MathEngine Karma™ Simulation Toolkit

Developer Guide

MathEngine Karma Simulation Toolkit. Developer Guide.

Contents

Preface

Getting Started

Preface

About the Karma Simulation Toolkit Guide

This guide explains how to use the Karma Simulation Toolkit to integrate Karma Collision with Karma Dynamics. 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 Simulation Toolkit 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 Dynamics uses some special type definitions and macros that make it more portable. 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) • Directory and file names • Commands
`Courier` indicates:	• Program code
Italics indicates:	• Document and book titles • Cross-references

Naming Conventions for C Identifiers

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

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.

MathEngine Karma Dynamics

Karma Dynamics lets you add believable, realistic, complex physical behavior to real-time 3D environments. Karma Dynamics includes the following documentation:

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

Karma Collision

Karma Collision is a collision detection package. It provides the contact information required to produce real-time, geometrically-realistic collisions between 3D models. Karma Collision can be used with Karma Dynamics or on its own. The following documentation discusses Karma Collision:

- *MathEngine Karma Collision. Developer Guide*.
- *MathEngine Karma Collision. 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 Enquiries

sales@mathengine.com

Getting Started

What is the Karma Simulation Toolkit?

The Karma Simulation Toolkit (the Mst Library) provides an API that is a bridge between Karma Dynamics (Mdt Libraries) and Karma Collision (Mcd Libraries). A schematic of the Mst Library integrated architecture is shown in *Figure 1: Karma Simulation Architecture*.

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

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

The Mst Library automates all of these processes, and provides high-level control over each aspect of its activity. Use of the Mst Library ensures that combined use of Karma Dynamics and Collision is efficient and takes full advantage of the features available in both.

This library provides useful functions for creating and simulating objects using both Karma Dynamics and Collision. An `MstUniverse` contains an `McdSpace`, an `MdtWorld`, an `MstBridge`, and some buffers for moving contacts between Mcd and Mdt. The function `MstUniverseCreate` creates these things in one handy function. A collision `McdModel` is the main structure, with an optional dynamics `MdtBody` associated with it.

Mst contains functions for creating these together (for convenience), and for associating them when they have been created with the Mdt and Mcd APIs, using `McdModelSetBody()`. It can also set the mass and inertia tensor of an `MdtBody` to a sensible default based on the collision geometry and a density. The function `MstUniverseStep()` is a dynamics and collision *main loop*.

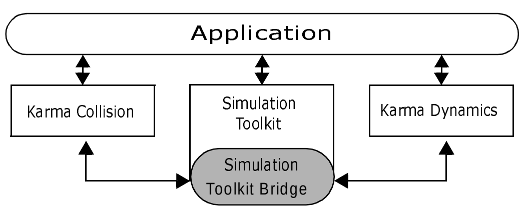

Figure 1: *Karma Simulation Architecture*

Introducing Mst

The tutorial program `RainbowChain.c` shows a chain of balls dropping on a plane and bouncing. This example demonstrates the use of contacts with Karma Dynamics, but without explicitly using Karma Collision. In the following introduction, a basic knowledge of Karma Dynamics and Karma Collision is assumed.

Creating a Universe

First, include the Mst library, declare variables and fill out constants and structures:

```
#include "Mst.h"

MstUniverseSizes sizes;

sizes.collisionModelsMaxCount = NBALLS + 1;
sizes.collisionPairsMaxCount = NBALLS * 16;
sizes.collisionGeometryTypesMaxCount = McdPrimitivesGetTypeCount();
sizes.dynamicBodiesMaxCount = NBALLS;
sizes.dynamicConstraintsMaxCount = NBALLS * 6;
sizes.materialsMaxCount = 1;
sizes.collisionGeometryInstancesMaxCount = 0;
```

The `MstUniverseSizes` structure contains all the important quantities used to manage the `MstUniverse`.

