initialize export fields
11:   register GolemPanel
12: else
13:   throw error
14: end if
```

Algorithm 5 Set Limits Operator

```
1: if bone selected and in pose mode then
2:    if bone has no constraint limits so far then
3:       add a Rotation Limit constraint to bone
4:    end if
5:    set active constraint space to LOCAL
6:    if bone has ctype 'conetwist' then
7:       swing_limit = upper limits field
8:       twist_limit = lower limits field
9:       set X-rotation limits to ( - swing_limit / 2, + swing_limit / 2)
10:      set Z-rotation limits to ( - swing_limit / 2, + swing_limit / 2)
11:      set Y-rotation limit to twist_limit
12:   else if bone has ctype 'hinge' then
13:      min_rot = upper limits field
14:      max_rot = lower limits field
15:      rotation axis = axis field
16:      set (X, Y, Z) axes minima to axis * min_rot
17:      set (X, Y, Z) axes maxima to axis * max_rot
18:   end if
19:   activate all axes rotation limitations
20: end if
```

Algorithm 5 describes the process of transforming the limits-tuple specified in the Golem Panel into a Blender-specific LimitRotationConstraint, which can be visualized and tested in the Blender 3D-view. Blender has no specific hinge or ball-and-socket constraint types. Instead, it uses generic constraints limiting either rotation or translation. Both hinge and cone-twist constraints have to be modelled by defining upper and lower rotation limits for all axes of the bone. A cone-twist joint may be rotated around all three axes, while a hinge joint will typically have two axes limited to 0° rotation. Hence the necessity to translate our component parameters into a format suitable for Blender, and eventually back to our component format in the final export.

The actual export script can be called through the panel when selecting a mesh with an assigned armature. The exporter script originally wrote the mesh and armature information to external files. Information for the mesh includes all vertex positions, all faces of the mesh as describe by triangles and the vertices' corresponding bone groups and weights (responsible for mesh deformation). The armature was described by bone identifiers and their hierarchical configuration. Since we wish to keep the tracking system modular, the physical attributes belong in a separate, dedicated file simply called *physics.tsv*. It contains the mass and preferred collision shape for every bone in the hierarchy, including those that will not be rendered, eventually. Like we already mentioned, the extended tracker is also able to create the formerly hand-written components file from Blender rotation constraints. First we identify the type of constraint and retrieve the rotation limits accordingly. Then the relevant rotational axes are determined, which is trivial for hinges but a little crafty for cone-twist constraints, since they have two important axes: a swing axis and a twist axis. The former needs to be aligned with the body's Y-axis in order to translate properly into Bullet, hence the centring of the bones in the rest-pose. The latter describes the initial orientation of the Y-axis (i.e. 0° Y-rotation). Lastly, the custom noise attribute is read and included in the components file, as well.

When working with ragdolls, an essential factor for the physics simulation's performance is the segmentation chosen for the ragdoll body. Although it is relatively easy to break the shape of a body down to a combination of rigid shapes, it is important for collision and constraint solving to maintain a certain dimension of size and mass throughout all rigid bodies. It is important to note that this segmentation is not subject to the same demands as the model's armature. When defining an armature for mesh deformation, an increase in bone numbers - for instance vertebral bones, leads to more vertex weight overlapping and smoother mesh deformation. Translated to a segmented model, however, this means a higher effort to maintain constraints, an increase in unjustified collisions between these overlapping segments. Though this could be solved by introducing collision flags, movement would still cause unnatural gaps between the rigid segments, further degrading simulation accuracy. An intuitive segmentation of the human body is pictured in figure 5.8.

Armstrong et al. [1] divided the body into 17 main volumes, separated by major articulations. These volumes include the head, neck, thorax, abdomen, pelvis, upper arms, forearms, hands, thighs, calves and feet. Using regression equations, they calculated the relative masses of these volume segments for small, medium and large sized males. A comprehensive overview of similar studies can be found in [14, p. 2, 3]. We used this weight distribution chart to estimate the mass values for each rigid body in our physics ragdoll. Nevertheless we opted to introduce our own segmentation for the human body. Considering that most human movements do not contain a lot of torso movement, many tracking implementations even consider the whole trunk of the body a single rigid segment. We divided the torso just below the chest, combining abdomen and pelvis in a single

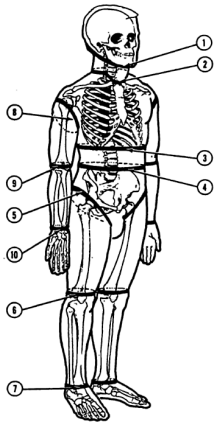


Figure 5.8: Typical segmentation of the human body when creating a human model with rigid bodies. Each segment remains more or less rigid throughout any movement the person may take. Individual segments are: head, neck, thorax, abdomen, pelvis, humeri, forearmes, hands, femora, tibiae and feet. This segmentation scheme is very intuitive and prominent in both computer and anatomy sciences [1, 14, 42].

segment. Arms and legs do not allow for much variation in this regard. Hands, however have a history of being notoriously hard to model. Being a tool for both interaction and expression, human hands are a vital asset to any character. Animating a human hand can be painstaking because of the enormous variety of subtle pose it can take. Similarly, recognizing and tracking a human hand can be difficult because of the large variation of forms and poses it can assume. In whole body tracking tasks, we segment the hand as detailed as possible with the RAMSIS model: one segment for the palm and one for the extended fingers. Because none of our sources took measure of the exact weight distribution inside a single hand, we guess a relative mass of 60% for the palm and 40% for the fingers. We proceeded similarly for the feet, resulting in 30% foot weight in the ball of the foot and toes, and 70% in the main body of the foot. Figure 5.9 illustrates the final human ragdoll setup, including segment masses, articulations and constraints.

5.3 Interface Implementation

Integration of our Bullet extension into the existing tracking system is achieved by a neutral physics interface. The point was to create an interface that does essentially not depend on Bullet, so it may be possible to add an implementation using another physics engine at a later point. This general interface does only contain a few essential function calls: initialize and cleanup functions, a method to add a model to the simulation, the update call to step the simulation and two methods that compute collisions and forces, respectively. A visualization function provides the opportunity to render the simulation within an OpenGL context. Our implementation then makes full use of the new Golem Physics Library, the contents of which are illustrated in Figure 5.10.

The physics interface implementation is the controller class for the physical simulation. The dynamics world object is stored within, which is responsible for handling the whole simulation. It contains all relevant parameters for world size, references to the collision algorithms and constraint solvers in use, as well as data structures to store and handle all rigid bodies and constraints in the simulation. During initialization, a dynamics world is created with default parameters. By default, utilization of an *axis sweep* broadphase collision algorithm and the *sequential impulse* constraint solver are designated.

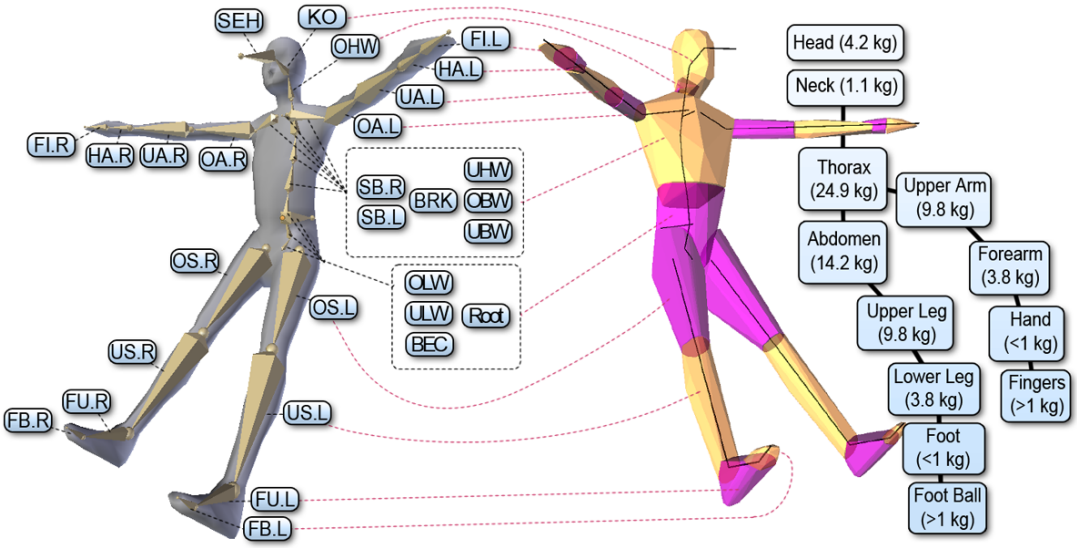


Figure 5.9: To the left is the RAMSIS based model consisting of bone hierarchy and skin (bones are named). The right hand model is the convex hull representation of the same model used within Bullet. The dotted lines indicate to which rigid body each bone is assigned.

Since Bullet does not include annotations for related hierarchical structures but instead saves all rigid bodies in a single list, the interface additionally stores a vector of all the *Bullet Ragdoll* objects that have been added to the simulation. Each ragdoll can be called individually to gain access to its assigned rigid bodies and constraints (if any). Ragdolls are added to the world by loading a Golem Model. So there is a dependency between the interface and the model data library already. Otherwise, loading models works exactly as previously described for the Bullet Ragdoll.
The update function accepts two parameters that specify the time step size as well as the maximum number of substeps performed. Bullet uses a substep mechanic to maintain a real-time simulation. The reason for this is that the time step size may not be constant, and Bullet automatically create a number of substeps for interpolation. The problem we encountered by using substepping was that access to intermediate results during those substeps was limited. We consequently removed substepping by setting the parameter for max substeps to 0. Doing this however requires us to include a constant time step management when calling for updates. We will see that the tracker does not need a constantly running simulation, so this is only an issue in our demo application. There, time steps are restricted to a constant 50 milliseconds.

In order to perform collision detection, our library introduces a *Collision Domain* for each ragdoll in the dynamics world. By calling the compute collisions method in the interface, these collision domains retrieve all contact points within a pose space, including the local poses of every ragdoll in the scene. Mind that collision resolve is not of interest here, since we want to determine the validity of a specific pose more than finding contact resolution. The result is a list of collision data objects, which include the intersecting bones and which models they belong too, as well as collision depth. The process is lined out in more detail in section 5.3.1 below.

In a similar fashion, the interface includes a *Forces Handle*, which provides force analysis functionality. We already mentioned that getting results from inside the dynamics step loop is problematic. This also applies to the acting forces in the simulation. Bullet clears all force information after every time step. Recapitulating, Bullet deduces dynamic objects'

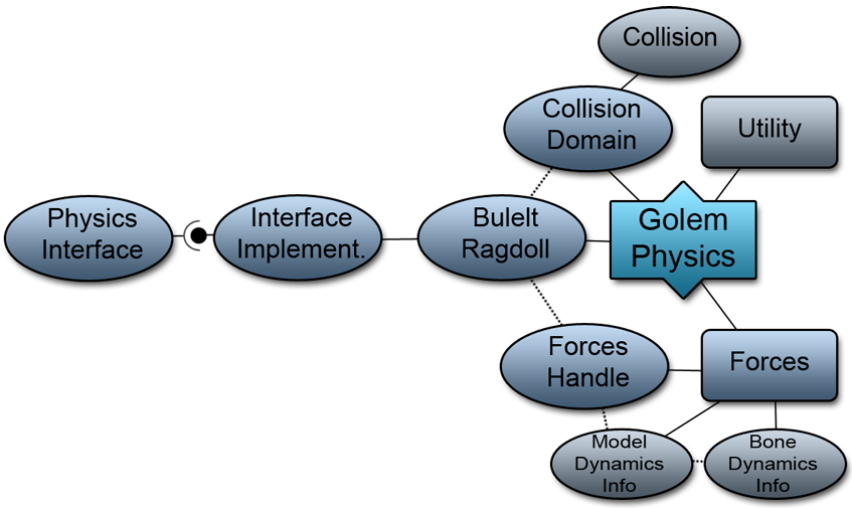


Figure 5.10: Overview of our Bullet Physics framework, titled Golem Physics due to being part of the Golem library. Aside from the obligatory physical description of the Golem model, the package provides functions for collision and force detection. An interface provides outside access to these methods.

positions by evaluating all forces acting upon the object. These forces are in turn computed according to velocities, accelerations, masses and moments of inertia information of single objects and interaction between objects like constraints and collisions. Our own approach to force calculation is less complicated, and will be explained in more detail in section 5.3.2 below. The basic idea is to take samples into the next couple of steps of the simulation. We sample every body's position and rotation in the world, then step the simulation by a constant time interval, and continue taking samples. To acquire the starting pose again, we reset all bodies to their initial positions, rotations, velocities and accelerations. Results of this step are stored for each body independently as *Bone Dynamics Info* objects, which are aggregated in their respective ragdoll's *Model Dynamics Info*.

Lastly, the visualization for the dynamic world is meant to be used to draw all contents of the world to an OpenGL context. More importantly though, the visualize function also takes a single parameter: a *GlobalPose* vector that describes the poses of each model in world coordinates. Each ragdoll immediately assumes the given pose. This allows us to align the ragdoll in the physics simulation with the actual model during optimization in the tracker. In order to have an actual visual representation of the physics ragdoll, each collision shape can be drawn in OpenGL. We based this feature on the OpenGL demos that are supplied by Bullet. To better distinguish between the model mesh and the ragdoll, the physics model can be drawn semi-transparent. We already showed this feature in figure 5.3 within section 5.1.

An overview of the complete library dependencies is depicted in Figure 5.11. Bullet is used in the evaluation component of the Model State Space library. An exact outline of how hypotheses are evaluated due to contact restrictions will be given in the next section. Following the collision evaluator, we present the implementation of our force calculation algorithm.

