MathEngine Karma™ User Guide

March 2002.

Accompanying Karma Version 1.2.

Preface

MathEngine Karma is a physics and collision detection software package. Software libraries are provided that contain routines that users can call on to quickly and easily add physical behaviour to their 2D or 3D environment. While the Karma product is suitable for a wide range of applications users should note that it is targeted at the games and entertainment markets. This userguide provides a background to the subject and introduces the Karma package, before going on to explain in detail how Karma can be used and integrated into a project. Karma is aimed at developers of real-time entertainment simulation software who are familiar with the C programming language and have a basic knowledge of maths. Experience with Microsoft Visual C++ is an asset. The source that is provided is in C. Karma uses a C API.

In broad terms Karma consists of collision detection and dynamic simulation modules that may be used alone or together. The Karma Bridge (Mst Library) provides an API that simplifies the interoperation of Karma Dynamics (Mdt Library) and Karma Collision (Mcd Library). A basic cross platform renderer that wraps the DirectX and OpenGL graphics libraries is provided. While this allows users to build 3D applications with simple scenes, it is intended that users will integrate Karma with their own rendering solution.

Karma is available for:

- Win32 built in single precision against the Microsoft LIBC, LIBCMT or MSVCRT libraries.
- the Sony PlayStation®2 games console.
- the Xbox games console.
- Win32 double precision and Linux versions on request.

Information about each library function is provided in the html online documentation that can be found by following the ‘Demos and Manuals’ hyperlink in the index.html file in the metoolkit directory.

The origins of physical simulation, and how it has developed from the early days of electronic games through to the recent introduction of what have become known as *Physics Engines*, are discussed in chapter 1.

Chapter 2 introduces MathEngine's Karma product - a physics engine - and outlines what it can be used for, and how it works. The internal data structures are discussed, along with the data flow during a game. An example of two spheres colliding is used to demonstrate this.

Chapter 3 presents Karma Dynamics, including discussions on the units, scale, coordinate system and reference frames used. The dynamics library functions are discussed and demonstrated with explanations of relevant physical behavior and the provision of user examples - from basic user functionality through to more advanced usage of Karma's libraries. *World properties*, *bodies*, *constraints* and *forces* are introduced.

In chapter 4 an overview of collision detection is given, with an explanation of how the three main parts of Karma, namely Karma Dynamics, Karma Collision and the Karma Simulation Toolkit, work together. The distinction between collision primitives, aggregates, and static models is made and examples of their implementation provided. Determination of object intersection in a game environment is demonstrated, along with line of sight tests to determine any objects that lie on a line joining two points in 3D space. *Change blocks* are introduced and the user is told how, and (importantly) when, to update collision model information.

Chapter 5 discusses the simulation toolkit or *bridge*, which can be used at the high level to control Karma dynamics and collision.

Chapter 6 provides valuable developer information that summarizes the main do's and don'ts when using Karma. A number of important points are listed and later discussed. These will help when building simulations using Karma. Following this is a section that contains, in part, answers to questions that have been asked by developers using Karma.

The internal performance of Karma is discussed in Chapter 7, with specific information on x86 and PlayStation®2 performance provided for game relevant scenarios. Recent computational methods that are used in physical simulation are introduced - Mirtich' method, Penalty Methods and LCP-type methods. The improvements made by platform specific optimization is demonstrated and the improvements in speed through identifying platform specfic internal 'hotspots' in the software discussed.

Finally the eighth chapter comprises a user tutorial that takes a user through the steps required to build a Karma simulation of a character riding a *quadbike* over a static terrrain.

The appendices provide:

- default values of Karma properties
- a basic discussion of the Karma viewer
- an explanation of how Karma manages its memory, with example game scenarios provided to demonstrate memory allocation
- a constraint reference specification

A glossary of terms and a bibliography form the last two sections.

Typographical Conventions

Bold Face:	• User Interface element names (except for the standard OK and Cancel buttons) • Commands • Document and book titles
• `Courier`:	• Program code • Directory and file names
Italics:	• Cross-references • Introduction of a new word or a new concept

About MathEngine

Founded in Oxford, UK in 1997, MathEngine provides physical simulation software that gives developers the ability to add physical behavior to applications for use in the games and entertainment markets.

Contacting MathEngine

Head Office

MathEngine PLC, 60, St. Aldates, Oxford, UK. OX1 1ST.

Tel.+44 (0)1865 799400 Fax +44 (0)1865 799401

Web Site

www.mathengine.com

Customer Technical Support

support@mathengine.com

General inquiries

sales@mathengine.com

Introduction

This chapter provides a background to physical simulation, from a historical perspective right up to the techniques currently employed on modern day computing systems. From the empirical physical laws first formulated hundreds of years ago, to the present techniques that are based on these straightforward laws and employed in software such as the Karma package supplied by MathEngine, physical simulation is finding its way into a diverse range of applications.

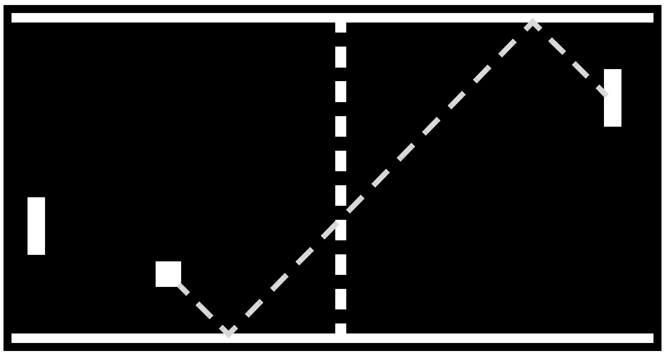
Historical Background

It is convenient to begin during the seventeenth century when Isaac Newton stated his three empirical physical laws describing the dynamical response of an object under the application of an external force. We are all introduced to these laws in secondary school, and they form the basis of the deterministic physical simulation methods that are employed today. Newtonian rigid body dynamics (RBD) has been around for a long time, and the mathematics describing such systems in the public domain. However, it is only in comparatively recent years that extensive investigation, modelling and research into simulation has been made possible by the advent of automatic calculating machines with the ability to perform the necessary calculations. The current status of this technology is such that mathematical models that describe the time evolution of physical systems of rigid bodies can be implemented on low end PCs and games consoles targeted at the home computer market.

RBD describes the behavior of a system whose elements are considered to be rigid. Rigid bodies have infinite hardness and do not deform. Even though this restriction simplifies the mathematics used to describe such systems, analytical solutions exist only for very simple systems, and thus we must turn to numerical methods. Multi-body systems are also subject to the so-called butterfly effect, in which very small changes or inaccuracies in initial configurations can build up over time into large deviations, producing widely varying behavior. For example, when breaking in a game of snooker or pool, no matter how carefully you set the game up, the slight errors in the position of the balls build up to the macroscopic level, resulting in a unique break every time.

In physical computer simulations, the inherent uncertainties in initial configuration and the limits imposed by the calculating machine mean that systems cannot be modelled exactly, only a possible evolution path predicted. MathEngine's Karma software utilises predictive methods to find such a path.

Application to entertainment



Since the early days of TV and computer games the entertainment software market has been striving to provide better entertainment solutions. Within the home user budget, increasingly more powerful computers systems are becoming available, and in turn better gaming experiences. While the improvements in graphics over the last decade are very apparent, the solutions available to improve realism are not so well known. A potential revolution may be in the making in the area of 'game / virtual world' response and realism. While things have been slow to take off - mainly because of the changes this will bring about in gaming methodology - the coming years will bring a new genre of coding tools to the hands of the games programmer in the form of methods to add physical behavior to an environment. These new tools have led to the coining of the phrase 'Physics Engine'.

Physics Engines

To explain the potential of physics engines, we need to consider the current solution to handling object behavior in a simulated game environment. This falls into two categories, namely scripted behavior (animation), and bespoke solutions:

Scripted behavior is a predefined sequence of events used to define the path through time of a game object. For example, consider a racing car rounding a bend. The way the car behaves is determined by certain properties when it comes into the bend. A scripted solution chooses a sequence from a library of pre-recorded sequences, the chosen sequence depending on the approach parameters of the car to the bend.

A bespoke solution looks at the specific problem and solves it by applying the appropriate mathematical equations to work out what a body should do. For example if a car is being driven over a terrain with the accelerator down and the accelerator pedal is released, friction between the wheels and the ground, and frictional drag caused by air resistance, are the cause of two forces that act to slow the car down. The drag force could be coded with an equation that decreases the wind resistance drag force as the car slows down.

The paradigm shift with a physics engine is that the engine implements a general mathematical model of real-world objects and their interactions. The application configures the model and defines the objects within it, and the physics engine evolves the positions and velocities of the objects over time in response to inputs from the application. So in the example above, the application would define a model for the terrain, and a model of the car, including such things as its wheels, its suspension and steering mechanisms. To simulate wind resistance on the car, the application would use an API function, perhaps of the form `SetDrag(car, drag value)`. Similarly, for a car rounding a corner, the properties of friction, velocity, drag, mass and mass distribution could be entered into the physics engine and the cars natural behavior calculated according to physical laws that act in the real world. This option can be used on its own or combined with some other solution. You can move from your scripted scene to a physics engine and back if you want to use physics to provide a more realistic approach to a certain situation.

To use a physics toolkit effectively requires the understanding of concepts such as drag, and the learning of new skills such as knowing how best to join one body to another. But the underlying mathematical functions are already there to use, which can save time prototyping and building a project. Game physics programmers will have a new set of tools at their disposal that will provide fast, stable implementations of foundational algorithms, allowing them to focus on advanced modeling techniques and game-specific features, and therefore to implement better physical behavior within their available budget. Physics providers are working to improve the speed, stability and accuracy, through improved mathematical methods, optimized code and more accurate physical modelling. This would usually be beyond the resources of a single games project.

A physics engine that solves mathematical systems of rigid articulated objects with frictional contacts can realistically simulate

- car games with obstacles
- large mechanical systems
- robots, humans and creatures
- fighting games with a large number of contacts and occasional jointed systems

Another important point to note about physics engines is that while the term 'more realism' is often used when discussing the addition of physics to games, this does not restrict development to natural behavior. A physics simulator can alter the real world behavior to give 'cartoon physics' by implementing other physical laws. Gravity can be high or low, and be made to act up or down. Cars can be light or heavy and the mass properties altered to give the required response.

Usage†

Let’s take a look at what may or may not be needed by the game developer who is just about to start work on the next game, and is considering a third party physics solution. What should a developer be considering when deciding on whether or not to invest in a physics engine? It should be noted that a physics engine might not always be the most appropriate approach. This may be the case when the amount of physics required is minimal, or the expertise exists in-house to implement a solution.

The goal of any rigid body physics engine is to calculate the motion of a collection of objects giving the impression that the objects have real world properties such as mass, and respond to real world events and conditions, such as collisions and forces. The calculated motion should also obey a number of constraints. For example, objects should not be allowed to pass through each other or the world, and when objects touch they should experience friction. Objects that are part of articulated structures, such as humans and vehicles, should move in a way that respects the joints between them.

How might one proceed? The specification requires that game objects behave realistically, and research leads to some relatively new software solutions on offer, namely physics engines. Why choose to license one of these in favor of an in-house solution? Let’s look at what needs to be considered for an in-house solution.

There are several algorithms available, but will they fulfil the specification? Featherstone's algorithm[†] is fast and can be used for chain-like structures, but problems are encountered when there are contacts or if the chain needs to be broken to add another object. Mirtich type methods[†], where one constraint at a time is satisfied are difficult to make stable, and objects suffer from jitter.

If you need to

- add and remove objects
- simulate complex articulated objects - such as chains, or structures with branches and loops
- simulate friction
- limit joint movement
- simulate motors
- handle contacts

a more general approach such as a Lagrange Multiplier method[†] is needed. MathEngine's Karma software uses such a method.

A model using such a general approach which is fast and stable, deals with stiff forces without numerical instability, and has a reasonable friction model, requires some fairly complicated mathematics that must then be carefully and efficiently implemented. Given the stringent real time gaming requirements on current low end platforms, the algorithm may well need simplifying, may not provide the complete solution, and need a lot of optimization. A collision detection system is also required to produce good predictive behavior.

There are numerous papers in the public domain[†] that discuss the problems that need to be addressed, and provide solutions in some of the areas. However, the time available to the game developer to build and optimize this basic functionality may be limited. Physics Engines provide a solution: the basic problems that a game physics programmer would need to overcome have been solved, allowing the programmer to build a solution more quickly and concentrate on adding further, better physics and features to improve playability.

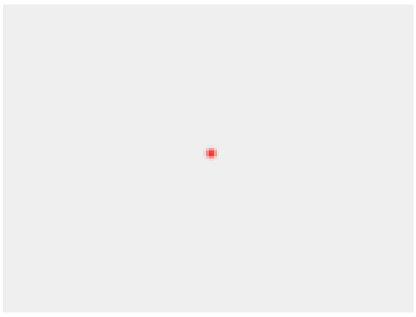
Whether you decide to go it alone or buy a package to make it easier for you, physics engines are here to stay. Current games processors are fast enough to satisfy many of the requirements of game physics, and future improvements will lead to ever better real-time simulation.

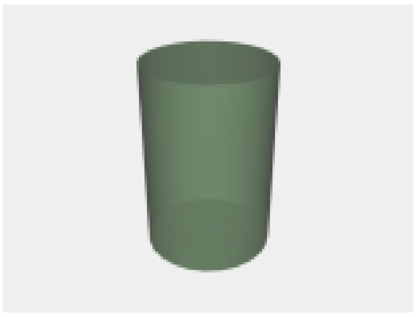
† - Please refer to the bibliography at the end of this userguide.

The Structure of Karma

Overview

It is convenient in simulation to represent a virtual world object as three separate parts, namely the dynamic body, collision frame and rendered object. These can be manipulated individually, but are more likely to be linked by using the same position and orientation information for each. The dynamic body has no knowledge of its extent save for the mass distribution of the extended shape. The collision properties of an object determine its extent in 3D space and are used to determine whether this object collides with another. An object is in collision with another object when it is not the only object to occupy a particular volume in the 3D space in which it exists. The rendered object is a visual display of what is going on in the virtual world. A physics simulation with collision detection can evolve without any graphical display. However it is often convenient for the user to render the scene to observe the behavior.


Dynamics Object


Collision Object

Render Object

Karma Simulation

At a basic level, the Karma simulation process can be described in three steps:

1. The user describes at the API level a closed system at time instant t.
2. An internal mathematical representation of the system is built by the software.
3. The system is evolved, with account taken of any new objects entering the system or additional forces arising, at a future time t'.

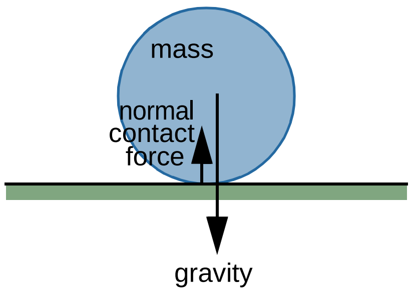
Step 1.

This involves defining:

- the objects that make up the system. Physical properties such as mass, position and velocity are specified.
- any constraints (joints or contacts) on the system.

  These are contact forces arising from two bodies interacting.
- world properties such as any fields that all objects in that world will interact with, for example a gravitational force field.

  Force fields are non-contact forces that act on objects without physically contacting them.



Constraints are of two types:

- **Equality constraints**. A particular restriction on a body, or a pair of bodies is imposed. For example, you could say that specified points on two objects must remain coincident, yet the bodies can rotate freely around one another. This constraint is a Ball and Socket joint, and the application programmer would specify it as such.
- **Inequality constraints**. As an example consider that the user has specified body fifteen in a twenty body simulation as being a solid sphere of radius ten meters. No other object can penetrate the surface of body fifteen, or occupy the same space. Mathematically this is conveniently expressed as an inequality constraint.

To maintain constraints, the system must be continually checked as it is dynamically evolved to see if any objects are colliding. The user would create collision models to describe the shapes of the bodies in the world, that the collision detection system would then use.

Contact properties such as the type and parameters of the friction model can be specified, to govern how colliding bodies behave.

Step 2.

A matrix equation is built to represent the way forces and constraints act on each body. This is used to compute the forces on each body required to maintain the constraints.

Step 3.

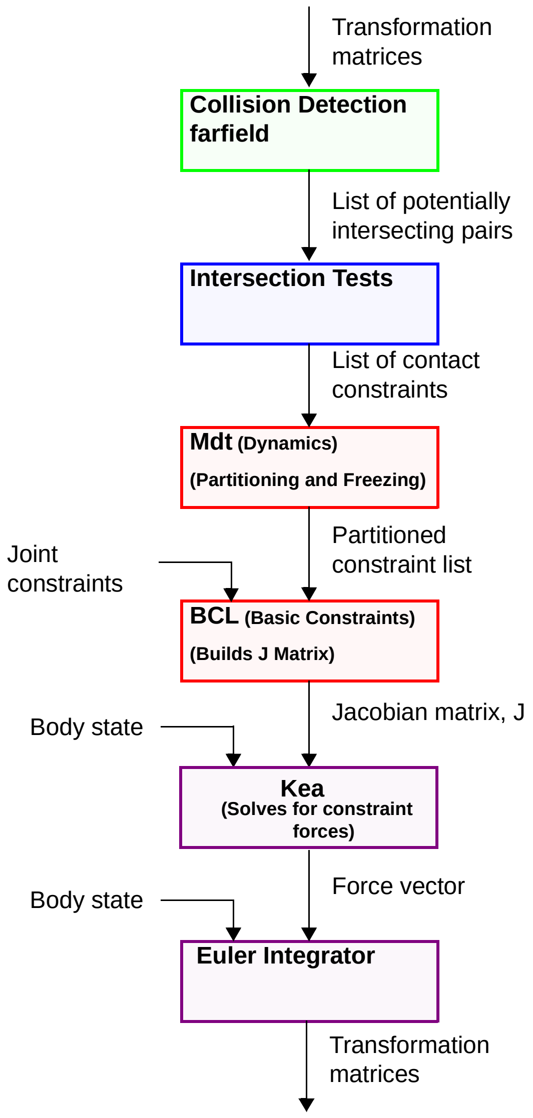
An integrator is used to move the system forward in time by some user specified increment, Δt. The requirements of this process are that:

- the system be evolved at a certain speed e.g. real time.
- the system remain stable.
- evolution is accurate from the perspective of the user.

These requirements are interdependent. For example, reducing Δt usually increases accuracy and may improve stability, with an associated increase in the time required for simulation.

The Karma Pipeline

The execution of Karma within a particular frame can be viewed as a pipeline, with each stage producing information which is used by the next. Typically the pipeline stages execute in each frame as shown in the following diagram, with rendering and user interaction being performed between runs of the pipeline.

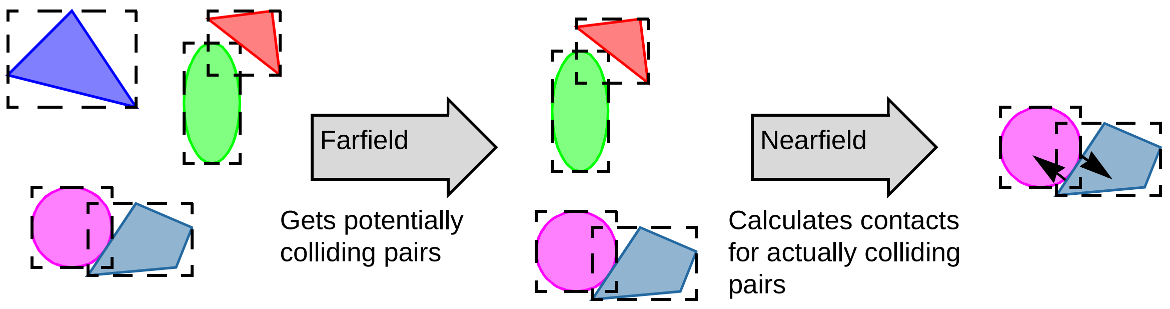


Farfield Collision Detection

The function of the farfield is to detect those pairs of objects which are nearby. In Karma, pairs of objects are considered to be nearby if their axis-aligned bounding boxes (*AABBs*) overlap.

Nearfield Tests

When the pairs of objects that might be in collision have been identified, it is the job of the nearfield tests to determine whether the objects actually do overlap. Karma dynamics is designed to respond to collisions by applying forces at a finite number of points, called *contacts*. Each contact is specified by a position, a normal and a penetration depth. If the objects intersect, the nearfield test chooses a set of contact points that best represent the intersection..


Farfield
Gets potentially colliding pairs
Nearfield
Calculates contacts for actually colliding pairs

Karma's nearfield algorithms allow arbitrary sets of triangles to be used for non-dynamic objects such as the terrain. Dynamic objects are typically represented by:

- sphere, box, cylinder or sphyl primitives.
- a collection, or *aggregate* of these primitives.
- arbitrary convex meshes.

Partitioning and Freezing

Physical simulation becomes more computationally expensive as the number of interacting objects increases. If there are ten objects in a world that are all mutually constrained, the scene is more difficult to simulate than when there are ten uncoupled objects because of all the interactions among the linked structures. Karma separates the world into groups of objects that interact among themselves, but not with objects in other groups and treats each group separately.

Each group is then examined to determine whether or not the accelerations and velocities of the objects in the group, and the forces acting on them, are all small enough that simulation will produce no perceptible change. If this is the case, all the objects in the partition are deactivated: dynamics bodies are *disabled*, and their associated collision models are *frozen.*

BCL

BCL converts Karma's high-level constraint representation, formulated in terms, for example, of hinge axes and joint positions, to a set of matrices and vectors that depend on the parameters of the constraints and the positions and velocities of the bodies they constrain.

Kea

Karma uses a Lagrange multiplier method to model jointed systems and contacts. In such a model, the effect of constraints is modeled by forces that act to maintain the constraint. In order to calculate these forces, a type of matrix problem called a *linear complementarity problem* (LCP) is solved. Karma's LCP solver is called Kea. Kea calculates forces which when applied, satisfy the constraints at the end of the time-step.

The word *complementarity* refers to the fact that related quantities can be required to satisfy a *complementarity condition*: that both are non-negative, and at least one is zero. For example, the complementarity condition for a typical contact specifies that the separation velocity and the force which maintains non-penetration both have a non-negative component in the direction of the contact normal, and at least one of them is zero. This means that at the end of the time-step either the contact will be separating and no force is being applied to enforce non-penetration, or there will be a contact force that prevents the bodies penetrating and the velocity of the objects towards each other will be zero at the contact point.

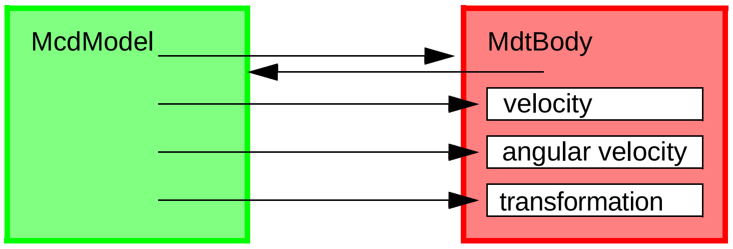
Euler Integrator

Because the forces calculated by Kea satisfy the constraints at the end of the time-step, Karma can use an Euler method to integrate the constraint forces. This provides the guaranteed stability offered by implicit methods, while taking exactly the same number of operations as an explicit Euler method. This makes Kea naturally stable, and capable of stably simulating stacks and piles of arbitrary objects without the need for user tuning or damping. External forces, such as gravity, are explicitly integrated.

Karma Data Structures

Although Karma has many datatypes, to understand the basic implementation of the pipeline it is necessary to describe just a few and outline the relationships between them. McdModels and McdModelPairs, and MdtBodys and MdtContactGroups are small datatypes, that may be created and destroyed frequently during gameplay. The MdtWorld, McdSpace, and MstBridge are large datatypes that there would usually only be one of. Naturally these would be created and destroyed infrequently, perhaps when starting or ending a game level.

McdModels and MdtBodys



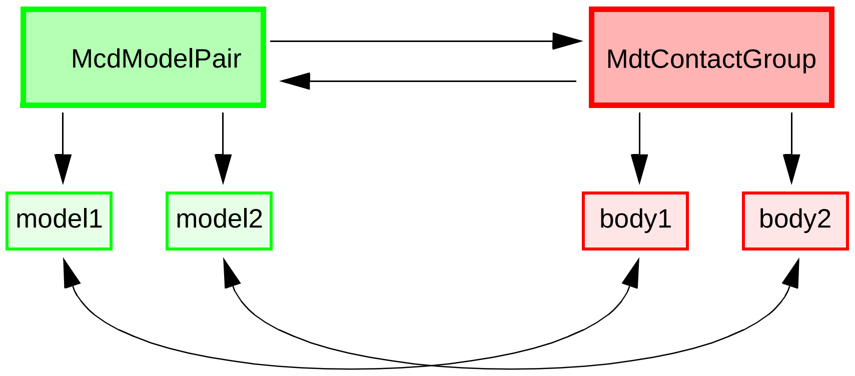
An McdModel is the top-level collision data structure containing geometric information about the extent of an object. It corresponds fairly closely to an MdtBody, which is the data structure containing the physical properties of an object, such as its mass, position, and velocity. A model may or may not have an associated body: it may represent a fixed piece of the world for which dynamic simulation is neither desirable nor necessary.

Typically there are many more objects in a Karma simulation than are being actively processed during a given time-step. The most important optimization is that when objects are not moving they have the minimum affect upon performance, which requires that they be culled from the pipeline as early as possible. This is accomplished by *freezing* collision models and *disabling* dynamics bodies that are at rest, where:

- Freezing a model informs collision that the model’s transformation matrix will not change. A frozen model will not have its AABB updated automatically by the farfield, nor will it be tested for collision against any other frozen models.
- Disabling a body instructs Karma not to include it in dynamic simulation.

If a model is frozen but has a body that is enabled, the body will be simulated, but contacts against frozen models (such as the world) will not be generated. So the body will typically fall though the world. If a body is disabled but its model is not frozen, fresh contacts will be generated for it every frame even though it is not moving. This will result in correct behavior, but with a loss of performance.

McdModelPairs and MdtContactGroups



A *McdModelPair* is a collision data structure that corresponds to a pair of models that have been detected by the farfield to be sufficiently close that they may be intersecting. McdModelPairs are created as soon as the axis-aligned bounding boxes (AABBs) of two models overlap, and thus the models possibly, but not necessarily, intersect. They persist until the AABBs cease to overlap.

If the models in an McdModelPair do intersect, the McdModelPair will have an associated *MdtContactGroup*. A MdtContactGroup is a dynamics data structure that contains a list of dynamics contacts together with dynamics information about the intersection that may be useful on subsequent frames. By contrast with McdModelPairs, MdtContactGroups are created only when the models in a McdModelPair do actually intersect. They are destroyed when the McdModelPair is destroyed. Like joints, MdtContactGroups are a type of constraint.

Since the MdtContactGroup is created from the McdModelPair, generally the order of bodies within it is inherited from the order of models within the McdModelPair, so the above diagram is representative. However, it is possible for a collision model not to have a corresponding dynamics body, if for example it represents an immovable part of the world geometry. In this case, the dynamics body is always body1 in the MdtContactGroup, and the body2 is set to NULL. Hence body1 will be pointed to by whichever of model1 and model2 corresponds to the dynamic body.

MdtWorld, McdSpace, and MstBridge

The MdtWorld is a dynamics data structure that contains all of the dynamics bodies and MdtContactGroups, along with other entities such as joints, and general data pertaining to dynamics, such as the gravitational force and threshold values for freezing partitions.

The McdSpace contains the McdModels and McdModelPairs. Its primary function is to detect which pairs of models have overlapping AABBs, and produce a McdModelPair for each. As a secondary function, in order to permit the allocation and deallocation of resources associated with colliding pairs, the space implements a simple protocol for their creation and destruction. Each McdModelPair in the space is in a state that is one of *Hello*, *Goodbye*, and *Staying*.

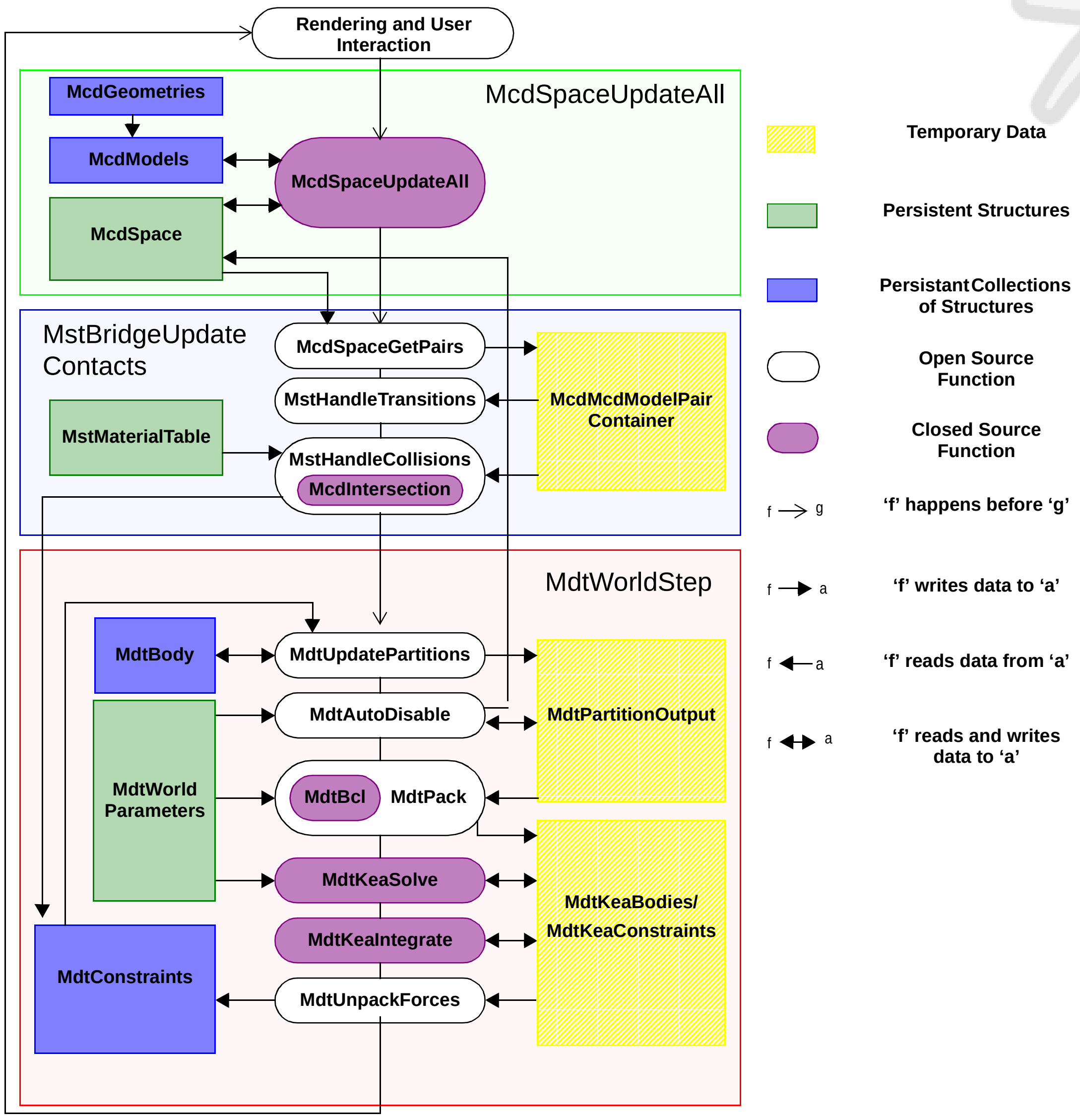
- *Hello* McdModelPairs correspond to pairs of models whose AABBs overlap at the current time-step, but did not overlap at the previous time-step.
- *Staying* McdModelPairs correspond to pairs of models whose AABBs overlap at the current and previous time-steps.
- *Goodbye* McdModelPairs correspond to pairs of models whose AABBs overlapped on the previous time-step but not on the current time-step.

Hello and Goodbye pairs are sometimes referred to as *transitional* pairs, for obvious reasons.

The API function that calculates the set of McdModelPairs on the current time-step is a function on the space.

The MstBridge contains a *material table*, which is a table that enables contacts found by nearfield tests to be augmented with dynamics contact properties such as restitution and friction parameters.

Implementation of the Pipeline



McdSpaceUpdateAll

The Collision farfield is implemented as an axis-aligned sort. It keeps track of the number of axes on which pairs of models overlap, and when this reaches the number of axes on which sorting is taking place, a McdModelPair is generated. McdSpaceUpdateAll performs this sorting operation. In order to do this, it invokes on each model that is not frozen a function to update the model's transformation matrix, and compute its AABB. The transformation matrix and AABB are then cached in the model for later use.

McdSpaceGetPairs

This function reads a fixed number of McdModelPairs from the collision farfield into a container, called a McdModelPairContainer. Typically the functions McdSpaceGetPairs, MstSpaceHandleTransitions, and MstSpaceHandleCollisions are executed repeatedly until the set of McdModelPairs in the farfield has been exhausted.

MstSpaceHandleTransitions

This routine handles the processing for each model pair in a container that is in the Hello or Goodbye state.

- For Hello pairs Karma orders the models within the pair, because nearfield tests are only implemented "one way round", that is, there is a box-sphere test but no sphere-box test. Then it determines the intersection function according to the geometry types of the two models, and caches it in the model pair.
- The only special processing performed by Karma for a Goodbye pair is to destroy an associated MdtContactGroup if one exists.

MstSpaceHandleCollisions

This function calls a nearfield test for every Hello or Staying model pair in a container where at least one of the models is not frozen. Some of the pairs may represent two moving objects colliding, but others may represent the collision of a moving object with a mesh of terrain triangles. In the latter case, a callback is executed from each nearfield test to interrogate the game's terrain format and determine the set of triangles that intersect the bounding sphere of the moving object.

The primitive-primitive intersection algorithms are fairly standard and the convex-convex intersection algorithm used is the Gilbert, Johnson, and Keerthi *(GJK)* algorithm - please refer to the bibliography. In both cases proprietary enhancements have been made to calculate penetration depth and ensure that good contact sets are made.

Each model has an associated material property, and if two models are colliding, the two materials are used to index into the material table to determine the dynamics parameters for the contact. Dynamics parameters are copied into the contact structure by value so that they may be changed by the application on a per-contact basis. The material table also contains callbacks which can be used to parameterize the contact conversion process.

Unless both models in a McdModelPair are frozen, collision contacts (and therefore dynamics contacts) are re-created every frame.

MdtUpdatePartitions

A partition is a set of bodies and constraints that can be simulated without reference to any other bodies and constraints. An alternative definition is that it is a minimal non-empty set of bodies and constraints such that if a body in the partition shares a constraint with another body, that body and constraint are also in the partition. Partitions are generated by iterating over the set of enabled bodies that are not yet in a partition, and adding new bodies by performing a breadth first search of constraints. If a disabled body is found connected by a constraint to an enabled body, it is automatically enabled. Enabling the body causes a routine to be invoked in the bridge to unfreeze the corresponding collision model.

It is possible to set a threshold on the number and complexity of a partition, in order to trade simulation fidelity for speed. If such a threshold is set, the partition may be automatically simplified at this point in order to bring it down to the required size.

MdtAutoDisable

Karma examines each partition after it has been generated to see if the velocities of objects are sufficiently small that simulation will produce no perceptible change, and if so, disables all the bodies. Disabling a body causes a routine to be invoked in the bridge which freezes the corresponding collision model.

MdtPack

MdtPack iterates over the set of partitions, calling the appropriate routine in BCL for each constraint in the partition. The BCL routine computes the matrix of partial derivatives for the constraint, as well as upper and lower bounds for constraints such as force-limited motors, friction limits on contacts, and movement limits on joints.

MdtKeaSolve

Kea, Karma's LCP solver, contains the highly optimized per-platform implementations of the matrix and vector routine required to solve an LCP.

Kea's constraint formulation is based upon a semi-implicit time-stepping method. Its friction model and degeneracy protection produce symmetric positive definite LCPs that are guaranteed to have a solution. It uses an iterative procedure to generate successive candidates that should be closer approximations to the correct solution, and usually finds the correct solution in a small number of iterations.

However, there is no general procedure for efficiently solving sets of inequality and complementarity constraints: in the worst theoretical case, finding the correct solution requires time exponential in the number of inequality constraints. Physical simulations do not give rise to such pathological LCPs, but it is important nonetheless to have the option to trade simulation fidelity for speed. There are two possible mechanisms that a Karma application can use to accomplish this:

- relax the constraint conditions, so that a generated solution is more likely to satisfy the constraints.
- limit the maximum number of iterations.

Either of these mechanisms will result in a loss of fidelity in the simulation. But they offer the option of reducing the execution time spent in Kea, based on the degree of constraint violations and other non-physical artifacts that the application can tolerate.

MdtKeaIntegrate

MdtKeaIntegrate invokes Karma's Euler integrator to evolve the positions and velocities of bodies using the forces calculated in MdtKeaSolve.

MdtUnpackForces

Sometimes it is useful to be able to access the constraint forces computed by Kea, for example when using a friction model that takes as input the normal force at the previous time-step. MdtUnpackForces extracts the constraint forces for each constraint from Kea's internal data representation, and writes them into the constraint structures.

High Level API

The function MstBridgeUpdateContacts invokes the following functions repeatedly until all pairs in the space have been processed:

- McdSpaceGetPairs
- MstBridgeUpdateTransitions
- MstBridgeUpdateCollisions

The function MdtWorldStep invokes the following functions (some indirectly):

- MdtWorldUpdatePartitions
- MdtAutoDisable
- MdtPack
- MdtKeaSolve
- MdtKeaIntegrate

So the application code to invoke the pipeline can be as simple as

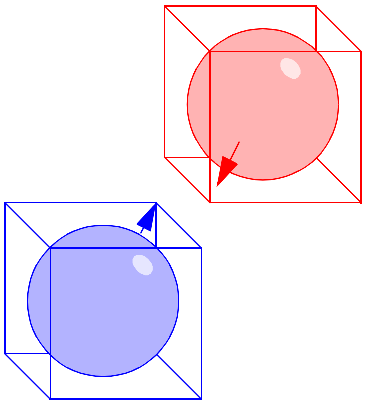
```
McdSpaceUpdateAll(space);
MstBridgeUpdateContacts(bridge, space, world);
MdtWorldStep(world,stepSize);
```

If you are using the MstUniverse API, which provide a convenient wrapper around some of the dynamics and collision code, the API function MstUniverseStep invokes exactly this sequence of functions.

Two Spheres Colliding

This example of two spheres moving towards each other, colliding, and rebounding again, illustrates a simple example of the pipeline in action.

AABBs do not overlap.



McdSpaceUpdateAll updates the AABBs for each of the models because neither is frozen. Since the bounding boxes do not overlap, no McdModelPair is created.

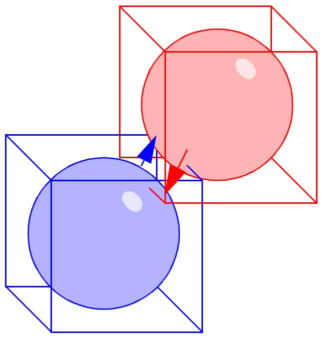
MstBridgeUpdateContacts:

> **McdSpaceGetPairs** leaves the McdModelPairContainer empty because there are no McdModelPairs in the space.
>
> **MstHandleTransitions** does nothing, because there are no Hello or Goodbye pairs.
>
> **MstHandleCollisions** does nothing, because there are no Hello or Staying pairs.

MdtWorldStep:

> **MdtUpdatePartitions** creates a partition for each sphere, since there is no constraint between them.
>
> **MdtAutoDisable** examines each partition, but because both of the bodies velocities are above the threshold, both remain enabled.
>
> **MdtPack** does nothing, as there are no constraints.
>
> **MdtKeaSolve** does nothing.
>
> **MdtKeaIntegrate** moves the bodies as a result of the current velocities, since there are no forces and therefore no accelerations.
>
> **MdtUnpackForces** does nothing.

AABBs overlap, but the objects do not.



McdSpaceUpdateAll updates the AABBs of each of the models from the previous frame. Now that the AABBs are overlapping, a McdModelPair is created and classified as a Hello pair.

MstBridgeUpdateContacts:

McdSpaceGetPairs reads the McdModelPair into the McdModelPairContainer.

MstHandleTransitions reads the pair from the ModelPairContainer, looks up the intersection function to test for intersection between two spheres, and stores it in the McdModelPair. Because both models have the same geometry, there is no need to order them appropriately for input to the intersection test.

MstHandleCollisions calls the intersection test and deduces that the spheres do not intersect, so no contacts are produced.

MdtWorldStep:

MdtUpdatePartitions creates a partition for each sphere.

MdtAutoDisable examines the partitions, and leaves both bodies enabled.

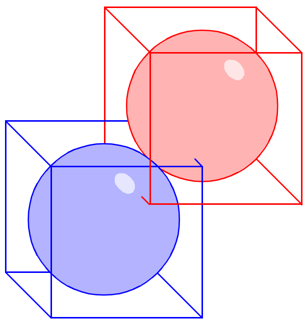
MdtPack does nothing.

MdtKeaSolve does nothing.

MdtKeaIntegrate moves the objects according to their current velocities.

MdtUnpackForces does nothing.

The AABBs overlap and the spheres touch.



McdSpaceUpdateAll updates the AABBs of the spheres. The AABBs are still overlapping so the pair is classified as a Staying pair.

MstBridgeUpdateContacts:

McdSpaceGetPairs reads the McdModelPair into the McdModelPairContainer.

MstHandleTransitions does nothing, since there are no Hello or Goodbye pairs.

MstHandleCollisions calls the intersection test for the sphere-sphere contact, and this time determines that the spheres are touching. It creates a MdtContactGroup, and inserts into it an MdtContact containing the position, normal and penetration distance for the contact point together with the friction and restitution parameters.

MdtWorldStep:

MdtUpdatePartions creates just one partition, since the bodies are now constrained by a contact.

MdtAutoDisable examines the partition, but because both spheres have velocities above the threshold velocities, the bodies remain enabled.

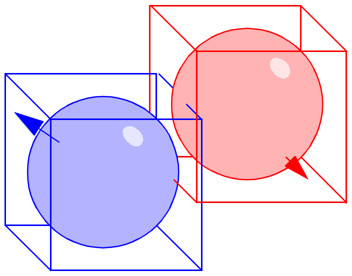
MdtPack finds the function in the basic constraint library (**MdtBcl**) that processes MdtContactGroups. This function computes the appropriate input data to Kea based on the position, normal, and penetration of the contact, the positions and velocities of the spheres, and the restitution and friction parameters.

MdtKeaSolve calculates the force required to prevent the spheres penetrating, and to generate the required restitution velocity. It also calculates the accelerations produced by those forces.

MdtKeaIntegrate uses these accelerations to update the velocities of the spheres, and uses the new velocities to update the spheres' positions.

MdtUnpackForces unpacks the forces from Kea's internal data structures into the MdtContactGroup.

The AABBs overlap, but the spheres are not touching as they move away.



McdSpaceUpdateAll updates the AABBs for each model from the previous frame. The AABBs are still overlapping, so the pair is still classified as Staying.

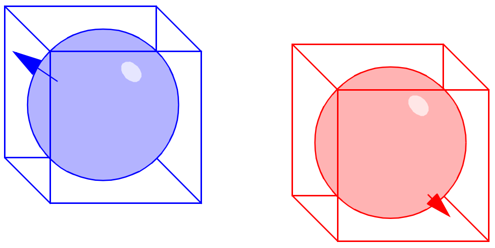
MstBridgeUpdateContacts:

> **McdSpaceGetPairs** reads the McdModelPair into the McdModelPairContainer.
>
> **MstHandleTransitions** does nothing, since there are no Hello or Goodbye pairs.
>
> **MstHandleCollisions** calls the intersection test for the sphere-sphere contact. There is now no contact between the two spheres and so the contact is removed from the MdtContactGroup.

MdtWorldStep:

> **MdtUpdatePartitions** creates a partition for each sphere.
>
> **MdtAutoDisable** leaves the spheres enabled.
>
> **MdtPack** does nothing.
>
> **MdtKeaSolve** does nothing.
>
> **MdtKeaIntegrate** moves the objects according to their current velocities.
>
> **MdtUnpackForces** does nothing.

AABBs do not overlap as the spheres move away from each other.



McdSpaceUpdateAll updates the AABB for each sphere from the previous frame. Now the AABBs are not overlapping, so the McdMcdModelPair becomes a Goodbye pair.

MstBridgeUpdateContacts:

> **McdSpaceGetPairs** reads the McdModelPair into the McdModelPairContainer.
>
> **MstHandleTransitions** reads the pair from the McdModelPair container and, because it is a Goodbye pair, destroys the McdModelPair’s MdtContactGroup.
>
> **MstHandleCollisions** does nothing, as the only McdModelPair is a Goodbye pair.

MdtWorldStep:

> **MdtUpdatePartitions** leaves each sphere in its own partition.
>
> **MdtAutoDisable** leaves each sphere enabled.
>
> **MdtPack** does nothing.
>
> **MdtKeaSolve** does nothing.

MdtKeaIntegrate moves the objects according to their current velocities.

MdtUnpackForces does nothing.

NOTE: The workings of MdtWorldStep are exactly the same for stages 1, 2, 4 and 5 because the spheres are not touching in these cases.

Dynamics

Conventions

Units and Scaling

There is no built-in system of units in Karma, which is not to say that quantities are dimensionless. Any system of units may be chosen, either meter-kilogram-seconds, centimeter-grams-seconds or foot-pound-seconds. However, the programmer is responsible for the consistency of values and dimensions used.

The range of masses and lengths within which Karma will perform well is limited by the precision of the underlying floating point implementation: single precision floating point (the default build option for Karma on all platforms) provides for approximately six significant decimal digits. In order to support a variety of scales of length and mass, Karma requires the application to supply default values of mass and length.

For example, if a human is a medium-sized object in your application, with height 1000 units, and mass 4 units, 1000 is a reasonable value for default length, and 4 a reasonable value for default mass. Input masses and lengths should not be enormously different to the defaults: about two orders of magnitude either way is reasonable in single precision. Karma scales all its internal tolerance and threshold values in accordance with the values you supply, but if you override any of these defaults, you too need to scale your values appropriately. Time is assumed to be in seconds, and angles measured in radians. If your time and angle units are different from this Karma's defaults will not be appropriate, and you will need to rescale some of the default values yourself in order to get the best behaviour from Karma. (The units for the default parameters to the world are given in appendix A)

If, for example, you are working in degrees for angular measurement, you will need to scale Karma's default angular velocity and acceleration thresholds for auto-disabling by $180/\pi$. And if the time system with which you wish to use Karma advances the simulation by 1 unit of time for $1/60$ s, you should scale all Karma's thresholds for autodisabling: velocities by $1/60$, and accelerations by $1/3600$. If

Coordinate Systems and Reference Frames

Karma uses rectangular Cartesian coordinates for virtually all vector quantities. Karma does not use polar coordinates. The components of any 3-dimensional vector are the projections along the x, y, and z axes. Those axes are orthogonal to each other and the scale along each one is identical. Right-handed coordinate systems are used everywhere except in the Karma Viewer.

The reference frame used most is called alternately the world, global, inertial or Newtonian reference frame. These are all names for the same thing: a fixed reference frame which defines the overall orientation and position of the scene being worked on. This is the common frame of reference for all the objects in an MdtWorld or an MstUniverse.

There are other useful reference frames however. Each rigid body has defined a reference frame centered at its center of mass. Points on a rigid body, such as the attachment positions for joints, are naturally expressed in terms of a fixed location on the rigid body. Note that the joint position is given in the world reference frame in Karma. As the rigid body moves under the influence of applied forces, that attachment point will move as seen from the origin of the world reference frame.

Since a 3D model is designed independently of mass properties of the objects it is based on, an artist is likely to pick a convenient origin that is not the center of mass of the object. This reference frame is called the model reference frame. A relative transformation between this reference frame and the rigid body center of mass reference frame can be used to simplify manipulation of geometric models.

The relationship between reference frames consists of a translation and a rotation. If the transformation from a frame F to a frame G is specified by the translation vector X and the rotation matrix R, then a point whose coordinates in the frame F are specified by the vector x will have coordinates X + Rx in the frame G.

As is customary in well-known 3D graphics API's, a transformation between reference frames is conveniently stored in a 4 by 4 affine matrix containing a 3x3 rotation submatrix $R = r_{ij}$ and a translation vector $t = (t_x, t_y, t_z)$.

The MeMatrix4 and MeVector4 data types implement matrices and vectors for use in this way. In this representation, regular 3D vectors are represented as the first three components of the four dimensional vectors. The last element in the 4 dimensional vector must be set to 1 if the translation is to be taken into account and 0 if one is only interested in the rotation.

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

Newcomers to this sort of algebra may like to consult a textbook on 3D graphics to familiarize themselves with the concepts and notations. Utility functions are provided to convert vectors from one reference frame to another.

Type Conventions

Karma uses some special type definitions and macros that make it more portable. These are defined in `MePrecision.h.` For example:

- `MeReal`: floating point numbers
- `MeVector3`: a vector of 3 `MeReals`
- `MeVector4`: a vector of 4 `MeReals`
- `MeMatrix3`: a matrix of 3 `MeVector3s`
- `MeMatrix4`: a matrix of 4 `MeVector4s`
- `MeSqrt()`: `sqrt()`

Naming Conventions for C Identifiers

`Me`	MathEngine types and macros for controlling precision
`Mdt`	Karma Dynamics
`MdtBcl`	Basic Constraint Library
`MdtKea`	Kea Solver
`Mcd`	Karma Collision
`Mst`	Karma Simulation Toolkit
`R`	Karma Viewer

Calling Conventions

MEAPI is defined in MeCall.h and defines the calling convention used by API functions.

MEPUBLIC is defined in MeCall.h. It expands as appropriate to __declspec(dllexport), __declspec(dllimport)), or does nothing.

The MathEngine Dynamics Toolkit (Mdt)

The Mdt Library contains functions that use physical data to mathematically simulate a physical situation. Real world information such as mass, friction, and gravity can be specified by the user. Because these simulations take place in a virtual world, the values can be changed by the user to obtain the given behavior, such as gravity acting upwards, or unphysically heavy objects.

The dynamics library can be used to create rigid bodies and articulated bodies corresponding to the models (people, robots, vehicles, and other objects) used in games or other 3D simulations. The abstract space occupied by these bodies is called a *world*. This world typically corresponds to the scene described by the rendering software and to the *space* occupied by any collision models.

The *world* is a collection of data structures. More than one world can be used in an application, perhaps corresponding to different regions / levels in a game. Different worlds are totally disconnected from each other and may never interact. Any interaction would be handled by the programmer. A world may be partitioned into smaller groups of interacting rigid bodies, where rigid bodies can move from partition to partition

This world may contain rigid bodies that interact via forces, constraints, and input and output signals. The bodies may have properties assigned to them that can be used to determine the interaction mechanism. An input signal enables the user to interact with objects in the world. By reading the:

- (x, y) coordinates of the mouse pointer,
- angle signal from a joystick,
- or input from a steering wheel or special pedal,

the user can reposition or apply forces to scene objects.

Similarly, an output signal may correspond to the orientation of one of the scene objects. This could be sent to a force feedback device or a motion platform.

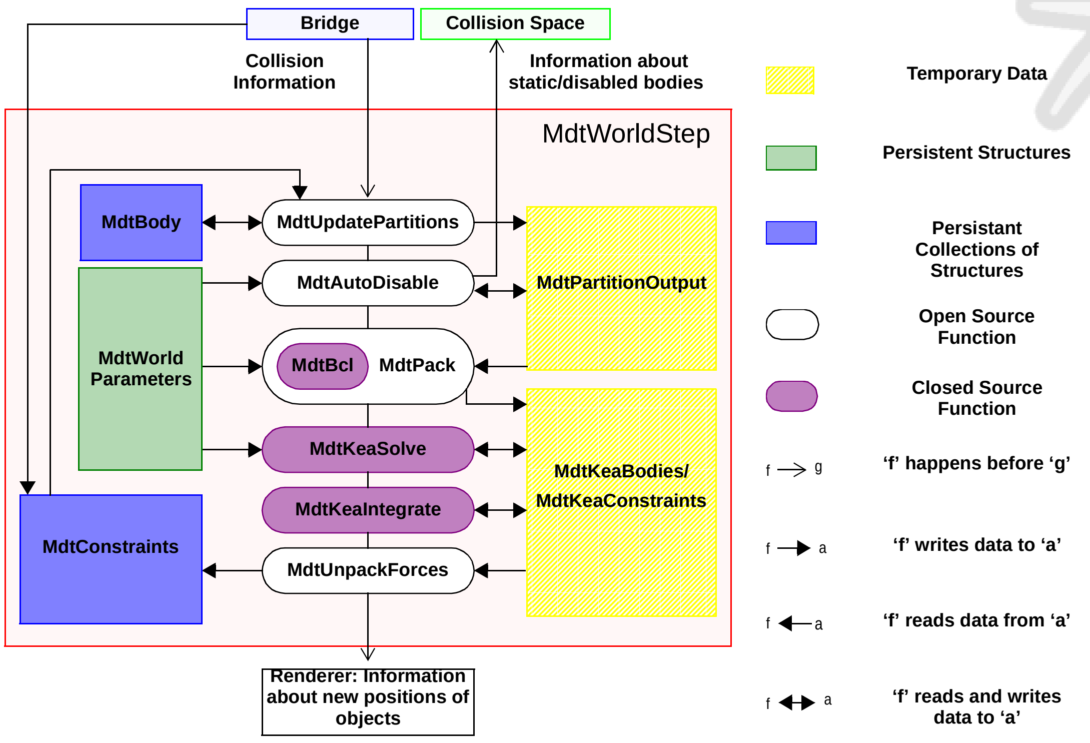
An `MdtWorld` structure is a container that tracks the `MdtBody` structures along with their joint and contact constraints. It also stores world (global) properties such as the gravitational field strength.

