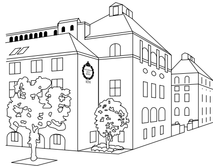
KTH Numerical Analysis
and Computer Science

Physically Based Character Simulation – Rag Doll Behaviour in Computer Games

Johan Gästrin

TRITA-NA-E04008



Numerisk analys och datalogi
KTH
100 44 Stockholm

Department of Numerical Analysis
and Computer Science
Royal Institute of Technology
SE-100 44 Stockholm, Sweden

Physically Based Character Simulation – Rag Doll Behaviour in Computer Games

Johan Gästrin

TRITA-NA-E04008

Master's Thesis in Computer Science (20 credits)
at the School of Computer Science and Engineering,
Royal Institute of Technology year 2004
Supervisor at Nada was Kai-Mikael Jää-Aro
Examiner was Lars Kjelldahl

Abstract

The focus of this thesis is the feature known in the games industry as *rag doll* behaviour. Rag doll is the feature that, in a computer game, lets a character be influenced by actual physical laws. This enables the computer game characters to move in a more realistic way. It also gives the characters an infinite amount of possible motion patterns in contrast to animated motion. The interaction with the environment also enables more realism and better feedback.

Rag doll behaviour and the techniques necessary for implementation in real-time 3D simulation are investigated. Many different approaches have been used during the last few years to simulate physics in computer games. Some are more physically correct than others.

We describe and discuss benefits and drawbacks of the most accepted and used techniques that are suitable for simulation in real time, i.e. a computer game.

We will furthermore investigate how vertex blending can be used to mix standard animations with physical calculations to enhance computer games.

Fysikbaserad figursimulering

’Rag Doll’-beteende i dataspel

Sammanfattning

Detta examensarbete fokuserar på den egenskap som inom dataspels-industrin är känd som *rag doll*-beteende. Rag doll-beteende är den egenskap som, i ett dataspel, låter fysiska lagar inverka på figurer. Detta ger dataspelsfigurer ett mer realistiskt rörelsemönster. Det ger också figurerna ett oändligt antal möjliga rörelsemönster till skillnad från animerade rörelsesekvenser. Figurens interaktion med omgivningen ger också större realism och bättre återkoppling.

Rag doll-beteende och de tekniker som är nödvändiga vid implementering av en realtids 3D-simulering undersöks. Under de senaste åren har flera olika tillvägagångssätt prövats för att simulera fysiska lagar i dataspel. Vissa av dessa är mer fysiskt korrekta än andra.

Vi beskriver och diskuterar för- och nackdelar med de mest accepterade och använda metoderna som är lämpliga för realtidssimuleringar, d.v.s. dataspel.

Vidare undersöker vi hur vertex-blending, hörn sammansmältning, kan användas för att blanda standardanimationer med fysiska beräkningar för att förhöja dataspel.

Table of Contents

Chapter 1 Introduction **1**

Chapter 2 Animation **4**

2.1 The Graphics Rendering Pipeline 5

2.2 Vertex Blending 6

Chapter 3 Physics **9**

3.1 Kinematics 10
3.1.1 Position and Orientation 10
3.1.2 Velocity 12

3.2 Dynamics 13
3.2.1 Mass and Inertia Tensor 13
3.2.2 Force and Torque 14
3.2.3 Momentum 16

Chapter 4 Collision Handling **18**

4.1 Collision Detection 18

4.2 Collision Determination 20

4.3 Collision Response 21

Chapter 5 Constraints **22**

5.1 Springs 24

5.2 Reduced/Generalized Coordinates 25

5.3 Lagrange Multipliers 25
5.3.1 A Sparse Solution Method 26

Chapter 6 Design **29**

6.1 Bones 30

6.2 Joints 32

6.3 Tree Structure 33

Chapter 7 Evaluation **34**
7.1 Animation 34
7.2 Physics 35
7.3 Collision Handling 35
7.4 Constraints 36
7.5 Design 36
Chapter 8 Conclusion **38**
8.1 Benefits 39
8.2 Future discussion 40
References **42**

Chapter 1

Introduction

In today's computer games character motion is performed through a series of animations created in some form of three dimensional (3D) modelling program like Alias|Wavefront's Maya or Discreet's 3ds max. These animations can be produced with the help of *motion capture* [Steed, 2002]. These animations are then imported into the game.

While this is an interesting technique in itself with many difficulties, the possibility of simulating realistic physical behaviour gives us a more realistic behaviour as well as an infinite amount of movement patterns.

In the history of 3D computer graphics there has always been substantial differences between the approaches in the computer gaming industry and academic researchers doing physics simulation. While computer games have mainly been focused on achieving realistic character behaviour through pre-rendered animation, the academic community have been more interested in getting correct physical simulation.

During the last few years we have seen this difference diminish and the two fields learn from each other. The gaming industry has made a great effort to use Newtonian physics and the academic world has started to use and appreciate the fast rendering techniques developed within the gaming industry for convincing visualization.

So what has the gaming industry to gain from physics simulation and is it feasible to think that today's computers can handle this type of computation in real time? The benefits are obvious if you look at a snooker or pinball game which is difficult, if at all possible, to make without

physics, particularly if there is to be any interaction.

In 3D simulation the concept of rigid bodies is extensively used. Not only because of the simplification of physics but also the simplification of collision handling and rendering.

Lately the games development community has started to look at the possibilities of using physics in character animations as well. This has resulted in something the game developers refer to as *rag doll behaviour*.

Rag doll, as the name implies, is the ability to let the characters be influenced by the surrounding environment while apparently limp themselves, like rag dolls. This has resulted in the press using the oxymoron "Lifelike death animations" in their reviews [Jakobsen, 2001].

It is however important to emphasise that rag doll behaviour does not imply controlling the user controlled characters with physics, it merely concerns simulating physical effects on lifeless characters or lifeless limbs. These simulations can however be mixed with existing animations as discussed in chapter 2.2.

There is however research treating the subject of shifting the character control from the animations system to the physics system and thus simulating 'muscles'. This research area is called *dynamic animation* [Bruderlin & Calvert, 1989] or *physics-based character animation control* [Faloutsos et al., 2001].

What we have seen the last years is that a wide range of physics engines have emerged from third party companies and open source projects.

Among the commercial physics engines can be mentioned MathEngine's Karma (http://www.mathengine.com/) and Havok (http://www.havok.com/). Open source projects have emerged and disappeared a bit faster than their commercial counterparts, but Open Dynamics Engine (http://opende.sourceforge.net/) and Darwin2k (http://www.darwin2k.com/) are two projects that still keep evolving.

Among the open source projects some are impelled and produced by robotics and computer science institutes while others are maintained by people solely interested in game development.

At present the gap between the physics engines for games and academic

research is large. Physics engines for games have focused on vehicle and projectile physics while academic research projects lack stability and generality making them hard to use for anything other than the special purpose they where designed for.

Experts predict a growing market of middleware for the computer game industry where game physics seems to be one of the most attractive areas after graphics engines.

Rag Doll effects need the basic concept of Articulated Rigid Bodies. Articulated Rigid Bodies are based on constraining ordinary rigid bodies. These constraints are both a tricky and computationally demanding area.

We will go through the most important areas to get the basic knowledge of 3D simulation and then concentrate on constraints, which make rag doll behaviour possible. The areas we will discuss are: animation, physics, collision handling, constraints and articulated rigid bodies.

The possibilities that new techniques expose will be presented as well as the three most common ways to reinforce constraints.

Chapter 2

Animation

The largest contribution to the progress in graphic animations has come from the film industry with Pixar Animation Studio as one of the main contributors. For over a decade Pixar Animation Studio have produced short and feature films such as *Toy Story* and the early *Luxo Jr* both displaying amazing photorealistic graphics.

Behind Pixar's animated films you can find technical innovation in a graphics language known as *Photorealistic RenderMan*.

According to [Engel, 2002] one key to RenderMan's success is that the programmability of the rendering pipeline has allowed RenderMan to evolve as major new rendering techniques were invented. However, this programmability has limited RenderMan to software-only implementations, thus only working for off-line rendering.

In the last few years we have seen the ability to program the rendering pipeline, as opposed to fixed functions, appear in graphics hardware for home computers with real-time performance. This technique is similar to RenderMan's graphics language and is referred to as programmable shading and is dependent on the graphics card's graphics-processing unit (gpu).

The principal 3D graphic APIs (DirectX and OpenGL) have also evolved alongside the graphics hardware. One of the most important new features in DirectX Graphics is the addition of a programmable pipeline that provides an assembly language interface to the transformation and lighting hardware (vertex shader) and the pixel pipeline (pixel shader).