5.3.1 Golem Physics: Collisions

We already described the entities involved in collision detection: a *Dynamics World* forms a *Collision Domain* with all *Bullet Ragdolls* within. These are all accessible through the

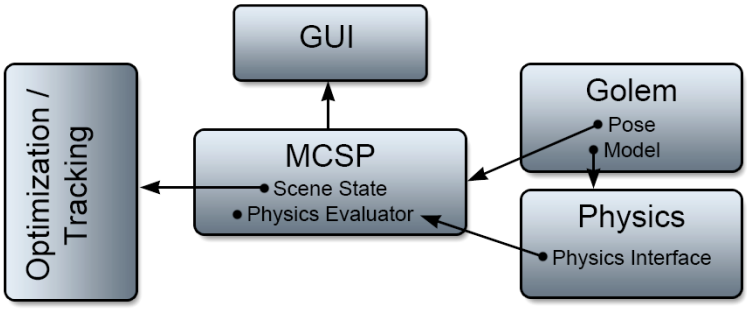


Figure 5.11: Top-level view of the tracking system's components and their main access points, including the graphical user interface and the new Golem Physics package.

Bullet Physics Interface, which may invoke a collision computation. This invocation is accompanied by a global pose parameter for each ragdoll in the collision domain. The ragdolls are immediately moved to their respective poses, regardless of their active constraints or other forces. Once in position, the collision domain is called upon to perform the collision test. Bullet features a *contactPairTest* method, which is presented with a pair of rigid bodies and returns the contact results as a callback object. We use this method to check for collisions between all rigid bodies in all ragdolls, effectively covering both self- and foreign collisions. Remember that the Bullet engine does not explicitly know of the ragdolls or any cohesiveness among the rigid bodies. Hence we need to forward the model and bone IDs to the resulting *Collision* data structure. The remaining collision data is extracted directly from the aforementioned *Contact Threshold Callback* object. The callback allows access to contact information that is otherwise hidden within the narrowphase collision detection algorithm. The one and only attribute we are interested in, is the depth of penetration, i.e. how far one body is intersecting with another. The way Bullet's continuous collision detection works, this attribute may sometimes be a negative value, meaning that no actual penetration has yet occurred, but the bodies are about to collide. This information, however, is unimportant to us at this stage, so we introduce a contact threshold of 0. All penetration values beneath that threshold are consequently ignored by the callback and not included as a collision result. At the end of the process the interface provides a list of collisions, which in turn contain both bones' identifiers, which ragdoll they belong to (also if they belong to the same ragdoll), the penetration depth of the collision and lastly both bones' approximated radius. The latter is calculated under the assumption that most rigid bodies in a Bullet Ragdoll will be aligned to one another along their Y-axis and are roughly cylindrical in shape. Hence collisions are more likely to occur on the body's sides than on its top or bottom. Therefore we approximate the radius by calculating the hypotenuse for the width and depth of the bone's AABB: $radius = 0.5 * \sqrt{width_X^2 * width_Z^2}$.

Collision Evaluator The Physics Collision Evaluator is not part of the same library as the Bullet Interface. It belongs to the Component State Space Library described in section 4.1. Consequently, the evaluator is independent of the Bullet engine itself and works with the mentioned collision data structures provided by the physics interface.
Each occurring collision in the list is evaluated individually and given a distinctive score from 0 to 1. A final score is then computed from these fraction values, representing the score for the overall pose. We tried two different approaches for this task. The first was defining a sigmoid function that evaluates individual collisions, which is shown in

equation 2.

$$S_{total} = \prod_{i=0}^{c} \exp\left(0.5 * \left(\frac{x_c - t}{\sigma}\right)^2\right) \tag{2}$$

This function will produce a result between 0 and 1 for any given collision depth x_c, while allowing a threshold of t, meaning that if $x_c < t$ the score will remain at its maximum. The total score would then be the product of all individual scores. The expectancy value σ defines the function's compression, effectively ruling the sensitivity of the collision evaluation. Because models might be of vastly different dimensions, we derived σ from the overall average bone length in each model. This provided somewhat more homogeneous results for differently sized models. The problem we encountered with this approach was that the scores where very diverse. Poses with total scores in between 0.1 and 0.9 were very common. For a quick and reliable evaluation it was preferable to have the score drop from 1 to 0 more rapidly.

For that matter we decided to scrap the sigmoid function and go with the more linear approach presented in equation 3.

$$S_{total} = \prod_{i=0}^{c} \left(1.0 - \left(\frac{x_c - t}{o - t}\right)\right) \tag{3}$$

Single scores are set to 1.0 to start with. The degression in score is dependent on the collision depth x_c, a tolerance level t and a zero-score threshold o. With this function we do not need to make extra assumptions as to the size of the model, since tolerance and zero-score threshold are both derived from the currently colliding rigid bodies. This is the reason we included the bone radius in the collision data object. More precisely, the smaller of the two radii is used to determine the tolerance in which the collision is ignored, and the depth that will cause the score to drop to zero. Both values are defined as percentages of the given radius and are freely customizable. We encountered satisfactory results with a 5% tolerance and 10% zero-score setting. Figure 5.12 illustrates the impact of using percentile values in order to accommodate for models of varying sizes. Fix numbers would likely lead to inconsistency when evaluating models of vastly different sizes. The total pose score is then again the product of all individual scores.

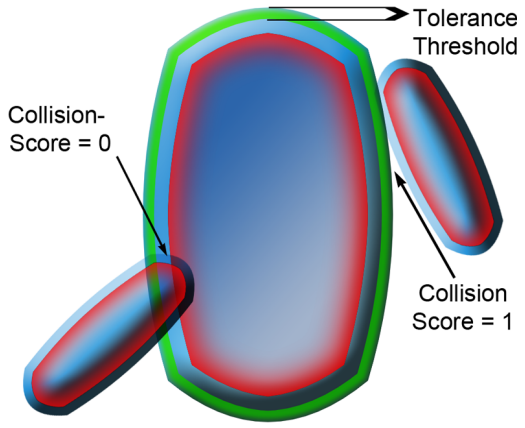


Figure 5.12: Schematic scene containing three rigid bodies of varying sizes. The red marking indicates the 0-score threshold. Any penetration that exceeds this depth is given a collision score of 0. The green border is the tolerance threshold. Penetration up to this depth is ignored. A score between 0 and 1 is given to collisions with a penetration depth in between the two threshold values.

A certain collision tolerance is advisable, since the model representation is not a perfectly accurate one. Even when using convex hull shapes, some dissimilarities are created. The best example are the collision shapes for human thighs. The natural concavity below the gluteal area is lost due to the convex modelling. Inaccuracies like this may cause the tracker to discard perfectly correct hypotheses because of slight collisions. A similar example would be a collision that does really occur. The correct hypothesis may be discarded when in reality the tracked person is really pressing a finger into the soft tissue of his forearm. To allow some freedom of touching body parts, we specifically set the tolerance for the collision to be ignored. The linear approach works better in sustaining perfect scores where no collision is encountered and quickly drops to 0.0 when noticeable collisions occur.

5.3.2 Golem Physics: Forces

Physics engines use forces generated by environmental circumstances to compute realistic motions and trajectories for dynamic objects. In our case, we want to declare a specific pose and extract the forces that need to act to maintain that pose. This sort of force recognition is primarily meant to be of use in sports analysis, but it may also serve to improve tracking results. An inter-frame evaluator may be able to spot and prevent unrealistic movements during tracking.

Forces in Bullet Forces and torques (angular moment) in Bullet and in general are described as vectors. A linear force has a direction it acts in and a magnitude, which are the same basic attributes as those of vectors. The force responsible for rotational acceleration works similarly, but describes the turning of a certain mass around a pivot. The general term for an angular force is *torque*, and we use the two synonymously. A torque is also described as a vector. The direction of which describes the axis of rotation and its magnitude represents the torque's intensity.
In section 3.3, we described how Bullet computes forces and impulses on objects in order to integrate their resulting position. Searching for a way to access the forces calculated within Bullet, we found that they are hidden within the Bullet dynamics integration loop. For every rigid body in the simulation, both linear (affecting translation) and angular forces (affecting rotation) are calculated. This is redone in every subsequent step of the simulation, which means that Bullet clears out all force information upon finishing a cycle. If we try to access the forces acting on a single rigid body directly by invoking the implemented functions *getLinearForce()* or *getAngularForce()*, the returned value is always $(0, 0, 0)$, which describes a force (or torque) of 0 magnitude. Limited access into the Bullet dynamics loop is possible by defining a *Simulation Tick Callback* function. Algorithm 6 roughly outlines the temporal succession of events during the simulation loop. The callback can be called either before or after the main sequence. The callback function may contain any input that could otherwise be performed on the simulation in between step cycles. According to the official Bullet media-wiki, it is preferable to clear the forces inside the callback function when using a pre-tick callback, because at that point in the loop, *"Bullet might or might not have cleared forces and applied gravity. Clearing it again ensures that we are always in the same situation"* [2]. At the end of the line, we were not able to retrieve any forces or torques directly from Bullet. Although that is most unfortunate in itself, we did not really have need for them in the first place. The forces used by Bullet are those that are momentarily acting at a specific point in time. They are influenced not only by natural gravitation but also by the impulses used to solve constraints and by collision resolutions. What we try to simulate, however, is the force necessary for an individual to maintain a singular pose.

Algorithm 6 Simulation Tick Loop

```
1: manage sub stepping into clamped time steps
2: if Internal Pre Tick Callback then
3:     execute callback function
4: end if
5: predict unconstraint motion
6: perform collision detection
7: solve constraints
8: integrate transformations
9: update actions
10: if Internal Tick Callback then
11:     execute callback function
12: end if
13: synchronize motion states
14: clear forces
```

Forces Handle We decided to implement a separate force computation based on a sampling method. For this matter we implemented a *Forces Handle* to control force calculations for all Bullet Ragdolls in the simulation. The handle includes a list of all rigid bodies pertaining to the dynamics world, defines the number of maximum samples to be taken and stores the two lists of samples themselves. One contains the origin position, the other the rotation for every rigid body. Most importantly though, the handle provides functions to calculate linear as well as angular velocities, accelerations and forces. So instead of simply grabbing the force attributes from the simulation, we sample the future rigid body transformations. The basic idea is to let the simulation run for several steps and take samples of how the bodies move. We can subsequently guess the forces at work. After samples have been collected, the simulation is reset to the initial state. On this account, every rigid body's motion state, position, rotation and both linear and angular velocities are temporarily stored. The simulation is then stepped repeatedly with a fixed step size, until sufficient samples are collected. We used a relatively large step with 0.02 seconds, performing 10 sampling steps on each call. The sampling procedure is the same for all bodies. The body's center of mass position is directly on hand as vector. As rotational alignment, we use the body's rotation matrix and pick only the middle column vector. This represents the direction vector for the body's Y-axis, which is typically aligned with the model's original bone structure. These pieces of information, together with body masses, inertia tensors and time stamps, provide enough data to guess the forces required to maintain a specific body pose.

Derive linear forces Linear force can be described as the force that pulls or pushes a body in a single direction, acting on its center of mass. The object orientation or turning throughout the motion are irrelevant to determine the linear force applied. This does however not mean that the force is constant throughout the time we take samples. For instance, a body may be held in place by a constant force applied to it, or by applying repeated impulses of greater force. Think of a wheel being spun repeatedly to maintain its speed instead of being turned at a constant rate.
By fitting our series of samples to a model function (equation 4), we basically interpret whatever short bursts of force are acting during the sampled time into a steady force estimate.

$$\begin{aligned} \dot{p_t} &= \vec{v} = \vec{a}t + \vec{v_0} \\ \Rightarrow p_t &= p_0 + \vec{v_0}t + \frac{1}{2}\vec{a}t^2 \end{aligned} \tag{4}$$

The model is nothing more than a standard parable describing the current body position sample p_t, the derivative of which is the object's velocity $\vec{v_t}$. $\vec{v_t}$ can in turn be seen as the product of time t and acceleration $\vec{a}_t$ plus the initial velocity v_0. We use a least squares approach to minimize the error of fitting the sampled rigid body to this model. Least squares fitting is a common tool to find parametrized functional representations of data point clouds. It is explained in full detail in the appendix (section 8). Given the independent variable t, we want to obtain the dependant variable p_t. The variables we need to solve to get the fitted function are the ones describing position, velocity and acceleration throughout the sampled time interval. We use a matrix based solving approach as described in [44], outlined in equation 5. Therein, a is the vector of variables, X is a matrix containing the data points for the independent variable and y is the vector with the dependent variable.

$$a = (X^T X)^{-1} X^T y \tag{5}$$

In order to solve the fitting for all the three axes of movement, we include all three axes in the dependant variable matrix, the contents of which are shown in equation 6. We consequently also exchange the vector y of the above equation with the matrix Y, providing a result in the form of a 3x3 matrix A. The columns of this matrix are the solutions for the X, Y and Z-axis, respectively.

$$Y = \begin{pmatrix} 1 & t & t^2 \\ .. & .. & .. \\ 1 & t & t^2 \end{pmatrix},\ X = \begin{pmatrix} p_{0,X} & p_{0,Y} & p_{0,Z} \\ p_{1,X} & p_{1,Y} & p_{1,Z} \\ .. & .. & .. \\ p_{n,X} & p_{n,Y} & p_{n,Z} \end{pmatrix} \tag{6}$$

Even though the acceleration values are sufficient to calculate acting forces, we wanted to include all dynamics information in the process. We discovered the velocities in this calculation to be slightly off, so we use a separate function to calculate these by simply dividing the linear distance travelled between the first and last time sample by the total time span (see equation 7).