A constraint - which may conveniently be one of two types - restricts the motion of a rigid body:

- A *joint* between two bodies, such as a hinge to represent the elbow of a virtual human, is a type of constraint. This constraint restricts at least one of the six degrees of freedom of the attached pair of bodies (or a body attached directly to the world). The six degrees of freedom comprise three linear and three rotational degrees of freedom. In 3 dimensional space, the 3 linear degrees of freedom might correspond to the movement along each of the 3 Cartesian axes, while the 3 rotational degrees of freedom relate to rotation around each of these axes.
- A *contact* is a type of constraint that prevents objects inter-penetrating. No permanent restriction is placed on any of the six degrees of freedom of the contacting objects.

The Mdt Library is the Karma Dynamics top level library. If the Mdt Library is used to build a simulation, then the lower level MdtBcl Basic Constraint Library does not need to be called directly, although it must still be linked in. The source code of the Mdt Library can be used as a guide to using the lower level libraries. For developers using Karma Collision in addition to Karma Dynamics, the Mst library provides tools to integrate both of these into a simulation library.

.



Designing Efficient Simulations Using the Mdt Source Code

Karma Dynamics is distributed with full source code for the Mdt Library. Hence applications can be optimized by, for example:

- Selecting only the required parts of the Mdt Library.
- Customizing the Mdt code.
- Creating new joint types, based upon Mdt joints.

Designing Efficient Simulations Using the Mdt Library

There are lots of ways of making your simulation go faster using the provided library functions. For example:

- Choice of friction model.
- Partitioning of objects in a world.
- Disabling bodies.
- Keeping the constraint matrix size down.
- Level of detail representations.

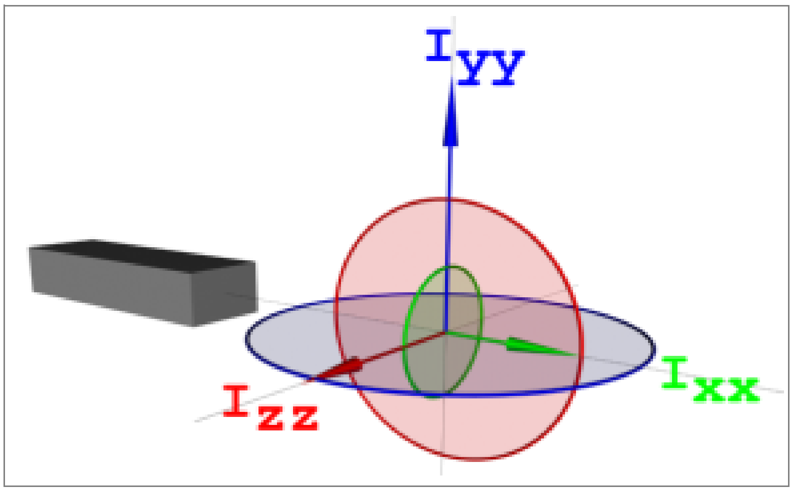
Rigid Bodies: A Simplified Description of Reality

All the objects in the real world have finite extent and are therefore not well simulated using the 'point mass model' of classical physics. The rigid body is an idealization of objects that do not deform easily. It represents objects that have finite extent but that never deform at all; any two points on the objects are always exactly the same distance apart, no matter what forces and stresses are applied to the body. In the real world, objects are not infinitely hard or rigid since they all deform to some extent - even diamond. Having said that, infinite rigidity provides a good approximation when modelling the dynamics of solid objects. The assumption of infinite hardness allows faster simulation speeds to be realized, and hence this approximation is used by Karma.

Rigid bodies have real world physical properties such as mass and moment of inertia that are used by Karma. This makes them easy to simulate, but some care must be taken when ascribing parameters such as the inertia tensor, which gives a mathematical description of how easy it is to turn an object about its axes. The inertial tensor of a body, `I`, is represented by a 3 by 3 matrix, where the off-diagonal components of the matrix are usually zero, and the diagonal components I_{xx}, I_{yy} and I_{zz} give the moments about the principle x, y and z axes.

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

I_{xx}, I_{yy} and I_{zz} determine how hard it is to rotate an object about a principle axis. For example, this diagram gives a schematic pictorial representation of the magnitude of these moments for a box. The box would rotate most easily about the x axis. To represent this we can picture a disk with its axis along the x axis, the smaller the disk is, the easier it is to spin. This is because small disks have a small moment of inertia, much like a figure skater spinning more easily (faster) when their arms are drawn in, but the more extended the skater is (arms out), the higher the moment of inertia, and the slower they spin.



Rotation about the y (blue) axis is much harder than about x, as represented by the larger associated disk, while the z axis is somewhere in-between. The sizes of these disks correspond approximately to the inertial tensor components along the individual axes, with the component I_{xx} being the smallest, followed by I_{zz} and I_{yy}.

Friction and restitution are properties that describes how bodies in contact interact. Karma does not model all physical phenomena directly and properties such as viscous drag coefficient (related to the geometry of the body), electric charge, magnetic dipole etc., can be modelled by the developer to introduce more interesting behavior based on the existing API.

There are no dynamic properties that describe a body's shape, i.e. its geometry. The only information that a dynamic body has of its extent is its mass distribution. For simulations that do not involve collisions, knowledge of geometry is not required. For example, it is not necessary to know anything about an object's shape in order to simulate it moving freely under gravity. A body does need to have a specific geometric shape assigned to it to determine whether it has collided or come into contact with another body. And even when specific geometry information is not needed, bodies should be assigned inertia tensors that are appropriate to their geometry and mass. Failing to set inertia tensors properly can cause non-physical behavior.

Rigid bodies have kinematic attributes that describe their position and movement, such as:

- Position of the center of mass
- Orientation of the body.
- Velocity of the center of mass
- Angular velocity, which describes the rate of change of orientation

There are several possible ways to represent the orientation of bodies, such as Euler angles, rotation matrices, or quaternions. Karma uses quaternions internally, although API functions are provided to read and write the orientations of bodies as rotation matrices.

Rigid bodies have dynamic attributes that consist of net applied forces and torques.

These properties can be set to appropriate values using Karma Dynamics, and objects moved by the application of forces such as gravitational field strength, and impact with another object.

A brief description of the program flow is:

- Set the initial (at time $t_{Init} = 0$) world, object and joint properties.
- Specify any forces or contacts at t_{Init}.
- Set the simulation time-step, Δt_S, i.e. the time amount by which the simulation should be repeatedly incremented. A typical value might be Δt_S = 1/60 of a second, corresponding to a 60Hz frame rate.
- Karma Dynamics then calculates the new positions at ($t_{Init} + \Delta t_S$) and the other dynamical properties for each body in the world according to the underlying newtonian physical model. This is called *solving*. The Karma Dynamics *solver* is called Kea.
- The positions, orientation and so forth of the models in the scene are updated, based on the data returned by Karma Dynamics. This renderer updates the scene from this information.
- The values of the positions etc at $t_{Init} + \Delta t_S$ are used to calculate the positions and orientations at the next time-step $t_{Init} + 2\Delta t_S$.
- The sequence continues (t_{Init}, $t_{Init} + \Delta t_S$, $t_{Init} + 2\Delta t_S$, $t_{Init} + 3\Delta t_S$, ...) whereby values at the previous time-step are used to calculate those at the current time-step.

Karma Dynamics - Basic Usage

The following section discusses some of the common basic procedures that a program that uses Karma Dynamics will include.

Creating the Dynamics World

The function:

```
MdtWorldID MEAPI MdtWorldCreate      ( const unsigned int maxBodies,
                                       const unsigned int maxConstraints,
                                       const MeReal lengthScale,
                                       const MeReal massScale )
```

is used to create and initialize the dynamics world where

- `maxBodies` is the maximum number of bodies
- `maxConstraints` is the maximum number of constraints

in the world.

- `lengthScale` is a typical value for the length of an object in your simulation.
- `massScale` is a typical value for the mass of an object in your simulation.

An `MdtWorldID` identifier is returned. This points to an `MdtWorld` structure that identifies the world that has been created. Most of the functions used to manage world properties are prefixed by `MdtWorld`.

Setting World Gravity

The gravity (gravitational field strength) for a virtual world may be set using:

```
void MEAPI MdtWorldSetGravity   ( const MdtWorldID world,
                                  const MeReal gx,
                                  const MeReal gy,
                                  const MeReal gz )
```

Where `gx`, `gy` and `gz` are the components of the gravity vector. For a close approximation to earth surface gravity the appropriate values would be:

```
MdtWorldSetGravity(world, 0, -(MeReal)(9.81), 0);
```

The gravity is set along the negative y axis to simulate earth’s gravity value, while the earth surface is conveniently represented by the x and z axis.

While we perceive gravity as acting in a “down” direction, in a virtual world it may be set along any direction and with any magnitude.

Defining a Body

The following defines and creates a dynamic body and places it in the world:

```
MdtBodyID body;
body = MdtBodyCreate(world);
```

The dynamic body would normally correspond to a rendered graphical object such as a sphere, cube, or more complex shape. A dynamic body is represented internally by a structure storing the following physical properties:

MdtBody	**Members Default Value**
mass	1
moment of inertia	{ {0.4,0,0}, {0,0.4,0}, {0,0,0.4} }
position	{0,0,0}
quaternion	{1,0,0,0}

transformation matrix	{ {1,0,0,0}, {0,1,0,0}, {0,0,1,0}, {0,0,0,1}}
fast spin axis	{0,1,0} (fast spin is disabled by default)
force applied	{0,0,0}
torque applied	{0,0,0}
velocity	{0,0,0}
angular velocity	{0,0,0}
acceleration	{0,0,0}
angular acceleration	{0,0,0}
velocity damping	0
angular vel damping	0

The function:

```
MdtBodyID MEAPI MdtBodyCreate  ( const MdtWorldID world )
```

creates a body structure containing the above default parameters.

The default is that newly created bodies are not included in the simulation i.e. they are disabled. The body must be enabled in order for it to be included in the simulation. The function:

```
void MEAPI MdtBodyEnable       ( const MdtBodyID body )
```

enables the body.

Note that bodies are automatically enabled when hit by other enabled bodies.

There are a large number of *mutator* functions to change the value of members of the MdtBody structure. Common examples are such functions as:

```
void MEAPI MdtBodySetPosition        ( const MdtBodyID b,
                                       const MeReal x,
                                       const MeReal y,
                                       const MeReal z )
void MEAPI MdtBodySetLinearVelocity  ( const MdtBodyID b,
                                       const MeReal dx,
                                       const MeReal dy,
                                       const MeReal dz )
void MEAPI MdtBodySetAngularVelocity ( const MdtBodyID b,
                                       const MeReal wx,
                                       const MeReal wy,
                                       const MeReal wz )
void MEAPI MdtBodySetQuaternion      ( const MdtBodyID b,
                                       const MeReal qw,
                                       const MeReal qx,
                                       const MeReal qy,
                                       const MeReal qz )
```

The corresponding functions that read structure member values are called *accessor* functions. The corresponding accessors that read the values of the aboves are:

```
void MEAPI MdtBodyGetPosition        ( const MdtBodyID b,
                                       MeVector3 p)
void MEAPI MdtBodyGetLinearVelocity  (const MdtBodyID b,
                                       MeVector3 v)
void MEAPI MdtBodyGetAngularVelocity (const MdtBodyID b,
                                       MeVector3 v)
void MEAPI MdtBodyGetQuaternion      (const MdtBodyID b,
                                       MeVector4 q)
```

Not all of the structure members that can be read can be modified.

The mutators and accessors for any `MdtBody` structure member all use the `MdtBodySet` and `MdtBodyGet` prefixes respectively. A complete list of accessors and mutators can be found in the HTML *Karma API Reference.*

Evolving the Simulation

The simulation is evolved forward by the increment `step` using the function:

```
MdtWorldStep(world, step);
```

where a typical value for `step` might be 1/60s, which is a reasonable game refresh rate.

The formal definition of `MdtWorldStep is:`

```
void MEAPI MdtWorldStep(const MdtWorldID world, const MeReal stepSize);
```

`MdtWorldStep` partitions the world and uses the Kea solver to work out the forces required to maintain constraints. The integrator then evolves the world, calculating the new positions and velocities of each object in the world.

Cleaning Up

The API function

```
void MEAPI MdtWorldDestroy     (const MdtWorldID world)
```

frees the dynamics memory at the end of the simulation. It destroys an `MdtWorld` and all bodies and constraints contained in it.

Evolving a Simulation - Using the Basic Viewer Supplied with Karma

The renderer accompanying Karma is a straightforward, minimal functionality, renderer that wraps DirectX and OpenGL. It is not meant to be a rendering solution, but rather provide a means of visualizing the virtual world being evolved using Karma dynamics and collision. It is intended that developers will have their own rendering solution that they will interface with Karma using the position and orientation information calculated by dynamics. Further information on the renderer can be found in Appendix B.

The callback function `void MEAPI Tick(RRender* rc)` is called from main through the render callback function call `RRun(rc, Tick, 0)`. The viewer calls the `Tick()` callback function in its main loop that steps the simulation forward by the set time-step. The viewer `RRun()` does the following:

```
while no exit-request
{
     Handle user input
     call Tick() to evolve the simulation and update graphic transforms
     Draw graphics
}
```

The Tick function may be used to step through a simulation by inserting the function `MdtWorldStep(world, step)` into it to evolve the simulation by one time-step for each call of `void Tick(RRender* rc)`.

```
void MEAPI Tick(RRender* rc)
{
     /* Dynamics code to evolve the world */
     MdtWorldStep(world, step);
}
```

The choice of time-step should meet the simulation requirements. If the time-step is too large, a simulation may become unrealistic. For example, a falling ball can pass through a floor of finite thickness if its position is not determined at small enough intervals i.e. the contact is missed. If the time-step is too small, the simulation may run at an unacceptably slow speed. After stepping forward in time, some or all of the enabled bodies may have moved to new positions, and some of the properties may have changed.

The cleanup routine is placed in the callback function specified in `atexit`. This must be used because some graphics APIs on which the renderer is built do not exit from the evolve loop cleanly to execute subsequent code.

```
void MEAPI_CDECL cleanup(void)
{
     /* Cleanup code including the dynamics cleanup code.*/
}
main()
{
     /* Program code. */

     /* Set the callback to cleanup properly. */
     atexit(cleanup);
     }
```

The Basic Steps Involved in Creating a Karma Dynamics Program

The following example demonstrates all of the basic concepts discussed to put together a working dynamics demo. The demo simulates a sphere moving under gravity. Each line contains an explanatory comment.

```
/***************************\
* Karma Dynamics Basic Steps *
\***************************/

/* Will need the dynamics and the viewer library for this example */
#include "Mdt.h"
#include "MeViewer.h"

/* The following 3 functions are used. */
/* The standard Karma calling convention, MEAPI, is __stdcall */
int MEAPI_CDECL main(int, const char**);
void MEAPI_CDECL cleanup(void);
void MEAPI tick(RRender*, void*);

/* Karma dynamics variable world_ is a pointer to an MdtWorld struct */
MdtWorldID world_;
/* sphereD_ is a pointer to an MdtBody struct */
MdtBodyID sphereD_;

/* Karma renderer structures */
RRender* rc;
RGraphic* sphereG_;
/* The following struct contains command line parameters passed to the program */
MeCommandLineOptions *opts;

void MEAPI_CDECL cleanup(void)
{
     /* Deallocate memory assigned to the body */
     MdtBodyDestroy(sphereD_);
     /* Destroying the world deallocates memory assigned to all bodies and
        constraints contained in it.  The body destroy above is not critical */
     MdtWorldDestroy(world_);
     /* Clean the graphics memory */
     RRenderContextDestroy(rc);
}

void MEAPI tick(RRender *rc, void *userdata)
{
     static int count = 0;
     /* exit the simulation after 600 time-steps */
     if(++count==600) exit(0);
     /* Evolve the dynamics world by 0.01 seconds */
     MdtWorldStep(world_, 0.01f);
}

int MEAPI_CDECL main(int argc, const char *argv[])
{
     /* Define the color red for the body. */
     /* The parameters passed to red correspond to
        Red Green Blue and Alpha (how opaque the object is) */
     MeReal red[] = {1, 0, 0, 0};
     opts = MeCommandLineOptionsCreate(argc, argv);
     rc = RRenderContextCreate(opts, 0, 1);

     /* Create a dynamics world with unit sizes. */
     /* The corresponding structure allocates
        enough memory for one body and no constraints */
     world_ = MdtWorldCreate(1, 0, 1, 1);
```

```
        /* Set the world property gravity to act with
           equal strength along the -x and -y directions */
        MdtWorldSetGravity(world_, (MeReal)-0.3, (MeReal)-0.3, 0);

        /* Make a body and add it to the dynamics world world_. */
        /* While this dynamics body has been called sphereD_,
           it does not yet have any knowledge about it's shape */
        sphereD_ = MdtBodyCreate(world_);
        /* Turn the body sphereD_ on so that it is included
           when the world world_ is evolved. */
        MdtBodyEnable(sphereD_);

        /* Render a sphere.  Assign it the color red, give it a radius
           of 1.2 and update the graphical spheres position and
           orientation with that contained in the sphereD_ struct */
        sphereG_ = RGraphicSphereCreate
             (rc, (MeReal)1.2, red, MdtBodyGetTransformPtr(sphereD_));

        /* Register the function to clean up the memory when the program exits */
        atexit(cleanup);

        /* Begin the update loop by calling the renderer, that
           in turn repeatedly calls the callback tick. */
        RRun(rc, tick, 0);
        return 0;
   }
```

Constraints: Joints and Contacts

A constraint is a restriction on the allowed motion of a physical object. This restriction gives rise to a force acting on the constrained body that affects its motion. Constraints are used to model large scale effects without worrying about the underlying physics of the situation. For instance, the forces that prevent a coffee cup from falling through a table arise from electrostatic forces between the respective molecules making up the solid materials comprising the cup and table. However, the net effect of all those complicated forces is simply that the cup doesn't penetrate the table. This can be expressed as a kinematic restriction on the motion and the force resulting from that familiar constraint - called the normal force - calculated.

There are many restrictions that can be imposed on rigid bodies using mechanical coupling, all of which are based on a constraint of some description. The more familiar ones are the revolute or hinge joint, the prismatic or sliding joint, the universal joint, and the ball and socket or spherical joint. Note however, that these joints are an idealization of the real couplings that one can construct with physical components and that are, for example, commonly found attaching doors to door frames, and to transmit drive forces from an engine to a car wheel. No real joint behaves exactly like an ideal joint since real physical bodies are never perfectly rigid, and there is always some small clearance in any given assembly.

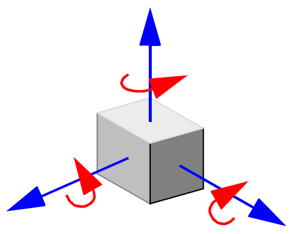
There are numerous types of constraints that can be imposed on position, angular freedom, velocity, angular velocity or certain combinations of these, in addition to restrictions on the forces required to impose a constraint (if pulled hard enough the elastic limit of a spring can be exceeded) etc. The motion of a single rigid body or the relative motion of two or more rigid bodies can be restricted. Joints are described by what are known as *equality* constraints and contacts by *inequality* constraints. While these can become quite complicated, it is the purpose of the Karma dynamics library to deal with all the details.

Constraints are a very powerful modeling tool. Karma dynamics provides an API through the dynamics library for a selection of useful constraint types.

Degrees of Freedom

A free body has six degrees of freedom, that allow it to:

- move freely in any direction in 3D space relative to another object
- freely rotate about any axis in 3D space relative to another object.



Hence, by joining two objects together up to six degrees of freedom from the attached pair of objects can be removed. To have an effect at least one degree of freedom must be removed. The number of degrees of freedom removed by a joint is a measure of the computational cost of using that joint in a simulation, the more degrees of freedom a joint removes, the more costly it is to implement.

Joint Constraints and Articulated Bodies

Articulated bodies are rigid bodies connected by *joint constraints*. A body representing a forearm can be connected to a body representing an upper arm by an ‘elbow’ hinge joint. But joints don’t just connect bodies. They also *constrain* the motion of rigid bodies: the forearm cannot be pulled far without moving the upper arm. A hinge joint can have limited rotational movement - a real human elbow cannot move more than about 160 degrees without breaking.

The Mdt library includes models of hinges, ball and socket, universal, and other joints. Hinge and prismatic joints have *limits* so that simple models of real-world behavior can be created: a hinge can be limited, for example, so that it only opens 270 degrees.

Some joints have *soft limits*, that give ‘bouncy’ effects.

Other joints are motorized, allowing control of the movement of the bodies. Motorized joints with power limits provide stable modeling of, for example, engines, brakes, motors, and stiff springs.

Joint Types

Articulated bodies are made up of two or more rigid bodies connected together by joints. Joints, like contacts, are constraints upon the behavior of bodies. This following joint types are supported by the dynamics library.

- Ball and Socket or spherical.
- Hinge or revolute.
- Prismatic or slider.
- Universal.
- Angular3.
- Car Wheel.
- Linear1.
- Linear2.
- Fixed Path.
- Relative Position Relative Orientation (RPRO).
- Skeletal
- Spring6
- Spring.
- Cone Limit constraint.

These joints can be used to attach two simulated objects to each other - that is, their relative position or orientation (or both) is constrained. Although the discussion below refers always to attached bodies, these joints may be used to attach a single object to the inertial reference frame (world) by setting it to `NULL`. Complex dynamic structures can be created by *linking* bodies together using these joints. When such structures are cross-linked (multiply connected), the model must be physically realistic, because the Mdt Library cannot deal with unphysical structures that cannot be constructed in the real world.

Hinge and Prismatic joints can have their motion restricted or the movement powered by using *limits* (stops) and *actuation* (motors).

Constraint Functionality

Contacts and joints each have a dedicated set of functions to manage their properties. An abstract, generic set of function provide a common set of basic functionality for all joint structures. Functionality specific to each joint exists in addition to this basic functionality.

The identifiers below are used in the functions that apply to each joint type.

Identifier	**Constraint**
`BSJoint`	Ball and Socket
`Hinge`	Hinge
`Prismatic`	Prismatic
`Universal`	Universal
`Angular3`	Angular3
`CarWheel`	Car Wheel
`Linear1`	Linear1
`Linear2`	Linear2
`FixedPath`	Fixed Path
`RPROJoint`	Relative Position Relative Orientation
`Skeletal`	Skeletal joint limit
`Spring6`	Spring

`Spring`	Spring
`ConeLimit`	Cone Limit constraint
`Contact`	Contact

As an example consider the function that is used to create the above constraints. An `Mdt*ID` variable must first be declared. This will point to the `Mdt*` structure where the information about that joint will be stored. The create function is common between all the constraints. It's specification is:

```
Mdt*ID MEAPI Mdt*Create (const MdtWorldID world)
```

where `*` represents one of the above identifiers. `Mdt*Create()` creates a new joint or contact and adds it to the world. A joint must be created if an articulated body is needed.

A full specification of constraint functionality is provided in the Reference Manual.

Constraint Usage

Joints

A joint attaches two objects, by restricting one or more of the degrees of freedom between them. Because it is a common and easily visualized joint, consider a hinge. This joint restricts five degrees of freedom - three translational and two rotational between the two hinged objects, i.e. the objects cannot change their position in space relative to one another, and are only free to remove by rotation about one axis. The obvious example is a door hinged to the door jamb. The following section of code demonstrates how a door hinged to the world might be created with Karma. Please refer to the reference manual for a definitive explanation of the functions.

```
/* Create a door body and define some of its properties */
const MeReal doorSize[] = {(MeReal)0.8, (MeReal)2.0, (MeReal)0.04};
doorBody_ = MdtBodyCreate(world_);
MdtBodySetPosition(doorBody, 0, 0, -7);
MdtBodySetMass(doorBody, (MeReal)2.0);
/* Give the body a kick around the rotation axes */
MdtBodySetAngularVelocity(doorBody, 0.0, (MeReal)5.0, 0.0);
/* Damping determines how quickly a body will slow down */
MdtBodySetAngularVelocityDamping(doorBody, (MeReal)0.2);
/* When evolving the world, include the body */
MdtBodyEnable(doorBody);

/* Create a hinge and attach the door to the world */
hinge = MdtHingeCreate(world_);
/* doorBody is body zero.  The world is body one. The world is
   identified with either a 0 or NULL */
MdtHingeSetBodies(hinge, doorBody, 0);
/* Position the hinge in the center of a vertical side */
MdtBodyGetPosition(doorBody, doorPosition);
MdtHingeSetPosition(hinge,
      *(doorPosition)-doorSize[0]/(MeReal)2.0,
      *(doorPosition+1),
      *(doorPosition+2));
/* Allow the door to rotate about a vertical hinge axis */
MdtHingeSetAxis(hinge, 0, 1, 0);
/* Enable the hinge so that it is included when the world is evolved */
MdtHingeEnable(hinge);
```

Most of the functions included above for setting object and hinge properties are reasonably self explanatory. While the code section could be further reduced to get across the salient points, some of the property functions are included to give users a feel for using Karma, and to allow the following minor points to be made:

- The angular velocity value above is the rotation speed around the y axis. Generally, the axis of rotation is given by the normalized rotation vector with the direction being specified in the world (Newtonian) reference frame.
- The damping parameter should be used to slow down a body's angular rotation rate. Similarly MdtBodySetAngularVelocityDamping should be used to reduce a body's linear velocity rather than setting friction.
- When a joint is created, it should be followed with the Mdt*Joint_or_Contact*SetBodies function that assigns the bodies to the constraint. This must be done before the joint position is set so that the relative positions of the bodies with respect to the joint can be computed.
- The hinge axis should be normalized and given in the Newtonian reference frame.
- Because the hinge restricts five of the six degrees of freedom between two bodies, it is a a costly joint to simulate. Each restricted degree of freedom adds one additional row to the matrix that the constraint solver solves for a sensible set of forces to satisfy all the world constraints.

Limits

The hinge and prismatic (slider along one axis) joints can have limits imposed on them to restrict their range of movement. In the case of the hinge this restricts its angular motion. The restriction that can be imposed is not limited to one revolution i.e. hinge joint limits range from -nπ through nπ for real number n hence multiple rotations are supported and a hinge passing a limit will always be detected and the correct response simulated. In the case of the prismatic joint, motion along one axis is limited. A familiar example of this would be a piston. Each joint limit can be accessed individually and its specific properties changed.

The limit properties accessible to the programmer include limit stiffness, which can be either hard (high limit stiffness) or soft. Hitting a hinge limit that is hard results in a hard bounce that reverses the bodies' angular velocities in a single time-step. A soft bounce may take many time-steps to reverse the angular velocity.

When soft limits are used, damping can be set, so that beyond the limits the joint behaves like a damped spring. If the limits are hard, the limit restitution can be set to a value between zero and one to govern the loss of angular momentum as the bodies rebound.

Powering Joints

A prismatic or hinge joint can be powered or actuated. This could be used to simulate a motor driving the constraint in it's remaining degree of freedom - angular for a hinge, and linear for a prismatic. The powered joint is force-limited and provides a useful, stable way of getting joints to move. Using them is better than applying forces or torques to the joint bodies directly, because the joint velocity is controlled directly instead of the joint acceleration.

For a hinge motor, a desired angular speed and the motor's maximum torque need to be set. The motor is assumed to be symmetric, so that the maximum torque can be applied in either direction. A torque no greater than this is applied to the hinged bodies to change their relative angular speed, until either the

- hinge angle hits a pre-set limit.
- desired angular speed is achieved. If the force limit is low or the desired velocity is large, the joint will take several time-steps to reach the desired velocity, or possibly not reach the desired velocity - see the following bullet. This action is somewhat similar to an engine with a maximum amount of output torque.
- maximum torque specified reaches a maximum angular speed that is less than the desired angular speed. This may occur, for example, if there is damping that the motor is not powerful enough to overcome.

The response of an actuated joint hitting a limit depends on the stiffness and restitution or damping properties that have been chosen for the relevant limit, but in general the joint will (quickly or slowly) come to rest at the set limit. If a soft limit has been specified, the rest position will be beyond the limit by an angle determined by the motor's maximum torque and the limit stiffness factor.

Whenever a hinge or prismatic is actuated, or is at (or beyond) one of its limits, the computational cost is equivalent to constraining six degrees of freedom - actuation adds one constraint row to the constraint matrix, increasing the computational cost.

The following section of code shows how a hinge limit is set up and a hinge joint powered:

```
/* Declare hingeLimit as a pointer to some limit struct */
MdtLimitID hingeLimit;
/* Access the limit struct of the hinge constraint */
hingeLimit = MdtHingeGetLimit(hinge);

/* Limit the hinge motion to ±0.5 radians */
MdtSingleLimitSetStop(MdtLimitGetLowerLimit(hingeLimit), (MeReal)-0.5);
MdtSingleLimitSetStop(MdtLimitGetUpperLimit(hingeLimit), (MeReal)0.5);
/* Make the limit hard, by setting the maximum limit stiffness to the maximum
  possible for the given platform */
MdtSingleLimitSetStiffness(MdtLimitGetLowerLimit(hingeLimit), MEINFINITY);
MdtSingleLimitSetStiffness(MdtLimitGetUpperLimit(hingeLimit), MEINFINITY);
/* When the hinged objects reach the limit they will not bounce i.e. the limit
  restitution is zero */
hingeLimit->limit[0].restitution = 0;
hingeLimit->limit[1].restitution = 0;

/* Turn the limits on.  The default status of the hinge is 'inactive' */
MdtLimitActivateLimits(hingeLimit, !MdtLimitIsActive(hingeLimit));
/* Activate the motor to drive the hinged objects.  Apply a maximum force of 5N
to try to attain a maximum velocity of 5m/s.  Assuming SI units */
MdtLimitSetLimitedForceMotor(hingeLimit, 10, 5);
/* Include the hinge in the evolution of the world */
MdtHingeEnable(hinge);
```

The hinge struct contains the member `bpowered`, which is an MeBool indicating whether the joint is powered (0) or not (1). The default for a joint is not powered. `MdtLimitSetLimitedForceMotor()` automatically turns the powering on.

Hint: To model dry friction in a joint set a target velocity of zero (`desired_vel` = 0), where the braking force is controlled by `fmax`. This provides a simple model of a disk brake - heat and other nonlinear effects are not included.

Limiting Motion in Two Dimensions

The limits discussed above can be used to restrict the motion in a single linear or angular degree of freedom. To restrict motion in two angular dimensions the cone limit constraint should be used in conjunction with the chosen joint - a ball and socket or universal are two examples in which the angular freedom can be constrained. For a ball and socket the cone limit does not place a limit on the 'twist' freedom, while the twist is already constrained in a universal joint. The cone limit restricts the angle between a pair of axes, one axis being fixed in each body. Cone limit behavior is ill defined for small cone angles, hence half angles less than about 5° should not be used.

The following code section demonstrates how you would set up a cone limited universal joint between an object and the world:

```
/* Create a world and add a body to it.  Set world gravity = (0, -9.81, 0) and
position the body at (2, 0, 0).  The body then falls under gravity within the
 constraint of the universal joint defined below, until it hits the limit imposed
by the cone, whereupon it comes to rest touching the ’imaginary’ cone side.  */

/* Create a universal joint between _world and _body. */
_univJoint = MdtUniversalCreate(_world);
MdtUniversalSetBodies(_univJoint, _body, 0);
/* Position the joint origin in the Newtonian reference frame 4m away from the
body. */
MdtUniversalSetPosition(_univJoint, -2, 0, 0);
/* Define the universal joint axes.  The
MdtUniversalSetAxis(_univJoint, 0, 0, 0, 1);
MdtUniversalSetAxis(_univJoint, 1, 0, 1, 0);
MdtUniversalEnable(_univJoint);

_coneLimit = MdtConeLimitCreate(_world);
MdtConeLimitSetBodies(_coneLimit, _body, 0);
MdtConeLimitSetConeHalfAngle(_coneLimit, ME_PI/3.);
MdtConeLimitEnable(_coneLimit);
```

Attaching Bodies Together with a Spring

To simulate the motion of bodies attached by a spring, Karma provides what is most accurately described as a configurable distance constraint. The identifier we use for this constraint is ‘spring’, because it is most often used to implement spring functionality. It should be noted that this configurable distance constraint can also be used to simulate the following:

- String can be simulated that can decrease in length but not increase.
- Elastic, that can decrease in length and can stretch can be simulated.
- A solid rod that cannot change its length can be simulated.

Using the Configurable Distance Constraint to Simulate a Spring

Mathematically, Hooke’s Law may be used to describe the force between objects connected by springs, and calculate the resulting motion. When using Karma to simulate a spring note that:

- The term natural length is not the separation between the two objects constrained by this constraint, but is an additional distance that is added to the separation between the two objects.
- Unlike a real spring, there is no angular constraint between the attached bodies. Only one linear dimension is constrained, restricting one degree of freedom. This adds one row to the constraint matrix.
- For high values of stiffness (over 1000) the parameter epsilon (please refer to the discussion of epsilon later in this chapter) may need decreasing to improve the stability. This is a similar to using a large range of masses.
- Offsetting the attachment position of a spring joined to two objects generates angular motion when the spring is released. Setting angular velocity damping will reduce this motion. Similarly, if the objects start to move around apply linear velocity damping.
- Set an appropriate inertia tensor for the objects - they will spin rapidly if this is too small.
- Increasing the masses of the attached objects for a given spring constant, does not change the resulting motion as would be expected for a normal spring. One would expect the joined objects to move more quickly to the equilibrium position for a smaller mass. Rather the response time is constant for different masses. This is because the constraint solver is used to calculate forces to satisfy the given constraint, hence increasing the masses joined by a spring results in an increased force being generated to move them to the specified position and satisfy the constraint, hence the response time is the same.
- The motion is symmetric even when different masses are joined by a spring, which is not physically accurate. Again this is because the forces are generated to reposition each mass individually. i.e. a larger mass has a larger force applied.

- The separation between the two attached bodies is governed by two limits that may both be *hard* (which simulates a rod or strut joint) or both *soft* (simulating a spring) or hard on one limit but soft on the other (e.g. an elastic attachment that may be stretched but not compressed). The default behavior is spring-like, with two soft, damped limits, both initialized at the initial separation of the bodies.

The following code section demonstrates how you would set up a configurable distance constraint to simulate spring motion between two bodies:

```
/* Define an inertia tensor for the boxes of the right order of magnitude */
MeReal mass = (MeReal)100.0;
MeMatrix3 boxInertia =
{
     mass, 0, 0,
     0, mass, 0,
     0, 0, mass
};
/* Need the box positions when setting up this constraint */
MeVector3 boxPosition;

/* Code (not included) here to:
1. Specify a world with two bodies and one constraint.
2. Add two bodies to the world.
3. Assign appropriate mass, mass distribution, and damping values to the boxes.
4. Position the boxes and include them in the world simulation. */

/* Create a spring and use it to constrain the two boxes */
spring = MdtSpringCreate(world_);
MdtSpringSetBodies(spring, boxOne, boxTwo);

/* Set the attachment positions of the spring to the respective dynamic body
positions of the boxes, with an offset in the y positions */
MdtBodyGetPosition(boxOne, boxPosition);
MdtSpringSetPosition(spring, 0, boxPosition[0], boxPosition[1]-1, boxPosition[2]);
MdtBodyGetPosition(boxTwo, boxPosition);
MdtSpringSetPosition(spring, 1, boxPosition[0], boxPosition[1]+1, boxPosition[2]);

/* Set the property natural length L.  The equilibrium separation of the boxes will
then be equal to the box position separation plus L */
MdtSpringSetNaturalLength(spring, 4);
/* The stiffness of the Hooke's Law constant K determines how fast the spring
reacts.  The higher K, the larger the force generated */
MdtSpringSetStiffness(spring, 10);
/* Turn the damping off.  The constraint will now be springy. Make sure that linear
damping of the bodies is zero to see this effect */
MdtSpringSetDamping(spring, 0);
/* Include the spring in the world simulation */
MdtSpringEnable(spring);
```

The Car Wheel Constraint

Vehicle simulation forms a large part of the target simulation environment for physics simulators aimed at games and entertainment. A specific constraint aimed at simulating car wheels, providing features for steering, driving and suspension, is packaged as part of the Karma constraint library. This car wheel joint makes it easier to simulate cars using Karma.

Description

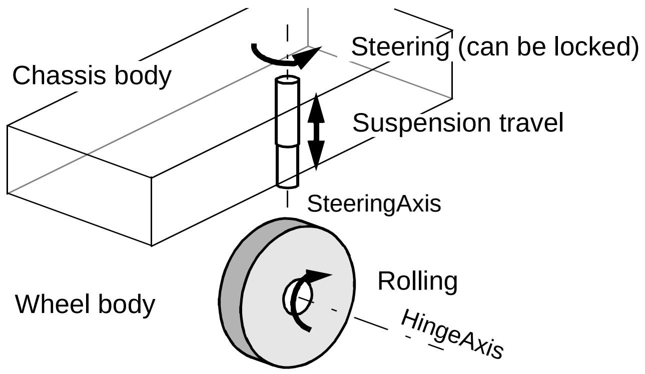
The car wheel constraint connects a wheel body to a chassis body removing three or four degrees of freedom depending on whether the wheel is steerable or not (a fixed, nonsteerable wheel is just a car wheel with steering locked). The remaining degrees of freedom are;

- rotation of the wheel about its rolling, or hinge axis
- travel along the suspension direction and
- steering of the wheel (optional)

Additional constraint rows can be added to drive the wheel, to drive the steering or, in the case of suspension, to implement spring/damper behavior with suspension travel limits, so a full six constraint rows may be generated.

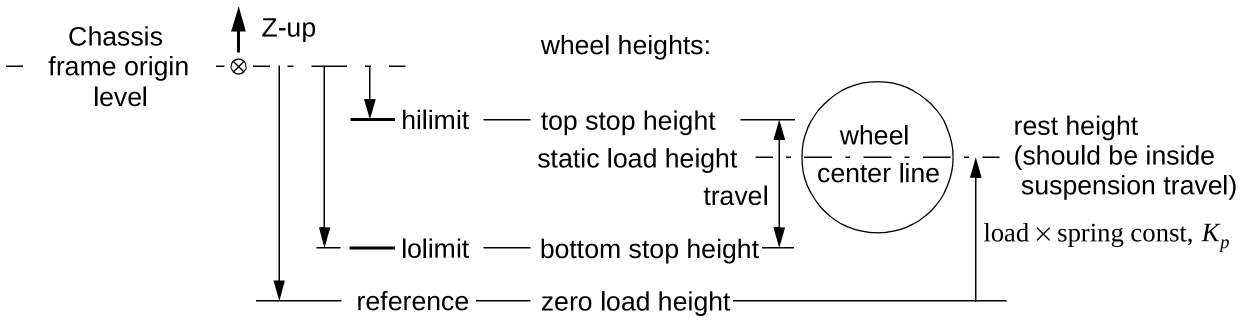
Note that the steering axis also defines the suspension direction, as in a telescopic fork, and it is assumed to pass through the center of mass of the wheel. There is no option at present to offset the wheel, to model ‘trail’ on a wheel for example, or to have separate steering and suspension axes.

The following sketch shows the degrees of freedom and axes of the car wheel constraint.



Usage

Defining the suspension is the trickiest part of using car wheel constraints. The suspension stops are relatively straightforward to set up. All heights are specified as vertical offsets (up the z-axis) measured from the chassis body origin to wheel center positions. Note that all these 'heights' will usually be negative as the full travel of the wheel suspension will usually be entirely below the chassis origin level;



Springing and damping are best set up once the chassis mass and geometry are known because the resting height of each suspension depends on the static load it carries. With zero load acting on a suspension the spring will attempt to relax to its natural height; this is the reference height used in setting up the suspension (in fact this height will not be achieved if the bottom stop is hit first).

The displacement of the resting, or static-load height from the reference height of the suspension can be estimated by multiplying the load by the spring constant.

The suspension limits are hard by default, meaning that if a wheel is forced onto one of its limits there is little discernible motion past the limit. An adjustable ‘softness’ can also be defined on the limits. Increasing the limit softness will cause visible springiness such that displacement past the limit is roughly proportional to the applied force.

The traction behavior of the car wheel depends on setting up the wheel/ground contact properly; see the section on contact constraints below.

Fast spinning wheels

The car wheel itself is hinged to the suspension. When the wheel is rotating with a high angular velocity it can be difficult to accurately simulate in real time. This is a general problem, the reason being that the integration process uses only the first few terms of a series when integrating to find new positions and velocities. For situations, such as an object falling in gravitational field, the integrator will give an exact result because position is represented exactly by a second order equation. However, for rotational motion where the series is infinite, errors become more apparent when higher angular velocities are used, which may be observed for a car traveling at high speed by the wheel wobbling. To stop this:

- Always set the hinge axes of the car wheel joints to be a fast spin axis (please refer to Chap 6 point 13). This will give a more accurate calculation of the wheel velocity when moving at high velocity. This ensures that the body orientation is updated correctly after each time increment, because discrepancies may otherwise become quite large because of the integration process.
- Use sensible mass and inertia tensors for your wheels. A large mass and inappropriate inertia tensor can make the wheels difficult to turn.
- We recommend limiting the maximum angular velocity of the wheels. If you continue to apply torque when your car leaves the ground, possibly after hitting a ramp or bump in the terrain, the angular velocity of the wheels will increase quickly because there is no resistance to the motion. The angular velocity values reached will probably be unrealistic, hence it is sensible to limit them. In addition, angular velocity damping will reduce the angular acceleration of the wheels in this case.
- If your car has small wheel radii, then the angular velocities will be higher. Can you increase the wheel radius?
- Check that you are using a sensible value for epsilon. Epsilon affects the solve process and may need adjusting to suit the masses used in your application (please refer to the discussion of epsilon later in this chapter).

Contact Constraints and Collision Detection

In a 3D simulation, models are likely to come into contact with each other. The points where two bodies intersect are called *contacts*. A contact limits how bodies can move, and is therefore a type of constraint. The Mdt library includes high-level representations of *contact constraints*.

To specify whether two bodies are in contact, create and populate a contact data structure by setting the position of the point of contact as well as the unit normal vector in the direction perpendicular to the local contact plane. If you are using Karma Collision and the Karma Bridge this will be done automatically when objects collide.

Karma Dynamics uses this contact information to model the behavior of the two bodies. For example, if a body representing a stone falls onto the floor, Karma Dynamics will detect the collision and apply the appropriate impulse so that the stone stops when it strikes the floor. The amount of rebound of the stone can be adjusted by varying the restitution between the stone and the floor.

Every time two bodies touch i.e. intersect, a *contact* must be created. The contact constraint generated will ensure that there is no relative motion in the direction opposite to the contact normal. The solver computes the constraint force required to prevent the bodies from moving against the direction of the normal as well as the tangential friction forces. The type of friction used, the computed forces exerted at the contact point on the contacting bodies, and the position and normal of the contact, form part of the information stored in a `MdtBclContactParams` structure.

Most contacts will not be actual contact points. This is because the dynamics rigid body solver may leave bodies in slightly overlapping positions. Therefore, Karma uses the idea of an “effective contact point”. For example the geometrically-visible points of contact between two shallowly-intersecting spheres forms a circle, but the best “effective contact point” to communicate to the solver would correspond to a point at the center of this circle.

These contact point directions may or may not correspond exactly to points of contact between the two bodies in terms of the visually-rendered appearance, but are a way of summarizing the “inter-surface relationship” in a way that is both efficient for the rigid body solver, and that produces the expected non-interpenetration behavior.

Interpenetration and Contact Strategies

Interpenetration of two surfaces must be prevented by carefully choosing an appropriate set of contact points. The choice depends on the type of *contact behavior*. For a bouncing ball, one contact should suffice, while resting and sliding of a box would usually require at least three contacts.

Karma Collision offers a number of alternative *contact strategies* that produce contacts for different types of situation that can be selected as necessary. Strategies that use fewer contact points have the advantage that MdtKea solves them more quickly. For those using a third-party collision package or their own in-house custom collision routines, a contact strategy must be devised that models behavior realistically without creating too many contact points.

Further features

An important feature of a physics engine is its ability to sensibly determine how to save CPU load by not simulating objects that are stationary. If you drop a box onto a terrain, you want your software to be able to recognize when it becomes motionless and to automatically remove it from the evolve. For a single box on a plane this is fairly straightforward, but as your simulation becomes more complicated it can become more difficult for the software to recognize that a particular configuration should be static.

The following sections present and discuss parameters in Karma that you should know how to adjust when creating more advanced simulations. An example code section is then provided that builds a stack of boxes and switches them off quickly by recognizing that the structure is stable.

Automatic Enabling and Disabling of Bodies

There is no need to calculate the behavior of a stationary body, unless it is struck by another body. The Mdt Library *will automatically disable stationary bodies*, removing them from the list of bodies passed to the solver. This reduces the computational time that the solver needs to update the dynamic properties of the bodies in the world.

To prevent a particular body from moving it is not enough to disable it. Any rigid body registered in a world will come to life when an enabled body becomes constrained to it, for example via a contact constraint generated by a collision. Objects that have a very small velocities and accelerations will be disabled automatically, since this typically means that the forces acting on that rigid body have come near to equilibrium.

Bodies that never move at all, such as buildings, shouldn't have any dynamics properties associated with them. They should simply be implemented as collision models, which generate appropriate contacts when dynamic objects collide with them.

Karma can check for moving objects at each time-step. If an object is found to be motionless, i.e. not interacting with other objects and with no applied motion inducing forces acting on it, it will be turned off (disabled), thus saving CPU cycles. This auto-disable feature can be turned on and off with the function MdtWorldSetAutoDisable, 0 being off and 1 being on.

When using the autodisable feature, the parameters linear and angular values of both the velocity and acceleration can be adjusted to determine when objects should be turned off. All four values need to be satisfied for this. The higher these values - that are stored in an MdtAutoDisableParams structure - the sooner an object will be disabled. When disabled, a body is not 'processed' by the solver until re-enabled by a force or collision event.

To wake up an object, Karma checks whether the force and torque values exerted on that object are larger than the force and torque threshold values required to wake up resting bodies. When Karma decides that an object should be woken up, and enables it, it may be that it will take a short time for its motion to exceed one of the four threshold values (linear and angular, velocity and acceleration) that will keep it awake. To prevent it being turned off too soon the 'alive window' property should be used that sets the number of evolve steps before an object can be disabled again, after it has been awakened. The function MdtWorldSetAutoDisableAliveWindow gives objects enough time to gain a minimum velocity after being awakened.

The Meaning of Friction, Damping, Drag, etc.

There are lots of words used to describe frictional effects, often with similar meanings. Friction is an overall term used to describe the macroscopic effect of the resistance to motion that a body experiences, the origin of which lies in the effect of physical processes on the atomic scale. Here, friction mostly refers to the forces produced at a contact between solid objects. The usual term for friction caused by motion through the air or some other fluid is drag, or viscous friction.

Friction acts either to prevent the motion of a body at rest or to reduce the velocity of a body in motion. Sticking is a characteristic of dry friction at an unlubricated contact between bodies, also called static friction. A contact in static friction will reflect applied forces up to a certain limit above which motion will start. This causes a transition from static to dynamic friction.

- Drag refers to the frictional process occurring at the solid-fluid or fluid-fluid interface. Examples include an aeroplane in flight, a cannon ball falling through water, a football moving through air, and a falling raindrop.
- Damping is a property of an oscillation / vibration and usually refers to the attenuation of resonant or harmonic motion. Examples include the damping of a car spring that reduces the oscillatory motion providing a smoother ride.

In Karma the term 'friction' is used to describe the property at the interface between solid objects, while 'damping' describes both drag and damping.

Contact properties are associated with pairs of materials. Friction is a property of the interface interaction between pairs of materials, such as aluminium with aluminium or aluminium with steel. The friction is different and currently no formulation exists that adequately predicts contact properties i.e. values are determined by direct measurement rather than by theoretical models. There is no functional representation that will take you from steel-steel through aluminium-steel to aluminium-aluminium contacts. Generally, a series of tables or graphs are used from which the appropriate values are obtained. Note that values of friction for given materials in contact are not unique. For example friction depends on temperature.

In Karma, friction parameters are chosen by the user to give the desired behavior.

The following situations demonstrate the effects of friction and damping:

- Consider a ball at rest on a surface in a vacuum. Gravity acts to constrain the ball to this surface. The friction between the ball and the surface is zero. Give the ball a quick push - don't maintain the push. Because this is a vacuum there is no air and hence no resistance to the balls motion through drag. Hence, for this highly idealized situation, the ball will continue to move along the direction in which it was pushed for ever. Because there is no friction, the ball will slide as opposed to roll.
- Now add some friction to the ball-surface interface. Maintain the vacuum. If the friction is high enough the ball will stick at the contact point resulting in an 'out of balance' force which causes the ball to rotate. The ball is in static friction i.e. the relative velocity between the ball and the surface at the contact point is zero. The ball will roll. For this idealized situation, and because the ball is in static friction, it will roll for ever. The velocity at the opposite side of the ball from the contact point is moving at twice the velocity of the ball centre.
- Now consider the same situation, but through air rather than in a vacuum. Drag occurs. The motion of the ball, be it sliding or rolling, will eventually cease.
- Now swap the ball for a box, moving in a vacuum. Friction applies. Give it a push and keep pushing. You have to push it hard enough to start it moving. Start off with a low push and push harder - increase the force. When a certain force has been reached the box will start to move. The box was in static friction. When the static friction was overcome, the box started sliding (note not rolling). When it is moving, it experiences dynamic friction. Dynamic friction acts to slow the box down, so you have to maintain the push to keep it moving. You may often find that it is easier to keep the object moving when it has started moving.

In summary, with:

- zero friction and no damping your object will SLIDE forever.
- friction at the contact and no damping your object will ROLL forever.
- linear and angular, velocity damping, a resistance to motion will occur and the object will stop.

For simplicity dynamic friction is considered to be constant. However, this is not a true model of the real world. The values used closely approximate ideal systems over a small working range - such as small velocities. In the real world, there are a number of things to be considered. For example, a football moving across the football pitch experiences ground contact at a contact area and frictional drag through the air (fluid) medium. There is a functional variation of the friction. On a still day the football at rest is in static frictional contact with the ground and air. When it is moved along the ground the drag varies with the velocity - wind resistance increases and the friction at the contact can change. While not so relevant in most cases frictional heating occurs between bodies moving in contact - consider the heat generated when using sandpaper.

Simulating Friction

Friction can conveniently be categorized as viscous friction and dry friction. Viscous friction occurs when the contact is wet i.e., when there is a lubricant at the contact like oil. Dry friction occurs when the surfaces are dry i.e., when two solid surfaces are in direct contact without any lubricant.

Viscous friction produces a force opposite to the motion of the objects in contact with a magnitude related to the relative sliding velocity. Viscous friction does not keep objects from sliding but it can slow them down. Note also that viscous friction depends on the relative velocity of the objects and produces more force at high relative velocity.

Dry friction produces a very different sort of behavior that has two aspects, namely kinetic (dynamic) friction, and static friction. A sliding object subject to dry friction experiences kinetic friction during its sliding phase and static friction once it has come to rest. Importantly, static friction prevents sliding from rest altogether while the applied tangential force acing on the object is below a threshold value. During sliding the kinetic dry friction opposes the motion with a force which (unlike viscous friction) does not depend on the relative velocity of the objects. The static threshold force is usually a little higher than the kinetic friction force.

Measurements of dry friction show that the magnitude of the kinetic friction force increases with the contact load, as does the static threshold friction force. The usual analytical model of dry friction, referred to as Coulomb friction, approximates this dependence on normal force as a linear relationship.

