MathEngine Karma™ Collision

Developer Guide

MathEngine Karma Collision. Developer Guide.

Contents

Preface

1 Introduction to Karma Collision

2 Getting Started

Contents

Preface

About the Karma Collision Guide

This guide explains how to use Karma Collision to accurately detect collisions between 3D models. Karma Collision can be used with Karma Dynamics or alone.

Karma is available for:

- The Sony PlayStation2 games console.
- Xbox as a beta to paying customers.
- Single precision Win32 built against the Microsoft LIBC, LIBCMT or MSVCRT libraries.
- Linux on request.

The single precision Win32 build of Karma against MSVCRT is provided for evaluation.

This manual is aimed at developers of real-time entertainment simulation software, familiar with the following:

- The C programming language. Knowledge of Microsoft Visual C++ is an asset.
- Basic mathematical concepts.

Accompanying Documentation

Detailed information about each function is given in the HTML Karma Collision reference manual that can be found by following the ‘Demos and Manuals’ hyperlink in the index.html file in the metoolkit directory.

Conventions

Units

There is no built-in system of units in Karma, which is not to say that quantities are dimensionless. Any system of units may be chosen, either meter-kilogram-seconds, centimeter-grams-seconds or foot-pound-seconds. However, the developer is responsible for the consistency of values and dimensions used. This analysis becomes important when tuning an application, or changing several parameters simultaneously.

Type Conventions

Karma uses some special type definitions and macros. These make the code more portable and make it easier to move between single and double precision.

For example:

- `MeReal`: floating point numbers.
- `MeVector3`: a vector of 3 `MeReals`.
- `MeVector4`: a vector of 4 `MeReals`.
- `MeMatrix3`: A 3x3 matrix of `MeReals`.
- `MeMatrix4`: A 4x4 matrix of `MeReals`.

These and others are defined in `MePrecision.h`.

Typographical Conventions

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

Naming Conventions for C Identifiers

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

Related Software

Karma Viewer

Karma Viewer is a basic cross platform wrapper around the GLUT and Direct 3D libraries. While this enables developers to build 3D applications with simple scenes, it is not meant to replace the chosen rendering tool. Rather the developer should hook Karma up to the renderer they are using. Some basic performance monitoring tools are provided. The Viewer is documented in the MathEngine Karma Viewer Developer Guide.

Karma Dynamics

Karma Dynamics lets you add physical behavior to real-time 3D environments. The following documentation discusses Karma Dynamics:

- *MathEngine Karma Dynamics. Developer Guide*.
- *MathEngine Karma Dynamics. Reference Manual*.

Karma Simulation Toolkit

Karma Simulation (Mst Library) provides an API that bridges Karma Dynamics (Mdt Library) and Karma Collision (Mcd Library). The Mst Library simplifies the control of dynamics and collision by providing tools and utilities in an integrated programming environment. The following documentation discusses Karma Simulation:

- *MathEngine Karma Simulation Toolkit. Developer Guide*.
- *MathEngine Karma Simulation Toolkit. Reference Manual*.

About MathEngine

MathEngine PLC is the provider of natural behavior technology for leading-edge developers committed to injecting life into 3D simulations and applications. Founded in Oxford, England in 1997 and staffed by a team of physicists, mathematicians and programmers, MathEngine provides tools that give software developers the ability to add natural behavior to applications for use in the games and entertainment markets.

Contacting MathEngine

Head Office

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

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

Web Site

www.mathengine.com

Customer Technical Support

support@mathengine.com

General inquiries

sales@mathengine.com

Chapter 1 • Introduction to Karma Collision

Overview

Karma Collision provides a collection of computational geometry algorithms to address the collision detection needs of 3D games and other applications. These algorithms are designed specifically to produce the information needed for real-time simulations of rigid bodies bouncing, sliding, rolling or coming to rest against each other.

With Karma Collision, the collision related code will consist primarily of:

- Selecting a geometrical shape for each collision model
- Selecting the types of information to be computed for each pair of collision models

Collision detection is well known to be critical both in terms of performance and geometric accuracy. Writing competitive 3D applications requires the ability to experiment with implementations that reflect a variety of different performance-accuracy trade-offs. To address these issues, some additional tuning activities may be necessary, such as:

- Selecting the best representation for each individual collision model (for example, a ball vs. a cylinder)
- Selecting the best algorithm for each pair of collision models

Karma's design lets developers experiment with these variations without affecting the rest of their collision detection code.

Karma Collision is a general purpose, comprehensive collision detection system that provides many geometrical types and intersection algorithms suitable for a wide variety of applications. Karma can be used for non collision related problems, such as line-of-sight determination or sensor-like functionality, where the proximity of two objects is required.

Karma Collision provides three levels of configuration control, to minimize the memory and CPU resources needed. These are:

- System-wide resources (geometry types and interactions)
- Single geometrical type resources
- Single interaction type resources

The binaries to load with an application are specified, and the requisite geometrical types and interactions registered with the system. Only those will be linked with the application. The user can configure the controls for each type and for each interaction.

How Your Application Uses Karma Collision

Any application that includes Karma Collision should accomplish the following tasks in the following sequence.

- Defining Geometrical Types
- Defining Collision Models
- Creating Collision Spaces

and then repeatedly

- Testing for Collisions
- Responding to colliding pairs of models

To use Karma Collision, define a collision model for each 3D object that needs collision information computed. To do this, select one of the available collision geometries that corresponds closely to the rendered mesh that was created by your 3D graphics package, and associate the 3D collision and render objects with each other. For example, a sphere could be chosen as the collision model for a character's head mesh.

Karma provides geometric primitives (defined in `McdPrimitives.h`) such as `McdSphere` and `McdBox`, that can be used to describe suitable object collision geometry. Each geometry can be used to create an infinite number of identical models.

When the models have been created intersection queries can be performed on pairs of collision models, and then, using the resulting contact data, an appropriate response generated. Karma Dynamics could be used to generate this physically accurate response using the contact information from Karma Collision.

The collision model's coordinates must match what is seen in the rendered display. To do this update the position of each model from the position of its rendered mesh.

Separate collision geometry make code portable across different platforms and configurations that may require different renderers. Karma provides an independent framework for the spatial and geometrical properties of collision models and the environment in which they are defined. From this framework all the relevant information about potential geometrical interactions that can take place between models can be extracted, such as intersections or proximity queries.

Geometrical Types, Instances, and Models

Geometrical types are basically molds of specific shapes that can be used to create an infinite number of models with the same specifications. For example, if a sphere geometry with a certain radius is created, any number of identical spheres with the same radius can be created.

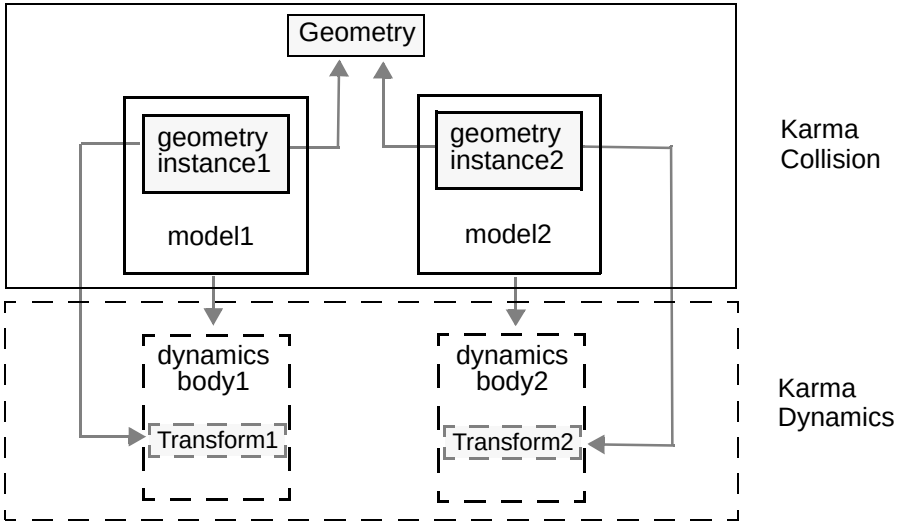
Each time the sphere is used, a corresponding Geometry Instance is created that contains properties specific to this occurrence of the geometry. This would include for example, a pointer to a transformation matrix that specifies the position of the object in space, and a material property. (Refer to the Karma Simulation Toolkit Developer Guide for information about materials).

