MathEngine Karma™ Viewer

Developer Guide

Karma Overview

Introduction

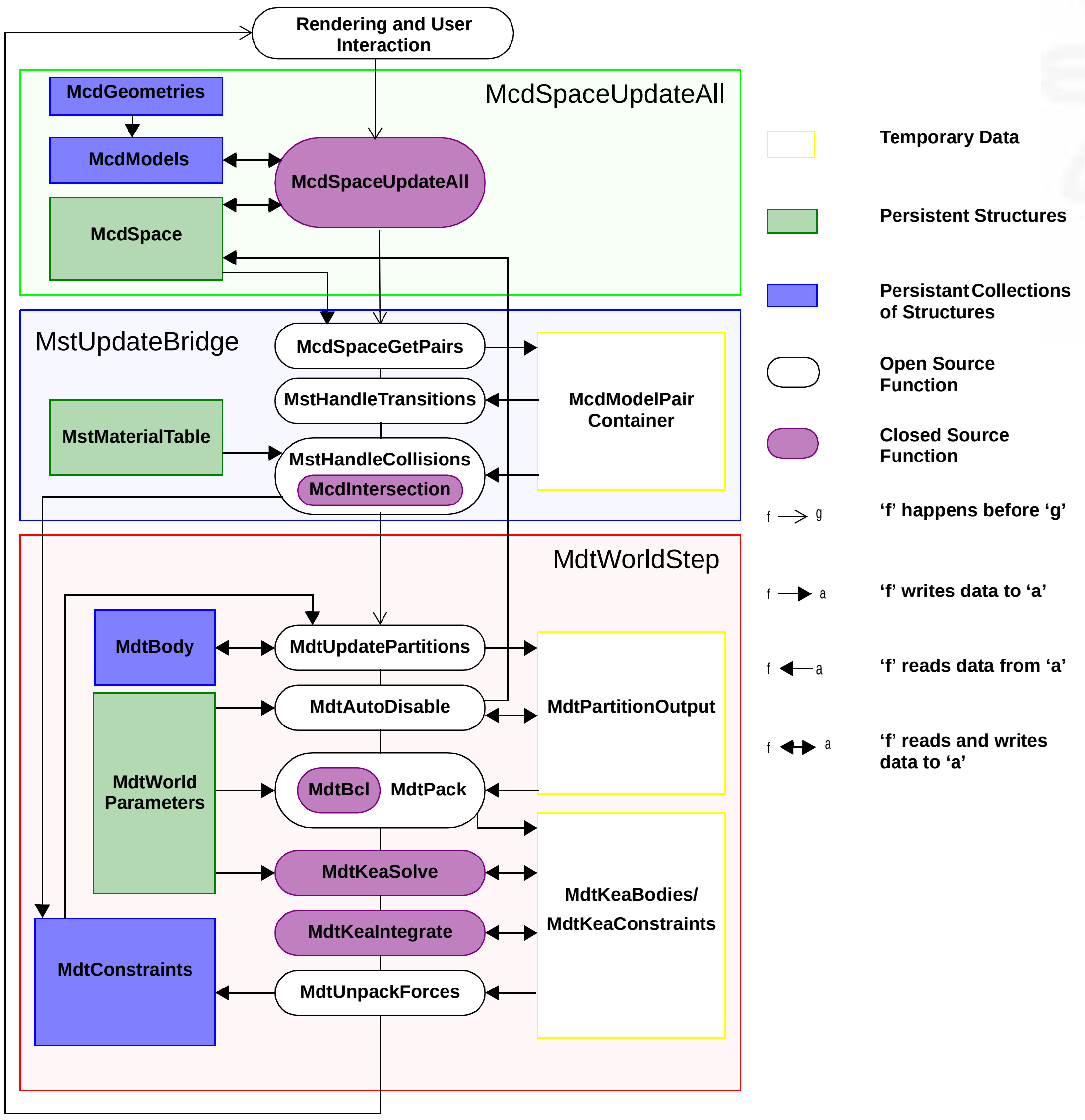
It is assumed that the reader of this manual is familiar with the basic mechanisms and facilities of Karma, as detailed in the *Karma User Guide*. The *Karma Reference Manual* is a more detailed guide to the behaviour and implementation of the toolkit.

Karma’s rigid-body simulation toolkit consists of two major modules: collision and dynamics. Each consists of a framework library which provides simple data management and high-level operations, for which source is included, together with several libraries which implement the underlying numerical algorithms. Karma collision and dynamics are designed to work together, but are usable independently.

There are, in addition, a number of smaller modules, one providing bridging utilities between dynamics and collision, and others providing services such as persistence and simple rendering which do not represent core functionality but can be useful for constructing sample simulations and interfacing with other software.

Karma is not thread-safe: you cannot run a single simulation using several threads, unless that simulation can be separated into pieces which do not interact. However, it is re-entrant, so you can run different simulations within different threads in the same executable.

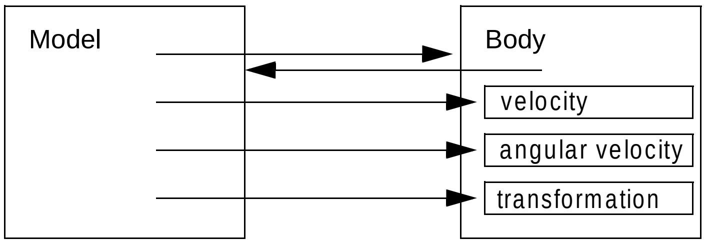
To use Karma most effectively and avoid many of the potential pitfalls, it’s important to have a good understanding of the components of Karma and how they work together. One useful perspective on Karma’s operation is as a data pipeline, described briefly in the *Karma User Guide*. This chapter will examine the pipeline in detail, explaining the basic data structures and how they link together to implement rigid-body simulation.



Dynamics and Collision

The Karma pipeline is partitioned into Karma Collision, Karma Dynamics, and the Karma Bridge. Collision and the bridge generate the contacts which constrain dynamics, and dynamics generates transformation matrices which inform collision. The two are fairly loosely coupled by just a few linkages, and in particular, neither has knowledge of the other's types. Routines which require information about types in dynamics and collision reside in the Karma bridge library.

Models and Bodies

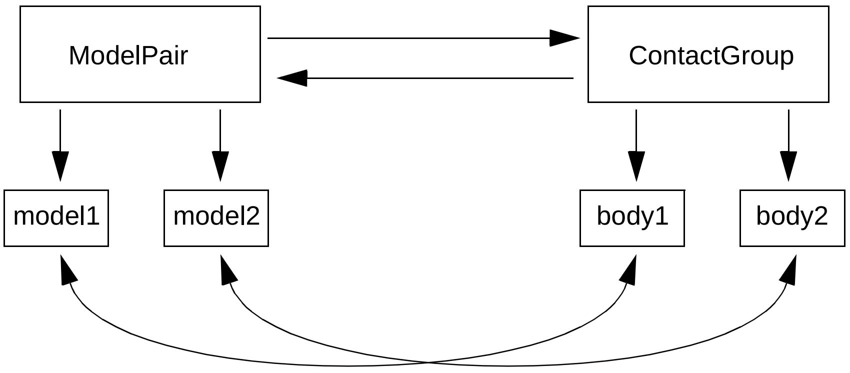


Although a collision model does not have to be associated with a dynamics body, if it is, then it holds not just a pointer to the body, but also pointers to the fields of kinematic data within the body. Generally collision tests only use the position and orientation information in the transformation matrix, but if you are using SafeTime collision tests, the velocity information is also required.

It is possible for a single dynamics body to have several associated collision models (although this is not recommended). The reference to the collision model is only used to freeze the model when the body is disabled, so if you do have several models for a body, it is your responsibility to ensure the models are frozen by keeping track of these associations and overriding the callbacks in the bridge.

[list callbacks]

Modelpairs and Contactgroups



A *modelpair* is a collision data structure which corresponds to a pair of models which are judged by the far field to be sufficiently close that they may intersect, and thus generate contacts. If a modelpair does correspond to a pair of intersecting models, then it will have an associated *contactgroup*, a dynamics data structure which contains the list of dynamics contacts corresponding to the intersection. Modelpairs are created when pairs of models are close, but not necessarily intersecting. Contactgroups, by contrast, are created only when the models in a modelpair do actually intersect.

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

Disabling and Freezing

Typically there are many more objects in a Karma simulation than are being actively processed during a given timestep. The most important optimisation is that when objects are not moving, they have the least possible impact upon performance, which requires that they be culled from the pipeline as early as possible. There are two closely related mechanisms which support this optimisation. Dynamics models which are at rest can be *disabled*, and collision models which are at rest can be *frozen*.

Being frozen is a property of a model within a far field space. To freeze a model is to inform collision that the model's transformation matrix will never change, so that a a frozen model will not be tested for collision against any other frozen models. If a model is frozen but has a dynamics body which 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.

Being disabled is a property of a body within a world. Bodies which are disabled are not simulated. 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 behaviour, but not in optimal performance.

