

MathEngine™ Collision Toolkit 1.0

Developer's Guide

Alpha 0.0.5 – August 2000

© 2000 MathEngine PLC. All rights reserved.

MathEngine Collision Toolkit Developer's Guide.

MathEngine is a registered trademark and the MathEngine logo is a trademark of MathEngine PLC. All other trademarks contained herein are the properties of their respective owners.

This document is protected under copyright law. The contents of this document may not be reproduced or transmitted in any form, in whole or in part, or by any means, mechanical or electronic, without the express written consent of MathEngine PLC. This document is supplied as a manual for the MathEngine Collision Toolkit. Reasonable care has been taken in preparing the information it contains. However, this document may contain omissions, technical inaccuracies, or typographical errors. MathEngine, PLC does not accept responsibility of any kind for customers' losses due to the use of this document. The information in this manual is subject to change without notice.

Printed in Canada.

The MathEngine Collision Toolkit uses Qhull (c) to construct convex models from points. Qhull is a software package used for computing the convex hull and related geometrical structures. It is available free of charge from the University of Minnesota Geometry Center.

The following copyright notice applies:

Qhull, Copyright (c) 1993-1998

The National Science and Technology Research Center for Computation and Visualization of Geometric Structures (The Geometry Center) University of Minnesota 400 Lind Hall 207 Church Street S.E. Minneapolis, MN 55455 USA email: qhull@geom.umn.edu URL: <http://www.geom.umn.edu/software/qhull>

Contents

Preface

About This Manual	viii
Audience for this Manual	viii
Related Documentation	viii
Conventions	ix
Units	ix
Type Conventions	ix
Typographical Conventions	ix
Naming Conventions for C Identifiers	x
Related Software	xi
MathEngine Viewer	xi
MathEngine Dynamics Toolkit	xi
About MathEngine	xii
Contacting MathEngine	xii

1 What Can the Collision Toolkit Do?

Overview	2
How Your Application Uses the Collision Toolkit	3
Near Field vs. Far Field	4
Collision Detection vs. Collision Response	5
Integrating the Dynamics Toolkit	5
Integrating Your Own Tools and Third-Party Tools	6
Geometrical Types and Interactions	8
Geometrical Types	8
Interactions	9
Configuration Control and Code Size	9
Contact Generation and Physical Simulation	10
What is a Contact?	10
How Do I Use Contacts to Perform Collision Response?	11
Non-Dynamics Related Functionality	12
Sensors and Event Handling	12
Rays	12
Far Field Functionality	13

Pairwise Proximity Culling	13
Other Spatial-Global Queries	13
Alternative Implementations	13
Simulation Support Layer	14
Specific Product Integrations	15
Dynamics Toolkit Bridge	15
RenderWare Models	16
 2 Installing the Collision Toolkit	
Overview.	18
 3 Getting Started	
Overview.	20
Organizing Your Programs	20
Including the Required Header Files	20
Linking the Required Libraries.	20
Initialization	20
Defining Collision Models	21
Creating Collision Spaces	21
Testing for Collisions	21
BallHitsWall1.c: A Sample Program	23
What This Program Shows	23
Supporting the Dynamics Toolkit	31
BallHitsWall2.c: Let's Get Dynamical.	31
What This Program Shows	32
BallHitsWall3: Complex Behavior	32
Changing Geometrical Properties	32
Changing Dynamical Properties	33
 4 Advanced Features	
Collision Models and the Geometry Library	36
Transform and Synchronization with Graphics	36
Optimizations.	36
Culling Management and Culling Optimizations.	37
The "Freeze" Feature	37
Alternative Implementations and Specializations	37

5 Integrating with the Dynamics Toolkit

Overview	40
Relative Transforms	41
Coordinate Systems and Implied Geometry	42
Inertia Matrix	43
Composite Geometries	44
Material Properties	45
Interactions with the Static Environment	46
Optimizations	47

6 Geometrical Types and Their Interactions

Overview	50
Geometrical Primitives	51
Sphere	52
Box	52
Plane	52
Cylinder	53
Cone	53
Registering Geometrical Types and Intersection	54
Non-primitive Geometrical Types	55
ConvexMesh	55
RGHeightField	55
Intersection Functions	57

MathEngine Toolkits Glossary

Preface

About This Manual

This manual, the *MathEngine Collision Toolkit Developer's Guide*, explains how to use the Alpha version of the MathEngine Collision Toolkit to produce real-time, geometrically-realistic collisions between 3D models. The Collision Toolkit can be used with the MathEngine Dynamics Toolkit or alone.