Finally, each geometry instance is contained within a model that specifies the non-geometric non-material properties of the instance, such as its association with a dynamics body. A model is always associated with at least one (and usually only one) geometry instance, and to reflect this a geometry instance data structure is contained within each Karma collision model.

To specify the geometrical shape of collision models, choose it from Karma's library of geometrical types. The collision geometry should closely resemble the geometry of the 3D graphics model.

Simpler geometrical types require simpler algorithms and less memory. Having a wide selection of geometry types available gives flexibility in choosing trade-offs between performance, memory and geometrical accuracy. The geometrical representation for each collision model can be changed without altering the other code. This means that developers can continue experimenting with collision detection far into the development cycle.

Currently there are two kinds of geometry types: *Primitives* and *Non-Primitives.*

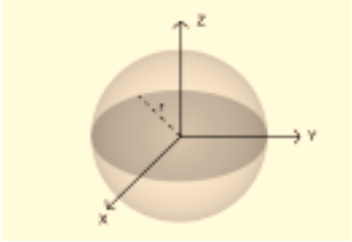


Geometrical Primitives

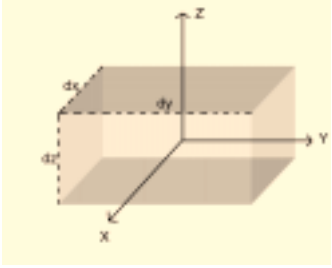
Primitive geometrical types are shapes defined by a small and fixed number of parameters. They can represent various specific types of curved surfaces exactly by parameters and specialized algorithms, not by discretized approximations. They are lightweight, fast and geometrically accurate. A number of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

Currently, Karma supports the following primitive geometrical types:

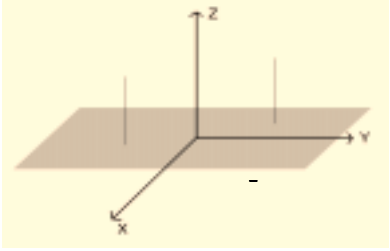
Sphere



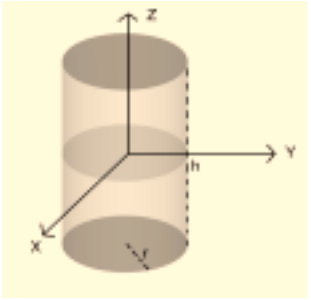
Box



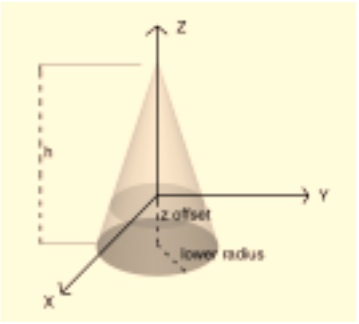
Plane



Cylinder



Cone



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

- In the case of the cylinder, cone and plane, z is the special axis - about which there is a symmetrical distribution of volume.
- The (uniform density) center of mass is placed at the origin. This is convenient and efficient for working with dynamics.

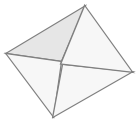
Geometrical Non-Primitives

Non-primitive geometrical types allow definition of more general classes of shapes for collision models. They are specified by a variable number of parameters. The number is large for complex models and small for simpler models. Any level of complexity can be used, based on a tradeoff between memory use and geometrical accuracy. This flexibility is due to the fact that the the non-primitive geometry types approximate surface geometry by a set of discrete surface elements.

Karma supports the following non-primitive geometrical types:

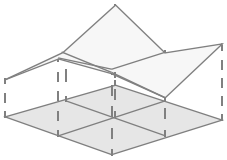
ConvexMesh

To define an arbitrary convex surface



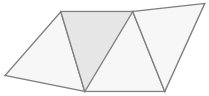
RGHeightField

To define a mesh with regular grid of height field



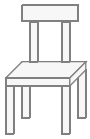
TriangleMesh

To define an arbitrary mesh (**Note:** TriangleMesh is unsuitable for typical game applications. Consider using TriangleList instead)



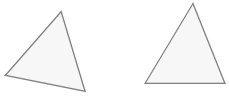
Aggregate

To build a collision model by combining other models, using both primitive and non-primitive geometries.



Triangle List

To test for collisions with any set of surfaces that can be triangulated.



Creating a Space

As the number *N* of collision models in a system increases, a new problem emerges: how to efficiently determine which of the N^2 possible pairs of collision models are actually near enough to each other to warrant calling an intersection function for them.

Such pairwise proximity culling is the principal function of the `McdSpace` module. The `McdSpace` module is a representation of the structure of 3D space and of the region of space occupied by each model present in it.

It is useful to think of a collision space as a container holding the collision models, where the models exist and evolve. The main feature of the collision space is that it orders locations in 3D space. Like any ordered container, elements can be inserted and removed, and benefit obtained from efficient update-sorting and fast global searches.

A `McdSpace` structure manages the large-scale spatial properties of a region of 3D space populated by `McdModel` objects by keeping track of proximities between them.

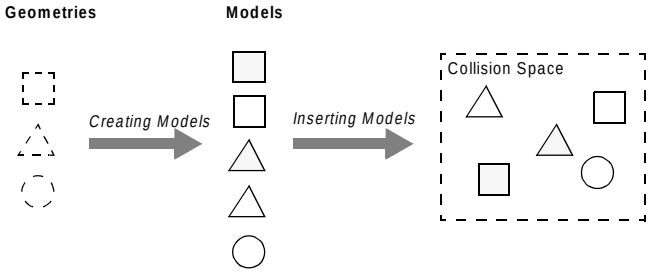


Figure 2: *Inserting Models in a Collision Space*

When creating a space, three or fewer axes along which the data will be organized must be chosen. The choice depends on the pattern of spatial distribution of collision models, and the known constraints on how they can move about in the scene. The `McdSpace`'s representation for specific scenarios may be optimized when the most likely behavior of the models is know in advance.

For example, in an application where objects stay close to an XY plane - perhaps cars on a highway - the XY coordinates of the models will be more significant for collision detection than their Z (height in the collision reference frame) coordinates, since the objects are more likely to collide along that plane. This feature provides a more efficient and a faster mechanism to retrieve relevant information. The `McdSpace` module data structures and updating mechanisms can be used to provide fast determination of other properties involving spatial relationships.

The McdSpace module detects pairs of models in close proximity, thus substantially reducing the number of pairs to consider for intersection tests. McdSpace ignores the geometric details of individual models by abstracting the models by their axis aligned bounding boxes (AABB). A model's AABB is the smallest box whose edges are parallel to the world reference frame, capable of enclosing this model.

As long as the bounding box of a model is not interpenetrating the bounding box of another model, the models will only be processed by the McdSpace module. If the bounding boxes interpenetrate, the models are considered to be possibly colliding, and anMcdModelPair object will be generated.