$$\vec{v} = \frac{(p_n - p_0)}{n * \Delta t} \tag{7}$$

In a final step, the estimated forces are calculated using the bodies' respective weights in the classic Newtonian formula $F = m * \vec{a}$. Since forces, velocities and accelerations are generally described as vectors, it is very easy to visualize them as such. We simply draw a straight line from the respective body's center of mass in the direction the force acts in. The length of the line is determined by the magnitude of the force vector, but can easily be scaled in relation to the model dimensions, in order to be properly distinguishable.

Derive angular forces The samples of the bodies' orientation exist as vectors for the time being. These vectors do not provide the information necessary to calculate torques acting on the body. It should be clear that like linear movements, rotations need not to be constant in either axis or speed. Again, we need to simplify the motion in order to calculate the angular speed and torque for rigid bodies. We assume that most body segment rotations would take place roughly in one plane, even though in reality, rotation axes are naturally loose. The way to picture this is comparing two ways of raising an

outstretched arm. One way is to raise it in a straight line, moving it from the lowest to its highest point without detours. Another way is to describe a wavy line with the arm, but ending the motion in the same topmost position. This kind of motion entails a transformation of the rotation plane throughout the motion. The first step in processing the taken samples is hence to fit them onto an appropriate rotation plane.

The easiest way for us to describe a plane in 3D space is to finds a corresponding normal vector. To find the normal for the rotation of a specific rigid body, we first create a matrix from each sample as shown in equation 8. For every sample p, $\vec{n} = \vec{p_1} \times \vec{p_i}$ describes the normal vector on the plane of rotation. We again want to minimize the error of fitting a single normal vector through all the samples' normals. In other words, we want to find the solution for the derivatives of the normal function described above, resulting in the rather long term of the second line in equation 8. The parameter $p_{i,t}$ stands for the three-dimensional vector $(p_{i,x}, p_{i,y}, p_{i,z})^T$. We can therefore transform the term to a matrix equal to $p_i\, p_i^T$.

$$
\begin{aligned}
\frac{\partial}{\partial n_t} &= \sum_{i=1}^{n} (\vec{n} \circ \vec{x_i i}) \vec{x}_{i,t} \stackrel{!}{=} min \\
\Rightarrow n_x \sum_{i=1}^{n} p_{i,x} p_{i,t} + n_y \sum_{i=1}^{n} p_{i,y} p_{i,t} &+ n_z \sum_{i=1}^{n} p_{i,z} p_{i,t} \stackrel{!}{=} min \\
\Rightarrow min \stackrel{!}{=} \vec{n} &\begin{pmatrix} \sum_{i=1}^{n} p_{i,x}^2 & \sum_{i=1}^{n} p_{i,y} p_{i,x} & \sum_{i=1}^{n} p_{i,z} p_{i,x} \\ \sum_{i=1}^{n} p_{i,x} p_{i,y} & \sum_{i=1}^{n} p_{i,y}^2 & \sum_{i=1}^{n} p_{i,z} p_{i,y} \\ \sum_{i=1}^{n} p_{i,x} p_{i,z} & \sum_{i=1}^{n} p_{i,y} p_{i,z} & \sum_{i=1}^{n} p_{i,z}^2 \end{pmatrix} \\
\Rightarrow \vec{n} &\left(\sum_{i=1}^{n} \vec{p_i} \vec{p_i}^t \right) \stackrel{!}{=} min
\end{aligned}
\tag{8}
$$

We proceed by creating the matrix $A_i = \vec{p_i} \vec{p_i}^t$ for every sample and summing them up to a single matrix A. The solution to the above equation is then the eigenvector of A with the lowest corresponding eigenvalue.
Once the general plane of rotation has been determined, we need to adapt our samples accordingly. By reducing the motion to a single axis of rotation, we would create measurement errors if we did not also reduce our samples to that dimension. Think of it like looking at the whole motion "from the side", all the while taking *new* measurements of the rotation in our samples. The samples simply need to be projected onto the plane of rotation and the angle in between measured. We simply use the first sample x_1 and the last sample p_n for every body, and compute the arc between their projections as in equation 9.

$$
\phi = \arccos \left(\frac{p_n \circ p_1}{(|p_1|\ |p_n|} \right) \tag{9}
$$

This way, we acquire the actual rotation angle for the fitted axis. For the sake of completeness, we actually calculate the angles for all samples in between the first and final one, too. Nevertheless, it is only the full angle that is used in the subsequent steps.

We already explained how angular velocity, acceleration and force can all be described as vectors. We already have calculated the directional component for all of them by finding a fitting rotation axis $\vec{n}$. We now want to determine the scalar magnitude of both the velocity ω and acceleration α. The product of scalars and axis finally leads to the angular velocity vector $\vec{\omega}$ and angular acceleration vector $\vec{\alpha}$. Similarly to our estimation of linear velocities, we use the difference between the latest and first sampled angles $\phi_n - \phi_1$ and divide by the total time span t, in accord with equation 10, this yields the angular

velocity for the body.

$$\vec{\omega} = \frac{\Delta\phi}{\Delta t}\vec{n} \tag{10}$$

For the calculation of angular accelerations, the same principles we discussed for linear accelerations apply. We want to fit the sampled angels and axes to a consistent model. The basic formulas to calculate angular motion are very much the same as for linear motion, hence equation 11 differs from the linear model only in nomenclature. The body's rotation $\vec{r}$ is our dependant variable here, time t is again the independent variable. Angular velocity ω and angular acceleration α are the parameters we're interested in.

$$\vec{r_t} = \vec{r_0} + \vec{\omega_0}t + \frac{1}{2}\vec{\alpha_t}t^2 \tag{11}$$

Like for the linear fitting, the result is a matrix containing parameters $\vec{r}_0$, ω and α for X, Y and Z-axes, respectively. Luckily, we do not have to determine the inertia tensor for the body. As described previously, the inertia tensor (in 3-dimensional space) is a 3x3 matrix describing the resistance of a rigid body to rotation around each of its axes. Bullet already stores the inertia tensor for every rigid body in the scene that is assigned a convex collision shape, by which we mean convex primitive shapes, too. Bullet however stores the tensor in inverted form, since it generally performs all the equations the other way around, integrating positions based on acting forces. The general equation to calculate the torque acting on a body is $\tau = \frac{\Delta L}{\Delta t}$, where t is time and L is the body's *angular momentum*. Angular momentum is in turn composed as $L = I\vec{\omega}$. So the final torque formula we follow is the one described in equation 12 below.

$$\tau = I\frac{\Delta\vec{\omega}}{\Delta t} = I\vec{\alpha} \tag{12}$$

The easiest form of visualization for angular velocities and torques is to draw the respective vector as a straight line again. The length of the line represents the size of the value, while the direction describes the rotation axis according to the right hand grip rule.

Logging So far we have taken samples, reset the simulation and calculated linear and angular forces for all ragdolls. As a result, a *Bone Dynamics Info* (BDI) object is created that contains all available data in a Bullet-independent format, so it can be handed to entities outside the Physics library. The BDI contains the bone ID, its mass, position and orientation, the linear velocity, acceleration and force as well as torque in vector representations and lastly the rotation axis and the scalar magnitude of angular velocity and acceleration. Additionally, all BDIs of related bones are wrapped in a ragdoll-specific *Model Dynamics Info* (MDI) structure, along with the ragdoll's model ID.

The Forces Handle is also responsible for logging purposes. The MDIs are essentially written to a tab-separated-value file in a machine-readable format. The data set is completed by adding a time stamp. Results are logged at the end of each dynamics calculation cycle, and the clock is only reset on restart or by manual reset. We used the dynamics log to evaluate our dynamics calculation. The full process is described below.

6 Evaluation

In order to test the quality of our new physics engine interface, we performed a series of tests within our modified physics demo. The primary goal for these tests was to verify that force calculation and collision detection within the interface are performed correctly.

6.1 Force Calculation

Since we do not extract forces directly from Bullet, but derive them from body movements, we wanted to make sure that our calculation bear accurate results. Three different test cases have been introduced to the Physics Demo application. Remember that the demo is nothing more than an OpenGL application that runs an instance of our physics interface and renders the Bullet Physics simulation. It therefore provides ways to add new ragdolls, access their data and manipulate them through the interface. Depending on the test case selected, the demo loads a ragdoll from a certain directory and applies the actions and transformations specific to that case.
The test cases were designed to produce easily readable logging output and being open to scrutiny. This means they are easily checked for actual physical accuracy by applying physics formulas and common sense. Case A describes a free fall, where the ragdoll is spawned with an offset high above the ground. Case B spawns the ragdoll in mid-air, but gives it a strong impulse to produce an arcing trajectory. Case C loads a custom, single-body ragdoll, which is given a spinning impulse to determine rotational forces.

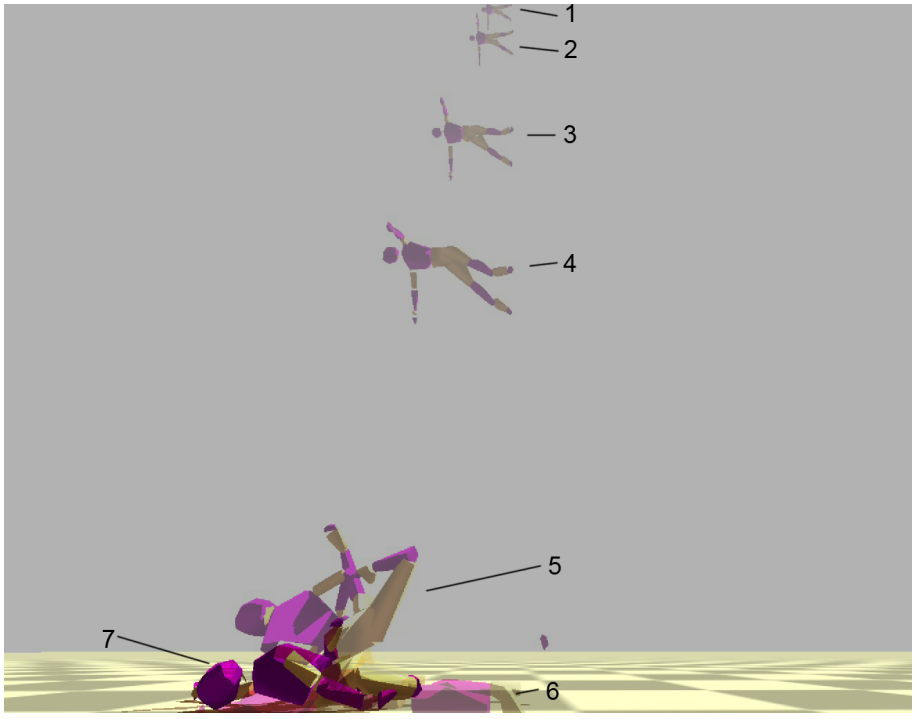


Figure 6.1: Frame overlay of test case A: Free Fall. The simulation begins with the ragdoll in 50m height (1) and ends as soon as it reaches a resting state (7).

Test Case A: Free Fall This very simple test involves a human body ragdoll as described in section 5.2. It consists of 20 different rigid bodies connected by natural joints. The size of the ragdoll has been scaled up to exactly two meters. It may be noteworthy to say that neither Blender (the export editor) nor Bullet manage dimensions as metrical

units. We therefore assume that *1 Blender-unit = 1 Bullet-unit = 1 meter*. Otherwise masses and joint components are unaltered from the model presented in section 5.2. Like in the regular physics demo application, a static ground plane is present.
Gravity acceleration in the dynamics world is set to $(0, -9.81, 0)$, which represents a constant "downwards" acceleration, similar to earth's gravity. Upon initializing the dynamic simulation, one ragdoll is spawned with an initial offset of 50 meters above the ground plane (i.e. an offset of (0, 50, 0) in global coordinate space), and without any initial velocities or momentum. The simulation runs until the ragdoll hits the floor and reaches a state of rest.
In this first test, we are specifically interested in the ragdoll's linear velocity, linear acceleration and linear force. Angular movement does hardly exist in this scenario, except for very erratic values due to the rebounding forces upon hitting the ground. These are near impossible to investigate, however, and therefore not of any value in this test. Plotting all logging parameters would thus be silly, which is why we use a Python script to extract the parameters of interest. The script runs through the log file, exporting each parameter of interest for each single rigid body into one specific, easy to plot table. For the sake of completion, we also plotted the center of mass positions.