Coulomb Friction

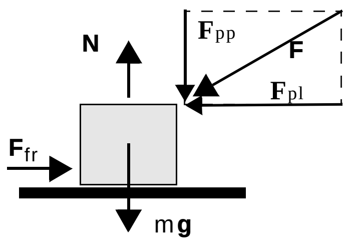
The most accurate model for modelling macroscopic friction properties between two solid objects is the Coulomb model. However, it should be appreciated that this is a model that simply describes the large scale observed phenomena. It assumes nothing about the underlying materials or nature of the surface. Microscopically, surfaces are typical very rough - even glass is smooth only to the wavelength of light. On an atomic scale 'mountains and valleys' exist on a usual glass surface, complicating any microscopic explanation of the large scale force we observe as friction. The Coulomb coefficient of friction, μ, is expressed as a function of the normal force, the normal force being the reaction force at the interface that is caused by the objects weight. Hence true Coulomb friction is known as 'cone friction', where the magnitude of the frictional force is isotropic and depends on the normal force.   For example, given a four wheeled car. If the car is of mass 1000kg (weight approx 10 000N), it might seem appropriate at the simple level to assume that one quarter of its mass acts through each wheel. Hence the normal force at each wheel is equal to the car weight divided by 4, i.e. 2500N, or 250kg mass equivalent force. If $\mu = 0.5$, then the friction force at each wheel is 1250N.

Modeling friction this way is computationally expensive, hence the Coulomb model would be used for research, engineering and visual simulation, where accuracy is important and better computing resources available. For real time simulations on low end PCs and games consoles of the type used in the entertainment and games markets, the accuracy required is such that this model can be simplified to speed things up.

Mathematically, imagine that the bodies in contact are a box and the ground: the box is sitting on the ground. The normal force **N** is the force exerted by the ground in reaction to the gravitational field strength m**g** plus the perpendicular (relative to the ground) component, $\mathbf{F}_{pp,}$ of the total external force, **F**, exerted upon the object. These forces must be equal to prevent any movement perpendicular to the ground. The force $\mathbf{F}_{fr}$ is the force of friction that spontaneously opposes the tangential (parallel to the ground) component of the external force, $\mathbf{F}_{pl}$.

Assuming that the box is initially at rest, then these forces are in equilibrium and the resulting net force is zero, as long as $\mathbf{F}_{pl}$ is smaller than the value $\mu_s\mathbf{N}$, where μ_s is the static friction coefficient, the force $\mathbf{F}_{fr}$ scales with the force $\mathbf{F}_{pl}$ such that they end up being equal to each other.

When $\mathbf{F}_{pl}$ is greater than the maximum static frictional force, $\mathbf{F}_{m\_s} = \mu_s\mathbf{N}$ the frictional force fails to scale up with $\mathbf{F}_{pl}$ and the force imbalance will cause the box to start sliding. Once the box is in motion the frictional force $\mathbf{F}_{fric}$ still impedes the movement of the box but to a lesser degree. This new frictional force is called *kinetic* friction and is written $\mathbf{F}_k = \mu_k\mathbf{N}$ where μ_k is the dynamic coefficient of friction and, in general, $\mu_k < \mu_s$.



In summary:

- At rest, static friction applies. $\mathbf{F}_{fr} \leq \mathbf{F}_{s\_m} = \mu_s\mathbf{N}$
- In motion, kinetic friction applies. $\mathbf{F}_{fr} = \mathbf{F}_k = \mu_k\mathbf{N}$

N and μ are directly related to the friction force. **N** is related to the weight of the object and to the forces exerted upon this object: the larger **N** is, the larger the friction. The friction coefficient is related to the relative smoothness of the surfaces in contact: the smaller μ is, the smoother the surface contact and the lower the friction between the two.

Note that the Coulomb friction law is itself merely an approximation to the way that real objects behave in contact. It does not have the same status as fundamental physical laws like Newton's law.

Various rigid body dynamics libraries try to simulate Coulomb friction forces. However, there are a number of serious difficulties with this since the Coulomb friction model is neither well-posed nor consistent. That is, some contact problems have multiple valid answers and some other problems have no valid answer consistent with the constraints and the Coulomb model.

Karma offers approximations to Coulomb friction that are both stable and efficient, namely Box Friction and Normal Force Friction - see below. Other alternatives (below) are to use frictionless contacts, or to use infinite friction.

Box Friction

Box Friction is the term given to one of the simplified friction models that Karma provides. The user decides on the force that must be overcome for an object to move. If, for example, this force is 10N, then an applied force under 10N won't move the object. A force of 11N will move it with a resultant of 1N. This is an adequate approach for visible realism and it benefits from being fast.

Box friction is a 2D friction model that may be used to specify the friction along two independent perpendicular directions. These directions are chosen automatically. However, you can set these yourself using MdtContactSetDirection. The primary direction must be perpendicular to the contact normal. The secondary direction is obtained from the cross product of the normal and primary direction. The frictional force for box friction, unlike coulomb friction, does not depend on the normal force. Because friction is fixed along, and calculated separately for, each direction, the matrix solve is a lot easier. This works nicely if your object is moving along one of the principal directions because only the frictional force along that direction is applied. When your object slides in a direction that coincides with one of these axes, it will not change its direction of motion. When you are moving off-axis, the maximum friction force is applied separately in both directions and the magnitude is the Pythagorean result of the 2 axial friction values in both directions - hence the name box (rectangular) friction. You can independently set the frictional force, slip etc. in the primary and secondary direction. This off axis friction will be greater than the on axis values, and, importantly, it will apply a force in a direction given by:

$$-\tan^{-1}[(\text{secondary friction})/(\text{primary friction})]$$

to the primary friction direction. Box friction results in an increase in friction when motion is not along one of the principal friction axes.

If the object happens to be moving along the principal off axis direction then it will maintain this direction. If it is moving off the principal off axis direction, the forces will act to turn the object to the nearest principal on-axis direction.

Note that Coulomb (normal force dependent) and Box friction (not normal force dependent) applies when objects are in sliding contact (dynamic friction), not rolling contact (static friction).

The tangential friction forces are applied with an upper limit (`friction1` and `friction2` for each friction direction). This can be done in either or both of the tangential friction directions. It is a simple approximation to friction because it does not depend on the normal force and is therefore equivalent to setting $\mathbf{F}_{s\_m}$ to a constant value.

Anisotropy in the friction force may be observed as a change in direction of sliding objects. This can be overcome as follows:

- Set the orientation of the friction axes using MdtContactSetDirection.
- The primary direction must remain perpendicular to the contact normal.
- The secondary direction is then automatically perpendicular to the primary direction and the contact normal.
- You can independently set the frictional force in the primary and secondary direction.
- If you set the orientation of the friction box based on the direction of travel each frame you should find that the objects slide in a straight line.
- To avoid the 'swerve', set the direction along either the primary friction axis, secondary friction axis or the off axis friction direction. We recommend the principal off axis direction.

Normal Force Friction

Normal Force friction is a refinement of box friction where at each time-step the size of the friction box is determined by multiplying the coefficient of friction between the two bodies by the normal force computed during the previous time-step. For bodies in resting contact, this adaptively configures box friction to provide a more faithful approximation to Coulomb friction.

Frictionless

No tangential friction force is applied at all, so the contacting objects are free to slide over each other. This is equivalent to setting $\mathbf{F}_{s\_m} = \mathbf{F}_k = 0$.

Infinite friction

The tangential friction forces are applied with no upper limit, so that the contacting objects can not slide over each other at all. This is equivalent to setting $\mathbf{F}_{s\_m} = \infty$.

Slip: An Alternate Way to Model the Behavior at the Contact Between Two Bodies.

You can opt to use slip whether using friction or not. The advantage of not using friction is that it makes the maths a bit easier to solve, giving a simulation speed up. Like friction, slippiness and slidiness act in the plane of contact. When setting slidiness on a contact, then one body will act like a conveyer belt, carrying the other body along its surface.

Slippiness is a property that can be useful in modeling certain special effects, such as the sideways motion of a rolling tire. When applying force to a “slippy” object, the object reaches a proportional velocity immediately; it does not accelerate to that velocity.

With the box friction model, force applied along the friction direction of two objects in contact will cause them to accelerate along that direction (when the applied force exceeds some threshold). When setting Kea’s slip property, the result is a different behavior. An applied force along the friction direction will result in a relative velocity in that direction between the objects. The velocity will be the force multiplied by the slip factor, so that $\mathbf{F}_{s\_m} = \mathbf{F}_k = 0$ and $\mathbf{v} \propto \mathbf{F}_{pl}$ where $\mathbf{v}$ is the tangential component of the velocity.

This is useful, for example, when modeling the tires of a vehicle. Applying slip along the transverse direction of the tire contact point with the ground, provides a good model of “tire slip” that will result in improved vehicle behavior. The slip should be set to a value that is proportional to the rotational velocity of the tire. If the vehicle is not moving, set the slip to zero.

Slip is faster to simulate than friction, because there is no discontinuity in the resistive force.

Setting Properties of the Virtual World - Epsilon ε and Gamma γ

The Kea integrator provides two user parameters, ε and γ**.** These can be adjusted to affect how the integrator steps the system from one state to the next.

Epsilon

At every time-step, Kea must solve a matrix equation of the form (A + ε I)X = B where *A* is positive semi-definite matrix and I is the identity matrix, to determine the forces on the rigid bodies. ε is added to the diagonal of the constraint matrix to improve Kea's ability to solve this problem. Sometimes the matrix problem may not have a solution that satisfies all the constraints. For example, some systems can get into what are known as “singular” configurations, or redundant constraints may have been specified on the system. Degenerate systems result in *A* being singular. If ε is zero, this equation may not have a solution. By making ε greater than zero a solution for X can be guaranteed.

In terms of developer usage, the important point is to understand how changing ε affects the simulation. The following symptoms suggest that the solver is having difficulty determining the forces:

- The simulation may jitter around or strange forces may appear with no apparent source. This is the result of errors in the approximate solution being amplified too much.
- The constraint solver may take too many iterations to find an approximate solution. On some platforms you will get a warning message indicating this.

In both cases the problem can be fixed by increasing ε. The only disadvantage of increasing ε is that it makes the joints and constraints a little bit springy instead of being solid. This is usually not a problem. Values of ε (such as the default 0.01) can be effective, and a value of ε near 1 is large and will almost certainly give visible springiness in the constraints - an indication that ε may be too large. Hence, it may be necessary to decrease ε to make contacts 'harder' and prevent visible penetration when:

- dealing with large mass objects (please refer to points 3, 4 and 7 in Chapter 6). A heavy mass sitting on the ground may penetrate the ground.
- using large gravitational forces.
- using a small time-step.

ε has units of time squared over mass. 1/ε is like a spring constant. If a system has no singularity, then ε can be set close to 0.

Gamma

Every time-step, after Kea has determined the forces on the rigid bodies, the positions and velocities of those bodies are updated. The way that this is done can cause the joints and other constraints to pull away from each other, so that objects will not be in their correctly constrained configurations in the new state.

Kea minimizes this problem through a relaxation process, which is controlled by γ, a positive number that is a relaxation rate.

When positions are updated, projection moves the rigid bodies γ of the way back to their correctly constrained configurations. γ applies to contacts as well - objects may penetrate a bit before a contact is generated, and the bodies then have to be projected apart again.

- If γ = 0, no relaxation is done. Joints will separate as the simulation progresses.
- If γ is large, a big correction is applied that attempts to zero the constraint violation. Make sure that the product γh (h is the time-step) is some reasonable number, preferably less than 0.5 and certainly less than 1.0. A simulation may become unstable otherwise.

Try and keep γh constant when varying the time-step. γ is more sensitive to time-step variation than ε. For most simulations a value of $0.1<\gamma<0.8$ will work well. $\gamma < 0$ or $\gamma>1$ will cause problems and should be avoided.

Stacking Boxes

The following code section demonstrates the usage of the important properties that need to be considered when trying to simulate box stack.

```
/* ....  In this example the setup was as follows:
#define NO_BOXES 21

const MeReal box_side[] = {(MeReal)1.0, (MeReal).3, (MeReal)1.0};
const MeReal box_separation = (MeReal)2.0

A mass of 5.0kg and mass matrix {5,0,0, 0,5,0, 0,0,5} were used for each box
The y position for each box was given by (i+1)*box_side[1]*box_separation for box i

.... */

world->params.gravity[1] = -(MeReal)9.81;
/* Epsilon reduced slightly to 'harden' contacts and make things less 'springy' */
world->params.epsilon = (MeReal)0.005;
/* Gamma increased to 'force' objects apart more quickly */
world->params.gamma = (MeReal)5;
/* Turn on the world option to turn objects off when they are moving below
certain user determined thresholds */
MdtWorldSetAutoDisable(world, 1);
/* The autodisable parameters may need increasing to turn objects off sooner
Note that not all parameters need to be increased by the same factor - this was
done for convenience. Recommend setting each individually. Set r=2 for 21 boxes*/
MdtWorldSetAutoDisableVelocityThreshold(world, r*(MeReal)0.02);
MdtWorldSetAutoDisableAccelerationThreshold(world, r*(MeReal)0.5);
MdtWorldSetAutoDisableAngularVelocityThreshold(world, r*(MeReal)0.001);
MdtWorldSetAutoDisableAngularAccelerationThreshold(world, r*(MeReal)0.002);

/* For each box, set the mass, mass distribution and start position */

/* Karma collision was initialised and used to determine the box collisions.  Refer
to Chapter 4 for information on using Karma Collision

/* Use the bridge to set up materials for the box and ground and then define box-
box and box-ground contact properties */

/* Use the Normal Force Friction model for box-ground contact */
MdtContactParamsSetPrimaryFrictionModel(box_ground, MdtFrictionModelNormalForce);
MdtContactParamsSetSecondaryFrictionModel(box_ground, MdtFrictionModelNormalForce);
/* Use friction in both the primary and secondary directions */
MdtContactParamsSetType(box_ground, MdtContactTypeFriction2D);
/* Set the coefficient of friction */
MdtContactParamsSetFriction(box_ground, (MeReal)0.4);
/* A low restitution stops objects bouncing */
MdtContactParamsSetRestitution(box_ground, (MeReal)0.01);

/* The default Box Friction model is used for box-box contacts */
MdtContactParamsSetType(box_box, MdtContactTypeFriction2D);
MdtContactParamsSetFriction(box_box, (MeReal)0.1);
MdtContactParamsSetRestitution(box_box, (MeReal)0.01);
```

External Forces, Torques and Impulses

External forces, torques and impulses can be applied to virtual world objects at any time by the user.

They can be applied at:

- the dynamic position - not the center of mass - of the object using the MdtBodyAdd* function.
- some position vector in the objects reference frame using MdtBodyAdd†AtPosition.

where * is Force, Torque, or Impulse and † is Force or Impulse. Impulse imparts a definite amount of momentum to a body. Force features:

- When applying a force, note that it is applied only at the next simulation time-step at which the object is included in the simulation. If your object is not included in the simulation for ten time-steps, the force will be applied at the following time-step when it is.
- Each body has a force accumulator that is used to sum the forces to be applied to that body at the next time-step.
- After the time-step, the object’s force accumulator is reset to zero.
- Because the force applied is reset to zero after the simulation of the body, to apply a constant force to a body, the force has to be added at each simulation step.
- The accumulated force is applied for the duration of the time-step, not the sum of the time-steps since the individual forces was applied.
- External forces applied to a body do not increase the required CPU time when the body to which they are applied is simulated.
- If the center of mass of a body is changed - which is independent of dynamic position - the user will need to account for this offset when applying forces.

The above applies to torques and impulses. However, note that applied impulses are time-step independent. The impulse imparts a specific amount of momentum (force x time) to an object. Impulse is useful for simulating fast impacts such as gun shots.

Collision

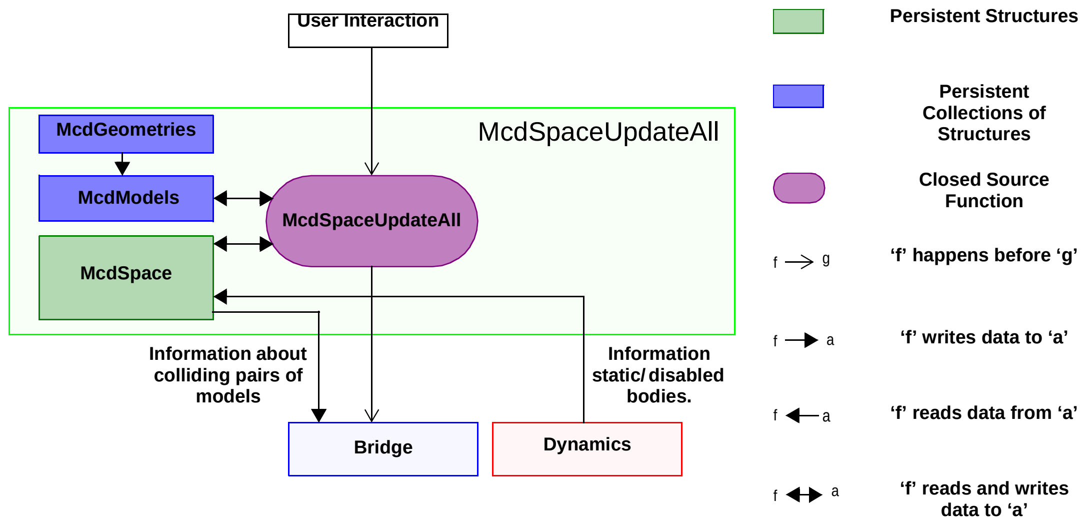
Overview of Collision Detection

The Karma Collision library (Mcd) provides a collection of algorithms that address the collision detection needs of 3D games and other applications. These algorithms are designed specifically to produce the information needed for real-time simulations of rigid bodies.

The Karma Collision library has two kinds of collision tests, the *farfield* and *nearfield*. The farfield algorithm uses bounding boxes to efficiently generate all potentially colliding model pairs. The nearfield is then used to test each pair and to generate contact information.

In Karma, the *space* manages the farfield, and the *framework* manages the nearfield. The framework contains a table of all geometry types and a table of intersection functions to test nearfield collision between geometry pairs.

The following diagram illustrates how collision is integrated with the Karma pipeline.



The main farfield function, `McdSpaceUpdateAll`, generates potentially colliding pairs. The bridge then tests each pair using the nearfield functions to generate contacts. The contacts are input to the dynamics system.

Using Collision Detection within a Universe

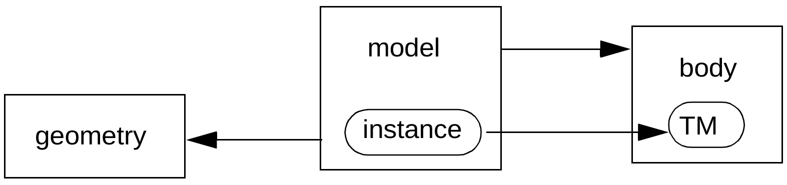
The MstUniverse offers a convenient way of integrating collision and dynamics. The three principle universe functions call the collision library as follows.

- `MstUniverseCreate` creates the framework using McdInit, and creates the space using McdSpaceAxisSortCreate.
- `MstUniverseStep` calls farfield test McdSpaceUpdateAll, and nearfield tests via MstBridgeUpdateContacts.
- `MstUniverseDestroy` includes deallocation of the framework and the space.

Explanation of Model, Geometry, and Body

The Karma Collision Detection library tests for collisions between entities in a space. These entities are called *models*. Each model contains a *geometry instance*, which indicates the shape of the model. The model may also contain a pointer to a *body*, which describes all the dynamical properties of a model. A model does not have a body if it is fixed in space (non-dynamic).

The geometry instance has a pointer to a *geometry* that describes the shape of the model, and a pointer to a transformation matrix (TM) that describes the location and orientation of the model. Usually it is convenient to make the geometry instance point to the TM of the model's body (unless the model has no body, in which case the geometry instance is given its own TM.) A geometry can be shared by multiple geometry instances.



Making a Model with a Box Geometry

Before you create a model, you should first create a universe. This will create a collision space and a framework. The following code segment creates a universe.

```
/*  Default universe sizes. */
MstUniverseSizes sizes = MstUniverseDefaultSizes;

/*  Create the universe and get the framework ID. */
MstUniverseID universe = MstUniverseCreate(&sizes);
McdFrameworkID framework = MstUniverseGetFramework(universe);
```

To create the model, first create the geometry that the model will use. The following creates a geometry and a model.

```
/*  Create geometry for a box with size dimensions 2x1x3. */
McdGeometryID boxGeom = McdBoxCreate(framework, 2, 1, 3);
/*  Create a model using the box geometry.  The density of the box is 1. */
McdModelID boxModel = MstModelAndBodyCreate(universe, boxGeom, 1);
```

To create another box that has the same shape and size, another geometry does not have to be created. The same geometry can (and should) be used to create further models. For example,

```
/*  Create another model using the box geometry. */
McdModelID boxModel2 = MstModelAndBodyCreate(universe, boxGeom, 1);
```

MstModelAndBodyCreate:

- creates a model using `McdModelCreate`.
- creates a body using `MdtBodyCreate`, enables it and sets mass properties.
- sets the model to point to the body using `McdModelSetBody`.
- inserts the model into the space using `McdSpaceInsertModel`.

Making a Model with a Convex Mesh Geometry

Convex mesh is a geometry that can represent any finite convex polyhedron. A convex polyhedron is a shape made of flat polygon faces. It cannot include any holes or any edges that are indented.

To create a convex mesh, only the vertices of the polyhedron need to be specified. The geometry code will use the Qhull (please refer to README_Qhull.txt in ....\ metoolkit\3rdParty) algorithm to compute the faces and edges of the polyhedron. The vertices are specified as an array of MeVector3s. Each MeVector3 contains three floating point numbers containing the 3D space vector position of a vertex.

The following code creates a four-sided pyramid.

```
/*  These are the five points of the pyramid. */
MeVector3 pyramidPoints[] = {
    { 0, 2, 0 }, { -3, -1, -3 }, { 3, -1, -3 }, { -3, -1, 3 }, { 3, -1, 3 }};
/*  Create a pyramid geometry. */
McdGeometryID pyramidGeom = McdConvexMeshCreateHull(framework,pyramidPoints,5,0);
```

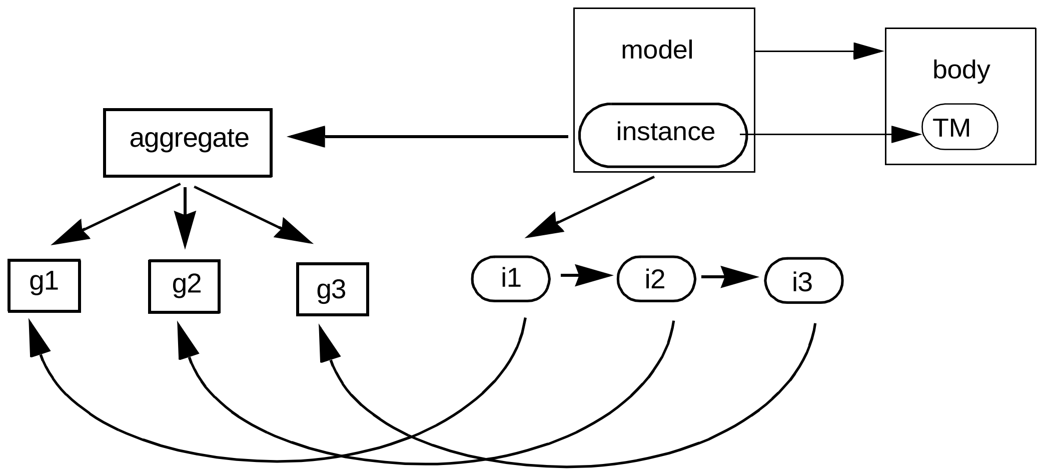
```
/*  Create a model using the pyramid geometry. */
McdModelID pyramidModel = MstModelAndBodyCreate(universe, pyramidGeom, 1);
```

Note that by default, the center of mass of the body is at the origin of the geometry. This should be taken into consideration when defining the points of the convex mesh. The body center of mass can be offset from the geometry origin using `MdtBodySetCenterOfMassRelativePosition`. Note that this will cause a small performance overhead in dynamics.

It is permissible to over-specify the points of the convex mesh. Any points that are redundant, or in the interior of the convex hull, are ignored.

Creating a Model with an Aggregate Geometry

A model can contain only one geometry instance, so what if a model that consists of several geometries is needed, such as a basic chair consisting of a rectangular box for the seat and four cylinders for the legs. To accomplish this, use a special geometry called an *aggregate*. The aggregate geometry may contain any number of simple primitive geometries (sphere, box, cylinder or sphyl), g1, g2, g3, as described in the following diagram..



When the model is created with an aggregate geometry, the geometry instance in the model is created with child instances, i1, i2, i3; one instance for each primitive geometry comprising the aggregate.

Each child geometry, g1, g2, g3 contains a relative TM that specifies the translation and rotation of the child with respect to the parent reference frame. The transformation of each child instance, i1, i2, i3 is the composition of its respective geometry relative TM and the parent geometry instance TM (which is usually the body TM).

The child geometry instances are allocated from a pool that has a default size of zero. You must set the pool size to at least the total number of aggregate child instances you will need. The following code does this:

```
/*  Set pool size for aggregate child instances. */
sizes.collisionGeometryInstancesMaxCount = 100;
universe = MstUniverseCreate(&sizes);
```

Now create a stool where the top of the stool is a box and the four legs are cylinders:

```
MeMatrix4 tm;
McdAggregateID ag;
McdGeometryID seat, leg;
McdModelID model;

/*  Create seat and leg geometries. */
seat = McdBoxCreate(framework, 2.2f, 2.2f, 0.1f);
leg = McdCylinderCreate(framework, 0.1f, 2);

/*  Create an aggregate geometry to hold five parts. */
ag = McdAggregateCreate(framework, 5);

/*  Combine a seat and four legs to make the stool. */
MeMatrix4TMMakeIdentity(tm);
```

```
    MeMatrix4TMSetPosition(tm, 0, 0, 1);
    McdAggregateAddElement(ag, seat, tm);
    MeMatrix4TMSetPosition(tm, 1, 1, 0);
    McdAggregateAddElement(ag, leg, tm);
    MeMatrix4TMSetPosition(tm, 1, -1, 0);
    McdAggregateAddElement(ag, leg, tm);
    MeMatrix4TMSetPosition(tm, -1, 1, 0);
    McdAggregateAddElement(ag, leg, tm);
    MeMatrix4TMSetPosition(tm, -1, -1, 0);
    McdAggregateAddElement(ag, leg, tm);

    /*  Create a model and body for the stool. */
    model = MstModelAndBodyCreate(universe, ag, 1);
```

While all four legs of the stool use the same geometry, each leg has a different relative TM.

The dynamics system computes the TM for an instance of a component of an aggregate by composing the relative transform of the component and the transform of the instance corresponding to the component's parent. That is, by mapping the component into the parent's space, and then into world space. Although this transformation is not stored by default, it can occasionally be useful, such as when rendering the aggregate by rendering each component separately. In this case, assign a pointer to an `MeMatrix4` to the instance corresponding to that component, by using the function `McdGeometryInstanceSetTransformPtr`. Then, when the model is updated the TM of the component will be stored in the allocated space. The following code allocates memory for the instances of the stool aggregate.

```
    McdGeometryInstanceID parent, child;
    MeMatrix4Ptr newTM;

    parent = McdModelGetGeometryInstance(model);

    for (i = 0; i < 5; ++i)
    {
        child = McdGeometryInstanceGetChild(parent, i);
        newTM = (MeMatrix4Ptr) malloc(sizeof(MeMatrix4));
        McdGeometryInstanceSetTransformPtr(child, newTM);
    }
```

Materials are specified on a per-instance basis, so different materials may be specified for each instance of each component of an aggregate.

Constructing an Immovable Model, such as a Terrain

Often in games immovable objects are required for terrain, buildings and other inanimate non-dynamic objects. Immovable objects are needed by collision detection, but do not exhibit a dynamic response. Therefore an immovable object has a collision model, but no dynamic body. The following code creates the geometry and model for a infinite flat plane.

```
    /*  Allocate a TM and set it to the identity matrix. */
    MeMatrix4Ptr planeTM = (MeMatrix4Ptr) malloc(sizeof(MeMatrix4));
    MeMatrix4TMMakeIdentity(planeTM);

    /*  Create geometry and model for an infinite flat plane. */
    McdGeometryID planeGeom = McdPlaneCreate(framework);
    McdModelID planeModel = MstFixedModelCreate(universe, planeGeom, planeTM);
```

It is very important to note that the `MstFixedModelCreate` API *does not* make a copy of the TM. Therefore the matrix must be allocated using malloc, or some other persistent location. The functions that retain a pointer to the TM are:

- `MstFixedModelCreate`.
- `McdModelSetTransformPtr`.
- `McdGeometryInstanceSetTransformPtr`.
- `McdModelSetRelativeTransformPtrs`.

`McdPlaneCreate` creates a plane through the origin whose normal points along the z. This is fine if the universe uses the z-axis for the "up" direction. However, if the universe uses the y-axis for "up" then the plane TM needs to be rotated around the x-axis by $\pi/2$.

Disabling and Enabling Pairs of Models

```
      MeBool MEAPI McdSpaceDisablePair     ( McdModelID m1, McdModelID m2 )
```

prevents a pair of collision models, `m1` and `m2`, from appearing in the output pairs of a `McdSpace`, thus preventing any collision between the two models.

After a call to `McdDisablePair()`, if in the last time-step the list of pairs returned by `McdSpaceGetPairs()` contained a hello or staying pair involving the collision models `m1` and `m2`, then that pair will appear as a goodbye pair in the next call to `McdSpaceGetPairs()`. After that point, no more references to that pair will appear, regardless of whether they are in close proximity to each other. If

```
      MeBool MEAPI McdSpaceEnablePair      ( McdModelID m1, McdModelID m2 )
```

is subsequently called on the same pair of models, the pair will reappear as a new hello `McdModelPair` object, if the pair is in close proximity to each other. Refer to "Testing Collision" p. 74 for more explanation of hello, staying, and goodbye pairs.

The status of the pair can be checked with

```
      MeBool MEAPI McdSpacePairIsEnabled   ( McdModelID m1, McdModelID m2 )
```

Pairwise disabling of objects is generally quite flexible. It can be cumbersome if many pairs of objects are disabled, or objects are constantly added and removed from the collision space, since the collision space tracks disabled pairs. An alternative is to create a *culling table*, which is a symmetric pairwise matrix of flags, using the function

```
      McdCullingTable * MEAPI McdCullingTableCreate( int size)
```

The function

```
      McdCullingTable * MEAPI McdCullingTableSet( McdCullingTable *const table,
                                                  MeU32 a,
                                                  MeU32 b,
                                                  MeBool flag)
```

can then be used to set the value of an entry in a culling table. When the model is inserted into the farfield, it can be assigned a culling table, a culling index, and a culling ID, using the functions

```
      void MEAPI McdSpaceSetModelCullingParameters( McdSpace *space,
                                                    McdModelID cm,
                                                    McdCullingTable *table,
                                                    int cullingIndex,
                                                    int cullingID);
```

Or alternatively (and more efficiently) it can be inserted into the collision space with culling parameters with the function

```
      void MEAPI McdSpaceInsertModelWithCulling( McdSpace *space,
                                                 McdModelID cm,
                                                 McdCullingTable *table,
                                                 int cullingIndex,
                                                 int cullingID);
```

Two models in the space will act as though they are pairwise disabled if the following conditions are met:

- they both have culling parameters.
- they have identical culling table and culling ID values.
- the flag indexed by their culling indices in the culling table is set (note that the table is symmetric).

Although it is possible to change the culling parameters of a model within the space to point to a different culling table, or change the culling ID or index, note that the table itself should be considered constant during the simulation. Behavior is undefined when assigning a model a set of culling parameters and then changing the underlying table.

Time Of Impact

Since Karma simulates physics numerically, with a discrete time-step, some collisions may not be detected. Consider a ball travelling at high velocity toward a thin box. If at time t the ball was near to entering the box (not touching it), and that a time t+Δt later the ball position had placed it on the other side of the box, the system would not detect any collision between the ball and the box. For this system, if there was not an intersection between the two models at a given time-step then no collision occurred.

To prevent this from happening, utility functions were created to help detect such virtual collisions. These utilities work by using approximations and are therefore not foolproof.

```
MeBool MEAPI McdSafeTime             ( McdModelPair* pair,
                                       MeReal maxTime,
                                       McdSafeTimeResult* result )
```

performs appropriate SafeTime (time of impact and swept volume) computation on input pair. The estimated time of impact is returned in `McdSafeTimeResult result`. Models are assumed to be synchronized, that is, their transforms correspond to positions at identical instants of time. The model's linear and angular velocities are used to describe the motion of the object. The input value of `maxTime` indicates the maximum time for travel given the model linear and angular velocities. The time-step for a dynamical simulation is a typical value for `maxTime`. For a value of 1, the object is assumed to translate by the entire linear velocity vector. The return value is 1 if the proper `SafeTime()` function was available, 0 otherwise.

NOTE: Currently, only some model interactions can use the SafeTime utility: {sphere, box, cylinder}/{sphere, box, cylinder} and plane/{sphere, box} interactions. Other interactions will return a time of impact of maxtime.

Querying Line Segment Intersections

Two utility functions from `McdSpace` are available in Mcd to perform intersection tests, one that finds the first point of intersection between a directed line and geometries, and one that finds all the points of intersection between a directed line and geometries.

```
int MEAPI McdSpaceGetLineSegFirstIntersection
                                     (
                                       McdSpaceID space,
                                       MeVector3Ptr inOrig,
                                       MeVector3Ptr inDest,
                                       McdLineSegIntersectResult *outResult
                                     )
```

finds first intersection of an oriented line segment with all models in the collision space. The arguments `inOrig` and `inDest` are pointers to `MeVector3` variables representing the first point and the second point on the line segment. The argument `outResult` is a structure containing the returned line segment intersection data. 1 is returned if an intersection is found, 0 otherwise. See the notes below.

```
int MEAPI McdSpaceGetLineSegIntersections
                                     (
                                       McdSpaceID space,
                                       MeVector3Ptr inOrig,
                                       MeVector3Ptr inDest,
                                       McdLineSegIntersectResult *outList,
                                       int inMaxListSize
                                     )
```

finds all the intersections of an oriented line segment with all models in the collision space. The arguments `inOrig` and `inDest` are pointers to `MeVector3` variables representing the first point and the second point on the line segment. The argument `outList` is a structure containing the returned line segment intersection data. The value `inMaxListSize` is the maximum number of intersections that will be reported. It returns the number of intersection results. See the notes below

The `McdLineSegIntersectResult` structure stores the intersection data.

Structure Member	Description
`McdModelID  model`	Collision model intersecting with the line segment.

Structure Member	Description
`MeReal position[3]`	Intersection point.
`MeReal normal[3]`	Model normal at intersection point.
`MeReal distance`	Distance from the first end point of line segment to the intersection point.

To perform intersection tests between line segments and individual models use

```
unsigned int MEAPI McdLineSegIntersect(const McdModelID cm,
                                       MeVector3Ptr inOrig,
                                       MeVector3Ptr inDest,
                                       McdLineSegIntersectResult* outOverlap )
```

which intersects a line segment with a collision model. The variable `cm` represents the collision model. The variables `inOrig` and `inDest` are pointers to `MeVector3` representing the first and second point on the line segment. The variable `outOverlap` structure containing the line segment intersection data. Returns 1 if an intersection was found, otherwise 0. See notes in the `McdSpaceGetLineSegIntersections()` function above.

```
int MEAPI McdSpaceGetLineSegFirstEnabledIntersection
                                     ( McdSpaceID space,
                                       MeVector3Ptr inOrig,
                                       MeVector3Ptr inDest,
                                       McdLineSegIntersectEnableCallback filterCB,
                                       void * filterData,
                                       McdLineSegIntersectResult *outResult )
```

is identical to `McdSpaceGetLineSegFirstIntersection()`, but allows a callback function to be provided that selectively exclude some models from the query. For example, in a line of sight application, invisible or transparent may need to be excluded. The argument `filterCB` is a pointer to a function that takes an `McdModelID` and `filterData` and returns an `int`. The line query is performed on every model for which the callback returns a non-zero value. The argument `outResult` is a structure containing the returned line segment intersection data. It returns 1 if an intersection is found, otherwise 0.

Notes:

- If the point `inOrig` is inside any model, then the first `McdLineSegIntersectResult` structure is returned with distance zero and position equal to `inOrig` and the normal is undefined.
- The tests for line intersection work with all geometries except TriangleList. Intersections with, and occlusions by, TriangleList models are ignored. The TriangleList intersection testing is best handled by application specific code.
- At most one intersection is reported per model, where the line segment first enters the model. An intersection is not reported where the line passes out of the model, nor if the line re-enters the model. In the case of an aggregate model, only the first intersection is reported, even though the line segment may pierce the aggregate several times.

Testing Collision Directly

In most cases, collision detection is performed by the bridge component in the function `MstUniverseStep`. However, there may be some cases when a direct test for collision between two models is needed. To test collision first create and initialize a model pair. Then invoke the farfield test with `McdNearby`, and the nearfield test with `McdIntersect`. Both of these return a boolean that is false if the models do not touch. The following example code demonstrates testing collision directly.

```
McdModelPair *pair;
McdIntersectResult result;

/*  Create and initialize the model pair. */
pair = (McdModelPair *) calloc(1, sizeof *pair);
pair->model1 = myModel1;
pair->model2 = myModel2;
McdHello(pair);

/*  Call farfield test. */
if (McdNearby(pair))
```

```
    {
        /*  Allocate space for the result. */
        result.pair = pair;
        result.contactMaxCount = 20;
        result.contacts = (McdContact*) calloc(20, sizeof(McdContact));

        /*  Call nearfield test which also generates contacts. */
        if (McdIntersect(pair, &result))
        {
            /*  Process result here... */

        }
        free(result.contacts);
    }
    /*  Deallocate the pair. */
    McdGoodbye(pair);
    free(pair);
```

Be aware that `McdSpaceUpdateAll` is a far more efficient way to generate pairs than calling McdNearby for every pair of models in a universe.

Creating a Model Without Using the Universe

The `MstUniverse` functions hide many of the details required to initialize the collision library and create a model. Eventually, it may be useful to understand these details, especially for performance tuning an application.

This following example demonstrates initialization of the collision library instead of calling `MstUniverseCreate`. The function `McdInit` takes four parameters: *geoTypeMaxCount* is the maximum number of user-defined geometry types, *modelCount* is the maximum number of models in the universe, *instanceCount* is the maximum number of aggregate component instances in the universe, and *lengthScale* is the length of a medium-sized object in the universe. The function `McdSpaceAxisSortCreate` takes four parameters: the framework, *axes* is which axes are tested for overlap, *objectCount* is the maximum number of objects the space will hold, *overlapCount* is the maximum number of overlapping pairs the space will handle.

```
    /*  Create the collision detection framework for geometry types. */
    McdFrameworkID framework = McdInit(0, 50, 0, 1);
    McdPrimitivesRegisterTypes(framework);
    McdPrimitivesRegisterInteractions(framework);

    /*  Create the space for farfield collision testing.  */
    McdSpaceID space = McdSpaceAxisSortCreate(framework, McdAllAxes, 50, 100);
```

The following example creates a fixed model plane. The body of the plane model is NULL, hence the TM for the geometry instance must be allocated and set.

```
    /*  Create a plane geometry and model for the terrain. */
    planeGeom = McdPlaneCreate(framework);
    planeModel = McdModelCreate(planeGeom);

    /*  Set the plane body to NULL and set the TM. */
    McdModelSetBody(planeModel, 0);
    McdModelSetTransformPtr(planeModel, planeTM);

    /*  Insert the plane into the space and flag it as frozen. */
    McdSpaceInsertModel(space, planeModel);
    McdSpaceUpdateModel(planeModel);
    McdSpaceFreezeModel(planeModel);
```

This example creates a box and insert it into the collision space. It is assumed that boxBody is a valid dynamics body.

```
    /*  Create a box and insert it into the collision space. */
    boxGeom = McdBoxCreate(framework, 2, 1, 3);
    boxModel = McdModelCreate(boxGeom);
    McdModelSetBody(boxModel, boxBody);
    McdSpaceInsertModel(space, boxModel);
```

Geometrical Types and Their Interactions

Overview

Karma Collision offers many concrete geometrical types to choose from when defining collision model shape. The selected geometry should correspond closely, but not necessarily exactly, to the geometry of the 3D graphics model rendered onto the screen.

Simpler geometrical types require less memory to hold the representation, and more straightforward algorithms to operate on them. A wide selection of geometry types gives flexibility in choosing trade-offs between performance, memory and geometrical accuracy.

Register Types of Geometries to be Used

`MstUniverseCreate` calls `McdPrimitivesRegisterTypes()`, which registers all the primitive geometry types and all the interaction functions (nearfield collision tests) between the primitive types.

To register only the required geometry types, hence saving some space in memory, replace or rewrite `MstUniverseCreate` to call successively `Mcd*RegisterType()` where the asterisk is one of the collision geometry types, Aggregate, Box, ConvexMesh, Cylinder, Plane, Sphyl, Sphere or TriangleList.

`Mcd*RegisterTypes` can be called, where the asterisk is either

- `Primitives` to register all primitive geometries.
- `SphereBoxPlane` to registers sphere, box and plane.

Remember that both the geometry types and the interaction algorithms must be explicitly registered with the Mcd system at initialization. If, for a particular geometry type – geometry type combination, there is no algorithm registered or available, the interaction will be ignored.

NOTE: **The currently supported primitive collision geometries are:**

Box

Cylinder

Plane

Sphere

Sphyl

TriangleList

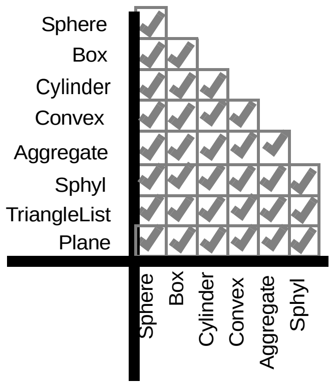
The currently supported non-primitive collision geometries are

Aggregate

ConvexMesh

Intersection Functions

The following figure lists the intersection functions available for all pairs of geometry types provided in Karma Collision.



	Sphere	Box	Cylinder	Convex	Aggregate	Sphyl
Sphere	✓					
Box	✓	✓				
Cylinder	✓	✓	✓			
Convex	✓	✓	✓	✓		
Aggregate	✓	✓	✓	✓	✓	
Sphyl	✓	✓	✓	✓	✓	✓
TriangleList	✓	✓	✓	✓	✓	✓
Plane	✓	✓	✓	✓	✓	✓

Development State of the Intersection Functions of All Pairs of Geometrical Types.

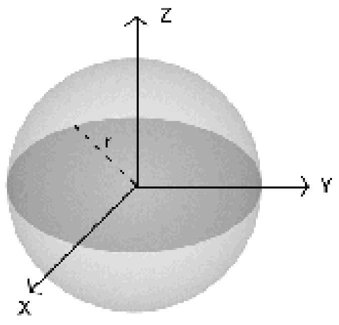
There is no need to test static geometry types (trilist and plane) against one another.

Geometrical Primitives

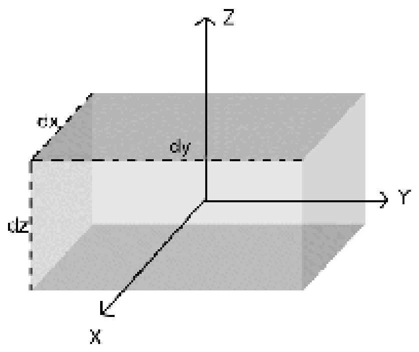
Primitive geometrical types are shapes defined by a small and fixed number of parameters. They can represent various specific types of curved surfaces exactly by parameters and specialized algorithms, not by discretized (triangulated) approximations. They are lightweight, fast and geometrically accurate. A number of non-primitive types are also available for defining models having more general surface shapes.

The following primitive types are available:

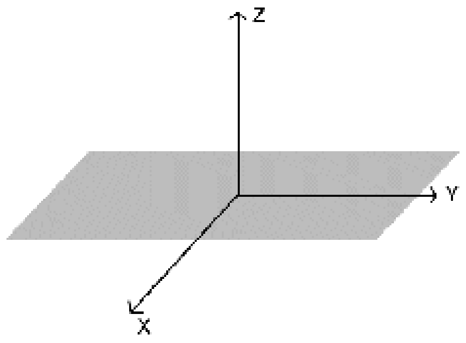
Sphere



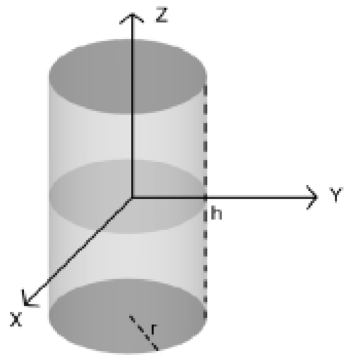
Box



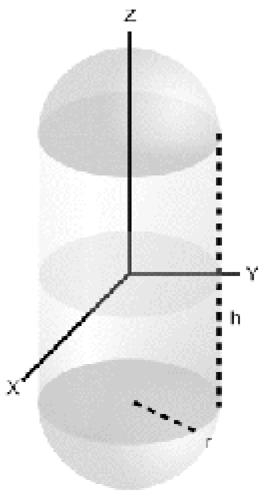
Plane



Cylinder



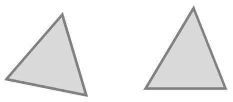
Sphyl



The figures above illustrate the primitive geometrical types in their local coordinate system, and indicate the parameters used to specify each. There are two conventions common to all primitives:

- The (uniform density) center of mass is placed at the origin. This is convenient and efficient for working with dynamics.
- Where relevant, the z-axis is the one about which there is a symmetrical distribution of volume. This applies to the cylinder, plane and sphyl.

Triangle List

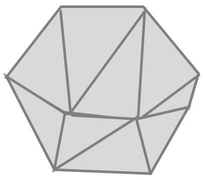


Triangle List is a primitive intended to allow triangulation of static geometry such as terrain. A triangle list is created using a bounding box and a user-supplied querying callback. If the Karma farfield detects a collision between the bounding box of the triangle and the bounding box of another object, it uses the callback to query for a list of `McdUserTriangles` near the object, and then produces a list of contacts by checking for intersection of the object with each triangle.

When representing a surface as a triangle list it is useful to be able to distinguish between triangle edges representing edges in the surface, and those created as artifacts of the triangulation. In order to facilitate this, the `McdUserTriangle` structure contains a `flags` field that allows combinations of the following values:

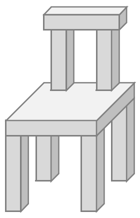
Flag	Description
`kMcdTriangleUseSmallestPenetration`	Use the normal that minimizes translational distance required to separate the object and triangle. If this is not set, use the triangle face normal.
`kMcdTriangleUseEdge0`	Make edges sharp, that is, generate contacts where the triangle edges intersect with the object.
`kMcdTriangleUseEdge1`	
`kMcdTriangleUseEdge2`	
`kMcdTriangleTwoSided`	Triangle is two sided. If not set, contact normal is reversed if its dot product with the supplied normal is negative.
`kMcdTriangleStandard`	Set all the above flags (produces behavior equivalent to Karma 1.0)

ConvexMesh



The ConvexMesh geometry type can be used to specify a geometry representing a closed convex object.

Aggregate



The Aggregate geometry type can be used to group together several existing geometries to produce a new geometry. Aggregates can be nested arbitrarily, and like other geometries, shared between several models.

NOTE: Composite geometries do not exist in Karma 1.2.

In order to create an aggregate, the maximum number of component geometries it contains must be specified.

```
McdAggregateID MEAPI McdAggregateCreate (McdFramework *frame, int maxChildren)
```

Components can then be added to the aggregate using

```
int MEAPI McdAggregateAddElement     (McdAggregateID agg,
                                      McdGeometryID geom,
                                      MeMatrix4 relTM)
```

where `agg` is the aggregate that is being added to, `geom` is the geometry to be added to the aggregate and `relTM` is the relative transform of the component that is being added. The aggregate element will keep a pointer to the relTM, therefore relTM must be allocated on the heap, or some persistent location.

Advanced Features

Collision Spaces

When the collision models have been created, a collision space to hold them must be made. The following function creates a `McdSpace` object whose models are sorted by x, y, and z axes:

```
McdSpaceID MEAPI McdSpaceAxisSortCreate   ( McdFrameworkID fwk, int axes,
                                            int objectCount, int pairCount )
```

The axis sorted space produced by this command is efficient at determining when pairs of models are nearby. The axes variable is a bit field that specifies which axes to test for model proximity. Its possible values are a combination of the bits `McdXAxis`, `McdYAxis`, and `McdZAxis` or the value `McdAllAxes`, which is a constant equal to `(McdXAxis+McdYAxis+McdZAxis)`. The variables `objectCount` and `pairCount` are the maximum number of objects the space may hold and the maximum number of overlapping pairs the space will handle respectively.

The function `McdSpaceAxisSortCreate()` calls `McdSpaceBeginChanges()`, so that `McdSpaceIsChanging()` is true directly after creation, allowing the `McdSpace` to be populated immediately via `McdSpaceInsertModel()`. Hence,

```
McdSpaceID space = McdSpaceAxisSortCreate(McdFrameworkID fwk, McdAllAxes,
                                    MAX_BODIES, AV_COUNT*MAX_BODIES);
```

creates a space with room for `AV_COUNT*MAX_BODIES`, where the constant `AV_COUNT` is the expected number of objects near to a given object. This is less than the space needed for the maximum number of pairs, `MAX_BODIES`^{2}.

Note that `AV_COUNT` must be empirically determined by the programmer. If a simulation uses a few objects placed in a large room, it is reasonable to expect that any object could be nearby to at most two other objects, yielding a value of two for `AV_COUNT`.

However, if a few marbles are placed in a large bag, where any one of these marbles may collide with any other marble at one time, then `AVE_NEARBY` would be of the order of `MAX_BODIES`. Of course, if the number of marbles is large and / or the bag is small enough, the marbles would not be as free to move and interact with each other. In this case, every marble could only be in close proximity with at most twelve other marbles, leading to a value of `AV_COUNT` of at most twelve.

NOTE: Be careful, if the number of pairs of nearby objects is higher than pairCount then a simulation may seem to run correctly but the computed physics will be wrong. This is because contacts will not be generated. In the debug library a warning will be output.

When a collision space is first created, it is empty. Models must be inserted into the collision space, one by one, and the space built. Note that a `McdModel` object can only be present in one `McdSpace` at a time.

```
MeBool MEAPI McdSpaceInsertModel     ( McdSpaceID space, McdModelID collModel )
```

inserts a collision model into a collision space. It returns 1 if successful or 0 if not. This can only be called when `McdSpaceIsChanging()` is true.

The `space` will now keep track of the volume occupied by `collModel` and detect any close proximities with other models present in `space`. Note that the function `McdSpaceInsertModel()` does not imply `McdSpaceUpdateModel()`: the latter function must be called explicitly (or via `McdSpaceUpdateAll`).

When all the collision models have been added to the collision space, the space itself needs to be built. This is accomplished by calling

```
void MEAPI McdSpaceBuild             (McdSpaceID collSpace)
```

to indicate that most models have been inserted. Calling `McdSpaceBuild()` does not prevent later insertions or deletions, but for large changes the running time might be slightly affected for certain implementations of `McdSpace`. The function `McdSpaceBuild()` should be called after the last insert before the simulation begins.

To add models after `space` has been built is a bit different than at initialization time. When using Karma Dynamics through the Simulation Toolkit bridge, the procedure to add a model is the same as during initialization. When only using Karma Collision, the models need to be added in a change block. Here is an example:

```
*     McdSpaceBeginChanges(space)          // if outside a change block
      McdSpaceInsertModel(space,model)
*     McdSpaceUpdateModel(model)
*     McdSpaceEndChanges(space)
*     [handle hello pairs]
```

Steps marked with * are usually done in the simulation loop and typically need not be done again upon insert, unless special handling of pairs involving inserted models is needed.

Transform and Synchronization with Graphics

Ensure that the collision model's coordinate system matches that of the 3D graphics model that is being displayed. This often involves a three-way synchronization between the dynamic body position and orientation, the graphic model rendered on the display, and the collision model used for determining contacts.

The collision model's transform is an `MeMatrix` object. It must be explicitly allocated, initialized, and then assigned to its `McdModel` using:

```
void MEAPI McdModelSetTransformPtr (McdModelID cm, MeMatrix4 geometryTM);
```

Failure to perform this step, results in the default value returned by `McdModelGetTransformPtr()` being NULL, rather than a pointer to an identity matrix.

If an `McdModel` is not inserted in, and hence not updated by, an `McdSpace`, the user must call `McdModelUpdate()` whenever the transform changes. This is handled automatically by `McdSpace`.

NOTE: When integrating with Karma Dynamics, the `McdModel` by default shares the `MdtBody`'s TM, in which case the above step is not required.

`McdModelUpdate()` can optionally invoke a user-written callback function. This can be set by the function `McdModelSetUpdateCallback()`.

Transition

```
int MEAPI McdSpaceGetTransitions     ( McdSpaceID s,
                                       McdSpacePairIterator* iter,
                                       McdModelPairContainer* a )
```

has the same effect as `McdSpaceGetPairs()`, only no staying pairs are reported.