`MstUniverseSizes` **Member**	**Description**
`unsigned int dynamicBodiesMaxCount`	Maximum number of dynamic bodies allowed.
`unsigned int dynamicConstraintsMaxCount;`	Maximum number of dynamic constraints (joints & contacts) allowed.
`unsigned int collisionGeometryTypesMaxCount`	Maximum number of collision geometry types allowed.
`unsigned int collisionModelsMaxCount;`	Maximum number of collision models allowed.
`unsigned int collisionGeometryInstancesMaxCount;`	The number of additional geometries used when implementing aggregates.
`unsigned int collisionPairsMaxCount;`	Maximum number of simultaneously overlapping models allowed.
`unsigned int materialsMaxCount;`	Maximum number of materials allowed (at least `1`).

Then, create the `MstUniverse` using the function `MstUniverseCreate()`:

```
MstUniverseID MEAPI MstUniverseCreate(
                          const MstUniverseSizes* const sizes );
```

Allocate memory for an `MstUniverse` simulation container. An `MstUniverse` is a useful container for a collision `McdSpace` farfield, a dynamics `MdtWorld`, an `MstMaterialTable`, and containers used for moving contact information between collision and dynamics.

In `RainbowChain.c`:

```
MstUniverseID universe = MstUniverseCreate(&sizes);
```

The returned value, `MstUniverseID`, is a handle pointing to the newly created universe.

Fitting Out the Universe

The universe can be populated with dynamic and static objects. Static objects are created by a special function called `MstFixedModelCreate()`. This function takes a collision geometry and a tranformation matrix containing the static objects position and orientation. The collision model is created, positioned and inserted into the collision space. The model is then updated and frozen in place.

```
McdModelID MEAPI MstFixedModelCreate( const MstUniverseID universe,
              const McdGeometryID geometry, MeMatrix4Ptr transformation );
```

Create a fixed model with the supplied geometry for simulation. This function creates a collision `McdModel` with no dynamics. The model is automatically inserted into the universe's collision space and frozen.

In `RainbowChain`:

```
McdGeometryID planeGeom = McdPlaneCreate();
MeMatrix4 groundTransform= {1, 0, 0, 0,
                            0, 0, -1, 0,
                            0, 1, 0, 0,
                            0, 0, 0, 1 };
plane = MstFixedModelCreate(universe, planeGeom, groundTransform);
```

MstModelAndBodyCreate creates and then associates a `McdModel` with a `MdtBody` when provided with a geometry and a density:

```
McdModelID MEAPI MstModelAndBodyCreate( const MstUniverseID universe,
                  const McdGeometryID geometry, const MeReal density );
```

Create a *dynamic* model with the supplied geometry for simulation. This function creates a collision `McdModel` and attaches a dynamics `MdtBody` to it. The supplied `McdGeometry` is used to calculate sensible values for the inertia tensor of this body using the supplied density & geometry. The model is automatically inserted into the universe collision space.

In `RainbowChain`:

```
McdGeometryID ballGeom = McdSphereCreate(ballRadius);
ball[i] = MstModelAndBodyCreate(universe, ballGeom, (MeReal)(0.1));
```

`McdModel` objects are associated with `MdtBody` objects via `McdModelSetBody()`. When the `MstBridge` receives a list of `McdModelPair`'s representing potential or actual collisions, it generates `MdtContact` objects so that appropriate collision response occurs in the next call to `MstUniverseStep()`:

```
void MEAPI McdModelSetBody(const McdModelID model, const MdtBodyID body);
```

Utility function for assigning a dynamics body to a collision model. This is used so that contacts created during `MstBridgeUpdateContact` are attached to the correct dynamics body.

When an `MdtBody` is associated with a `McdModel`, a handle to the `MdtBody can be obtained by` using:

```
MdtBodyID MEAPI McdModelGetBody(const McdModelID model);
```

Return the dynamics body associated with this `McdModel` (if present).