2.1 The Graphics Rendering Pipeline

The basic rendering primitives used by most graphics hardware are points, lines and triangles. When grouped together these rendering primitives constitute objects.

A scene is the collection of several objects and the environment needed to generate, or render, a visual presentation of the virtual model. A scene can also include material descriptions, lightning and viewing specifications.

The graphics rendering pipeline renders a two dimensional image from an underlying 3D representation. The rendering pipeline conceptually consists of three stages: application, geometry and rasterizer, as illustrated in Figure 1.

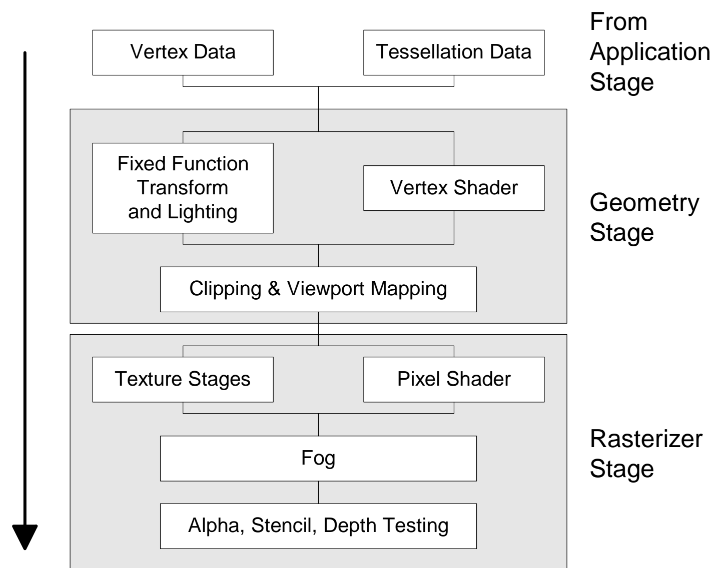


Figure 1 The graphics pipeline; with shaders.

These stages are in turn divided into one or several pipeline stages depending on implementation and hardware.

As the name implies, the application stage is implemented in software. The application stage may, for example, contain collision detection, collision response, acceleration algorithms and animations.

The geometry stage, which can be implemented either in software or hardware, deals with transforms, projections, lighting, etc.

The rasterizer stage draws an image from the data generated from the previous stages that is possible to display, usually on a computer screen.

In this thesis we are mainly concerned with the application stage. However we will briefly cover a concept known as *vertex blending* because of the possibilities it conveys to rag doll behaviour.

2.2 Vertex Blending

Characters in computer games used to consist of several rigid bodies linked together. This made the building and animation of characters tedious work. When characters moved one could often see fractures between the polygons and other unwanted behaviour.

Vertex blending addresses this problem by making the character consist of a set of bones and having an elastic skin react to changes in the poses of the bones.

The skin thus makes the character consist of only one part, the polygon mesh representing the skin of the character, see Figure 2. The vertices of the mesh are connected to one or more bones of the underlying skeleton with weights. This enables the mesh to appear soft and flexible in the joints, see Figure 3. It also makes the representation of the character more efficient. When moving it is actually only the skeleton being processed, the skin just follows the bones it is connected to.

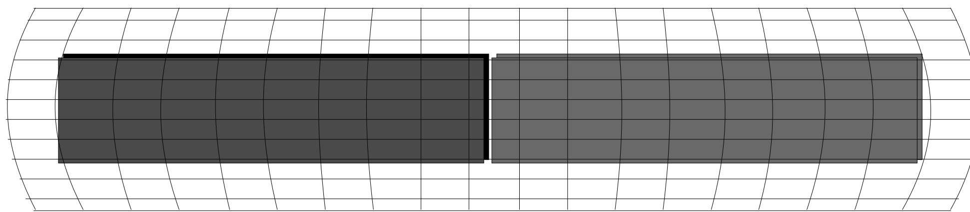


Figure 2 Skinning; how vertices are connected to the bones.

Vertex blending is not a new concept, [Laperrière et al., 1988], and is also known as skinning, enveloping and skeleton-subspace deformation.

This technique has made the use of skeletal animation widespread, therefore making the implementation of rag doll behaviour easier. We no

longer have to bother with a whole new framework. We only need to define how many and which of the bones from the graphic representation should be used in the physical description and add physical properties to them.

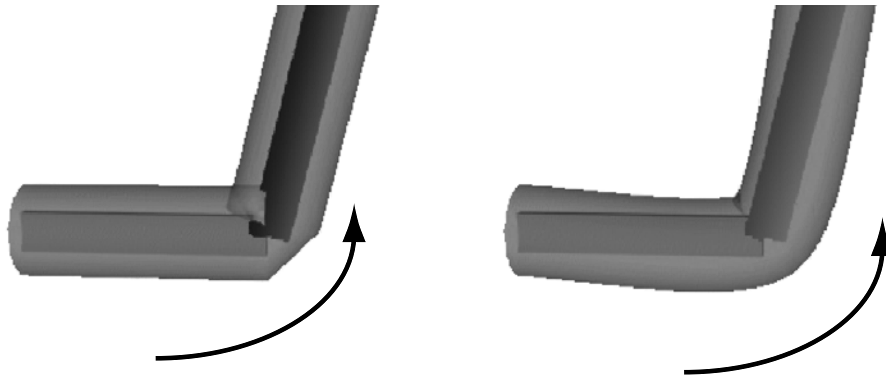


Figure 3 Vertex blending: smoothes the joints by connecting vertices on the skin to two different bones.

Vertex blending can also, and more importantly for the object of this thesis, be used to blend between multiple animations and/or physics.

Analogously with the previous case we define weights to connect the vertices to several bones. The difference in this case is that the vertices are weighted between two copies of the same skeleton, or subpart of the skeleton, performing different movements rather than two connected bones. Whether the motion of the bones is predefined by an animation or real time computed by the physics system is irrelevant for the vertex blending.

This makes it possible not only to reduce the amount of animations but also to produce an infinite amount of new moment patterns in real-time. Mixing rag doll effects with animations can for instance make recoil and hit animations redundant in *first person shooter* games.

Vertex blending can be implemented in several different ways. Except implementation in software, today's graphics hardware offer two different approaches, either fixed function or vertex shader.

Fixed function vertex blending, referred to as *indexed vertex blending*, which is the older of the two techniques, has the disadvantage of being

restricted to the specific hardware implementation, that is the graphics card.

The vertex shader on the other hand is a small assembly-language that is run on the *gpu* for each vertex replacing the transform and lightning computations of the fixed-function. This enables the developer greater control and the use of more advanced effects, see [Gosselin, 2002].

When using vertex blending to blend between different animations or physics response it might however not be desirable to use a hardware implementation because this is later in the pipeline than collision handling. This implies that the new 'blended' position is unknown to the collision handling system and thus will only be the graphical representation and no collision or response can be taken on this pose but only the 'true' pose. This can however be viewed as a feasible approximation to increase speed.

Chapter 3

Physics

During the last few years the interest in using physics in game development has increased significantly. Some say the physics engine now is more important than the graphics engine to produce a strong game title. This is mainly because the graphics, which used to be the focus of the game development community, has made such progress during the last years that realism and detail is more an issue of taste and game play than technology. Lately focus has been shifting to achieve realism through believable physics and correct response.

This can be seen in the vast number of books, articles and conference talks that have emerged during the last half-decade. Among the most renowned are the popular seminars by David Baraff and Andrew Witkin, which they have held several times at SIGGRAPH since 1995 [Baraff & Witkin, 2001]. It is also interesting to note the merging of offline and online rendering techniques in animated Hollywood films and computer games.

Physics is however a huge area and we are only concerned with the branch referred to as *classical mechanics*.

Classical mechanics, often called "Newtonian mechanics" after Isaac Newton, who made major fundamental contributions to the theory, is the physics of forces acting on bodies. It is subdivided into *statics*, which deals with objects in equilibrium, and *kinematics* and *dynamics*, which deal with objects in motion.

Classical mechanics produces very accurate results within the domain of everyday experience. But in the late 19th century inconsistencies in

classical mechanics were discovered and was thus in time superseded by relativistic mechanics for systems moving at velocities near the speed of light, quantum mechanics for systems at small distance scales, and relativistic quantum field theory for systems with both properties [Halliday et al., 1997].

Nevertheless, classical mechanics is still very useful, because it is much simpler and easier to apply than these other theories, and it has a very large range of approximate validity.

3.1 Kinematics

Kinematics is the branch of mechanics concerned with the motions of objects without being concerned with the forces that cause the motion.

Often it is the case that kinematics is sufficient to produce plausible animations or interactions.