It is useful to isolate the new events produced by a set of modifications in a *changes* block, and have them processed separately from (or earlier than) the remaining events. For example:

```
/* Start a change block */
McdSpaceBeginChanges(s);
McdSpaceRemoveModel(s,m1);
McdSpaceRemoveModel(s,m2);
/* End a change block */
McdSpaceEndChanges(s);

/* Get just the transition events due to RemoveModel() calls */
McdSpaceGetTransitions(s,pairs);
/* handle bookkeeping for goodbye pairs*/
MstHandleTransitions( pairs );

McdSpaceBeginChanges(s);
/* do other modifications, updates .. */

McdSpaceEndChanges(s);

/* Now get all the model pairs - including staying pairs */
McdSpaceGetPairs(s,pairs);
/* handle hello,staying and goodbye pairs*/
MstHandleTransitions( pairs );
```

Static Models

Collision models that are not going to move for a while can be *frozen* in the collision space. This reduces the number of models that need to be updated, thus increasing speed. To do this use

```
MeBool MEAPI McdSpaceFreezeModel(McdModelID cm)
```

to inform the system that a collision model's TM will not change value. The collision model remains in the frozen status until `McdUnfreezeInSpace()` is called.

`McdModel` objects are by default in unfrozen status. This information is used for optimization opportunities by the `McdSpace` object in which a model is currently active. `cm` will be ignored by subsequent calls to `McdSpaceUpdateAllModels()`.

Subsequent calls to `McdSpaceModelIsFrozen()` using `cm` will return true. The bounding volume associated with `cm` will no longer be changed, keeping the values it had the last time that `McdSpaceUpdateModel()` was called on it. `McdModelUpdate()`, which updates relative transforms and other data, will also not be called on a frozen model. It is an error to call `McdSpaceUpdateModel()` on a model for which `McdSpaceIsFrozen()` returns true.

Note that `McdSpaceFreezeModel()` does not imply a call to `McdSpaceUpdateModel()`. If a collision model transform has changed since the last changes block (see below) or has just been inserted, and you wish to freeze the model in the configuration corresponding to its new transform value, be sure to call `McdSpaceUpdateModel()` beforehand. The frozen status is also used for optimization opportunities by the `MstBridge` component.

Similarly, to *unfreeze* a model call

```
MeBool MEAPI McdSpaceUnfreezeModel   (McdModelID cm)
```

and to check the status of a model

```
MeBool MEAPI McdSpaceModelIsFrozen   (McdModelID cm)
```

Change Blocks

Change blocks are a mechanism to circumscribe changes to `McdSpace` by allowing specific changes to occur only between a pair of functions. There are two types of operations on McdSpace objects: state-query and state-modification. State-query functions can only be used when the state is well-defined, i.e. not in the process of being modified. Once modifications to the state begin to be applied (signalled by a call to `McdSpaceBeginChanges()`), the original state is no longer available for query. When the set of modifications have been completed, (indicated by `McdSpaceEndChanges()`) the new state is properly defined and ready to be queried again.

```
void MEAPI McdSpaceBeginChanges(McdSpaceID space)
```

indicates that a new set of state-modification operations will be applied to `space`, and

```
void MEAPI McdSpaceEndChanges(McdSpaceID space)
```

indicates that no more state-modification operations will be applied to `space`, i.e. the end of the changes block.

```
int MEAPI McdSpaceIsChanging(McdSpaceID space)
```

can be used to determine which mode is currently in effect, either inside (returns a non zero positive int) or outside (zero) a changes block. It is an error to call `McdSpaceBeginChanges()` inside the block, or to call `McdSpaceEndChanges()` outside the block. See the tables below for a complete list of restrictions on the use of `McdSpace` functions.

`McdSpace` functions applicable only inside the changes block: (i.e. when `McdSpaceIsChanging()` returns a non zero positive int):

Functions Valid Only Inside Change Blocks
`McdSpaceInsertModel()`
`McdSpaceInsertModelWithCulling()`
`McdSpaceRemoveModel()`
`McdSpaceUpdateModel()`
`McdSpaceUpdateModels()`
`McdSpaceEnablePair()`
`McdSpaceDisablePair()`
`McdSpaceSetModelCullingParameters()`
`McdSpaceEndChanges()`

`McdSpace` functions applicable only outside the *changes* block: (i.e. when `McdSpaceIsChanging()` returns zero):

Functions Valid Only Outside Change Blocks
`McdSpaceGetPairs()`
`McdSpaceGetLineSegIntersections()`
`McdSpaceGetLineSegFirstIntersection()`
`McdSpaceSetAABBFn()`
`McdSpaceBeginChanges()`

All other `McdSpace` functions can be used regardless of the value of `McdSpaceIsChanging()`. These are:

Functions Valid Anywhere
`McdSpaceFreezeModel()`
`McdSpaceUnfreezeModel()`
`McdSpaceIsChanging()`
`McdSpaceModelIsFrozen()`
`McdSpacePairIsEnabled()`
`McdSpaceGetModelCount()`
`McdSpaceModelIteratorBegin()`
`McdSpaceGetModel()`
`McdSpaceSetUserData()`
`McdSpaceGetUserData()`

Updating The Models

Each time objects included in the collision space are moved or rotated (their TMs have been modified), the collision space must be updated by calling

```
void MEAPI McdSpaceUpdateAll         ( McdSpaceID space )
```

to update nearby pair lists according to the latest object transform. Similarly we can update a single model

```
void MEAPI McdSpaceUpdateModel       ( McdModelID collModel )
```

which only updates the bounding volume in space associated with `collModel`. This can only be called when `McdSpaceIsChanging()` is true, and is also called implicitly via `McdSpaceUpdateAll()`.

If a model is not updated, proximities involving it will continue to be reported, but they will be based upon the bounding volume computed the last time `McdSpaceUpdateModel()` was called, which may no longer be correct.

If it is known that the bounding volume properties are not changing, then consider using `McdSpaceFreezeModel()` in combination with `McdSpaceUpdateAll()`.

The models can be *frozen* and *unfrozen* depending on the kind of behavior required. The *frozen* `McdModel`'s are models related to static objects, such as walls and floors, that should not normally move when interacting with other smaller and lighter objects.

Freezing models is a way not to waste computing resources by testing for proximity, collision and then generating useless contacts between pairs of static models. For example, a ground plane is frozen, since it will not be moved by collisions with other objects:

```
McdSpaceUpdateModel(groundCM);
McdSpaceFreezeModel(groundCM);
```

Note that the ground collision model was updated before being frozen. This is to make sure that the collision space is aware of the position of its bounding volume relative to other unfrozen models that may interact with it.

There is a difference between freezing a model and not updating it: In the latter case, pairs will continue to be reported, even if both models are not being updated; in the former, pairs in which both models are frozen are not reported. For additional details about the freeze feature, please refer to ‘Static Models’ earlier in this chapter.

Testing for Collisions

When a pair of `McdModel` objects are first detected to be in close proximity to each other, a `McdModelPair` object is assigned to that pair.

That same `McdModelPair` object will be re-used to refer to the same pair in subsequent queries, until the pair of models are no longer in close proximity to each other. After that point, the `McdModelPair` object becomes invalid, and is reused by `McdSpace` to track other new proximities as they are detected.

All potentially colliding pairs of collision models, as determined by `McdSpace`, are stored in a `McdModelPairContainer` structure. This structure is created with,

```
McdModelPairContainer* MEAPI McdModelPairContainerCreate (int size)
```

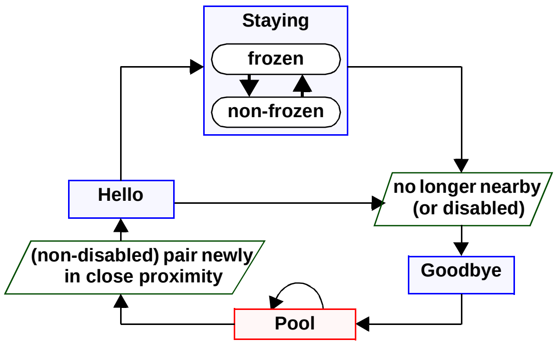
that creates a `ModelPairContainer` large enough to hold `size` model pairs.

Inside the `McdModelPairContainer` structure, there is an array containing the hello, staying and goodbye pairs. To reiterate:

- Hello pairs are `McdModel` pairs that `McdSpace` identifies as being in close proximity to each other, but that were not in close proximity after the previous call to `McdSpaceUpdateAll()`.
- Staying pairs are `McdModel` pairs that `McdSpace` identifies as being in close proximity to each other, and were also in close proximity after the previous call to `McdSpaceUpdateAll()`.
- Goodbye pairs are `McdModel` pairs that `McdSpace` had identified as being in close proximity after the previous call to `McdSpaceUpdate()`, but are no longer in close proximity after the current call to `McdSpaceUpdateAll()`.

For obvious reasons, the Hello and Goodbye pairs are often called *transient* pairs.

Here is a diagram showing the relationship between the pairs:


Staying
frozen
non-frozen
Hello
no longer nearby (or disabled)
(non-disabled) pair newly in close proximity
Goodbye
Pool

In order to retrieve a list of one type of colliding pair, an *iterator* is needed. This is an index pointing to the beginning of the sub-array of the desired type. First declare an iterator variable of the type `McdSpacePairIterator` then initialize the iterator variable for the collision space by calling

```
void MEAPI McdSpacePairIteratorBegin ( McdSpaceID collSpace,
                                       McdSpacePairIterator *iter )
```

The variable iter is not valid until this function is called. This must be called before using `McdSpaceGetPairs`, and after **McdSpaceUpdateAll** is called.

Then to retrieve successive sub-arrays of `McdModelPair` use

```
int MEAPI McdSpaceGetPairs          ( McdSpaceID space,
                                      McdSpacePairIterator* iter,
                                      McdModelPairContainer* pairContainer )
```

to get all pairs of `McdModel` objects in `space` that are in close proximity to each other and put them in `pairContainer`. This can only be called outside of the changes block, i.e. when `McdSpaceIsChanging()` is false. The return value indicates overflow condition (call again to get remaining pairs).

The function `McdSpaceGetPairs()` gets all pair-events since the last call to `McdSpaceUpdateAll()`. By default, this includes goodbye events due to models being removed from the `McdSpace` via `McdSpaceRemoveModel()`. Close proximity is determined by a conservative bounding volume for each model, and does not necessarily indicate contact or penetration between the two models.

The `McdModelPairContainer` structure filled in by `McdSpaceGetPairs()` identifies these states and transitions by holding three distinct lists (hello, staying and goodbye) of `McdModelPair` objects. Over a sequence of time-steps, each `McdModelPair` goes through the same life cycle sequence:

- It first appears as a hello pair.
- It reappears zero, one or multiple times as a staying pair
- When no longer in close proximity, last appears as a goodbye pair.

In the step after that, the pair is invalidated and ready for reuse. This setup guarantees two key properties that enable efficient management of collision response:

- the identity of `McdModelPair` objects is preserved across successive time-steps.
- every hello event will eventually be matched by a goodbye event.

Note that goodbye and hello pairs share the same array block. In the case of overflow, the goodbye pairs are filled first and should be processed first, because it will free up memory further down the control chain. After goodbye is filled, hello and staying are filled independently.

Below is a typical loop for this process. Note that the loop begins with a `McdSpaceEndChanges()` function, and ends with a `McdSpaceBeginChanges()` function. This means that the operation of retrieving pairs of models and testing them for intersection must occur outside of the change block.

To handle collisions every simulation step, in the main program loop do something like:

```
MeBool pairOverflow;
McdModelPairContainer* pairs;

/* Move the object in the simulation */
move();
```

```
    /* end the change block so we can handle collisions */
    McdSpaceEndChanges(space);
    McdSpacePairIterator spaceIter;
    /* Initialise iterator */
    McdSpacePairIteratorBegin(space, &spaceIter);

    do {
        /* Loop through the model pairs, handling each pair */
        McdModelPairContainerReset(pairs);
        pairOverflow = McdSpaceGetPairs(space, &spaceIter, pairs);
        /* Handle goodbye pairs */
        McdgoodbyeEach(pairs);
        /* Handle hello pairs */
        McdhelloEach(pairs);
        /* Handle collision */
        handleCollision(pairs);
    } while(pairOverflow);

    /* Start the change block again */
    McdSpaceBeginChanges(space);
```

The `move()` subroutine, that updates each collision model position, is located inside the changes block. The change block indicates a new step in the life-cycle of `McdSpace` pair-events: goodbye pairs reported in the previous call to `McdSpaceGetPairs()` are no longer valid after `McdSpaceBeginChanges()`, and will not appear in the next call to `McdSpaceGetPairs()`.

`McdHello()` must be called on a pair before using it in any queries, such as `McdIntersect()` or `McdSafeTime()`. It prepares `McdSpace` by allocating internal cached data, preprocessing some information for use by `McdIntersect()` or `McdSafeTime()`, or selecting the appropriate algorithm based on some of the pair's `McdRequest` values. After `McdHello()` has been called, `McdModelPairReset()` and `McdModelPairSetRequestPtr()` calls are not allowed on this pair until `McdGoodbye`() is called. Otherwise, undefined behavior may result.

Conversely, a `McdGoodbye()` call should be made on any goodbye pairs obtained from `McdSpace` for which `McdHello()` has been previously called, to inform `McdSpace` that no more queries will be performed on these pairs. This will free up any internal data associated with any of these pairs, if applicable. As a consequence, the `McdIntersect()` and `McdSafeTime()` queries could no longer be called on this pair.

Note that it is important to process all the goodbye and hello pairs by calling the appropriate functions (or using Mst utility functions that do this) after an `McdSpaceEndChanges` call and before an `McdSpaceBeginChanges` call. Since the hello and goodbye pairs are transient, these hello and goodbye events will be missed and either pairs will not be correctly initialized (`McdHello`) or memory will not be freed (`McdGoodbye`) causing undefined behavior.

```
    void MEAPI McdHelloEach   ( McdModelPairContainer* pairs )
```

and

```
    void MEAPI McdGoodbyeEach ( McdModelPairContainer* pairs )
```

can be called instead. This will accomplish the same thing as `McdHello()` or `McdGoodbye()`, but for a whole `McdModelPairContainer` structure at once.

Now that we know which pairs of collision models are near each other and possibly colliding (hello and staying), we must check if there is an actual collision between each of them. To test for potential collisions between a pair of collision models call

```
    MeBool MEAPI McdIntersect ( McdModelPair *pair, McdIntersectResult *result,
                                MeReal time )
```

This will find an appropriate algorithm for the given pair of collision models, and compute the desired intersection data, given an `McdModelPair` structure of nearby collision models and a `McdIntesectionResult` structure that will hold all relevant collision information, including newly generated `McdContact` objects for that pair. The return value is `1` if the proper collision function was available, `0` otherwise. The members of the `McdIntersectResult` structure are given in the table

Field Name	Usage
`McdModelID` model1	one of the collision models.
`McdModelID` model2	the other collision model.
`McdContact*` contacts	array of contacts to be filled.
`int` maxContactCount	size of array.
`int` contactCount	number of contacts returned in array.
`int` touch	`1` if objects are in contact, `0` otherwise.
`MeVector3` normal	representative normal of the set of contacts.

An array of `McdContact` objects and its length must be provided before calling `McdIntersect()`:

```
McdInteractionResult result;

/* Set the maximum number of contacts to be returned */
result.contactMaxCount = 10;
result.contacts = contacts;

/* Get a pair of models */
McdModelPairID pair = pairs->array[i];

/* Test for intersection */
McdIntersect(pair, &result);
```

Cleaning Up

When the simulation is completed, `MstUniverseDestroy` can be used to destroy all the `Mcd` variables and structures.

A model and its related geometry cannot be destroyed while the model is still in a space. Models must be removed from space using

```
MeBool MEAPI McdSpaceRemoveModel( McdModelID collModel )
```

function before being destroyed. This takes the model out of its space and returns `1` if successful, `0` if not. Any associated staying or hello pairs in that space will become goodbye pairs. This can only be called when `McdSpaceIsChanging()` is true.

To remove model(s) from space then a procedure such as (in pseudo-code)

```
McdSpaceBeginChanges(space)     [ if not in a change block ]
McdSpaceRemoveModel(model)      [ repeat as needed ]
McdSpaceEndChanges(space)
McdSpaceGetTransitions(space)   [ must be done out of change block otherwise
                                  currently no removed pairs output ]
[
     handle all transitions.
     i.e. goodbye pairs from space involving the removed model(s).
     The model must be valid, including a valid geometry up to this point.
]

McdGeometryDestroy(McdModelGetGeometry(model)) [ optional - may still be
                                                 being used by other models ]
McdModelDestroy(model)
```

can be employed.

All collision models that reference a particular geometry must be destroyed before the geometry itself is destroyed. This is because geometries are reference counted i.e. every persistent reference (in a collision aggregate or a collision model) to a geometry increases the reference count by 1 while that model exists. If you destroy a geometry with non-zero reference count, you get a warning in debug builds.

To remove a geometry use

```
    void MEAPI McdGeometryDestroy   ( McdGeometryID geometry )
```

The `geometry` is no longer valid after this call.

To remove a collision space use

```
    void MEAPI McdSpaceDestroy      ( McdSpaceID space )
```

This deletes a `McdSpace` object performing memory deallocation.

```
    void MEAPI McdFrameworkDestroyAllModelsAndGeometries( McdFrameworkID )
```

destroys all models and geometries allocated by a framework. Finally free up the memory used by the framework by calling

```
    void MEAPI McdTerm              ( McdFrameworkID )
```

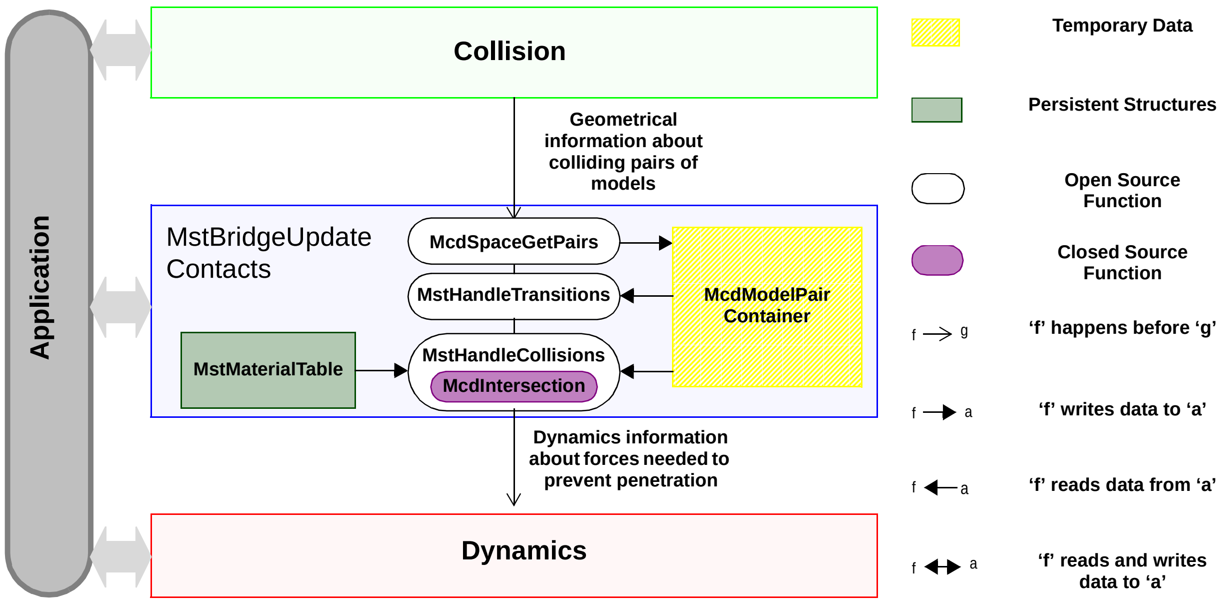
to shut down the `Mcd` framework. This frees all memory allocated in `McdInit()`, and any memory that may have been allocated by any `RegisterType()` or `RegisterInteraction()` calls.

The Bridge

Introduction

The Karma Bridge provides a way of passing information between Karma Dynamics and Collision when both are used.

The Karma Bridge resides in the Mst library (hereafter called Mst) which also contains utilities for simplifying the simultaneous use of Karma Collision and Dynamics. Mst provides an API that is a bridge between Karma Dynamics (Mdt libraries) and Karma Collision (Mcd libraries). A schematic of the Mst integrated architecture is shown below.



Mst ensures that all necessary operations are carried out at the right times, and that the memory is efficiently managed. These operations include:

- Updating the transformation matrix of each collision model and its corresponding dynamics body.
- Obtaining data about intersections and contact points for each collision event.
- Preparing and sending contact data to Karma Dynamics.
- Ensuring that the dynamic properties of each contact are set to the values appropriate for the given pair of models.

Mst automates all of these processes, and provides high-level control over each aspect of its activity. Use of the Mst library ensures that combined use of dynamics and collision is efficient and takes full advantage of the features available in both, providing useful functions for creating and simulating objects in both. An `MstUniverse` contains an `McdSpace`, an `MdtWorld`, an `MstBridge`, and some buffers for moving contacts between Mcd and Mdt. The function `MstUniverseCreate` creates these in one function.

A collision `McdModel` structure is created for each object. For non static collision models a dynamic `MdtBody` is associated with it. No dynamic body is associated with static collision models.

Mst contains functions for creating these together (for convenience), and for associating them when they have been created with the Mdt and Mcd APIs, using `McdModelSetBody`. The mass and inertia tensor of an `MdtBody` can be automatically set to a sensible default using the collision geometry and a density. `MstUniverseStep` calls the collision farfield and nearfield tests, and steps the dynamics.

Creating a Universe

Mst contains functions that coordinate collision and dynamics. One of the most used is the `MstUniverse` structure (see `MstTypes.h`).

`MstUniverse Members`	Description
`MdtWorldID world`	Handle to the Dynamics world
`McdFrameworkID frame`	Handle to the Collision framework
`McdSpaceID space`	Handle to the Collision farfield
`MstBridgeID bridge`	Handle to the Karma bridge
`MstUniverseSizes sizes`	Structure containing information about the size of the universe

`MstUniverseSizes` contains information that allows Karma to allocate memory to contain all the physics components in this `MstUniverse`. An `MstUniverseDefaultSizes` structure contains default values for a typical universe. You should adjust these to suit your application. The `MstUniverseDefaultSizes` are shown below.

`MstUniverseSizes Member`	Default Value	Description
`dynamicBodiesMaxCount`	100	Maximum number of dynamic bodies allowed.
`dynamicConstraintsMaxCount`	500	Maximum number of dynamic constraints (joints & contacts) allowed.
`collisionUserGeometryTypesMaxCount`	0	Maximum number of user collision geometry types allowed.
`collisionModelsMaxCount`	100	Maximum number of collision models allowed.
`collisionPairsMaxCount`	500	Maximum number of simultaneously overlapping models allowed.
`collisionGeometryInstancesMaxCount`	100	The number of additional geometries, geometries not directly associated with a model. Used when implementing aggregates.
`materialsMaxCount`	10	Maximum number of materials allowed which must be at least 1. Materials are used to define the interaction between two bodies, the parameters are set on a per material pair basis.
`lengthScale`	1	Length of a medium-sized object
massScale	1	Mass of a medium-sized object

To set up NBALLS bouncing freely on a ground plane, include Mst, declare variables and fill out constants and structures:

```
#include "Mst.h"

MstUniverseSizes sizes;
/* Set some default sizes */
sizes = MstUniverseDefaultSizes;

/* One collision model for each ball, and one for the ground plane */
sizes.collisionModelsMaxCount = NBALLS + 1;
```

```
/* In an extreme case all the balls will be touching another ball or the plane */
sizes.collisionPairsMaxCount = NBALLS * NBALLS;
/* One dynamics body for each balls */
sizes.dynamicBodiesMaxCount = NBALLS;

/* In an extreme case all the balls will have six contacts */
sizes.dynamicConstraintsMaxCount = NBALLS * 6;

/* We have one material for the balls and one for the ground plane */
sizes.materialsMaxCount = 2;

/* We are not using aggregates so there are no 'unused' geometries */
sizes.collisionGeometryInstancesMaxCount = 0;
sizes.collisionUserGeometryTypesMaxCount = McdPrimitivesGetTypeCount();

/* Now use the size structure to create the Universe */
MstUniverseCreate(&sizes);
```

Now that the `MstUniverseSizes` structure contains all the important quantities used to manage the `MstUniverse`, this can be passed to `MstUniverseCreate`, which will return an ID to our `MstUniverse`.

Fitting Out the Universe

The universe can be populated with dynamic and static objects. Static objects are created with `MstFixedModelCreate`, which takes a collision geometry and a transformation matrix containing the static object's position and orientation. The collision model is created, positioned and inserted into the collision space, and then updated and frozen in place.

When creating a collision model and a dynamics body, they need to be attached such that collision contacts created during `MstBridgeUpdateContact` are associated with the correct dynamics body. To do this use `McdModelSetBody` to associate the collision and dynamics bodies.

Dynamics bodies contain no information about their spatial extents, except for mass distribution. If the body has an associated collision model, then this information can be used to calculate approximate values for the mass and inertial tensor of the dynamics body. To do this call `MstAutoSetMassProperties` providing collision geometry and average body density as parameters.

`MstModelAndBodyCreate` creates and associates a `McdModel` with a `MdtBody.` The `McdGeometry` and density that are provided are used to calculate sensible values for the mass and inertia tensor of this body. The model is automatically inserted into the universe collision space. This effectively calls `McdModelSetBody` and `MstAutoSetMassProperties`.

Resetting

If a collision model has a dynamic body associated with it, the position can be reset to the world origin, the orientation set to default and the velocity set to zero with `McdModelDynamicsReset`.

Setting the Universe in Motion

```
void MEAPI MstUniverseStep(const MstUniverseID u, const MeReal stepSize)
```

evolves the universe by

- updating the collision space.
- passing the generated contact to the dynamics world via the bridge.
- evolving the world for a time-step by executing the following code.

```
McdSpaceUpdateAll(u->space);
MstBridgeUpdateContacts(u->bridge, u->space, u->world);
MdtWorldStep(u->world,stepSize);
```

Cleaning Up

Before closing the application, free up the memory used by the structures and objects using `MstFixedModelDestroy`, `MstModelAndBodyDestroy`, and `MstUniverseDestroy`.

`MstUniverseDestroy` not only de-allocates memory and destroys a `MstUniverse` simulation container, but will also destroy

- all `MdtBodies` and all `MdtConstraints` regardless of whether they are enabled or disabled.
- all `McdModels` that have been created, regardless of which `McdSpace` they are in.
- the `MstBridge`.

Building a Bridge

Each newly created universe contains what is referred to as a *bridge*. An `MstBridge` object is responsible for all operations and communication between Karma collision and dynamics and is used during `MdtStep` to pass contact geometry information from collision to dynamics.

To use collision and dynamics together when not using the `MstUniverse` container, a bridge must be created from scratch using `MstBridgeCreate`.

Each `MdtBody` receives a `MstMaterialID` identifier that acts like an index in a matrix of contact parameters and callback functions, also called the *material table*. As soon as a contact is created, the `MstMaterialID` of both bodies in contact is used to retrieve the proper `MdtContactParams` structure and the name of the three callback functions it is using. The `MdtContactParams` interface can then be used to modify friction, restitution etc. for a pair of materials.

For example, if a rubber ball with a `MstMaterialID rubber` had fallen on a wooden floor with a `MstMaterialID wood,` then the newly created contact would have used the `MdtContactParams` structure and the appropriate callback functions located at `(wood, rubber)` in the material table. The `MstBridgeGetNewMaterial` function creates a new `MstMaterialID`.

When just using the single default material, i.e. there are no user defined materials, the material table contains one material entry. The default `MstMaterialID`, returned by the macro function `MstBridgeGetDefaultMaterial` is `0`. The three default contact callback functions are valid for the default material. This is why the value of `materialsMaxCount` must be at least `1` in the `MstUniverseSizes` structure.

An `MstMaterialID` identifier needs to be attached to every `McdModel` by using the `McdModelSetMaterial` mutator function. The `McdModelGetMaterial` accessor function returns the `MstMaterialID` identifier of a `McdModel`.

So if we wished to make two materials, say one for a ball object and one for the ground plane, we could use code such as this to set the restitution to 1.0 (very bouncy).

```
MstMaterialID ballMaterial,floorMaterial;
MdtContactParamsID p;

ballMaterial  = MstBridgeGetNewMaterial(MstUniverseGetBridge(universe));
floorMaterial = MstBridgeGetNewMaterial(MstUniverseGetBridge(universe));

/* Get the contact params */
p = MstBridgeGetContactParams(MstUniverseGetBridge(universe),
                                ballMaterial, floorMaterial);
/* Set the restitution for this pair of materials */
MdtContactParamsSetRestitution(p,(MeReal)1.0);

/* Now set the materials to the appropriate models */
McdModelSetMaterial(ball,ballMaterial);
McdModelSetMaterial(floor,floorMaterial);
```

Callback Functions

A callback function is a user defined function that is automatically called when a predetermined event takes place. Callback functions are designed to return a specific type of variable and to take specific arguments.

A callback provides users with a means of modifying a given structure before it is used. They provide control over `McdContacts` before they are converted to `MdtContacts`. Callbacks can be used for many applications:

- adding a sound everytime there is a collision, such as a ricochet noise as a bullet rebounds of a wall.
- dynamically altering properties of the generated contacts, such as setting the slip of a car wheel to depend on its rotation speed.

There are three different callback functions that can be used. The functions `MstBridgeSet*CB` and `MstBridgeGet*CB` assign and retrieve these. The asterisk denotes the callback type, namely `Intersect`, `PerContact` or `PerPair` . To set or retrieve the callback the function should be passed the `MaterialIDs` that correspond to the interaction that are to be processed in the callback. The callbacks themselves must return a MeBool to indicate whether the contact should be kept and acted upon.

Intersect

Set the optional per-intersection user callback for the given pair of materials. This will be executed once for each pair of colliding models with the given materials, with the `McdIntersectResult` and the set of collision contacts. It allows control over `McdContacts` (geometrical contact information) before they are converted to `MdtContacts` (dynamic contact information including surface properties).

To obtain the whole set of collision contacts in one go use the intersect callback. The contacts can then be culled by the developer.

PerContact

Set the optional per-contact user callback for the given pair of materials. This will be executed once for each contact between models with the given materials, with the `McdIntersectResult` and the Mcd and Mdt contacts. It allows control of parameters in the dynamics contact based on data contained in the collision contact. If the callback returns 0, the `MdtContact` will be deleted.

The per-contact callback should be used to modify the properties of a dynamic contact using information from the collision contact that created it. The per-contact callback is slightly less efficient to use than either of the other two, because it is called many times for each intersection rather than just once.

PerPair

Set the optional per-pair user callback for the given pair of materials. This will be executed once for each pair of colliding models with the given materials, with the `McdIntersectResult` and the set of dynamic contacts. It allows, for example, further culling of dynamics contacts based on the entire set of contact values. If the callback returns 0, the set of contacts will be deleted.

The per-pair callback is used to operate globally on the set of dynamics contacts. This would be used to add an extra contact between two objects.

As an example we could create a callback that would allow a ball to pass through a wall if it is travelling fast enough, and we make it slow down slightly to simulate the resistance as it passes through the wall. First, set the callback using the wall and ball materials to tell Karma which interaction is of interest.

```
/* set the call back between the wall and ball material */
MstBridgeSetPerPairCB(universe->bridge,mat_Wall, mat_Ball, WallCollisionCB);
```

Define the callback to process the collision

```
MeBool MEAPI WallCollisionCB(McdIntersectResult* c, MdtContactGroupID cg){
     MeVector3 vel;
     MdtBodyID body

     /* First find which of the models is the ball */
     if (McdModelGetMaterial(c->pair->model1)==mat_Ball)
     {
          ball=McdModelGetBody(c->pair->model1);
     }
     else
     {
          ball=McdModelGetBody(c->pair->model2);
     }
     MdtBodyGetLinearVelocity(ball,vel);
     if (MeVector3Magnitude(vel)>thresh){

          /* Body is going fast enough to pass through, so slow the ball down */
          MeVector3Scale(vel,(MeReal)0.5);
          MdtBodySetLinearVelocity(body,vel);
          /* Return 0 to delete this contact */
          return 0;
     }
     else
     {
```

```
                /* return 1 to maintain this contact and make the ball bounce off */
                return 1;
            }
        }
```

Note that there are default callbacks setup in `MstUniverseCreate` when it calls `MstSetWorldHandlers` that ensure that when a dynamic body stops moving its collision model is frozen, and when an interaction causes it to start moving again, its collision model is unfrozen. To coordinate collision and dynamics via the bridge when not using an `MstUniverse`, call `MstSetWorldHandlers`.

A bridge created outside of an `MstUniverse` structure will need to be destroyed explicitly with `MstBridgeDestroy`.

Creating Good Simulations with Karma

After mastering the basic usage of Karma there are several issues that should be borne in mind so that well-behaved, efficient and reliable simulations can be constructed. These are discussed in this chapter.

Points to Remember When Using Karma

The following list may prove useful as a troubleshooting guide. Each point is further explained below.

1 Always use the check build when developing.

2 The most common reason for a 'crash' is that the pool of bodies or constraints in an MdtWorld is not big enough.

3 Care must be taken when setting a range of certain object parameters, such as mass and size, so as not to lose accuracy.

4 If large masses are needed, or large forces used, the parameter epsilon may need to be decreased.

5 Applying forces or torques to bodies will not slow down a simulation.

6 Adding forces that change rapidly, for example springs, between time steps can cause simulations to gain energy and become unstable. Use constraints instead.

7 Ensure that an object’s inertia tensor corresponds to its mass and collision size, and hence is sensible for the torque applied to it.

8 To speed up a simulation use small, separate partitions.

9 A body can be moved by

- applying a force to it.
- setting its velocity.
- setting its position.

Always use a force where possible. While the velocity or position can be set directly, care should be taken when doing this.

10 The visualized physical object that is created is composed of SEPARATE dynamic, collision and graphics objects.

11 Use a single dynamic body. Do not try to fix dynamic bodies rigidly together.

12 Mass distribution, Moment of Inertia, Inertia Tensor and Mass Matrix refer to the same property. Set it using MdtBodySetInertiaTensor().

13 Use the fast spin axis option when hinging two objects together that will rotate at high speeds.

14 Avoid Over-Determinancy.

15 Set contact softness to prevent contact jitter.

16 Use joint limits rather than contacts.

17 Position and enable bodies before assigning them to a joint. Then set the joint position and enable the joint.

Further details:

1 Always use the check build when developing.

The check / debug builds provide lots of useful warnings and should be used. Note that MathEngine does not provide debug builds itself. Developers should create a debug build for publicly released libraries.

2 The most common reason for a 'crash' is that the pool of bodies or constraints in an MdtWorld is not big enough.

If your simulation does crash, check that you are using the check / debug library and look for any warnings. The most common problem (the most common reason for a 'crash’) is that the memory pool assigned to hold the body and constraint information in the MdtWorld is not big enough. You should increase this. Only use the release libraries to get a measure of performance of a working check / debug build.

3 Care must be taken when setting a range of certain object parameters, such as mass and size, so as not to lose accuracy.

For stable efficient simulations, it is sensible to keep lengths and masses in a relatively small range around unity. How critical this is depends on the precision that is being used. Because one of the main requirements for Karma for the entertainment market is that it be fast, single precision float is used, hence it is sensible to keep all length and mass numbers in the range of 0.01 through 100, for example. For higher precision, a much wider range is acceptable. This will help ensure that numerical error and inaccuracy do not become too much of a problem.

Note that problems can occur if you have a large range of values for certain object properties. For example, certain ratios relating to mass must be close to one. Here are some guides to world construction that should be followed:

- Largest mass to smallest mass ratio should ideally not exceed 100.
- It is sensible that the velocity of an object is such that the distance it moves in a time step is less than the object size. The reason for this is that collisions are more easily detected.

The angular velocity should be such that the angle swept out in a time step does not exceed 60 degrees. The exceptions to this are in the carwheel joint that was specifically designed for high-speed rotation, or for bodies where the fast rotation axis (keaBody.fastSpinAxis) has been set. This is because floating-point is most accurate for values in the middle of its range. In decimal equivalent, the working range is between about 10e-6 & 10e+6. Adding and subtracting values whose exponents differ by more than this results in errors in the mantissa. Similarly multiplying 2 numbers with large exponents and dividing numbers with small exponents causes problems in that the exponent suffers overflow and underflow respectively.

The critical point is that mass scales as the third power of object size and moments of inertia scale as the 5th power. E.g. A cube has mass = (density)x(volume) = (density)x(length cubed). This cube has inertia = (mass)x(length squared) where mass is given above. If you have 2 objects, one of size 0.1, the other of size 10, a difference factor of a mere 100, the mass difference is of the order $10^3/0.1^3 = 10^6$, and the inertia difference = $10^5/0.1^5 = 10^{10}$. This demonstrates how accuracy can be lost.

Consider an articulated system consisting of heavy and light objects. The heavy objects can transfer their energy to the light bodies causing them to move quickly because of conservation of momentum. Instabilities will occur if bodies move faster than they can be reasonably simulated. This problem can be prevented by ensuring that the ratio of large to small masses in the system is not too high. For example, a ratio of 100:1 is a reasonable single precision maximum in many situations.

4 If large masses are needed, or large forces used, the parameter epsilon may need to be decreased.

If you use larger masses, you may need to decrease 'epsilon' (using MdtWorldSetEpsilon). Epsilon is a ‘global constraint softness' and directly affects the constraint solution. If the forces in your simulation are stiff, you will need to make constraints (contacts, joints) 'harder'.

For large or small epsilon, the maths describing the system will be solved, but the way the solution is arrived at results in different visual behavior of the system.

As epsilon decreases it takes longer to home in on a mathematical solution within the required bounds. As an example, consider a large mass colliding with the ground. The contact is modelled by a spring. Epsilon relates to the stiffness of the spring. For more realistic behavior a stiff spring is needed i.e. small epsilon. However, it will take longer to arrive at a solution within the bound specified. You might get a warning message saying that the number of LCP (don’t worry about LCP here) cycles have been exceeded. This means that while the solution is almost certainly going to be good enough for your application it might not have fallen into the required range within the number of LCP cycles specified.

To decrease the time (i.e. number of LCP cycles) to find a solution of the given accuracy you should increase epsilon. However, this makes it more difficult to model systems that require stiff springs - e.g. the large mass system described above. A large mass object in collision with the ground will show ‘springiness’ in the contact. Hence the term ‘global constraint softness' that is used to describe epsilon physically.

Ideally you should set the largest epsilon possible so that your system behaves properly as visually observed. We recommend that you tweak with epsilon. While epsilon is not limited, a sensible working range is between 10^{-5} and 0.1.

5 Applying forces or torques to bodies will not slow down a simulation.

and

6 Adding forces that change rapidly, for example springs, between time steps can cause simulations to gain energy and become unstable. Use constraints instead.

Applying forces or torques to bodies will not slow down your simulation. However, adding forces that change rapidly between time steps (e.g. springs) can cause your simulation to gain energy and become unstable. You should use a constraint for this instead.

7 Ensure that an object's inertia tensor corresponds to its mass and collision size, and hence is sensible for the torque applied to it.

NOTE: **Version 1.x of Karma Dynamics internally overrides the inertia tensor set by the user with a uniform tensor that is a multiple of the identity matrix. This design choice has been made to reduce a number of stability problems that arise due to momentum transfer from high to low inertia axis. These can produce high angular velocities, especially during collisions, and make it harder to construct robust simulations. Karma 1.1.1 contains experimental code for non-spherical inertia tensors, although there may be some dynamical stability issues which you will need to deal with at the application level.**

Ensure that an object's inertia tensor corresponds to its mass and collision size. Even if it is a crude approximation such as that of a sphere bounding your object, this will help. If you create a large heavy body with a small inertia tensor, this can cause jittering and odd behavior. This is because physically it would be like, for example, creating a sphere with nearly all of its mass in the center. It is easy to rotate this object because of its small inertia. If you apply even what appears to be a sensible force (strictly a force that acts to rotate the object i.e. a torque) in comparison to the object size and mass, the small inertia results in rapid rotation.

Objects having inertia tensors with components that are large on one axis and small on another axis are inherently unstable. Long thin objects are susceptible to gaining rotational energy on the long axis, about which the moment of inertia is low. This can lead to rotations that are difficult to simulate accurately. If an object is rotating quickly about a particular axis then the fast spin axis (please refer to point 13 below) option should be used.

8 To speed up a simulation use small, separate partitions.

The speed of simulation is, in the worst case, proportional to the number of constraints in a partition* cubed. For example, a stack of 5 boxes can be 8 times faster than a stack of 10 boxes ($10^3/5^3$). Hence you can speed up your simulation by using lots of small, separate partitions.

* A 'partition' is a group of bodies connected by constraints. Constraints to the 'world' do not connect partitions. So two boxes sat on the ground are two partitions, but two boxes in a stack are 1 partition.

9 A body can be moved by:

- applying a force to it.
- setting its velocity.
- setting its position.

Always use a force where possible. While the velocity or position can be set directly, care should be taken when doing this.

When you want to reposition or move an object, there are three ways that you can do this using Karma. You can set the object position directly, set the object velocity or apply a force to it. The preferential order for doing this is wherever possible use forces, the next best option is to set the velocity, and finally repositioning directly which will work but is not recommended. To think about this you should consider how objects move in the real world. Leaving quantum behavior well alone, within the Newtonian framework - our perceived reality - objects do not simply move instantly from one place to another (like setting position). Similarly, they don't suddenly develop a particular speed i.e. they are not stationary one instant and the next they are moving (like setting velocity). Rather there is a smooth increase in the velocity and position changes gradually as a force is applied. It is the same with the simulation software. While you can set position and velocity, you should use forces wherever possible. Mathematically we say that the functions should be continuous i.e. there are no sudden kinks or discontinuities that correspond to position or velocity being set rather than force. Setting position results in bigger discontinuities than setting velocity, hence the reason for 'preferring' velocity to position.

A problem from setting position directly is that an object may inadvertently be place inside another object. Or if you reposition an object that is joined to another body or that is part of another structure.

An example of a problem from setting velocity directly is that an object may unintentionally be directed at high speed toward another object. Or if a velocity is given to an object attached to another or that is part of another structure

However, please note that if your object is isolated and is being moved to a position not occupied by another object, then setting position or velocity is okay. Likewise, you can do this as long as you take care of the objects that the object you want to move is attached to. Hence, either detach it or move the entire structure (partition).

10 The visualized physical object that is created is composed of SEPARATE dynamic, collision and graphics objects.

No additional notes.

11 Use a single dynamic body. Do not try to fix dynamic bodies rigidly together.

Do not try to fix dynamic bodies rigidly together, but rather use a single dynamic body. Fixing objects rigidly together causes a large performance hit - relatively speaking, as the fixed joint is not necessary - on the constraint solver. The constraint solver works out the physical properties of the state of the system as the system evolves. A fixed joint constrains the six degrees of freedom (three linear and three rotational) between two objects and as a result is the most computationally expensive constraint to deal with. You can create aggregate structures by attaching a composite collision model and several graphic objects to a single rigid body. The only knowledge that a dynamic body has about its extent is through its mass distribution. You should set the mass matrix of your single dynamic body to something close to that of your perceived structure consisting of multiple rigid dynamic bodies. The exception to this is when you need a more accurate inertia tensor than the approximate one calculated by the solver - see point 7 above.

12 Mass distribution, Moment of Inertia, Inertia Tensor and Mass Matrix refer to the same property. Set it using MdtBodySetInertiaTensor().

No additional notes.

13 Use the fast spin axis option when hinging two objects together that will rotate at high speeds.

Every rigid body can have an optional fast spin axis specified, about which it rotates. If this is set, and it must be done at each time step, it alters the way that the body's orientation is updated at the end of every time step.

This alternative “fast spin” update is more accurate in the case where the body is spinning quickly around the fast spin axis, and relatively slowly around the other axes. This is particularly useful in the case of a wheel on a car, that may be rotating very quickly. Using the standard orientation update may result in a large error that makes the wheel's hinge axis appear to bend.

14 Avoid Over-Determinancy.

A system is described as being over-determined if there are more joints and contacts than are actually required to constrain the motion of the bodies in that system. Extra contacts result in an over-determined system because one of them is redundant. Because there are more contacts, the solver requires extra time to resolve the contact forces. However, this should not cause problems unless there are too many extra contacts, or the system is over-constrained in a way that means there are no consistent solutions. Some examples include:

- When simulating a box resting on a ground plane, three contact points between the box and the ground are the minimum to constrain the box to rest on the plane. However, it may be advantageous to use four contacts, since this may result in a system which is slightly more stable and will be automatically disabled more quickly.
- A hinge joint removes five degrees of freedom (DOFs) between the two connected bodies. Imagine modeling a hinge as two ball-and-socket joints spaced a little distance apart on the hinge axis. This constrains the bodies in the correct way, i.e. the correct hinged motion is produced, but the two joints together take away six DOFs (three each). This is one more than is necessary for a hinge.

In some circumstances, more contacts can be generated. Consider a box in a corner. Generally more contacts are generated than are strictly necessary, but it can be hard for a physics simulator to recognise the redundant contacts. When you view contact information as boxes are moved around on surfaces and stacked on one another, contacts come on or off and often the number fluctuates as the system changes. However, these reduce quickly as the system settles down and the solver works out the forces.

Ideally over-determinacy would never arise, because every rigid body system would be described optimally. This is not so easy to achieve in practice - it can be difficult to tell whether a given system is, or is not, over-determined.

Here are a few guidelines:

- Use the minimum number of constraints between objects that gives the correct behavior. Generally speaking, the fewer the constraints, the faster the simulation. Finding this minimum involves a certain amount of trial and error.
- Use the correct joint types for the required motion. E.g. don't use two ball-and-socket joints to model a hinge.
- Don't have conflicting joints between the same bodies, i.e. don't use more than one joint to constrain the motion of the bodies in the same way.
- If in doubt, increase `epsilon`. This will always have the effect of reducing the over-determinacy of any system.

15 Set contact softness to prevent contact jitter.

When two objects collide, there will be some initial inter-penetration. The amount of penetration depends on how fast the two objects were going before they collided.

After collision, Kea's projection feature will push the objects apart to reduce the penetration to zero. However, sometimes Kea will push the objects too far, and the contact will be broken. This can be a problem, for example, when objects are resting on the ground. When the contact is broken, the object will “fall” a short distance into the ground, the contact will be re-made and the object will be pushed out again. This process can result in resting objects that jitter or twitch from time to time.

One solution is to set the softness option on the contact. This will cause two objects that are being forced together to naturally inter-penetrate slightly, preventing contact breaking:

- Set `keaContactOption` to `keaContactOptionSoft`.
- Use `mdtContactSetSoftness()` to set the degree of softness. A fairly small number, like 0.0001, is usually suitable. A larger number will result in more natural penetration.

There is no efficiency loss in using soft contacts.

16 Use joint limits rather than contacts.

Hinge and prismatic joints can have their movement limited to prevent self-collision. Wherever possible, joint limits should be used rather than contacts to control the movement of joints. Using a contact to prevent movement of two objects connected by a joint, rather than joint limits, is more computationally costly. This is because more contacts are generated and there is additional collision detection.

17 Position and enable bodies before assigning them to a joint. Then set the joint position and enable the joint.

Positioning the joint and attaching the bodies in the wrong order can result in objects jumping around when a simulation starts. To avoid this, set up and enable bodies before applying constraints.

Karma Project Usage

The following section contains Q&A obtained in part from field experience gained by implementing Karma in developer projects.

Dynamics

How do I attach a body to the world at a fixed point?

There are 2 options:

- Attaching a body to the world with a joint. For example, a ball and socket joint can be used to attach a body to the world that is free to rotate.
- Fixing a body in space so that all 6 degrees of freedom (3 linear and 3 rotational) are constrained. This is effectively fixing a dynamic body to the world so that it becomes part of the world.

To use a joint to attach a body to the world the body should first be positioned and enabled. Bodies need to be positioned and enabled before being joined in Karma. A joint should be disabled before a joint body is changed. The Karma Debug libraries will give a warning if a joint body is changed when a joint is enabled.

For an object fixed in space, a dynamic body is not required. The reason for this is that it is unnecessary and, because the 6 degrees of freedom are constrained, the matrix that needs to be solved to work out object positions and velocities in the world is unnecessarily complicated. For each extra constraint, one row is added to the matrix. Use a collision (and render) model only and fix the collision model in space. Any additional objects that need to be attached to this body should be jointed directly to the world if a joint is needed, or their collision model used as with the first body.

Can I obtain forces from contacts to use for extra gameplay elements? E.g. such as the sound of an impact or damage to an object.

You check at each time step to see if objects are in contact. The force that you need to check for damage in joint can be obtained using:

```
void MEAPI MdtConstraintGetForce ( MdtConstraintID c,
                                       unsigned int bodyindex,
                                       MeVector3 f )
```

This returns the force, in the world reference frame, applied to a body by constraint c on the previous time step.

To generate sounds between two objects colliding you first need to work out the forces from the contacts. When two objects collide they generate a contact group. In the per pair callback, obtain the first contact by using `MdtBodyGetFirstContact` function, and then `MdtBodyGetNextContact` for the others until 0 is returned. Use `MdtContactGetForce` to obtain the collision force for each of the contacts.

I have dropped an object onto a surface, which works fine. However, when I apply an explicit force to it nothing happens, except when another object touches it, sending it quickly off in a random direction. What is happening?

What is happening is that the body has come to rest on the ground and been automatically disabled. When the explicit force is applied it does not re-enable the body and what happens is that the force accumulates in the system until the body is enabled. Enabling will occur when the other object touches it. You will need to enable the body when applying an explicit force.

What's the best primitive to use for wheels?

You may find that using spheres for wheels works better than using cylinders. A sphere wheel model usually only has one contact with the ground. This makes tuning the handling easier, it’s much easier to ensure you have good contact information when using a simpler primitive, and the test is a bit faster.

What about steering and suspension?

First have a look at the CarTerrain demo in MeTutorials for an example of running a simple car on an RGHeightfield.

You will probably want to 'orient' the friction direction to lie along the direction of the wheel. This allows you to set different properties for the 'rolling' and 'slipping' directions of the tyres. The CarTerrain demo does this using the cross-product of the contact normal direction and the wheel hinge direction (although you might want to use a PerContactCB rather than PerPairCB).

Make sure you set the 'FastSpin' axis each time step for each of your wheels. This ensures that you don't 'lose' small components of rotation (eg. steering) when the wheels are spinning at high speeds.

Take care when developing your tire model. If the force values you calculate are from slip angles at low speeds you may find the angle flipping a lot. When translated into a force that may cause weird sideways swaying effects. As a suggestion you could use slip velocity at low speeds and slip angle at high speeds.

Collision

How should I represent my terrain collision model?

You should use trilists. There are a number of reasons for this. Having listened to our customers, trilist provides the flexibility and speed that is needed when integrating with various triangle storage systems. It was designed for collision between terrain and dynamic objects, and static and dynamic objects, where dynamic objects are represented by a simple geometry.

Trilist provides game developers with a lot of flexibility so that they can integrate it with their application more easily. They specify the triangle vertices / normals and any other information - such as triangle textures - through their application. This will enable them to supply information to the collision that their particular application knows about - for example, they may already have triangle vertex information stored for use with their choice of renderer.

Convex mesh - trilist and aggregate - trilist collision are supported in Karma.

While trimesh and heightfield are recognised collision representations, there are reasons not to use these. Karma no longer supports trimesh. Trimesh was developed with engineering solutions in mind, i.e. collision between high count polygon meshes of arbitrary topology. It is not really applicable to games where the collisions between two trimeshes is slow. If you need to do collisions between two complex shaped objects use an "aggregated" set of convex hulls. Heightfield was believed to be useful for terrain but is not flexible enough to deal with the ever-expanding number of methods of storing triangles for terrain.

What is the best way to use triangle lists?

Karma collision stores the extents of the bounding box around the terrain that can be changed dynamically if necessary.

Karma collision does the bounding box - object farfield collision test and if there is an intersection, calls the user defined callback and passes control to your application. It is then left to the developer to generate the (culled) list of triangles within the bounding box to pass back to Karma to perform the nearfield test on and generate contacts. You could pass all of your triangles back, but this would be excessive. In the ideal, the number of triangles in the list will be as low as possible. As an alternative method you could do all your triangle culling before the main call to Karma which may reduce the callback work for some applications.

The main problem you may come across is when apparently spurious contacts are generated, causing the colliding object to move in unexpected direction suddenly. This may be for a number of reasons. Firstly, edge contacts are notorious for creating strange contacts, mainly because if the surface is relatively smooth, edges are not expected. Tri-list contains no connectivity information so if you know that you have a smooth surface then it is best to turn off edge contact generation. Similarly, since there is no connectivity information, if triangles are passed to tri-list and they occupy the same triangle space, two identical contacts may be generated. This sometimes causes problems with Kea. If you suspect this is happening then it is best in the per pair callback routine to cull similar contacts (see MstBridge in documentation).

Remember that the order of the vertices has to correspond to the normal you give it according to a 'right-hand' system. So it looks like your 'top' triangle has the vertices in the wrong order. Also - just setting the contact position/penetration to fixed numbers in the callback might cause some odd behavior.

While using tri-lists, try to draw your generated contacts for debugging purposes, it will help a great deal if any of the above quirks occur.

How do I add an object into a collision space that interacts with only one other specified object?