Building a Bridge

Each newly created universe contains a *bridge*. To obtain a `MstBridgeID` handle to the `MstUniverse` bridge, `MstBridge`, to set properties such as the material properties, use the following function:

```
MstBridgeID MEAPI MstUniverseGetBridge( const MstUniverseID universe );
```

Get the Mst collision - dynamics bridge (part of the `MstUniverse`).

An `MstBridge` object is responsible for all operations and communication between Karma Collision and Karma Dynamics. In this guide, this object is often referred to as *The Bridge*.

Two other accessors to a `MstUniverse` are available: one to its `McdSpace` and another to its `MdtWorld`.

```
MdtWorldID MEAPI MstUniverseGetWorld( const MstUniverseID universe )
```

Get the dynamics world part of the `MstUniverse`.

```
McdSpaceID MEAPI MstUniverseGetSpace( const MstUniverseID universe )
```

Get the collision space part of the `MstUniverse`.

To use Karma Collision and Dynamics together when not using the `MstUniverse` container a bridge must be created from scratch using `MstBridgeCreate()`:

```
MstBridgeID MEAPI MstBridgeCreate( const unsigned int maxMaterials );
```

Create an `MstBridge`. This is used during `MdtStep()` to pass contact geometry information from Karma Collision to Karma Dynamics. The value `maxMaterials` represents the size of the material-material properties table.

Each `MdtBody` receives a `MstMaterialID` identifier that acts like an index in a matrix of contact parameters and callback functions, also called the *material table*. As soon as a contact is created, the `MstMaterialID` of both bodies in contact is used to retrieve the proper `MdtContactParams` structure and the name of the three callback functions it is using. For additional details about the material table callback functions, see *Callback Functions* on page 11.

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

```
MstMaterialID MEAPI MstBridgeGetNewMaterial ( const MstBridgeID bridge );
```

Get a new, unused material from the material table to assign to a model.

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

An `MstMaterialID` identifier needs to be attached to every `McdModel` by using the `McdModelSetMaterial()` mutator function. The `McdModelGetMaterial()` accessor function returns the `MstMaterialID` identifier of a `McdModel`. For additional details about these or any other `Mcd` functions, please consult the *Karma Collision Reference Guide*.

To obtain an `MdtContactParamsID` handle to a `MdtContactParams` structure, use:

```
MdtContactParamsID MEAPI MstBridgeGetContactParams (
                              const MstBridgeID bridge,
                          const MstMaterialID m1, const MstMaterialID m2 );
```

Get the current dynamics contact parameters for contacts between the given pair of materials `m1` and `m2`. The `MdtContactParams` interface can then be used to modify friction, restitution etc. for this pair of materials.

In RainbowChain the operations, `MstBridgeGetContactParams()`, `MstUniverseGetBridge()` and `MstBridgeGetDefaultMaterial()` are carried out consecutively:

```
MdtContactParamsID p;

p = MstBridgeGetContactParams(MstUniverseGetBridge(universe),
    MstBridgeGetDefaultMaterial(),
    MstBridgeGetDefaultMaterial());
```

After obtaining a `MdtContactParamsID` handle, friction and restitution etc, for the `MdtContact` object can be set.

Setting the Universe in Motion

Evolve the universe using.

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

Dynamics and Collision 'main loop', using `MstUniverse` container.

The step function in `RainbowChain.c` is located inside the `Tick()` routine, which is a callback function linked to MeViewer by the `RRun()`MeViewer function (see the *MathEngine Karma Viewer Developer Guide* for additional details).

```
MeReal step = (MeReal)(0.03);
MstUniverseStep(universe, step);
```

Resetting

To reset the non-static objects to their default values.