Inverse kinematics is really no different from kinematics. The implication of "inverse" is that the system will 'work backwards' from the desired end position of one or more control points on a mechanical linkage, to infer the positions of other parts of the system so that the goal is achieved.

Inverse kinematics is known to be computationally expensive and has been thoroughly researched in the robotics community. This has resulted in many open source projects, for example http://cal3d.sourceforge.net/ which is a skeletal based 3D character animation library that can be used in many different kinds of projects involving inverse kinematics.

3.1.1 Position and Orientation

The basis of classical mechanics is the motion of particles. A particle does not have any orientation and hence the only thing we need address is the location. The location is dependent on the velocity, and the derivative of the velocity - the acceleration - is in its turn dependent on the forces acting on the particle.

To represent the location at time t we form the vector $x(t)$, which describes the offset from the world origin, thus defining the translation from the *body fixed reference frame* to the *world fixed reference frame*, which we will describe in more depth shortly.

Bodies on the other hand also have an orientation. When simulating rigid body physics we therefore need to extend the simple particle physics with an orientation and an angular velocity. The angular velocity is in its turn dependent on the torque, the part of a force that produces rotation about an axis.

The *body fixed reference frame,* often referred to as *body space,* is the local coordinate system fixed on the body. To go from the general *world fixed reference frame,* or *world space*, to a body's body space coordinates we need to both translate and rotate the body, see Figure 4. Even though the body moves, i.e. translates and/or reorients, the body space stays intact.

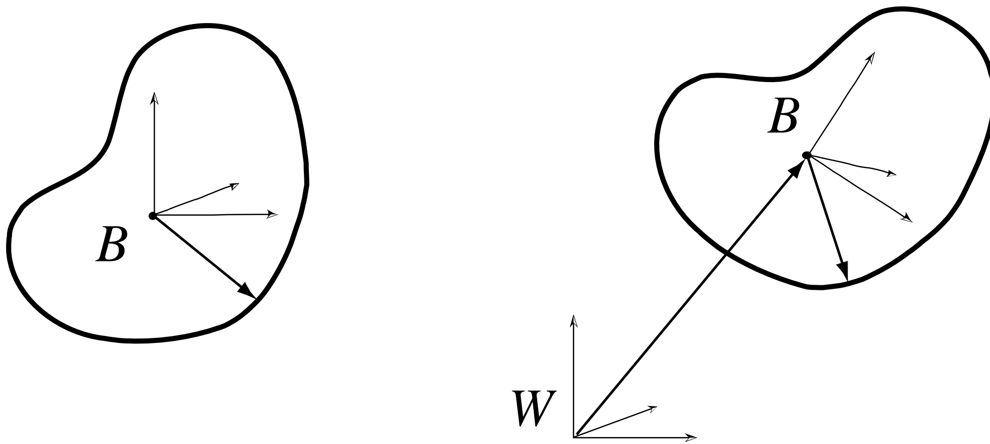


Figure 4 Translation and rotation from body space to world space.

There are two commonly used ways to represent orientation. The first and simpler is a 3×3 matrix, *R(t)*, consisting of three vectors spanning the body space of the body. The other is *unit quaternions q(t),* see [Shoemake, 1985] and [Eberly, 2002]. A quaternion is a four-dimensional representation of a three-dimensional rotation and consists of four parameters as $q(t)=(r,i,j,k)$, where r is the real part and i,j,k represent the vector part. In three dimensions we should be able to represent a rotation uniquely with only three parameters. So why do we not simply represent a rotation with a three element vector? The problem is that this could be interpreted in two different ways due to the ambiguity of the vector, a positive and a negative direction.

A simple representation with a 3×3 matrix has many advantages such as easy to comprehend and fast transformation performance. But there are also drawbacks. Representing a rotation with a 3×3 matrix means we use nine parameters. This gives us a redundancy of 6 parameters representing the tree degrees of freedom of the rotation. Compared with quaternions only $4-3=1$ parameter overhead. Apart from the excess usage of memory, there is also a huge difference in numerical drift. Both the matrix and the quaternion need to be normalized to be consistent with representing only a rotation. However renormalizing a quaternion to adjust for floating point errors is cheaper than renormalizing a rotational matrix.

Numerical drift of a rotational matrix can also convey in a more severe consequence, the applying of skewing. This is of course disastrous since it distorts the shape of a body, and can for example convert a football to a rugby ball.

Apart from being more exact and resisting numerical drift, quaternions also perform composition of rotations much more efficiently [Eberly, 2002]. Furthermore, because conversion between quaternion and rotational matrix representations is an inexpensive task, we can consider converting to rotational matrix representation only while performing the transformations.

Thus it is recommendable to represent the bodies' spatial variables as a vector $x(t)$ representing position and a quaternion $q(t)$ representing orientation.

3.1.2 Velocity

The velocity defines how our spatial variables change over time. Thus we need to define $\dot{x}(t)$ and $\dot{q}(t)$. The change of position over time is called linear velocity and denoted $v(t)$. Linear velocity in world space is defined as $v(t)=\dot{x}(t)$, and is simple enough to comprehend.

In addition to translating, a rigid body can also spin. If we fix the position of a point in the body, any movement of the body must be due to the body spinning about some axis that passes through the fixed point, otherwise the point would itself be moving. We can describe that spin as a vector $\omega(t)$.

The direction of $\omega(t)$ gives the direction of the axis about which the body is spinning and the magnitude $|\omega(t)|$ tells how fast the body is spinning. This quantity $|\omega(t)|$ is called the angular velocity.

The change of orientation over time is a bit trickier to define than the linear counterpart, especially if quaternions represent the orientation. Therefore we only state for the quaternion case that $\dot{q}(t) = \frac{1}{2}\omega(t)q(t)$ and refer to [Eberly, 2002] for the derivation.

3.2 Dynamics

If kinematics is the field describing movement over time then dynamics is what causes it. Thus dynamics is the study of forces and masses.

3.2.1 Mass and Inertia Tensor

The *inertia tensor I* describes the distribution of the mass in the body. This is what causes the spin of a body to vary depending on where the force is applied, even though the magnitude of the force is the same, see Figure 5.

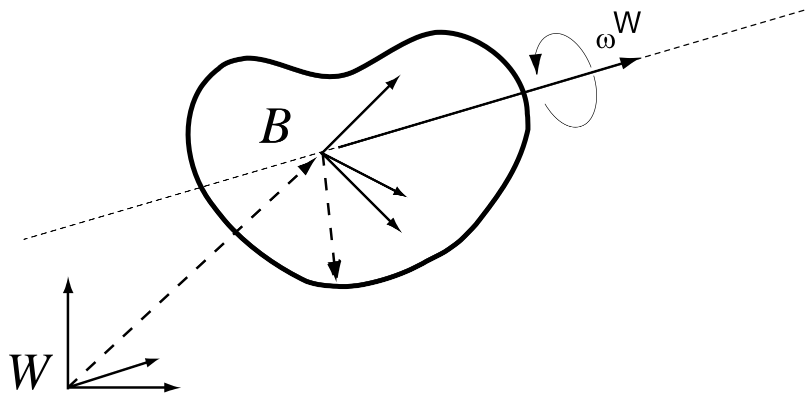


Figure 5 Distribution of mass over the body, inertia tensor.

When treating bodies the intrinsic mass distribution of the body in body space is fixed, as is usually, but not necessarily, the total mass *m* of the body. To simplify calculations and minimize the amount of state parameters the origin of body space is usually chosen to coincide with the centre of mass.

If the centre of mass coincides with the origin of the body space the inertia tensor *I* does not change with time. Hence we only need to calculate the inertia tensor in world space as $I(t) = R(t)I_{bs}R(t)^T$.

Since the inertia tensor *I(t)* is needed to transform between angular momentum and angular velocity *w(t)*, as we will discuss in more depth shortly, it is important that this is easy and inexpensive to compute.

Note that the inertia tensor is a 3×3 matrix and the mass only a scalar.

The inertia tensors for the most common and simple geometrical shapes are easy to look up, while deriving them is another issue and will not be dealt with here, [Stejskal & Valásek, 1996]. Table 1 shows the inertia tensors for some of the simpler geometrical shapes that are of interest when modelling characters.

Table 1 Inertia tensors for some common geometries

Geometry	Moment of Inertia
Solid cylinder	$I = \frac{1}{2}mR^2$
Solid sphere	$I = \frac{2}{5}mR^2$
Rod	$I = \frac{1}{3}mR^2$

3.2.2 Force and Torque

A *force* may be thought of as any influence that tends to change the motion of an object, such as gravity, drag or contact. In dynamics, a force is divided into one part that cause linear motion and one part that cause rotational motion. The latter is called *torque*, denoted τ. Newton's second law describes the action of forces in causing motion as $F = \partial(mv)/\partial t$, where *F* is the net external force, *m* the mass and *v* the velocity. Typically, the mass *m* is constant in time, and Newton's second law can be written in the simplified form $F = ma$.