Two ways to do this:

- Insert into the McdSpace farfield as usual, and just make lots of calls to McdSpaceDisablePair to disable interaction with each model in turn. Easy, but not very elegant.
- Don't insert the model into the McdSpace at all. Each time you call McdSpaceUpdate, update the model yourself by calling McdModelUpdate. Updating a model recalculates its position using any relative transforms, updates its AABB etc.

Now, look in MstBridge.c, function MstBridgeUpdateContacts, to see where you currently get pairs of potentially-colliding models out of the McdSpace and process them. You can change this to do all the pairs generated by the farfield, and then make a model pair yourself containing the model you didn't insert, and the one thing it can hit in the space, e.g. (please use this as a guide only):

```
MeBool pairOverflow;
McdSpacePairIterator spaceIter;
McdModelPair* pair;

/* end state-modification mode, ready for state queries */
McdSpaceEndChanges(s);

/* Initializeiterator for this space. */
McdSpacePairIteratorBegin(s, &spaceIter);

/* Keep getting pairs from farfield until we're done. */
do
{
     McdModelPairContainerReset(b->pairs);
     pairOverflow = McdSpaceGetPairs(s, &spaceIter, b->pairs);

     /* Initialises 'Hello' pairs and clears 'Goodbye' pairs. */
     MstHandleTransitions(b->pairs);

     /* Generate collision information and pass to dynamics. */
     MstHandleCollisions(b->pairs, s, w, b);
}
while(pairOverflow);

McdModelPairContainerReset(b->pairs);

/* Make a model pair containing my model (not in the space),
   and the one it can hit. */
pair = McdModelPairCreate(myModel, hitModel);

/* Put the pair into the container. */
b->pairs->helloFirst = b->pairs->helloFirst - 1;
b->pairs->array[helloFirst] = pair;

/* As before. This will 'hello' this model pair and do the test. */
McdHello(pair);
MstHandleCollisions(b->pairs, s, w, b);
McdGoodbye(pair);

/* Now clean up. */
McdModelPairDestroy(pair);

/* end of state-query mode, ready for state modifications. */
McdSpaceBeginChanges(s);
```

This could be improved further by doing a quick AABB test on the two models before creating a model pair for them.

```
MeBool overlap = 1;
MeVector3 myMin, myMax, hitMin, hitMax;

/* Get bounding box from each model. */
McdModelGetAABB(myModel, myMin, myMax);
McdModelGetAABB(hitModel, hitMin, hitMax);

/* Do test. */
if   (myMax[0] < hitMin[0] || myMin[0] > hitMax[0] ||
      myMax[1] < hitMin[1] || myMin[1] > hitMax[1] ||
      myMax[2] < hitMin[2] || myMin[2] > hitMax[2])
      overlap = 0;

if(overlap)
{
     /* Make model pair, hello, add to container, handle collisions, goodbye,
        destroy. */
}
```

It should also be easy to extend the many of your own models, but make sure the ModelPairContainer is big enough when you add them.

I'm using cylinders for lampposts in my game and there are many of them. Is it worth reusing an existing model or is it cheaper to release the old one and create a new one?

First of all, remember that many models can share the same geometry. So if you only have a couple of different sizes of lampposts, you can just create one McdCylinder for each size, and direct each model to the correct geometry. The RainbowChain and ManyPendulums examples show this.

As for re-using McdModels, this is probably a good idea. There is actually very little overhead in creating or destroying McdModels, because they are simply added or removed from a pool that is allocated at setup time. However, inserting and removing the model from the farfield McdSpace does have some overhead. It would probably be much quicker to simply change the position of an McdModel to represent a new lamppost object.

McdSpace works by keeping three axis-sorted lists of 'start' and 'end' markers for objects in the scene. Whenever an object moves, the farfield updates for any markers that are passed. When you remove or add an object to the farfield it has to first be moved to or from infinity, passing any markers on the way. Moving it a small distance inside the farfield passes fewer markers.

I'm confused as to where the McdModel transform is held.

Each McdModel has a pointer to a transform, rather than storing one internally itself. When you call McdModelSetBody it sets the McdModel to point to the transform held inside the MdtBody. If you call McdModelSetTransformPtr after this, you will change where the McdModel looks for its transform. If, for example, the transform you pass in to McdModelSetTransformPtr is only declared locally, it could cause a crash when it tries to access it inside collision. Also, because the McdModel no longer points to the dynamics transform, it won't move as the MdtBody does.

If you have tied an MdtBody and McdModel together, and you want to move it, use MdtBodySetPosition etc. instead.

Performance Considerations

I've prototyped my game with many objects in the world and that works fine. However, our level designers have now populated the world with lots and lots of objects, and things are starting to bog down a bit. What can you suggest we do to speed up the frame rate?

There are several things that can be done to speed simulations up. To start with, check your performance bar to see if it's a constant slowness, or just 'spiking' occasionally. Are there particular situations that cause it to slow down?

The first thing to know is that the basic static/dynamic box friction model can make large systems go more slowly. Try (as a test) turning off friction for objects against the world and see if it helps. When friction is turned on and bodies come into contact it will call the Linear Complementary Problem (LCP) solver. If a body comes in and out of contact it creates discontinuities in the forces causing an action in the solver called 'pivoting'. This also occurs when an object moves from static to dynamic friction.

Pivoting can cause slowness for large systems and is more pronounced on the PlayStation®2 version than the PC. One solution we've found is not to use box friction and use slip instead. This eliminates the change from static to dynamic friction. The way to totally turn off LCP is to also to set MaxAdhesiveForce to ME_INFINITY for every material, or in the contact parameters. This eliminates LCP pivots on colliding objects. The down side of this is that for one frame, the colliding objects will stay in contact with one another giving the impression of stickiness. However, when used in conjunction with slip it allows the body to move.

Using no friction and increasing damping works, but isn't great.

Large systems occur either when there is a object containing many joints or situations where many objects touch each other i.e. a wall with lots of bricks in. This forms a large matrix which needs solving.

For the objects with many joints you could use level of detail (LOD) physics modelling similar to LOD in graphics. If you can't see something or you are a long way from it, you can degrade the friction model/cull more contacts etc. The GreaseMonkey demonstrates this by having a mode where the car is made up of five bodies (one chassis, four wheels) and another mode where there is one body and four contacts for the wheels. Think about how your physical models could be reduced in this way. It's worth remembering that however fast Karma is, and will be, it will always take up a finite amount of time and avoiding unnecessary calculations is usually a good thing to do.

Could you provide a table of numbers, which indicates the relative speed of collision detection and contact generation between different types of primitive and non-primitive geometries? This would be very a helpful guide in making decisions about what kind of collision geometries to use.

In principal there's no reason why not, and we'd be happy to publish this information if we had it, but the relative speed of geometries varies by platform and by the extent of completion of collision optimisation. In general the order (fastest first) is spheres and planes, boxes, cylinders, convex objects, triangle meshes. The speed of trilist is dependent on too many factors outside Karma to be predictable independent of the application.

What causes the constraint solver to take a variable amount of time when solving a set of constraint equations?

Every time Kea takes a step, the constraint solver must go through a number of iterations to find a good solution for the forces on the rigid bodies. This is the only part of Kea that takes a variable amount of time. If it were not for the constraint solver, then each step would take the same amount of time for the same system.

Sometimes the constraint solver will fail to find a solution, and the warning message “cycle has been detected” or “The maximum number of iterations has been exceeded” will be output. Even with a warning from the solver the behavior will usually be stable enough. If there are any visible problems, epsilon may need to be increased.

Sometimes the constraint solver takes a large amount of time when many objects make or break simultaneous contact, or when the contacts involve very large collision forces. Because of the large number of physical interactions taking place at a particular time, the amount of time required to do the physics can increase. There are two ways in the current version of Karma that will help here:

- Design the simulation carefully to try and avoid this situation
- Limit the constraint matrix size, as discussed below.

Is there any way of reducing the constraint matrix size to speed up my simulation?

Each time step, Mdt groups all bodies into 'partitions' that are constrained together either by contacts or joints. A collection of bodies joined together (e.g. a rag-doll) will always be in the same partition. If the rag-doll hits a box, the doll and the box will be in the same partition while in contact. Contacts to the world do not join partitions together e.g. two boxes sitting separately on a static plane will be in different partitions.

When Kea solves for a group of bodies, it constructs a matrix representing the way the constraints limit the freedom of movement of the bodies in the partition. This matrix has one row for every degree of freedom limited by the constraints. The number of constraint rows in a partition is a factor in the time taken to solve that partition, and the amount of memory that Kea uses to build and solve the matrix for each partition is related to the square of the number of rows. If you have a large number of bodies connected by joints and contacts, the matrix size (and the consequent memory requirements) may be inconveniently large.

By setting a limit on the matrix size, you can instruct Karma to remove constraint rows from the simulation. This allows an application to degrade simulation fidelity to meet memory constraints. Deleting constraints may also decrease Kea’s running time significantly. The maximum matrix size for Karma to attempt to reduce a partition to, is set using

```
void MdtWorldSetMaxMatrixSize(const MdtWorld w, const int size)
```

To try to meet this limit, Karma infers which constraint rows can be removed with least loss of fidelity. By default, first rows which enforce friction constraints are removed, then contacts between pairs of bodies, then contacts between bodies and the world. Contacts are removed in order of depth, from the shallowest penetration to the deepest. Joints are never deleted, and at least one contact is always left in each contact group. While a more aggressive constraint reduction strategy is possible, large constraint violations are more likely to be avoided by an application-specific strategy based on the properties of a particular simulation.

MdtWorldSetContactImportanceCB can be used to define a callback that replaces the above method for ordering contacts, with a user-defined order of importance.

The MdtKea library contains a function MdtKeaMemoryRequired, that returns the amount of memory required to solve a particular set of partitions. The MeChunk utility in the MeGlobals library holds a pool of memory that is automatically resized as necessary and used by MdtKea each time step.

Implementation Considerations

Are the force and torque accumulators MeVector4s?

Yes they are, inside the MdtKeaBody struct, held inside the MdtBody struct. The MdtKea structures are designed to ensure quad-word alignment for data where needed. This is needed by PS2/SSE etc. for SIMD speed-ups. The final element of each is just treated as 'padding'.

Why are the impulse accumulators MeVector4s instead of MeVector3s?

Just for symmetry with the force/torque accumulators in the MdtKeaBody struct. The last elements should be kept as zero for safety though.

My simulation is deterministic if I stop it and re-execute it. However, if I simply restart it without resetting, it isn’t - how can I make my simulation deterministic in this case?

Because Karma simulations use deterministic Newtonian dynamics to model the virtual world it is reasonable to expect that a simulation with identical starting conditions, run on a machine with a given configuration, would evolve in the same way provided there is no external interactivity with the system. Indeed, this is the case if a simulation is stopped and re-executed. However, if a Karma simulation is simply restarted, the simulation will almost certainly not proceed along an identical path each time. This is because the order in which information is passed to the constraint solver, and hence the order of the data on which calculations are performed, will very likely change when a dynamic object or collision model is removed from the simulation and then added again when it restarts. This is because Karma optimizes the arrangement of objects in memory which permutes the order in which they are tied to the solver.

To get your simulation to follow the same path, the order in which data is sent to the solver must be identical. Sort Keys are used to accomplish this. This allows Karma to deterministically order the input to the constraint solver. Sort keys must be positive and are in the range 0 to 2**15 - 1 for collision objects and 0 to 2**31 - 1 for dynamics objects. Similarly, joint keys run from 0 to 2**31 - 1.

Which exceptions might Karma generate on x86 platforms?

In normal operation Karma may generate denormalized operand, numeric underflow, or loss of precision exceptions. Karma should never produce overflow, divide-by-zero, and invalid operation exceptions.

Is Karma thread-safe?

No. Karma implements no protection against concurrent access to data.

On x86 platforms, Karma is fully reentrant: independent instances of Karma can run safely in different threads, where "independent" means the MdtWorld, MstBridge, and McdFramework are different share no writable data structures, such as McdGeometries or compound transformation matrices for McdModels.

On PlayStation®2 , you cannot run the Karma pipeline concurrently in multiple threads, since many of the algorithms assume control of the scratchpad and/or VU0.

Performance

Frame time used by Karma

Each frame the proportion of the processing time that is taken by Karma depends on the number of objects in collision and the number of joints used. It is desirable for dynamics to use as small a proportion of the frame time as possible. For this reason, considerable effort has been put into optimizing Karma to make efficient use of both the PlayStation®2 and x86 platforms. However, as the time spent by Karma depends on the simulation complexity, it is important that the game programmer limit the number of contacts and joints to reduce the proportion of the frame time used.

The amount of time taken depends on the total number o*f constrained degrees of freedom of the system*. Consider a pair of objects. Their relative position is specified by three values, and relative orientation by a further three values. These six values are the *six degrees of freedom* of the pair of objects. A constraint, such as contact or a joint, limits motion in a certain number of the degrees of freedom. For example, a *frictionless contact* just prevents penetration of the objects at a given point, and so constrains one degree of freedom. A *friction contact* prevents penetration and also applies friction to the remaining two linear degrees of freedom, and so constrains three degrees of freedom. A *ball and socket joint* ensures that the constrained pairs of objects can rotate relative to each other but not move relative to each other, thus constraining three degrees of freedom.

Example 1 – A car with physically modelled wheel rotation, steering and suspension.

A car can be physically modelled as five bodies - a chassis and four wheels. When the car is in mid air, there are four constraints. These constraints join the wheels to the car, and apply forces due to steering and suspension. They each constrain all six degrees of freedom, so the total number of degrees of freedom of the system is twenty four (4x6).

When the car is on the ground, there are four additional constraints, one friction contact for each wheel. These prevent the car from falling through the ground and also apply friction forces to the wheels. They each constrain three degrees of freedom, bringing the total number of degrees of freedom constrained to thirty two. (4x5 + 4x3).

Example 2 – A simplified car

With the above model, the user applies a torque to the wheels and MathEngine calculates the friction force applied by the ground and the resulting linear force that moves the car forwards. This is quite nice because the linear motion of the car is just something that happens as a consequence of applying torque to the wheels, just like in real life. However, the gamer doesn’t really care how the linear motion happens, they just care about the physical and graphical effect of steering and suspension. With this in mind, a much simpler, cheaper model can be used.

In this model, the car is one physical body. Whilst in the air, there are no constraints, so the car is free to fall under gravity. The number of degrees of freedom constrained is zero.

When on the ground, the wheels are represented by four friction contacts. The softness parameter of the contact is set to provide suspension and steering is achieved by setting a greater maximum friction in the direction you want to steer. Four friction contacts constrain a total of twelve degrees of freedom.

Example 3 – Thrownning a box at a wall

Suppose you are simulating a bar-room brawl. You may want to allow characters to throw boxes, tables, chairs etc. at each other. The simplest case is a box in the air. Like the car in the air example, the number of degrees of freedom constrained is zero.

You will want the box to collide with the walls. In the worst case, three contacts will be required to stop the box going through the wall. Suppose you used three friction contacts. The number of degrees of freedom constrained will be nine (3x3).

However, it is not necessary to model the friction of the wall unless you want to rest the box against the wall, so you could use three frictionless contacts, reducing the number of degrees of freedom constrained to three (1x3).

Example 4 – A human falling down stairs

Most of the time, the best way to move humans in games is to use motion capture animation. However, if for example you want a character to fall backwards over a table or down a staircase, then the amount of motion capture required can become impractical. Physical modelling can be more cost effective in these cases. Take the example of a human falling down a staircase. A simple model of a human could consist of ten bodies - a head, a torso, two forearms, two upper arms, two thighs and two calves. Nine limited ball and socket joints are used to hold the ten bodies together and provide joint limits and muscle modelling. This constrains twenty seven (3x9) degrees of freedom. In the most extreme case, twenty seven contacts are required to prevent the human falling through the staircase. The total number of degrees of freedom constrained is one hundred and eight (3x9 + 3x27).

Dynamics Algorithms - Technical Information

An overview of the rigid body dynamics simulation algorithms currently being employed in game physics follows. These algorithms are conveniently categorised as Mirtich-style methods, penalty methods and Linear Complementarity Problem based methods.

Mirtich's method

In Mirtich's method it is assumed that only one pair of objects can be in contact at any given time. When a contact occurs, an impulse is calculated to prevent penetration. Time is then advanced until the next collision occurs. Resting contact is modelled by micro impulses. The advantage of this method is that it is not necessary to solve a big matrix equation, because each collision is considered in isolation. The disadvantage is that it is very hard to make the algorithm cope with arbitrary stacks and piles of objects and arbitrary external forces.

Penalty Methods

Penalty methods prevent penetration by modelling the contact points between objects as stiff springs. In general, a semi-implicit or implicit integrator is required to achieve stability. Such an integrator requires a matrix equation to be solved. With penalty methods, it is difficult to simulate stable stacks and piles of objects.

Consider an inequality constraint where objects can come into contact but are not, as with a joint, bound. A (stiff) spring can be used to model the contact. If the objects come into contact then the spring acts to separate them. The spring is called stiff because the force it produces varies rapidly with respect to changes in displacement. When objects do not penetrate there is no constraint and the spring does not exist. A 'cheat' to allow 'bounciness' between contacting objects is to reduce the spring stiffness. The object's penetration increases i.e. the constraint is violated.

When modelling collisions using penalty methods, a good collision detection system is required. Object penetration can result in large forces being applied to separate objects. Hence there can be inaccuracies here - for example, the simple case of a ball being dropped from different heights onto a plane. The height that it bounces to may not be predicted exactly. As far as games are concerned this is usually not a problem - the solution is accurate enough. In robotic research the ability to violate a constraint can be advantageous. The software is very stable and robust which is useful for a number of applications.

Linear Complementarity Problem (LCP) based methods

The most common Linear Complementarity Problem based method used in graphics was developed by David Baraff of Pixar. In Baraff's method, different techniques are used for colliding contact and resting contact. For colliding contact, a matrix equation is solved that provides the impulse required to simultaneously repel colliding objects. For resting contact, a matrix problem called a Linear Complementarity Problem, or LCP, is solved. These methods naturally model articulated bodies and friction. In order to solve the LCP Baraff suggests using a Dantzig Solver

Baraff's method has a number of speed and stability problems, and since Baraff first published his methods, researchers have been working to overcome them.

Firstly, there are a number of physical situations where Newton's laws with Coulomb friction are inconsistent, whatever type of solver is used. The most famous of these situations is Painleve's problem. Unfortunately, Painleve's problem is a very simple situation involving a rotating bar and a plane. This means that solvers that don't cope with Painleve's problem are likely to fail even in the simplest of scenes. Baraff uses an acceleration based constraint solver which fails on Painleve's problem - the way Coulomb friction is modelled by Baraff leads to large LCPs that do not necessarily have a solution.

Solving for impacts and resting contact as separate stages increases frame time and produces complicated code.

Baraff’s method is an explicit method. This means that accumulated numerical error can potentially cause the simulation to explode. To overcome this, Baraff adds damping which has to be tuned by the user to achieve stability. Ideally, a simulator should work in all situations without the need for user tuning. Often, Baraff’s method is used with higher order integrators such as Runge Kutta to achieve stability. This method requires 4 times as many force calculations per frame as 1st order methods such as implicit Euler.

Baraff doesn’t cope well with degenerate constraints which naturally arise when dynamics is used together with collision detection. Degenerate constraints lead to LCPs that are not symmetric positive definite. Such LCPs are not guaranteed to have a solution.

Other LCP based methods

Recent research has produced LCP based methods that are somewhat confusingly named *time-stepping methods.* These methods formulate the contacts and constraints in terms of force and velocity rather than force and acceleration. The solver then calculates the force to apply over the timestep to simultaneously satisfy all the velocity constraints. One advantage of such methods is that both impact and resting contact are calculated by the same simple algorithm. Another is that such methods provide solutions for Painleve type situations. Researchers have proved stability properties of time-stepping methods which are not enjoyed by Baraff’s method.

Karma x86 and SSE Optimizations

Aside from algorithmic optimizations, considerable effort has been expended in optimizing the code to run well on x86 and SSE (Streaming SIMD Extensions) platforms.

The collision detection and high-level Mdt algorithms operate on small vectors, and they are written to use a vector maths library.

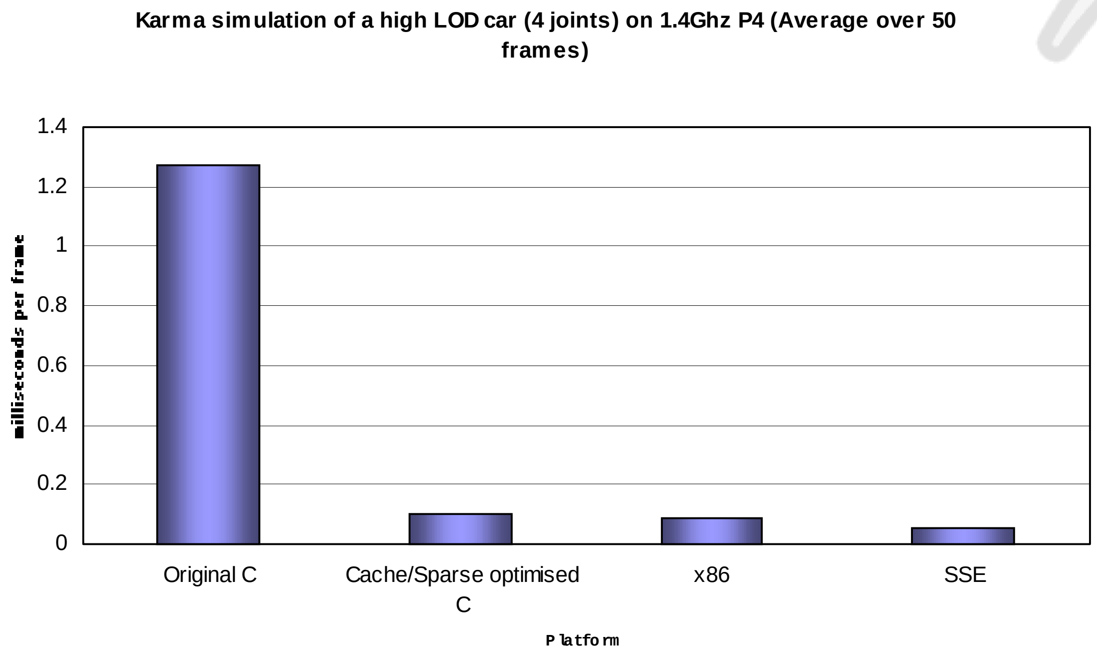
The dynamics algorithms perform much more exotic matrix operations than the collision detection algorithms. For this reason, these algorithms have been rewritten in hand coded assembly language. Table 1 shows the optimization status of each algorithm in the Karma pipeline.

The dynamics matrix is typically very sparse i.e. most of the element entries are zero. All the algorithms that manipulate it, such as the Cholesky factoriser, solve and matrix-vector multiplier have been optimized to take advantage of its sparsity. The J matrix is also very sparse, and its sparsity is well defined. All operations that access J access small submatrices of J. In a naive implementation, the submatrices are not necessarily aligned on 4 float boundaries, so the fast aligned load instructions of SSE cannot be used. For this reason, Kea uses a highly novel storage format for J which ensures that the submatrices required are stored consecutively in memory, to increase cache efficiency whilst keeping them aligned to 4 float boundaries.

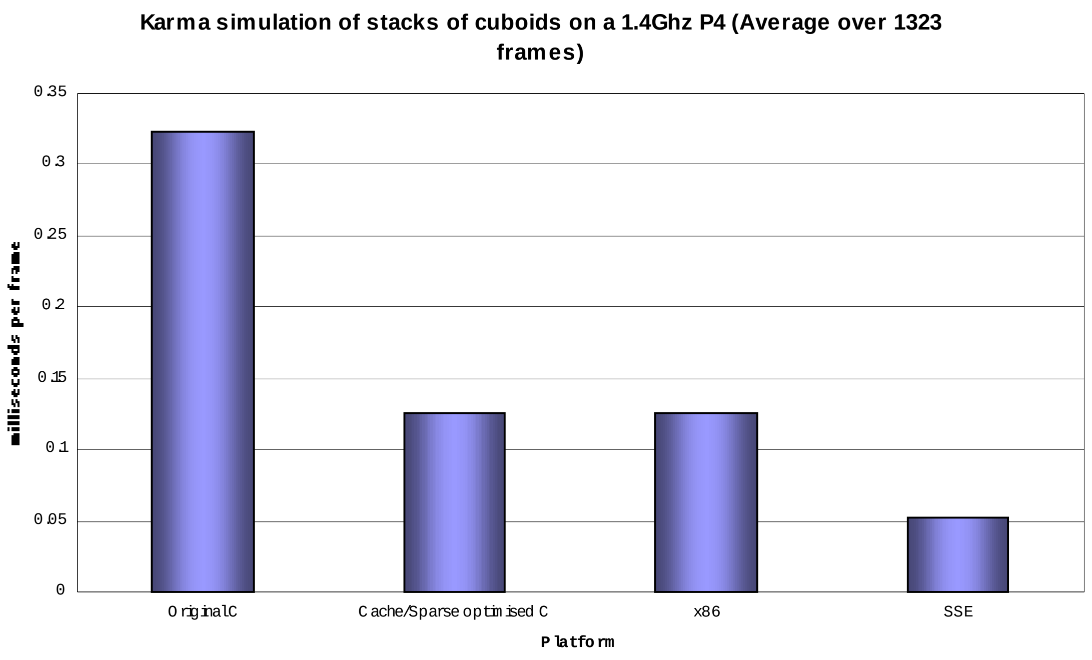
All algorithms that have been hand coded in assembly language have been implemented twice, once for x86 (Pentium, PentiumII and AMD) and again for SSE/SSE2 (PentiumIII and Pentium4). The vector instruction set of SSE provides impressive speed gains.

Section	**Math Library**	**Cache optimized**	**Handcoded X86**	**Handcoded SSE**
Farfield (AABB tests)	●			
Nearfield (Intersection tests)	●			
Mdt (partition and freeze)	●			
Bcl (make constraint matrix)	●			
Calc J*M^{-1} and rhs		●	●	●
Calc A=JM^{-1}*J^{T}		●	●	●
Factorise A		●	●	●
Solve A		●	●	●
Factor Q		●	●	●
Solve Q		●	●	●
Calc constraint forces		●	●	●
Update position and velocity		●		

Benchmarks



This figure shows the performance of the original C code, the C code optimized for cache and sparsity, the x86 code and the SSE code when simulating a high level of detail car consisting of 5 bodies, 4 joints and 4 contacts. The original C code took 1.3ms per frame, the SSE code took 0.055ms per frame, a twenty-three fold speed increase



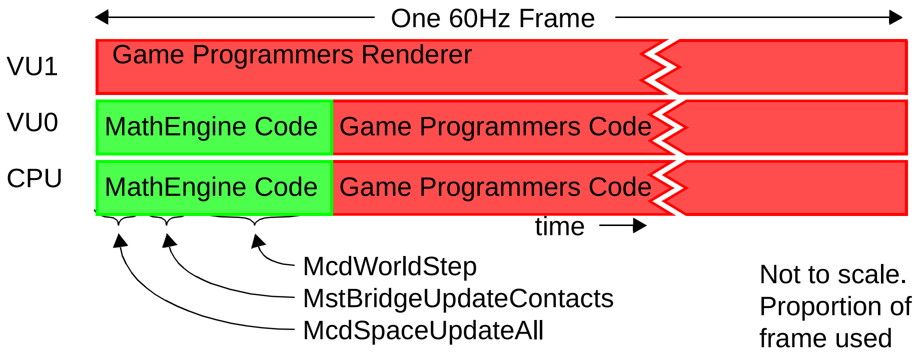
This figure shows the time taken by each platform to simulate the ‘Topple’ example. Topple consists of 6 stacks of between 3 and 6 cuboids. The average simulation time of the original C code was 0.32ms. The simulation time of the SSE code was 0.054ms, a six-fold increase.

Karma for PlayStation®2

Processor usage

Each of the three main MathEngine functions (McdSpaceUpdateAll, MstBridgeUpdateContacts and MdtWorldStep) use the CPU and VU0. These functions don't spawn threads, patch interrupts, or otherwise take any CPU or VU0 time outside of their scope. The functions will kick off VIF0 chains, but the functions will not return until the last VIF chain has finished. VU1 is not used by the MathEngine toolkits.

It is common, (though not necessary) to call the 3 MathEngine functions at the start of each frame. Once MdtWorldStep has returned, the game programmer is free to use the cpu and VU0 for rest of the frame.



Code size

The table in the 'Object Code and Static Data Section' in 'Appendix C Memory Allocation in Karma' shows the size of each Karma library on the PlayStation®2.

Collision

The MathEngine Collision Detection library consists of seven libraries, namely McdCommon.lib, McdConvex.lib, McdConvexCreateHull.lib, McdFrame.lib, McdPrimitives.lib, McdRGHeightField.lib and McdTriangleMesh.lib. The main library is libMcdFrame.a, which provides the collision framework. The other six libraries provide specific types of collision detection and are optional. They can be used alone or in combination. LibMcdPrimitives.a detects collision between primitive objects such as cuboids, cylinders, spheres etc. These tests are very fast and good results can often be achieved by approximating complex objects by a number of primitives for the purposes of collision. If this does not give good enough results then LibMcdConvex can be used. This detects collision between convex objects. This library can be used with non-convex objects by splitting up the object into convex pieces. Although this library can be used with a wider variety of objects, it can be slower than the Primitives library. The most difficult collision detection problem is detecting collisions between two arbitrary triangle meshes. This type of collision is used when it is not possible to approximate a mesh by a union of primitives or convex pieces. This type of collision detection is very slow, and we do not recommend that it is used in PlayStation®2 games.

The libraries have minimal interdependency and many are optional.

Scratchpad, VUMEM0, MICROMEM0 and VU0 registers.

These three MathEngine functions make extensive use of the scratchpad, MICROMEM0 and VUMEM0. The three MathEngine functions do not preserve the state of the scratchpad, MICROMEM0, VUMEM0 or VU0 registers.

Vif0 chains

As MathEngine makes extensive use of VIF0 and VU0 within the three high level functions, the game programmer should not attempt to run a VIF0 dma chain in parallel with the functions. The dynamics core, kea disables VIF0 interrupts on entry due to a conflict between VIF interrupts and the bc0f instruction.

Compatibility with compilers

The example code and toolkit source can be compiled with the SCE provided ee-gcc (version 2.9.5 or later) or the SN Systems' Pro-DG compiler. A Metrowerks Codewarrior build environment is also supplied`.

Compatibility with renderers

Karma is supplied with a straightforward renderer, MeViewer2. This is an unoptimized renderer and its main use is for viewing the MathEngine examples and tutorials.

Karma can be used with middleware renderers and has been tested with Renderware by Criterion Software and Alchemy by Intrinsic Graphics.

Optimizations

Karma was initially developed on the SGI and PC in C. Before optimization, Karma was slower on the PlayStation®2 than on a 300MHz PC. Optimization has substantially increased the speed of Karma. This section describes how the code has been optimized.

Vectorisation

All components of Karma are rich in matrix and vector operations. The dynamics core, kea contains little else.

VU0

As the majority of the instructions executed by the core components are floating point multiply accumulate instructions, a good measure of the optimality of the code is the number of floating point operations executed per second. This is measured in GFlops, where 1 Gflop is 10^9 floating point operations per second. A floating point operation is either 1 multiply or 1 add. By this definition, the peak performance of the cpu is 0.6GFlops, VU0 2.4GFlops, and VU1, 3.0GFlops. This gives a total of 6.0GFlops for the emotion engine.

Vectors are not first class types in C, so there is no way for a C compiler to compile a vector expression into a VU assembly program. This meant that when the unoptimized C code was run on the PlayStation®2 only the cpu was used, so the code could only utilise 0.6 GFlops of the total 6.0 GFlops available.

It was clear that the code would need re-writing to use VU0. This would take the total number of GFlops available to 0.6 + 2.4 = 3.0.

Macrocode or Microcode

VU0 can be used in 1 of 2 modes, macromode or micromode. Macromode is used by some components of the toolkit, and micromode by others. When using macrocode, you have 16k of fast memory available (the scratchpad), when using micromode you only have 4k of fast memory available (the VUMEM0). This was the main factor used when deciding whether to use macromode or microcode for a particular algorithm.

Scratchpad double buffering

The PlayStation®2 accesses main memory through a Level 1 cache. As the main memory is Rambus® memory, cache line loads are very slow (~40 cycles). Our original C code, like most C code developed on the PC, often missed the cache.

On the PlayStation®2 there is an alternative to accessing the main memory via the cache, that alternative is scratchpad double buffering. The scratchpad has a one cycle access time and can be accessed concurrently by the dma controller and the cpu. In the simplest case, scratchpad double buffering is used to accelerate algorithms whose input and output are lists of the same length. The scratchpad is divided into two 8k buffers. At every iteration of the algorithm, one buffer is being processed, and concurrently the other buffer is being written to the output list and then refilled from the input list. At the end of the iteration, the buffers are swapped.

Once a scratchpad double buffer algorithm is correctly tuned, the processor never stalls for data. In these cases the PlayStation®2 can often outrun a pc running at several times the clock rate! The biggest hotspot of the toolkit was kea, the dynamics core. Kea consists of around six sparse linear algebra algorithms run in sequence.

Scratchpad double buffering can only be used if the algorithm is able to operate on one chunk of data (typically 8k long) at a time, without accessing any other chunks. Originally, the sparse linear algebra algorithms we used did not have this property. A sparse matrix format has been developed that can be scratchpad double buffered which is just as efficient as the original.

Chunkable data is also required when running microcode, because the vector unit can only access the 4k of data in that is in the VUMEM at any given time. The new sparse format allows each algorithm to be implemented in either scratchpad double buffered macrocode or microcode.

Summary

- The collision detection toolkit uses macrocode to perform vector operations on geometry data.
- The dynamics core, kea, uses microcode to perform its most time intensive algorithms and scratchpad double buffered macrocode for its other significant algorithms.
- No significant algorithm in kea accesses main memory. Almost all memory access happens via dma, in parallel to processing.

Performance of a ‘Rag Doll’ Simulation on the Sony PlayStation®2 using MathEngine Karma

The PlayStation®2 has a complex but potentially powerful architecture. For this reason, the amount of optimization required to get good performance is often greater than is required on the PC. Karma has been optimized to take advantage of powerful features of the PlayStation®2 such as VU0 microcode and to overcome pitfalls such as Rambus® latency. This optimization work has been ongoing for the past 2 years, and is projected to continue.

Despite the success of the optimizations performed so far, care is required when setting up simulations to get the best performance out of the PlayStation®2. This document describes the current state of optimization, some of the planned future optimizations and also tips for tuning simulations to run well on PlayStation®2.

In this document, we will analyse the ‘Ballman’ example. This example consists of a ‘rag doll’, a staircase and a stack of boxes. The first thing to note is that the simulation time does not depend on the number or type of objects present, but on the interactions between objects. In Ballman for example an interaction between the rag doll and the stack of boxes is more expensive than the interaction between the rag doll and the floor. Not only do different types of interaction take different amounts of time, but they also may stress different parts of the Karma pipeline, i.e. different interactions cause different parts of the code to be the ‘hot spot’. For this reason, optimization is best performed in the context of a specific application, rather than trying to optimize for the average case.

Application level Optimizations

Apart from the ongoing low-level optimization of the Karma code, there are a number of application-level techniques for improving a simulation’s performance on PlayStation®2. The types of interactions in Ballman cause a part of Karma called the Linear Complementarity, or *LCP* solver, to become the bottleneck. The LCP solver is responsible for calculating the forces to apply to a set of objects in order that certain constraints on their relative velocities are met. In particular, the LCP solver calculates friction forces and non-penetration forces.

The LCP is iterative in that it makes an initial guess of the forces, then if the velocity constraints are not met immediately, it refines the guess until they are. Unfortunately, the code that makes the refinements is currently limited by the DMA bandwidth from the CPU to VIF0. For this reason, interactions that generate many LCP iterations run much slower on PlayStation®2 than on PC, even when clock speed is taken into account. For now, the best way to make simulations run well on the PlayStation®2 is to minimise the number of LCP iterations.

There are two main ways of reducing the number of LCP iterations. The first is to use viscous friction instead of static/dynamic friction. This is done by setting the maximum friction force to infinity. The amount of dynamic friction is set using the slip parameter.

```
/* Sets the friction type between material1 and material 2 to viscous */
/* Amount of friction is determined by VISCOCITY */
    #define VISCOCITY 0.3f
    p = MstBridgeGetContactParams(bridge, material1, material2);
    MdtContactParamsSetType    (p, MdtContactTypeFriction2D);
    MdtContactParamsSetFriction(p, MEINFINITY);
    MdtContactParamsSetSlip    (p, VISCOCITY);
```

The second optimization technique is to make surfaces adhesive. This is done by setting the max adhesive force parameter of the contact params to a large number. For maximum speed, the max adhesive force should be set to infinity.

```
/* Allow contacts between material1 and material2 to become adhesive during the
frame */
    p = MstBridgeGetContactParams(bridge, material1, material2);
    MdtContactParamsSetMaxAdhesiveForce(p, MEINFINITY);
```

In simulations without joint limits, implementing these two optimizations for all possible pairs of materials ensures that the LCP solver will always guess correctly the first time, and hence never execute the slow iteration code. The disadvantage is that viscous friction doesn’t look as good as proper static/dynamic friction and that adhesion causes surfaces to be slightly sticky. However, somewhere between applying these optimizations for all material pairs and no material pairs there often lies a suitable compromise between behavior, quality and speed.

The Ballman demo has been modified to demonstrate these optimizations. To toggle between high cost and low cost material interactions, press select on the PlayStation®2 controller to display the options menu, move the cursor to the 'High Quality Friction model' and press cross to toggle. By holding down square button, the rag doll can be propelled towards the stack of boxes. Despite the adhesion, the boxes still collapse, and the peak dynamics time is reduced from over 16ms, to around 3ms.

Karma architecture

Karma can be thought of as a pipeline of stages. The stages are executed in sequence, and the output of each stage is the input of the next.

The first stage is collision detection. This takes the current positions of the bodies being simulated and uses knowledge of their geometry to identify points of contact between them. Mdt partitions the contact list into groups of non-interacting objects. It then removes partitions that are not moving. It passes the partitioned list of constraints to Bcl, that builds the Jacobian constraint matrix. The next few stages construct and solve an LCP that gives the forces required to satisfy the constraints. An integrator takes this force and updates the velocities and positions of the bodies, outputting their updated transformation matrices.

A number of sections of the pipeline have two implementations, a sparse implementation and a dense implementation.

Section	**Macrocoded**	**Microcoded**	**SPR / VUMEM buffered**
Collision detection			
Mdt (partition and freeze)			
Bcl (make constraint matrix)			
Calc vhmf	●		
Calc jlen			
Calc J*M and rhs	●		●
Sparse Calc J*M*J^T	●		●
Sparse Factorise A		●	●
Sparse Solve LCP		●	●
Dense Calc J*M*J^T	●		●
Dense Factorise A		●	●
Dense Solve LCP		●	●
Calc constraint forces	●		●
Integrate	●		
Mdt unpack			

Table 1 – The optimizations that have been implemented in each section

Current state of PlayStation®2 Optimization

The PlayStation®2 has a large potential floating-point performance due to the presence of its two vector units. The PlayStation®2 also makes use of Rambus® memory which has high burst speed but poor random access speed. This means that contiguous DMA transfers are very fast, but Icache and Dcache misses are very slow. The PlayStation®2 optimizations implemented so far fall into two categories, rewriting code to take advantage of VU0 in either macro or micro mode, and rewriting code to replace cache use with DMA.

Planned Optimizations

As mentioned earlier, a common bottleneck in the Karma pipeline is often the iterative part of the sparse LCP solver, which is implemented in VIF0 sequenced microcode. Initial timings showed that the microcode execution time is a very small proportion of the LCP execution time. Experiments with the Sony Performance Analyser hardware have confirmed that the bottleneck is the speed of the DMA transfer between the main memory and VIF0. Further experiments showed that the bottleneck is the Rambus® memory interface, not the DMA controller, so that even uncached accelerated loads would not be fast enough. In light of this, we are currently implementing an algorithm that requires less data transfer.

The collision detection part of the pipeline is currently bottlenecked by Icache misses. This is due to a complex control flow, which is currently being replaced by a simple pipeline of functions.

Performance analyser results

Figure 2 shows the output of the Sony Performance Analyser for a frame of Ballman with all the previously described application level optimizations turned on. The Ballman example uses SCE’s PS2GL renderer.

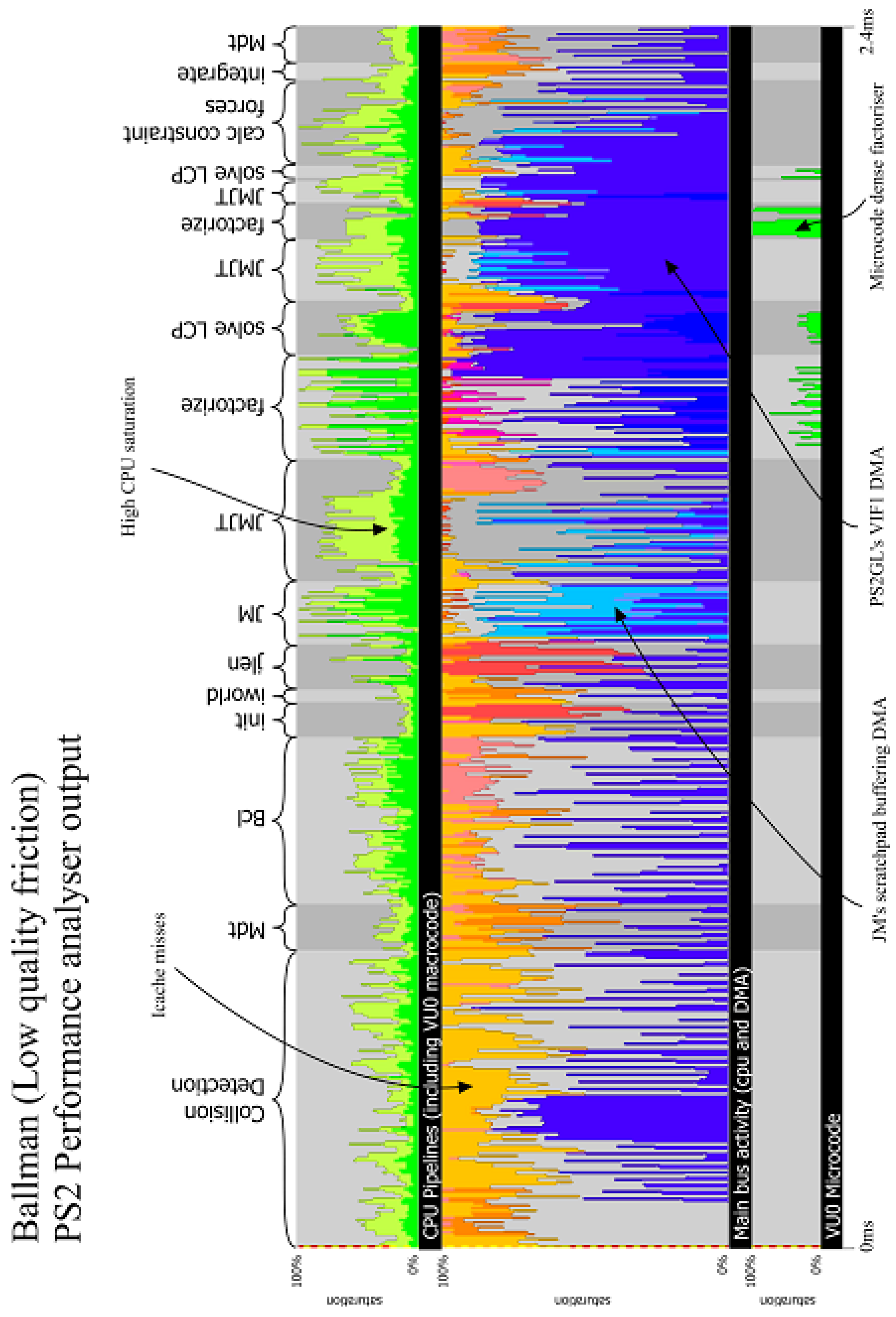
The graph consists of 3 horizontal slices representing CPU usage, bus activity and VU0 microcode usage. The vertical axes represent saturation and the horizontal axis represents time, hence the area underneath the CPU graph represents the amount of useful work done, and the area above represents the amount of time wasted due to stalls. The amount of useful work done remains the same after optimization, but the saturation increases, reducing the execution time.

CPU bus activity is drawn at the top of the graph as yellow pink and red bars, and DMA initiated bus activity is drawn at the bottom of the graph in 3 shades of blue. CPU bus activity is caused by Icache misses, Dcache misses and uncached loads and stores. Notice that peaks in the CPU bus activity graph correspond to troughs in the CPU saturation graph. This is particularly evident in the collision detection section.

The main cause of DMA bus activity is the PS2GL renderer which DMAs to VU1, this is the mid-blue bar. Sections of the pipeline that have been rewritten to buffer data through the SPR/VUMEM such as J*M and $J*M^{-1}*J^T$ use DMA to load their data, instead of the Dcache refill mechanism. This DMA can be seen as a light blue bar. The advantage of this is that unlike Dcache misses, DMA does not stall the processor. Note that sections employing this optimization have very few Dcache misses and very high CPU saturation.

The remaining part of the graph shows VU0 microcode usage. Unfortunately, the performance analyser cannot distinguish between CPU instructions and VU0 macrocode instructions. So, although the VU0 microcode graph looks quite sparse, bear in mind that much of the pipeline uses VU0 macrocode. Note that the dense factoriser achieves 100% VU0 saturation for the majority of its runtime.

.


Ballman (Low quality friction)
PS2 Performance analyser output
Collision Detection
Mdt
Bcl
init
iworld
jlen
JM
JMJT
factorize
solve LCP
JMJT
factorize
JMJT
solve LCP
calc constraint forces
integrate
Mdt
Icache misses
High CPU saturation
CPU Pipelines (including VU0 macrocode)
Main bus activity (cpu and DMA)
VU0 Microcode
JM's scratchpad buffering DMA
PS2GL's VIF1 DMA
Microcode dense factoriser
0ms
2.4ms
100%
0%
saturation

Quadbike Tutorial

Introduction

The aim of this exercise is to work progressively through the key elements of a MathEngine Karma application involving a character riding a quadbike over rough terrain. Most of the application framework is already in place including the rendering aspects. To build up the application, paste the code sections described in this document into the application at the appropriate points. If there is any doubt, search for the identifying PASTE_## comment string. At each step the application can be built and executed to observe the effect of each component.

The basic sections are:

1. Initializing the Karma collision and dynamics framework.
2. Adding the update functions to the main loop.
3. Creating the terrain.
4. Creating a four wheeled vehicle.
5. Adding user control to the vehicle.
6. Creating a rider for the vehicle.

Karma does not have an inherent coordinate system or use any specific units. Some of the utility functions take angles as parameters. These are assumed to be in radians. The main requirement is that a consistent convention is chosen and used throughout. Quadbike uses a right-handed coordinate system with the y axis as up. SI meters and kilograms are chosen as the respective length and mass base units.

Initializing the Karma Collision and Dynamics Framework

This is carried out once in any application, generally at startup. It ensures that any memory allocation required for creating pools of bodies, collision models etc takes place before the main loop.

File: main.c

Function: `InitialiseMEWorld()`

Identifier: PASTE_01

```
/* Set the default Universe Pool sizes */
sizes = MstUniverseDefaultSizes;

/* Initialize Universe Pool sizes */
sizes.dynamicBodiesMaxCount = 20;
sizes.dynamicConstraintsMaxCount = 1000;
sizes.materialsMaxCount = 10;
sizes.collisionModelsMaxCount = 300;
sizes.collisionPairsMaxCount = 1000;
sizes.collisionUserGeometryTypesMaxCount = 0;
sizes.collisionGeometryInstancesMaxCount = 0;

/* Create a basic Karma Environment */
universe = MstUniverseCreate(&sizes);
if(!universe)
     return 0;

world = MstUniverseGetWorld(universe);
space = MstUniverseGetSpace(universe);
bridge = MstUniverseGetBridge(universe);
framework = MstUniverseGetFramework(universe);

/* Initialize some fundamental world properties */
MdtWorldSetEpsilon(world, 0.0001f);
MdtWorldSetGamma(world, timeStep*10);
MdtWorldSetGravity(world, 0, (MeReal)-20, 0);
```

Since this is just a standard application, `MstUniverseCreate` is the easiest way to create and initialize the Karma components. You could do this manually using `MdtWorldCreate`, `McdInit` etc if multiple worlds or collision spaces were required. The universe is queried for handles to the world, space, framework and bridge since these will often be required during the creation of bodies and collision models. The world parameters (epsilon, gamma and gravity) are set at this stage. In this application they remain unchanged but they could be modified at any time. Gravity is as an acceleration vector acting down the world y axis.

File: main.c

Function: `InitialiseMaterials()`

Identifier: PASTE_02

```
groundMat = MstBridgeGetDefaultMaterial();
wheelMat  = MstBridgeGetNewMaterial(bridge);
chassisMat = MstBridgeGetNewMaterial(bridge);

/* Set contact parameters for wheels */
params = MstBridgeGetContactParams(bridge,groundMat, wheelMat);
MdtContactParamsSetType(params,MdtContactTypeFriction2D);
MdtContactParamsSetPrimarySlip(params,0.0001f);
MdtContactParamsSetSecondarySlip(params,0.005f);
MdtContactParamsSetSoftness(params,0.0001f);

/* Set contact parameters for chassis */
params = MstBridgeGetContactParams(bridge,groundMat, chassisMat);
MdtContactParamsSetType(params,MdtContactTypeFriction2D);
MdtContactParamsSetSoftness(params,0.0001f);
```

Materials determine the type of contact that is generated when two collision models hit each other. The material itself does not have any properties. The properties are explicitly defined for the interaction between one material and another. The material friction is modified to 2D friction (frictionless is the default). For wheeled vehicles, friction with the ground is vital, hence additional wheel and chassis materials are defined. Defining properties here sets the default parameters for every generated contact between those materials. The parameters of individual contacts can be modified on their own - this will be demonstrated later.

Adding the Update Functions to the Main Loop

When the application is running the Karma components need to be periodically updated to calculate new positions and orientations of the bodies and detect any subsequent collisions. This is normally done once per frame.

File: main.c

Function: `tick()`

Identifier: PASTE_03

```
McdSpaceUpdateAll(space);
MstBridgeUpdateContacts(bridge, space, world);
```

This is the collision detection phase. `McdSpaceUpdateAll()` keeps track of all the collision models and produces a list of pairs of models whose bounding boxes are overlapping. This information is accessible and could be used for AI or game logic, but in this case it will just be used for generating the contacts. `MstBridgeUpdateContacts()` takes the list of overlapping bounding boxes and performs the precise geometry tests between the collision models. Any models that are actually touching will produce a set of `MdtContacts` between the appropriate bodies. All the `MstBridge` code is provided with the toolkit so its functionality can be customized if required

File: main.c

Function: `tick()`

Identifier: PASTE_04

```
MdtWorldStep(world, timeStep);
```

This is the dynamics update phase. For simplicity, this application uses a fixed time step however for real time behavior the elapsed time since the last update would be used. The position, orientation and velocity of each enabled body is updated during this phase. If the application is run the world will still appear a very empty place but it is now ready for the insertion of dynamic bodies and collision models.

Creating the Terrain

Collision models can either be attached to bodies or exist on their own. The difference is that a model without a body will not react if another model collides with it. Generally these are used for parts of the terrain and static objects.

File: terrain.c

Function: `InitialiseTerrain()`

Identifier: PASTE_05

```
#if USE_TRI_LIST
#define TRI_LIST_SIZE 50
    MeVector3 min = {-10000,-10000,-10000}, max = {10000,10000,10000};

     MeALIGNDATA(MeMatrix4,tm,16) =
     {
       {  1,  0,  0,  0},
       {  0,  1,  0,  0},
       {  0,  0,  1,  0},
       {  0,  0,  0,  1}
     };

     /* Terrain height field */
     HeightFieldFromBMP(&landscape.heightField, "terrain2", 15);

     /* Just use a real big bounding-box... crude */
     landscape.collGeom =McdTriangleListCreate(framework,
                                          min, max,TRI_LIST_SIZE,
                                          TriListGeneratorCB);
     ((McdTriangleList*)landscape.collGeom)->triangleMaxCount = TRILIST_SIZE;
#else

     MeALIGNDATA(MeMatrix4,tm,16) =
     {
     /* Defaults to XY plane so rotate by PI/2 about x-axis to orientate with XZ
        plane*/
       {  1,  0,  0,  0},
       {  0,  0, -1,  0},
       {  0,  1,  0,  0},
       {  0,  0,  0,  1}
     };
     landscape.collGeom = McdPlaneCreate(framework);
#endif

     landscape.collModel = McdModelCreate(landscape.collGeom);
     MeMatrix4Copy(landscape.transform, tm);
     McdModelSetTransformPtr(landscape.collModel, landscape.transform);
     McdSpaceInsertModel(space, landscape.collModel);
     McdModelSetMaterial(landscape.collModel, groundMat);
     McdSpaceUpdateModel(landscape.collModel);
     McdSpaceFreezeModel(landscape.collModel);
```

USE_TRI_LIST determines whether the terrain geometry is a flat plane or an arbitrary polygonal surface. It is useful to be able to quickly switch back to a flat plane for assessing vehicle behavior and other testing purposes. Creating a flat plane is straighforward but it defaults to having its normal along the Z axis. In this application, gravity acts down the Y axis so the plane is rotated accordingly. For rough terrain, using a TriangleList geometry rather than a flat plane is the best approach. The main difference between a TriangleList and every other type of `McdGeometry` is that the actual collision geometry is undefined. It is created with the extents of its bounding box and an application level callback function. When `McdSpaceUpdateAll()` detects that this bounding box is overlapping with a second model, the bridge will subsequently call this callback function. It is here that the application decides which triangles should be tested against the second collision model. This provides a flexible mechanism that can be integrated with any proprietary terrain data storage format. In this application, the terrain is a regular grid height field constructed from a bitmap. This makes it very easy to return a set of triangles local to the position of another collision model (See `TriListGeneratorCB()` in Terrain.c).