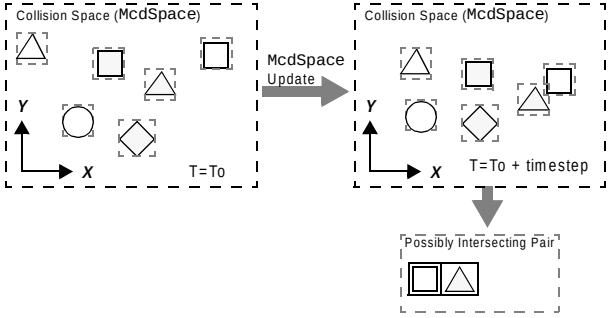


Figure 3: McdSpace *and* McdInteraction *View of the Collision Space*

When a McdModelPair object is created, it is added to an array of possibly intersecting pairs.This array of McdModelPair's is itself divided into three sub-arrays: one for newly created pairs of models (*Hello Pairs*), one for newly outdated pairs of models (*Goodbye Pairs*) and one for continuous pairs of models (*Staying Pairs*). Here is a table showing the possible evolution of a McdModelPair from one timestep to another:

Previous Timestep	**Current Timestep**	**Type of** McdModelPair
Not colliding	Possibly colliding	Hello
Possibly colliding	Possibly colliding	Staying
Possibly colliding	Not colliding	Goodbye

Interactions

While the `McdSpace` module is concerned with the large-scale properties of the 3D space, the interaction libraries provide a local view, focusing on the detailed geometrical properties of individual models.

Since each of these tests requires a dedicated and possibly computationally expensive algorithm to perform a specific intersection test, testing any of these pairs for intersection is optional and must be explicitly requested by the user. Such an algorithm is called an intersection function or simply an *interaction.* Note that when using Karma Collision with the Karma Simulation Toolkit, this will be done automatically.

Once you decide to test a pair for intersection, data about their geometrical relationship will be computed, such as their points of contact and their average normal. Since each collision model may be represented with a different geometrical type, separate intersection functions are needed, one for each possible pair of geometry types. See *Figure 4: Development State of the Intersection Functions of All Pairs of Geometrical Types* for a detailed table of the currently supported intersection functions.

> **NOTE:** The cone geometry type does not currently generate contact information, except for the cone-sphere and cone-plane interaction.

The current set of intersection functions generate contacts optimized for use with the Mdt Library, so that realistic physically-based responses can be created. Triangle Meshes can only intersect with other triangle meshes, and the following figure indicates which intersection functions are available for all possible other pairs of geometry types provided in Karma..

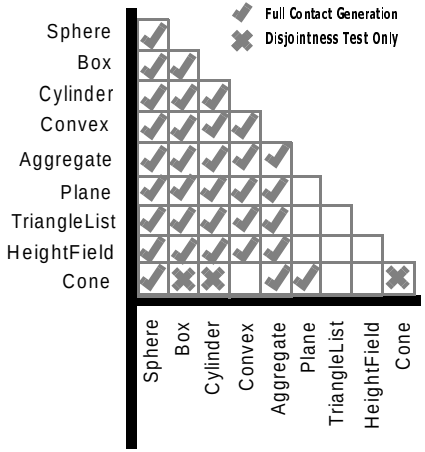


Figure 4*: Development State of the Intersection Functions of All Pairs of Geometrical Types*

Triangle List, HeightField, and Plane geometries are intended to represent large scale static geometry such as terrain. Thus Karma does not support collisions between geometries of these types.

Contact Generation

Intersection algorithms can be used to produce intersection data, which, in turn, are used to compute some type of collision response. Our experience has shown that the determination of a response that is both physically based and geometrically accurate is best supported by characterizing an intersection condition in terms of contact points.

For this reason, the current set of intersection algorithms available in Karma Collision are devoted to the generation of contact points, and to the building-up of a library of distinct contact generation algorithms reflecting different performance and accuracy trade offs.

What is a Contact?

A contact occurs when two objects have one or several points in common. An interpenetration is a measure of how deep one object intersects another, typically a distance by which one object has to be translated to avoid contact. For small interpenetration, one can define a contact normal, which is the local tangent to the object's surface at the point of contact. The properties associated with a contact can include:

- location of the contact
- contact normal (the direction along which penetration is to be prevented)
- estimate of penetration depth
- dimensional characterisation (whether the contact is a point, a line or a surface)
- local curvature of both objects at the contact position

This data is stored as members in `McdContact` structure. Note that a `McdContact` structure only contains geometrical informations about a contact between two objects. When using the Mdt Library to generate a physically accurate collision response, dynamical information related to the same contact would be stored and computed in a `MdtContact` structure. For example such dynamical information would include the force required to prevent an interpenetration between the two objects.

The number and location of contacts used to characterize a given intersection depends on a number of factors, including:

- surface geometry
- type of contact behavior
- required accuracy
- computation time available

If the required collision response is the simple bounce of a ball on a floor, for example, it can often be achieved with a single contact and using a relatively crude estimate. A plausible surface to surface contact behavior usually requires at least three contacts and can be very sensitive to geometric accuracy.

Collision Response

The response to a collision event is typically a change in motion, or a change in the allowable motions of the elements in a scene. The response could be something simple such as preventing a character from advancing in a given direction or a projectile sticking to the surface it hits; or very sophisticated such as an articulated toy falling down a staircase. Usually Karma dynamics generates the response behavior, but it could equally be handled by application-specific code.

No collision response takes place between any two objects until the time step when penetration first occurs. At this point, the intersection is non-zero, contact information is generated, and collision response acts so as to prevent further penetration. An estimate of penetration depth is provided so that, for example, a collision response algorithm knows how much “overshoot” to compensate for.

Related Functionality

Karma Collision needs to know about the geometrical shape and movement of the models in an application in order to handle collisions and contact situations. Once this is in place, it can be used as a high-performance geometrical database, that can perform other geometric calculations at the same time that it handles any collision detection needs.

Sensors and Event Handling

Many application events and application level-transitions are triggered by some type of geometric condition. Since Karma is already keeping track of proximity conditions between an application's models, it is a very small cost to have it function as an event detector at the same time.

To do this, create models that do not correspond to visually-rendered elements in the application, but define rather a spatial location. These models then function as sensors that can be used to detect which of the moving models are (for a game, say) inside a given room, inside a “danger zone”, close to a bomb or within hearing distance of an enemy.

These sensor collision models define regions of space, and detect which other collision models are inside of them or touching them. The region can be defined by any of the available geometrical types.

Rays

Karma’s ray functionality can help with many types of operations, including: see, occlude and shoot.

Instead of a list of contacts, the intersection data produced by intersection algorithms for rays is held in a ray-specific data structure that describes the surface where the ray has hit.

Use the global ray query that intersects a ray with the entire collision space, in which the space's internal representation is used, to efficiently determine the “first hit'”.

Chapter 2 • Getting Started

Overview

This chapter demonstrates how to write a simple generic program using Karma Collision, and extends the discussion of *Chapter 1 •Introduction to Karma Collision* on page 1 by introducing the Mcd Library API.

Organizing Your Programs

This section describes how to write a simple program using Karma Collision. In order to facilitate the learning of the Collision API, this chapter will closely integrate an actual Collision tutorial program to the introduction of the API basic set of functions.

Any application that includes Karma Collision should include specific pieces of code in the following sequence.

- Including the Required Header Files
- Linking the Required Libraries
- Initialization
- Defining Collision Models
- Creating Collision Spaces
- Updating the Models
- Testing for Collisions
- Cleaning Up

All tutorial programs reviewed in this guide will follow this sequence. We begin with `BallHitsWall1`, a program that does not use Karma Dynamics to handle the physical response to collisions, but rather uses handcoded response functions.

BallHitsWall1.c: A Sample Program

The motion of the ball is handled by the application program using a `move()` function. When Karma Collision detects a collision, a simple `handleCollision()` response function is used to produce the change in motion that corresponds to a bounce.

> **NOTE:** The collision response algorithm used here is of limited use outside of this simple scenario.

Two calls are required to obtain a complete list of all potentially colliding pairs of collision models. The collision space distinguishes between “newly nearby” pairs and pairs that were already nearby at the previous time step.