So it is important that frozen bodies and disabled models be kept in step. Every frame, the partitioner traverses set of bodies to create partitions, which are the smallest groups of bodies such that any two bodies connected by a constraint are in the same partition. If a disabled body is found in during partitioning, it is automatically enabled. And if the enabled bodies within a partition seem to have stopped moving, they are all disabled. To inform the far field about these events, there is a callback in dynamics which forwards them to collision.

Karma Contact Generation

Karma dynamics is organised as a set of rigid bodies whose positions evolve over time according to a system of constraints, which are either joints[1] or contact groups. Joints and bodies are inserted into the simulation by the user, and persist until they are deleted. By contrast, contact groups are recreated every frame as a result of collision detection. The entire Karma collision and bridging sections of the pipeline can be seen as the process of locating contacts and parameterising them for dynamics - although of course collision detection has other uses too.

A brief summary of the contact generation pipeline is as follows:

- The collision far field space determines which models are in sufficiently close proximity to possibly generate contacts
- The bridge extracts those pairs, and invokes for each pair an intersection test which, if it succeeds, produces collision contacts. Exactly which intersection test is invoked is determined by looking up the geometry types in collision's Interaction Table.
- For each successful test, the bridge takes the set of collision contacts, looks up the material properties at the contact site in the bridge's Material Table, and generates dynamics contact groups.

The Collision Space and ModelPairs

Algorithmically the collision far field space performs gross culling, to determine which pairs of models are close enough that they might generate contacts. For efficiency reasons the space is always in one of two possible states: the *changing* state, where its contents can be updated (models moved, inserted, and deleted) but the space cannot be interrogated, and the *unchanging* state, where the space can be interrogated to find the modelpairs, but not updated. For most of the simulation, the far field is in the changing state, and can be freely updated. It is automatically moved to the unchanging state by the Karma Bridge while dynamics contacts are being generated, then automatically returned to the changing state.

The space also performs a secondary function: that of managing the modelpairs' lifecyles. Modelpairs exist in three states:

- Hello, for pairs of models which have just come into close proximity.

1.A slightly loose term, including, for example, the Karma spring constraint

- Staying, for pairs of models which have previously been in close proximity and are determined to still be so
- Goodbye, for pairs of models which have previously been in close proximity but no longer are.

Hello and Staying pairs are those for which we expect contact generation to occur. The Hello and Goodbye states allow the intersection and contact generation process to be configured, and any resources associated with the modelpair to be allocated and deallocated.

- The only special processing performed by Karma for Hello pairs is to order the models within the pair and then call the Hello callback.
- The only special processing performed by Karma for a Goodbye pair is to destroy an associated contactgroup if one exists.

When the state moves from the unchanging to the changing state, Hello pairs become Staying pairs, and Goodbye pairs are deleted. So code (such as resource allocation/deallocation) which depends on catching all modelpair state transitions must always be executed between farfield state changes.

Dynamics Contacts and Contact Groups

Successful collision intersection tests produce IntersectResult and collision contact structures. Collision contacts contain geometrical information about the contact point, such as its position, the normal vector representing the line of contact between the two bodies, and the depth of penetration of one of the bodies into the other at that point. However, they contain no dynamics information, such as for example the coefficients of friction and restitution at the contact site.

Collision contacts are converted to dynamics contacts by the Karma bridge. At each contact site there is a material for each collision model, which is, roughly speaking, a property of the collision model. The two materials together are used to index into a material table which includes all the relevant dynamics properties for the contact. Currently these parameters are copied by value into the dynamics contact structure, so that they may be changed by the application on a per-contact basis before the dynamics constraint solver is invoked to perform collision response.

Unless both models in a modelpair are frozen, collision contacts (and therefore dynamics contacts) are re-created every frame as a result of collision response. However, there are some dynamical properties corresponding to the entire set of contacts between two bodies (such as the total normal force applied) which can be useful, for example, as inputs to certain of Karma's friction model. Analogously to the way modelpairs exist for the entire duration of the time when two models are judged by the far field to be close, dynamics *contactgroup* structures exist from the first successful intersection of two models to the time when the modelpair is processed as a Goodbye pair. The contactgroup contains a list of all the contacts between the two models, together with other information which may be useful from one frame to the next.

Karma collision models which represent dynamics bodies contain pointers to those bodies, but dynamics bodies do not contain references to collision models, because it is possible (although inadvisable) to have several collision models all generating contacts on the same dynamics model. After a successful collision however, Contactgroups and modelpairs are in one-to-one correspondence, and do mutually reference each other.

Ordering and Determinism

Floating point operations are sensitive to ordering[1], so generally floating point algorithms are sensitive to the order of their inputs. However, Karma is deterministic: that is, two simulations which are indistinguishable at the API level produce identical results under identical inputs. The API mechanism to ensure that the inputs to numerical algorithms are always ordered identically despite e.g. an object being deleted then reinserted into a simulation is to provide models, joints and bodies with *sort key* fields. Regardless of the order in which bodies and constraints are inserted into the simulation, if they have identical properties and identical sort keys, Karma Dynamics will produce identical output. If an application does not supply sort keys, the simulation will, in general, not be deterministic.

1.for example, in single precision floating point (a+b)+c != a+(b+c), if e.g. a=1,b=-1,c=1e-10

For intersection tests to be deterministic, it is necessary to deterministically order the models within the model pair. For each pair of geometries there is only one intersection test, i.e. there is a test for when the first model is a box and the second a sphere, but not conversely. So if the geometries are different, the models are ordered according to which test exists. If the two geometries are identical, the models are ordered with the smallest sort key first.

Apart from the requirement on Karma Collision that intersection tests in successive runs produce identical sets of collision contacts, this imposes the requirement on the contact generation process that contact groups in successive runs will have identical sort keys. If the intersection test results in a collision contact group being created, the contact group is given the sort key

-((key1<<15)+key2)

where key1 is the sort key of the first model, and key2 the sort key of the second. Collision model sort keys should always be in the range 0 to (1<<15)-1, and dynamics body and joint sort keys in the range 0 to (1<<31)-1.

Callbacks

There are four callbacks which are involved in the contact generation process, called the *Hello*, *Intersection*, *Contact*, and *Per Pair* callbacks.

- The Hello callback is a property of the collision framework, and is called once for each modelpair, after the models have been ordered. It provides an opportunity to configure the intersection and contact generation process for the pair, for example to decide how many contacts should be generated by an intersection test
- The Intersection callback is a property of the material table, dependent on the materials of the two collision models.It is called for each pair that passes the intersection test
- The Contact callback is property of the material table, dependent on the materials of the two collision models.and is called every time a collision contact is converted to a dynamics contact. If it returns zero, the dynamics contact is removed from the contactgroup.
- The Per Pair callback is a property of the material table, dependent on the materials of the two collision models. It is called for each successful intersection, with all of the collision and dynamics contacts. If it returns zero, all contacts are removed from the contactgroup.

The contact and intersection callbacks are more convenient for most applications, but the per pair callback is more powerful, since it provides an opportunity to perform complex operations on dynamics contacts which require knowledge about all of the contacts together, using information such as the contact dimensions (face, vertex, or edge for each model) which only exists in the collision contacts.

These callbacks are all executed when the collision space is in the unchanging state. So it’s not legal to do anything in a callback which modifies the state of the space, such as removing models from it.

Karma Dynamics

Once the contacts have been generated for the frame, the dynamics solver can be invoked to perform collision response. The Karma Dynamics pipeline roughly consists of the following stages:

- Partition the scene graph into sets of bodies and constraints which can be solved separately.
- For each partition, transform the constraints and bodies from their high-level description and kinematic data into a format understood by the constraint solver
- Solve the constraints to find the forces which enforce them
- Integrate these forces on the bodies to find the new velocities and positions
- Update the bodies with their new transformation matrices, and the constraints with the forces required to generate them

In almost all simulations, the bulk of time taken by Karma dynamics is spent inside the constraint solver, Kea. The resource consumed by Kea depends on the total number of constrained degrees of freedom of the system. Consider a pair of objects. To specify their relative position requires at least 3 values, and their relative orientation a further 3 values. These 6 values are the 6 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 1 degree of freedom. A friction contact prevents penetration and also applies friction to the remaining 2 linear degrees of freedom, and so constrains 3 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, so constrain 3 degrees of freedom. The following examples illustrate this:

Car with physically modelled wheel rotation, steering and suspension.

A car can be physically modelled as 5 bodies, a chassis and 4 wheels. When the car is in mid air, there are 4 constraints. These constraints join the wheels to the car, and apply forces due to steering and suspension. They each constrain all 5 degrees of freedom, so the number of constrained degrees of freedom of the system is 4*5 = 20.

When the car is on the ground, there are 4 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 3 degrees of freedom, bringing the total number of degrees of freedom constrained to 4*5 + 4*3 = 32.

Observe that in this case, the number of constrained degrees of freedom of the car model exceeds the total number of degrees of freedom. Such systems are *overconstrained*: they do not typically have exact solutions, but Kea generates approximate solutions for them.