The Collision Toolkit is a software development kit (SDK) that you can use on platforms such as Linux and Windows NT, and on game consoles such as the Sony Playstation2.

Audience for this Manual

This manual is aimed at game developers and developers of 3D simulations who are familiar with:

- 3D animation.
- The C programming language.

Related Documentation

This manual is accompanied by:

- The *MathEngine Collision Toolkit Reference Manual*, which provides detailed information about each function.
- The *MathEngine Glossary*, which is part of this manual.

Conventions

Units

The Collision Toolkit can work in any system of units, but the documentation assumes that you are using SI units: kilograms, meters, seconds, Newtons, and so forth.

Type Conventions

The MathEngine Collision Toolkit uses some special type definitions and macros. These make the toolkit 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 `MeGlobals.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.
Mcd	MathEngine Collision Toolkit
Mdt	MathEngine Dynamics Toolkit
R	MathEngine Viewer

Related Software

MathEngine Viewer

The MathEngine Viewer lets developers get visual results from MathEngine code quickly and simply. It allows developers to easily obtain statistics on the performance of their application. The Viewer is included with the Collision Toolkit, and is used in much of the sample code. It is also included with other MathEngine products.

The Viewer is documented in the *MathEngine Viewer Developer's Guide*.

MathEngine Dynamics Toolkit

The MathEngine Dynamics Toolkit lets you add believable, realistic, complex physical behavior to real-time 3D environments.

- The *MathEngine Dynamics Toolkit Developer's Guide* explains how to use the Alpha version of the MathEngine Dynamics Toolkit 2.0.
- The *MathEngine Dynamics Toolkit Reference Manual*, which provides detailed information about each function.

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 creates tools that give software developers the ability to add natural behavior to applications for use in the games and entertainment, education, engineering, and e-commerce markets.

Contacting MathEngine

Head Office

MathEngine plc, Oxford Centre For Innovation, Mill Street, Oxford, UK, OX2 0JX

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

Web Site

www.mathengine.com

Technical Support

support@mathengine.com

Partnership program and customer contacts

General inquiries: Rhian Steel, sales@mathengine.com

Chapter 1 • What Can the Collision Toolkit Do?

Overview

You can use the Collision Toolkit to perform collision-detection-related computations in your 3D games and other applications. Underlying algorithms are designed specifically to produce the information needed for real-time, geometrically-realistic simulations of bodies bouncing, sliding, rolling or coming to rest against each other.

The Collision Toolkit provides an interface between your simulation's collision needs and computational geometry algorithms, so that you can focus on what you want to *do* with the geometrical information you need, instead of *how* you are going to compute it.

With the Collision Toolkit, 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 become important for you:

- 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

The Collision Toolkit's implementation-transparent design lets you experiment with these variations without affecting the rest of your Collision Toolkit code.

You can also use the Collision Toolkit for non-collision-related problems, such as *line-of-sight determination* or sensor-like functionality, where you wish to determine the proximity of an object relative to another.

How Your Application Uses the Collision Toolkit

To use the Collision Toolkit, you define a collision model for each 3D object that needs collision information computed. To do this, select one of the available collision geometry such that it corresponds closely to the visually-rendered geometry you created using your 3D graphics package and then pair it with the 3D object. For example, a sphere could be chosen as the collision model for a character's head.

You must ensure that the collision model's coordinates match what is seen in the rendered display. You then perform intersection queries on pairs of collision models, and use the resulting contact data to compute the response using, for example, the MathEngine Dynamics Toolkit.

Separate collision geometry makes your Collision Toolkit code portable across different platforms/configurations which may require different renderers. The Collision Toolkit provides an independent framework for the spatial and geometrical properties of your models and the environment in which they are defined. From this framework you can extract all the relevant information about potential geometrical interactions, like intersections or proximity queries, that can take place between models.

Near Field vs. Far Field

The Collision Toolkit provides both Near Field and Far Field views of your 3D scene. Near Field and Far Field collision tests are carried out on a virtual representation of your rendered world.