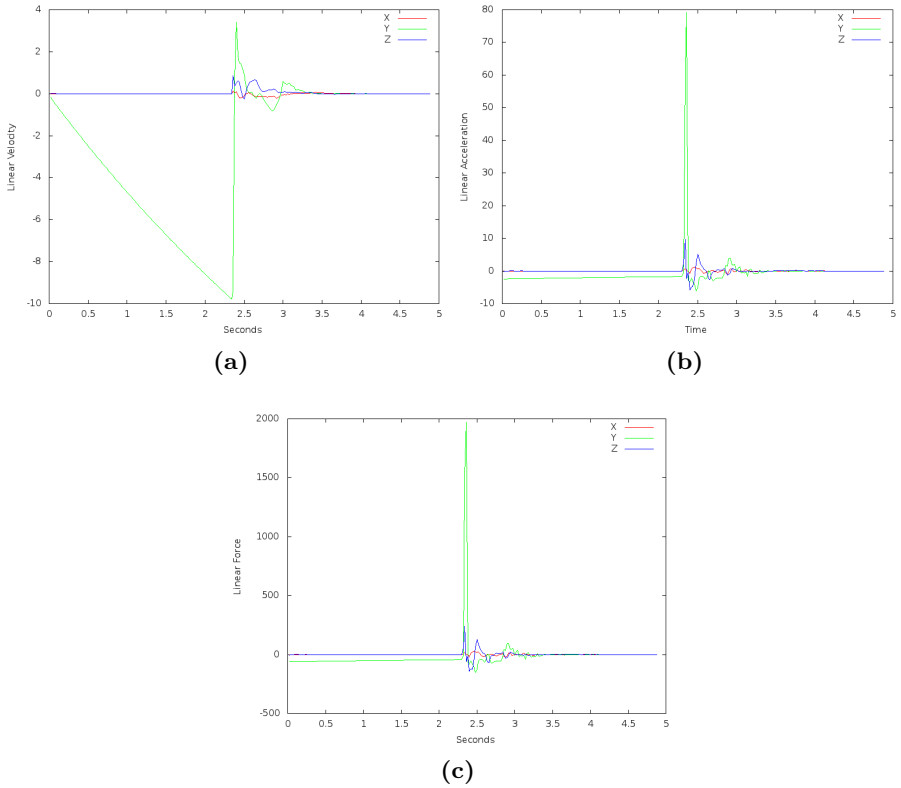


Figure 6.2: Test case one results. Linear velocity, linear acceleration and force of the chest segment with linear damping activated. The falloff in speed and acceleration is easily visible up to 2.3 seconds.

What one would expect to find in a free fall simulation are:
1. Constant acceleration with $a = (0, -9, 81, 0) = gravity$.
2. Constant force depending on rigid body mass, as described by $F = m\,a$.
3. A linear increase in velocity, due to the constant acceleration, provided by $v_t = a\,t$.
Likewise, after one second of free fall, velocity should equal acceleration: $v_{1s} = a$.

We encountered none of the above in our first test run. Of course we would not mention the bug in our time stamps that caused all the parameters to be off by factors two to four. With that inconvenience fixed, we still encountered acceleration and thus force values to drop in magnitude (see figures 6.2) instead of being constant throughout the few seconds of free fall. The change in velocity was similarly not linear, as can be seen in the velocity plot in figure 6.2. The reason behind this behaviour is that Bullet assigns every rigid body in the simulation a *damping* factor for both linear as well as angular motion. The higher the damping value, the quicker the falloff of linear and/or angular velocities. It is comparable to an omnipresent friction, that is constantly applied to the rigid body. Its purpose in the simulation is twofold: on one hand, it serves to simulate diverse behaviour of physical objects like general air drag or grinding, slow-moving objects, which would require excessively large mass properties. On the other hand, it helps to bring sliding or stacked objects to reach a static resting position more quickly, easing some effort off the simulation.
The latter is not a problem in our simulations, because we do not expect an abundance of free moving dynamic objects. Air drag simulation on the other hand may be seen as a realistic environmental circumstance and a desirable property of the simulation. However, Bullet does not calculate actual aerodynamic features or properties, but rather uses damping factors to influence velocities linearly. Given the very unlikely effects of damping factors on our initial test run, we preferred to insinuate an air-evacuated environment over configuring damping factors to borderline acceptable values. For the remainder of the tests, we deactivated damping by setting linear and angular damping factors to (0, 0, 0).

Results for the second test run without damping are exactly as we expected them to be during the free fall. Also expectable were the extremely high values for acceleration and force in the moment of impact with the ground object (figure 6.3 (c) and (d)). Upon reaching a state of rest, velocity, acceleration and force values fall off to zero, just as they should do.

Test Case B: Impulse For this test, we used the very same ragdoll and environment as in test case A. It was also performed without damping factors. This time the single ragdoll is spawned in mid-air, at an offset of (-20, 30, 0), meaning it starts at a lower height and off to one side of the simulation world center. Immediately after being introduced to the world, the ragdoll is given a strong impulse, so as to make it fly in a slight arc towards the central point of the simulation world. Bullet provides a function to apply a central impulse of variable intensity to any single rigid body in the simulation. We tried two methods to apply the impulse. First: Applying a central impulse to every one of the ragdoll's rigid bodies simultaneously, using relatively low impulse vectors of (40, 50, 0). And second: Applying a central impulse to the chest segment exclusively, using a very strong impulse vector of (400, 500, 0). Although the latter method is visually more coherent, results were very much alike in both variations (see figure 6.4). The ragdoll is thrown a short distance through the air, describing an arching trajectory before hitting the ground and sliding to a halt. As in the previous test case, we limit the output to linear velocity, linear acceleration and linear force.

Overall, we expected the impulse to cause a very strong acceleration (hence force as well), which will diminish quickly and eventually return to the gravitational acceleration. This was indeed the case as shown in figure 6.5 (g) and (h). The velocity was expected to be constant in x-direction and falling linearly in y-direction. Figure 6.5 (f) confirms this, and further shows the influence of friction on the the x-bound velocity after the flight has ended and turned into a sliding movement. We also expected the first few samples

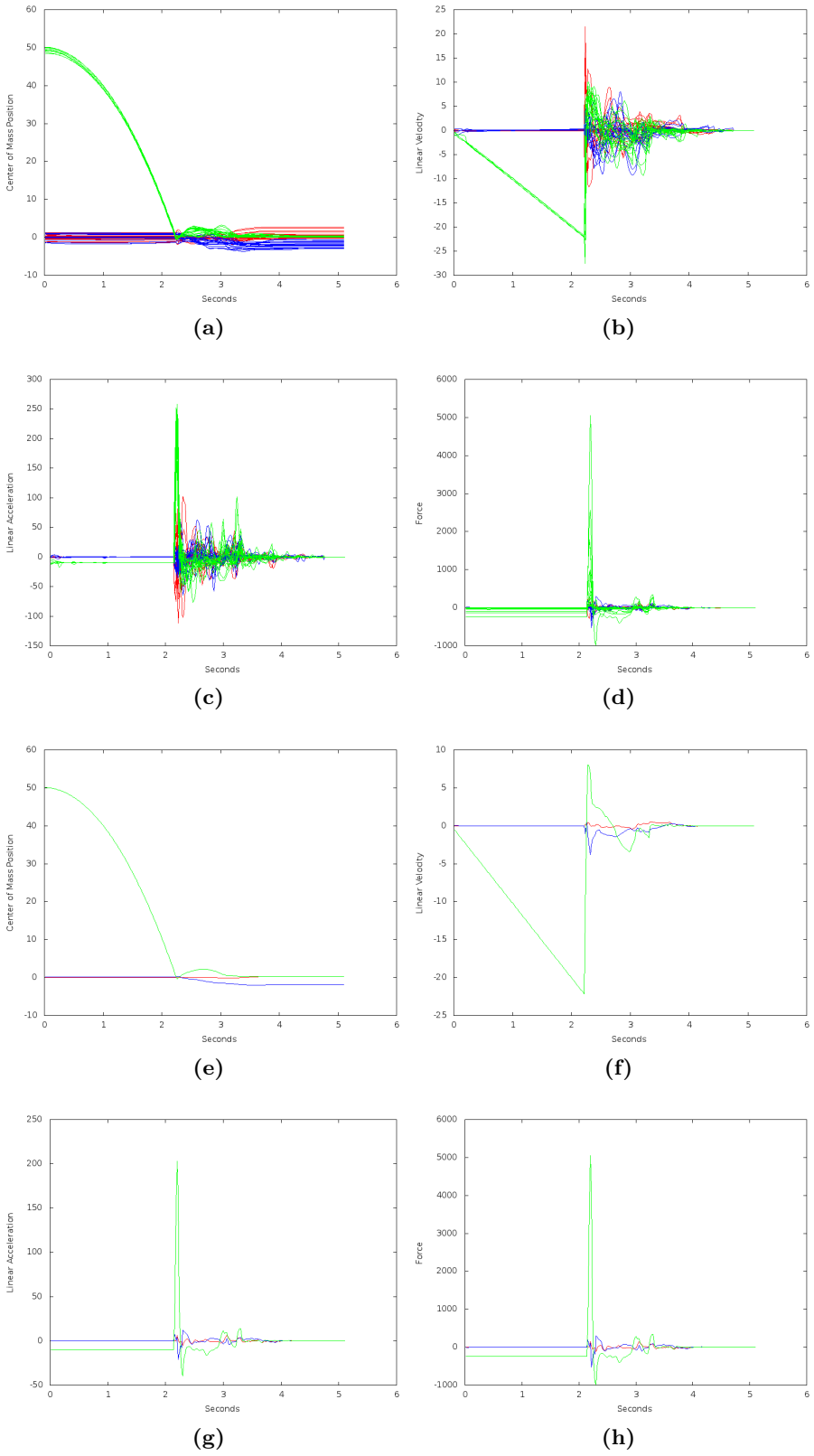
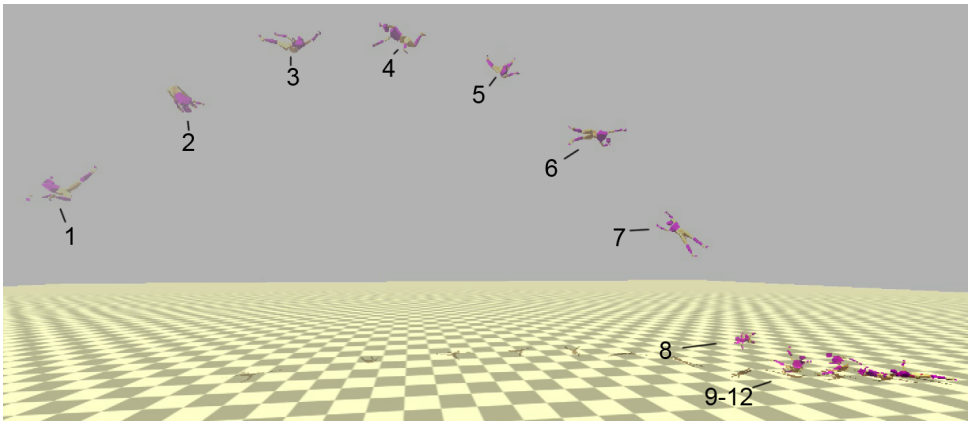
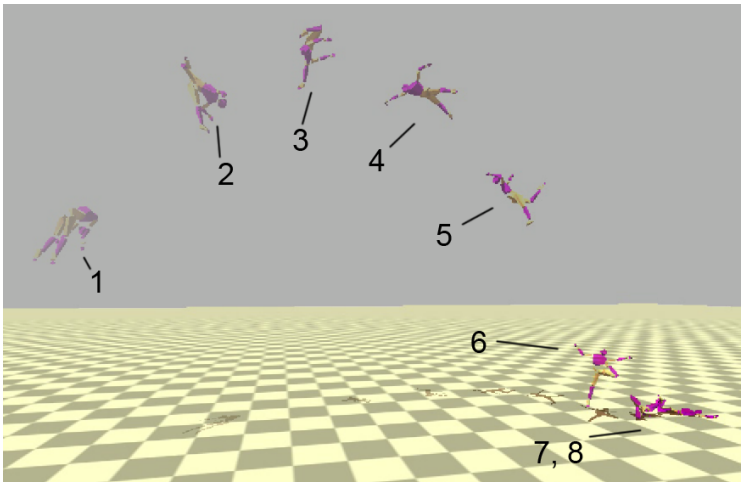


Figure 6.3: Test case A results. Position, linear velocity, linear acceleration and force with linear damping deactivated. Graphs (a) - (d) show all ragdoll segments simultaneously, (e) - (h) describe only chest segment.



(a)



(b)

Figure 6.4: Frame overlay of test case B: Impulse. The simulation begins with the ragdoll in 10m height (1) and is given a strong impulse that sends it flying along the x-axis(2-8; 2-6). Simulation ends as soon as it reaches a resting state (12; 8). The test was performed twice, first giving all ragdoll segments a simultaneous impulse (a), and again giving only a single, but stronger impulse to the chest segment (b).

to be disturbed by very erratic movements due to the application of the impulse being comparable to the ragdoll hitting the floor in test case A. The actual amplitude of these movements did in fact fall short of what we expected, which in itself is a compliment to Bullet's constraint solver.

Test Case C: Rotation As a final test, we want to evaluate the results of the angular force calculations. Angular forces are inherently harder to visualize and assess than linear forces. Especially in hierarchical models, the angular forces at work are very numerous and extremely diverse. In order to create a test scenario that is easily checked for correct angular movement results, we used a very simple, capsule-shaped model. This model, originally created for constraint tests, consists of two bones, enveloped in a capsule-like mesh. For the sake of this test, components were removed from the setup, so that the capsule is in fact a single rigid body with a mass of 4.0. Mind that the only reason for the two-bone armature is that our interface automatically creates triangle mesh shapes when given one-bone armatures. We wanted to avoid this due to the negative implications of triangle mesh shapes discussed both previously and in the upcoming limitations section.