What This Program Shows

The program shows how to use Karma Collision to:

- Obtain collision events
- Identify the events (“ball-wall” collision vs. “ball-floor” collision)
- Call the intersection function
- Read the contact data structures

The code for the motion algorithm shows how to extract information from the contact data structures and how to use this information to produce appropriate changes in the motion of the rendered models.

The program shows how the new positions computed from the motion algorithm are used to update both the collision model and the rendered model.

Including the Required Header Files

The simplest applications using Karma Collision need access to the framework interface, as well as to the interface for creating primitive geometry objects. This involves two header files:

```
#include "McdFrame.h"
#include "McdPrimitive.h"
```

These header files are located in the `include` directory of the Karma distribution.

Linking the Required Libraries

The libraries for Karma Collision are located in the `bin` directory of the Karma distribution:

- The Karma Collision libraries are prefixed with `Mcd`.
- `McdFrame.lib` is always needed. `McdPrimitives.lib` is neededn to use any of Karma's primitive geometry types.
- The `Mst.lib` library is only required when using the bridge to Karma Dynamics.

Initialization

You control exactly what code is loaded into your application. You can tell the system which geometry types and interactions you intend to use at initialization time. For our first examples, we will begin by loading all the primitive geometry types:

```
McdInit( McdPrimitivesGetTypeMaxCount(), 100, 0);
McdPrimitivesRegisterTypes();
McdPrimitivesRegisterInteractions();
```

The last three lines are typical and appear in the same order at the beginning of any application that includes Karma Collision.

```
McdFrameworkID MEAPI McdInit ( int geoTypeMaxCount, int modelCount,
                                                    int instanceCount);
```

This function initializes the framework by performing allocation of tables.

- `geoTypeMaxCount.` The number of primitives required by the program
- `modelCount`. The maximum number of collision models that will exist simultaneously asT arguments.
- `instanceCount`. This is zero unless aggregate geometry's are used. Please refer to the section discussing aggregates in Chap 4.

The `McdPrimitivesGetTypeMaxCount()` returns a constant value, the number of primitives geometry types currently supported by Karma. The number of supported primitives may increase in future releases.

The `McdPrimitivesRegisterTypes()` function registers all the available geometry types. To register only the required geometry types, hence saving some space in memory, call successively some of the following functions:

```
void MEAPI Mcd*RegisterType ( McdFramework *frame )
```

This function registers a geometry of type * to the Mcd system. The wildcard * stands for one of the following:

- `Aggregate` Register the *Aggregate* geometry type.
- `Box` Register the *Box* geometry type.
- `Cone` Register the *Cone* geometry type.
- `ConvexMesh` Register the *ConvexMesh* geometry type.
- `Cylinder` Register the *Cylinder* geometry type.
- `Plane` Register the *Plane* geometry type.
- `RGHeightField` Register the *RGHeightField* geometry type.
- `Sphere` Register the *Sphere* geometry type.
- `RwBSP` Register the *Renderware BSP Tree* geometry type.
- `TriangleList` Register the *TriangleList* geometry type.
- `TriangleMesh` Register the *TriangleMesh* geometry type.

```
void MEAPI Mcd*RegisterTypes ( McdFramework *frame )
```

This function registers a geometry of type * to the Mcd system. The wildcard * stands for one of the following:

- `Primitives` Registers all *primitive* geometry types at once.
- `SphereBoxPlane` Registers the *sphere*, *box* and *plane* geometry types.

Each of the following functions can only be called if the corresponding geometry types have been previously registered. Refer to Chap 1 Figure 4, for supported interactions.

```
MeBool MEAPI McdPrimitivesRegisterInteractions ( McdFramework *frame )
```

Register interactions of *primitive* geometry types with the Mcd system.

```
void MEAPI McdSphereBoxPlaneRegisterInteractions ( McdFramework *frame )
```

Register interactions of the *sphere, box* and *plane* geometry types with the Mcd system.

```
void MEAPI McdRwBSPPrimitivesRegisterInteractions ( McdFrameworkID )
```

Register *Renderware BSP Tree* interactions with the Mcd system.

```
void MEAPI McdRGHeightFieldPrimitivesRegisterInteractions (
                                                  McdFramework *frame )
```

Registers interactions between *RGHeightField* and *primitive* geometry types with the Mcd system.

`ConvexMeshPrimitives` Registers the interactions between *ConvexMesh* and all *primitive* geometry types.

```
MeBool MEAPI McdConvexMeshPrimitivesRegisterInteractions (
                                                  McdFramework *frame )
```

Registers interactions between *ConvexMesh* and *primitive* geometry types with the Mcd system.

There is one interaction to register for the following, hence the singular of the `McdRegisterInteraction()` function name:

```
MeBool MEAPI McdTriangleMeshTriangleMeshRegisterInteraction (
                                              McdFrameworkID frame )
```

Register the interactions for the *TriangleMesh* geometry type.

```
MeBool MEAPI McdConvexMeshRGHeightFieldRegisterInteraction (
                                                  McdFrameworkID frame )
```

Register the interactions for the *RGHeightField* and *ConvexMesh* geometry type.

This is only a partial listing of available interactions between geometry types. Note that interactions including the cone geometry type does not currently generate any contact structure. For a complete description of available interactions, see *Intersection Functions* on page 64.

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

Defining Collision Geometries

Now we can create primitive geometries, for example:

```
MeReal Radius = 0.5;
McdGeometryID ball_prim = McdSphereCreate(radius);

MeReal wallDims[3] = { 0.5f, 2.0f, 5.0f };
McdGeometryID   box_prim   =   McdBoxCreate(2*wallDims[0],   2*wallDims[1],
                                        2*wallDims[2]);

McdGeometryID plane_prim = McdPlaneCreate();
```

This is one of the following `Mcd*Create` functions that returns an `Mcd*ID` handle to a geometry:.

```
Mcd*ID MEAPI Mcd*Create(args);
```

This function creates a geometry of type * . The *`args`* variable stands for a required list of arguments.The wildcard * stands for one of the following geometry types:

- `Box` Creates a *Box* geometry : `args = (McdFramework *frame, MeReal dx, MeReal dy, MeReal dz)` where `dx, dy` and `dz` stand for the `X, Y` and `Z` dimension of the box.
- `Cone` Creates a *Cone* geometry: `args = (McdFramework *frame, MeReal length, MeReal lowerRadius)` . In its local coordinate system, the cone's principal axis is aligned with the z-axis. The cone's center-of-mass is at the origin: this means that the base of the cone is below the `z=0` axis. The `length` variable represents the length of the cone along the `Z` axis and the `lowerRadius` variable represents the radius centered at the center of mass sweeping the `XY` plane.
- `Cylinder` Creates a *Cylinder* geometry: `args = (McdFramework *frame, MeReal radius, MeReal height).` The axis of the cylinder is along the `z-axis` of its local coordinate system. The radius and height variables represent the height of the cylinder along the Z-axis and the radius of the cylinder in the XY plane respectively.
- `Plane` Creates a *Plane* geometry: `args = (` McdFramework *frame). The plane is the xy-plane in its coordinate system.
- `Sphere` Creates a *Sphere* geometry: `args = (McdFramework *frame, MeReal radius).`

For a detailed description of the `Mcd*Create` functions for non-primitive geometry types, please consult “ConvexMesh” on page 58.

Defining Collision Models

Each of these functions creates the desired primitive geometry but does not create its associated collision model. To create a collision model from one of these shapes call the `McdModelCreate()` function. Following our example:

```
McdModelID ballCM = McdModelCreate(ball_prim);
McdModelID boxCM = McdModelCreate(box_prim);
McdModelID groundCM = McdModelCreate(plane_prim)
```

The formal description of `McdModelCreate()` follows:

```
McdModelID MEAPI McdModelCreate ( McdGeometryID geoPrimitives );
```

Creates a collision model with specified geometry *`geoPrimitives`*.