The Far Field module provides a global view of the 3D scene that ignores the geometric details of individual models.

The Near Field module provides a local view, focusing on the detailed geometrical properties of individual and individual pairs of models.

Figure 1: Collision Toolkit Architecture shows the organization of the Collision Toolkit. The kernel is a small core element defining a component model for 3D geometry. It defines collision models, how they communicate with each other, and how they communicate with the far field module, and defines the plug-in architecture for geometry types and algorithms.

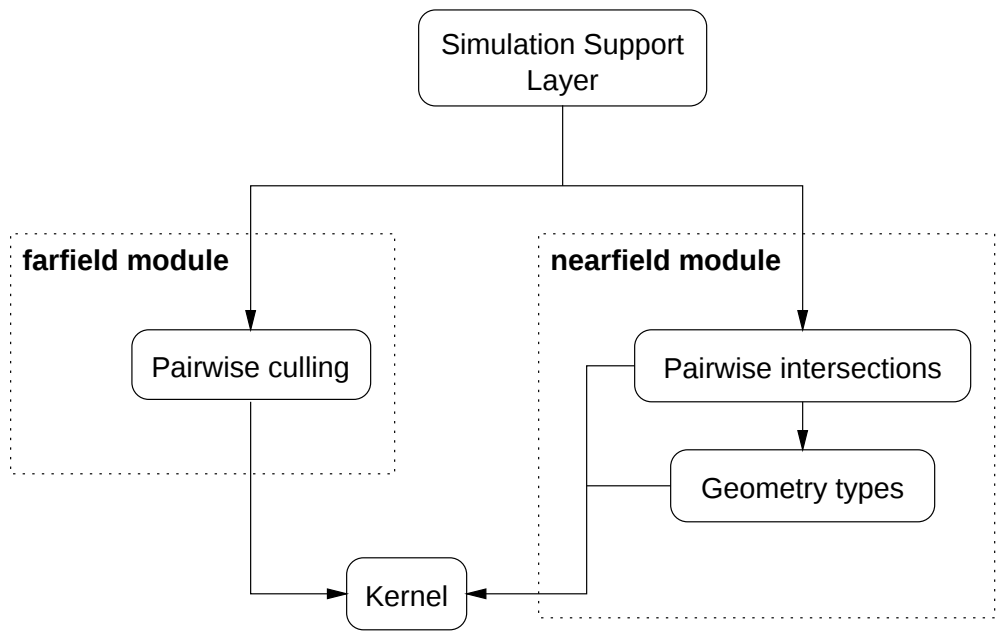


Figure 1: Collision Toolkit Architecture

The simulation support layer supports the integration of multiple distinct response/behavior solutions for different groups of collision models within the same application.

The system is modular, allowing you to re-use lower-level elements independently of the higher-level ones. This modularity also helps you to introduce Collision Toolkit functionality incrementally into your existing projects.

Module	Services Provided
Near Field	• Geometrical types for characterizing collision models • Intersection tests between each pair of nearby objects
Far Field	• Monitors the separation of object pairs

Collision Detection vs. Collision Response

The Collision Toolkit determines when objects are colliding and provides appropriate collision information that may be used by the MathEngine Dynamics Toolkit or any third-party packages.

No collision response (e.g., action taken as a result of collision detection) is performed as a result of collision detection. You are responsible for the response code that actually moves or affects the motion of your models.

The response to a collision event is typically a change in motion, or a change in the allowable motions of the elements in your scene. The response could be something simple such as preventing a character from advancing in a given direction; a projectile sticking to the surface it hits or very sophisticated such as an articulated toy falling down a staircase. The response behavior could be handled by application-specific code you have written yourself, by a more general module developed by yourself or within your company, or by a separate software package provided by a third party.

Integrating the Dynamics Toolkit

The Dynamics Toolkit can be used to provide physically realistic collision response, using information provided by the Collision Toolkit .

Figure 2: Integrating the Dynamics Toolkit into the Collision Toolkit shows how you can integrate this product into the Collision Toolkit's system. The Dynamics Toolkit Bridge obtains contact information from Collision Toolkit and sends this information to the Dynamics Toolkit in an appropriate format. The Bridge is responsible for keeping the two packages in sync, ensuring, for example, that the coordinate system updates for the collision models are correctly obtained from the Dynamics Toolkit.