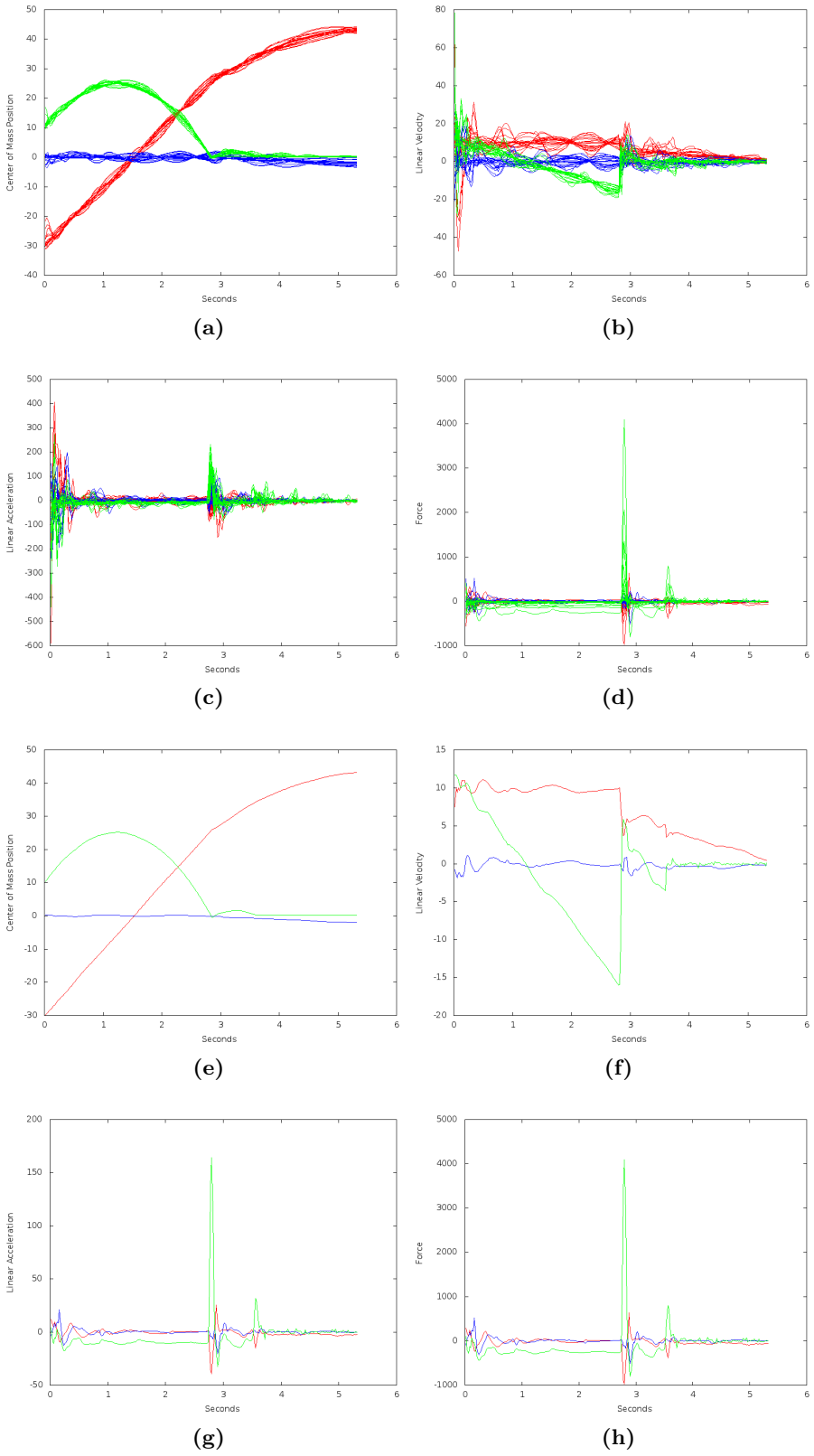


Figure 6.5: Test case B results. Position, linear velocity, linear acceleration and force with linear damping deactivated. Graphs (a) - (d) show all ragdoll segments simultaneously, (e) - (h) describe only chest segment.

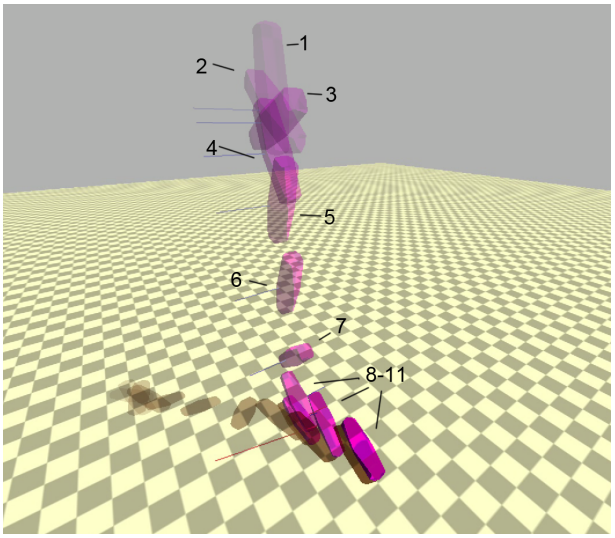


Figure 6.6: Frame overlay of test case C: Rotation. For this test, an inarticulate ragdoll was used. The simulation begins with the ragdoll in 10m height (1) and applies a rotational impulse to make the body rotate around its x-axis (2-7). Simulation ends as soon as it reaches a resting state (11).

Apart from the model, environmental properties are almost the same as in test cases A and B. We decided to introduce an angular damping factor of 0.075 for this test case, because the ground friction seems to stop linear movements as one would expect, but has little consequence for rotational motions. In this test case, the one-body ragdoll is again spawned in mid-air and immediately given a strong impulse. The impulse is an angular one, causing it to rotate rapidly around the axis the impulse is applied to. We use X as rotational axis with a magnitude of +20. The resulting motion is depicted in figure 6.6. For this test, we limited the relevant output to rotational axis, angular velocity magnitude, angular acceleration magnitude and torque (in vector form).

The first and most obvious observation we expected to make was that the rotational axis would be the X-axis, which is indeed the case up to the point when the model hits the floor and is deflected (figure 6.7(a)). Furthermore, we expected to see a very high rise in angular acceleration (and likewise torque) due to the impulse, with an immediate return to a constant negative acceleration resulting from damping. Angular velocity would thus be catapulted during the initial impulse and slowly decrease for as long as the model remains airborne. With the impact on the ground, the model's rotation should be stopped and deflected, resulting in a very messy, hard to assess compound of angular forces.
Our expectations were met by the results for the most part. The assessment of the rotational axis is easy enough for the first few iterations, but becomes problematic once the body starts to roll around on the floor. It can however easily be checked in the actual demo, where it is rendered as a red line along the rotation axis and looks to be sensibly related to the rotation. The initial torque spike we expected is omitted due to the fact that the sampling only starts *after* the impulse has already been applied. The graphs have to be interpreted accordingly. The remaining assumptions were satisfied and the expected torque spike upon hitting the ground is easily identified at roughly 1.5 seconds. The object also fell in a convenient way so that the velocity plot 6.7 (b) shows how the capsule began swaying from one side to the other after hitting the floor.

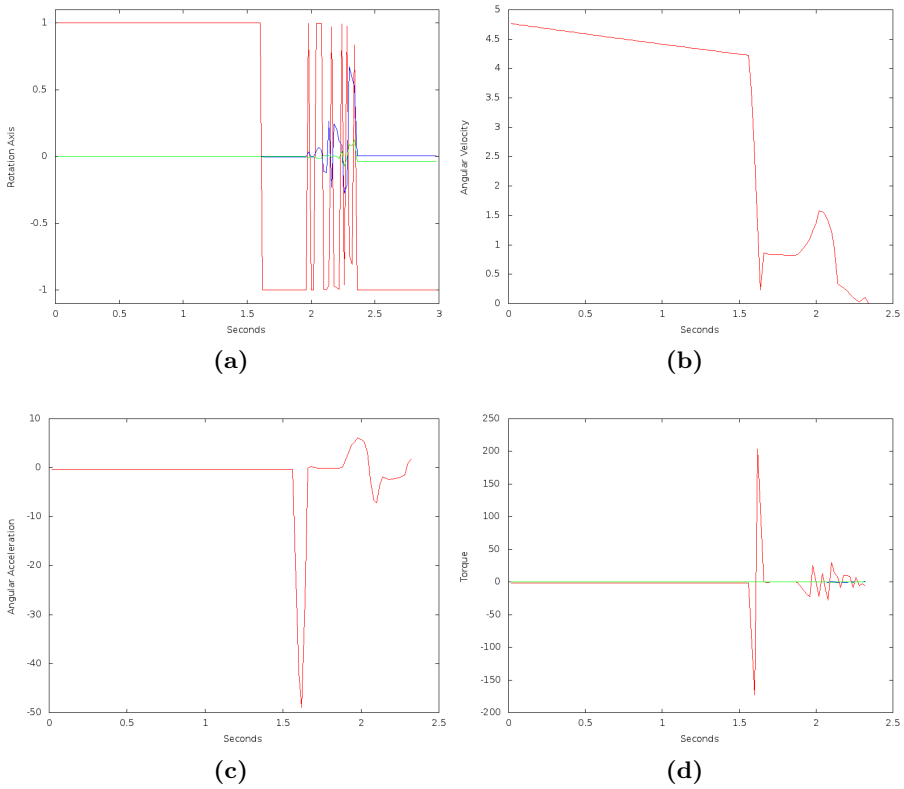
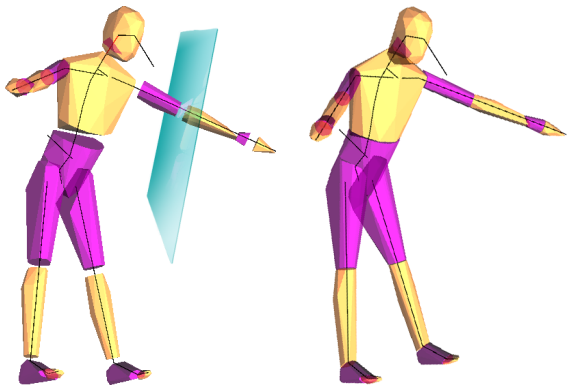


Figure 6.7: Test case C results. Rotation axis, angular velocity magnitude, angular acceleration magnitude and torque vector with angular damping activated.

6.2 Collision Evaluation

Collision Evaluator Test The collision evaluator can be tested fairly well in the tracker's GUI. It allows selection and movement of individual structure parts while applying forward kinematics. Simultaneously, the ragdoll inside the physics engine is adjusted to match the resulting pose. All at the same time, the Physics Collision Evaluator calculates a collision score for the given pose in real time.
We already used this method while adjusting the evaluation algorithm in section 5.3.1. Using the same human model already described, we made sure the collision evaluation algorithm worked as intended for self-collisions by posing the model accordingly. Additionally, we tested inter-model collision detection with the human ragdoll and a simple plane shape. As far as collision detection is concerned, inter-model collisions work just like self-collisions do, because Bullet does not actually keep track of rigid body affiliations. Unfortunately, there are issues occurring in this special case, which produce erroneous collision scores. As can be seen in figure 6.8 (a), there are gaps in between the ragdoll's constrained segments (i.e. the elbows, knees, wrists and even between thorax and abdomen). If another object face runs straight through one of these gaps without their surfaces touching, Bullet's narrowphase collision detection will not register a collision. Hence the pose in figure 6.8 (a) gives a perfect score, even though the person's arm would in fact be projecting through a solid surface. This is also the case if the rest of the arm were stuck *completely inside* another object (for instance if we used a large cube instead of a plane).
A workaround for this problem might be to increase the model's number of vertices. Since in our Bullet Ragdoll we do not allow the inclusion of the same vertex in two different segments, an increased number of vertices would minimize the gap size between the seg-

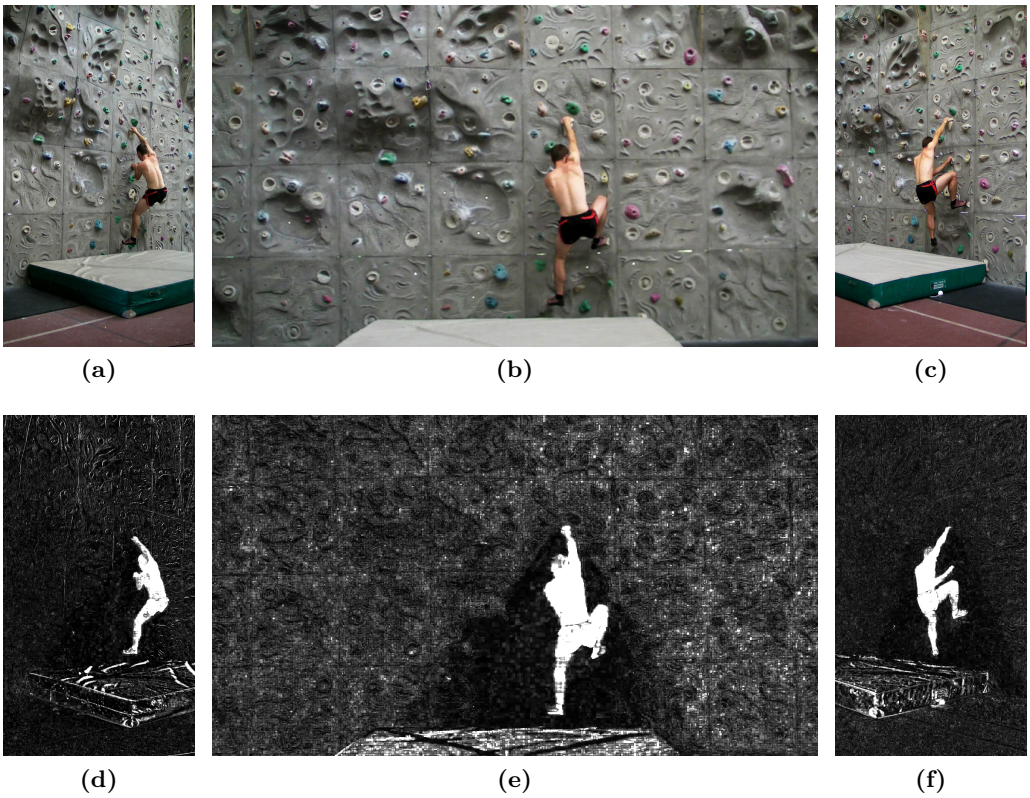


(a) (b)

Figure 6.8: (a) Low resolution ragdoll with 864 vertices. Gaps in between segments are very prominent because edges between the respective vertices are lost in translation. The semi-transparent plane object fits in between upper and lower arm segments, and is unnoticed by collision detection. (b) High resolution ragdoll with 20589 vertices. Gaps are not visible due to the lost edges being extremely small in size.