Collision models are the principal objects in Karma Collision. Their position and orientation in 3D space is represented as a 4x4 transformation matrix that must be allocated by the develolper if the bridge to Karma Dynamics is not being used, as in `BallHitsWall1`.

```
MeMatrix4 ballTM;
MeMatrix4MakeIdentityTM(ballTM);
ballTM[3][0] = 0;
ballTM[3][1] = 4;
ballTM[3][2] = 0;
McdModelSetTransformPtr(ballCM, ballTM);

MeMatrix4 boxTM;
MeMatrix4MakeIdentityTM(boxTM);
boxTM[3][0] = 3;
boxTM[3][1] = wallDims[1];
boxTM[3][2] = 0;
McdModelSetTransformPtr(boxCM, boxTM);

MeMatrix4 groundTransform =
    { {1, 0, 0, 0}, {0, 0, -1, 0}, {0, 1, 0, 0}, {0, 0, 0, 1} };
McdModelSetTransformPtr(groundCM, groundTransform)
```

A transformation matrix contains a 3x3 rotation submatrix $R = r_{ij}$ that determines the orientation of a model coordinate system in reference to the space coordinate system, and a translation vector $t = (t_x, t_y, t_z)$ that specifies the model center of mass position in the space coordinate system.

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

If a collision model has been paired to a `MdtBody` by `McdModelSetBody()`, then its transform pointer will point to the `MdtBody`'s transform. If `McdModelSetTransformPtr()` is called after this point, a warning will be issued.

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

Sets the geometry's 4X4 transformation matrix. The user must allocate his/her own `geometryTM` transform for a `cModel` model geometry, and pass a pointer in with this function. This transform determines the global coordinates of the model's geometry, that is specified locally.

Unless this model has been inserted in `McdSpace`, the user must call `McdModelUpdate()` every time the transform changes, before calling any of the intersection query functions such as `McdIntersect()`or `McdSafeTime()`.

Creating Collision Spaces

Now that the collision models are created, a collision space to hold them must be created. The following function creates a collision space:

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

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

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

In the example, we can write:

```
#define MAX_BODIES 10
McdSpaceID space = McdSpaceAxisSortCreate(McdFrameworkID fwk, McdAllAxes,
                                  MAX_BODIES, AVE_NEARBY_COUNT*MAX_BODIES);
```

Rather than creating space for the maximum number of pairs, that is MAX_BODIES^2, we are leaving space for a smaller number: AVE_NEARBY_COUNT*MAX_BODIES, where the constant AVE_NEARBY is the expected number of objects nearby a given object.

Note that AVE_NEARBY must be empirically determined by the programmer. If a simulation uses a few objects placed in a large room, it is reasonable to expect that any object could be nearby to at most two other objects, yielding a value of 2 for AVE_NEARBY.

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

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

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

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

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

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

In our example:

```
McdSpaceInsertModel(space, ballCM);
McdSpaceInsertModel(space, boxCM);
McdSpaceInsertModel(space, groundCM);
```

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

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

Indicates that most models have been inserted. Used for optimizing internal data structures in certain implementations of `McdSpace`.

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

In our example:

```
McdSpaceBuild(space);
```

To add models after the `McdSpace` has been built is a bit different than at initialization time. When using Karma Dynamics through the Simulation Toolkit bridge, the procedure to add a model is the same as during initialization. When only using Karma Collision, the models need to be added in a change block. For additional information regarding change blocks, consult the *Change Blocks* section in the *Advanced Features* chapter.

Here is an example:

```
*  McdSpaceBeginChanges(space)        //if outside change block
   McdSpaceInsertModel(space,model)
```

```
*  McdSpaceUpdateModel(model)
*  McdSpaceEndChanges(space)
*  [handle hello pairs]
```

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

Updating The Models

Each time objects included in the collision space are moved or rotated (i.e., their transform matrices have been modified), the collision space must be updated by calling:

`void MEAPI` **`McdSpaceUpdateAll`**`( McdSpaceID` *`space`* `);`
Updates nearby pair lists according to the latest object transform.

In our example:

```
McdSpaceUpdateAll(space);
```

To update a single modell

`void MEAPI` **`McdSpaceUpdateModel`**`( McdModelID collModel );`
Updates the bounding volume in space associated with `collModel`. Can only be called when `McdSpaceIsChanging()` is true. Is also called implicitly via `McdSpaceUpdateAll()`.

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

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

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

Freezing models is a way not to waste computing resources by testing for proximity, collision and then generating useless contacts between pairs of static models. In the example, the plane (i.e., the ground) is frozen, since it will not be moved by the ball or box:

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

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

There is a difference between freezing a model and not updating it: In the latter case, pairs will continue to be reported, even if both models are not being updated; in the former, pairs in which both models are frozen are not reported. For additional details about the freeze feature, see *Static Models* on page 41.

Testing for Collisions

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

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

All potentially colliding pairs of collision models, as determined by `McdSpace`, are stored in a `McdModelPairContainer` structure. This structure must be created by the following function:

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

Creates a ModelPairContainer of size `size`.

In our example:

```
pairs = McdModelPairContainerCreate(MAX_PAIRS);
```

Inside the `McdModelPairContainer` structure, there is an array containing the so-called *hello, staying* and *goodbye* pairs:

- *Hello*

  The *hello* pairs are those pairs of `McdModel`'s that `McdSpace` identifies as in close proximity to each other, but which where not in close proximity after the previous call to `McdSpaceUpdateAll()`.

- *Staying*

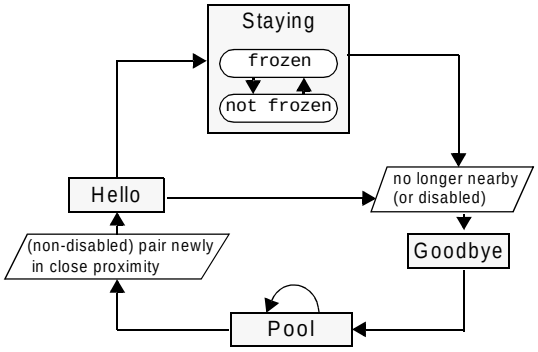
  The *staying* pairs are those pairs of `McdModel`'s that `McdSpace` identifies as in close proximity to each other, and were also in close proximity after the previous call to `McdSpaceUpdateAll()`.

- *Goodbye*

The *goodbye* pairs are those pairs of `McdModel`'s that `McdSpace` had identified to be in close proximity after the previous call to `McdSpaceUpdate()`, but are no longer in close proximity after the current call to `McdSpaceUpdateAll()`.

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

Here is a diagram showing the relationship between *hello, staying* and *goodbye* pairs:



In order to retrieve a list of one type of colliding pair, you need an *iterator*, that is, an index pointing to the beginning of the sub-array of the desired type. You must first declare an iterator variable of the type `McdSpacePairIterator`.

Once an iterator variable is declared, you need to initialize the iterator variable by calling the following function:

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

Initialise an iterator `iter` for the `collSpace` space. The variable `iter` is not valid until this function is called. This must be called before using `McdSpaceGetPairs()`. Must be called after `McdSpaceUpdateAll()` is called.

Finally, to retrieve successive sub-arrays of `McdModelPair` use:

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

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

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

The `McdModelPairContainer` structure filled in by `McdSpaceGetPairs()` identifies these states and transitions by holding three distinct lists of `McdModelPair` objects: *hello*,*staying* and *goodbye* pairs. Over a sequence of time steps, each `McdModelPair` goes through the same life cycle sequence: first appears as a *hello* pair, reappears zero, one or multiple times as a *staying* pair, and, when no longer in close proximity, last appears as a *goodbye* pair. In the step after that, the pair is invalidated and ready for reuse.