Thrown box hitting a wall

A box flying through the air has no constrained degrees of freedom. Suppose the box hits a wall. There are three possibilities

- a corner of the box hits the wall. There is only one contact, at the corner.
- an edge of the box hits the wall. In order to prevent the edge penetrating, at least two contacts are required, which are typically at the extremes of where the edge intersects the wall
- a face of the box hits the wall. Now three contacts are required to prevent any of the face penetrating. Any three points which are not colinear will do, although the response is usually better the further apart they are.

In the worst case, 3 contacts will be required to stop the box going through the wall. If 3 friction contacts are used, the number of degrees of freedom constrained will be 4*3 = 12. However, it is usually not necessary to model the friction of the wall unless you want to rest the box against the wall, so you could use 3 frictionless contacts, constraining only 3 degrees of freedom.

Human falling down stairs

Consider a human falling down a staircase. A simple model of a human could consist of 10 bodies: a head, a torso, 2 forearms, 2 upper arms, 2 thighs and 2 calves/feet. 9 limited ball and socket joints are used to hold them all together and provide joint limits and muscle modelling. This constrains 3*9 = 27 degrees of freedom. In the worst case, 27 contacts are required to prevent the human falling through the staircase. The total number of degrees of freedom constrained is 3*9 + 3*27 = 108

The amount of space required by Kea is proportional to the square of the number of constrained degrees of freedom, and this is also one of the factors in the amount of time it requires. Thus it makes sense wherever possible to minimise the number of degrees of freedom, by splitting the simulation into pieces which are disconnected and so can be solved separately. Beyond that, Karma is capable of automatically simplifying a system if its space requirements exceed a bound set by the application.

Partitioning

A partition is a set of bodies and constraints for which the forces which enforce those can be solved without reference to any other bodies and constraints. An alternative definition is the smallest set of bodies such that if a body in the partition shares a constraint with another body, that body is also in the partition.

Karma dynamics maintains a list of enabled bodies, and to find each partition, it finds the first body in the list not in any partition, and adds all bodies which are reachable by traversing joints in a breadth-first fashion. If any such body is disabled, Karma automatically enables it as soon as it is found.

When the partition is complete, Karma checks to see whether it can be disabled. This is controlled by five parameters which are properties of the world: thresholds on linear and angular velocities and accelerations, and the *keepalive* value. A partition can be disabled if for every body in it:

- the linear and angular accelerations and linear and angular velocities have fallen below threshold values which are properties of the world
- the body has been enabled for at least *keepalive* frames.

If these conditions hold, every body in the partition is disabled, and corresponding collision models frozen.

If the partition remains enabled, then, if matrix size limiting is enabled the partition is simplified in order to bring it down to the required size. The size of the input to the constraint solver can be measured in *rows*, roughly a measure of the degrees of freedom of the system which are affected by the constraints. The simplification is as follows:

Contacts with a two-dimensional friction model are simplified to a one-dimensional model.

If there are still too many rows, contacts with one-dimensional friction are simplified to be frictionless

If there are still to many rows, contacts are deleted in increasing order of importance. Let $group(c)$ be the contactgroup of a contact c, and $depth(c)$ be the order of depth within a contact group, so that $depth(c) = 1$ for the shallowest contact and $depth(c) = |group(c)|$ for the deepest. Let$world(c)$ be 1 if the contact is with the world, and 0 otherwise.

Then the importance of a contact is

$$\left(\frac{depth(c)}{|group(c)|} + world(c)\right) \times 100$$

So shallower contacts and contacts from larger contactgroups are deleted first, and contacts with the world are deleted last.

Solving Constraints

Standard rigid body dynamics algorithms fall roughly into 3 categories. These are penalty methods, Mirtich-style methods and LCP based methods.

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.

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.

LCP based methods

The most common LCP 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. Baraff's method has a number of speed and stability issues, which have been the subject of considerable research since its initial publication.

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 time step 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. Stability results which are not enjoyed by Baraff's method have been proven for timestepping methods.

Kea, is a partially implicit time-stepping LCP method, ‘partially implicit' meaning that constraint forces, such as non-penetration forces and joint forces are implicitly integrated to satisfy the constraints at the end of the time step, but external forces such as gravity are explicitly integrated. Implicit integration means that the system is extremely stable with regard to stiff forces (those which can undergo very large changes from small changes in position) so long as those forces arise from constraints, rather than being applied directly. So, for example, Kea can stably simulate stacks and piles of arbitrary objects, without the need for user tuning or damping.

Kea constraints fall into two classes, inequality constraints and equality constraints.

- Equality constraints, such as joints, simply specify that two quantities - the position of a point on one body and the position of a point on the other, say - are equal.
- Inequality constraints specify that a quantity, such as the force supplied by a force-limited motor, has an upper or lower limit. Inequality constraints can be required to satisfy the *complementarity* condition that of two quantities coupled by the condition, both are non-negative, and either one or the other is zero. For example, the complementarity condition for a typical contact specifies that the separation velocity and non-penetration force in the normal direction are both non-negative, 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 applied to enforce non-penetration, or there will be a contact force that prevents the bodies separating and the velocity of the objects towards each other will be zero at the contact point.

There is no general procedure for efficiently solving sets of inequality and complementarity constraints: in the worst possible case, solution requires time exponential in the number of inequality constraints, although the constraint sets which arise from physical systems very rarely approach this degree of intractability. Nonetheless, it is important to be able to trade simulation fidelity for speed. There are two possible mechanisms which an application can use to accomplish this:

- relax the conditions on inequality constraints to widen their bounds, so that a generated solution is more likely to satisfy the constraints
- limit the number of iterations Kea uses an iterative procedure to generate successive solution candidates

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 the application can tolerate.

Far Field Collision

The Far Field API

The purpose of far field collision is to cull the possible $O(n^2)$ collisions between a pairs from a set of models down to a small number of candidates. Far field performance is sensitive to the application, but the Karma far field is designed to be used with a large number of objects whose positions undergo relatively small changes between timesteps. It is not particularly optimised for ray collisions, or for the case where the set of simulated objects often changes radically. The McdSpace object is Karma's far field implementation.

The implementation of McdSpace is as a frame-coherent axis-aligned sort of the extents of the bounding boxes of the models. It can operate on any or all of the x, y, and z axes. Two models are considered to be in close proximity if their axis-aligned bounding boxes overlap on the axes on which the `McdSpace` is operating. Each collision model also has a small contact tolerance value, by which the radii of the bounding box are extended. Bounding boxes are considered to include their faces, edges, and corners, so that two bounding boxes which intersect exactly at a corner are considered to be in close proximity.

Creating and Destroying an McdSpace

Collision spaces can be created with the following function:

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

The axes argument is a bit field which specifies which axes of test. Its possible values are a combination of `McdXAxis`, `McdYAxis`, and `McdZAxis`*,* or the value `McdAllAxes` which is equal to `(McdXAxis+McdYAxis+McdZAxis).` *`ObjectCount`* is the maximum number of models which can be inserted into the space, and *`pairCount`* the maximum number of modelpairs the space can hold.

Destroying an McdSpace is straightforward, using the function

```
McdSpaceID MEAPI McdSpaceDestroy( McdSpaceID s);
```

However, if the McdSpace is being destroyed, but collision and dynamics are continuing, it's important that any resources which have been allocated for modelpairs be deallocated. One way to ensure this is to remove all the models from the space, call `McdSpaceUpdateAll()` to generate the goodbye pairs, call `MstBridgeUpdateContacts()` to deal with them, then destroy the space.

Inserting and Removing Models

An McdModel can only be in a single McdSpace. Models are inserted into a space using the function

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

which returns true if the model is already in the space, or if it is successfully inserted. If the space is already full, or the model is in a different space, it returns false. Inserting the model into the far field merely registers it for later update and testing, it does cause the far field to update its internal data structures regarding the model. The model is inserted unfrozen, and if you wish to immediately freeze it, you should first call `McdSpaceUpdateModel.`

The function

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

removes the model from the space. It returns false if the model is not in the space.

Modelpairs cannot be directly deleted, since this would interrupt their role in the allocation anf deallocation of resources corresponding to nearfield tests. Removing a model from the space causes any modelpairs pointing to the model to enter the Goodbye state. Karma does not require the presence of the collision models in order to deal with Goodbye modelpairs, however it does attempt to delete the pointer to the contactgroup in the modelpair, if one exists. If you are destroying the body corresponding to a model, and consequently destroying its associated contactgroups, you should sever the dangling reference by setting the pointers in the modelpairs to `NULL`.

Updating Models

McdSpace uses an explicit update model: that is, on every timestep the state of each far field corresponding to each model needs to be updated. The function

```
void MEAPI McdSpaceUpdateModel( McdSpaceID space, McdModelID model)
```

updates an individual model, so long as the model is not frozen. The function

```
void MEAPI McdSpaceUpdateAll( McdSpaceID space, McdModelID model)
```

updates all models which are not frozen.