The torque, in contrast to the linear part of the force, is dependent on where on the body the force is applied. The torque is defined as $\tau = p \times F$, that is, the cross product between the force *F* and the position vector of the point *p* where the force is applied on the body relative to the body centre of mass. As can be seen in Figure 6, the torque is perpendicular both to the applied force and to the straight line between the centre of mass and the point of contact.

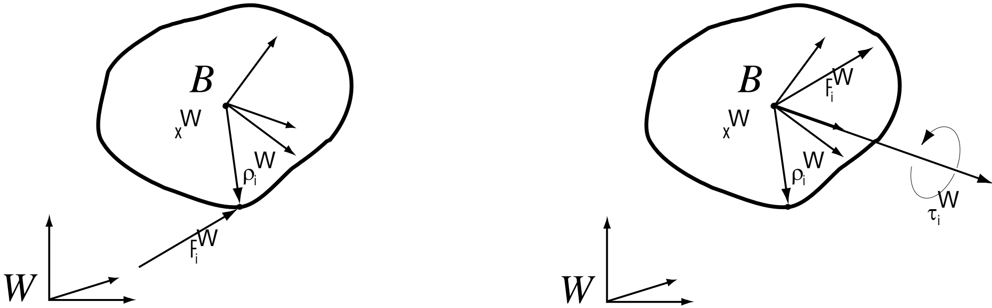


Figure 6 Applied force, point of contact and resulting torque.

In considering the effect of a force acting at a point on a body it sometimes seems that the force is being considered twice. That is, if a force *F* acts on a body at a point *p* (*p* + *x(t)* in world space), then we first consider *F* as accelerating the centre of mass, and then consider *F* as imparting a spin to the body.

This gives rise to what at first seems as a paradox if comparing two situations with identical bodies and forces but with different points on the bodies where the force is applied. In one case the force acts at the centre of mass, and in one off-centre, see Figure 7.

In both cases the bodies gain the same amount of linear acceleration, but the body where the force was applied off-centre also has picked up an angular acceleration and therefore a larger amount of energy from the same force. This is due to the fact that energy, or work, is the integral of force over distance. Hence the body where the force was applied off-centre gains more energy since the force was applied over a longer distance.

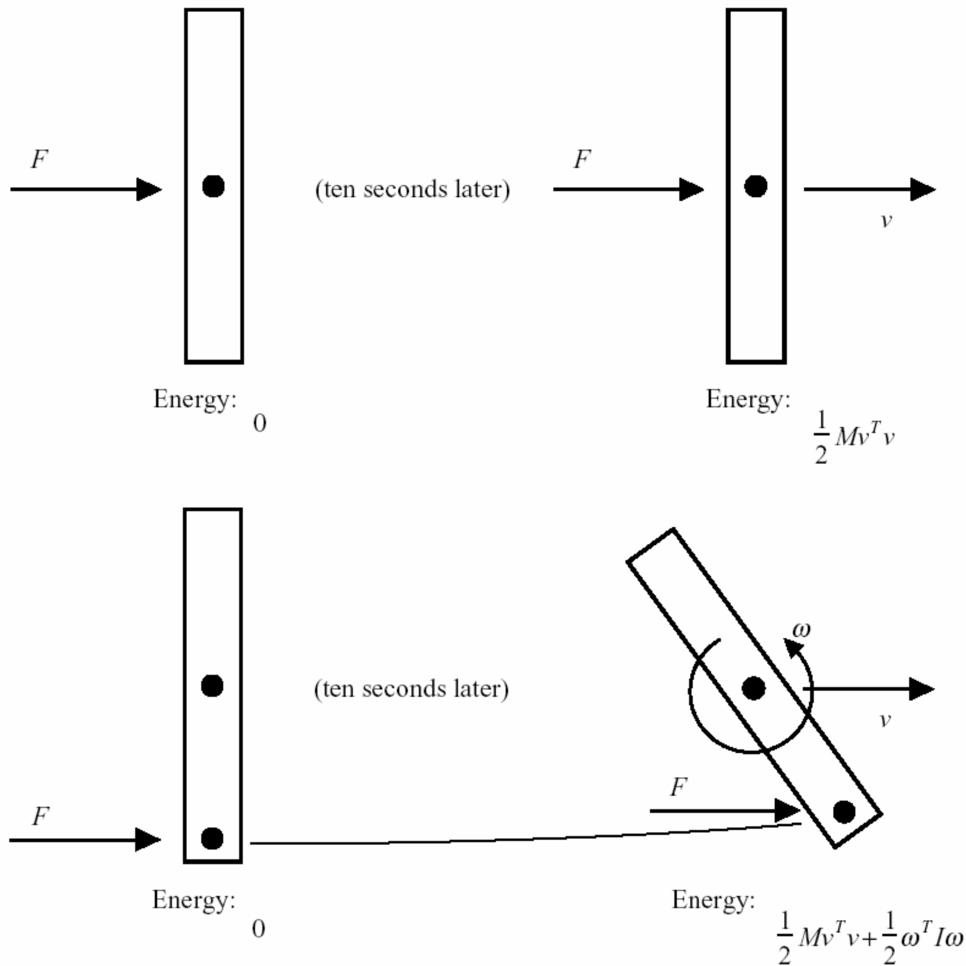


Figure 7 Force giving rise to both linear and angular acceleration.
Image from [Baraff & Witkin, 2001].

3.2.3 Momentum

Like all the other properties we have dealt with in this chapter, *momentum* can be divided into its position dependent and its orientation dependent parts.

Linear momentum is the property that deals with position and is defined as $P(t) = mv(t)$, where *m* is the mass of the body and *v(t)* the velocity of the centre of mass.

Analogously *angular momentum* is defined as $L(t) = I(t)\omega(t)$, where *I(t)* is the inertia tensor and $\omega(t)$ the angular velocity.

There is a simple relation between linear momentum and force $\dot{P} = F$ and between angular momentum and torque $\dot{L} = \tau$.

Because the inertia tensor *I(t)* is dependent on the orientation, hence time, the angular velocity is not necessarily constant even if the angular momentum is constant. Therefore it is advisable to use the angular momentum as a state variable rather than the angular velocity, which is easily calculated as $\omega(t) = L(t)/I(t)^{-1}$.

To be consistent with the angular case it is also desirable to use the linear momentum instead of the linear velocity as a state variable.

Chapter 4

Collision Handling

Collision handling is of very large importance in any 3D computer simulation. Without it no interaction between objects is possible.

Collision handling can be divided into three major parts: *collision detection*, *collision determination* and *collision response*. These parts are all both intellectually and computationally demanding areas and what follows here is only a short introduction. A more thorough introduction to the vast theory of collision handling can be found in [Akenine-Möller & Haines, 2002].

Collision detection and collision determination is closely coupled when considering actual implementation and when discussing collision detection one usually refer to them both when talking about collision detection.

Collision response on the other hand is usually separate from both collision detection and collision determination thus making the geometric representation rather transparent. But we still need to be informed of occurring collisions (collision detection) and collision data (collision determination) before computing the action to be taken (collision response).

4.1 Collision Detection

During each time step the collision detection system scans the scene for collisions or interpenetrations and reports them.

The distinction between interpenetration and collision is dependent on the

Euclidean distance between two body surfaces. If it is below zero the two bodies interpenetrate and if the Euclidean distance is equal to or within a positive threshold value and the bodies do not interpenetrate they are said to be colliding.

The collision detection is usually divided into two phases, the *broad phase* and the *narrow phase*. The object of the broad phase is to exclude as many objects as possible from further collision detection checking. The narrow phase will then perform a more thorough collision detection with the objects that are left after the broad phase. During the narrow phase we usually want to use different collision detection algorithms depending on what kind of objects we are dealing with.

Through out the collision handling the representation of objects is usually something simpler than the actual meshes representing the objects graphically, making the testing quicker. This simpler representation is referred to as *bounding volumes*.

Some common bounding volume choices are; Bounding Sphere, Axis Aligned Bounding Boxes (AABB), Oriented Bounding Boxes (OBB), Discrete Orientation Polytope (k-dop) and Convex Hull, see Figure 8. These are then usually ordered in a tree structure for faster traversal. Binary Space Partitioning (BSP) trees are another commonly used technique.