This setup guarantees two key properties that enable efficient management of collision response:

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

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

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

```
Typical usage loop: {
   stuff();
   McdSpace_changeOperations();
   McdSpaceEndChanges():
   McdSpaceGetPairs(pairs);
   usePairs();
   McdSpaceBeginChanges():
}
```

In our example, in the subroutine `tick()`, this sequence of function calls is written as:

```
MeBool pairOverflow;
move();
McdSpaceEndChanges(space);
McdSpacePairIterator spaceIter;
McdSpacePairIteratorBegin(space, &spaceIter);
do {
   McdModelPairContainerReset(pairs);
   pairOverflow = McdSpaceGetPairs(space, &spaceIter, pairs);
   Mcdgoodbye Each(pairs);
   McdhelloEach(pairs);
   handleCollision(pairs);
} while(pairOverflow);
```

Note that the `move()` subroutine, that updates each collision model position, is located inside the changes block.

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

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

Note that it is important to process all the goodbye and hello pairs by calling the appropriate functions (or using Mst utility functions that do this) after an McdSpaceEndChanges call and before an McdSpaceBeginChanges call. Since the Hello and Goodbye pairs are transient, these Hello and Goodbye events will be missed and either pairs will not be correctly initialized (McdHello) or memory will not be freed (McdGoodbye) causing undefined behavior. See *Chapter 3 •Advanced Features* on page 35 for more detail.

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

```
void MEAPI Mcd*Each( McdModelPairContainer* pairs );
```

Call `Mcd*` on each * pair. The wildcard * corresponds to one of the two transient types of colliding pair: *hello* and *goodbye*.

Now that we know which pairs of collision models are nearby each other and possibly colliding (*hello* and *staying*), we must check if there is an actual collision between each of them. This is done in the `handleCollision()` routine of `BallHitsWall1`.

To test for potential collisions between a pair of collision models call `McdIntersect()`. This will find an appropriate algorithm for the given pair of collision models, and compute the desired intersection data.

To do this supply a `McdModelPair` structure of nearby collision models and a `McdIntesectionResult` structure that will hold all relevant collision informations.

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

Performs appropriate collision test on input `pair`. All data, including newly generated `McdContact` objects for that pair of Mcd, are returned in `McdIntersectResult`'s `result`. The return value is `1` if the proper collision function was available, `0` otherwise.

In the example:

```
McdInteractionResult result;
result.contactMaxCount = 10;
result.contacts = contacts;

McdModelPairID pair = pairs->array[i];

McdIntersect(pair, &result);
```

Here are the members of the `McdIntersectResult` structure:

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

Note that an array of `McdContact` objects and its length must be provided before calling `McdIntersect()`.

We can now adjust the velocity of the ball based on the collision result, as it is done in the `handleCollision()` routine in `BallHitsWall1`.

Cleaning Up

When the simulation is completed, all the Mcd variables and structures must be destroyed explicitly. There are basically two cleaning up scenarios: one for applications using only Karma Collision and one for applications using Karma Dynamics through the Karma Simulation Toolkit bridge.

A model and its related geometry cannot be destroyed while the model is still in a space. Models must be removed from space with the McdSpaceRemoveModel() function before being destroyed:

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

Take model out of its space. Return 1 if successful, 0 if not. Any associated *staying* or *hello* pairs in that space will become *goodbye* pairs. Can only be called when McdSpaceIsChanging() is true.

To remove model(s) from space:

If using collision only:

```
*  McdSpaceBeginChanges(space) (if it hasn't been done already,
                               i.e. if not in a change block)
   McdSpaceRemoveModel(model)      // repeat as needed
*  McdSpaceEndChanges(space)
*  McdSpaceGetTransitions(space)   // must be done out of change block
                                     // otherwise currently no removed pairs
                                        output
*   [  handle all transitions, i.e.
      goodbye pairs from space involving
      the removed model(s).
      model _must_ be valid, including a valid geometry up to this point
   ]
McdGeometryDestroy(McdModelGetGeometry(model)) //optional
McdModelDestroy(model)
```

To remove a McdModel/MdtBody from McdSpace/MdtWorld the procedure is:

```
MdtBodyDisable(body)
McdSpaceRemoveModel(model) // the bridge keeps an open change block.
MstBridgeUpdateTransitions(bridge,world,space) // (or McdSpaceGetPairs and
                                                  MstHandleTransitions).
McdModelDestroy(model) // model must be removed from space before it is
                          destroyed.
McdGeometryDestroy(geometry) // (optional)
MdtBodyDestroy(body) // Must remove the collision model from the collision
                        space first (step 2).
```

All collision models that reference a particular geometry must be destroyed before the geometry itself is destroyed. This is because geometries are reference counted i.e. every persistent refer-

ence (in a collision aggregate or a collision model) to a geometry increases the reference count by 1 while that model exists. Geometries cannot be destroyed when their reference count is non-zero.

Typically steps marked by * (above) are done in the simulation loop and, if the user can wait until the end of the loop to destroy the model, do not have to be done explicitly at the point of deletion (the API was designed to allow this).

To remove a geometry:

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

Destroy a `McdGeometry` object. `geometry` is no longer valid after this call. All collision models associated with that geometry must be destroyed before the geometry is destroyed.

In `BallHitswall1`, the clean up is done by the `cleanup()` routine:

```
McdGeometryDestroy(ball_prim);
McdGeometryDestroy(box_prim);
McdGeometryDestroy(plane_prim);

McdModelDestroy(ballCM);
McdModelDestroy(boxCM);
McdModelDestroy(groundCM);

McdSpaceDestroy(space);
McdTerm();
```

You may notice that the change block functions are missing from the clean up code. This is because the entire application is running inside a change block by default, as only a handful of functions in the `tick()` routine are called outside the change block.

To remove a collision space.

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

Deletes a `McdSpace` object - performs memory deallocation.

When all this is done, free up the memory used by the framework by calling `McdTerm()`.

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

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

Chapter 3 • Advanced Features

Collision Models

Transform and Synchronization with Graphics

You need to ensure that the collision model's coordinate system matches that of the 3D graphics model that is being displayed. This often involves a three-way synchronization between the behavior module (i.e., the Mdt Library) determining the new positions and orientations, the graphics model rendered on the display, and the collision model used for determining contacts.

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

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

If you do not perform this step, the default value returned by `McdModelGetTransformPtr()` is NULL, rather than a pointer to an identity matrix.

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

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

`McdModelUpdate()` can optionally invoke a user-written callback function that can be used to perform synchronization of transforms automatically (in either direction). This can be set by the function `McdModelSetUpdateCallback()`.

Optimizations

An obvious optimization is to experiment using simpler geometrical shapes. The intersection algorithms are faster, and you may find that there are opportunities in your simulation where the reduced accuracy is not important.

One example is to replace a box geometry with a sphere geometry. This is usually reasonable only when the three dimensions of the box have similar values.

Another example is if you are using a box geometry to model a table-top, and the constraints of your game semantics ensure that all collision models that collide with the table-top never touch the edges. In this case, you can gain speed, without any loss in accuracy, by replacing the box geometry with the more specialized plane geometry.

Change Blocks

Change blocks are a mechanism to circumscribe changes to `McdSpace` by allowing specific changes to occur only between a pair of functions: `McdSpaceBeginChanges()` and `McdSpaceEndChanges()`. The change block 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.

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

To open a change block:

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

Indicates that a new set of state-modification operations will be applied to `space`. State-modifying operations can only be applied inside the `McdSpaceBeginChanges() / McdSpaceEndChanges()` delimiters (the *changes* block), and state-query operations, such as `McdSpaceGetPairs()`, can only be performed outside the *changes* block, i.e. after `McdSpaceEndChanges()` has been called.

To close a change block:

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

Indicates that no more state-modification operations will be applied to `space`, i.e. the end of the *changes* block. The user is now free to call state-query operations such as `McdSpaceGetPairs()`.

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

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

Indicates the current mode, either inside (true) or outside (false) a *changes* block. Returns true Immediately after a call to McdSpaceBeginChanges() and remains true until McdSpaceEndChanges() is called, after which point it returns false.