Updating a model causes the function

```
void MEAPI McdModelUpdate(McdModelID model)
```

to be called on the model, which performs the following calculations: if the model has a relative transform, the compound transform is updated, along with the transforms of any sub-geometries if the geometry of the model is an aggregate or composite. The bounding box of the model is then recalculated and cached in the model, as are the bounding boxes of any sub-geometries.

If you update a transformation matrix whose pointer is stored in a model which is not contained in a space, you should call this function yourself to perform these updates before invoking any nearfield test on the model.

Freezing Models

Collision models which 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 by `McdSpaceUpdateAll`. `McdModel` objects are by default in *unfrozen* status. The function `McdModelUpdate()`, which updates relative transforms and bounding boxes will not be called on a frozen model.

Freezing and unfreezing of models are accomplished via the API functions

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

These functions always return true. To check the status of a model:

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

Note that `McdSpaceFreezeModel()` does not imply a call to `McdSpaceUpdateModel()`. If you wish to freeze a collision model transform which has just been inserted or which has changed since it was last updated, be sure to call `McdSpaceUpdateModel()` beforehand.

Freezing models is only an optimisation. It does not change the set of modelpairs returned by the McdSpace.

Disabling and Enabling Pairs of Models

The function `McdDisablePair()` prevents a pair of collision models, `m1` and `m2`, from appearing in the output pairs of a `McdSpace`, and so prevents any collision between the two models. You can disable and enable pairs, and test whether a pair is enabled, with the API functions

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

If `m1` and `m2` are the same model, these will all return `false.`

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. If `McdSpaceEnablePair()` is subsequently called on the same pair of models and the models are in close proximity to each other, the pair will reappear as a new *hello* `McdModelPair` on the next call to `McdSpaceGetPairs()`.

Whether a pair of models is disabled is a property of both the models and the space. When a model is inserted into a space, collision is enabled with all other models.

Querying the Far Field

Querying the far field results in a set of modelpairs being rendered into an McdModelPairContainter. The function

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

creates a container with room for `size` models. Since there may be more modelpairs than will fit in the container, an McdSpacePairIterator is used to keep track of the state of the query. The function

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

initialises an McdSpacePairIterator. Once the iterator has been initialised pairs can be obtained with calls to

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

This function fills the container, updates the iterator, and returns true if there are any more modelpairs to be read. The function

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

is similar, but returns only *hello* and *goodbye* pairs. Between calls to these functions

```
void MEAPI McdModelPairContainerReset( McdModelPairContainer* a);
```

should be called to reset the container to its empty state. An example of this protocol can be found in the source to the function `MstBridgeUpdateContacts`.

Iterating Over Models

In a similar fashion, you can iterate over models by initialising an McdSpaceModelIterator

```
void MEAPI McdSpaceModelIteratorBegin( McdSpaceID s,
       McdSpacePairIterator* iter);
```

and then retrieve successive models using

```
MeBool MEAPI McdSpaceGetModel( McdSpaceID s,
       McdSpacePairIterator* iter, McdModelID *modelp);
```

You can obtain a count of the models in a space using

```
int MEAPI McdSpaceGetModelCount( McdSpaceID s);
```

Far Field States

An McdSpace has two states: changing and unchanging. When the space is in the changing state, it can be updated, but not queried, and when it is in the unchanging state, it may be queried but not modified. Transitions between these two states are accomplished with the following API functions:

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

The state of the space may be ascertained using the API function

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

`McdSpaceBeginChanges()` also 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()`; *hello* pairs from the previous step will reappear either as *staying* or *goodbye* pairs; *staying* pairs can reappear again as *staying* pairs or become *goodbye* pairs, if they are no longer in close proximity.

`McdSpace` functions available only when `McdSpaceIsChanging()` is true:

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

`McdSpace` functions available only when `McdSpaceIsChanging()` is false:

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()`

Functions Valid Anywhere
`McdSpaceSetUserData()`
`McdSpaceGetUserData()`

Line Segment Queries

Two utility functions from `McdSpace` are available to perform ray intersection tests. To find all the points of intersection between a directed line and geometries, use

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

This finds all the intersections of an oriented line segment with all models in `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 intersection that will be reported. The return value is the number of intersection results.

- 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. No intersection is reported where the line exits 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.

To find just the first point of intersection between a directed line and geometries, use

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

The `McdLineSegIntersectResult` structure has the following format:.

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

The `McdSpaceGetLineSegFirstEnabledIntersection()` function is identical to the `McdSpaceGetLineSegFirstIntersection()` function, but it also allows you to provide a callback function to selectively exclude some models from the query. For example, for a line of sight application, you may want to exclude models that are invisible or transparent

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

Finds first intersection of an oriented line segment with selected models in `space`, 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 `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. Returns 1 if an intersection was found, otherwise 0. See notes in the `McdSpaceGetLineSegIntersections()` function..

Potential Pitfalls

Error Handling

The number of possible modelpairs corresponding to n models is

$$\frac{n(n-1)}{2}$$

However, it is usually not feasible to allocate space for this many modelpairs. It may be possible from the geometry and types of models in your application to compute an absolute upper bound on the number of modelpairs and allocate sufficient space, but generally this is not the case.

If the pool from which modelpairs are allocated is full, no modelpair will be created for the models, and instead a callback will be invoked to enable the application to deal with the problem. This callback has the type

```
void (MEAPI * McdSpacePoolErrorFnPtr)( McdModelID m1, McdModelID m2)
```

You can read and write the value of this callback with the functions

```
void MEAPI McdSpaceSetPoolErrFnPtr(McdSpaceID space,
                                   McdSpacePoolErrFnPtr handler);
McdSpacePoolErrFnPtr MEAPI McdSpaceGetPoolErrFnPtr(McdSpaceID space);
```

Reinserting Models

If the far field is in the changing state, and you remove and then reinsert a model, the far field is unaware that it is the same model. In this circumstance, if you insert the model in the same place where it was before, for every other model which overlaps this model both a Hello and a Goodbye pair will be produced. Since Karma disposes of Goodbye pairs simply by removing any associated contactgroups, this causes no internal problems, but if your use of the far field extends beyond letting Mst handle the modelpairs, keep this possibility in mind.

Constraints: Joints and Contacts

Overview

The interaction of objects through constraints, such as joints or contacts, is dealt with in this chapter through the use of the high level Karma Dynamics API. An articulated body is comprised of two or more jointed bodies.

By partitioning a world into a subset of objects that interact only with other objects in the sub-set, fast and efficient simulations can be created.

Constraints

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 effects it’s motion. Constraints may be used to avoid having to model the detailed interactions - the large scale effect is dealt with, without worrying about the underlying physical model. 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 material comprising the cup. 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 is called the normal force.

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, the ball and socket or spherical joint, strut joints, etc. 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, 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. Finally, what are known as either *equality* or *inequality* constraints, can be set up. This can become complicated, but it is the purpose of the Karma Dynamics library to deal with all the details.

Constraints are a very powerful modelling tool. Karma Dynamics provides an API through the Mdt Library for a selection of useful constraint types.

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 Mdt Library.

- Ball and Socket or spherical.
- Hinge or revolute.
- Prismatic or slider.
- Universal.
- Angular3.
- Car Wheel.
- Linear1.
- Linear2.
- Fixed Path.
- Fixed Position Fixed Orientation (FPFO). Deprecated in favor of RPRO (below).
- Relative Position Relative Orientation (RPRO).
- Spring.
- Cone Limit constraint.

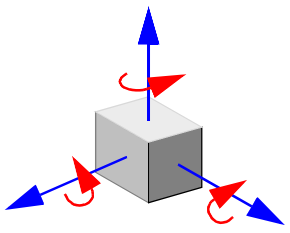
These joints can be used to attach two simulated objects to each other - that is, their relative position or orientation (or both) is constrained. Although the discussion below refers always to attached bodies, these joints may be used to attach a single object to the inertial reference frame (world) by not specifying a second attached body (or 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 unrealistic structures.

limits (stops) and *actuation* (motors) that can be applied to Hinge and Prismatic joints are discussed in this chapter.

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.

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.
- `FPFOJoint`, the Fixed-Position-Fixed-Orientation joint.
- `RPROJoint`, the Relative-Position-Relative-Orientation joint.
- `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:

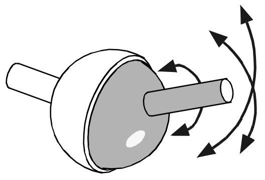
```
MdtWorldID MEAPI MdtConstraintGetWorld( const MdtConstraintID constraint )
```

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);
```

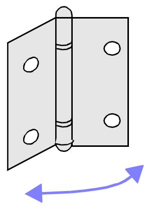
creates a ball and socket joint in an `MdtWorld world`. To use the joint to constrain a pair of objects (`body1` and `body2`) use;

```
    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 $-n\pi$ through $n\pi$ 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:

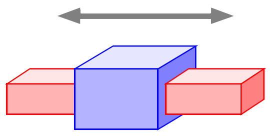
```
    void MEAPI MdtHingeSetLimit           ( const MdtHingeID joint,
                                            const MdtLimitID NewLimit );
```

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`:

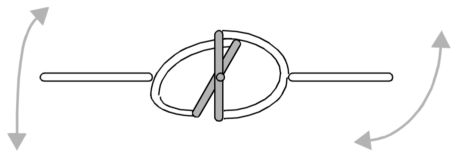
```
    void MEAPI MdtPrismaticSetLimit       ( const MdtPrismaticID joint,
                                            const MdtLimitID NewLimit )
```

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.