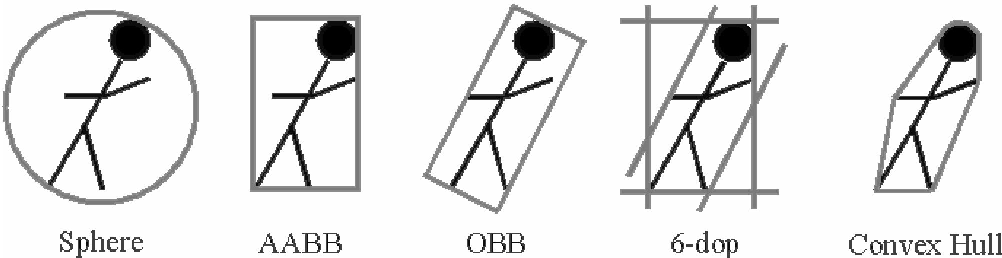


Figure 8 Different bounding boxes.

Note that it is left to a collision response system to resolve the collision. The collision detection system only detects the collisions.

A common problem with collision detection is tunnelling. Tunnelling is when an object penetrates another object, for instance a wall, and is able to

pass trough entirely during one time-step, thus making it hard to detect collisions. This is usually due to small objects moving at high velocities, [Moore & Wilhelms, 1988].

4.2 Collision Determination

Once a collision has been identified the collision determination system steps in. The collision determination system's main concern is to collect collision data. The needed collision data is time of collision, participating objects, point of contact on the objects, collision normal and the relative velocities of each object at the point of contact, see Figure 9.

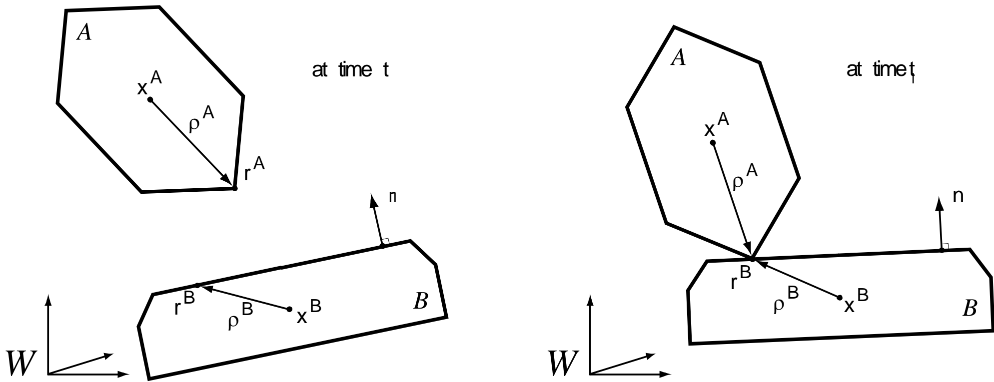


Figure 9 Two colliding bodies.

Depending on what kind of objects and how many different sorts we are dealing with this can be of varying difficulty, with many different complex objects the hardest to handle. This is why bounding volumes usually are used to approximate the objects during the collision determination as well to keep the complexity and number of geometries to a minimum.

This of course affects the correctness of the collision data computed, and we thus have to balance speed and correctness.

Because a collision does not necessarily occur at the end of a time step cycle another problem arises. When an interpenetration state has been found, thus the collision occurred sometime during the last time step, we have another choice between speed and correctness. Either the collision

detection system must backtrack to find the collision state prior to the interpenetration or we have to approximate the collision data.

4.3 Collision Response

The collision response system decides what action is to be taken when a collision has occurred. This involves computing and applying forces on the involved objects.

It is the area of collision handling that has evolved most the last few years, from simple boolean expressions deciding the response to using the laws of physics.

Some [Dingliana & O'Sullivan, 2001] argue that it is hard for humans to judge whether a collision response is more or less correct, especially in multiple dimensions, i.e. 3D, while others [Hecker, 1996] say that we underestimate the human perception capacity and that more correct response, even though not 'visible', gives a more believable response.

It is in the light of this development that resting contact, friction and trajectories have conquered the gaming industry.

Resting contact and friction are closely coupled and address the problem of objects sliding or standing on other objects, for instance a pile of boxes stacked on each other.

At the heart of handling contacts we find *constraints*. This is also what we need for rag doll behaviour.

Chapter 5

Constraints

As we have seen in the previous chapter the issues concerning rigid body dynamics are both well documented and widely used. But how are we going to link these bodies together to form composite articulated rigid bodies? The answer is constraints.

Constraints are today mainly used in 3D simulations to reinforce body solidness, to prevent interpenetration between bodies. These constraints are usually referred to as inequality constraints. For example the position of, for instance, a ball can be any as long as it is above ground,
ball altitude > ground level.

The other type of constraints is referred to as equality constraints and is for instance used to reinforce the geometric connection between bodies or to reduce the degrees of freedom, i.e. disabling motion along certain directions or axes. Consider for example a bead on a string, it can slide on the string but still has to follow the extent of the string, thus restricting the linear motion into one dimension.

When modelling articulated rigid bodies we usually divide them into positional and angular constraints. The positional constraints keep the bodies from separating or interpenetrating, these are equality constraints. The angular constraints restrict the bodies from occupying forbidden angular poses relative to each other. These are inequality constraints.

Both equality constraints and inequality constraints belong to the group of constrains called holonimic constraints. There is also another group of constraints called nonholonomic constraints. These constraints are related

to velocity rather than position as the former holonomic constraints are, see [Dankowicz, 1999]. These constraints make it possible to control relative velocities, useful when simulating for instance a gearbox.

Rigid bodies, whether articulated or not, have six degrees of freedom in 3D, see Figure 10. That is, they can move in six possible directions. Three linear degrees of freedom, consisting of the three spatial dimensions, and three angular degrees of freedom, consisting of roll, pitch and yaw. It is these degrees of freedom that the constraints will restrict.

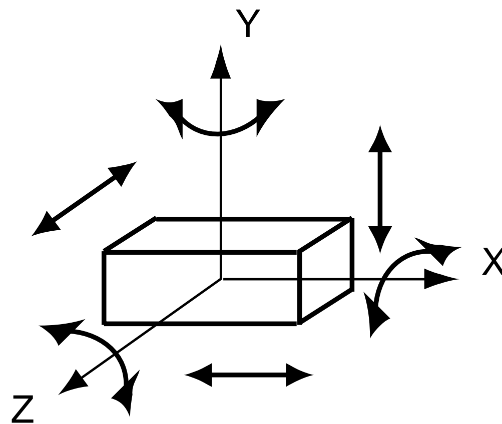


Figure 10 Illustration of the six degrees of freedoms in 3D.

Typically, in a system describing characters the constraints are sparse: each constraint directly affects only one or two bodies (for example, geometric connection constraints) and there is not extensive branching. Thus, for a system with n bodies, there are only $O(n)$ constraints.

In particular, the simulation of articulated figures and mechanisms falls into this category. Sparse constraint systems are also either nearly or completely acyclic: for example, robot arms are usually open-loop structures, as are animation models for humans and other mammals. Considerable effort has been directed toward efficiently simulating these types of systems.

Because of the complexity of constraints the need and choice of algorithm is important. However, without constraints we can only have simple single rigid bodies without interbody associations. This is why constraints

are of growing importance.

When modelling constraints there are several ways to go about the problem, some more physically correct than others.

Here follows a description of the three most well known implementations in real-time 3D rendering and a discussion of their advantages and disadvantages.

5.1 Springs

A spring is an elastic device, such as a coil of wire that regains its original shape after being compressed or extended. This concept is often used in 3D simulation in various ways. It can be used to simulate things like dampers on vehicles. But it can also be used in particle systems in various ways and also to simulate constraints.

There are however some problems with this approach. Even though springs are easy enough to implement the stability is far from acceptable as well as the problem with getting them to simulate hard constraints like angular stops in for instance an elbow.

In [Jakobsen, 2001] the physics engine for the game *Hitman – Codename 47* is described. Here the basis of the whole physics engine is a particle system that, with the aid of springs, simulates both rigid bodies and the articulated rigid bodies to become one of the first games to achieve rag doll behaviour.

In *Hitman – Codename 47* constraints are handled by relaxation and stiffness of the springs. The stiffness of the springs goes to infinity when a connection is wanted.

This is an instable way to simulate constraints and demands a big amount of tweaking and error correction as well. Apart from that it is not very physically correct. However, it is fast and, apart from the tweaking, simple to implement.

Springs are, though maybe not preferable when simulating constraints, useful in conjunction with constraints to simulate inverse kinematics.