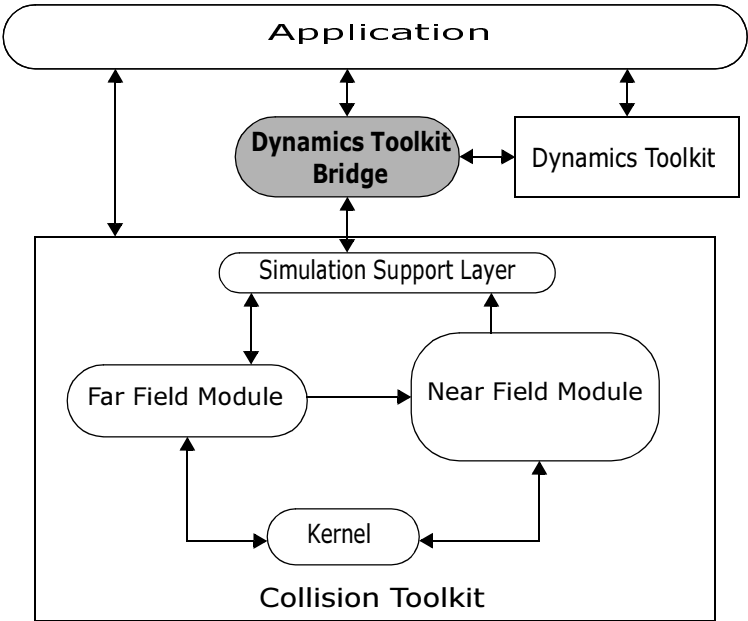


Figure 2: Integrating the Dynamics Toolkit into the Collision Toolkit

The Bridge relies on the Collision Toolkit's simulation support layer, which define a control flow that ensures Bridge operations are performed at the right times, and allows the Bridge itself to operate independently.

Integrating Your Own Tools and Third-Party Tools

You can integrate your own response code or any third-party tools in the same way that the Dynamics Toolkit is: by attaching it to a Bridge. Bridges let you simultaneously manage a variety of different response models and techniques within your application.

By treating in-house efforts as “products” on an equal footing with third-party solutions, you will build up an arsenal of tools that can be applied and experimented with in various combinations as your project evolves.

Figure 3: Collision Toolkit supporting multiple product integrations shows a high-level usage scenario which includes the Dynamics Toolkit and two other behavior-related toolkits, which are attached to the Collision Toolkit via product integration Bridges.

The scenario also includes a specialized module within your own code base which uses the Collision Toolkit to implement some sensor functionality needed in your application for, as an example, detecting if a game character is inside a room or not.

The operation of each of these Bridges occurs automatically each time step, triggered via the simulation support layer. In this scenario, your only responsibility is to initialize each depicted module, and to modify the parameters controlling how each operates. While this is happening, you can also access the Collision Toolkit directly to perform any other application-specific operations that you have not put under the control of the simulation support layer.