ments. This is illustrated in figure 6.8 (b). The smaller gaps would make it harder to find a pose where another object fits *exactly* between two segments. This would of course be at the cost of higher computational effort. At this point, we cannot exactly asses how much this would weigh in on performance.

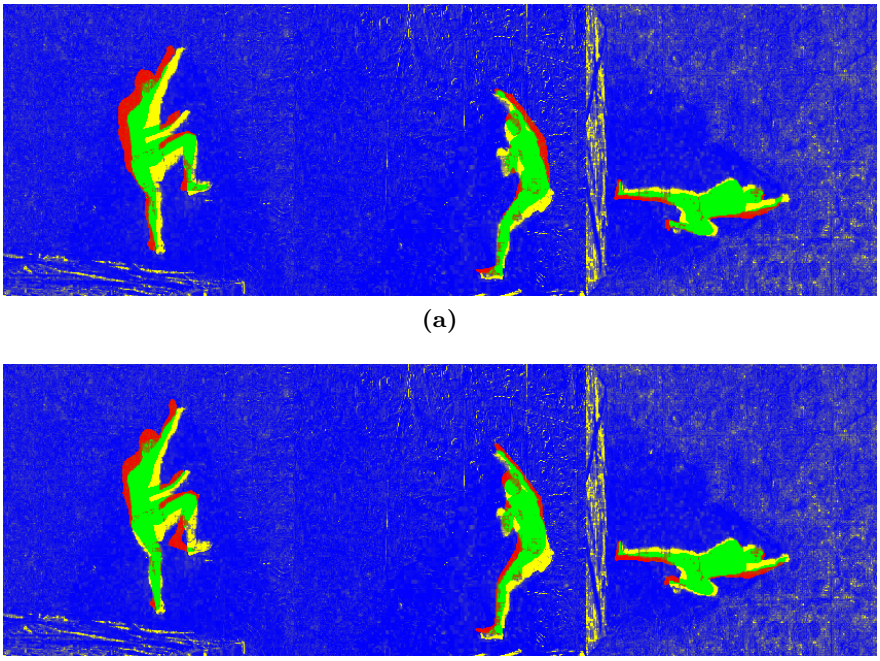
Application We would like to demonstrate the benefits of the physics collision evaluator on a first hand example. Below are two separate frames (frame A figure 6.9, frame B figure 6.12) from a scene of a climber. Three cameras simultaneously recorded the scene from viewpoints spread in roughly 45 degree intervals in front of the climbing-wall.
Apart from the human model, the wall is also included within the 3D-scene. For the model state space, it is ”invisible” when determining silhouette scores. This means it is rendered as decoration and does not actually influence the traditional tracking process in any way, which caused significant trouble in respect to mesh intersections. Hands and feet of the tracked person climbing the wall would repeatedly penetrate the wall model, disappearing within it. Inside the Bullet dynamics world, on the other hand, the wall is included as a static collision object, allowing us to determine collisions between the tracked person and the wall.
Figure 6.9 shows all three views of the demonstration frame as well as the corresponding silhouette descriptors. These silhouettes are the basis for the visual evaluation system in place, assigning each pixel either as *foreground* (white) or *background* (black). Each camera angle is transferred to the 3D-world and is used to create another two-bit silhouette of the 3D-scene from that viewpoint. Remember that the wall model is only "decoration”, and hence not visible in these silhouettes, so the only foreground seen is the model’s silhouette. The foreground/background match between original and the 3D-generated silhouette is then the score for that view. A combined score can be calculated from the views of all three cameras. Within figure 6.10 the match-up is shown in four bit images. Pixels are given one of four possible colors, depending on the result of the match. Blue indicates that both the original and the 3D-silhouettes contain a matching background pixel. Green is analogue for matching foreground pixels (i.e. the 3D-model is in a correct position). Red pixels are expected to be background but the 3D-silhouette contains foreground at that



(a) (b) (c)

(d) (e) (f)

Figure 6.9: Frame A. The tracked person is beginning to climb the wall. Pictures (a), (b) and (c) show the synchronized video frames as taken by the three cameras. Pictures (d), (e) and (f) are the silhouette image descriptors.

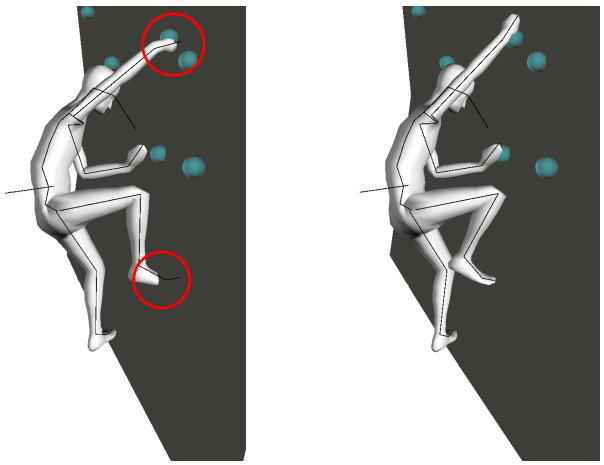


(a)

(b)

Figure 6.10: Silhouettes for poses a and b respective frame A. The images are green where the 3D model is correctly aligned with the person in the original video frame and red where the model is incorrectly posed. For both poses, silhouettes look nearly the same and visual scores are very similar.

pixel. Similarly, yellow pixels are where the foreground is expected but not encountered (i.e. 3D-model should be here). So for a perfect score, no red or yellow pixels would show. In general, a 90% score is already very well and hard to surpass. This is in part due to the 3D-model not fitting the actor perfectly, so it may not even be able to fill the same screen space as the original silhouette. The other reason is that the virtual camera coordinates are not exact. Both are subject to errors, and these errors accumulate and transfer into the final score as well.



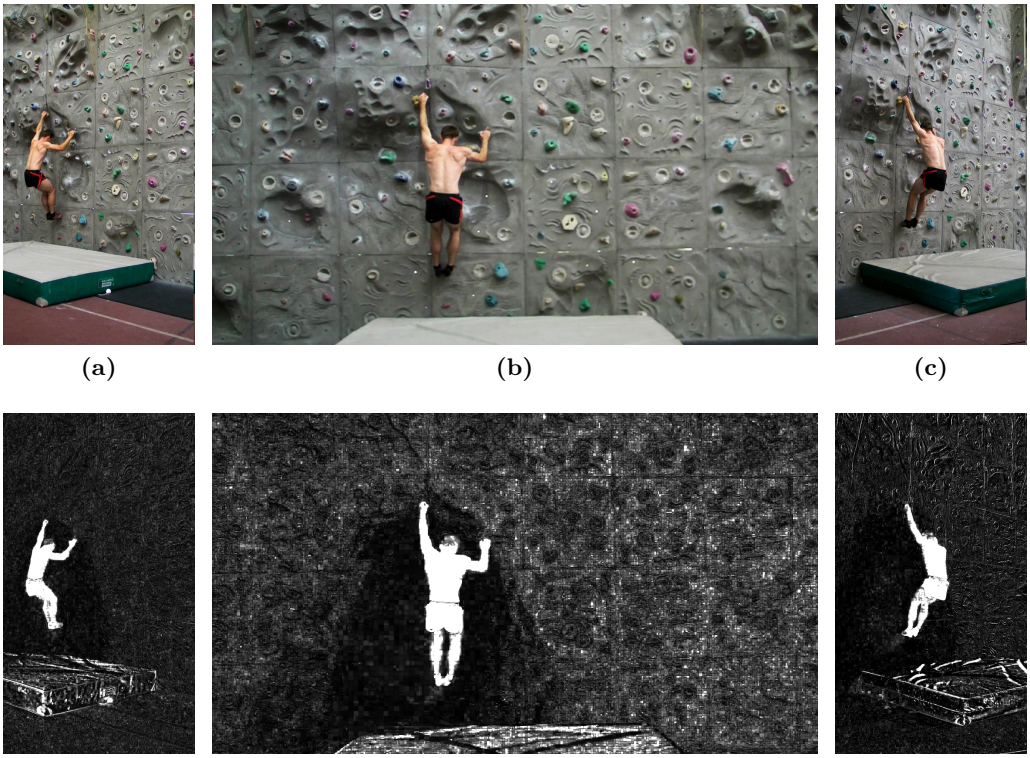
(a) (b)

Figure 6.11: Poses *a* and *b* as OpenGL rendering. Pose *a* contains visible mesh intersections, emphasized by red circles. Pose *b* is collision-free. The cyan balls are purely visual representations of the climbing handles on the wall.

After outlining the visual evaluator, we want to take a look at figure 6.11 (a) and (b). These are two possible solutions for the tracking in this frame. Both have very similar visual scores: (0.855157, 0.892069, 0.803696) and (0.864847, 0.890351, 0.80268), respectively, amounting to a very small 0.5% total difference. The first one, however, looks much more unrealistic, because the hand and foot of the person are projecting through the wall. Our physics collision score for this hypthesis is flat 0. The second, visually equivalent solution on the other hand has a near perfect collision score of 0.94415.

While these issues generally hold true to any scene involving object interaction, another reoccurring problem is the incorrect evaluation of physically impossible poses within articulated models. Figure 6.12 shows a relatively simple pose, where the climber puts plants both his feet on one small ledge. In figure 6.14, two different poses are again compared to the source image. Seeing that the man's legs intersect on another, the first pose is clearly completely wrong and is penalized with a flat 0 score by the collision evaluator. The second pose has neither self-collisions within the model hierarchy nor collisions with the wall, and hence scores a perfect 1. Nevertheless, visual scores for both hypotheses are close to equivalent with (0.852, 0.884594, 0.803173) for the unrealistic pose and (0.853906, 0.879717, 0.803249) for the one without collision. Overall, the incorrect pose even has a slightly higher visual score than the more intuitive second pose. This exposes the inaccuracy of silhouette based evaluators when handling compact or occluded poses, as well as the strengths of combining a physics based model with the tracker system.

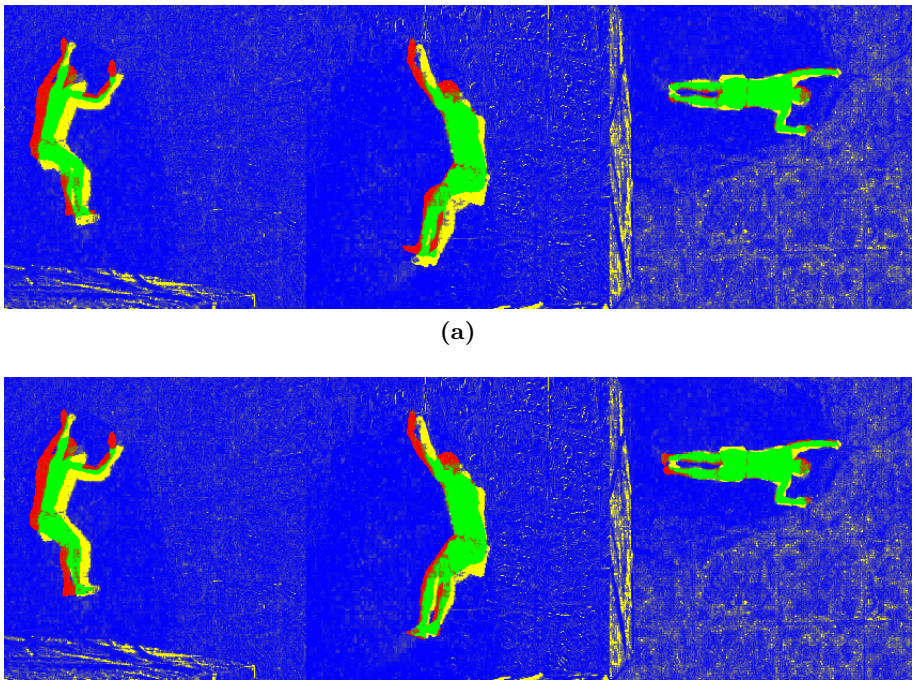
Object Manipulation The main point that restricts collision detection is a direct dependency between accuracy and the model's level of detail. Organic bodies have traits that are difficult to model in an efficient manner. The use of rigid body segmentation even



(a) **(b)** **(c)**

(d) **(e)** **(f)**

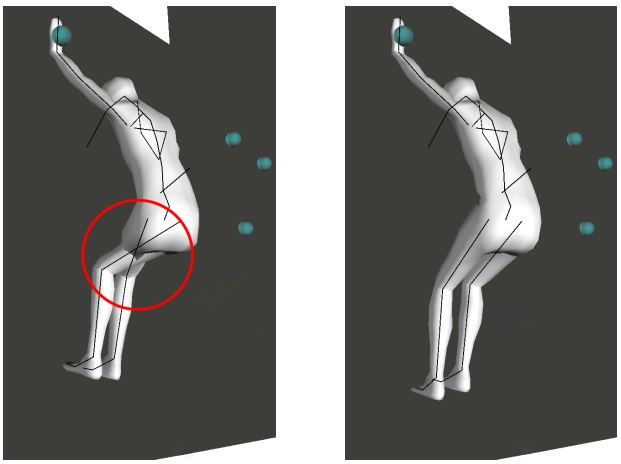
Figure 6.12: Frame B. The tracked person has placed both feet on a narrow ledge. Pictures (a), (b) and (c) show the synchronized video frames as taken by the three cameras. Pictures (d), (e) and (f) are the silhouette image descriptors.