5.2 Reduced/Generalized Coordinates

A more physically correct way to simulate constraints, and an approach thoroughly used and researched in the academic mechanics community, is to constrain the degrees of freedom by reducing the actual number of variables describing the body state.

This approach is called Generalized coordinates, as opposed to maximal coordinates, which is the name for describing the system with its full set of state variables. It is also referred to as Reduced coordinates due to the fact that this describes how the technique works.

In a 3D system consisting of n bodies, each with six degrees of freedom, a reduced coordinate formulation removes the c constraints from the system leaving a set of $6n - c = m$ parameterised coordinates to define the system.

Finding a generalized parameterisation for $6n$ maximal coordinates is however a tedious and arbitrarily hard work. If such a parameterisation can be found, $O(n^3)$ time is required to compute the acceleration of the m generalized coordinates at any instant.

However, loop-free articulated rigid bodies are trivially parameterised, and methods for computing the m generalized coordinate accelerations in $O(n)$ time are well known and described in [Featherstone, 1987].

One big advantage with the generalized coordinate approach is the elimination of drift giving us a stable system. Furthermore this method is possibly faster than others because of the possibility for the integrator to take larger time steps. This is however not any general rule and depends on the implementation.

The biggest problem with this approach however is that it assumes extensive analytical mathematical knowledge. This is needed not only during implementation of the system but also continuously while new sets of objects and constraints are introduced. This can to a certain extent be automated, but the amount of code grows very quickly, [Witkin et al., 1990].

5.3 Lagrange Multipliers

In the Lagrange multipliers approach the technique is to introduce

additional constraint forces to maintain the constraints rather than trying to eliminate degrees of freedom.

Lagrange multipliers are good for arbitrary sets of combined constraints and render possible a modular framework. They also handle nonholonomic constraints, velocity dependent constraints, in a simple way.

The standard Lagrange multiplier implementations have been seen to perform $O(n^3)$ but in [Baraff, 1996] is presented a Lagrange multiplier algorithm that performs in $O(n)$ for a sparse, open loop system. With 'sparse' is implied that constraints only affect two or fewer bodies, which is the case for a character.

The heart of any Lagrange multiplier formula is solving the matrix equation:

$$JM^{-1}J^T\lambda = c$$

where the vector λ contains the multiplier elements we wish to solve for.

J contains the constraint connectivity and M the mass properties of the bodies while the vector c contains the forces being applied to the bodies.

5.3.1 A Sparse Solution Method

Now, if we restrict the constraints to act only between pairs of bodies we will achieve a sparse matrix. Furthermore we need to ensure that the bodies are ordered correctly.

Following the algorithm described in [Baraff, 1996] we start by stating that we follow the second-order law $F = ma$, that is

$$F = m\dot{v} \quad (1)$$

as discussed in previous chapters.

We also divide the total force into the external force F^{ext} and the constraint force F^c like:

$$F = F^c + F^{ext} \quad (2)$$

The constraint force F^c should be workless, that is, it should (almost, see [Barr & Barzel, 1988]) not add any energy to the system. To ensure this we need to add the condition $F^c = J^T\lambda$.

Thus

$F = F^{c} + F^{ext} = J^{T}\lambda + F^{ext}$ (3)

and substituting F in (1) gives:

$M\dot{v} = J^{T}\lambda + F^{ext}$

solving for the acceleration gives:

$\dot{v} = M^{-1}J^{T}\lambda + M^{-1}F^{ext}$ (4)

As discussed earlier the constraints can be expressed as a linear condition on the bodies' accelerations. In matrix form this would look like:

$J\dot{v} + c = 0$ (5)

Substituting (4) in (5) gives:

$J(M^{-1}J^{T}\lambda + M^{-1}F^{ext}) + c = 0$ (6)

If we define b as: $b = -(JM^{-1}F^{ext} + c)$

$$\begin{pmatrix} M & -J^{T} \\ -J & 0 \end{pmatrix}\begin{pmatrix} y \\ \lambda \end{pmatrix} = \begin{pmatrix} 0 \\ -b \end{pmatrix}$$

$My - J^{T}\lambda = 0 \Rightarrow y = M^{-1}J^{T}\lambda$

$-Jy = -b$

Solving row one for y and substituting this into row two gives:

$Jy = J(M^{-1}J^{T}\lambda) = b$

For convenience we define H as:

$$H = \begin{pmatrix} M & -J^{T} \\ -J & 0 \end{pmatrix}$$

For four bodies connected like a tree H would look something like:

$$H = \begin{pmatrix} M_1 & 0 & 0 & 0 & J_{11}^T & 0 & 0 \\ 0 & M_2 & 0 & 0 & J_{12}^T & J_{22}^T & J_{32}^T \\ 0 & 0 & M_3 & 0 & 0 & J_{23}^T & 0 \\ 0 & 0 & 0 & M_4 & 0 & 0 & J_{34}^T \\ J_{11} & J_{12} & 0 & 0 & 0 & 0 & 0 \\ 0 & J_{22} & J_{23} & 0 & 0 & 0 & 0 \\ 0 & J_{32} & 0 & J_{34} & 0 & 0 & 0 \end{pmatrix}$$

It is due to the fact that H is always sparse that we can find a linear solution method to the articulated system.

However, to exploit this sparsity we need to permute H first.

This is easily done using the O(n) *sparsefactor* procedure described in [Baraff, 1996], producing a permuted H as:

$$H = \begin{pmatrix} M_3 & J_{23}^T & 0 & 0 & 0 & 0 & 0 \\ J_{23} & 0 & 0 & 0 & J_{22} & 0 & 0 \\ 0 & 0 & M_4 & J_{34}^T & 0 & 0 & 0 \\ 0 & 0 & J_{34} & 0 & J_{32} & 0 & 0 \\ 0 & J_{22}^T & 0 & J_{32}^T & M_2 & J_{12}^T & 0 \\ 0 & 0 & 0 & 0 & J_{12} & 0 & J_{11} \\ 0 & 0 & 0 & 0 & 0 & J_{11}^T & M_1 \end{pmatrix}$$

Solving this system is then done by using the O(n) *sparsesolve* procedure described in [Baraff, 1996].

Note that the *sparsefactor* procedure only needs to be called once for one specific configuration of bodies and constraints because H is not time dependent.

Chapter 6

Design

Following the results from previous chapters we are now ready to design a framework that enables rag doll behaviour.

The foundation of any 3D simulation is rigid bodies. Even if soft bodies are under research, see [Barr et al., 2000] and [Barr & Platt, 1988], rigid bodies still hold the upper hand in high performance 3D simulations such as computer games.

Articulated rigid bodies can be used to describe not only characters but also other forms of objects like an articulated tank in a computer game, enabling motion of for instance the canon turret separate from the body of the tank.

As the name implies an articulated rigid body consists of two or more rigid bodies connected to each other through joints, thus forming one coherent body.

The framework presented here is specialised on handling character simulation. This does however not prevent parts of the framework to be used to simulate other forms of *articulated rigid bodies,* like the tank described above.

The building blocks, and also the data structure, that we compose the framework of are bones and joints.

6.1 Bones

Bones is how we choose to represent the abstract physical bodies constituting the parts of an articulated rigid body. There are several reasons why choose bones for this representation.

First this is the most contiguous approach compared to reality, i.e. the human body, which consist of bones. Secondly the way characters are constructed in 3D modelling programs, like Alias|Wavefront's Maya or Discreet's 3ds max, depend on skinning as described in Chapter 2. This technique depends on forming a skeleton covered with a skin. The bones of this skeleton generated in the modelling phase can easily be extended to contain the data needed for the framework. Thus using these bones will reduce the data redundancy as well as improve comprehension while this is consistent.

Note that the shape of a rigid body is not a dynamical property (except insofar as it influences the various mass properties). It is only collision detection and visualization that is concerned with the detailed shape of the body.

A rigid body has various properties from the point of view of the simulation. Some properties change over time:

- Position vector of the body's point of reference.
- Linear velocity of the point of reference.
- Orientation of a body, represented by a quaternion or a 3×3 rotation matrix.
- Angular velocity vector, which describes how the orientation changes over time.

Other body properties are usually constant over time:

- Mass of the body.
- Position of the centre of mass with respect to the point of reference.
- Inertia matrix.

Conceptually each body has an x-y-z coordinate frame embedded in it, which moves and rotates with the body, *body space*. The origin of this reference frame is the body's point of reference. Some values in the system (vectors, matrices etc) are relative to the body coordinate frame, and others are relative to the global coordinate frame.

We have now covered all the concepts we need to describe the state of a rigid body and also their differentials. In addition to this we have some

auxiliary quantities associated with our state variables.