Once the collision geometry has been created, the corresponding model can be created and inserted into the collision space. Only models inserted into the same space will collide with each other. Because this model isn't associated with a dynamic body, it needs a substitute transformation matrix to position and orientate it in the world. The memory for this matrix needs to be preserved for the lifetime of the collision model. Since this model is static it can be set as frozen in the space. This means that its bounding box doesn't get updated in `McdSpaceUpdateAll()`. However, it does have to be updated once so `McdSpaceUpdateModel()` is called before freezing. If the collision model needed to be moved then `McdSpaceUpdateModel()` would need to be called again after changing the transformation matrix. Note that only the physics geometry is created by these calls – the rendered geometry is created separately by calls to `Create**Graphics()`.

It should now be possible to build and run the application with either flat or rough terrain.

Creating a Four Wheel Vehicle

A basic four wheel vehicle consists of five dynamic bodies (one chassis + four wheels). Each wheel body is connected to the chassis body using a suspension joint. Each of the suspension joints incorporates the steering and drive control for the wheel as well as the basic suspension action.

File: vehicle.c

Function: `InitialiseVehicle()`

Identifier: PASTE_06

```
veh->chassisBody = MdtBodyCreate(world);
veh->chassisGeom = McdBoxCreate(framework,
                                 veh->data->chassisDims[0],
                                 veh->data->chassisDims[1],
                                 veh->data->chassisDims[2]);
veh->chassisCM = McdModelCreate(veh->chassisGeom) ;
McdModelSetBody(veh->chassisCM, veh->chassisBody);
McdModelSetMaterial(veh->chassisCM, chassisMat);
McdSpaceInsertModel(space, veh->chassisCM);
MdtBodyEnable( veh->chassisBody );
MdtBodySetPosition(veh->chassisBody,startPos[0],startPos[1],startPos[2]);
MdtBodySetMass(veh->chassisBody, 150);

Ixyz[0][0] = 5;      /* 1/12 * m * (y2 + z2) */
Ixyz[1][1] = 45;     /* 1/12 * m * (x2 + z2) */
Ixyz[2][2] = 40;     /* 1/12 * m * (x2 + y2) */
MdtBodySetInertiaTensor(veh->chassisBody, Ixyz);
MdtBodySetCenterOfMassRelativeTransform(veh->chassisBody, comTM);
```

The first stage is to create the chassis body and collision model. In this case, the collision model used is a single box primitive. By changing the geometry type, however, the chassis collision could also use an aggregate of smaller primitives or even a convex hull although this is usually unnecessary. The center of mass of the chassis is lowered from the center of the body using a relative offset. This makes the vehicle less likely to roll over when cornering. Again note that we have to insert the collision model into the collision space. We also have to set the inertial tensor to some suitable value; the default is for a sphere of radius 1 and as such is inappropriate for a long box such as our chassis. Bear in mind that there are utility functions, such as `MstModelAndBodyCreate()` that automate some of these tasks. Build and run.

File: vehicle.c

Function: `InitialiseVehicle()`

Identifier: PASTE_07

```
for(i = 0; i < 4; i++)
{
     veh->wheelBody[i] = MdtBodyCreate(world);
     veh->wheelGeom[i] = McdSphereCreate(framework,veh->data->wheelRadius);
     veh->wheelCM[i]   = McdModelCreate(veh->wheelGeom[i]) ;
     McdModelSetBody(veh->wheelCM[i], veh->wheelBody[i]);
     McdModelSetMaterial(veh->wheelCM[i], wheelMat);
     McdSpaceInsertModel(space, veh->wheelCM[i]);
     MdtBodyEnable( veh->wheelBody[i] );
     MdtBodySetMass(veh->wheelBody[i], 10);

     Ixyz[0][0] = Ixyz[1][1] = Ixyz[2][2] = 0.25;   /* 2/5 * m * r2 */

     MdtBodySetInertiaTensor(veh->wheelBody[i], Ixyz);
     MeVector3Add(pos, startPos, veh->data->wheelOffset[i]);
     MdtBodySetPosition(veh->wheelBody[i], pos[0], pos[1], pos[2]);

     /* Disable collision between wheel and chassis */
```

```
            McdSpaceDisablePair(veh->wheelCM[i], veh->chassisCM);

            /* Disable collision between wheel and other wheels */
            for(j = i-1; j >= 0; j--)
            {
                McdSpaceDisablePair(veh->wheelCM[i], veh->wheelCM[j]);
            }
        }
```

The next stage is to create each wheel body and collision model. Each wheel is positioned at the appropriate offset to the starting position of the chassis. A spherical collision model is used since it produces the minimum number of contact points with the terrain. A section of cylinder could be used or a convex hull although this would require a large number of facets to give smooth rolling behavior. Generally a sphere will give the best overall performance. As each model is inserted into the collision space the interaction between it and the other vehicle components is disabled by disabling the appropriate model pairs. Build and run.

File: vehicle.c

Function: `InitialiseVehicle()`

Identifier: PASTE_08

```
        for(i = 0; i < 4; i++)
        {
            veh->wheelJoint[i] = MdtCarWheelCreate(world);
            MdtCarWheelSetBodies(
                        veh->wheelJoint[i],
                        veh->chassisBody,   /* chassis body must be specified first */
                        veh->wheelBody[i]);

            MdtBodyGetPosition(veh->wheelBody[i], pos);
            MdtCarWheelSetPosition(veh->wheelJoint[i], pos[0], pos[1], pos[2]);
            MdtCarWheelSetSteeringAndHingeAxes(veh->wheelJoint[i],
                        0, 1, 0/* along the Y axis */,0, 0, 1/* along the Z axis */);

            if (i == BACK_LEFT || i == BACK_RIGHT)
        {
                /* We don't want steering on the back wheels so lock them. */
                MdtCarWheelSetSteeringLock(veh->wheelJoint[i], 1);
            }

            MdtCarWheelSetSuspension(veh->wheelJoint[i], 50000, 0.4f, 0.001f,
                                    -0.15f, 0.15f, 0);
            MdtCarWheelEnable(veh->wheelJoint[i]);
            MdtBodySetUserData(veh->wheelBody[i], (void *)veh->wheelJoint[i]);
        }
```

Once the bodies have been created and initialized to their starting positions, the suspension joints are initialized. Because a joint contains information about the relative position of attached bodies, it is vital to set the bodies that the joint attaches first. Also, when setting the bodies for the suspension joints, it is essential that the chassis is specified first otherwise the resulting behavior will not be what is expected. The initial position is specified as the position of the wheel with the hinge axis aligned with the world z axis and the hinge axis aligned with the world Y axis. The two rear wheels have the steering constraint locked since for this vehicle it is just the front wheels that are used for steering. The wheel body’s user data element is used to store a void pointer to the suspension joint. This provides a useful method of accessing the joint from the body or collision model. This will be apparent when using callback functions from the collision detection. Build and run.

File: vehicle.c

Function: `InitialiseVehicle()`

Identifier: PASTE_09

```
        veh->hBarBody = MdtBodyCreate(world);
        MdtBodyEnable( veh->hBarBody );
        MeVector3Add(pos, startPos, veh->data->hBarOffset);
        MdtBodySetPosition(veh->hBarBody, pos[0], pos[1], startPos[2]);

        veh->hBarJoint = MdtHingeCreate(world);
        MdtHingeSetBodies(veh->hBarJoint, veh->chassisBody, veh->hBarBody);
        MdtHingeSetAxis(veh->hBarJoint, 0, 1, 0);
        MdtBodyGetPosition(veh->hBarBody, pos);
        MdtHingeSetPosition(veh->hBarJoint, pos[0], pos[1], pos[2]);
        MdtHingeEnable(veh->hBarJoint);
```

The final additional step is to add some handlebars to the vehicle. They provide the mechanism for actuating the rider’s arms. If the handlebars were just for effect then it would probably be easier to just animate them. A hinge joint is used to connect the handlebar body to the chassis body. For simplicity, the handlebars haven’t been given a collision model although this would be easy to add if required. Build and run.

Adding User Control to the Vehicle

The handling of the quadbike depends on the interface. Because of the limited nature of the key presses in the renderer (i.e. the PC only detects ’key down events’), you may like to investigate more sophisticated control. An effect of this is that you cannot steer and accelerate at the same time. If you are accelerating and then steer you will stop accelerating. If you then stop steering, you will need to release the acceleration key and press it down again to accelerate.

Three basic control inputs for acceleration, braking and steering are added to the vehicle.

File: vehicle.c

Function: `UpdateVehicleControls()`

Identifier: PASTE_10

```
torque = veh->throttleInput * -200;

/* Apply torque to wheel body */
MdtBodyAddTorque(veh->wheelBody[i],
                torque * haxis[0],
                torque * haxis[1],
                torque * haxis[2]);

/* Apply opposite torque to chassis body */
MdtBodyAddTorque(veh->chassisBody,
                -torque * haxis[0],
                -torque * haxis[1],
                -torque * haxis[2]);
```

To accelerate the vehicle forward, a torque is directly applied to the wheel body around the wheel’s hinge axis. For each torque applied to a wheel, an equal but opposite torque is applied to the chassis. Although it is more correct to apply this opposing torque, it can be omitted. This can help if the vehicle has a tendency to tip over backwards under acceleration. The torque is directly proportional to the throttle input. This is very simplistic and will cause the vehicle to accelerate indefinitely. A better engine model would have the torque as a function of the angular velocity of the wheels and tending to zero as the speed increases. Build and run.

File: vehicle.c

Function: `UpdateVehicleControls()`

Identifier: PASTE_11

```
else
{
     /* Apply brakes */
     MeReal maxBrakeTorque = veh->brakeInput*500;
     MdtCarWheelSetHingeLimitedForceMotor(veh->wheelJoint[i], 0,
                                          maxBrakeTorque);
}
```

The vehicle could be slowed in a similar way to acceleration by applying a torque directly to the wheels. It is more convenient, however, to use a limited force motor with a desired angular velocity of zero around the hinge axis. This eliminates the problem of the braking torque causing the angular velocity to oscillate around zero. It also makes it easy to hold the vehicle stationary on a slope. Similarly, the limited force motor could be used to accelerate the vehicle and would act as a direct speed control rather than acceleration control. The braking control can be pasted in 2 places for braking on the front wheels, back wheels or both. Build and run.

File: vehicle.c

Function: `UpdateVehicleControls()`

Identifier: PASTE_12

```
/* Front wheel do the steering proportional gap (radians). */
theta = MdtCarWheelGetSteeringAngle(veh->wheelJoint[i]);

desired_vel = veh->steeringInput * width + theta;

desired_vel = max(-pgap, min(desired_vel, pgap));

MdtCarWheelSetSteeringLimitedForceMotor(veh->wheelJoint[i],
                                        maxSpeed * desired_vel, maxForce);
```

The steering control also uses a limited force motor on the suspension joint although this time around the vertical steering axis. The desired wheel angle is proportional to the steering input value. Since the limited force motor will control the steering rate, a required speed is derived from the difference between the desired wheel steer angle and the wheel's current steer angle. It is possible to set the wheel angle directly by animating the body manually but this is actually more complicated since the angular velocity vector and the joint fast spin axis need to be rotated manually with the body. Build and run.

File: vehicle.c

Function: `UpdateVehicleControls()`

Identifier: PASTE_13

```
MdtLimitController(MdtHingeGetLimit(veh->hBarJoint), -veh->steeringInput* width,
                                     pgap, maxSpeed, maxForce);
```

A limit controller is used to steer the handlebars. This is very similar to using a limited force motor although the parameters are passed in one function call. It is used here to demonstrate the different methods of controlling joints. The handlebars will act as the actuation mechanism for the rider's arms. Build and run.

File: main.c

Function: `InitialiseMaterials()`

Identifier: PASTE_14

```
MstBridgeSetPerContactCB(bridge, wheelMat, groundMat, WheelGroundCB);
```

Finally, to modify the exact behavior of the contacts between the wheels and the ground, an application level callback function is implemented for the interaction between the wheel material and the ground material.

File: vehicle.c

Function: `WheelGroundCB()`

Identifier: PASTE_15

```
body = MdtContactGetBody(dynC, 0);
/* terrain has no body so wheel must be body 0 */

wj = (MdtCarWheelID)MdtBodyGetUserData(body);

if(wj)
{
     /* Create principal friction axis (normal x hinge_axis). */

     MdtCarWheelGetHingeAxis(wj, haxis);
     MdtContactGetNormal(dynC, normal);

     MeVector3Cross(dir, normal, haxis);
     MeVector3Normalize(dir);

     MdtContactSetDirection(dynC, dir[0], dir[1], dir[2]);

     /* Increase lateral slip with increasing camber angle */
     params = MdtContactGetParams(dynC);
     slip = 0.001f + MeFabs(MeVector3Dot(normal, haxis));
     MdtContactParamsSetSecondarySlip(params, slip);
}
```

The callback function performs two tasks. Firstly, it aligns the contact direction with the direction in which the wheel is rolling. This is not essential if the primary and secondary friction parameters are similar but this is not often the case. Secondly, the lateral slip parameter is increased with increasing camber of the wheel. This means that the wheel will slip sideways more as the roll angle between it and the ground increases. This helps to reduce the tendency of the vehicle to tip over when cornering. The callback function could also be used to adjust the friction parameters depending on other factors such as localized terrain texture etc. Build and run.

Creating a Rider for the Vehicle

The process for creating the rider follows the same steps as creating the vehicle. The bodies and collision models are created and then connected together with various joints. Joint axes and limits are used to give posture. The controls for the rider adjust some of the joint limits to add additional articulation.

File: rider.c

Function: `InitialiseRider()`

Identifier: PASTE_16

```
for(i = 0; i < NUM_LIMBS; i++)
{
     rider->limbBody[i] = MdtBodyCreate(world);
     MeVector3Add(pos, startPos, rider->data->limbPos[i]);
     MdtBodySetPosition(rider->limbBody[i], pos[0], pos[1], pos[2]);
     MeMatrix3FromEulerAngles(R, rider->data->limbAng[i][0],
                                 rider->data->limbAng[i][1],
                                 rider->data->limbAng[i][2]);

     MdtBodySetOrientation(rider->limbBody[i], R);
     MdtBodyEnable(rider->limbBody[i]);

     /* Collision */
     rider->limbGeom[i] = McdBoxCreate(framework,
                                       rider->data->limbDim[i][0],
                                       rider->data->limbDim[i][1],
                                       rider->data->limbDim[i][2]);
     rider->limbCM[i] = McdModelCreate(rider->limbGeom[i]);
     McdModelSetBody(rider->limbCM[i], rider->limbBody[i]);
     McdSpaceInsertModel(space, rider->limbCM[i]);

     /* Disable collision between all the various limbs */
     for(j = i-1; j >= 0; j--)
     {
          McdSpaceDisablePair(rider->limbCM[i], rider->limbCM[j]);
     }

     /* Disable collision rider and vehicle */
     McdSpaceDisablePair(rider->limbCM[i], quadBike.chassisCM);
     for(j = 0; j < 4; j++)
     {
          McdSpaceDisablePair(rider->limbCM[i], quadBike.wheelCM[j]);
     }
}
```

All the initial positions and orientations are stored in the data structure. In this application the rider's right and left lower limbs are combined for simplicity. If the rider was intended to fall off the vehicle then it would probably be necessary to make the legs independent. Each limb uses a box geometry to represent the collision surface. The collision between every body part and between each body part and the components of the vehicle is disabled.

File: rider.c

Function: `InitialiseRider()`

Identifier: PASTE_17

```
for(i = 0; i < NUM_JOINTS; i++)
{
     /* Determine joint bodies */
     body1 = rider->limbBody[rider->data->jointBodies[i][0]];

     if(rider->data->jointBodies[i][1] == -1)
     {
          if(rider->data->jointBodies[i][0] == LOWER_LEGS)
          {
               body2 = quadBike.chassisBody;
```

```
                } else {
                    body2 = quadBike.hBarBody;
                }
            } else {
                body2 = rider->limbBody[rider->data->jointBodies[i][1]];
            }

            /* Create appropriate joint */
            if(rider->data->isHinge & 1<<i)
            {
                hinge = MdtHingeCreate(world);
                MdtHingeSetBodies(hinge, body1, body2);
                MeVector3Add(pos, startPos, rider->data->jointPos[i]);
                MdtHingeSetPosition(hinge, pos[0], pos[1], pos[2]);
                MdtHingeSetAxis(hinge,
                                rider->data->jointAxis[i][0],
                                rider->data->jointAxis[i][1],
                                rider->data->jointAxis[i][2]);

                rider->joint[i] = MdtHingeQuaConstraint(hinge);

                /* Add hinge joint limit code here */
                /* PASTE_18 */
            }
            else
            {
                bsj = MdtBSJointCreate(world);
                MdtBSJointSetBodies(bsj, body1, body2);
                MeVector3Add(pos, startPos, rider->data->jointPos[i]);
                MdtBSJointSetPosition(bsj, pos[0], pos[1], pos[2]);

                rider->joint[i] = MdtBSJointQuaConstraint(bsj);
            }
            MdtConstraintEnable(rider->joint[i]);
        }
```

The rider uses both ball and socket joints and hinge joints. The hinge joints are used where the motion needs to be more controlled. The shoulder uses a hinge with the axis rotated forwards slightly. This causes the elbow to move outwards as the arm is moved back. The axis of the hinge in the riders hips is aligned front to back so that the rider has some lateral flexibility. It is important not to over constrain the system by using too many hinges. For instance, a hinge could not be simultaneously used in the elbow and shoulder because the arm would not have enough flexibility to move when the wrist is jointed to the handlebars. Build and run.

File: rider.c

Function: `InitialiseRider()`

Identifier: PASTE_18

```
            if(rider->data->jointParam[i][0])
            {
                limit = MdtHingeGetLimit(hinge);
                MdtSingleLimitSetStiffness(MdtLimitGetUpperLimit(limit),
                                           rider->data->jointParam[i][0]);
                MdtSingleLimitSetStiffness(MdtLimitGetLowerLimit(limit),
                                           rider->data->jointParam[i][0]);
                MdtSingleLimitSetDamping(MdtLimitGetUpperLimit(limit),
                                           rider->data->jointParam[i][1]);
                MdtSingleLimitSetDamping(MdtLimitGetLowerLimit(limit),
                                           rider->data->jointParam[i][1]);
                MdtLimitActivateLimits(limit, 1);
            }
```

To give the character the correct posture, the limits are activated on each of the hinge joints. If the upper and lower limits have the same value, the hinge will always return to that position. Since the character is initialized in the correct posture, the value for upper and lower limits can be left as zero. The stiffness and damping of each limit is set to give the desired compliance of the rider. Build and run.

File: rider.c

Function: `UpdateRiderControls()`

Identifier: PASTE_19

```
    /* Make the crouch angle proportional to forward velocity */
    MdtBodyGetLinearVelocity(quadBike.chassisBody, vel);
    xAxis = (MeVector3Ptr)MdtBodyGetTransformPtr(quadBike.chassisBody);
    speed = max(0, MeVector3Dot(xAxis, vel));
    angle = speed * 0.035f;
    angle = min(angle,maxAng);
    angle = -quadBike.steeringInput * angle;

    /* Move hip joint limit to force rider to lean into turn */
    limit = MdtHingeGetLimit(MdtConstraintDCastHinge(rider->joint[HIPS]));
    MdtSingleLimitSetStop(MdtLimitGetLowerLimit(limit), angle);
    MdtSingleLimitSetStop(MdtLimitGetUpperLimit(limit), angle);

    /* Force rider to bend knees to crouch lower */
    angle = MeFabs(angle);
    limit = MdtHingeGetLimit(MdtConstraintDCastHinge(rider->joint[KNEES]));
    MdtSingleLimitSetStop(MdtLimitGetLowerLimit(limit), angle*2);
    MdtSingleLimitSetStop(MdtLimitGetUpperLimit(limit), angle*2);

    limit = MdtHingeGetLimit(MdtConstraintDCastHinge(rider->joint[ANKLES]));
    MdtSingleLimitSetStop(MdtLimitGetLowerLimit(limit), -angle*1.2);
    MdtSingleLimitSetStop(MdtLimitGetUpperLimit(limit), -angle*1.2);
```

The rider will passively move in response to the accelerations of the vehicle and the movement of the handlebars. To make the character look more involved with riding, the hinge limits can be actively controlled. Here the riders posture is altered as a function of the steering input and vehicle speed. By changing the upper and lower limit values for the knees, ankles and hips, the rider can be made to actively crouch and lean as the vehicle maneuvers. Build and run. You may at this point want to use a heightfield terrain rather than a flat ground plane to study the response of the rider moving over undulating terrain. This can be done by defining `USE_TRI_LIST` as `1` in `Terrain.c`.

Appendix A - Default Values

The following files contain the Karma default settings:

Precision

MdtDefaults.h, MePrecision.h

World

MdtWorld.c

Bodies

MdtBody.c

Joints

MdtAngular3.c, MdtBsJoint.c, MdtCarWheel.c, MdtConeLimit.c,MdtFixedPath.c, MdtHinge.c, MdtLimit.c, MdtLinear1.c, MdtLinear2.c, MdtPrismatic.c, MdtRPROJoint.c, MdtSpring.c, MdtUniversal.c

Contacts

MdtConstraint.c, MdtContact.c, MdtContactGroup.c, MdtContactParams.c

Universe

MstUniverse.c

Precision Values

MEINFINITY	3.402823466e+38F

World Properties

AutoDisable	1	s

AutoDisableVelocityThreshold	0.02	ms^{-1}
AutoDisableAngularVelocityThreshold	0.001	rad s^{-1}
AutoDisableAccelerationThreshold	0.5	ms^{-2}
AutoDisableAngularAccelerationThreshold	0.002	rad s^{-2}
AutoDisableAliveWindow	0.2	
MaxMatrixSize	MEINFINITY	
DebugDrawOptions	0	
Gravity	(0,0,0)	
Epsilon	0.01	kg^{-1}
Gamma	0.2	s^{-1}
MinSafeTime	0.001	

Although γ scales inversely with time, currently its value is given in *frames*, not seconds. Thus if you have a varying frame rate, you should adjust the value of γ each frame to ensure uniform relaxation.

Body Properties

Mass	m
InertiaTensor	0.4* ml^2 * (3x3 Identity Matrix)
LinearDamping	0
AngularDamping	0
FastSpinAxis	(0,1,0)

where m is the mass scale parameter to MdtWorldCreate and l is the length scale parameter

Joints

Joint Limit:

Position	0
Restitution	1
Stiffness	MEINFINITY
Damping	0
Powered	0
Desired velocity	0
Max force	0

Angular2:

RotationIsEnabled	0
axis1	(1,0,0)
axis2	(1,0,0)
ref_axis1	(0,1,0)
ref_axis2	(0,1,0)
stiffness	MEINFINITY
damping	MEINFINITY

Angular3:

RotationIsEnabled	0
stiffness	MEINFINITY
damping	MEINFINITY

Ball and Socket:

Position (in both bodies' reference frame)	(0,0,0)
Primary axis (in both bodies' reference frame)	(1,0,0)
Orthogonal axis (in both bodies' reference frame)	(0,1,0)
Car Wheel:	
Position	(0,0,0)
IsSteeringLocked (whether the wheel is steered)	0
Steering	
Axis	(0,0,1)
MotorMaxForce	1e9
MotorDesiredVelocity	0
Hinge	
Axis	(0,1,0)
MotorMaxForce	0
MotorDesiredVelocity	0
Suspension	
HighLimit	1e9
LowLimit	-1e9
Reference	0
LimitSoftness	0
Kp (Stiffness coeffiecient)	0
Kd (Damping coefficient)	0
Cone Limit:	
Position	(0,0,0)
PrimaryAxis	(1,0,0)
SecondaryAxis	(0,1,0)
Stiffness	MEINFINITY
Damping	MEINFINITY
Hinge	
Position	(0,0,0)
Axis	(1,0,0)
Full Constraint	
Position	(0,0,0)
relative orientation	(0,0,0)
Fixed Path	
Position	(0,0,0)
Velocity	(0,0,0)
Linear1	
Position	(0,0,0)
Linear2	
Position	(0,0,0)
Direction	(0,0,0)
Prismatic	
sliding axis (in body 0 reference frame)	(0,1,0)

body 0 initial position (in inertial reference frame)	(0,0,0)
Spring	
pos1	(0,0,0)
pos2	(0,0,0)
Universal Joint	
body1 = body2	0
PrimaryAxis	(1,0,0)
SecondaryAxis	(0,1,0)

Contacts

Contact:	
Position	(0,0,0)
Normal	(0,1,0)
FrictionDirection	(0,1,0)
Penetration	0
Contact Group:	
Position	(0,0,0)
Normal	(0,1,0)
friction direction	(0,1,0)
penetration	0
Contact Parameters:	
FrictionType	MdtContactTypeFrictionZero
PrimaryFrictionModel	MdtFrictionModelBox
PrimaryFriction	MEINFINITY
PrimaryFrictionCoefficient	0
PrimarySlip	0
PrimarySlide	0
SecondaryFrictionModel	MdtFrictionModelBox
SecondaryFriction	MEINFINITY
SecondaryFrictionCoefficient	0
SecondarySlip	0
SecondarySlide	0
Restitution	0
VelocityThreshold	0.001
Softness	0
Stickiness	0

MstUniverseDefaultSizes

dynamicBodiesMaxCount	100
dynamicConstraintsMaxCount	500
collisionUserGeometryTypesMaxCount	0
collisionModelsMaxCount	100
collisionPairsMaxCount	500
collisionGeometryInstancesMaxCount	100

materialsMaxCount	10
lengthScale	1
massScale	1

Appendix B - The Karma Viewer

The Karma Viewer (MeViewer) provides a set of functions designed to provide Karma developers with the basic rendering functionality required to demonstrate their simulation code. It is a basic cross platform wrapper around the OpenGL and Direct 3D libraries. It is only intended to be used for test code or prototyping, allowing basic display of, and interaction with, 3D scenes. `MeProfile` and `MeCommandLine` will be used together with MeViewer but are not covered. The source files and header files for these can be found in `...\metoolkit\src\components\MeGlobals\src and ...\metoolkit\include` respectively.

Using the Viewer in an Application

Let’s look at the `RainbowChain.c` sample program from Karma. It provides a good example of how you can use the Viewer in your own programs.

First Steps

You need first to include `MeViewer.h` and to declare a rendering context (a structure that holds the state of the viewer) of type `RRender`, which is traditionally named `rc`. Then the application declares pointers to the two graphics structures of type `RGraphic` that will appear in the RainbowChain tutorial: a ball and a plane.

```
#include "MeViewer.h"

/* Render context */
RRender *rc;

/* graphics */
RGraphic *sphereG[NBALLS];
RGraphic *planeG;
```

Initializing the Renderer

In the main routine of RainbowChain, we pick up the renderer type (`-gl`, `-d3d`, `-null`, `-bench`, `-profile`) from the command line:

```
/* Initialise rendering attributes. */

MeCommandLineOptions *options;
options = MeCommandLineOptionsCreate(argc, argv);
rc = RRenderContextCreate(options, 0, 1);
MeCommandLineOptionsDestroy(options);
if (!rc)
     return 1;
```

Then we initialize the camera in polar coordinates:

```
RCameraRotateElevation(rc, (MeReal)1.1);
RCameraRotateAngle(rc, (MeReal)0.2);
RCameraZoom(rc, 10);
```

Add a visual performance bar. This will show time taken up by collisions, dynamics and rendering:

```
RPerformanceBarCreate(rc);
```

Create the help system:

```
/* Create your help array. */
char *help[3] =
{
    "$ACTION1 - toggle options menu",
    "$ACTION2 - reset",
    "$MOUSE - apply mouse force"
};
RRenderCreateUserHelp(rc, help, 1);
RRenderToggleUserHelp(rc);
```

And assign the keyboard call-back function to resest objects on pressing return:

```
RRenderSetActionNCallBack(rc, 2, Reset, 0);
```

Creating the Graphics

Finally, we create the graphical objects. Spheres form the chain and a plane forms the floor. A pointer to each object’s collision model transformation matrix is used to link the graphic with the collision model.

```
for(i=0; i<NBALLS; i++)
{
     /* Convert Hue Saturation Value to Red Green Blue. */
     MeHSVtoRGB(((MeReal)i/(MeReal)NBALLS)*360, 1, 1, color);
     sphereG[i] = RGraphicSphereCreate(rc, ballRadius, color,
                                    McdModelGetTransformPtr(ball[i]));
}

color[0] = color[1] = color[2] = color[3] = 1;
planeG = RGraphicGroundPlaneCreate(rc, 24, 2, color, 0);
RGraphicSetTexture(rc, planeG, "checkerboard");
```

Running the Simulation

Then we call `RRun()` to run the simulation and render the graphics.

```
RRun(rc, tick, 0);
```

`RRun()` controls the event loop (the main loop) for the simulation. `tick` is a function which is called before rendering each frame, and in this application contains the code which updates the positions of each ball in the chain using Karma dynamics.

Terminating the Program

Calling `RRun()` sends the viewer into a loop, which—depending on your platform and your underlying graphics package—may not return. For safety, assume that your program will not return from `RRun()`.

To stop the program, the user presses <Esc> or clicks the close button, which results in a call to `exit()`, terminating your program. The effect is that no code placed after `RRun()` can execute.

So if you have any cleanup to do before your program terminates, put it in a function and register that function with `atexit()`. This way MeViewer will call that function before terminating the program. For example, here is a typical `cleanup()` function used in Karma:

```
void MEAPI_CDECL cleanup(void)
{
     RRenderContextDestroy(rc);
}
```

Register cleanup with `atexit()` before calling `RRun()`:

```
atexit(cleanup);
RRun(rc, tick);
/* No code placed after here will ever execute */
```

Render Contexts

A *Render Context* is a `RRender` structure (see the `MeViewerTypes.h` header file), that holds the state of the viewer. Because there is no global render context, nearly all API calls will require a valid `RRender*` as their first parameter . Therefore the first task of an application using MeViewer is to create a render context.

```
RRender * MEAPI RRenderContextCreate MeCommandLineOptions* options,
                                      MeCommandLineOptions* overrideoptions,
                                      MeBool eat);
```

creates a new render context, fills the `RRender` with default values and calls `RInit` in platform-dependent code. The return value is zero if creation or `RInit` fails. The `options` argument is an input from the command line, and options specified there are overridden by those specified in `overrideoptions`.

```
int MEAPI RRenderContextDestroy (RRender *rc);
```

cleans up and frees a render context.

```
void MEAPI RRenderQuit (RRender *rc);
```

tells the back-end to quit the `rc` render context next frame.

Render Lists

Graphics objects can be either 2D or 3D. When the renderer draws a frame, it walks through a linked lists for each type. The lists are of `RGraphic` objects. `RRender` holds the first element of each list, and the `RGraphics` themselves point to the next in the list:

```
void MEAPI RGraphicAddToList         ( RRender *rc,
                                       RGraphic *rg,
                                       int is2D)
```

Puts `RGraphic` into render-list in render context `rc`. The argument `is2D` determines whether `rg` is added to the 3D or 2D list

If a `RGraphic` object has been created, but is not in a list, it will not be rendered. This provides a mechanism for *disabling* objects:

```
void MEAPI RGraphicRemoveFromList    ( RRender *rc,
                                       RGraphic *rg,
                                       int is2D)
```

Removes `RGraphic` from render-list. The argument `is2D` specifies whether to look in 2D or 3D list.

Creating Primitives

In the following:

- the variable `color` is the RGBA color of the object
- `matrix` is a pointer to the object's transformation matrix
- the argument `rc` is the render context into whose (2D, 3D) list the resulting `RGraphic` is placed. The function returns a pointer to the resulting `RGraphic`, or `0` for failure.

The primitive creation functions follow.

```
RGraphic* MEAPI RGraphic2DRectangleCreate (RRender *rc, AcmeReal x, AcmeReal y,
                AcmeReal width, AcmeReal height, const float color[4]);
```

creates a rectangle where

- x and y are the x coordinate of the left edge of the rectangle and the y coordinate of the top edge of the rectangle respectively
- `width` and `height` are the x and y dimensions of the rectangle

```
RGraphic* MEAPI RGraphicGroundPlaneCreate (RRender *rc, AcmeReal length,
                int triangles, const float color[4], AcmeReal y_pos);
```

creates a ground-plane that is a square in x, z, and where

- `length` is the length of the side of the square
- `triangles` specifies the number of triangles per side of the square
- `y_pos` sets the y position of the square

```
RGraphic* MEAPI RGraphicBoxCreate (RRender *rc, AcmeReal x, AcmeReal y,
                AcmeReal z, const float color[4], MeMatrix4Ptr matrix);
```

creates a box of width x, height y and depth z.

```
RGraphic* MEAPI RGraphicConeCreate (RRender *rc, AcmeReal radius, AcmeReal uheight,
                AcmeReal lheight, const float color[4], MeMatrix4Ptr matrix);
```

creates a cone where

- `radius` is the radius of the base of the cone
- `uheight` is the length along z from the origin to the apex
- `lheight` is the length along z from the origin to the base

```
RGraphic * MEAPI RGraphicCylinderCreate (RRender *rc, AcmeReal radius,
                AcmeReal height, const float color[4], MeMatrix4Ptr matrix);
```

creates a cylinder where

- `radius` is the radius of the cylinder
- `height` is length along z of the cylinder
- The origin is located at the half height

```
RNativeLine* MEAPI RNLineCreate( RRender*const rc, const AcmeReal start[3],
                const AcmeReal end[3], const AcmeReal color[4],
                const MeVector4 *const matrix);
```

creates a native line where

- `start[3] is the position vector of the line origin`
- `end[3] is the position vector of the line end`

Note that native lines are not supported on the PlayStation®2, hence the following function should be used.

```
RGraphic * MEAPI RGraphicLineCreate (RRender *rc, AcmeReal *origin, AcmeReal *end,
                const float color[4], MeMatrix4Ptr matrix);
```

creates a line where

- `origin` is the (x, y, z) start location of the line
- end is the (x, y, z) end location of the line

Note that native line support also exists in Karma.

```
RGraphic * MEAPI RGraphicSphereCreate (RRender *rc, AcmeReal radius,
                const float color[4], MeMatrix4Ptr matrix);
```

creates a sphere where `radius` is the sphere radius.

```
RGraphic * MEAPI RGraphicSquareCreate (RRender *rc, AcmeReal side,
                const float color[4], MeMatrix4Ptr matrix);
```

creates a square, where `side` is the square side length.

```
RGraphic * MEAPI RGraphicTorusCreate (RRender *rc, AcmeReal innerRadius,
                AcmeReal outerRadius, const float color[4], MeMatrix4Ptr matrix);
```

creates a torus where `radius` is the outer radius of the torus.

```
RGraphic * MEAPI RGraphicTextCreate (RRender *rc, char *text, AcmeReal x,
                AcmeReal y, const float color[4]);
```

creates a `RGraphic` representing text.

- “font” as the texture
- `text_in` is the text to display (this is parsed first, allowing for variable substitution to take place in RParseText)
- x and y represent the respective x coordinate of the left edge of the text and the y coordinate of the top edge of the text

```
RGraphic * MEAPI RGraphicFrustumCreate (RRender *rc, AcmeReal bottomRadius,
                AcmeReal topRadius, AcmeReal bottom, AcmeReal top,
                int sides, const float color[4], MeMatrix4Ptr matrix);
```

creates an arbitrary frustum where

- `bottomRadius` and `topRadius` represent the radius of the approximation to a circle that forms the bottom and the top of the frustum respectively
- `bottom` and `top` represent the z coordinates, in the frustum's reference frame, of the bottom and the top of the frustum respectively
- `sides` is the number of sides of the regular polygon at each end of the frustum

Creating Objects

The platform-specific section of MeViewer only displays data in the format discussed in the later section discussing Geometry Data Formats, which holds lists of triangles, grouped into objects each with associated color and texture properties. The format is based around the `RGraphic` structure. MeViewer provides functions to create `RGraphic` structures representing a number of primitive objects, and to create empty structures for application defined objects.

To create an `RGraphic` object from an object file:

```
RGraphic* MEAPI RGraphicCreate (RRender *rc, char *filename, AcmeReal xScale,
          AcmeReal yScale, AcmeReal zScale, const AcmeReal color[4],
          MeMatrix4Ptr matrix, int is2D, int bKeepAspectRatio);
```

Creates a `RGraphic` from object file. A new `RGraphic` is created using the parameters passed. The value `filename` specifies the object geometry file.The values `xScale, yScale and zScale` specifies the x, y and z scaling factor respectively. The value `color` specifies the object RGBA color. The value `matrix` is the pointer to the associated transformation matrix. The value `is2D` indicates if object is to be put in 2D render-list And finally, the value `bKeepAspectRatio` indicates if the aspect ratio is to be preserved when object is normalized

Or you may want to create an empty one:

```
RGraphic *MEAPI RGraphicCreateEmpty (int numVertices);
```

Allocates memory for `RGraphic`. The `numVertices` argument specifies how many vertices in `RGraphic` and should be multiple of 3. Fills in `numVertices` and pointers in `RGraphic.` This function returns a pointer to the resulting `RGraphic`, or `0` if it fails to do so.

When you have created a `RGraphic` object, you may delete or destroy it using:

```
void MEAPI RGraphicDestroy (RGraphic *rg);
```

Frees memory allocated for `RGraphic`. The argument `rg` is the `RGraphic` to destroy. This function also frees memory allocated for the associated `RObjectHeader` and vertex list.

```
void MEAPI RGraphicDelete (RRender *rc, RGraphic *rg, int is2D);
```

Removes `RGraphic` from list and then frees up memory. Hence, there is so no need to call `RGraphicDestroy` afterwards. The arguments `rc` and `rg` are the render context that the graphic is associated with, and the graphic to delete respectively. The `is2D` argument queries whether the graphic is in the 2D list.

Manipulating RGraphic Objects

```
void MEAPI RNLineMoveEnds( RNativeLine*const line,
    const AcmeReal start[3], const AcmeReal end[3] );
```

changes the native line position to the new positions specified in `start[3]` and `end[3]`.

```
int MEAPI RGraphicLineMoveEnds (RGraphic *lineG, AcmeReal *origin, AcmeReal *end);
```

moves the ends of an `RGraphic` line. `lineG` is a pointer to the RGraphic representing a line, `origin` is a three-vector containing the co-ordinates of the start of the line and `end` is the three-vector containing the co-ordinates of the end of the line. This function returns `0` for success or `1` for failure (an `MeWarning` will be printed before returning in this case).

```
void MEAPI RGraphicSetTransformPtr(RGraphic *g, MeMatrix4Ptr matrix);
```

sets the transform pointer for an `RGraphic`. The argument `g` is the `RGraphic` in question and `matrix` is the transformation matrix to assign to g.

```
void MEAPI RGraphicSetColor           ( RGraphic *g,
                                        const float color[4] )
```

Sets the color of an `RGraphic`. The argument `g` is the `RGraphic` in question and `color` is the RGBA color to assign to the `RGraphic`. Note that this sets the ambient and diffuse components of the color and that the emissive and specular components are zeroed.

```
void MEAPI RGraphicGetColor           ( RGraphic *g,
                                        float color[4] )
```

Gets the color of an `RGraphic`. The argument `g` is the `RGraphic` in question and `color` is the returned ambient RGBA color of the `RGraphic`

```
void MEAPI RConvertTriStripToTriList ( RGraphic* rg,
                                       RObjectVertex* strips,
                                       int* stripSize,
                                       int* stripStart,
                                       int numStrips )
```

Converts a set of triangle strips to a list of triangles. Useful for back ends that do not support triangle strips, converting convex hulls and meshes to triangle lists. `strips` is a pointer to the first vertex of the first strip to be converted. All vertices for each strip follow in one contiguous chunk. The argument `stripSize` is an array whose i^{th} element contains the number of vertices in the i^{th} strip, `stripStart` is an array whose i^{th} element contains the index in`strips' of the first vertex of the $i^{th}$ strip and `numStrips` is the number of strips to process.

Menu System

The Viewer implements a simple menu system to simplify the use of an MeViewer2 application.

```
RMenu* MEAPI RMenuCreate (RRender* rc, const char* name);
```

Create a new on-screen menu. The argument `rc` is the render context that will display the menu and `name` is the title of the new menu. This function returns a pointer to the new menu.

```
void MEAPI RMenuDestroy (RMenu* rm);
```

Destroy a menu. The argument `rm` is the menu to destroy.

```
void MEAPI RRenderSetDefaultMenu (RRender *rc, RMenu* menu);
```

Make `menu` the default menu in a given render context. This means that this menu will be the one that appears when the menu key is pressed. The argument `rc` is the render context in which the menu is to be made the default.

```
void MEAPI RMenuDisplay (RMenu* rm);
```

Display a menu. The argument `rm` is the menu to display. Note that the render context in which this menu will be displayed is found out from rm.

```
void MEAPI RMenuAddToggleEntry (RMenu* rm, const char * name,
                RMenuToggleCallback func, MeBool defaultValue);
```

Add a `toggle' entry to a menu. This is an entry type that has two states, on and off, which are toggled by pressing the button when the entry is highlighted. The callback is called with the new value for each change of state.

The argument `rm` is the menu to which the entry will be added, `name` is the text to be displayed for this menu entry and `func` is the function that will be called when this menu entry is selected. The value `defaultValue` is the initial value for this toggle when it is created

```
void MEAPI RMenuAddValueEntry (RMenu* rm, const char * name,
                RMenuValueCallback func, MeReal hi, MeReal lo,
                MeReal increment, MeReal defaultValue);
```

Add a *value* entry to a menu. This is a menu entry type which provides a variable which can be modified by a specified stepsize between a minimum and maximum value by pressing the appropriate buttons when the entry is highlighted. The callback is called with the new value for each change of state.

The argument `rm` is the menu to which the entry will be added, `name` is the text to be displayed for this menu entry and `func` is the function to be called when the value is changed. The values `hi` and `lo` specify, respectively, the maximum value and minimum value this entry can take. The value `increment` is the amount by which the value is changed for each button press and `defaultValue` is the starting point for the value.

```
void MEAPI RMenuAddSubmenuEntry (RMenu* rm, const char * name, RMenu* submenu);
```

Add a *sub-menu* entry to a menu. This is an entry type that, when selected, opens another menu. The argument `rm` is the menu to which the entry will be added, `name` is the text that represents that menu entry and `submenu` is the menu that will be displayed when this entry is selected.

Help System

```
    void MEAPI RRenderCreateUserHelp (RRender *rc, char *help[], int arraySize);
```

builds the `RGraphic` representing user help. The argument `help` is an array of null-terminated strings and `arraySize` is the number of elements in the array.

```
    void MEAPI RRenderToggleUserHelp (RRender *rc);
```

toggles the display of user help. It is called by platform specific back-end. Toggles the pause state of associated render context. The rc argument is the render context for which to toggle the display of the help text.

The text passed to this function will be put through `RParseText`, and should make use of the variables that this function substitutes to describe the controls for an application.

The Camera

The camera position is set using spherical polar coordinates to specify a position relative a specified look-at point. The spherical polar coordinates are the distance from the lookat point, the angle `theta` on the xz-plane anticlockwise from the z-axis, and elevation `phi` above the xz plane. Angles are specified in radians, `theta` in the range from -π to +π and `phi` in the range -π/2 to +π/2. Whilst these values are held in the `RRender` structure, their values should not normally be set or read directly, but rather through the use of the functions listed in this section.

```
void MEAPI RCameraSetLookAt     ( RRender *rc,
                                  const AcmeVector3 lookAt )
```

This sets the look-at point to the specified world position `lookAt`. The camera position is automatically updated.The argument `rc` is the render context of the camera whose lookat to be set.

```
void MEAPI RCameraGetLookAt     ( RRender *rc,
                                  AcmeVector3 camlookat )
```

The look-at point in world coordinates of the `rc` render context is stored into the `camlookat` vector.

```
void MEAPI RCameraSetView       ( RRender *rc,
                                  AcmeReal dist,
                                  AcmeReal theta,
                                  AcmeReal phi )
```

This causes the camera's position to be calculated and updated from the `RRender` look-at point and the coordinates supplied. The `theta` and `phi` angles specify the angle and elevation in radians.

```
void MEAPI RCameraGetPosition   ( RRender *rc,
                                  AcmeVector3 campos )
```

The camera's position in world coordinates is calculated and stored into the `campos` vector.

Camera Movement

MeViewer provides a selection of functions for moving the camera by an incremental distance or angle. These are

- `RCameraZoom( RRender *rc, AcmeReal dist );`
- `RCameraPanX( RRender *rc, AcmeReal dist );`
- `RCameraPanY( RRender *rc, AcmeReal dist );`
- `RCameraPanZ( RRender *rc, AcmeReal dist );`
- `RCameraRotateAngle( RRender *rc, AcmeReal d_theta );`
- `RCameraRotateElevation( RRender *rc, AcmeReal d_phi );`

The argument `rc` is the render context whose camera is to be manipulated, `dist` is the displacement added to current camera distance, and `d_theta` and `d_phi` are the two rotation angles in spherical coordinates. Note that these functions will not allow the camera to get closer than `0.01f` from look-at. For more details, consult `MeViewer.h`.

```
void RCameraUpdate( RRender *rc );
```

Calculates the camera position and updates the camera matrix in the `RRender` structure from the look-at and spherical coordinates in that structure. The argument `rc` is the render context whose camera is to be manipulated.

It effectively synchronizes the various camera variables. It does not need to be called after using the other camera functions detailed in this section, but if any values are altered directly it should be called in order that the changes take effect.

Lighting

The number and type of lights in a render context is fixed. They are

- One ambient light source
- Two directional light sources
- One point light

The functions `RSwitchLightOn` and `RSwitchLightOff` switch lights on and off.

```
void RLightSwitchOn  ( RRender *rc, RRenderLight light)
void RLightSwitchOff ( RRender *rc, RRenderLight light)
```

Ambient Lighting

The RGB value of the ambient light is held in the `m_rgbAmbientLightColor[3]` member of `RRender`. Any changes to this will take effect immediately (From the beginning of the next frame). The `m_bUseAmbientLight` member of `RRender` can be set to zero to disable the ambient lighting.

Directional Lighting

`RRender` holds two `RDirectLight` structures (`m_DirectLight1` and `m_DirectLight2`) that describe the directional lights. These contain the RGB values of the ambient, specular and diffuse components of the lights, as well as a 3-vector that defines the direction in which the light points. The light is active if the `m_bUseLight` member is non-zero.

If alterations to the `RDirectLight` structures are made after `RRun` has been called, then it is necessary to set the `m_bForceDirectLight1Update` or `m_bForceDirectLight2Update` members of `RRender` to a non-zero value to instruct the renderer to update the lighting for the next frame.

Point Light

`RRender` holds a single `RPointLight` structure, `m_PointLight`, that defines the point light. As with the directional lights, this structure holds the RGB values and active state of the light. In place of direction, the 3-vector `m_Position[3]` member defines the location of the light in world coordinates. The attenuation of the light is controlled by the members `m_AttenuationConstant`, `m_AttenuationLinear` and `m_AttenuationQuadratic`.

As with the directional lights, if alterations to `m_PointLight` are made after `RRun`, then `m_bForcePointLightUpdate` should be set to a non-zero value.

Textures

The Viewer supports a maximum of 25 different textures. These should be 128X128X24bpp Windows `.bmp` files. It also supports 256X256 images, but as these take 4 times the memory, one should take care to reduce the number of textures used appropriately -- this will not be enforced automatically.

Every time a new texture is specified for an object, an identifier is created for it. The files are loaded when RRun is called. This means that all textures should have an identifier before the call to RRun. See `RCreateTextureID` below for details

```
int MEAPI RRenderTextureCreate  ( RRender *rc,
                                  char *filename )
```

Creates a Texture ID for `filename`. The argument `rc` is the render context into which the texture will be loaded and `filename` is the name of the texture file to attempt to load. Returns an ID for a texture filename or returns `-1` if all IDs are allocated.

```
int MEAPI RGraphicSetTexture    ( RRender *rc,
                                  RGraphic *rg,
                                  char *filename )
```

Sets the texture of a `RGraphic`. The argument `filename` specifies the name of the texture (the filename should not include the extension). Returns an ID for a texture filename or returns `-1` if all IDs are allocated.

Disabling an object's texture

To disable an object's texture, set its texture ID to `-1`. See *Geometry Data Format* on page 146 to find where this is stored. It may also be achieved by calling `RSetTexture` with an invalid filename, but this will have a larger overhead.

Controls

The Viewer provides ten buttons and mouse analog controls to an application. The application can assign a function to each of these using the following identifiers.

Button Press Controls

Exactly which key corresponds to which callback being called is determined by the platform-specific layer. The call-back will be invoked only when the button is pressed, with the single argument specifying which render context is calling the function.

The functions

```
      void MEAPI RRenderSet*CallBack (RRender *rc, RButtonPressCallBack func);
```

where the wildcard * is one of Up, Up2, Down, Down2, Left, Left2, Right, or Right2, sets the callback for `*`. Each of these takes a `RRender*` render context and a function pointer as arguments. `rc` is the render context whose callback is to be set and `func` is the callback to assign to this button.

```
      void MEAPI RRenderSetActionNCallBack (RRender *rc, int N,
                      RButtonPressCallBack func, void *userdata);
```

sets the call-back for the Nth Action. The value `N` represents the number of the callback that is to be set (usually from 0 to 5). `func` is called when the specified action key is pressed.

```
      void MEAPI RRenderSetActionNKey (RRender* rc, const unsigned int N,
                      const char key);
```

assigns a key to a given Action Callback. The value `N` represents the number of the callback to assign the key to (usually in the range 2 to 5). The argument `key` is the key character to assign to the nth action callback.

Analog Controls

The analog call-back will be called when the platform-defined analog control has *changed* position. The arguments to the call-back specify the associated render context and the current (x, y) position of the controller. The range of these position values is *not* specified, and so the call-back should only use the difference in values between successive calls to be truly cross-platform compatible.

```
      void MEAPI RRenderSetMouseCallBack (RRender *rc, RMouseCallBack func,
                      void *userdata);
```

sets the mouse callback where `rc` is the render context whose callback is to be set and `func` is the callback to assign to this button.

Performance Measurement

```
RPerformanceBar * MEAPI RPerformanceBarCreate  ( RRender *rc )
```

This function will add a performance bar to the render context `rc`. It is the responsibility of the renderer to update this bar with timings. This function returns a pointer to the newly created performance bar, or `0` if it fails to do so.

Utilities

```
void MEAPI RParseText                ( RRender *rc,
                                       char *text_in,
                                       char *text_out,
                                       int outbuffersize )
```

Parses text, substituting text for variables. Variables defined by $ followed by capitals or numbers are substituted by text in `RRender`. The argument `rc` is the rendering context source for strings to be substituted. The argument `text_in` is the input string containing variables to be substituted and `text_out` is the output string returned with text substituted for variables. The value `outbuffersize` represents the amount of memory you allocated that is pointed to by `text_out` (i.e., maximum size of output string).

Any series of capital letters or numbers following a $ character will be considered a variable for substitution. Those that are currently recognized are

- `$UP $DOWN $LEFT $RIGHT`
- `$UP2 $DOWN2 $LEFT2 $RIGHT2`
- `$ACTION1 $ACTION2 $ACTION3 $ACTION4 $ACTION5`
- `$APPNAME`

When these are found, they are replaced by the text held in the `RRender` structure. This allows the platform-specific renderer to provide names for the buttons and controls that are described in *Controls* on page 143. The application should set the `m_AppName` member of `RRender` to a null-terminated string that names the application.

Geometry Data Format

As well as being able to render primitives, the viewer can display any object described by a triangle list. These can be loaded from files, as covered in *Object Geometry Files* on page 148, or created at run-time by using the structures detailed below.

An object in MeViewer consists of a single `RObjectHeader` structure, followed by a number of `RObjectVertex` structures, each of which describes a single vertex in the triangle list. The `RGraphic` structure holds pointers to these as well as providing the linked-list mechanism.