McdSpace functions applicable only inside the *changes* block: (i.e. when McdSpaceIsChanging() is true):

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

McdSpace functions applicable only outside the *changes* block: (i.e. 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()
McdSpaceSetUserData()
McdSpaceGetUserData()

Transition

Same effect as McdSpaceGetPairs(), only no *staying* pairs are reported. A convenience wrapper that is equivalent to calling McdSpaceGetPairs() and then setting the head of the *staying* list to NULL.

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

Returns *hello* and *goodbye* pairs only.

Useful for isolating the new events produced by a set of modifications in a *changes* block, and having them processed separately from (or earlier than) the remaining events. For example:

```
    {
      McdSpaceBeginChanges(s);
      McdSpaceRemoveModel(s,m1);
      McdSpaceRemoveModel(s,m2);
      McdSpaceEndChanges(s);
         McdSpaceGetTransitions(s,pairs);  /*  hello,staying  empty:  possible
                              goodbye events due to RemoveModel() calls */
   MstHandleTransitions( pairs ); /* handles bookkeeping for goodbye pairs*/
   McdSpaceBeginChanges(s);
      /* do other modifications, updates .. */
      McdSpaceEndChanges(s);
```

```
  McdSpaceGetPairs(s,pairs);
    MstHandleTransitions(  pairs  );  /*  handle  hello,staying  and  goodbye
                                        pairs*/
}
```

Static Models

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

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

Inform the system that a collision model `cm` 's transform will not change value. Changes the collision model `cm` status from *unfrozen* to *frozen*. A `cm` remains in the *frozen* status until `McdUnfreezeInSpace()` is called.

`McdModel` objects are by default in *unfrozen* status. This information is used for optimization opportunities by the `McdSpace` object in which a model is currently active. Nearby pairs for which both `McdModel`'s are frozen will be ignored by `McdSpace`, and will not appear in its nearby pair list. The collision model `cm` will be ignored by subsequent calls to `McdSpaceUpdateAllModels()`.

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

Frozen models also introduce the phenomenon of frozen pairs. A frozen `McdModelPair` is one which refers to a pair of frozen models in close proximity to each other.

This happens when a pair, after the last *changes* block, is in either the *hello* or *staying* state, typically with one model (but not the other one) frozen. In the new *changes* block, the other model of the pair becomes frozen via `McdSpaceFreezeModel()`. Such frozen pairs are suppressed from the `McdSpaceGetPairs()` output, and reappear only when the pair becomes unfrozen, i.e. when one of the models is unfrozen via `McdSpaceUnfreezeModel()`.

The reason is that, during successive time steps, any queries on a frozen pair will always return the same results, since both models are frozen. The data computed in their last non-frozen state can be re-used without recomputation. Suppressing the output of frozen pairs saves unnecessary recomputation of unchanging information.

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

To *unfreeze* a model:

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

Inform the system that a `cm` 's transform may change. Changes a `cm` 's status from “frozen” to “unfrozen”. Reverses the effects described in `McdModelFreezeInSpace()`.

To check the status of a model:

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

Returns a `cm` 's *frozen* status.return `TRUE` if model is frozen, else `FALSE`.

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.

After a call to `McdDisablePair()`, if in the last time step the list of pairs returned by `McdSpaceGetPairs()` contained a *hello* or *staying* pair involving the collision models `m1` and `m2`, then that pair will appear as a *goodbye* pair in the next call to `McdSpaceGetPairs()`. After that point, no more references to that pair will appear, regardless of whether they are in close proximity to each other. If `McdSpaceEnablePair()` is subsequently called on the same pair of models, the pair will reappear as a new *hello* `McdModelPair` object, if the pair is in close proximity to each other. This holds true even if `McdSpaceEnablePair()` is called within the same *changes* block that the original `McdSpaceDisablePair()` is called.

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

Cease to track proximity between `m1` and `m2`. If `m1` and `m2` come into close proximity after this point, no `McdPair` object will be assigned to this pair, and nothing will be reported about the pair in the next call to `McdSpaceGetPairs()`.

This behaviour is different to the one caused by a pair becoming frozen via `McdSpaceFreezeModel()`. In that case, the frozen pair does not appear as a *goodbye* pair, but simply ceases to be reported until unfrozen again. When it does become unfrozen, the original `McdModelPair` object reappears as a *staying* pair, and from that point on follows the usual semantics for regular non-frozen pairs.

`McdSpaceDisablePair()` can only be called when `McdSpaceIsChanging()` is true. Subsequent calls to `McdSpacePairIsEnabled()` with `m1` and `m2` as arguments will return false.

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

Resume tracking of proximity between `m1` and `m2`. If `m1` and `m2` come into close proximity after this point, or are in close proximity at the time that the call is made, a new `McdPair` object will be assigned to this pair, and it will appear as a *hello* pair in the next call to `McdSpaceGetPairs()`.

`McdSpaceEnablePair()` can only be called when `McdSpaceIsChanging()` is true. Subsequent calls to `McdSpacePairIsEnabled()` with `m1` and `m2` as arguments will return true.

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

Returns the `enabled` status for the pair `m1`, `m2`. See `McdSpaceEnablePair()`, `McdSpaceDisablePair()`.

Time Of Impact

Since Karma simulates physics numerically, with a discrete time-step, some collisions may not be detected. For example, imagine a ball travelling at high velocity toward a thin box. If at time `T` the ball was just entering the box without touching it and that a `timestep` later, at time `T+timestep`, the ball was just exiting it, the system would not detect any collision between the ball and the box. For the system, if there was no intersection between two models at a given time step then no collision occurred.

To prevent such mishaps from happening, utility functions were created to help you detect such virtual collisions. Note that these utilities work by using approximations and are therefore not foolproof. An example of an implementation of these functions can be found in the tutorial program `hispeed.c`,.

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

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

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

Sort Keys - Deterministic Simulation

Please refer to Chapter 4 - Advanced Features, Optimization and Utilities, in the Karma Dynamics Developer Guide, for an explanation of sort keys.

To assign a sort key to a collision model use:

```
void MEAPI McdModelSetSortKey(McdModelID cm, MeI16 key);
```

Assign sort key `MeI16 key` to collision model `McdModelID cm.`

To access a collision model sort key use:

```
MeI16 MEAPI McdModelGetSortKey(McdModelID cm);
```

Return the sort key `MeI16 key` of collision model `McdModelID cm.`

Line Intersection Utilities

The situation where you want to know the point of intersection between a line and a geometry, to determine the line of sight or the point of impact of a projectile for example, is one likely to occur often.

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

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

Finds first intersection of an oriented line segment with all 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 `outList` is a structure containing the returned line segment intersection data. The value `inMaxListSize` is the maximum number of intersection which will be reported.

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

Finds all the intersections of an oriented line segment with all 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 `outList` is a structure containing the returned line segment intersection data.

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.