The state variables are position, orientation, linear momentum and angular momentum. The mass and the inertia tensor are body specific constants and because the latter's inverse is frequently used to calculate the inertia tensor in world space, this is also stored as a body constant.

The derivatives of the state variables are:

$$\dot{x}(t) = v(t)$$

$$\dot{q}(t) = \frac{1}{2}\omega(t)q(t)$$

$$\dot{P}(t) = F(t)$$

$$\dot{L}(t) = \tau(t)$$

The auxiliary quantities we need are linear velocity, angular velocity and the inverse of the inertia tensor in world space and these are computed accordingly:

$$v(t) = \frac{P(t)}{m}$$

$$\omega(t) = I(t)^{-1}L(t)$$

$$I(t)^{-1} = R(t)I_{bs}^{-1}R(t)^T$$

$$R(t) = \begin{pmatrix} 1-2j^2-2k^2 & 2ij-2rk & 2ik+2rj \\ 2ij+2rk & 1-2i^2-2k^2 & 2jk-2ri \\ 2ik-2rj & 2jk+2ri & 1-2i^2-2j^2 \end{pmatrix}$$

In addition to this we need to store the quantities that change the body state, force, and calculate its rotational influence, torque.

These bones we define as a subclass of the bones that graphically constitutes the character. As we mentioned in Chapter 2, we import the characters from a 3D modelling program. What we actually import is the skeletons and the attached skin.

We now need to conclude which of the bones that the framework needs to take into account. We are not interested in having the framework perform calculations on bones that do not move, for instance the bones in the hand.

The bones are constructed during modelling to generate a detailed character but during simulation including all the bones of the fingers would make the calculations much to heavy and not add much realism.

Which of the bones to be excluded from the calculations is dependent on each title and what movement is desired for that character. This selection must, as I can see it, always be done by hand. It can however be indicated with a parameter by the graphics artist and then automated in the import phase.

6.2 Joints

Bones are connected to each other with joints. These joints handle all the constraints between a pair of bodies, i.e. bones. Every character, containing bones and joints, is separate from other characters. A character is thus a group of bodies that cannot be pulled apart and each bone is connected somehow to every other body in the character. Each character in the world is treated separately when a simulation step is taken.

Joints need to be attached to a point in each of the two bones it connects making the bones inseparable. A joint can furthermore restrict the movement between the bones. There are two ways joints can restrict how the bones moves relative to each other. It can restrict the degrees of freedom, thus disabling rotation around certain axis, and it can restrict the permitted angles in certain directions, *angular stops*.

To enforce the properties of the joints, connection and relative poses, we need *constraints.* The most favourable way to implement constraints for an application like this is the sparse solution method presented in [Baraff, 1996]. As we have described in Chapter 5, this is due to the low complexity and modularity it conveys.

In the design of this framework I prefer viewing joints as similar to human joints as possible while still as simple to simulate as possible. We therefore design three types of joints: hinge joints, ball-and-socket joints and pivot joints. There are other joint types in the human body, like condyloid joints, gliding joints and saddle joints, but the previous three are sufficient to simulate human motion.

Hinge joints restrict the relative movement of the connected bones to only

one degree of freedom simulating an elbows or a knee. Ball-and-socket joints enable movement in all directions simulating fort instance the carpal joint. Pivot joints enable a twisting movement simulating the twisting motion in for instance the neck.

We might also want to implement slider joints and contact joints for other parts of the simulation, for instance vehicles.

The joints are composed of several different constraints. One constraint is needed to fix the connection points of the two connected bones. This connection point does not necessary lay within the graphical representation of the bones. Furthermore there are constraints restricting the relative movement thus forming the different joints presented.

The joints also need to contain the constraints controlling angular stops. These, as well as contact constraints, are inequality constraint and cannot be solved using the sparse solutions method we use to solve the equality constraints.

Thus we need to implement the ordinary Lagrange Multiplier solver as well. Because the higher complexity involved in this solver it is important to bear this in mind when defining the constrains and angular stops.

6.3 Tree Structure

When, as we are, considering a character skeleton the bones and joints are related to each other without loops. This makes it convenient to approach it as a tree structure. This tree structure is also necessary for the constraint solution method we wish to use. This order is an intuitive father-son relationship with any root node. The ordering does however not necessarily have to be specified by hand. This can rather be automated by a subroutine as shown in [Baraff, 1996]. The bones composing the characters need though be specified by hand as discussed in chapter 6.1.

Chapter 7

Evaluation

The design of a rag doll system is a fairly complicated task. It not only has to interact with many different systems it is also a computationally expensive task.

7.1 Animation

When using vertex blending to mix two or more key-frame animations it is clearly advantageous to make use of the new graphics hardware available on today's graphics cards while this increase performance.

In this thesis we have also developed the idea of using vertex blending to mix one or more existing animations with the output from a physics framework. We can thus achieve rag doll features such as recoil and impact effects, on parts or whole of moving characters.

Using hardware rather then software when implementing vertex blending does however imply a short deviation from the true position of the character.

When the visual movement and position do not coincide with the true position and movement the collision handling system, which is earlier in the graphics pipeline, will not be able to take action to these visual positions. This is however a feasible approximation as long as the movements still are fairly small.

When choosing between the possible hardware implementations for

vertex blending it is preferable to use vertex shaders rather than fixed functions. Not only because of the greater freedom it gives but also because this is a newer technique that supports upgrade and is therefore probable to be supported for a longer time and thus making the framework less sensitive to hardware changes.

7.2 Physics

All physics merely try to describe the reality and are thus only approximations of it. As David Baraff writes in the opening paragraphs of his PhD thesis [Baraff, 1992], "we do not consider the dynamics models used in this thesis to be empirically correct. That is not to say our dynamics models are ad hoc or deliberately wrong ... our 'simple' dynamics models are incomplete descriptions of more 'complicated' dynamic models".

In a real time application like a computer game we need to find a balance between correctness and performance. We do however have some minimum criteria. The simulation is not allowed to crash and it also need to be consistent. That is, it should produce the same output given a certain input.

Classical mechanics is a very good approximation of the reality that we are imitating in a computer game. It is also far more correct than the methods used in older computer games.

In older ad hoc approximations, parameter tweaking is common. This parameter tweaking is undesirable since if N bodies each have M tuneable parameters, and if the stability is very sensitive to the values of the parameters, then finding a 'stable' configuration by tweaking parameters is essentially a search in M^N dimensional space!

Classical mechanics copes with our criteria and is feasible to use on a modern home computer.

7.3 Collision Handling

Collision detection and collision determination is not directly dependant on the rag doll system and can therefore be of any choice. The collision

response system on the other hand is where the resulting actions are calculated. The physics and rag doll system is a part of this general system. The rag doll system is concerned with resolving the collision response for the characters.

The resulting forces are then feed back to the main system.

7.4 Constraints

When implementing constraints for a rag doll framework we are concerned with finding a solution method that can handle a varying number of joints. It is therefore desirable to have as low complexity as possible.

There are two ways of achieving as low complexity as linear for constraints, *reduced coordinates* and *Lagrange multipliers* using the sparse solution method described in [Baraff, 1996]. It is possible to attain linear complexity with Lagrange multipliers because our characters all have a tree structure.

The modularity that Lagrange multipliers offer is far easier to both implement and maintain then the complex reduced coordinates.

7.5 Design

The calculations that are needed for a rag doll framework is dependent on how many joints that are active at a certain instance. So how many joints are needed for a character and how many characters can we handle simultaneously?

The amount of joints in a character is dependent on the number of bones they need to join. Furthermore the degrees of freedom of each joint affects the number of calculations. The more degrees of freedom the more calculations.

A human body consists of about two hundred bones. This would be very demanding for one of today's home computers. Most of the bones would not add any visual effect to the game either. To compose a human character of around twenty bones is more appropriate. Then one of

today's home computers would be able to handle between ten and a hundred characters simultaneously depending on configuration.

It is also not needed to have the characters active all the time, but only when they interact with the environment. For example a corpse lying still need not be considered until something hits it, then it is activated in the physics framework.

Because skinning is already in use in virtually all computer game development nowadays the character skeleton is already a natural part in the production line. The structure of the characters does therefore not imply any additional work. Only defining the joints and the active bones is necessary.

Chapter 8

Conclusion

In this thesis we have unravelled the mystic around the much-discussed feature called rag doll. It has proven to require a physics framework of which we have described the necessary parts and equations.

We have given a background and the building blocks necessary to complete this framework. We have also delved deeper into the different approaches to solve constraint, which is the computationally expensive part of implementing rag doll behaviour.