(a)

(b)

Figure 6.13: Silhouette projections of poses a and b respective frame B. Again, the poses have almost equal visual score and silhoutes look almost identical. The rotation of the third image is due to the cameras' original tilt during recording.

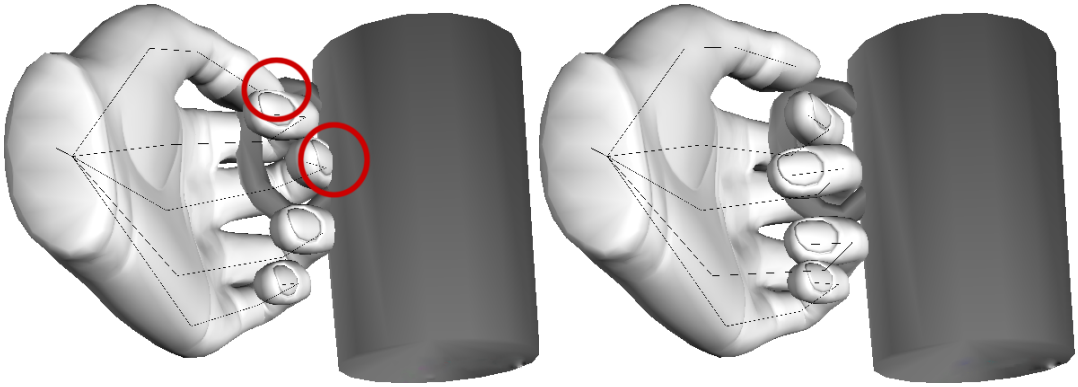


(a) (b)

Figure 6.14: 3D view of poses *a* and *b* for frame B. Pose *a* includes an extraordinary case of self collision. The model's legs are running through each other, swapping the feet's positions on the ledge. This goes unnoticed in the silhouette projections. Pose *b* is collision-free.

prohibits these traits. To name a few common ones: stretching and bulging of muscles, deformation by pressure or gravity and hard to track skeletal motion like shoulder movements. Human hands notoriously display all of these features, on top of at least 22 DOFs (depending on model simplification). Our hands allow extremely versatile object manipulation, often resulting in a plethora of contact points, diverse torques and deformation through strain and pressure.
While silhouette evaluation is more robust to these natural body deformations, our collision evaluation is very sensitive to them. The climbing task contains a good example in that our model does not contain individual fingers. Therefore, the exact grip on the wall cannot be recreated within the 3D-model, resulting in slightly awkward mitten-like hands barely touching the wall at all. Including the hands in full detail, however, would not be beneficial due to the small size of the reference silhouette.
For true object manipulation tasks, it is best to record the hands exclusively, using a fully articulated hand-only model. This implies the overall scale of the dynamics world is increased, which in turn means that all the issues we talked about earlier weigh in stronger. An example of such a detailed hand model manipulating a cup is presented in figure 6.15. Although lacking actual video footage, we decided to manually create two distinct poses for a hand holding a cup. One including mesh intersections between cup and hand and the other being physically correct. Silhouettes were generated with a similar virtual camera setup as in the climbing example (figure 6.16).

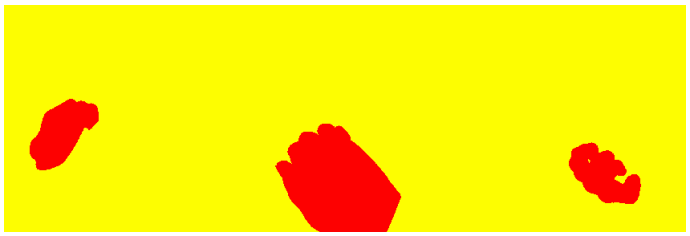
Given the actual *need* for contact points in order to hold the cup up, one would expect the two models to actually touch at some points. This would however result in a low physics collision score. One solution for this is to increase the collision tolerance within the evaluator, bearing the risk of missing invalid collisions. The pictures of silhouettes (figure 6.16) and renderings (figure 6.17) on the other hand demonstrate that the same limitations as before still apply for visual evaluation, because the silhouettes for correct and incorrect poses are almost equivalent.



(a) (b)

Figure 6.15: 3D rendering of the hand model holding a cup. Pose *a* contains some collisions, which even with full lighting and color information, are hard to recognize from this angle (thumb and middle finger do intersect with the cup). Pose *b* is (almost) collision-free.

(a)



(b)

Figure 6.16: Silhouette projections for poses *a* and *b* of the hand and cup model. The silhouettes show full errors due to the lack of a reference image. We still include them to demonstrate their likeness.

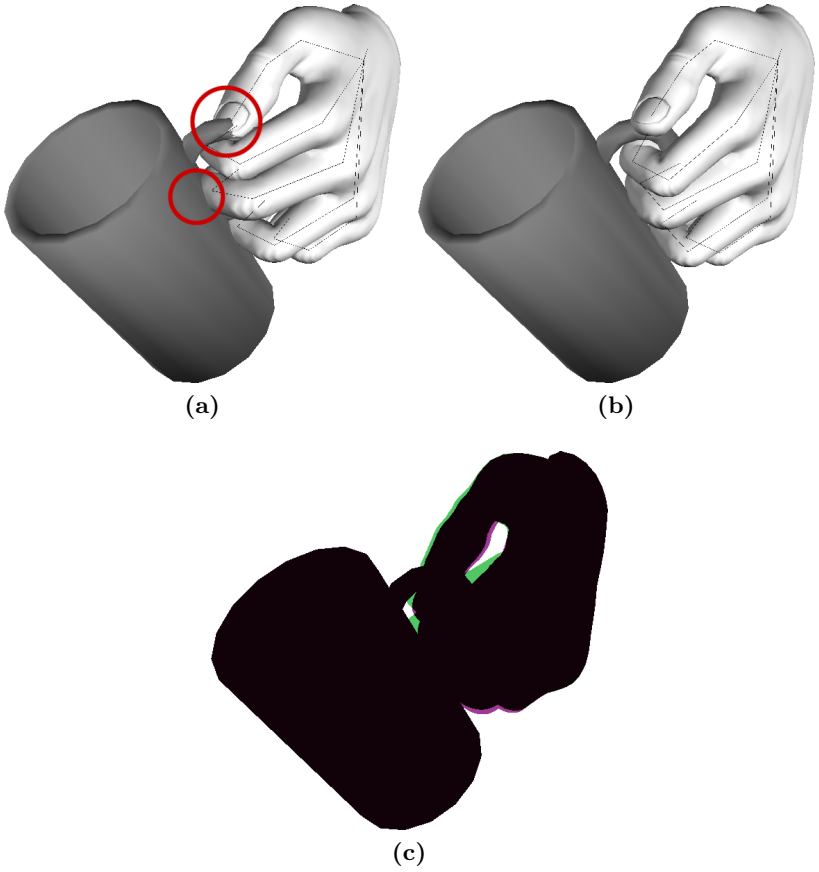


Figure 6.17: The same poses as before, but from a different angle. Here, the collisions in pose *a* can be determined more easily. Picture (c) shows a difference overlay of both poses' silhouettes. It demonstrates that they are still near identical.

7 Conclusion

As final section, we would like to recapitulate our efforts to incorporate a physics engine into a tracking framework, give a brief overview of the restrictions encountered in doing so, and lastly suggest some ideas how to proceed from here on in future research projects.

7.1 Summary

In this thesis, our aim was to improve a condensation-based human motion tracking system by using an open source physics engine as middle-ware. Common errors in model-based tracking are intersecting models and unrealistic, jerky movements. Recent research (e.g. [42, 43]) has shown that combining motion descriptors with physics based descriptors can significantly improve tracking results. Introducing a physics-based model representation as additional information layer allows for a dynamic simulation of the observed scene. This dynamic representation was implemented using the open source engine Bullet Physics Library. We have developed an interface that allows to compute collision data and provides force analysis features within the scene, using the new physics based layer.
A collision evaluator was created for the tracker, which computes a score for each hypothesis, based on the penetration depths of collisions within the scene. This allows use to detect otherwise unrecognised invalid hypotheses.
To determine the forces of dynamic objects within the scene, we us a sampling technique, letting the simulation run for several steps and resetting it afterwards. Velocities, accelerations and forces are then calculated by minimizing the error of a parabolic function, using a least squares fitting technique. The benefits are smoothed force results for both linear and angular forces.
Three different test scenarios were designed and performed to make sure our force calculations are sound. An Open-GL application was set up to produce visual and logging output for these test cases, which we evaluated individually based on general physical laws and common sense. The collision detection feature was tested within the tracker GUI on preselected, meaningful video frames, and proved to be conceptually sound in improving the avoidance of both self and foreign collisions.

7.2 Limitations

We would like to point out some unfortunate restrictions that come from using existing physics engines as middle-ware. Choosing a physics engine from the get-go, we had no prior experience with any of the potential candidates and have already mentioned some of Bullet's more problematic facets concerning our requirements.

Collision Shape Primitives The possibility to choose a preferred collision shape in our current model setup option is yet unstable. Like we mentioned several times throughout section 5.2, the main issue with Bullet's primitive collision shapes is their centric frame origin. This has to be accounted for in every transformation, meaning that affected methods always need two implementations, one for convex hull shapes and one for primitive shapes. This worked to some degree, up to a point where the number of sequential transformations grew confusingly high during constraint setup. Remember that inactive bones in the model's structure are not modelled independently in Bullet. Given for example a series of three inactive bones followed by a bone configured with a hinge constraint, there would be one rigid body representing all three lower bones and another rigid body bound to that compound by the constraint. The anchor point of the constraint would have to be specified relative to the root bone of the compound, including the offsets for the other two

bones. As easy as that may be, we encountered literally inexplicable gaps and misconfigurations between constrained body pairs. Disconcertingly, we could find neither reason nor remedy for these conditions, and therefore suggest to limit the collision shapes to convex hull shapes, especially if there are compounds of multiple sequential inactive bones in the model. The actual performance gain when using primitives as opposed to convex hull shapes is subject to further investigation. Nevertheless, we suppose it is not high enough to justify the trade-off between performance and loss of accuracy.

Dynamic Concave Meshes Both collision detection and force analysis can be useful for single-model environments, but they become especially interesting if there are more dynamic objects in the scene. For instance a human handling any sort of rigid object, be it a cup of coffee or a golf club, carries far more relevant physical information than the person alone. The main problem when introducing rigid objects into the Bullet world is the incapability of Bullet to reliably perform collision detection between multiple concave objects. Limiting the dynamic bodies to only convex objects or even their convex hulls is no solution, since it is actually very hard to find completely convex objects in the real world. Convex hulls on the other hand may omit critical item features. Try for instance stacking convex hulls of salad bowls - you may find it notoriously frustrating. On top of that, personal experience with compound shape objects in Bullet have been underwhelming in their believability. Compound shapes display "baloony" properties during simulations, having awkward centres of mass and inertia properties.
Modelling concave objects as packed groups of primitive shapes may be a solution to this issue, but requires to either manually set up any dynamic object in the scene as compound shape, or use a CPU-intensive convex decomposition algorithm like HADC. This method was introduced to Bullet in version 2.79 and works by breaking a triangle mesh down into numerous convex hulls. Bullet also offers an alternative convex-convex collision algorithm with *btGImpactCollisionAlgorithm* together with a triangle mesh shape type specific to that algorithm. Bullet's own demos as well as our testing however show that these algorithms are not reliable, either. Due to the disappointing results we would suggest to launch performance tests for the convex hull decomposition of concave objects, and closely observe the dynamic behaviour of the resulting compound shapes.

7.3 Future Research

As far as our own motives are concerned, we highly recommend to further investigate the impact of physics based simulation in augmentation of the present tracking system. A Forces Evaluator was planned to decrease the likelihood of random, jerky motion in between otherwise plausible trajectories. For this matter, force analysis capabilities of the Golem Physics Interface should be extended to inter-frame operability, taking into consideration the object's movement from frame to frame. In addition to this, the aforementioned close evaluation of Bullet's compound shape objects and convex decomposition features would be a complimentary effort.
Looking further ahead, it would be a good idea to create a thorough force analysis application for use with sports recordings and the like. Creating a framework that allows to load a tracked scene and play it back with controllable camera angles, force and torque visualization and custom activity scores for analytical reviews. This task may require the acting forces to be translated to human muscle strain, calling for a technique to distribute linear and angular forces to muscle groups in an ergonomically sound fashion.

8 Appendix: Least Squares Fitting

Generally speaking, the least squares approach is a technique to solve equation sets with more equations than unknown variables. Its main utilization is to fit a specific model function to a given set of data points. What this means is that, given a number of points y determined by values x, a function $f(x) = y$ may exist that approximately describes the set of data points. Provided a parameterisable model function (for instance a basic polynomial equation), the best fitting curve is found by minimizing the squared offset values between the points and the parametrized curve (residuals). The use of squared residuals instead of unaltered error values has two major implications. On one hand this guarantees that the residuals are continuously differentiable, on the other hand it amplifies the impact of outliers and observation errors on the fitted curve. In any case, this is the reason for the name *least squares fitting*.