Creating New Objects

The functions `RGraphicCreate() and RGraphicCreateEmpty()`, described in *Creating Objects* on page 136, should be used to allocate the memory and fill in the `RGraphic` structure for any object. It is possible to create an object with any number of vertices, but it should be borne in mind that only multiples of three will produce *valid* objects (remember that MeViewer uses triangle *lists* and not strips). Consisting of merely a triangle list, an object is simply a set of triangles that share the same transformation matrix, color and texture.

The `RGraphic` structure holds pointers to the `RObjectHeader` and the first `RObjectVertex` in an object. The first `RObjectVertex` structure *must* immediately follow the `RObjectHeader` in memory, and so the pointer to it in `RGraphic` is purely for ease of programming vertex manipulation routines. The `m_pLWMatrix` member points to the transformation matrix for the object.

The member `m_nMaxNumVertices` should always be set to the maximum number of vertices that have memory allocated for them following the `RObjectHeader`. This is not necessarily the same as the number of vertices in the object, which is held in the `RObjectHeader`. This allows objects with varying numbers of vertices to be created with a minimum of memory allocation and copying.

Every graphical object in MeViewer has a single `RObjectHeader`. Its members are:

Member	Description
`m_Matrix`	This is filled in by the renderer, but should be set to the identity matrix if the `m_pLWMatrix` of the parent `RGraphic` is null.
`m_nNumVertices`	This indicates the number of RObjectVertex structures that make up the object. It *must* be a multiple of three.
`m_nTextureID`	The identifier for the object's texture. See *Textures* on page 142 for details. If it is -1 then the object is not textured.
`m_bIsWireFrame`	If this is non-zero, the renderer is requested to draw the object in wire-frame mode. This is not implemented on all platforms.
`m_ColorAmbient`	The RGBA ambient color of the object.
`m_ColorDiffuse`	The RGBA diffuse color of the object.
`m_ColorEmissive`	The RGBA emissive color of the object.
`m_ColorSpecular`	The RGBA specular color of the object.
`m_SpecularPower`	A value that indicates the shininess of the object.

RObjectVertex

Each vertex in a MeViewer object consists of a position (`m_X`, `m_Y`, `m_Z`), a normal (`m_NX`, `m_NY`, `m_NZ`) and a pair of texture coordinates (`m_U`, `m_V`). These can be updated at any time, and the changes will be reflected as soon as the next frame is drawn. Note that this does not apply to the PlayStation®2.

The normal should have a modulus of 1. Each texture coordinate should be between 0 and 1.

Object Geometry Files

The viewer can load geometries from .ASE files

File Format

```
     RGraphic * MEAPI RGraphicLoadASE (RRender *rc, char* filename,
                    AcmeReal xScale, AcmeReal yScale, AcmeReal zScale,
                    const float color[4], MeMatrix4Ptr matrix);
```

loads in ASE file and creates a graphics object for it. Material information from the ASE file is ignored (although texture co-ords are used). The overall color and texture are set in the same way as all other RGraphics.

- `filename` is the ASE file to be loaded
- `scaleFactor` is the amount to scale graphics file by on loading
- `color` is the RGBA colour of the graphic
- `matrix` a pointer to the local-world transform for this graphic.

A pointer to the resulting RGraphic, or 0 for failure, is returned.

Appendix C - Memory Allocation

Memory Allocation in Karma

Introduction

When developing games (especially for consoles) it is necessary to track and limit the amount of memory used. All memory allocation in Karma goes through the MeMemoryAPI. This is a struct containing function pointers to the functions used for memory allocation and freeing (see MeMemory.h and MeMemory.c in MeGlobals for more information.

Object Code and Static Data

Static data is data that is known at compile time. The table below provides information about Karma memory usage for the PlayStation®2. This applies to the Karma 1.2 build:

Library Name	**Function**	Code (KB)	Static Data (KB)	Source Code Provided
libMcdCommon.a	Farfield and collision math library	19.5	2.2	●
libMcdConvex.a	Convex object collision detection	78.9	4.1	
libMcdConvexCreateHull.a	Utility for creating convex hull from a mesh	270.1	5.4	●
libMcdFrame.a	Collision framework. Includes farfield management	31.5	4.2	
libMcdPrimitives.a	Primitive (cylinder, sphere, cube etc) collision detection.	150.8	4.6	
libMdt.a	Partitioning, freezing, dynamics and accessors	47.3	0.0	●
libMdtBcl.a	Constraint library	58.1	0.3	
libMdtKea.a	Core dynamics solver	120.6	7.8	
libMeApp.a	Demo utilities eg mouse picking	9.1	0.0	●
libMeAssetDB.a	Manages asset data structure	26.0	0.0	●
libMeAssetDBXMLIO.a	Loads and Saves asset database from and to XML	21.0	0.0	●
libMeAssetFactory.a	Instances asset database and turns asset structure into Karma object	8.9	0.0	●

Library Name	**Function**	Code (KB)	Static Data (KB)	Source Code Provided
libMeGlobals.a	Memory management, basic maths functions etc	52.9	0.3	●
libMeViewer.a	Demo renderer	73.2	1.4	●
libMeXML.a	XML read/write support	16.1	0.0	●
libMst.a	Collision/dynamics bridge	7.1	0.0	●

Note that the PC version of Karma requires less than this although it usually has more memory at its disposal. The consumer PlayStation®2 has 32MB of memory and the PlayStation®2 developer kits have 128MB of memory.

Dynamic Data

Dynamic data is data that is obtained from the fixed size memory pools containing static data. These static pools are allocated once at the start. A memory pool is an area of memory allocated for the exclusive use of a particular type of structure.

Most of the memory allocation occurs in Mdt, McdSpace and McdFrame. The source code supplied with Karma allows the user to alter the way state is allocated and managed if required.

In the following examples, the variables that determine the amount of memory used are:

- Maximum Number of objects, N_{obj}.
- Maximum Number of constraints, N_{const}.
- Maximum Number of degrees of freedom constrained, N_{dof}.

The table below shows some example values for these variables

Model	N_{obj}	N_{const}	N_{dof}
Box impacting on the ground	1	4	12
A car (with suspension and wheel friction)	1	4	12
Complex car (with suspension, wheel friction and wheel rotation)	5 [chassis and four wheels]	4	32
Skeleton in mid air	11	10	30
Skeleton impacting on frictionless ground	11	21	41

Example - A Typical Car Simulation

A single car simulation requires approximately 40k of memory. As more cars are MdtWorldUpdatePartitions takes the bodies and constraints and partitions them so that they can be simulated as separate groups that are independant of the other groups. MdtPartitionOutput stores the partitions, simulating separate cars which requires a lot less memory and is significantly faster than the simulation of both cars when non-partitioned.

Re-Routing Memory Management In Karma

Karmas default memory functions, that wrap malloc by default, can be changed if required. The user must change the contents of the MeMemoryAPI struct before calling any Karma functions, then the users new functions will be called instead. For example,

```
MeMemoryAPI.create = MyCreateFunc;
myWorld = MdtWorldCreate(100,100);
```

will call MyCreateFunc to allocate memory for its pools.

MeGlobals contains the MePool and MeChunk functions that build on top of MeMemory, so the user can get control of memory allocation higher up if required.

Setting Karma Memory Usage

When applying Karma to specific user applications, there are several ways that the user can set how and where Karma uses its memory. The default implementation of Karma uses fixed size memory pools for both bodies and constraints. This is a good starting point, though this may not be suitable for all user applications. One of the reasons for supplying Karma source code is so that users can change it to cater for their specific applications.

The memory allocation used for the MdtBody and MdtConstraint pools is set in the MePool utility located in the MeGlobals library. The user can change this to connect it with a call to their own memory allocator, instead of using the default fixed size pool.

Other memory is allocated in MdtWorldCreate that is based on the maximum number of bodies and constraints that are ever going to exist. These include the MdtPartitionOutput struct, the MdtKeaConstraints struct, and the body arrays passed to Kea. All of these structs are temporary storage used during MdtWorldStep. Therefore, the user can change the Mdt source to allocate / resize them to the requisite size each time-step.

The only part of Mst the user might wish to change is the MstMaterialTable. This contains an element for every pair of materials. Each element contains the contact parameters for the pair of materials and the callbacks for manually tweaking contact parameters. However, this is quite complicated to change. A simpler solution would be to use one material with the existing material system and write contact-parameter setting code instead of a per-pair or per-contact callback.

Memory Usage In Kea

MeChunk is a utility in MeGlobals for feeding Kea its workspace memory. The default size of MeChunk is 1024 bytes. When a system of constraints is passed to Kea to solve, a certain amount of memory is needed to build the matrix. The utility function `MdtKeaMemoryRequired` returns how much memory will be required from MeChunk. MeChunk increases to the required size each time if this is too small. It does so by doubling the memory allocated, hence it is not required to change size very often because this slows down the frame rate. If the user is experiencing problems because the frame rate drops in the first few frames due to MeChuck resizing, `MeChunkSetAuto` can be used to increase the initial value of MeChunk. The memory used by Kea is only used during the solve to hold temporary data. Because a matrix is being formed, the amount of memory required is related to the square of the number of constraint rows in a partition.

Matrix Capping

Matrix capping allows the user to bound an upper limit on the amount of memory (and to some extent the time) used by Karma.

If the dynamics matrix is small enough, the PlayStation®2 version of Kea can manipulate the matrix from VU0MEM in microcode. As the matrix is symmetric, only half of it needs to be stored. As a result the maximum matrix size that can entirely fit into VU0MEM is 36x36. Limiting matrices to this size has considerable performance advantages. Kea can cope with larger matrix sizes, but needs to use a "matrix blocking" strategy to buffer data in and out of the VU0 memory. When the size is less than 36x36, the whole problem fits in the memory at once, and it is possible to compute the solution very rapidly.

However, there is a trade-off. Matrix capping works by heuristically eliminating constraint rows. This means that in some cases, noticeably "incorrect" behavior can result.

This can sometimes give the impression of unstable behavior in extreme cases, but the behavior is not due to mathematical instability. For example, in Chair.elf if a wrecking ball is thrown at the chairs when matrix capping is enabled small 'explosions' could occur. If the user experiences difficulties, matrix capping can be disabled. To do this, do not set the maximum matrix size (MdtWorldSetMaxMatrixSize).

If matrix capping is turned off, no unphysical behavior will occur, but the program will run more slowly.

Per Constraint Memory Used by Kea

Different types of constraints require different amounts of memory but typically the number of each type of constraint required in a scene is not known at initialization. However, it is desirable to allocate memory at initialization rather than in the game loop. With this in mind, we have chosen the following compromise. At initialization time, the user specifies the maximum number of constraints required in total. The memory allocated is the memory required by Karma's largest constraint multiplied by the maximum number of constraints. This memory is allocated in `MdtWorldCreate`. A consequence of this is that if the user has a scene containing one hundred frictionless contacts, and a scene containing one hundred ball and socket joints, the amount of memory requided by Mdt will be the same.

Appendix D: Constraints Reference

Common Constraint Functions

Contacts and joints each have a dedicated set of functions to manage their properties. In order to describe these functions while avoiding redundancy, an abstract, generic set of functions (that unless specified otherwise is common to all joint structures) is first discussed. The specific type of joint will be identified with the wildcard character * to replace one of the following identifiers:

- `BSJoint`, the Ball and Socket joint.
- `Hinge`, the Hinge Joint.
- `Prismatic`, the Prismatic joint.
- `Universal`, the Universal joint.
- `Angular3`, the Angular3 joint.
- `CarWheel`, the Car Wheel Joint.
- `Linear1`, the Linear1 joint.
- `Linear2`, the Linear2 joint.
- `FixedPath`, the Fixed-Path joint.
- `RPROJoint`, the Relative-Position-Relative-Orientation joint.
- Skeletal, the Skeletal joint limit constraint.
- Spring6, a configurable compliance constraint.
- `Spring`, the Spring joint.
- `ConeLimit`, the Cone-Limit constraint.
- `Contact`, a contact.

A constraint can only be created by using the appropriate `Mdt*Create()` function for that constraint. A joint must be created if an articulated body is needed. First create a `Mdt*ID` variable (called `joint_or_contact` in the descriptions below) that will point to the `Mdt*` structure where all information about that joint will be stored. The function that creates a joint and returns a `Mdt*ID` variable is:

```
Mdt*ID MEAPI Mdt*Create        ( const MdtWorldID world )
```

This function creates a new joint or contact in the world. This should be followed with the

```
Mdt*SetBodies                  ( const Mdt*ID joint_or_contact,
                                 const MdtBodyID body0,
                                 const MdtBodyID body1 )
```

function to assign bodies to the contact.

The Reset function is common to all joints and contacts, but the default values it resets to are specific to each of them. See the individual constraint descriptions for details.

```
void MEAPI Mdt*Reset           ( const Mdt*ID joint_or_contact )
```

Set `joint_or_contact` to its default value. Note that the bodies attached to the `joint_or_contact` will have their parameters reset too.

When a joint or a contact is no longer needed, remove it using the following function:

```
void MEAPI Mdt*Destroy         ( const MdtBSJointID joint_or_contact )
```

This function destroys the joint or contact named `joint_or_contact.`

When a constraint is created it needs to be *enabled* to be processed by the system. Conversely, disable a constraint that is not required.

```
void MEAPI Mdt*Enable          ( const MdtConstraintID joint_or_contact )
```

Enable the simulation of `joint_or_contact`.

```
void MEAPI Mdt*Disable         ( const MdtConstraintID joint_or_contact )
```

Disabling a constraint or joint stops it being simulated.

```
MeBool MEAPI Mdt*IsEnabled     ( const MdtConstraintID joint_or_contact )
```

`Returns TRUE if the constraint is enadled`.

A function to obtain a Joint or Contact ID from a Constraint ID: i.e. the converse of `Mdt*QuaConstraint.`

```
    Mdt*ID MEAPI MdtConstraintDCast*( const MdtConstraintID cstrt )
```

Returns an `Mdt*ID` from an MdtConstraintID. If this constraint is not of the expected * type, 0 is returned

Common Accessors

The `Mdt*GetPosition` function that accesses the position of the contact or joint in *world coordinates*, is common to contact and all of the joints except `Prismatic` (does not exist) and `Spring` (an additional argument is needed):

```
    void MEAPI Mdt*GetPosition      ( const Mdt*ID joint_or_contact,
                                      MeVector3 posVector )
```

The position vector of `joint_or_contact` is returned in `posVector`. The undocumented constraint accessor function MdtConstraintGetPosition(MdtConstraintID constraint, MeVector3 position) should not be used. This function does not exist for Angular3. Excludes FixedPath, FPFO and RPRO. The following function is used for these joints

```
    void MEAPI Mdt*GetPosition      ( const Mdt*ID joint_or_contact,
                                      const unsigned int bodyindex,
                                      MeVector3 posVector )
```

The position vector of `joint_or_contact` is returned in `posVector` for the FixedPath, FPFO and RPRO joints. The undocumented constraint accessor function `MdtConstraintGetPosition (MdtConstraintID constraint, MeVector3 position)` should not be used.

```
    MdtBodyID MEAPI Mdt*GetBody     ( const Mdt*ID joint_or_contact,
                                      unsigned int bodyindex )
```

Return one of the bodies connected to this `joint_or_contact`. The value of `bodyindex` is 0 for the first body, 1 for the second body.

```
    void MEAPI Mdt*GetForce         ( const Mdt*ID joint_or_contact,
                                      unsigned int bodyindex,
                                      MeVector3 force )
```

Return the force applied to a body identified by bodyindex by `joint_or_contact` (on the last timestep). Forces are returned in the world reference frame.

```
    void MEAPI Mdt*GetTorque        ( const Mdt*ID joint_or_contact,
                                      unsigned int bodyindex,
                                      MeVector3 torque )
```

Return the torque applied to a body by `joint_or_contact` (on the last timestep). The torque is returned in the world reference frame in `torque`.

```
    MdtWorldID MEAPI Mdt*GetWorld   ( const Mdt*ID joint_or_contact )
```

Return the world that `joint_or_contact` is in.

```
    void *MEAPI Mdt*GetUserData     ( const Mdt*ID joint_or_contact )
```

Return the user-defined data of `joint_or_contact.`

```
    MeI32 MEAPI Mdt*GetSortKey      ( const Mdt*ID joint_or_contact )
```

`Return the sort key of` *joint_or_contact*`.`

Common Mutators

The `Mdt*SetPosition` function that mutates the position of the contact or the joint in *world coordinates,* is common to contact and all of the joints except `Prismatic` (does not exist) and `Spring` (an additional argument is required).

```
void MEAPI Mdt*SetPosition      ( const Mdt*ID joint_or_contact,
                                  const MeReal xPos,
                                  const MeReal yPos,
                                  const MeReal zPos )
```

Set the joint_or_contact position in world coordinates at (`xPos`, `yPos`, `zPos`). The undocumented constraint mutator function `MdtConstraintSetPosition (MdtConstraintID constraint, MeReal x, MeReal y, MeReal z)` should not be used. This function does not exist for Angular3.

```
void MEAPI Mdt*SetBodies        ( const Mdt*ID joint_or_contact,
                                  const MdtBodyID body0,
                                  const MdtBodyID body1 )
```

Attach `body0` and `body1` to `joint_or_contact`.

```
void MEAPI Mdt*SetUserData      ( Mdt*ID joint_or_contact,
                                  void *data )
```

Set the `joint_or_contact` user data.

```
void MEAPI Mdt*SetSortKey       ( const Mdt*ID joint_or_contact,
                                  MeI32 key )
```

Assign a sort key to `joint_or_contact`.

.

NOTE: The similarity of the constraint functions used by joints and contacts arises because most of the *joints and contact* functions are macros that are *defined* through the *constraint* function. Karma header files for the joints and contacts give a complete function listing.

Base Constraint Functions

The `MdtConstraint` functions are a set of functions that apply to all constraints. The base constraint data consists of one attachment point per rigid body that is, a position and an orientation. This is the position of the joint as seen from each rigid body.

These base constraint functions have been used in implementing many of the individual constraint functions and, in some cases, offer an alternative to the individual functions. However, the effects of the base constraint functions are not well defined or documented and the individual constraint functions should always be used in preference.

When a joint or contact constraint is created, the functions can be accessed by converting the contact or joint ID to a `MdtConstraintID` variable using the following function:

```
MdtConstraintID Mdt*QuaConstraint         ( const Mdt*ID joint_constraint )
```

This function is used to convert a specific joint identifier to its abstract representation of a constraint identifier `MdtConstraintID`.

A constraint is destroyed by using

```
void MEAPI MdtConstraintDestroy           ( const MdtConstraintID constraint )
```

Destroys a constraint. The constraint is disabled automatically if necessary.

:The `MdtConstraint*` functions include a function to enable and a function to disable a constraint. The difference between the *destroy* function and the *disable* function is that the *disable* keeps the constraint structure in memory for later use. Destroying a constraint removes it from memory so that the it cannot be used at a later time.

```
void MEAPI MdtConstraintEnable            ( const MdtConstraintID constraint )
```

Enables simulation of a constraint.

```
void MEAPI MdtConstraintDisable           ( const MdtConstraintID constraint )
```

Disables simulation of a constraint.

```
MeBool MEAPI MdtConstraintIsEnabled       ( const MdtConstraintID constraint )
```

Determine if a constraint is currently enabled. Returns 1 if enabled, or 0 if not.

The Constraints Mutator Functions

To attach bodies to a constraint use:

```
void MEAPI MdtConstraintSetBodies         ( const MdtConstraintID constraint,
                                            const MdtBodyID body0,
                                            const MdtBodyID body1 )
```

Set the bodies `body0` and `body1` to be attached to the constraint

Note that a constraint must be disabled before changing the bodies attached to it. The constraint library provides a number of Set/Get functions to mutate and access the variables of a constraint structure. Please consult the Karma Dynamics Reference Manual.

```
void MEAPI MdtConstraintSetAxis           ( const MdtConstraintID constraint,
                                            const MeReal px,
                                            const MeReal py,
                                            const MeReal pz )
```

Set the constraint primary axis in the world reference frame.

```
void MEAPI MdtConstraintSetAxes           ( const MdtConstraintID constraint,
                                            const MeReal px,
                                            const MeReal py,
                                            const MeReal pz,
                                            const MeReal ox,
                                            const MeReal oy,
                                            const MeReal oz )
```

Set the primary and secondary constraint axes in the world reference frame.

The axes will be normalized automatically.

The axes must be orthogonal.

This effectively sets the rotational orientation of a constraint frame consisting of the two given axes and a third orthogonal axis corresponding to the cross product of the given axes.

An older, deprecated name for this function is `MdtConstraintSetBothAxis` (sic)

```
void MEAPI MdtConstraintBodySetPosition
                                          ( const MdtConstraintID constraint,
                                            const unsigned int bodyindex,
                                            const MeReal x,
                                            const MeReal y,
                                            const MeReal z )
```

Set the constraint position for the given body in the world reference frame.

```
void MEAPI MdtConstraintSetUserData       ( const MdtConstraintID constraint,
                                            void *data )
```

Set the constraint userdata.

```
void MEAPI MdtConstraintSetSortKey        ( const MdtConstraintID constraint,
                                            MeI32 key );
```

Set the constraint sort key.

The Constraint Accessor Functions

Most (but not all) mutator functions are paired to equivalent accessor functions:

```
MdtBodyID MEAPI MdtConstraintGetBody      ( const MdtConstraintID constraint,
                                            const unsigned int bodyindex )
```

Return the body connected to this constraint as determined by `bodyindex`. The value of `bodyindex` is 0 for the first body, 1 for the second body.

```
void MEAPI MdtConstraintGetAxis           ( const MdtConstraintID constraint,
                                            MeVector3 axis )
```

Get the constraint primary axis in the world reference frame and store its value in axis.

```
void *MEAPI MdtConstraintGetUserData      ( const MdtConstraintID constraint );
```

Return the user-defined data of this constraint.

```
void MEAPI MdtConstraintBodyGetAxes       ( const MdtConstraintID constraint,
                                            const unsigned int bodyindex,
                                            MeVector3 primary,
                                            MeVector3 ortho )
```

Get both the primary constraint axis and the orthogonal secondary constraint axis in the world reference frame for the given body.

An older, deprecated name for this function is `MdtConstraintBodyGetBothAxes`.

```
void MEAPI MdtConstraintGetAxes           ( const MdtConstraintID constraint,
                                            MeVector3 primary,
                                            MeVector3 ortho )
```

Get both the primary constraint axis and the orthogonal secondary constraint axis in the world reference frame.

An older, deprecated name for this function is `MdtConstraintGetBothAxes`.

To get the value of the force and torque applied to a given body by a constraint use:

```
void MEAPI MdtConstraintGetForce          ( const MdtConstraintID constraint,
                                            const unsigned int bodyindex,
                                            MeVector3 force )
```

Return the force applied to the body (identified by `bodyindex`) by this constraint on the last timestep. Forces are returned in the world reference frame.

```
void MEAPI MdtConstraintGetTorque         ( const MdtConstraintID constraint,
                                            const unsigned int bodyindex,
                                            MeVector3 torque );
```

Return the torque applied to the body (identified by `bodyindex`) by this constraint on the last timestep. Torque is returned in the world reference frame.

To determine which world a constraint belongs to use:

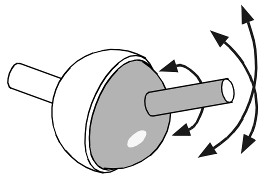
```
MdtWorldID MEAPI MdtConstraintGetWorld    ( const MdtConstraintID constraint )
```

Return the world that the constraint is in.

```
MeI32 MEAPI MdtConstraintGetSortKey       ( const MdtConstraintID constraint )
```

Return the constraint sort key.

Ball-and-socket (BS) Joint: MdtBSJoint



A ball and socket joint forces a point fixed in one bodies reference frame to be at the same location in the world reference frame as that of a point fixed in another bodies reference frame. This removes three (linear) degrees of freedom. In the diagram above, the center of the sphere always coincides with the center of the socket. This ideal joint allows all rotations about the common point. Real ball and socket joints have joint limits because a body attached to the ball will collide with the sides of the socket. The MdtBSJoint does not have limits built in but the MdtConeLimit constraint can be used with it to provide limits. This joint is sometimes referred to as a spherical joint.

A ball and socket joint, in conjuction with a cone limit, may be used to model a shoulder joint, or to connect links in a chain.

Ball-and Socket Joint Functions

There are no functions specific to `MdtBSJoint`. The reset function sets the joint position in each bodies' reference frame to {0, 0, 0}. The joint position can be set to change this default.

```
    MdtBSJointID bs = MdtBSJointCreate(world);
```

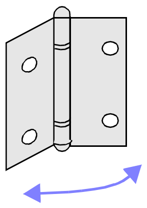
creates a ball and socket joint in an `MdtWorld world`. To use the joint to constrain a pair of objects (`body1` and `body2`) use;

```
    MdtBSJointSetBodies(bs, body1, body2);
    MdtBSJointSetPosition(bs, pos_x, pos_y, pos_z);
```

This sets the position of the joint in the world frame. The fixed positions of the joint relative to each body are then initialized. Alternatively the joint positions can be set individually for each body using the base constraint interface.

The bodies should already have been created and positioned in their requisite initial positions before attaching them to the joint.

Hinge Joint: MdtHinge



A hinge constrains a pair of bodies to rotate freely about a specific hinge axis. The remaining five degrees of freedom between the joined bodies are fixed. Because of this the hinge is more computationally costly than the previously discussed ball and socket joint. The hinge axis has fixed positions and orientations in each rigid body, and the hinge constraint forces those axes to coincide at all times. A hinge is sometimes referred to as a revolute joint.

A hinge joint could be used to attach a door to a doorframe, a lever, drawbridge or seesaw to its fulcrum, or to attach rotating parts such as a wheel to a chassis, a propeller shaft to a ship or a turntable to a deck.

Hinge Limits

Up to two stops, or limits, can be set to restrict the relative rotation of the bodies attached by a hinge joint. These limits may be specified independently to be either *hard* (if the limit stiffness factor is high) or *soft*. In a Karma Dynamics simulation, a hard bounce reverses the bodies' angular velocities in a single timestep, while a soft bounce may take many timesteps to reverse the angular velocity.

If the limits are soft, damping can be set so that, beyond the limits, the hinge behaves like a damped spring.

If the limits are hard, the limit restitution can be set to a value between zero and one to govern the loss of angular momentum as the bodies rebound.

Hinge joint limits range from -nπ through nπ for real number n hence multiple rotations are supported and a hinge passing a limit will always be detected and the correct response simulated.

Hinge Actuators

A hinge joint can be actuated (powered). This simulates a motor acting on the hinge's remaining degree of freedom, the hinge angle. To characterize a hinge motor, set a desired angular speed and the motor's maximum torque. The motor is assumed to be symmetric, so that the maximum torque can be applied in either direction. A torque no greater than this is applied to the hinged bodies to change their relative angular velocity, until either the desired velocity is achieved, or the hinge angle hits a limit (if set).

The response of an actuated hinge hitting a limit depends on the stiffness and restitution or damping properties that have been chosen for the relevant limit, but in general the hinge will (quickly or slowly) come to rest at the set limit. If a soft limit has been specified, the rest position will be beyond the limit by an angle determined by the motor's maximum torque and the limit stiffness factor.

Whenever a hinge is actuated, or is at (or beyond) one of its limits, the computational cost is equivalent to constraining six degrees of freedom.

Hinge Joint Functions

A Hinge joint is described by the position and direction of its axis. The reset function zeros the position in each body's reference frame, and sets the axis direction to each body's x-axis, i.e., position = {0,0,0}, axis={1,0,0}.

Accessors

The accessor functions specific to the `hinge` are:

```
MdtLimitID MEAPI MdtHingeGetLimit         ( const MdtHingeID joint );
```

Provide read and write access with the `MdtLimit` functions to the constraint limits parameters of the `joint` by providing the corresponding `MdtLimitID` identifier.

```
void MEAPI MdtHingeGetAxis                ( const MdtHingeID joint,
                                            MeVector3 axisVec );
```

The Hinge joint axis is returned in the vector `axisVec`.

Mutators

The mutator functions specific to the `hinge` are:

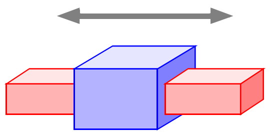
```
void MEAPI MdtHingeSetLimit               ( const MdtHingeID joint,
                                            const MdtLimitID NewLimit );
```

Reset the joint limit and then copy the public attributes of `NewLimit.`

```
void MEAPI MdtHingeSetAxis                ( const MdtHingeID joint,
                                            const MeReal xAxis,
                                            const MeReal yAxis,
                                            const MeReal zAxis );
```

Set the hinge axis of `joint` to `(xAxis, yAxis, zAxis)`.

Prismatic: MdtPrismatic



The prismatic, or slider, joint is like the hinge in that two axes, one fixed in each of the two constrained bodies reference frames, are forced to coincide. In the prismatic however, the 2 bodies move along the axis, not around it. Like a hinge, a prismatic joint removes five degrees of freedom from the relative motion of the attached bodies, leaving one linear degree of freedom. The relative orientation of the bodies are maintained by the joint. The prismatic can be imagined as a bar sliding inside a block with a hole in it, where the area of the hole matches the cross sectional area of the bar.

Prismatic Limits

Two limits may be set to restrict the linear motion of a prismatic joint. These limits may be either hard or soft, with the ability to set the stiffness, restitution and damping properties independently for each limit.

Prismatic Actuators

Speed and maximum force may be set to actuate the movement of a prismatic joint. The actuation force will be applied to slow down or speed up the attached bodies until their relative velocity reaches the specified speed, unless a limit is reached first.

Whenever a prismatic joint is actuated, or is at (or beyond) one of its limits, the computational cost is equivalent to constraining six degrees of freedom rather than five.

Prismatic Joint Functions

A Prismatic joint is described by the direction of its sliding axis. The reset function sets this to the bodies' x-axis, i.e. {1,0,0}.

When the constrained bodies have been initialized and set to their starting positions, all that is required to initialize the Prismatic joint is to set the direction of its sliding axis.

The initial position of the bodies specifies the zero displacement of the sliding degree of freedom used for the Prismatic limits. This is set automatically when the axis is set.

Accessors

Here are the accessor functions specific to `Prismatic`:

```
MdtLimitID MEAPI MdtPrismaticGetLimit     ( const MdtPrismaticID joint )
```

Provides read/write access to the constraint limits of `joint`.

```
void MEAPI MdtPrismaticGetAxis            ( const MdtPrismaticID joint,
                                            MeVector3 axisVec )
```

The prismatic joint axis is returned in the vector `axisVec`.

Mutators

Here are the mutator functions specific to `Prismatic`:

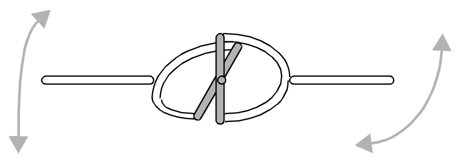
```
void MEAPI MdtPrismaticSetLimit           ( const MdtPrismaticID joint,
                                            const MdtLimitID NewLimit )
```

Reset the joint limit and then copy the public attributes of `NewLimit`.

```
void MEAPI MdtPrismaticSetAxis           ( const MdtPrismaticID joint,
                                            const MeReal xAxis,
                                            const MeReal yAxis,
                                            const MeReal zAxis )
```

Set the prismatic axis of `joint` to `(xAxis, yAxis, zAxis).`

Universal Joint: MdtUniversal



In this joint, two axes, one fixed in each of the two constrained bodies, are forced to have a common origin and to be perpendicular at all times. This is a lot like the ball and socket joint but here the ball is not allowed to twist in its socket.

A universal joint removes four degrees of freedom from the attached bodies. It fixes their relative position and constrains them not to twist about a third axis, perpendicular to the two given axes. This joint may be pictured as a joystick mechanism in which two hinges are joined, one on top of the other with perpendicular axes, to allow an attached stick to move first in the x-direction then in the y-direction.

This mechanism is also known as gimbal, and the mechanism suffers from dreaded ‘gimbal-lock’ at 90°. This relates to its use as an ‘engineering’ universal joint that can be used to transmit torque from one body to another around a small bend. The transmission becomes increasingly unsmooth as the angle of bend approaches 90° and finally cannot transmit torque at all.

In Karma, the singularity at 90° can be avoided by applying a Cone-Limit constraint in parallel with the Universal.

Note that a smoother type of universal joint with built in limits is implemented in the Skeletal joint.

Universal Joint Functions

A Universal joint is described by the position of the joint and the directions of the axes fixed in each body. The reset function zeros the position in each body frame and defaults the axes to the x-axis {1,0,0} in body1 and the y-axis {0,1,0} in body2.

Accessors

The accessor function specific to `Universal is`:

```
void MEAPI MdtUniversalGetAxes            ( const MdtUniversalID joint,
                                            const unsigned int bodyindex,
                                            MeVector3 primary_axis,
                                            MeVector3 orthogonal_axis )
```

The joint axes corresponding to `bodyindex` are returned in the vectors `primary_axis`, `orthogonal_axis`.

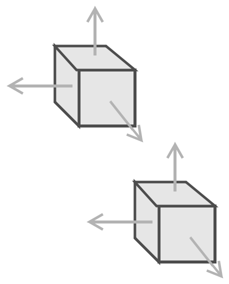
Mutators

The mutator function specific to `Universal` is:

```
void MEAPI MdtUniversalSetAxes             ( const MdtUniversalID joint,
                                             const unsigned int bodyindex,
                                             const MeReal px,
                                             const MeReal py,
                                             const MeReal pz )
                                             const MeReal ox,
                                             const MeReal oy,
                                             const MeReal oz )
```

Set the joint axes corresponding to `bodyindex`, to (`px,py,pz`) and (ox,oy,oz) in world coordinates. The given axes must be orthonormal.

Angular Joint: MdtAngular3 & MdtAngular2



The relative orientation of these two bodies is fixed but their relative position can vary freely

The Angular3 joint removes three rotational degrees of freedom by constraining one body to have a fixed orientation with respect to another body. While one body can move freely in space (irrespective of the other body's location) its orientation is fixed relative to the other body's orientation. It is possible to add one rotational degree of freedom about a specified axis, enabling a rotation of a body with respect to the other, effectively modifying the Angular3 joint to an *Angular2* joint.

Angular3 and angular2 joints are useful for keeping things upright, such as game vehicles that should not overturn. Springing and damping can be set to introduce some softness around the upright.

Angular3 Joint Functions

The default for the Angular3 is to align the two bodies' reference frames. The reset function also has this effect. The joint is initialized by the call to **`MdtAngular3SetBodies`** that notes the bodies initial orientations for use in maintaining the same fixed relative orientation.

Accessors

The accessor functions specific to `Angular3` are:

```
MeBool MEAPI MdtAngular3RotationIsEnabled      ( const MdtAngular3ID joint );
```

Return the current state of the `bEnableRotation` flag. If this flag is set (true), this constraint is effectively an `Angular2` joint.

```
void MEAPI MdtAngular3GetAxis                  ( const MdtAngular3ID joint,
                                                 MeVector3 axis )
```

The `Angular3` joint axis vector is returned in `axis`. Rotation is allowed about this axis if the `bEnableRotation` flag is set.

```
MeReal MEAPI MdtAngular3GetStiffness           ( const MdtAngular3ID j )
```

Return current 'stiffness' of this angular constraint.

```
MeReal MEAPI MdtAngular3GetDamping             ( const MdtAngular3ID j )
```

Return current spring 'damping' of this angular constraint.

Mutators

The mutator functions specific to `Angular3` are:

```
void MEAPI MdtAngular3EnableRotation      ( const MdtAngular3ID joint,
                                            const MeBool NewRotationState )
```

Set or clear the joint `bEnableRotation` flag to `NewRotationState`. If this flag is set (true), this constraint is effectively an Angular2 joint enabling the rotation of one body relative to another.

```
void MEAPI MdtAngular3SetAxis              ( MdtAngular3ID joint,
                                             MeReal xAxis,
                                             MeReal yAxis,
                                             MeReal zAxis )
```

Set the joint axis at `(xAxis, yAxis, zAxis)` in world coordinates. Note that this axis is used only if the `bEnableRotation` flag is set.

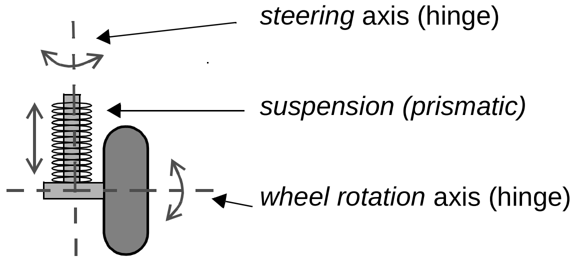
```
void MEAPI MdtAngular3SetStiffness         ( const MdtAngular3ID j,
                                             const MeReal s )
```

Set the angular constraint stiffness about the enabled axis. The default is MEINFINITY.

```
void MEAPI MdtAngular3SetDamping           ( const MdtAngular3ID j,
                                             const MeReal d )
```

Set the angular constraint damping about the enabled axis. The default is MEINFINITY.

CarWheel Joint: MdtCarWheel



The CarWheel joint models the behavior of a car wheel with optional steering and suspension. The CarWheel joint is a combination of two hinge joints, one for the steering and one for the rotation of the wheel, and one prismatic joint for telescopic suspension with built in springing.

Body 1 is the chassis and body 2 is the wheel. The connection point for the wheel body is its center of mass.

CarWheel Joint Functions

A CarWheel joint is defined by a steering axis and a hinge axis. The steering axis also acts as the suspension axis. It has a direction fixed in the chassis frame and passes through the origin of the wheel's reference frame. The hinge axis has a direction fixed in the wheel frame and also passes through the frame origin. The constraint keeps these two axes perpendicular.

The default steering axis is the body1 z-axis and the default hinge axis is the body2 y-axis.

Accessors

The accessor functions specific to the `CarWheel joint are:`

```
MeReal MEAPI MdtCarWheelGetHingeAngle( const MdtCarWheelID joint );
```

Return the wheel joint current hinge angle as a value between zero and 2*PI radians, inclusive.

```
MeReal MEAPI MdtCarWheelGetHingeAngleRate( const MdtCarWheelID joint );
```

Return the wheel joint angular velocity about the hinge axis.

```
void MEAPI MdtCarWheelGetHingeAxis( const MdtCarWheelID joint,
MeVector3 hingeAxis );
```

The wheel joint hinge axis is returned in `hingeAxis`.

```
MeReal MEAPI MdtCarWheelGetHingeMotorDesiredVelocity(
                                          const MdtCarWheelID joint );
```

Return the desired velocity of the hinge motor.

```
MeReal MEAPI MdtCarWheelGetHingeMotorMaxForce( const MdtCarWheelID joint
);
```

Return the maximum force that the hinge motor is allowed to use to attain its desired velocity.

```
MeReal MEAPI MdtCarWheelGetSteeringAngle( const MdtCarWheelID joint );
```

Return the wheel joint steering angle.

```
MeReal MEAPI MdtCarWheelGetSteeringAngleRate( const MdtCarWheelID joint );
```

Return the wheel joint angular velocity about the steering axis.

```
void MEAPI MdtCarWheelGetSteeringAxis( const MdtCarWheelID joint,
                                 MeVector3 steeringAxis );
```

The wheel joint steering axis is returned in `steeringAxis`.

```
MeReal MEAPI MdtCarWheelGetSteeringMotorDesiredVelocity(
                                            const MdtCarWheelID joint)
```

Return the desired velocity of the steering motor.

```
MeReal MEAPI MdtCarWheelGetSteeringMotorMaxForce (
                                            const MdtCarWheelID joint );
```

Return the maximum force that the steering motor is allowed to use to attain its desired velocity.

```
MeReal MEAPI MdtCarWheelGetSuspensionHeight( const MdtCarWheelID joint );
```

Return the wheel joint suspension height.

```
MeReal MEAPI MdtCarWheelGetSuspensionHighLimit( const MdtCarWheelID joint
);
```

Return the suspension upper limit.

```
MeReal MEAPI MdtCarWheelGetSuspensionKd ( const MdtCarWheelID joint );
```

Return the suspension "damping constant" (also known as the "derivative constant"). This gives rise to the damping term K_d in the suspension force equation: F = $-k_p$*displacement + k_d*velocity, where K_p is Hookes Law constant and K_d is the damping constant.

```
MeReal MEAPI MdtCarWheelGetSuspensionKp( const MdtCarWheelID joint );
```

Return the suspension "proportionality constant". This gives rise to the spring term k_p in the suspension force equation: F = $-k_p$*displacement + k_d*velocity, where K_p is Hookes Law constant and K_d is the damping constant

```
MeReal MEAPI MdtCarWheelGetSuspensionLimitSoftness(
                                            const MdtCarWheelID joint );
```

Return the suspension limit softness.

```
MeReal MEAPI MdtCarWheelGetSuspensionLowLimit( const MdtCarWheelID joint
);
```

Return the suspension lower limit.

```
MeReal MEAPI MdtCarWheelGetSuspensionReference( const MdtCarWheelID joint
);
```

Return the suspension attachment point (*reference*).

```
MeBool MEAPI MdtCarWheelIsSteeringLocked( const MdtCarWheelID joint );
```

Return the lock state of the steering angle. (lock is 1 if steering axis is locked at angle 0).

Mutators

The mutator functions specific to the `CarWheel joint are:`

```
void MEAPI MdtCarWheelSetSteeringAndHingeAxes( const MdtCarWheelID joint,
              const MeReal xSteer, const MeReal ySteer, const MeReal zSteer,
              const MeReal xHinge, const MeReal yHinge, const MeReal zHinge
);
```

Set the wheel joint hinge axis at (`xHinge, yHinge, zHinge`).

```
void MEAPI MdtCarWheelSetHingeLimitedForceMotor( const MdtCarWheelID
joint,
                        const MeReal desiredVelocity, const MeReal forceLimit
);
```

Set the hinge limited force motor parameters.

```
void MEAPI MdtCarWheelSetSteeringLimitedForceMotor(
                                const MdtCarWheelID joint,
                    const MeReal desiredVelocity, const MeReal forceLimit );
```

Set the limited force motor parameters of a car wheel joint.

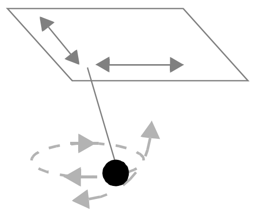
```
void MEAPI MdtCarWheelSetSteeringLock( const MdtCarWheelID joint,
                                                const MeBool lock );
```

Lock or unlock the steering angle (lock is 1 if steering axis is locked at angle 0).

```
void MEAPI MdtCarWheelSetSuspension( const MdtCarWheelID joint,
                  const MeReal Kp, const MeReal Kd,
                  const MeReal limit_softness, const MeReal lolimit,
                  const MeReal hilimit, const MeReal reference );
```

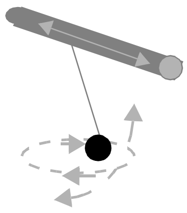
Set the suspension parameters.

Linear1 Joint: MdtLinear1



A Linear1 joint removes one degree of freedom by confining a point fixed in one of the attached bodies to a plane fixed in the other body.

Linear2 Joint: MdtLinear2



A Linear 2 joint removes two degrees of freedom by confining a point fixed in one of the attached bodies to a line fixed in the other body. This can be used to simulate a continuous sliding contact between an object and a line, such as a rail or pole.

Functions Specific to Linear2 Joint

Accessors

The accessor function specific to `Linear2 is`:

```
void MEAPI MdtLinear2GetDirection ( const MdtLinear2ID contact,
MeVector3 directVec );
```

The `Linear2` joint primary direction is returned in `directVec` in the world reference frame.

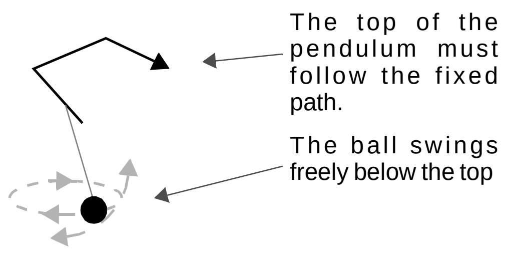
Mutators

The mutator function specific to `Linear2 is:`

```
void MEAPI MdtLinear2SetDirection ( const MdtLinear2ID contact,
const MeReal xDir, const MeReal yDir, const MeReal zDir );
```

Set the joint direction `(xDir, yDir, zDir)` in world coordinates. x, y and z should be `const`.

Fixed-Path Joint: MdtFixedPath


The top of the pendulum must follow the fixed path.
The ball swings freely below the top

The Fixed-Path joint is a Ball-and-Socket joint modified to allow motion of the joint attachment point. To enable this to be done correctly both position and velocity data for the moving joint attachment point are required as it moves along a given path. While it is possible to move the position of a Ball-and-Socket directly, this does not feed the correct forces into the attached bodies and relies on numerical relaxation to satisfy the constraint.

This joint can be used to attach an animated path to the simulation, in such a way that the forces generated by any animated motion will be transmitted correctly to the simulated, non-animated, objects. For example, a Fixed Path joint could be used to move the attachment point of a pendulum kinematically while the pendulum swings in response to the motion, as sketched above. The animation must supply the position of the fixed path joint at each timestep. The joint can feed back the forces and torques resulting from the pull of gravity and any contact with other simulated bodies.

A Fixed Path joint fixes the relative position of the two attached bodies, removing three degrees of freedom, while leaving them free to rotate freely with respect to one another.

Functions Specific to Fixed-Path Joint

Accessors

The accessor function specific to `FixedPath is:`

```
void MEAPI MdtFixedPathGetVelocity( const MdtFixedPathID joint,
                    const unsigned int bodyindex, MeVector3 velocity );
```

The fixed path joint velocity with respect to one of the constrained bodies is returned in `velocity`. The reference frame is determined by the third parameter, *`bodyindex`* .

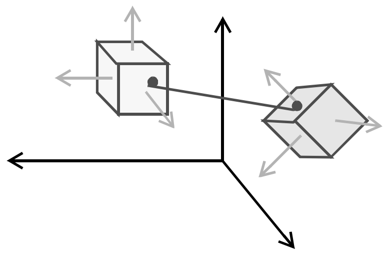
Mutators

The mutator function specific to `FixedPath is:`

```
void MEAPI MdtFixedPathSetVelocity( const MdtFixedPathID joint,
          const unsigned int bodyIndex,
          const MeReal xVel, const MeReal yVel, const MeReal
zVel);
```

Set the fixed path joint velocity with respect to one of the constrained bodies. This joint velocity is set in the bodies' reference frames. The reference frame is determined by the fifth parameter, `bodyIndex`.

Relative-Position-Relative-Orientation Joint: MdtRPROJoint



The RPRO joint is a feature added for use in ‘playing back’ an animation through an object, or in controlling the motion of an object via user interaction, while allowing for physical response to collisions and fast motions. Note that the current implementation supports animation of relative orientations only, addressing character motion use-cases where the animated objects are articulated chains connected by ball-and-socket joints. Support for animation of relative positions is scheduled for a future release.

The RPRO joint constrains all six degrees of freedom between the attached bodies, or in other words leaves no freedom to move between the two bodies. The exception is when the force-limit feature allows the joint to break when a specified maximum force is exceeded in a collision or fast motion. However, the relative orientation can be driven by an animation script or a stream of user input supplied as a sequence of relative quaternions and relative angular velocities (when support for relative position is added an input sequence of relative positions and relative linear velocities will be required to drive translations).

By default, the relative motion is specified between the center-of-mass frames of the primary and secondary bodies. It is also possible to specify joint attachment frames that are offset from the center of mass - useful if animation data is supplied in a different body-fixed frame of reference, though this feature does incur a small performance cost. At present, because relative positions are not yet supported, the only way to create an offset between the two body frames is to specify a joint attachment position using `MdtRPROJointSetAttachmentPosition()`.

Angular motion between the two bodies is achieved by updating the relative orientation using `MdtRPROJointSetRelativeQuaternion()` and the relative angular velocity using `MdtRPROJointSetRelativeAngularVelocity()`.

Force limits, or ‘strengths’, of the linear and angular parts of the constraint can be set using `MdtRPROJointSetLinearStrength() and MdtRPROJointSetAngularStrength().` Setting low strengths will cause the constraint to be broken easily by imposed accelerations or external forces. This provides a method by which animations can be played through an object while allowing physical reactions to fast motions or collisions with other objects in the simulation.

As an example of user interaction, the RPRO joint is useful in picking up and reorienting objects in the simulation environment. In a ‘first person’ game an object could be picked up at a fixed point in the player’s perspective and its position maintained in the field of view as the player moves. With linear strengths set to small values the picked object will react physically to collisions and fast motions. The object can be reoriented around its picked position by updating the relative quaternion and relative angular velocity inputs.

As with all joints the RPRO can be used to join a body to the world by specifying one of the bodies as NULL. This could be used to drive anchored robot arms, cranes or other mechanisms fixed in the simulation world frame. Note that, because relative positions are not yet implemented, full vehicle motions cannot be directly driven in this way.

Functions Specific to RPRO Joint

Accessors

The accessor functions specific to `RPRO are`

```
void MEAPI MdtRPROJointGetRelativeQuaternion( const MdtRPROJointID joint,
MeVector4 quaternion );
```

This retrieves the relative quaternion.

```
void MEAPI MdtRPROJointGetAttachmentOrientation( const MdtRPROJointID
joint, const unsigned int bodyindex, MeVector4 quaternion )
```

The full constraint joint attachment orientation with respect to one of the constrained bodies is returned in `quaternion`. The reference frame is selected by the parameter `bodyindex`.

```
void MEAPI MdtRPROJointGetAttachmentPosition( const MdtRPROJointID joint,
const unsigned int bodyindex,
MeVector3 position )
```

The full constraint joint position with respect to one of the constrained bodies is returned in `position`. The reference frame is selected by the second parameter (`bodyindex`).

Mutators

The mutator functions specific to `RPRO are`

```
void MEAPI MdtRPROJointSetRelativeQuaternion( const MdtRPROJointID joint,
const MeVector4 quaternion );
```

Set the relative orientation quaternion.

```
void MEAPI MdtRPROJointSetAttachmentQuaternion( const MdtRPROJointID
joint,
const MeReal q0,
const MeReal q1,
const MeReal q2,
const MeReal q3,
const unsigned int bodyindex );
```

Set the full constraint joint attachment orientation with respect to one of the constrained bodies described by the quaternion `(q0,q1,q2,q3)`.The reference frame is selected by the parameter `bodyindex`.

```
void MEAPI MdtRPROJointSetAttachmentPosition( const MdtRPROJointID joint,
const MeReal x,
const MeReal y,
const MeReal z,
const unsigned int bodyindex );
```

Set the constraint joint position with respect to one of the constrained bodies. The reference frame is selected by the parameter `bodyindex`.

```
void MEAPI MdtRPROJointSetAngularStrength( const MdtRPROJointID joint,
const MeReal sX,
const MeReal sY,
const MeReal sZ );
```

Set the limit on the maximum torque (`sX,sY,sZ`) that can be applied to maintain the constraints. If all values are set at `MEINFINITY`, the constraint will always be maintained. If some finite (positive) limit is set, the constraint will become violated if the force required to maintain it becomes larger than the threshold.

```
void MEAPI MdtRPROJointSetLinearStrength( const MdtRPROJointID joint,
const MeReal sX,
const MeReal sY,
const MeReal sZ );
```

Set the limit on the maximum force (`sX,sY,sZ`) that can be applied to maintain the constraints. If all values are set at `MEINFINITY`, the constraint will always be maintained. If some finite (positive) limit is set, then, the constraint will become violated if the force required to maintain it becomes larger than the threshold.

```
void MEAPI MdtRPROJointSetRelativeAngularVelocity( MdtRPROJointID joint,
MeVector3 velocity );
```

Set the relative angular velocity of the joint.

Skeletal limit constraint: MdtSkeletal

`MdtSkeletal` combines a ball-and-socket joint with a general orientation constraint which puts a limit on its 'swing and twist' freedoms. Thinking of a joystick, swing limits are defined in terms of maximum joystick angles in two orthogonal directions. Equal limit angles define a circular cone. Unequal limit angles define an elliptical cone. Twist is defined to be zero at the central joystick position in its reference alignment and at other positions achieved by a single great-circle rotation from this orientation.

This joint is currently experimental. Actuation options are to be added.

Skeletal Joint Functions

The reset function defaults to using the x-axes of the two body frames and limits the angle between them to PI radians, which is effectively no limit.

Accessors

The accessor functions specific to `the Skeletal joint are`:::

```
MeReal MEAPI MdtSkeletalGetConePrimaryLimitAngle(
                                         const MdtConeLimitID j);
```

Return the primary cone limit angle; i.e. the angle between the cone axis and the side of the cone.

```
MeReal MEAPI MdtSkeletalGetConeSecondaryLimitAngle(
                                         const MdtConeLimitID j);
```

Return the secondary cone limit angle; i.e. the angle between the cone axis and the side of the cone.

```
MeReal MEAPI MdtSkeletalGetTwistLimitAngle(const MdtConeLimitID j);
```

Return thetwist limit angle.

```
MeReal MEAPI MdtSkeletalGetConeStiffness(const MdtConeLimitID j);
```

Return the current cone limit stiffness.

```
MeReal MEAPI MdtSkeletalGetConeDamping(const MdtConeLimitID j);
```

Return the current cone limit damping.

```
MeReal MEAPI MdtSkeletalGetTwistStiffness(const MdtConeLimitID j);
```

Return the current twist limit stiffness.

```
MeReal MEAPI MdtSkeletalGetTwistDamping(const MdtConeLimitID j);
```

Return the current twist limit damping.