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 MdtUniversalGetAxis        ( const MdtUniversalID joint,
                                        const unsigned int bodyindex,
                                        MeVector3 axis )
```

The joint axis corresponding to `bodyindex` is returned in the vector `axisVec`.

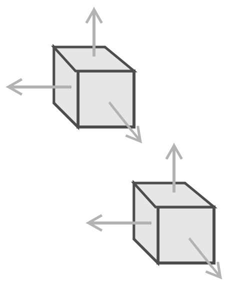
Mutators

The mutator function specific to `Universal` is:

```
void MEAPI MdtUniversalSetAxis         ( const MdtUniversalID joint,
                                         const unsigned int bodyindex,
                                         const MeReal Ax,
                                         const MeReal Ay,
                                         const MeReal Az )
```

Set the joint axis corresponding to `bodyindex`, to (`xAxis, yAxis, zAxis`) in world coordinates.

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.

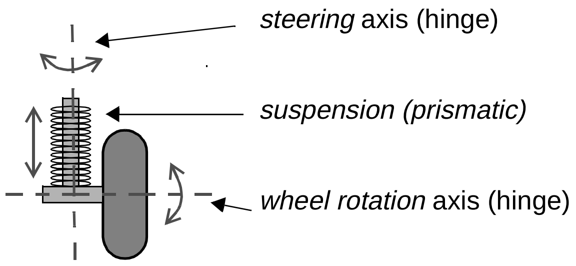
```
    void MEAPI MdtAngular3SetStiffness     ( const MdtAngular3ID j,
                                             const MeReal s )
```

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 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 MdtCarWheelSetHingeAxis( const MdtCarWheelID joint,
            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 MdtCarWheelSetSteeringAxis( const MdtCarWheelID joint,
                      const MeReal x, const MeReal y, const MeReal
z );
```

Set the wheel joint steering axis.

```
void MEAPI MdtCarWheelSetSteeringLimitedForceMotor(
                           const MdtCarWheelID joint,
                 const MeReal desiredVelocity, const MeReal
forceLimit );
```

Set the limited force motor parameters of a car wheel joint.

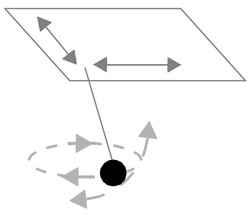
```
void MEAPI MdtCarWheelSetSteeringLock( const MdtCarWheelID joint,
                                         const MeBool lock
);
```

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 );
```

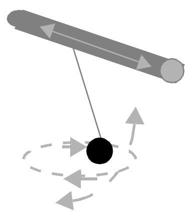
Set the suspension parameters.

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.

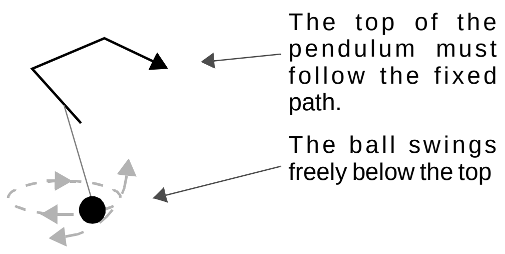
Mutators

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);
```

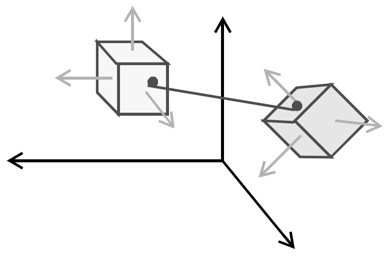
Set the fixed path joint velocity with respect to one of the constrained bodies. This joint velocity is set in the bodies' reference frames. The reference frame is determined by the fifth parameter, `bodyIndex`.

Fixed-Position-Fixed-Orientation Joint: MdtFPFOJoint

This joint is deprecated. The Relative-Position-Relative-Orientation joint replaces some of the intended use cases for this joint.

A fixed position, fixed orientation joint is a totally fixed joint that removes all six degrees of freedom, hence all relative motion, between the connected bodies. Because of this it is computationally costly and is not recommended for connecting two dynamic bodies to make a single dynamic body.

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.

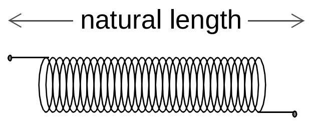
```
void MEAPI MdtRPROJointSetLinearStrength( const MdtRPROJointID
joint,
const MeReal sX,
const MeReal sY,
const MeReal sZ );
```

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.

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*.

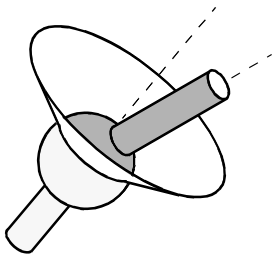
```
void MEAPI MdtSpringSetDamping( const MdtSpringID joint,
const MeReal NewDamping )
```

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 stiffness 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.

How to Use Joints

The tutorial *Hinge* provided in the release is a simple example of a hinge joint powered by a limited-force motor. A body is connected to the world with the hinge joint.

The limits can either be *hard* (if the limit stiffness factor is high) or *soft*. In terms of the simulation, a hard bounce reverses a bodies' angular velocities in a single timestep, while a soft bounce may take many timesteps to reverse a bodies' 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 between zero and one to govern the loss of angular momentum as the bodies rebound.

First, in the `main()` function, create a hinge joint in the world. When the hinge exists set its center of mass position at the origin and the orientation of its axis parallel to the `x` axis. By default `body[1]` is set to `0` i.e. the world.

```
hinge = MdtHingeCreate(world);
MdtHingeSetBodies(hinge, body, 0);
MdtHingeSetPosition(hinge, 0, 0, 0);
MdtHingeSetAxis(hinge, 1, 0, 0);
```

Get a handle to its `MdtLimit` structure and use it to toggle its limit and the limit’s motor.

```
MdtLimitID limit = MdtHingeGetLimit(hinge);
MdtLimitActivateMotor(limit, !MdtLimitIsMotorized(limit));
MdtLimitActivateLimits(limit, !MdtLimitIsActive(limit));
```

Then, by accessing directly the sub structures `MdtSingleLimit` of the limit structure, set the lower and upper limit angles of the hinge. Set the maximum force to a value `maxForce` to give a desired velocity of a value `desiredVelocity.` Note that the maximum force may not be strong enough to reach the desired velocity.

```
MdtSingleLimitSetStop(MdtLimitGetLowerLimit(limit), LO_LIMIT);
MdtSingleLimitSetStop(MdtLimitGetUpperLimit(limit), HI_LIMIT);
MdtLimitSetLimitedForceMotor(limit, desiredVelocity, maxForce);
```

Finally, *enable* the hinge joint, allowing it to be computed at the next update.

```
MdtHingeEnable(hinge);
```

The main difference between contacts and joints is that joints are most often created once only, as in *hinge* in its `main()` function, but contacts must be created and destroyed repeatedly, either by a user created function, as in *Bounce*’s callback function `tick()`, or by using Karma Collision and the Karma Simulation Toolkit in conjunction with Karma Dynamics.

Creating Contacts: MdtContact

Every time two bodies touch or intersect each other, a *contact* must be created. The contact constraint will ensure that there is no relative motion in the direction opposite to the contact normal. The solver will compute the constraint force required to prevent the bodies from moving against the direction of the normal as well as the tangential friction forces that correspond to Karma's approximation of the Coulomb friction model. The type of friction used, the computed forces exerted by a contact on the bodies, and the position and normal of the contact, form part of the information stored in a `MdtBclContactParams` structure. This is pointed to by an `MdtContactID` handle.

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 sphere, one contact should suffice. Resting and sliding usually require three contacts.

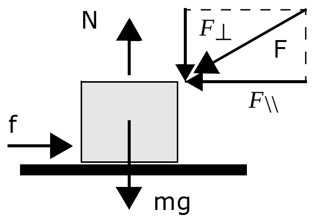
Karma Collision offers a number of alternative *contact strategies* that produce contacts for different types of situation that can be selected as necessary. Strategies that use fewer contact points have the advantage that MdtKea solves them more quickly. For those using a third-party collision package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

Simulating Friction

After the contact configuration of a set of bodies is determined, the behavior of contact points must be specified. Up to three forces may be present at a contact point.

These forces act on the bodies that are in contact. The first is a normal force that prevents the bodies from penetrating each other. The other two are tangential friction forces (that act at right angles to one another and to the normal force) that prevent the bodies from sliding against each other along the contact plane.