`void MEAPI **McdModelDynamicsReset**( const McdModelID m)`
Reset model dynamic body (if present) position to origin, orientation to default and zero velocity.

```
McdModelDynamicsReset(ball[i]);
McdModelDynamicsSetPosition(ball[i], (i*(MeReal)0.1), (i+1) * (2*ballRadius
                                        + ballSpacing), 0);
McdModelDynamicsEnable(ball[i]);
```

Cleaning Up

Before closing the application, free up the memory used by the structures and objects using the `Destroy` functions. I

`void MEAPI **MstFixedModelDestroy**( const McdModelID model )`
Destroy a fixed collision `McdModel`.

`void MEAPI **MstModelAndBodyDestroy**( const McdModelID mmodel )`
Destroy the collision `McdModel` and dynamic `MdtBody` (if present).

`void MEAPI **MstUniverseDestroy** ( const MstUniverseID u );`
De-allocate memory and destroy an `MstUniverse` simulation container. This is a useful way to clean most things up in one go. This will destroy: • all MdtBodies and all MdtConstraints regardless of whether they are enabled or disabled. • all McdModels that have been created, regardless of which McdSpace they are in. • the MstBridge. The only thing it does not destroy is McdGeometries, that must be destroyed explicitly before calling MstUniverseDestroy().

If a bridge was created outside of an `MstUniverse` using `MstBridgeCreate()`, the `MstBridgeDestroy()` function must be called:

`void MEAPI` **`MstBridgeDestroy`**`( const MstBridgeID` *`bridge`*`)`
Destroy an `MstBridge`.

More on the Bridge

Some additional MstBridge functions follow:

```
void MEAPI MstBridgeUpdateTransitions( const MstBridgeID bridge,
                    const McdSpaceID space, const MdtWorldID world );
```

Handle `Hello` and `Goodbye` farfield pairs. This removes any contacts and data for `Goodbye` pairs, and initialises `Hello` pairs. Use after `McdSpaceRemoveModel` before destroying the model. Calls `McdSpaceUpdate`, `McdSpaceGetPairs` and `MstHandleTransitions`.

```
void MEAPI MstBridgeUpdateContacts( const MstBridgeID bridge,
                    const McdSpaceID space, const MdtWorldID world );
```

Take all the `McdModelPairs` from the `McdSpace`, and handle them. Call this after `McdSpaceUpdate` to update the contact geometry information in the `MdtWorld`.

```
void MEAPI MstBridgeSetModelPairBufferSize( const MstBridgeID bridge,
                                            const unsigned int size);
```

Manually resize the `ModelPair` buffer used during Step. This will allocate memory.

```
void MEAPI MstBridgeSetContactBufferSize( const MstBridgeID bridge,
                                          const unsigned int size );
```

Manually resize contact buffer used during Step. This will allocate memory.

Callback Functions

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

A callback provides users with a means of modifying a given structure before it is used. They provide control over `McdContact`'s before they are converted to `MdtContact's`. Figure 2 shows when collision callbacks are called when using the simulation layer.