Mutators

The mutator functions specific to the Skeletal joint are:

```
void MEAPI MdtSkeletalSetConePrimaryLimitAngle(
                      const MdtSkeletalID j, const MeReal theta);
```

Set the primary limit angle to `theta`. The limit angle is the angle between the cone axis and the side of the cone. Defaults to ME_PI/4 at reset.

```
void MEAPI MdtSkeletalSetConeSecondaryLimitAngle(
                       const MdtSkeletalID j, const MeReal theta);
```

Set the secondary limit angle to `theta`. The limit angle is the angle between the cone axis and the side of the cone. Defaults to ME_PI/4 at reset.

```
void MEAPI MdtSkeletalSetTwistLimitAngle(const MdtSkeletalID j,
                                         const MeReal theta);
```

Set the twist limit angle to `theta`. Defaults to ME_PI/4 at reset.

```
void MEAPI MdtSkeletalSetConeStiffness(const MdtConeLimitID j,
                                       const MeReal kp);
```

Set the cone limit stiffness to k_p.

```
void MEAPI MdtSkeletalSetConeDamping(const MdtConeLimitID j,
                                     const MeReal kd);
```

Set the cone limit damping to k_d.

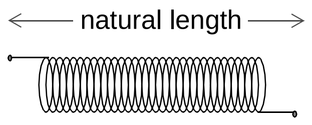
```
void MEAPI MdtSkeletalSetTwistStiffness(const MdtConeLimitID j,
                                        const MeReal kp);
```

Set the twist limit stiffness to k_p.

```
void MEAPI MdtSkeletalSetTwistDamping(const MdtConeLimitID j,
                                      const MeReal kd);
```

Set the twist limit damping to k_d.

Spring Joint: MdtSpring



This joint attaches one body to another, or to the inertial reference frame, at a given separation. The spring joint tends to restore itself to its natural length by opposing any extension (body relative distance > spring natural length) or any compression (body relative distance < spring natural length).

The mathematical relation between the force exerted by a spring (*F*), its natural length (*l*), its stiffness (*k*) and the distance between its two endpoints (*d*) is called *Hooke's Law* and is written as:

$$F = -k(d-l)$$

The separation between the two attached bodies is governed by two limits that may both be *hard* (which simulates a rod or strut joint) or both *soft* (simulating a spring) or hard on one limit but soft on the other (e.g. an elastic attachment that may be stretched but not compressed). The default behaviour is spring-like, with two soft, damped limits, both initialized at the initial separation of the bodies.

There is no angular constraint between bodies attached by a spring. The one linear dimension is constrained, restricting one degree of freedom. This adds just one row to the constraint matrix.

The spring is a configurable distance constraint.

- String can be simulated that can decrease in length but not increase.
- Elastic, that can decrease in length and can stretch can be simulated.
- A solid rod that cannot change it's length can be simulated.

Functions that are Specific to the Spring Joint

Accessors

The accessor functions specific to `Spring` are:

```
MdtLimitID MEAPI MdtSpringGetLimit ( const MdtSpringID joint );
```

Return the ID handle of a constraint limit.

```
void MEAPI MdtSpringGetPosition( const MdtSpringID joint, MeVector3
position,
const unsigned int bodyindex );
```

The spring joint attachment position to the body `bodyindex` is returned in `position`.

Mutators

The mutator functions specific to `Spring are:`

```
void MEAPI MdtSpringSetLimit( const MdtSpringID joint,
                              const MdtLimitID NewLimit );
```

Reset the joint limit and copy the public attributes of `NewLimit`.

```
void MEAPI MdtSpringSetNaturalLength( const MdtSpringID joint,
const MeReal NewNaturalLength );
```

Set the spring length under no load i.e. it's *natural length*.

```
void MEAPI MdtSpringSetStiffness( const MdtSpringID joint,
const MeReal NewStiffness );
```

Set the spring stiffness or *spring constant*.

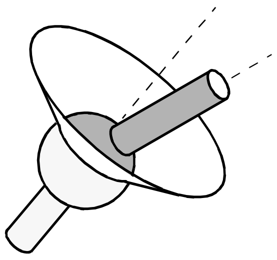
```
void MEAPI MdtSpringSetDamping( const MdtSpringID joint,
const MeReal NewDamping )
```

Set the spring damping value.

```
void MEAPI MdtSpringSetPosition( const MdtSpringID joint,
const unsigned int bodyindex,
const MeReal x, const MeReal y, const MeReal z )
```

Set the joint position in world coordinates. This function differs from the generic `Mdt*SetPosition` by requiring a `bodyindex`.

Cone Limit constraint: MdtConeLimit



The `MdtConeLimit` constraint places a limit on the angle between a pair of axes, one being fixed in each body. This constraint can be used in parallel with a ball and socket joint, for example, to limit its angular freedoms to a cone as shown in the sketch. Note that the cone limit does not place a limit on the ‘twist’ freedom.

The Cone Limit behaves more like a contact than a joint in that it adds no constraint while inside the limit. When the limit is hit, a single constraint is generated to enforce the angular limit.

The Cone-Limit constraint can be used in parallel with a UniversalJoint, that will also constrain the twist freedom, or on top of an Angular3 or an RPROJoint with similar effect.

The behavior of a Cone-Limit is ill defined for small cone angles, so angles less than about $5°$ should not be used.

Cone Limit Functions

The reset function defaults to using the x-axes of the two body frames and limits the angle between them to PI radians, which is effectively no limit.

Accessors

The accessor functions specific to `the Cone Limit are`:

```
MeReal MEAPI MdtConeLimitGetConeHalfAngle(const MdtConeLimitID j);
```

Return the cone half angle; i.e. the angle between the cone axis and the side of the cone.

```
MeReal MEAPI MdtConeLimitGetStiffness(const MdtConeLimitID j);
```

Return the current limit stiffness.

```
MeReal MEAPI MdtConeLimitGetDamping(const MdtConeLimitID j);
```

Return the current limit damping.

Mutators

The mutator functions specific to `the Cone Limit are:`

```
void MEAPI MdtConeLimitSetConeHalfAngle(const MdtConeLimitID j,
                                             const MeReal theta
);
```

Set the limit cone half angle angle to `theta`. The cone half angle is the angle between the cone axis and the side of the cone. Defaults to ME_PI at reset.

```
void MEAPI MdtConeLimitSetStiffness(const MdtConeLimitID j, const
MeReal kp);
```

Set the limit stiffness to k_p.

```
void MEAPI MdtConeLimitSetDamping(const MdtConeLimitID j, const
MeReal kd);
```

Set the limit damping to k_d.

Joint Limit: MdtLimit

A Joint is defined as a constraint on the free movement of bodies relative to one another. To achieve this, joints must themselves be constrained. The constraint of a joint is called a *Limit*. Each joint may have one or several limits. In practice, 2 limits would be a sensible maximum number of limits. Adding more limits would not further constrain the system, it would just increase the size of the constraint matrix that would need to be solved, slowing down the simulation. Adding limits to a prismatic or hinge joint, where a high and low limit are set, will add a row to the constraint matrix which is equivalent to losing one degree of freedom. In Karma, each limit is represented by a `MdtBclLimit` structure and is defined by a list of parameters that describe its action on a joint.

The Mdt Library provides a set of *accessors/mutators* and *indicators/actuators* to interact with the `MdtBclLimit` structure. Unlike a mutator, an actuator acts like a simple *on/off* switch, and does not require any value other than a boolean value. An indicator acts like an indicator light, telling you if an actuator is *on* or *off* by returning the appropriate boolean value.

Accessors:

```
MeReal MEAPI MdtLimitGetPosition( const MdtLimitID limit );
```

Return the relative position of the bodies attached to the joint.

```
MeReal MEAPI MdtLimitGetVelocity( const MdtLimitID limit );
```

Return the relative velocity of the bodies attached to the joint.

```
MeReal MEAPI MdtLimitGetStiffnessThreshold( const MdtLimitID limit );
```

Return the limit stiffness threshold. Please refer to`MdtLimitSetStiffnessThreshold()` in the mutator section.

```
MeReal MEAPI MdtLimitGetMotorDesiredVelocity( const MdtLimitID limit );
```

Return the desired velocity of the motor. A lower limiting velocity may be achieved if the attached bodies are subject to velocity or angular velocity damping.

```
MeReal MEAPI MdtLimitGetMotorMaxForce( const MdtLimitID limit );
```

Return the maximum force that the motor is allowed to use to attain its desired velocity.

Mutators

```
void MEAPI MdtLimitSetLowerLimit( const MdtLimitID limit,
const MdtSingleLimitID sl );
```

Set the lower limit properties by copying the single limit data into the `MdtBclLimit` structure. If the lower limit stop is higher than the current upper limit stop, the latter is also reset to the new stop value.

```
void MEAPI MdtLimitSetUpperLimit( const MdtLimitID limit,
const MdtSingleLimitID sl );
```

Set the upper limit properties by copying the single limit data into the `MdtBclLimit` structure. If the upper limit stop is lower than the current lower limit stop, the later is also reset to the new stop value.

```
void MEAPI MdtLimitSetPosition( MdtLimitID limit, const MeReal NewPosition
);
```

This sets an offset that is used to transform the measured relative position coordinate into the user's coordinate system. It does not change the actual position or orientation of any modelled object.

```
void MEAPI MdtLimitSetStiffnessThreshold( const MdtLimitID limit,
const MeReal NewStiffnessThreshold );
```

Set the limit stiffness threshold. When a limit stiffness exceeds this value, damping is ignored and only the restitution property is used. When the limit stiffness is at or below this threshold, restitution is ignored, and the stiffness and damping terms are used to simulate a damped spring. The stiffness threshold is enforced to be non-negative: the initial value is `MEINFINITY`.

```
void MEAPI MdtLimitSetLimitedForceMotor( const MdtLimitID limit,
const MeReal desiredVelocity,
const MeReal forceLimit );
```

Set the limited-force motor parameters, enforcing a non-negative value of `forceLimit`. If the latter is zero, this service deactivates the motor: otherwise, the motor is activated. This service does not enable attached disabled bodies.

Actuators

```
void MEAPI MdtLimitCalculatePosition( const MdtLimitID limit,
const MeBool NewState );
```

Set or clear the "calculate position" flag without changing the limit's activation state. Note that if the limit is currently activated or powered, the "calculate position" flag cannot be cleared.

```
void MEAPI MdtLimitActivateLimits( const MdtLimitID limit,
const MeBool NewActivationState );
```

Activate (if `NewActivationState` is non-zero) or deactivate (if zero) the limit, without changing any other limit property.

```
void MEAPI MdtLimitActivateMotor( const MdtLimitID limit,
const MeBool NewActivationState );
```

Activate (if `NewActivationState` is non-zero) or deactivate (if zero) the limited-force motor on this joint axis, without changing any other limit property.

Indicators:

```
MeBool MEAPI MdtLimitIsActive( const MdtLimitID limit );
```

Returns non-zero if the corresponding degree of freedom of the joint (i.e. the joint position or angle) has a limit imposed on it, and zero if it does not. Most joints have more than one degree of freedom. Joint limits are inactive by default, and will not affect the attached bodies until activated and non-zero stiffness and/or damping properties are set.

```
MeBool MEAPI MdtLimitPositionIsCalculated( const MdtLimitID limit );
```

Returns non-zero if the position or angle of the corresponding degree of freedom of the joint is to be calculated, and zero if it is not calculated. If the degree of freedom is either limited or actuated (i.e. powered), the joint position must be calculated. By default, joint positions are not calculated.

```
MeBool MEAPI MdtLimitIsMotorized( const MdtLimitID limit );
```

Returns non-zero if the limit is motorized, and zero if it is not. Joint limits are motorized by default.

The Single Joint Limit: MdtSingleLimit

Each limit contains two sub-structures of type `MdtBclSingleLimit`:

Structure Member	Description
`MeReal damping`	The damping term (k_d) for this limit. This must not be negative. The default value is zero. This property is used only if the limit hardness is less than or equal to the damping threshold. If the hardness and damping of an individual limit are both zero, it is effectively deactivated.
`MeReal restitution`	The ratio of rebound velocity to impact velocity when the joint reaches the low or high stop. This is used only if the limit hardness exceeds the damping threshold. Restitution must be in the range zero to one inclusive: the default value is one.
`MeReal stiffness`	The spring constant (k_p) used for restitution force when a limited joint reaches one of its stops. This limit property must be zero or positive: the default value is `MEINFINITY`. If the stiffness and damping of an individual limit are both zero, it is effectively deactivated.
`MeReal stop`	Minimum (for lower limit) or maximum (for upper limit) linear or angular separation of the attached bodies, projected onto the relevant axis. For a soft limit, the stop is a boundary rather than an absolute limit.

Accessors

```
MeReal MEAPI MdtSingleLimitGetDamping ( const MdtSingleLimitID sl )
```

Return the damping term (k_d) for this limit.

```
MeReal MEAPI MdtSingleLimitGetRestitution ( const MdtSingleLimitID sl )
```

Return the restitution of this limit.

```
MeReal MEAPI MdtSingleLimitGetStiffness ( const MdtSingleLimitID sl )
```

Returns the spring constant (k_p) used for restitution force when a limited joint reaches one of its stops.

Mutators

```
void MEAPI MdtSingleLimitReset ( const MdtSingleLimitID limit )
```

Initialize the individual limit data and set default values (`position = 0, restitution = 1, stiffness = MEINFINITY, damping = 0`).

```
void MEAPI MdtSingleLimitSetDamping ( const MdtSingleLimitID sl,
                                      const MeReal NewDamping)
```

Set the damping property of the limit. Damping is enforced to be non-negative. The initial value is zero.

```
void MEAPI MdtSingleLimitSetRestitution ( const MdtSingleLimitID sl,
                                      const MeReal NewRestitution )
```

Set the restitution property of the limit. Restitution is enforced to be in the range zero to one inclusive. The initial value is one.

```
void MEAPI MdtSingleLimitSetStiffness ( const MdtSingleLimitID sl,
                                      const MeReal NewStiffness )
```

Set the stiffness property of the limit. Stiffness is enforced to be non-negative. The initial value is `MEINFINITY`.

```
void MEAPI MdtSingleLimitSetStop ( const MdtSingleLimitID sl,
                                      const MeReal NewStop )
```

Set a limit on the linear or angular separation of the attached bodies.

Functions that are specific to Contacts

A handful of function are specific to contact constraints. This section lists all those functions and comments on them:

Accessors

```
void MEAPI MdtContactGetNormal (const MdtContactID contact,
                                                 MeVector3
normalVec);
```

Return the contact normal of `contact` in `normalVec`, in the world reference frame.

```
MeReal MEAPI MdtContactGetPenetration (const MdtContactID contact)
```

Return the current penetration depth at this contact.

```
void MEAPI MdtContactGetDirection (const MdtContactID contact,
                                                 MeVector3
directVec)
```

The contact primary direction is returned in `directVec`, in the world reference frame.

```
MdtContactParamsID MEAPI MdtContactGetParams (const MdtContactID
contact)
```

Return a `MdtContactParamsID` pointer to the contact parameters of this contact.

The following function is only used in conjunction with Karma Collision.

```
MdtContactID MEAPI MdtContactGetNext (const MdtContactID contact);
```

Return the pointer to the next contact associated with this body pair if it was set in collision. If the return value is equal to `MdtContactInvalidID` then no next contact exists.

Mutators

The following functions are mutators specific to contacts:

```
void MEAPI MdtContactSetBodies (const MdtContactID contact, const
MdtBodyID body1, const MdtBodyID body2);
```

Set the contact bodies body1 and body2 to be attached to `contact`.

```
void MEAPI MdtContactSetNormal (const MdtContactID contact, const
MeReal xNorm, const MeReal yNorm, const MeReal zNorm)
```

Set the contact normal of `contact` to `(xNorm, yNorm, zNorm);`

```
void MEAPI MdtContactSetPenetration ( const MdtContactID contact,
const MeReal penetration );
```

Set the value `penetration` of the penetration depth at the contact `contact`.

```
void MEAPI MdtContactSetParams ( const MdtContactID contact,
const MdtContactParamsID parameters );
```

Utility for setting all contact parameters. This allows the user to set all values in the `MdtContactParams` structure at once.

```
void MEAPI MdtContactSetNext ( const MdtContactID contact,
const MdtContactID nextContact );
```

Set the pointer of `contact` to the next contact `nextContact`. Used by Mcd collision.

```
void MEAPI MdtContactSetDirection ( const MdtContactID contact,
                const MeReal xDir, const MeReal yDir, const MeReal
zDir );
```

Set the primary direction for this contact at (`xDir, yDir, zDir`).

This is only necessary if surface properties are to vary depending on the direction. For isotropic contacts, this function should not be called, and the primary and secondary parameters should be set to the same value. The direction should always be perpendicular to the given normal, and the secondary direction is perpendicular to the primary direction.

MdtContactGroups

To simplify management of contacts, Karma organises contacts into ContactGroups. A contact group is a constraint between two bodies, or a body and the world, that holds all the contacts between those bodies. This makes it easy to deal with the contacts as a group.

Functions that are specific to MdtContactGroups

Accessors.

```
MdtContactID MEAPI MdtContactGroupGetFirstContact (
                                   MdtContactGroupID group );
```

Return the first contact in a contact group, or NULL if the contact group is empty.

```
MdtContactID MEAPI MdtContactGroupGetNextContact (
                     MdtContactGroupID group, MdtContactID contact
)
```

Return the contact following *contact* in the contact group, or NULL if *contact* is the last contact.

```
int MEAPI MdtContactGrouptGetCount ( MdtContactGroupID group )
```

Return the number of contacts in the group.

```
MeReal MEAPI MdtContactGroupGetNormalForce ( MdtContactGroupID
group )
```

Return the magnitude of the last timestep’s normal force between the two bodies connected by the group.

MeI32 MEAPI **MdtContactGroupGetSortKey**(const MdtContactGroupID group);

Return the sort key of *contactgroup*.

Mutators

The following functions are mutators specific to contacts:

```
MdtContactID MEAPI MdtContactGroupCreateContact (MdtContactGroupID
                 group);
```

Create a new contact and add it to the contact group. Returns the new contact.

```
void MEAPI MdtContactGroupDestroyContact (MdtContactGroupID group,
MdtContactID contact);
```

Remove *contact* from the contact group.

```
void MEAPI MdtContactGroupAppendContact ( MdtContactGroupID group,
MdtContactID contact);
```

Append *contact* to the contact group.

```
void MEAPI MdtContactGroupRemoveContact ( MdtContactGroupID group,
MdtContactD contact );
```

Remove *contact* from the contact group, but don’t delete it.

void MEAPI **MdtContactGroupSetSortKey**(const MdtContactGroupID `group`,
MeI32 key);

`Assign a sort key to` *contactgroup*`.`

The MdtBclContactParams Structure

All the properties of a given contact between two objects are stored in a `MdtBclContactParams` structure, such as the contact type, the friction model or the coefficient of restitution.

The list of members of the `MdtBclContactParams` structure follow.

Member	Description
`MdtContactType type`	Contact type (zero, 1D or 2D friction)
`MdtFrictionModel model1`	Friction model to use along primary direction.
`MdtFrictionModel model2`	Friction model to use along secondary direction.
`int options`	Bitwise combination of MdtBclContactOption's.
`MeReal FrictionCoefficient`	The friction coefficient for use in the normal force friction model.
`MeReal restitution`	Restitution parameter.
`MeReal velThreshold`	Minimum velocity for restitution.
`MeReal softness`	Contact softness parameter (soft mode). Violates ideal constraint, allows penetration and gives a ‘springy’ effect as well. It can cause objects to take longer to come to rest. Values range from 0 to 1, with 1 being very soft and .1 or .001 being typical.
`MeReal max_adhesive_force`	Contact maximum adhesive force parameter (adhesive mode). Sticky contacts may not work very well because when the penetration of a contact goes negative (the contact has separated) the contact is destroyed, and the 'sticky' force can't pull the objects back together again.
`MeReal friction1`	Maximum friction force in primary direction.
`MeReal friction2`	Maximum friction force in secondary direction.
`MeReal slip1`	First order slip in primary direction.
`MeReal slip2`	First order slip in secondary direction.
`MeReal slide1`	Surface velocity in primary direction.
`MeReal slide2`	Surface velocity in secondary direction.

There exist a large number of functions, mutators and accessors, to interact with this structure. These are listed in the `MdtContactParams.h` header file, in the reference manual. The more popular ones follow:

To apply the friction at a contact point:

```
void MEAPI MdtContactParamsSetType ( const MdtContactParamsID param,
                                     const MdtContactType conType );
```

Set the type `conType` of the contact parameters structure `param`.

Karma Dynamics supports three main friction modes:

Friction Type	Description
`MdtContactTypeFrictionZero`	Frictionless contact
`MdtContactTypeFriction1D`	Friction only along primary direction
`MdtContactTypeFriction2D`	Friction in both directions

When using `MdtContactTypeFrictionZero`, the direction or the coefficient of friction does not need setting.

When using `MdtContactTypeFriction2D`, friction acts in orthogonal directions on the plane of contact, and the properties for each direction can be set. The direction of a 2D contact will be set automatically.

The coefficients of friction can be specified separately in the primary and secondary directions. The primary and secondary directions are perpendicular to each other. To specify the primary direction, use:

```
void MEAPI MdtContactSetPosition ( const MdtContactID contact,
                       const MeReal x, const MeReal y, const MeReal z );
```

Set the primary direction for this contact. This is only necessary to define surface properties to vary depending on the direction. For isotropic contact, this function should not be called, and the primary and secondary parameters should be set to the same value. The direction should always be perpendicular to the given normal, and the secondary direction is perpendicular to the primary direction. The secondary direction is automatically set according to the right-hand rule.

To specify the friction model that a contact will use, use the function

```
void MEAPI MdtContactParamsSetPrimaryFrictionModel (
                                const MdtContactParamsID param,
                                const MdtFrictionModel fModel );
```

Set the friction model `fModel` to use along the primary direction.

:

Friction Model	Description
`MdtFrictionModelBox`	Box Model of friction (simplified Coulomb)
`MdtFrictionModelNormalForce`	Friction based on the normal force. Coulomb like.

To reset the contact structure to its default values, use:

```
void MEAPI MdtContactParamsReset ( const MdtContactParamsID param )
```

Initializes the contact parameters structure to the default values.

The following values are reset to their default values.

Member	Default Value
MdtContactType type	MdtContactTypeFrictionZero
MdtFrictionModel model1/model2	MdtFrictionModelBox
MeReal restitution	0.0
MeReal velThreshold	0.001
MeReal softness	0.0

Member	Default Value
MeReal max_adhesive_force	0.0
MeReal slip1/slip2	0.0
MeReal slide2/slide2	0.0

When a joint or a contact is no longer needed, it can be removed with the following function:

```
void MEAPI MdtContactDestroy( const Mdt*ID joint_or_contact);
```

This function destroys the joint or contact named `contact, where * represents the joint or contact type.`

TThe function that creates a contact and returns a `MdtContactID` variable is:

```
MdtContactID MEAPI MdtContactCreate ( const MdtWorldID world );
```

This function creates a new joint or contact in the world.

The formal description of these contact mutators are

```
void MEAPI MdtContactSetNormal ( const MdtContactID contact,
            const MeReal xNorm, const MeReal yNorm,const MeReal
zNorm );
```

Set the contact normal of `contact` to `(xNorm, yNorm, zNorm).`

and

```
void MEAPI MdtContactSetPenetration ( const MdtContactID contact,
const MeReal penetration );
```

Set the value `penetration` of the penetration depth at the contact `contact`.

Formal description:

```
MdtContactParamsID MEAPI MdtContactGetParams ( const MdtContactID
contact );
```

Return a `MdtContactParamsID` pointer to the contact parameters of this contact.

The `MdtContactEnable()` and `MdtContactDisable()` are not functions, but macros that were implemented to save writing two lines instead of one.

Glossary

Acceleration	Rate of change of velocity with respect to time, or how fast velocity is increasing.
Actuator	A mechanical device for moving or controlling something, such as a motor, a lever, a piston, and so on. Actuators can be implemented with forces and torques or with velocity constraints.
Angular Velocity	The rate of rotation about an axis. This is a vector quantity whose direction points along the instantaneous axis preserved by the rotation and whose magnitude is the rate of rotation about this axis.It is usually expressed in radians per second or revolutions per second.
Ballistics	Ballistics is the science of the motion and behavior of projectiles, missiles, bullets and other similar objects in flight submitted to a gravitational field and air resistance.
Ballistic Objects	In a games development environment, ballistic objects are usually single rigid body projectiles that are subject to ballistic calculations without contacts or impulses.
BSP Tree	Binary Space Partitioning (BSP) tree. BSP trees are tools used in 3D graphics for organizing the geometry of objects and extracting information about geometrical relationships. In 3D worlds they are often used for visibility determination and hidden surface removal.
Cartesian Coordinates	A coordinate system which is based on orthogonal axes. This is the standard x, y and z coordinate system, in contrast to polar coordinates on a sphere. In 3D graphics applications, x usually stands for left-right position, y describes vertical position and z gives the position along the line of sight, (i.e., depth). Traditionally, graphics viewers use a Left Handed system where x increases to the right, y increases towards the top, and z increases further along the line of sight i.e., into the screen.
Center of Mass	The point in a body or system of bodies at which the whole mass may be considered as concentrated. For the purpose of dynamics calculations, a body or system of bodies can be abstracted to a point mass located at its center of mass coordinates.
Collision Detection	A calculation which determines if two objects are intersecting, and if so, also computes the location and the local geometry of the object near the intersection. Collision detection may determine the spatial relationships between objects, such as the approximate separation distance between them.
Collision Model	A geometrical representation of an object used for collision detection. Vortex supports a variety of primitives such as sphere, box, plane, cylinder and cone as well as more complicated objects such as convex primitives and unstructured lists of triangles. The collision model may be identical to the model used for rendering or maybe an approximation that allows for faster collision detection. Collision models can also be made up of a combination of collision primitives.
Collision Response	The change of motion of an object which occurs when it collides with another object, or when a joint limit is reached. Note that collision response is related to the dynamics properties of the objects in contact. Collision response is determined by the degree of restitution and the friction properties at the point of impact.
Constraint	An external restriction on the motion of an object or body. There are equality and inequality constraints, constraints on positions and velocities, ideal constraints and non-ideal ones, and further constraints on forces resulting from constraints. There are constraints involving a single rigid body and others involving more. Constraints are maintained by computing a force that must be applied to all involved bodies so that the constraint remains satisfied. An example of an equality constraint could be that a joint attached to a specific location of a body should not be allowed to wander off a preset location, that is, the body may move as long as that point remains where it was initially (this still allows object rotation about that point). An *inequality* restriction could be a ball placed in a room: allowed to move freely on the floor as long as it stays inside the area bordered by the walls.

These restrictions are used to connect objects together, such as connecting two rigid bodies with a hinge, or to impose motion in a joint by applying a force limited velocity constraint which is used to model a motor. Constraints are called ideal or non-ideal according to whether or not they dissipate energy, i.e.: whether or not the constraint force does work on the system. A ball and socket joint is an ideal constraint but a box friction contact constraint is non-ideal.

Constraint Equation A mathematical relation between the coordinates of several bodies.

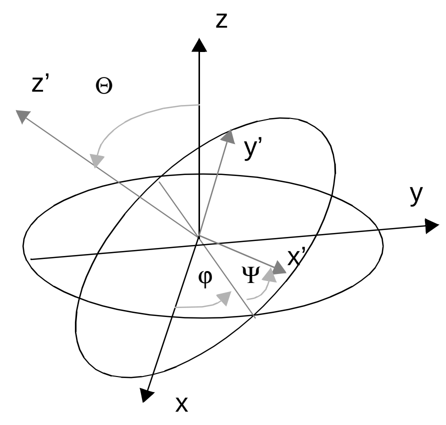
Culling Culling is the process of selectively removing items from a collection. For example, the process of eliminating certain polygons that shouldn't be seen before drawing a scene.

Convex Object A geometrical object with no holes in it, such as a sphere or a cube. More precisely, an object is convex if given any two points of the interior or the surface of the object, all the points that lie on the line segment between those two points are also inside or on the surface of the object. A doughnut is not convex for instance.

Damping This is a rate of energy loss of a body. It results in the body slowing down and being gradually brought to rest. Rate of energy loss is proportional to velocity.

Dynamics Dynamics is the part of mechanics (which, of course is a branch of physics) that is concerned with the causes of accelerating motion. It describes the change in motion of objects or particles using the concepts of force, momentum, energy, and mass. It is governed by the three basic laws of motion, called Newton's laws. See Newton's Laws.

Euler Angles The Euler angles are the three angles Ψ, Θ and φ (for yaw, pitch and roll) used to describe the orientation of a body's reference frame relative to the world reference frame.



The resulting rotation matrix R_{ij} would be written as:

$$\begin{bmatrix} \cos\Psi\cos\varphi - \cos\Theta\sin\varphi\sin\Psi & \cos\Psi\sin\varphi + \cos\Theta\cos\varphi\sin\Psi & \sin\Psi\sin\Theta \\ -\sin\Psi\cos\varphi - \cos\Theta\sin\varphi\cos\Psi & -\sin\Psi\sin\varphi + \cos\Theta\cos\varphi\cos\Psi & \cos\Psi\sin\Theta \\ \sin\Theta\sin\varphi & -\sin\Theta\cos\varphi & \cos\Theta \end{bmatrix}$$

Another way of representing a body's rotation is with a *quaternion.*

Force A force is what causes the state of motion (momentum) of an object to change. If you to change the way something is moving (change its direction or make it go faster, for example) you must apply a force to it. Forces are vector quantities with orientation and magnitude. The bigger the force, the faster the motion will change. In 3D games' worlds, forces typically include gravity and wind.

Force-limited Motor This is a special model for controlling the remaining degree of freedom in either a hinge or a prismatic joint in the Vortex Dynamics Library. It allows you to control the relative linear velocity of two bodies connected by a prismatic joint or the relative angular velocity of two bodies connected with a hinge joint. You set the desired target velocity, which will be achieved provided that

	the force required to do so is less than the limit specified, otherwise, the maximum force is applied. You may control the desired velocity using a mouse or a joystick input signal. Force limited actuators respond very quickly and are inherently stable; they are the best way to introduce user controlled motion in a joint in the Vortex Dynamics Library.
Frame	A single rendered screen of graphics,
Frame Rate	The number of screen images displayed per second, usually abbreviated by *fps* (frame per second). Common frame rates are 60fps for games consoles, 25fps for PAL video, and 30fps for NTSC video, although other speeds may be used
Friction Force	The force that resists relative motion between two bodies in contact. Viscous friction imposes a force which is proportional to the relative velocity and opposed to it. Dry friction has two modes: sliding mode and stick mode. When bodies are sliding, this force directly opposes the relative motion and has a magnitude proportional to the normal force which keeps the two bodies from interpenetrating. Otherwise, this force prevents the bodies from sliding, in which case it does not dissipate energy.
GJK Algorithm	A fast method of doing collisions between a set of convex objects invented by Gilbert, Johnson and Keerthi and described in: "A Fast Procedure for Computing the Distance between Complex Objects in Three-Dimensional Space" by E. G. Gilbert, D. W. Johnson and S. S. Keerthi, IEEE Trans. Robotics and Automation 4(2):193-203, April 1988.
GJK Engine	The part of the collision detection software that implements the GJK algorithm.
Gravity	In a Newtonian world, gravity is a force that attracts all objects to each other. Gravitational force between any two given objects is proportional to the products of their masses and inversely proportional to the square of the distance that separates them. However, we are mostly interested in constant and uniform gravity which is the force that keeps your chair on the floor. For objects near the surface of the earth, gravity causes objects to accelerate downward at a constant rate of 9.81 m/sec2, independent of their mass. This means that gravity causes an object's velocity to change by 9.81 m/sec2 downward for every second the object is under the influence of gravity.
Hooke's Law	This law states that the force on a object displaced from a point of equilibrium is proportional to the displacement and in the direction opposite to the displacement.
Impulse	The change in momentum of an object. This can happen over either a finite time interval or over a vanishingly small time interval as in the case of collisions. Most of the time, people mean the latter when referring to impulses i.e., forces which act over a very small amount of time and which cause sudden changes in the velocity of an object.
Inertia	The tendency of an object to maintain its state of rest or of uniform motion. The resistance to change in the quantity of linear motion (linear momentum) is given by the mass of the object. Resistance to change in the quantity of angular motion (angular momentum) about an axis is the moment of inertia about that axis.
Inertia Tensor	The set of numbers that describe the resistance to change in angular motion. This consists of several numbers because, in any given orientation, a body may have different inertia to rotation about all three axes, x, y and z. This quantity also changes with the orientation of the body, which is why it is a tensor. The inertia tensor can be represented as a 3x3 matrix which is symmetric and positive definite.
Inertial Reference Frame	A reference system in which Newton's laws of motion hold. See *Newton's Laws*.
Joint	An equality constraint or connection between objects or bodies. When two or more bodies or objects are connected by joints, they form a multi-body system which is also called a jointed body or an articulated body.

Kinematics	Kinematics is the study of how things move in relation with time. It describes the different types of motion a body can undergo: translational (change of position), rotational (change of orientation), vibrational (change of shape and size).
Local Coordinates	The origin of the local coordinate system of a body is usually located at its center of mass.
Jacobian Matrix	Given any vector function of several variables with components $F_1(x_1, x_2, \ldots, x_n)$, $F_2(x_1, x_2, \ldots, x_n)$, ... $F_n(x_1, x_2, \ldots, x_n)$, the Jacobian matrix is defined as: $J_{ij} = \frac{\partial F_i}{\partial x_j}$ In the case where the functions F_j represent a coordinate transformation between x_i and $q_j = F_j$, the Jacobian matrix is the linear approximation of the coordinate transformation in the neighborhood of the point where the derivatives of F_j are evaluated. In a more general way, the Jacobian matrix is the local linear approximation of a non-linear function in the neighborhood of a point x_0: $F(x) = F(x_0) + J(x-x_0)$ In Kea, we use the Jacobians of the constraint functions in the computation of the constraint forces, i.e. the forces required to maintain the constraint.
Mass	Mass is the measure of the amount of material contained within a body. It is independent of the location of the body.
Mechanics	A branch of physics that deals with the analysis of motion of physical objects. This includes the study of the equations of motion derived in dynamics.
Momentum	This is a quantitative measure of the state of motion of an object. Linear momentum is the product of mass and velocity: $\vec{p} = m\vec{v}$ and is a vector quantity. Angular momentum is a measure of the rotational motion of an object around some point.
Newton	The SI unit of force, derived from 3 basic units: mass, length and time.
Newtons' Laws	**Newton's first law of motion:** In an inertial frame, an object that is free of interaction has constant momentum. An object at rest remains at rest, and if it is in motion, that motion will be uniform and constant, that is, the center of mass will move on a straight line and the motion about the center of mass will be uniform. The tendency of an object to resist any change in motion is called the **inertia** of the object. **Mass** is a measure of inertia. **Newton's second law** states the relationship between an acting force, the momentum of a body, and its acceleration, as follows: $\frac{d\vec{p}}{dt} = \vec{F}$ where p is the momentum of the object and F is the applied force. In words, this means that the force exerted on a body is equal to the rate of change of momentum of that body. For point masses with constant mass, this simplifies to: $m\frac{d\vec{v}}{dt} = \vec{F}$ the familiar F = ma. As such, Newton's second law only applies to point masses and further analysis is required to derive the equations of motion for

rigid bodies and composite objects.

Newton's third law states that for every action there is an equal and opposite reaction. If two bodies interact, the magnitude of the force exerted by body 1 on body 2 is equal to the magnitude of the force exerted on body1 by body 2. These two forces are also opposite in direction.

Normals

Normals are unit vectors. The normal to a plane is a unit vector perpendicular to the plane. In 3D graphics, they indicate the visible side of an object.

Object

We use this term to describe virtually any entity that can move—a point mass, rigid body, articulated body, multi-body system, and so on. We use it as a general term when what is being discussed is not restricted to rigid bodies. Object does not refer to object-oriented programming constructs.

Ordinary Differential Equation

A differential equation is a relationship between the rate of change for the variables of a system and the state of that system. The term *ordinary* means that the derivatives are taken with *respect to an independent variable which is often labelled as time. When there is more than one independent variable, such as space for instance, the equations are called partial differential equations*.

Orientation

Describes a body’s direction relative to the world or reference frame coordinates.

Power

The time rate at which work is performed (energy is transformed) by a system.

Quaternion

Quaternions form a four-dimensional algebra which is an extension of normal complex numbers. They can be used to represent rotations of coordinate axes in 3D worlds in a way that is free of singularity, in contrast to Euler angles, for example. With Euler angles there are difficulties when the pitch angle reaches 90 degrees - a problem that never occurs when quaternions are used.

The four components of a unit quaternion which represents a 3D rotation can be thought of as a set of parameters for 3D rotations, as an alternative to Euler angles.

Because they lack the singularities that Euler systems have, quaternions allow for smooth interpolations between rotations. It is possible to convert *Euler Angles* to a quaternion *Q* using the following equation:

$$Q = q_o + q_x\boldsymbol{i} + q_y\boldsymbol{j} + q_z\boldsymbol{k}$$

where:

$$q_o = \cos\left(\frac{\Psi}{2}\right)\cos\left(\frac{\Theta}{2}\right)\cos\left(\frac{\varphi}{2}\right) + \sin\left(\frac{\Psi}{2}\right)\sin\left(\frac{\Theta}{2}\right)\sin\left(\frac{\varphi}{2}\right)$$

$$q_x = \cos\left(\frac{\Psi}{2}\right)\cos\left(\frac{\Theta}{2}\right)\sin\left(\frac{\varphi}{2}\right) + \sin\left(\frac{\Psi}{2}\right)\sin\left(\frac{\Theta}{2}\right)\cos\left(\frac{\varphi}{2}\right)$$

$$q_y = \cos\left(\frac{\Psi}{2}\right)\sin\left(\frac{\Theta}{2}\right)\cos\left(\frac{\varphi}{2}\right) + \sin\left(\frac{\Psi}{2}\right)\cos\left(\frac{\Theta}{2}\right)\sin\left(\frac{\varphi}{2}\right)$$

$$q_z = \sin\left(\frac{\Psi}{2}\right)\cos\left(\frac{\Theta}{2}\right)\cos\left(\frac{\varphi}{2}\right) + \cos\left(\frac{\Psi}{2}\right)\sin\left(\frac{\Theta}{2}\right)\sin\left(\frac{\varphi}{2}\right)$$

Restitution (coefficient of)

When designing your game, think of restitution as bounciness. In physics terms it is the ratio of the relative speeds of two bodies just after and just before a collision between them along their direction of impact; it ranges from 0, for perfectly inelastic collisions (no bounce), to 1 (maximum bounce), for completely elastic ones. If you increase the value to a number greater than one, you’ll get an effect like a pinball machine. It is a constant at moderate speeds.

Right—Hand Rule

A useful visual aid to represent a right-handed coordinate system. Here’s how it works: orient your right hand so that your thumb is perpendicular to the plane defined by the x and y axes, and so that you can curl your fingers in the direction from the x axis towards the y axis. Your thumb will be pointing in the direction of the z axis.

Rigid Body	A solid object that doesn’t change its shape or size, despite any forces being applied to it.
Rotation	In a 3D world, rotation of an object is its motion about an internal axis of symmetry relative to an internal coordinate system.
SI Units and Prefixes	SI stands for *Systeme International*. It is a standardized set of units for scientific measurement. Examples of SI units include: the Newton, metre, Hertz, second and Watt. Examples of SI prefixes include: mega, milli, tera and giga.
Stiff Springs	Stiffness is a measure of how a spring resists stretching. A very stiff spring will hardly move at all—even if you pull very hard, whereas a spring with very little stiffness will stretch easily even if you pull it gently. The stiffness is related to the 'spring constant' usually represented by the letter *K*. Writing this down as a formula called Hooke's law for the extension of springs we get; Force = - (spring constant) X (length increase) Stiff springs create very large forces for short times and this can be a problem in a simulation. The reason is that the force is never very large for very long. For a small mass attached on a stiff spring, the resulting motion for the mass is to stay *very* near the point of equilibrium. The force from the spring will change very rapidly from large to small to keep the mass in place. An integrator that is using a time step larger than half the period of the spring might overestimate the magnitude of the force and this can lead to an "explosion" i.e., a situation where the mass moves further and further away from the equilibrium point. Note also that the frequency of a spring is dependent on the mass attached on it and if you drop a small mass on a stiff spring, that will produce a very high frequency which might be difficult for the integrator. The constraint solver uses a first order integrator for user forces which doesn't check if forces are getting very large. This means that you should avoid stiff springs altogether. If you want to simulate a stiff spring, you should use a "relaxed" constraint instead i.e., a constraint that is allowed to be violated.
Time-Step	Time, like mass, is another of those rather undefined measures in physics. Everybody knows what it is, and agrees to measure it in seconds. A time step is an interval of time, typically of the order of the frame rate. If you run a simulation at 60 fps, your time step should be 1/60 s or shorter; if it is shorter, you will need to perform several steps between frames if you want to create a real-time simulation in which everything seems to move at the same rate as in the real world. The simulation computes what the state of the system will be after an interval of time equal to the time step has elapsed, using current state and derivative information. Essentially, the simulator uses current velocities and forces and extrapolates them to compute an approximation of the state into the future. The correctness of the approximation is related to the time step: the bigger the time step the worse the approximation.
Tracking Problem	The problem of keeping track of the distance between objects as simulated time is stepped forward. This is useful to estimate when objects will collide.
Torque	A torque can be described as a turning or twisting force. It’s the measure of a force's tendency to produce a rotation about an axis, which also produces a change in the angular momentum of a body. For an extended body—one that is not just a single point mass - a torque is generated by applying a force at a point other than at the center of mass. The distance between the point where the force is applied and the center of mass acts like a lever: the greater the distance, the bigger the resulting change in angular momentum.
Velocity	The rate of change of position along a straight line with respect to time. A velocity has speed and direction; it is a vector quantity whose magnitude is expressed in units of distance over time, such as kilometers per hour.
Work	Work is the product of force, and the distance moved by the point of application of the force in the direction that the force was applied. Work is directly related to the change in the energy of the system.

World Reference Frame	A coordinate system that is used to locate an object in relation to a world origin. This is sometimes referred to as a Global Coordinate system.
XYZ axes	In a Cartesian coordinate system, these are the three orthogonal axes which represent three-dimensional space.

Bibliography

References

Please note that the urls are correct as of Jan 2002.

Research Papers

Brian V. Miritch; ***'Impulse-Based Dynamic Simulation of Rigid Body Systems'***; PhD Thesis; University of California, Berkeley, 1996;

C. Lennerz, E. Schömer & T. Warken; ***'A Framework For Collision Detection And Response'***;

David Baraff. A Selection of David's Work follows:

- ***'Dynamic Simulation of Non-Penetrating Rigid Bodies'***; PhD Thesis; Mar. 1992.
- ***'Curved Surfaces and Coherence for Non-Penetrating Rigid Body Simulation'***; Computer Graphics; Vol. 24, N^{o}. 4, Aug. 1990.
- ***'Coping with Friction for Non-Penetrating Rigid Body Simulation'***; Computer Graphics; Vol. 24, N^{o}. 4, Aug. 1990.
- ***'Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies'***; Boston Computer Graphics; Vol. 23, N^{o}. 3, Jul. 1989.

David E. Stewart; ***'Time-Stepping Methods and the Mathematics of Rigid Body Dynamics'***; Chap. 9 in Impact and Friction of Solids, Structures and Machines, Vol. 1, Birkhäuser, Boston, 2000.

D. E. Stewart & J.C. Trinkle; ***'Dynamics, Friction, and Complementarity Problems'***; Complementarity and Variational Problems: State of the Art, Proc. 1995 International Conference on Complementarity Problems, J.-S. Pang and M.C. Ferris (Editors); pp. 425-39. SIAM Publ., Philadelphia, 1997.

D. E. Stewart & J. C. Trinkle; ***'An Implicit Time-Stepping Scheme for Rigid Body Dynamics with Inelastic Collisions and Coulomb Friction'***; To be published; Proceedings of the International Conference on Robotics and Automation (ICRA), 2000.

E. G. Gilbert, D. W. Johnson & S. S. Keerthi; ***'A Fast Procedure for Computing the Distance between Complex Objects in Three-Dimensional Space'***; IEEE Journal of Robotics and Automation; Vol. 4, N^{o}. 2, pp. 193-203, 1988.

Elmar Schömer & Christian Thiel; ***'Efficient collision detection for moving polyhedra'***; Proc. 11th Annual ACM Sympos. Comput. Geom., pp51-60, 1995.

Jeff Trinkle, Jong-Shi Pang, Sandra Sudarsk & Grace Lo; ***'On Dynamic Multi-Rigid-Body Contact Problems with Coulomb Friction'***; Accepted; Zeithschrift fur Angewandte Mathematik und Mechanik.

G. Hotz, A. Kerzmann, C. Lennerz, R. Schmid, E. Schömer & T. Warken; ***'Calculation of Contact Forces'***.

IO Interactive; http://www2.ioi.dk/Frontpage/Welcome.shtml Specifically the homepage of Thomas Jakobsen under the section ***'Interactive Physics Simulation Resources'***; http://www.ioi.dk/Homepages/tj/resources.htm

Jens Eckstein & Elmar Schömer; ***'Dynamic Collision Detection in Virtual Reality Applications'***; 7th International Conference in Central Europe on Computer Graphics and Visualization and Interactive Digital Media, WSCG'99, S. 71-78, 1999.

Jörg Sauer, Elmar Schömer; ***'A Constraint-Based Approach to Rigid Body Dynamics for Virtual Reality Applications'***; Proc. ACM Symposium on Virtual Reality Software and Technology, VRST'98, S. 153-161, 1998.

K. Kawachi, H. Suzuki, & F. Kimura; ***'Simulation of Rigid Body Motion with Impulsive Friction Force'***; Proceedings of IEEE International Symposium on Assembly and Task Planning; pp. 182-187, Aug. 1997.

Katsuaki Kawachi, Hiromasa Suzuki & Fumihiko Kimura; ***'Technical Issues on Simulating Impulse and Friction in Three Dimensional Rigid Body Dynamics'***; Proceedings of IEEE Computer Animation '98 Conference, Jun. 1998.

K. S. Anderson; ***'An Order n Formulation for the Motion Simulation of General Multi-Rigid-Body Constrained Systems'***; Computers and Structures; Vol. 43, N^{o}. 3, pp. 565-79, 1992.

Matthias Buck, Elmar Schömer; ***'Interactive Rigid Body Manipulation with Obstacle Contacts'***; The Journal of Visualization and Computer Animation; 1998. Presented in the 6th International Conference on Computer Graphics and Visulization, WSCG'98 pp. 49-56, 1998.

Mihai Anitescu and Florian A. Potra; ***'A Time-Stepping Method for Stiff Multibody Dynamics with Contact and Friction'***; To be published; Preprint ANL/MCS-P884-0501; Argonne National Laboratory, Argonne, IL, 2001f.

Mihai Anitescu & F. A. Potra; ***'Formulating Dynamic Multi-Rigid-Body Problems with Friction as Solvable Linear Complementarity Problems'***; Reports on Computational Mathematics; N^{o}. 93, Oct. 1996.

M. Anitescu, J. Cremer, & F. A. Potra; ***'Formulating 3D Contact Dynamics Problems'***; Mech. Struct. & Mach., Vol. 24, Nº. 4, pp. 405-437, Nov. 1996.

Mihai Anitescu, Florian A. Potra & David E. Stewart; ***'Time-Stepping for Three-Dimensional Rigid Body Dynamics'***; Comp. Methods Appl. Mech. Eng. Vol. 177, Nº. 3-4, pp. 183-197, 1999.

M. M. Kostreva, Communicated by F. Zirilli; ***'Generalization of Murty's Direct Algorithm to Linear and Convex Quadratic Programming'***; Journal of Optimization Theory and Applications; Vol. 62, Nº. 1, 1989.

Reinhold von Schwerin; ***'Multibody System Simulation'***; Springer-Verlag; 1999.

Relevant Articles

Andrew Witkin & David Baraff; ***'Physically Based Modeling: Principles and Practice'***; SigGraph Lecture notes; http://www-2.cs.cmu.edu/~baraff/sigcourse/index.html; 1997.

Chris Hecker; ***'Rigid Body Dynamics'*** - Four Physics Articles Written by Chris for Game Developer Magazine are Referenced, along with other Relevant References:

- ***'Physics Part 1: The Next Frontier'***; Oct. / Nov. 1996.
- ***'Physics, Part 2: Angular Effects'*** Dec. / Jan. 1996.
- ***'Physics, Part 3: Collision Response'*** Feb. / Mar. 1997.
- ***'Physics, Part 4: The Third Dimension'*** Jun. 1997.

http://www.d6.com/users/checker/dynamics.htm

A series of articles by Jeff Lander (you will need to register at Gamasutra to view these):

- ***'Collision Detection'***; Jan. 1999.

http://www.gamasutra.com/php-bin/login.php3?from=/features/20000210/lander_01.htm to

http://www.gamasutra.com/php-bin/login.php3?from=/features/20000210/lander_03.htm

- ***'Collision Detection'***; Feb. 1999.

http://www.gamasutra.com/php-bin/login.php3?from=/features/20000203/lander_01.htm to

http://www.gamasutra.com/php-bin/login.php3?from=/features/20000203/lander_03.htm

- ***'Collision Response: Bouncy, Trouncy, Fun'***; Mar. 1999.

http://www.gamasutra.com/php-bin/login.php3?from=/features/20000208/lander_01.htm to

http://www.gamasutra.com/php-bin/login.php3?from=/features/20000208/lander_01.htm

- ***'Lone Game Developer Battles Physics Simulator'***; Apr. 1999.

http://www.gamasutra.com/php-bin/login.php3?from=/features/20000215/lander_01.htm to

http://www.gamasutra.com/php-bin/login.php3?from=/features/20000215/lander_03.htm

An article on Gamasutra discussing ***'Quaternions'***; 1998.

http://www.gamasutra.com/php-bin/login.php3?from=/features/programming/19980703/quaternions_01.htm to

http://www.gamasutra.com/php-bin/login.php3?from=/features/programming/19980703/quaternions_07.htm

Books

Ahmed Shabana*;* ***'Dynamics of Multi-body Systems'*** and ***'Computational Dynamics'***; These books have much to offer but do not explain the details of simulations.

Ammon Katz; ***'Computational Rigid Vehicle Dynamics'***; Krieger Publishing Company; 1997.

Chris Watkins*;* ***'Taking Flight'***; A useful text for computer game programmers. It doesn't discuss physics.

David M. Bourg; ***'Physics for Game Developers'***; O'Reilly & Associates, Inc.; 2002.

Lev D. Landau & Evgenii M. Lifshitz; ***'Mechanics Third Edition. Course of Theoretical Physics Volume 1'***; Pergamon Press; 1994; This is a more difficult text often used by first year graduate students in physics. The presentation is excellent.

Halliday and Resnik*;* ***'Introduction to Physics'***; This is a good general introduction to physics.

Jack B. Kuipers; ***'Quaternions and Rotation Sequences - A Primer with Applications to Orbits, Aerospace, and Virtual Reality'***; Princeton University Press; 1998.

Mark DeLoura (Editor); ***'Game Programming Gems'***; Books I and II thus far in the series; Charles River Media, Inc.; 2001.

Mary Lunn; ***'A First Course in Mechanics'***; Oxford University Press; 1998.

Richard Feynman; ***'Lectures on Physics';*** Vol. I, Chap. 4, 8-14, and 18-21; Discusses the concepts of mass, force, energy and time. Useful for simulating masses in motion.

Thomas D. Gillespie; ***'Fundamentals of Vehicle Dynamics'***; Society of Automative Engineers, Inc.; 1992.

Thomas R. Kane & David Levinson; ***'Dynamics: Theory and Applications'***; McGraw-Hill; 1985.

Witkin and Kass; ***'Spacetime Constraints'***; In the Siggraph proceedings.

URLs

http://www.developmag.com/

http://www.gamanetwork.com/ Game Network leading to:

- http://www.gamasutra.com/
- http://www.gdmag.com/ Game Developer Magazine.

http://www.gamedev.net/

http://www.flipcode.com/

http://www.fgn.com/ Future Games Network.

Conferences

http://www.ects.co.uk/ Sep. 1-3 2002; London.

http://www.gdconf.com/ Mar. 19-23 2002; San Jose California.

http://www.gdc-europe.com/

http://www.microsoft.com/HWDEV/meltdown/default.asp Microsoft Meltdown.

http://www.milia.com/

http://www.siggraph.org/ Jul. 21-26 2002; San Antonio, Texas.

http://www.devnet.scea.com/research/