Coulomb Friction



Friction is a term used to describe the macroscopic effect of the resistance to motion that a body experiences. The origin of friction lies in the effect of physical processes that takes place between materials on the atomic scale. It is the electrostatic interaction between atoms and molecules comprising a material that gives rise to friction. Friction is a dissipative process that converts energy from motion to heat and sound.

Friction can conveniently be categorised 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.

What follows is a simple description of how Coulomb friction works. Imagine that the bodies in contact are a box and the ground: the box is sitting on the ground. The normal force $\vec{N}$ is the force exerted by the ground in reaction to the gravitational field strength $m\vec{g}$ plus the perpendicular (relative to the ground) component of the total external force, $\vec{F}_{\perp}$, exerted upon the object. : These forces must be equal to prevent any movement perpendicular to the ground. The force $\vec{f}$ is the force of friction that spontaneously opposes the tangential (parallel to the ground) component of the external force, $\vec{F}_{\backslash\backslash}$.

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

As soon as the tangential component of $\vec{F}$ becomes larger than $f_{s\,max} = \mu_s N$, the frictional force fails to scale up with $\vec{F}_{\backslash\backslash}$ and the force imbalance will cause the box to start sliding. Once the box is in motion the frictional force $\vec{f}$ still impedes the movement of the box but to a lesser degree. This new frictional force is called *kinetic* friction and is written $f_k = \mu_k N$ where μ_k is the dynamic coefficient of friction and, in general, $\mu_k < \mu_s$. In short, the two possible cases are:

- At rest => static friction: $f \le f_{s\,max} = \mu_s N$
- In motion => kinetic friction: $f = f_k = \mu_k N$

Two quantities, the normal force $\vec{N}$ and the friction coefficient μ, are thus directly related to the severity of the friction force. The normal force $\vec{N}$ is related to the weight of the object and to the forces exerted upon this object: the larger $\vec{N}$ 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 contact of the surfaces is and the lesser 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.

Friction in MdtKea

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.

Out of the several approximations available, Karma uses one that is both stable and efficient. However, anisotropy in the friction force may be observed. This can be overcome by reorienting the friction box. A friction model based on the normal force is included (see bullet 4 below) in this version of Karma.

Karma Dynamics supports three main friction modes:

- Frictionless

This means that no tangential friction force is applied at all, so the contacting objects are free to slide over each other. This is equivalent to setting $f_{smax} = f_k = 0$.

- 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 $f_{smax} = \infty$.

- Box 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 $f_{s max}$ to a constant value.
- Normal Force Friction

  Normal Force friction is a refinement of box friction where at each timestep 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 timestep. For bodies in resting contact, this adaptively configures box friction to provide a more faithful approximation to Coulomb friction.

Slip: An Alternate Way to Model Friction

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.

In Kea’s 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 (`slip1` and `slip2` are defined in the `MdtBclContactParams` structure), 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 $f_{s max} = f_k = 0$ and $\vec{V} \propto \vec{F}_{\backslash\backslash}$ where $\vec{V}$ and $\vec{F}_{\backslash\backslash}$ are the tangential component of the velocity and of the external force respectively.

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 behaviour. 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.

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

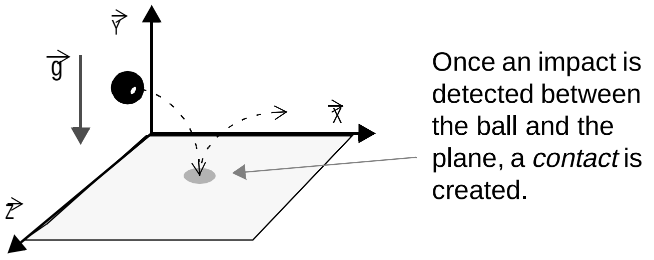
Member	Default Value
MeReal restitution	0.0
MeReal velThreshold	0.001
MeReal softness	0.0
MeReal max_adhesive_force	0.0
MeReal slip1/slip2	0.0
MeReal slide2/slide2	0.0

How to Use Contacts: Bounce.c

For simple programs (such as ones showing sphere-sphere, sphere-plane, or box-plane collisions), collision detection can be carried out with a few lines of code. But for anything more complex, a collision detection package such as Karma Collision may be needed.

After detecting a collision, Karma Dynamics uses contact constraints to determine the appropriate collision response. Contact structures (`MdtContact`) can be created and enabled using functions in the Mdt Library. When using Karma Collision and the Karma Simulation Toolkit, Karma will do both for you. For more information about Karma Collision, see the *MathEngine Karma Collision Developer's Guide*. For more information about the Karma Simulation Toolkit, see the *Mathengine Karma Simulation Toolkit Developer's Guide*.

In the Karma tutorial `Bounce.c`, a straightforward use of contacts is demonstrated - Karma Collision is not used in this example. A ball is thrown onto a surface and rebounds each time it touches the surface, with consecutive smaller amplitudes. To achieve this effect, a callback function `called tick() that is` called by MeViewer is responsible for checking for physical intersection between the ball and the surface. Each time an intersection is detected a contact must be created. It is important to remember that a contact, unlike a joint, is not a permanent constraint, since it is created dynamically whenever two objects come into physical contact, and is destroyed when the contact is no longer needed.


$\vec{Y}$
$\vec{g}$
$\vec{X}$
$\vec{Z}$
Once an impact is detected between the ball and the plane, a *contact* is created.

First, the `tick()` function checks for a previously created contact. An old contact contains dated informations that is useless and takes space in memory. It should be destroyed as soon as the two objects are no longer in contact:

```
if (contact){
     MdtContactDisable(contact);
     MdtContactDestroy(contact);
     contact = 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.`

To determine if intersection between the ball and the plane took place the position of the ball relative to the plane (positioned at y=0) and the radius of the ball are needed.

```
MdtBodyGetPosition(body, pos);
penetration = 0 - (pos[1] - ballRadius);
```

If positive, the value of `penetration` is the distance of penetration of the ball through the plane. If it is negative there is no physical intersection between the two. A contact constraint is created if there is an intersection. The first body in the contact is the ball, the second is the world (identified as `0`) since the plane is static and attached to the world.

```
    if (penetration > 0){
        MdtContactParamsID params;
        contact = MdtContactCreate(world, body, 0);
```

A contact can only be created through the function `MdtContactCreate()`. First create a `MdtContactID` variable called `contact` that will point to the `MdtContact` structure where all information about that contact will be stored.

The 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 position of the contact must be specified in world coordinates and is set to the value of the ball center of mass translated downward by the value `ballRadius`. The normal of the contact is the plane's normal, which is the y-axis. To compute the rebound amplitude, the penetration depth must be communicated to Karma dynamics.

```
        MdtContactSetPosition(contact, pos[0], pos[1] - ballRadius, pos[2]);
        MdtContactSetNormal(contact, 0, 1, 0);
        MdtContactSetPenetration(contact, penetration);
```

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`.

First obtain a `MdtContactParams` structure related to the contact

```
        params = MdtContactGetParams(contact);
```

Formal description:

```
MdtContactParamsID MEAPI MdtContactGetParams ( const MdtContactID
contact );
```

Return a `MdtContactParamsID` pointer to the contact parameters of this contact.

When a `MdtContactParams` structure exists, assign contact properties to it, such as the friction type, the friction value and restitution.

```
        MdtContactParamsSetType(params, MdtContactTypeFriction2D);
        MdtContactParamsSetFriction(params, (MeReal)(5.0));
        MdtContactParamsSetRestitution(params, (MeReal)(0.6));
    }//end if(penetration>0)
```

Finally, attach this new contact to the world before updating the world by one timestep. Remember that if the contact is not enabled before the world is updated, the Kea solver will ignore it.

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

The `MdtContactEnable()` and `MdtContactDisable()` are not functions, but macros that were implemented to save writing two lines instead of one.

Convert a contact or a joint to a constraint ID to enable or disable it, as shown below:

```
     MdtConstraintID cid = MdtContactQuaConstraint(contact);
```

and then

```
     MdtConstraintEnable(cid);
     MdtConstraintDisable(cid);
```

Another way to do this is:

```
     MdtContactEnable(contact);
     MdtContactDisable(contact);
```

mathengine

Geometrical Types and Their Interactions

Overview

Karma Collision offers many concrete geometrical types to choose from when defining the shape of your collision models. The geometry you choose 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 simpler algorithms that operate on them. Having a wide selection of geometry types available gives you a lot of flexibility in choosing trade-offs between performance, memory and geometrical accuracy.

You specify the geometrical shape of a collision model using

```
McdModelID McdModelCreate( McdGeometryID myGeometry);
```

`myGeometry` will refer to a geometrical object you have created such as