This new technique that is emerging as an important part of game development has not previously been covered in literature. This is probable due to the different fields it covers. Not only do we need extensive knowledge of game development and physics, we also need to familiarize ourselves with complex mechanical simulation techniques to solve the constraints necessary for articulated rigid bodies.

Except providing an easily comprehensive exposition of the comprising elements needed for a rag doll framework, we have also developed a new way of using our rag doll framework. With the help of this new idea of using vertex blending we can mix one or more existing animations with the output from our physics framework. We can thus achieve rag doll features such as recoil and impact effects, on parts or whole of moving characters, and not only on lifeless characters.

The idea we have developed in this thesis to extend the rag doll framework to work in synergy with the existing animations to enhance them with recoil and impact effects on characters limbs through vertex

blending has also economic benefits, as well as the feature effects.

8.1 Benefits

So what are the benefits of using rag doll behaviour and is it worth the extra computational time?

The most obvious benefit of using a physics framework and rag doll behaviour is the enhancement of the simulation. Making the use of rag doll behaviour produces much more realistic death scenes. This alone is a fair reason to use rag doll behaviour in a best-seller first person shooter game title.

The idea we have developed in this thesis to extend the rag doll framework to work in synergy with the existing animations to enhance them with recoil and impact effects on characters limbs through vertex blending has also economic benefits, as well as the feature effects.

Even though the threshold to complete the framework and to launch the first title is sure to be bigger than using standard techniques, i.e. offline animation rendering, it is sure to be economically beneficial in the long run.

If using the rag doll framework to produce new movement patterns the need for special animations are drastically reduced, thus cutting development time and costs for consecutive titles. Using traditional techniques the anticipated time to complete a full character is between three and six months for a fulltime graphics modeller.

The framework should though not be developed just for its own cause but only if there is need for it. A flight simulator has for example no real need for a rag doll framework when only using detached rigid bodies, except for maybe the ejecting scenes.

There is also the choice of licensing one of the existing third party products. Among the third party producers of physics frameworks for computer games Havok seems to have the upper hand right now, but they are sure to meet hard competition within soon.

The drawback with using a third party product is that you usually pay the license fee for each title. The possibility to make changes is also forfeited.

One could argue that using a third party product does not imply that you are locked to that product but can change framework for future titles. This should though be considered thoroughly since adapting the game development to a product usually requires some time while there still is no standard api's.

When developing a framework there is need for technically knowledgeable personnel. Right now the competence of both physics and 3D computer game development are hard to come by. But the level of knowledge concerning physic equations among game developers is constantly rising.

8.2 Future discussion

In a future can physics simulation replace character animation in computer games?

This is not so hard to imagine. In practice this would mean that the physics system also needs to handle virtual muscles. This is, as we mentioned in the introduction, something that is under research and usually referred to as dynamic animation. For dynamic animation a framework like the one we have discussed in this thesis is a necessity.

It is however not likely that we will see this in commercial use for home computers, such as computer games, for another couple of decades.

What we can expect in a near by future is the expanding use of physics simulation in home applications and computer games.

Physics have been used for quite some time in computer games such as flight simulators and other games to calculate parabolas and such. What we have not seen in any great extent is a physics framework that can handle the whole scene rather than special features.

During the last few moths Havok has been advertising that their new version of their third party physics engine will enable gamers to move and interact with virtually all the scene. They also say they will have a fully developed rag doll system for realistic death scenes. Further more they advertise that they will use vertex blending to simulate recoil and bullet impact forces into the character animations.

The first computer game to feature the new Havok2 physics system is the long awaited sequel Half Life 2, planned to be released later this year, October 2003.

References

[Akenine-Möller & Haines, 2002] Tomas Akenine-Möller, and Eric Haines. *Real-Time Rendering*. 2nd ed. A.K. Peters Ltd., 2002.

[Baraff, 1992] David Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. Ph. D thesis, Technical Report 92-1275, Computer Science Department, Cornell University, 1992.

[Baraff, 1994] David Baraff. Fast Contact Force Computation for Nonpenetrating Rigid Bodies. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 23–34. ACM Press, 1994. http://doi.acm.org/10.1145/192161.192168.

[Baraff, 1996] David Baraff. Linear-time dynamics using Lagrange multipliers. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques,* pages 137–146. ACM Press, 1996. http://doi.acm.org/10.1145/237170.237226.

[Baraff & Witkin, 2001] David Baraff, and Andrew Witkin. *Physically Based Modeling*. Online SIGGRAPH 2001 Course Notes, 2001 http://www.pixar.com/companyinfo/research/pbm2001/.

[Barr & Barzel, 1988] Alan Barr, and Ronen Barzel. A modeling system based on dynamic constraints. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques,* pages 179–188. ACM Press, 1988. http://doi.acm.org/10.1145/54852.378509

[Barr & Platt, 1988] Alan Barr, and John Platt. Constraint methods for flexible models. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques,* pages 279–288. ACM Press, 1988. http://doi.acm.org/10.1145/54852.378524.

[Barr et al., 2000] Alan Barr, Marie-Paule Cani, Gilles Debunne, and Mathieu Desbrun. Adaptive Simulation of Soft Bodies in Real-Time. *Computer Animation 2000*, pages 133144. http://www-artis.imag.fr/Publications/2000/DDCB00.

[Bruderlin & Calvert, 1989] Armin Bruderlin, and Tom Calvert. Goal-directed, dynamic animation of human walking. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques,* pages 233–242. ACM Press, 1989. http://doi.acm.org/10.1145/74333.74357.

[Dankowicz, 1999] Harry Dankowicz. *Mechanics for Computer Scientists*. KTH, Stockholm, 1999.

[Dingliana & O'Sullivan, 2001] John Dingliana, and Carol O'Sullivan. Collisions and Perception. *ACM Transactions on Graphics*, Vol. 20, No. 3, pages 151–168, 2001. http://doi.acm.org/10.1145/501786.501788

[Eberly, 2002] David Eberly. *Rotation Representations and Performance Issues.* January 2002. http://www.magic-software.com.

[Engel, 2002] Wolfgang Engel. *Introduction to Shader Programming,* Direct3D 8.1 Tutorials. June 2002. http://www.shaderx.com/direct3d.net.

[Faloutsos et al., 2001] Petros Faloutsos, Michiel van de Panne, and Demetri Terzopoulos. Composable controllers for physics-based character animation. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques,* pages 251–260. ACM Press, 2001. http://doi.acm.org/10.1145/383259.383287

[Featherstone, 1987] Roy Featherstone. *Robot Dynamics Algorithms*. Kluwer Academic Publishers, Boston/Dordrecht/Lancaster, 1987.

[Gosselin, 2002] David Gosselin. Character Animation with Direct3D Vertex Shaders, *ShaderX*, Wordware Inc., pages 196-208, June 2002.

[Halliday et al., 1997] David Halliday, Robert Resnick, and Jearl Walker. *Fundamentals of physics.* 5th ed. John Wiley & Sons, 1997.

[Hecker, 1996] Chris Hecker. Behind the Screen. *Game Developer Magazine*. Oct 1996 – Jul 1997. http://www.d6.com/users/checker/dynamics.htm

[Jakobsen, 2001] Thomas Jakobsen. Advanced Character Physics. Proceedings, Game Developers Conference 2001, San Jose, 2001. http://www.ioi.dk/Homepages/thomasj/publications/gdc2001.htm

[Laperrière et al., 1988] Richard Laperrière, Nadia Magnenat-Thalmann, and Daniel Thalmann. Joint-Dependent Local Deformations for Hand Animation and Object Grasping. *Graphics Interface '88*, pages 26-33, June 1988.

[Moore & Wilhelms, 1988] Matthew Moore, and Jane Wilhelms. Collision Detection and Response for Computer Animation. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques,* pages 289–298. ACM Press, 1988. http://doi.acm.org/10.1145/54852.378528.

[Shoemake, 1985] Ken Shoemake. Animating rotation with quaternion curves. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques,* pages 245–254. ACM Press, 1985. http://doi.acm.org/10.1145/325334.325242.

[Steed, 2002] Paul Steed. *Animating Real-Time Game Characters (Game Development Series)*. Charles River Media, 2002.

[Stejskal & Valásek, 1996] Vladimír Stejskal, and Michael Valásek. *Kinematics and dynamics of machinery*. Marcel Dekker, New York, 1996.

[Witkin et al., 1990] Andrew Witkin, Michael Gleicher, and William Welch. Interactive Dynamics. In *Proceedings of the 1990 symposium on Interactive 3D graphics,* pages 11–21. ACM Press, 1990. http://doi.acm.org/10.1145/91385.91400.