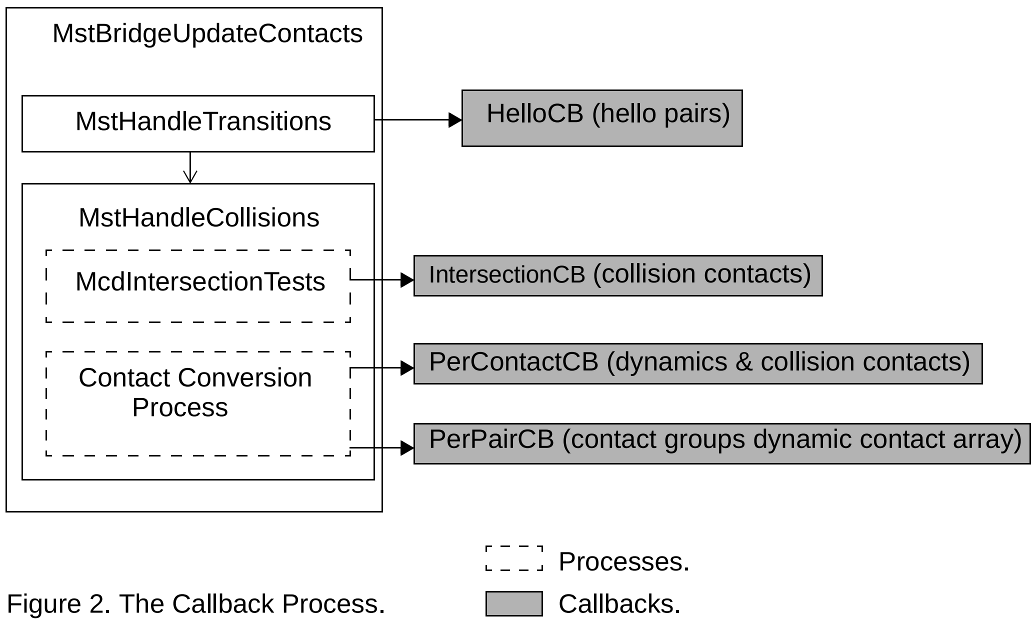

Figure 2. The Callback Process.

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

Setup callbacks on a world to ensure that when a body in a world stops moving its collision model is frozen, and when an interaction causes it to start moving again, its collision model is unfrozen.

Callback Accessors

To access the callback function associated with a specific pair of material identifiers

```
MstIntersectCBPtr MEAPI MstBridgeGetIntersectCB(
                const MstBridgeID bridge,
           const MstMaterialID material1, const MstMaterialID material2 )
```

Get the current, optional per-pair callback executed for each intersect.

```
MstPerContactCBPtr MEAPI MstBridgeGetPerContactCB(
                const MstBridgeID bridge,
      const MstMaterialID material1, const MstMaterialID material2 )
```

Get the current, optional per-pair callback executed for each contact between models with the given materials.

```
MstPerPairCBPtr MEAPI MstBridgeGetPerPairCB( const MstBridgeID bridge,
                                       const MstMaterialID material1,
                                       const MstMaterialID material2 )
```

Get the current, optional per-pair callback executed for each colliding pair of models with the given materials.

Callback Mutators

```
void MEAPI MstBridgeSetIntersectCB( const MstBridgeID bridge,
                                    const MstMaterialID material1,
                                    const MstMaterialID material2,
                                    const MstIntersectCBPtr cb );
```

Set the optional per-intersection user callback for the given pair of materials. This will be executed once for each pair of colliding models with the given materials, with the `McdIntersectResult` and the set of collision contacts. It allows control over `McdContact`'s before they are converted to `MdtContacts`.

To obtain the whole set of collision contacts in one go use the intersect callback. Developers doing their own contact culling would use this.

```
void MEAPI MstBridgeSetPerContactCB( const MstBridgeID bridge,
                                     const MstMaterialID material1,
                                     const MstMaterialID material2,
                                     const MstPerContactCBPtr cb );
```

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

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

```
void MEAPI MstBridgeSetPerPairCB( const MstBridgeID bridge,
                                  const MstMaterialID material1,
                                  const MstMaterialID material2,
                                  const MstPerPairCBPtr cb );
```

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

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

Utilities

```
void MEAPI MstAutoSetMassProperties( const MdtBodyID body,
                                     const McdModelID model,
                                     const MeReal density );
```

Utility for setting the mass and inertia tensor of a dynamics body based on the collision geometry and the supplied density.

```
void MEAPI MstHandleCollisions( McdModelPairContainer* pairs,
                                const McdSpaceID space,
                                const MdtWorldID world,
                                const MstBridgeID bridge );
```

Generate contact for potentially intersecting pairs and update dynamics. This will call the relevant geometry-geometry test to generate contacts for each pair, and will use the contact geometry to update the set of dynamics contacts. Call `MstHandleTransitions` first on the `McdModelPairContainer` to process initialise *Hello* pairs and clean up *Goodbye* pairs. Use `McdSpaceGetPairs` to fill the `McdModelPairContainer` with pairs from the farfield `McdSpace`.