```
McdSphereID myGeometry = McdSphereCreate(1);
```

or

```
McdBoxID myGeometry = McdBoxCreate(1,2,3);
```

An `McdGeometry` object defines a 3D shape in a local coordinate system. The `McdModel` that is created from it holds a transform that defines the position and orientation of that shape in the global coordinate system.

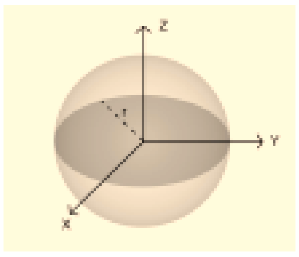
Geometrical Primitives

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:

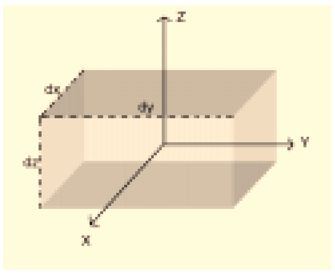
Sphere

Ball
Particle
Planet
Head



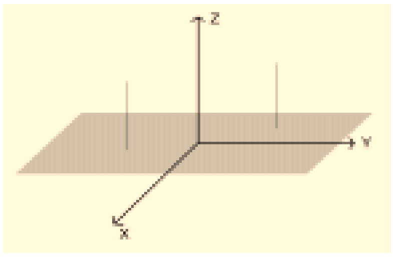
Box

Book
Skyscraper
Gambling Dice
Metal Bar



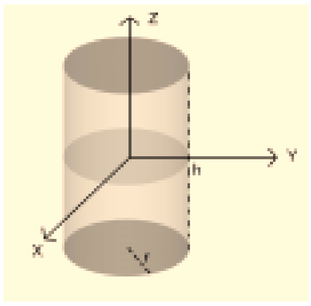
Plane

Floor
Wall
Ceiling
Ship-Deck



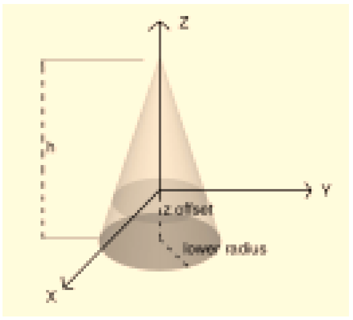
Cylinder

Wheel
Disk
Stick
Gun Barrel
Oil barrel



Cone

Tree Trunk
Post
Lamp Shade
Field of Vision
Field of Illumination



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:

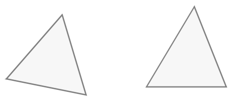
- z is the special axis[1], if there is one. This applies to cylinder, cone and plane.
- The (uniform density) center of mass is placed at the origin. This is convenient and efficient for working with dynamics.

Note that the first convention produces a somewhat unintuitive positioning for the cone primitive: the base of the cone does not coincide with the z=0 plane, but lies a bit below it. The function below will return the magnitude of this offset.

```
McdConeGetZOffset( McdConeID );
```

Triangle List

Terrain



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 far field 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.

NOTE: In Karma 1.0, the transformation matrices of models with TriangleList geometries were interpreted differently in different parts of the code. In Karma 1.1, they are interpreted as model-world transformations, in common with all other geometries

```
McdTriangleListID MEAPI McdTriangleListCreate(McdFramework *frame,
                                              MeReal dx,
                                              MeReal dy,
                                              MeReal dz,
                                              McdTriangleListFnPtr f);
```

Create a `TriangleList`, whose bounding box radii are `x, y,` and `z,` and whose callback function is `f`.

1.A special axis is an axis about which there is a symmetrical distribution of volume.

The query function takes as parameters a bounding sphere position and radius.

```
typedef int (MEAPI * McdTriangleListFnPtr) (McdModelPair* modelTriListPair,
                                            MeVector3 pos,
                                            MeReal radius);
```

It should set a pointer in the triangle list structure to point to an `McdUserTriangle` array, whose vertices and normal are specified in the geometry's local coordinates, and then return the number of triangles in the array. Karma assumes that the normal is calculated in a right-handed fashion, that is, if the triangle vertices are v_0, v_1, v_2, then the triangle normal is in the direction $(v_1-v_0)x(v_2-v_1)$. This means that if you want your triangle to be one-sided (and for terrain, you usually do) you must take care to insert the edges in the correct order in the `McdUserTriangle` structure.

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 which allows combinations of the following values:

kMcdTriangleUseSmallestPenetration	Use the normal which minimises translational distance required to separate the object and triangle. If 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 behaviour equivalent to Karma 1.0)

The examples ***Tank*** and ***TriangleList*** demonstrate the use of triangle lists.

Registering Geometrical Types and Intersection

An application using only primitive geometrical types can be configured by:

```
void McdInit( McdPrimitivesGetTypeCount());
void McdPrimitvesRegisterTypes();
void McdPrimtivesRegisterInteractions();
```

The selection can be narrowed to a smaller subset using:

```
void McdInit( 3 );
void McdSphereBoxPlaneRegisterTypes();
void McdSphereBoxPlaneRegisterInteractions();
```

Finally, you can select types and interactions on an individual basis, using calls such as:

```
void McdBoxRegisterType();
void McdBoxBoxRegisterInteraction();
```

The selective registration means that code for types and interactions that are not registered will not be loaded into the executable program.

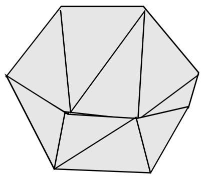
A recommended practice is to use, during initial development, the pair of registration functions that register all primitives and their interactions, then replacing this with a finer-tuned selection as it becomes clear exactly which subset is needed.

Non-primitive Geometrical Types

Non-primitive geometrical types allow you to define more general classes of shapes for your models. They are specified by a variable number of parameters. The number is large for complex models and small for simpler models. You can choose any level complexity, based on a trade-off between memory use and geometrical accuracy. This flexibility is due to the fact that Karma's non-primitive geometry types do not represent curved surfaces exactly, but approximate them by a set of discrete surface elements.

The following section is meant as an overview of the non-primitive types. For a more in depth coverage of these types and their API's, please consult the Karma Collision reference manual.

ConvexMesh



The ConvexMesh geometry type allows you to specify a geometry representing a closed convex object .

You must first register the convex mesh geometry type with the system, and then register any interactions you want to be in effect. For example, if you want to use both convex meshes and primitives in an application, you would use:

```
McdConvexMeshRegisterType();
McdConvexMeshPrimitivesRegisterInteractions();
```

The last registration function registers all interactions available that involve convex mesh. Currently, the convex mesh type can interact with some, but not all, primitive geometry types.

The example program `ConvexStairs.c` shows how a ConvexMesh geometry type is created:

The easiest way to create a convex mesh geometry is from a set of points representing the vertices. A convex hull will be computed from this:

```
McdConvexMeshID MEAPI McdConvexMeshCreateHull( McdFramework *frame,
                                   MeVector3* vertices, int vertexCount,
                                                  MeReal fatnessRadius );
```

Create new convex polyhedron object from its vertices. Computes the convex hull polygons and edges.

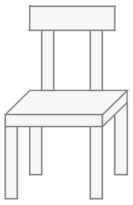
```
#define r1     (0.35f)
MeReal vertices1[12][3] = { { r1, 2*r1, r1},
                            {-r1, 2*r1, r1},
                            {-r1, 2*r1, -r1},
                            { r1, 2*r1, -r1},
                            { 2*r1, 0, 2*r1},
                            {-2*r1, 0, 2*r1},
                            {-2*r1, 0, -2*r1},
                            { 2*r1, 0, -2*r1},
                            { r1, -2*r1, r1},
                            {-r1, -2*r1, r1},
                            {-r1, -2*r1, -r1},
                            { r1, -2*r1, -r1} };

McdGeometryID geoPrim;

geoPrim = (McdGeometryID) McdConvexMeshCreateHull(vertices1, 12, 0);
```

Then a `McdModel` can be created from this `McdConvexMeshID` geometry:

```
    cModel = McdModelCreate( geoPrim );
```

The example ***ConvexStairs*** demonstrates the use of convex geometries.

Aggregate



The Aggregate geometry type allows you 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:** Aggregates replace the Composite geometries of Karma 1.0, which could be neither nested nor shared. Composites are deprecated in Karma 1.1, and will be removed in a future release.

In order to use an aggregate geometry, you need to register its intersection test:with the collision framework

```
    void MEAPI McdAggregateRegisterInteractions(McdFrameworkID frame);
```

In order to create an aggregate, you need to specify the maximum number of component geometries it may contain.

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

Create an aggregate geometry with at most `maxChildren` sub-geometries

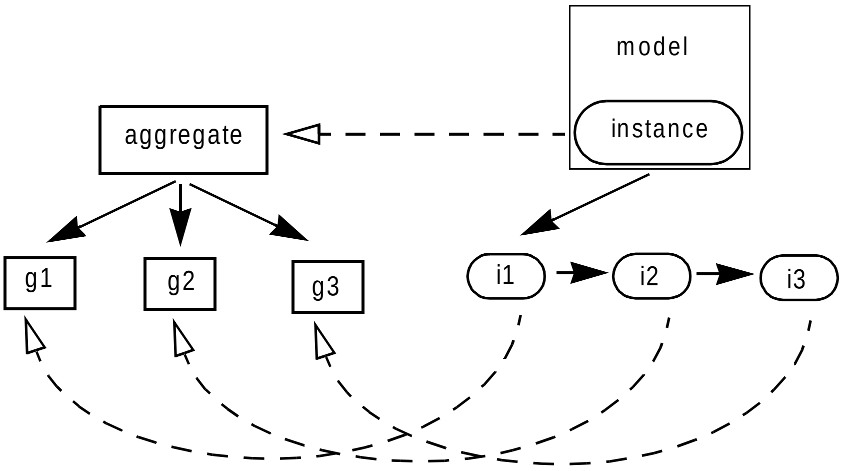
You can then add components to the aggregate:

```
     int               MEAPI McdAggregateAddElement(McdAggregateID,
                                                    McdGeometryID,
                                                    MeMatrix4 relTM);
```

Add an element to the aggregate, with a relative transform offset. The return value is the key to the sub-geometry within the aggregate, and can be used to remove or access properties of the sub-geometry

GeometryInstances and Aggregates

When an aggregate geometry is assigned to a model, a tree of `McdGeometryInstances` is allocated which matches the structure of the aggregate.

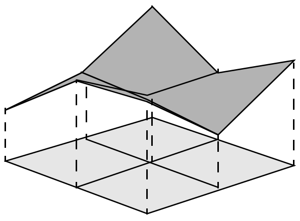


Geometry instances point to aggregate components, but since geometries exist independent of models and can be shared by several models, there are no pointers in the other direction You can access the instance corresponding to component *n* with the `McdGeometryInstanceGetChild`. function. Materials are specified on a per-instance basis, allowing you to specify different materials for each instance of each component of an aggregate. As with all geometries, the geometry instance which corresponds to to the top-level aggregate is inside a model. Child `McdGeometryInstances`, however, are allocated from the pool created by `McdInit`, so if you are using aggregate geometries you should set the third parameter of `McdInit` to be the number of geometry instances required by your application.

The transformation matrix for an instance of a component of an aggregate is computed by composing the relative transform of the component and the transform of the instance corresponding to the component's parent, i.e. mapping the component into the parent's space, and then into world space. Although this transformation is not stored by default, it can sometimes be useful, e.g. if you are rendering the aggregate by rendering each component separately. In such a case, assign a pointer to an `MeMatrix4` to the instance corresponding to the component using the `McdGeometryInstanceSetTransformPtr function.` Then, when the model is updated the transformation matrix of the component will be stored in the space you have allocated.

The ***Chair*** example demonstrates how to use aggregate geometries.

RGHeightField



The `RGHeightField` geometry type represents a terrain as a set of z-values defined on a regular grid of locations in the x-y plane. You can create a regular-grid height field using:

```
McdRGHeightFieldID MEAPI McdRGHeightFieldCreate( McdFramework *frame,
                                                MeReal* heightArray,
                                 int xVertexCount, int yVertexCount,
                                 MeReal xIncrement, MeReal yIncrement,
                                             MeReal x0, MeReal y0 );
```

Allocate memory for `RGHeightField`, and set its parameters. The height matrix is allocated by the user.

You register the height field geometry type the same way you register the convex mesh type. For an application using primitives, convex meshes and height field geometry types all at the same time, you could use:

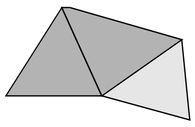
```
McdRGHeightFieldRegisterType();
McdRGHeightFieldRegisterInteractions();
```

As with the convex mesh type, the height field type currently interacts with some, but not all of the other types. See *Intersection Functions* on page 86 for more details.

The ***CarTerrain*** example program shows how to create a terrain using a `RGHeightField` geometry

```
/* height field grid data */
MeReal *heights;
int ixGrid = 50;
int iyGrid = 50;
MeReal deltaX = 1.5f;
MeReal deltaY = 1.5f;
MeReal xOrigin;
MeReal yOrigin;

terrainPrim = McdRGHeightFieldCreate(heights, ixGrid, iyGrid, deltaX, deltaY,
xOrigin, yOrigin);
terrainCM = McdModelCreate(terrainPrim);
```

TriangleMesh



Register available interactions between a type and primitive geometry types provided with the Mcd system.

```
void MEAPI McdTriangleMeshRegisterType()
void MEAPI McdTriangleMeshTriangleMeshRegisterInteractions()
```

NOTE: The TriangleMesh type currently only interacts with itself and no other types.

```
McdTriangleMeshID MEAPI McdTriangleMeshCreate( McdFrameworkID frame,
                                                    int triMaxCount)
```

Create a triangle mesh with given maximum number of triangles.

First, we create a TriangleMesh geometry:

```
box_geom = McdTriangleMeshCreate(12);
```

```
int MEAPI McdTriangleMeshAddTriangle( McdTriangleMeshID mesh, MeVector3
v0, MeVector3 v1, MeVector3 v2 );
```

Add a triangle to the triangle mesh. It can be referred to later by an index which starts at 0 with each triangle added and is incremented by one on each call.

Then we add each new triangle one by one:

```
      MeVector3[8];

    McdTriangleMeshAddTriangle( box_geom, vertex[0], vertex[1], vertex[2]);
    McdTriangleMeshAddTriangle( box_geom, vertex[0], vertex[2], vertex[3]);
    McdTriangleMeshAddTriangle( box_geom, vertex[1], vertex[6], vertex[2]);
    McdTriangleMeshAddTriangle( box_geom, vertex[1], vertex[5], vertex[6]);
    McdTriangleMeshAddTriangle( box_geom, vertex[5], vertex[7], vertex[6]);
    McdTriangleMeshAddTriangle( box_geom, vertex[5], vertex[4], vertex[7]);
    McdTriangleMeshAddTriangle( box_geom, vertex[4], vertex[3], vertex[7]);
    McdTriangleMeshAddTriangle( box_geom, vertex[4], vertex[0], vertex[3]);
    McdTriangleMeshAddTriangle( box_geom, vertex[1], vertex[0], vertex[4]);
    McdTriangleMeshAddTriangle( box_geom, vertex[1], vertex[4], vertex[5]);
    McdTriangleMeshAddTriangle( box_geom, vertex[2], vertex[7], vertex[3]);
    McdTriangleMeshAddTriangle( box_geom, vertex[2], vertex[6], vertex[7]);
```

```
unsigned int MEAPI McdTriangleMeshBuild( McdTriangleMeshID mesh);
```

Do precomputation necessary for fast collision detection. Allocates memory. Needs to be called before any call to `McdIntersect` on a model with this geometry.

```
McdTriangleMeshBuild( box_geom );
```

You can finally create a model from the geometry you just created, as for any other geometry, primitive or not:

```
boxCM = McdModelCreate(box_geom);
```

You can obtain the mass properties of your newly created model by using the following function:

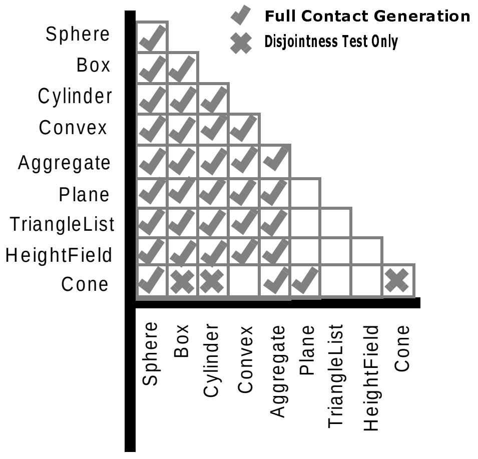
```
MeI16 MEAPI McdTriangleMeshGetMassProperties( McdTriangleMeshID mesh,
MeMatrix4 relTM,
MeMatrix3 m, MeReal* volume)
```

Compute the mass properties of the triangle mesh using an exact volumetric method that is robust in the presence of small holes in the model.

```
McdTriangleMeshGetMassProperties(box_geom, relTM, massP, &volume);
```

Intersection Functions

The following figure indicates which intersection functions are available for all possible pairs of geometry types provided in Karma Collision.


✓ Full Contact Generation
✗ Disjointness Test Only

	Sphere	Box	Cylinder	Convex	Aggregate	Plane	TriangleList	HeightField	Cone
Sphere	✓								
Box	✓	✓							
Cylinder	✓	✓	✓						
Convex	✓	✓	✓	✓					
Aggregate	✓	✓	✓	✓	✓				
Plane	✓	✓	✓	✓	✓				
TriangleList	✓	✓	✓	✓	✓				
HeightField	✓	✓	✓	✓	✓				
Cone	✓	✗	✗		✓	✓			✗

***Figure 4**: Development State of the Intersection Functions of All Pairs of Geometrical Types*

Line Segment Queries

You may perform intersection tests between line segments and individual models using the `McdLineSegIntersect()` function:

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

Intersect 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.

The`McdLineSegIntersectResult` structure is where the intersection data is stored.

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