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

```
unsigned int MEAPI McdLineSegIntersect ( const McdModelID cm,
                        MeReal* const inOrig, MeReal* const 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.

Integrating Dynamics

You can use Karma Collision to define the geometrical shape of bodies defined using Karma Dynamics (`MdtBody` structures), and to generate appropriate `MdtContact` structures characterizing the contact condition between the surfaces of two `MdtBody` structures.

Karma's dynamics solver uses this contact information so that it can respond to the contact condition, computing a motion which prevents objects passing through one another.

The Simulation Toolkit (Mst) provides a Bridge between Karma Collision and Dynamics. The Bridge manages all control flow and data exchange between the two, including keeping the `McdModel` and `MdtBody` coordinate systems synchronized with each other.

The following sample program shows you how to use it.

The scenario is identical to the previous one, only now the ball's motion is being computed by Karma Dynamics instead of in the application's code. To support this, the Simulation Toolkit Bridge (type `MstBridge`), computes the appropriate contact information for you, and communicates it automatically to the dynamics solver.

BallHitsWall2.c: Let's Get Dynamical

This example reproduces the identical behavior of `BallHitsWall1`, only using Karma Dynamics to control the motion. Aside from calls to Karma Dynamics and Simulation, no motion code appears anywhere in the program: it is a 3D simulation involving physics and geometry without any code involving physical or geometrical computations! In addition, there is no need for you to create or manipulate `McdContact` objects at all. The complete code of `BallHitsWall2.c` is distributed with Karma.

Creating a World

You need to initialize Karma Dynamics as well as Karma Collision before going on:

```
world = MdtWorldCreate(MAX_BODIES, MAX_CONSTRAINTS);
MdtWorldSetGravity(world, gravity[0], gravity[1], gravity[2]);
MdtWorldSetAutoDisable(world, 1);
```

Please refer to the *Karma Dynamics Developer Guide* for further details.

Initializing the Bridge

Once collision and dynamics are initialized, you may also create the bridge connecting them together:

```
bridge = MstBridgeCreate(10);
MstSetWorldHandlers(world);
```

Creating a Physical Body

You must create an MdtBody for Karma Dynamics to simulate:

```
ball = MdtBodyCreate(world);
MdtBodyEnable(ball);
MdtBodySetPosition(ball, 0.0, 4.0, 0.0);
MdtBodySetMass(ball, ballMass);
```

Assign a dynamics body to the ball

Attach the collision model to the ball that will represent the geometrical shape of a MdtBody:

```
McdModelSetBody(ballCM, ball);
```

Set parameters for contacts.

Create contact parameters to define the physical interaction between a ball and the box or the ball and the plane.

```
params = MstBridgeGetContactParams(bridge,
        MstBridgeGetDefaultMaterial(), MstBridgeGetDefaultMaterial());

MdtContactParamsSetType(params, MdtContactTypeFriction2D);
MdtContactParamsSetRestitution(params, 0.5);
MdtContactParamsSetSoftness(params, (MeReal)0.0005);
```

What This Program Shows

The Bridge handles all communication and synchronization between Karma Collision and Dynamics. The only responsibility of the application program is to specify the geometric and dynamic properties of the simulation (via Karma Collision and Dynamics respectively).

Chapter 4 • 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.

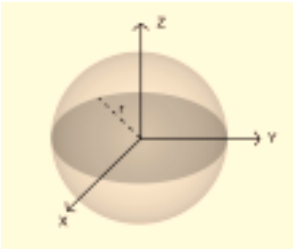
Geometrical Primitives

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:

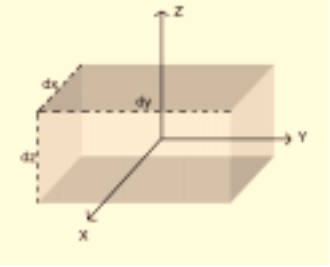
Sphere

Ball
Particle
Planet
Head



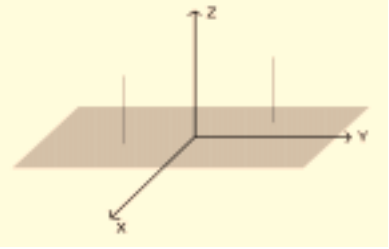
Box

Book
Skyscraper
Gambling Dice
Metal Bar



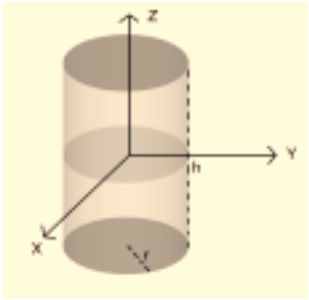
Plane

Floor
Wall
Ceiling
Ship-Deck



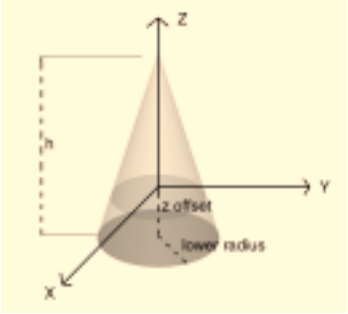
Cylinder

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 );
```

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

Triangle List

Triangle List is a primitive intended to allow triangulation of static geometry such as terrain. A triangle list is created using a bounding box and a user-supplied querying callback. If the Karma 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`.

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.

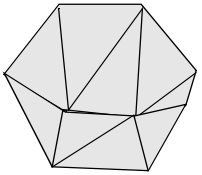
A recommended practice is to use, during initial development, the pair of registration functions that register all primitives and their interactions, then replacing this with a finer-tuned selection as it becomes clear exactly which subset is needed.

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.

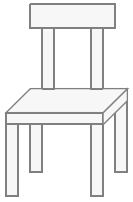
```
#define r1     (0.35f)
MeReal vertices1[12][3] = { { r1, 2*r1, r1},
                            {-r1, 2*r1, r1},
                            {-r1, 2*r1, -r1},
                            { r1, 2*r1, -r1},
                            { 2*r1, 0, 2*r1},
                            {-2*r1, 0, 2*r1},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

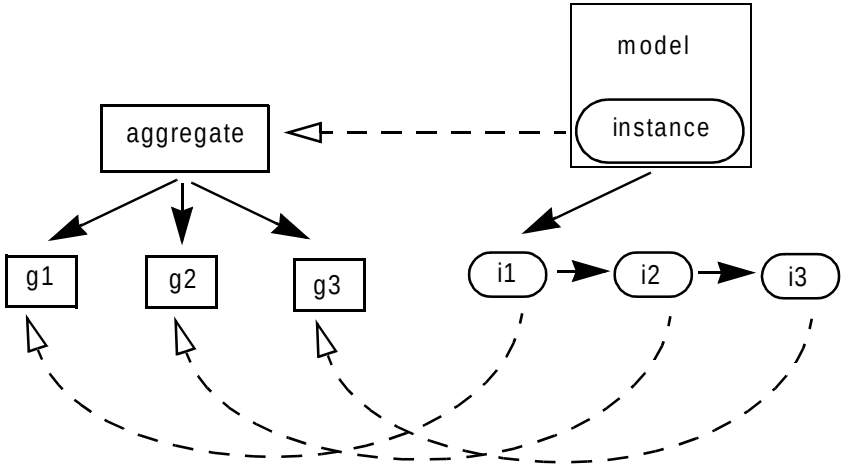
You can then add components to the aggregate:

```
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.



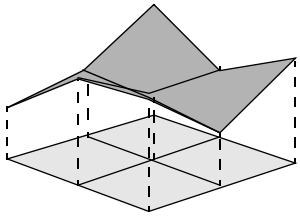
Geometry instances point to aggregate components, but since geometries exist independent of models and can be shared by several models, there are no pointers in the other direction You can access the instance corresponding to component *n* with the `McdGeometryInstanceGetChild`. function. Materials are 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:

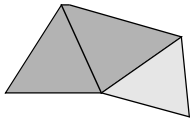
```
McdRGHeightFieldRegisterType();
McdRGHeightFieldRegisterInteractions();
```

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 64 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.

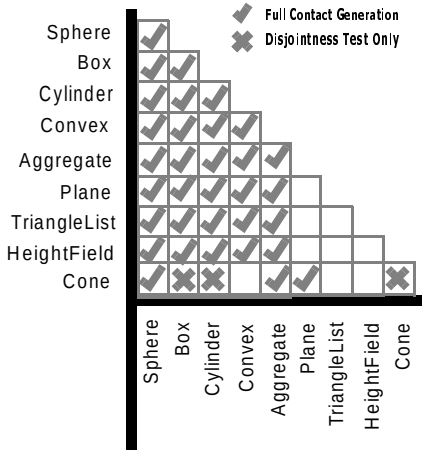


***Figure 4**: Development State of the Intersection Functions of All Pairs of Geometrical Types*