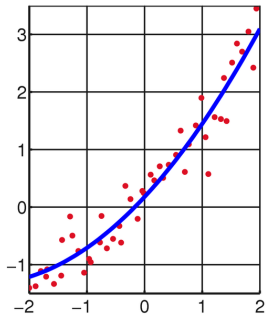


Figure 8.1: Least Square Fitting a data set to a parabolic model. The red dots are the observed data points, the blue curve shows the parametrized model that best fits the data. [3]

The most widely spread use for least squares is to minimize observation errors in statistical and experimental data. The observed entity is referred to as *dependent variable*, because it depends on one or more variables that are fixed by the experiment. These are chosen freely and are thus *independent variables*. Borrowing an example from our own force fitting model, the independent variable is the time at which a sample is taken and the dependent variable is the position of the object (the sample).
Having determined the independent variable x_i and the dependent variable y_i for a set of i samples, model function $f(x, \beta)$ needs to be found. Vector β contains a number of parameters for the function that may be varied freely in order to find a best fit. In our case, the model was a polynomial of second grade: $f_{poly}(x_i) = a + bx_i + \frac{1}{2}cx_i^2$, with $\beta = (a, b, c)$. Residuals are then simply calculated as the differences between the observed value y_i and the result of model function $f(x_i, \beta)$. The optimal fit is found by minimizing the squared residuals as shown in equation 13.

$$\begin{aligned} S &= \sum_{i=1}^{n} r_i^2 \stackrel{!}{=} min \\ r_i &= y_i - f(x_i, \beta) \end{aligned} \tag{13}$$

One way to do this is to follow a *gradient descent* approach. The gradient of a function may for simplicity's sake be considered the "inclination" towards a local maximum within the function. This means that for a point (x, y, z), the gradient $G(x, y, z)$ is a vector that points in the direction of the closest local maximum and its magnitude is a measure of how ”steep" the incline towards that maximum is.

In order to find a solution for the least squares fitting, the model's gradient must be set to zero. Every model has m number of gradient equations (see equation 14, [3]), where m is the number of model parameters.

$$\frac{\partial S}{\partial \beta_j} = 2\sum_{i=1}^{n} \frac{\partial r_i}{\partial \beta_j} = 0 \tag{14}$$

One easy way to minimize the gradient output is to choose any vector $\vec{v}$ and calculate the negative gradient output $-G(\vec{v})$ recursively, until the gradient reaches a minimum magnitude, which means it is very close to a local minimum. It may be that the resulting vector never reaches the desired level of accuracy, though.

An alternative way to solve the least squares fitting is presented in [44]. We want to limit our overview to the case of fitting a polynomial model, like the ones presented in section 5.3.2. A general polynomial term is used in equations 15 and 16, which can be directly used to generate a matrix X containing all n data points.

$$y = a_0 + a_1 x + ... + a_k x^k \tag{15}$$

$$\vec{y} = X\vec{a}$$

$$\begin{pmatrix} y_1 \\ y_2 \\ ... \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_1 & x_1^2 & ... & x_1^k \\ 1 & x_2 & x_2^2 & ... & x_2^k \\ ... & ... & ... & ... & ... \\ 1 & x_n & x_n^2 & ... & x_n^k \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ ... \\ a_k \end{pmatrix} \tag{16}$$

In general, the matrix X of equation 16 is over determined (i.e. there are more equations than variables, meaning X is not square and hence not invertible). This impedes the obvious solution by means of inverting X. We can however use basic matrix multiplication to get:

$$\begin{aligned} y &= Xa \\ X^T y &= X^T X a \\ a &= (X^T X)^{-1} X^T y \end{aligned} \tag{17}$$

This is the solution to the least squares problem. To make the connection, equation 18 calculates the squared residual values S (and derivatives thereof) for the polynomial model of grade k.

$$\begin{aligned} S &= \sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i + ... + a_k x_i^k)]^2 \\ \frac{\partial(S)}{\partial a_j} &= -2\sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i + ... + a_k x_i^k)] x^j;\ j = 0...k \end{aligned} \tag{18}$$

These can then be interpreted in matrix form:

$$\begin{pmatrix} \sum_{i=1}^n y_i \\ \sum_{i=1}^n x_i y_i \\ ... \\ \sum_{i=1}^n x_i^k y_i \end{pmatrix} = \begin{pmatrix} n & \sum_{i=1}^n x_i & ... & \sum_{i=1}^n x_i^k \\ \sum_{i=1}^n x_i & \sum_{i=1}^n x_i^2 & ... & \sum_{i=1}^n x_i^{k+1} \\ ... & ... & ... & ... \\ \sum_{i=1}^n x_i^k & \sum_{i=1}^n x_i^{k+1} & ... & \sum_{i=1}^n x_i^{2k} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ ... \\ a_k \end{pmatrix} \tag{19}$$

Equation 19 can then be written as $X^T y = (X^T X)a$, which is in turn solved by equation 17.

Contents of included CD

- Literature
  - papers
    - other
    - physics
    - tracking
  - websources
- Models
  - f360RomeoStates
  - f760RomeoStates
  - handcupRomeoStates
  - hand-centered.blend
  - ias-man-centered.blend
- PhysicsBasedTrackingDocument
  - images
  - bibfile.bib
  - Physics_Based_Tracking.tex
  - Physics_Based_Tracking.pdf
- Sourcecode
  - Golem
    - GolemPhysics
    - GolemPhysicsDemo
  - Romeo
    - Core_Evaluation_Collision
  - tools
    - gnuplotBatch
    - python
  - Romeo
- Videos
  - ContactSolverIterations50.ogv
  - ContactSolverIterationsDefault.ogv
  - FallingDolls.ogv
  - FallingDollsForces.ogv
  - FlyingDolls.ogv
  - FlyingDollsChest.ogv

References

[1] Anthropometry and mass distribution for human analogues, volume 1: Military male aviators. Tech. rep., Naval Biodynamics Laboratory, New Orleans, 1988.

[2] Bullet physics: Simulation tick callback. Restricted Wiki Article, 2010. Accessed 25.2.2012.

[3] Wikipedia: Least squares. Public Wiki Article, 2012. Accessed 25.2.2012.

[4] Wikipedia: Simulated annealing. Public Wiki Article, 2012. Accessed 26.2.2012.

[5] Aggarwal, J. K., and Cai, Q. Human motion analysis: A review. *Computer Vision and Image Understanding 73* (1999), 428–440.

[6] Albrecht, S., Ramirez-Amaro, K., Ruiz-Ugalde, F., Weikersdorfer, D., Leibold, M., Ulbrich, M., and Beetz, M. Imitating human reaching motions using physically inspired optimization principles. In *11th IEEE-RAS International Conference on Humanoid Robots* (Bled, Slovenia, October, 26–28 2011).

[7] Bandouch, J., Engstler, F., and Beetz, M. Accurate human motion capture using an ergonomics-based anthropometric human model. In *Proceedings of the Fifth International Conference on Articulated Motion and Deformable Objects (AMDO)* (2008).

[8] Boeing, A., and Bräunl, T. Evaluation of real-time physics simulation systems. *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia GRAPHITE 07 1*, 212 (2007), 281.

[9] Bregler, C. Motion capture technology for entertainment. *Signal Processing Magazine 24*, 6 (2007), 158 – 160.

[10] Carranza, J., Theobalt, C., Magnor, M. A., and Seidel, H.-P. Free-viewpoint video of human actors. In *ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), SIGGRAPH '03, ACM, pp. 569–577.

[11] Catto, E. Modeling and solving constraints.

[12] Cheng, F., Christmas, W., and Kittler, J. Recognising human running behaviour in sports video sequences. In *Pattern Recognition, 2002. Proceedings. 16th International Conference* (2002), pp. 1017–1020.

[13] Cheung, G. K. M., Baker, S., and Kanade, T. Shape-from-silhouette of articulated objects and its use for human body kinematics estimation and motion capture. In *Proceedings of the 2003 IEEE computer society conference on Computer vision and pattern recognition* (Washington, DC, USA, 2003), CVPR'03, IEEE Computer Society, pp. 77–84.

[14] Clauser, C. E., McConville, J. T., and Young, J. W. Weight, volume, and center of mass of segments of the human body. Tech. rep., Aerospace Medical Research Laboratory, NASA, 1969.

[15] Coumans, E. *Bullet 2.76 Physics SDK Manual*, 2010.

[16] Datta, V., Chang, A., Mackay, S., and Darzi, A. The relationship between motion analysis and surgical technical assessments. *Journal of the American College of Surgeons 184*, 1 (2002), 70–73.

[17] Datta, V., Mackay, S., Mandalia, M., and Darzi, A. The use of electromagnetic motion tracking analysis to objectively measure open surgical skill in the laboratory-based model1 1 no competing interests declared. *Journal of the American College of Surgeons 198*, 5 (2001), 479–485.

[18] Delaney, B. On the trail of the shadow woman: the mystery of motion capture. *Computer Graphics and Applications 18*, 5 (1998), 14 – 19.

[19] Deutscher, J., and Reid, I. Articulated body motion capture by stochastic search. *Int. J. Comput. Vision 61* (February 2005), 185–205.

[20] Drummond, T., and Cipolla, R. Real-time tracking of highly articulated structures in the presence of noisy measurements. In *In Proceedings of Eighth IEEE International Conference on Computer Vision* (2001), pp. 315 – 320.

[21] Erleben, K. *Stable, Robust, and Versatile Multibody Dynamics Animation.* PhD thesis, University of Copenhagen, 2004.

[22] Felzenszwalb, P. F., and Huttenlocher, D. P. Pictorial structures for object recognition. *Int. J. Comput. Vision 61* (January 2005), 55–79.

[23] G., J., and Richards. The measurement of human motion: A comparison of commercially available systems. *Human Movement Science 18*, 5 (1999), 589 – 602.

[24] Gavrila, D. M., and Davis, L. S. Tracking of humans in action: a 3-d model-based approach. In *In Proc. ARPA Image Understanding Workshop* (1996), pp. 737–746.

[25] Grauman, K., Shakhnarovich, G., and Darrell, T. A bayesian approach to image-based visual hull reconstruction. In *Proceedings of the 2003 IEEE computer society conference on Computer vision and pattern recognition* (Washington, DC, USA, 2003), CVPR'03, IEEE Computer Society, pp. 187–194.

[26] Grauman, K., Shakhnarovich, G., and Darrell, T. Inferring 3d structure with a statistical image-based shape model. In *In ICCV* (2003), pp. 641 – 647.

[27] Hamer, H., Gall, J., Weise, T., and Gool, L. V. An object-dependent hand pose prior from sparse training data. In *IEEE Conference on Computer Vision and Pattern Recognition* (June 2010), pp. 671–678.

[28] Hsiao, K., de Plinval-Salgues, H., and Miller, J. Particle filters and their applications. Online as part of Cognitive Robotics lecture, MIT, 2005.

[29] MacCormick, J., and Isard, M. Partitioned sampling, articulated objects, and interface-quality hand tracking. In *Proceedings of the 6th European Conference on Computer Vision-Part II* (London, UK, 2000), ECCV '00, Springer-Verlag, pp. 3–19.

[30] Marey, E. *Animal mechanism: a treatise on terrestrial and aërial locomotion.* International scientific series. D. Appleton, 1890.

[31] Moeslund, T. B., and Granum, E. A survey of computer vision-based human motion capture. *Computer Vision and Image Understanding 81*, 3 (2001), 231–268.

[32] Moeslund, T. B., Hilton, A., and Krüger, V. A survey of advances in vision-based human motion capture and analysis. *Comput. Vis. Image Underst. 104* (November 2006), 90–126.

[33] Mori, G., Ren, X., Efros, A. A., and Malik, J. Recovering human body configurations: Combining segmentation and recognition. In *In CVPR* (2004), pp. 326–333.

[34] Navaratnam, R., Thayananthan, A., Torr, P. H. S., and Cipolla, R. Hierarchical part-based human body pose estimation. In *BMVC* (2005), W. F. Clocksin, A. W. Fitzgibbon, and P. H. S. Torr, Eds., British Machine Vision Association.

[35] Paperno, E., Sasada, I., and Leonovich, E. A new method for magnetic position and orientation tracking. *Magnetics 37*, 4 (2001), 1938 – 1940.

[36] Peterson, S. Xbox kinect: One year later. Online News Article, 2011. Accessed 25.2.2012.

[37] Poppe, R. Vision-based human motion analysis: An overview. *Comput. Vis. Image Underst. 108* (October 2007), 4–18.

[38] Poppe, R. A survey on vision-based human action recognition. *Image Vision Comput. 28* (June 2010), 976–990.

[39] Raab, F., Blood, E., Steiner, T., and Jones, H. Magnetic position and orientation tracking system. *Aerospace and Electronic Systems 15*, 5 (1979), 709 – 718.

[40] Roh, M.-C., Christmas, B., Kittler, J., and Lee, S.-W. Robust player gesture spotting and recognition in low-resolution sports video. In *Proceedings of the 9th European conference on Computer Vision - Volume Part IV* (Berlin, Heidelberg, 2006), ECCV'06, Springer-Verlag, pp. 347–358.

[41] Sigal, L., Isard, M., Sigelman, B. H., and Black, M. J. Attractive people: Assembling loose-limbed models using non-parametric belief propagation. In *in NIPS* (2003), MIT Press, pp. 1539–1546.

[42] Wei, X., and Chai, J. Videomocap: modeling physically realistic human motion from monocular video sequences. *ACM Trans. Graph. 29* (July 2010), 1–10.

[43] Wei, X., Min, J., and Chai, J. Physically valid statistical models for human motion generation. *ACM Trans. Graph. 30* (May 2011), 1–10.

[44] Weisstein, E. Least squares fitting–polynomial. MathWorld–A Wolfram Web Resource. Accessed 26.2.2012.