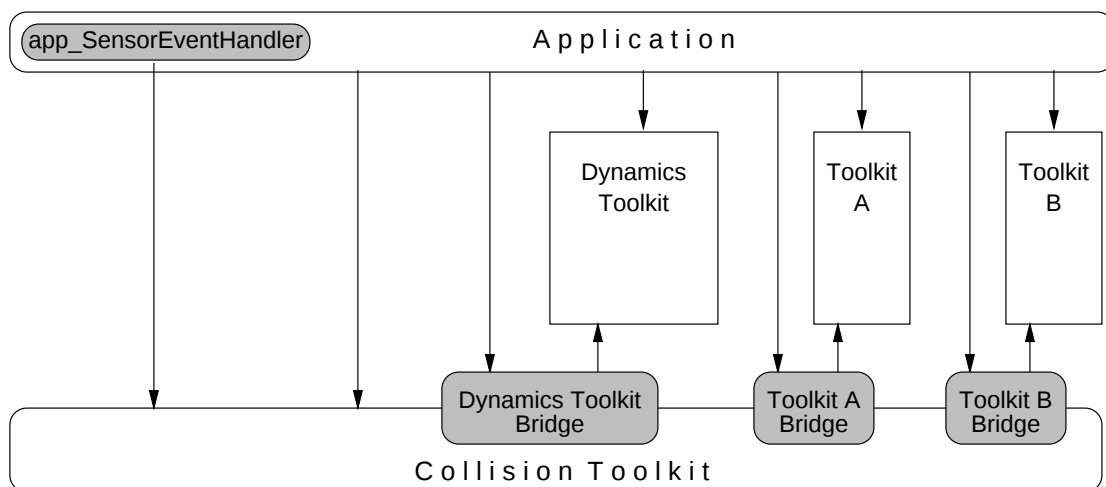


Figure 3: Collision Toolkit supporting multiple product integrations

Geometrical Types and Interactions

Geometrical Types

You specify the geometrical shape of your collision models by choosing from among the Collision Toolkit's library of geometrical types. The collision geometry should closely resemble the geometry of the 3D graphics model.

Simpler geometrical types requires simpler algorithms and less memory. Having a wide selection of geometry types available gives you a lot of flexibility in choosing trade-offs between performance, memory and geometrical accuracy. The Collision Toolkit's implementation-independent design lets you change the geometrical representation for each collision model without alerting the rest of your collision code. This means that you can continue experimenting with your collisions far into the development cycle.

Currently, the Collision Toolkit supports the following primitive geometrical types:

- Sphere
- Box
- Plane
- Cylinder
- Cone

Primitives do not undergo any internal triangulation to a common representation. They are truly primitive parametrized types: each internal representation is distinct, and consists of only a small number of scalar values.

The Collision Toolkit also supports the following non-primitive geometrical types:

Non-Primitive Type	Usage
Convex Mesh	Allows you to define an arbitrary convex surface
Height Field	Allow you to define a mesh with regular grid of height field
Particle System	Keep track of the particles in your particle system and determine their potential contacts with other collision models

Refer to *Chapter 6: Geometrical Types and Their Interactions* for a detailed description of each.

Interactions

When two 3D objects approach or come into contact with each other in your application, their associated collision models will be used to compute such information about their geometrical relationship, as their minimal surface-to-surface separation distance, or their points of contact. The algorithm that computes this information is called an intersection function or simply an interaction. Since each collision model may be represented with a different geometrical type, separate intersection functions are needed for each possible pair of geometry types that may interact with each other within your application.

The current set of intersection functions generate contacts optimized for use with the Mdt Library, so that you can create realistic physically-based responses.

Configuration Control and Code Size

The Collision Toolkit is a general purpose, comprehensive collision detection system that provides many geometrical types and intersection algorithms suitable for a wide variety of applications.

The Collision Toolkit also provides three levels of configuration control, so that you can minimize the memory and CPU resources you use:

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

You specify which binaries to load with your application, and which geometrical types and interactions are actually registered with the system. Only those will be linked to your application. You can even assign a configuration control for each type and for each interaction.

Contact Generation and Physical Simulation

You use intersection algorithms to produce intersection data, which, in turn, is used to compute some type of collision response. Exactly what you need in terms of intersection data depends on the needs of the response model you intend to apply to it.

Useful intersection data could be:

- Simple yes/no disjointedness test (are two objects touching each other?)
- Detailed list of edges and corners affected,
- Complete specification of the boundary of the intersection surface
- Estimates of the volume of intersection or the depth of penetration

The Collision Toolkit does not attempt to cover every single possibility. The determination of a response that is both physically-based and geometrically realistic is best supported by characterizing an intersection condition in terms of contact points. For this reason, the current set of intersection algorithms available in the Collision Toolkit are devoted to the generation of contact points, and to the building-up of a library of distinct contact generation algorithms reflecting different performance-accuracy trade-offs.

What is a Contact?

A contact occurs when two objects have a point in common. Penetration is a measure of how deep one object penetrates another, typically a distance by which one object has to be translated to avoid contact. For small penetration, 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 include:

- The location of the contact
- The normal, that is, the direction along which penetration is to be prevented
- An estimate of penetration depth

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 your collision response is a 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 coming-to-rest contact behavior usually requires at least three contacts, and can be very sensitive to geometric accuracy.

Our current set of contact generation algorithms have evolved and matured through their use in rigid-body dynamics packages such as the MathEngine Dynamics Toolkit. Such use has provided thorough and demanding testbeds for accuracy and robustness, as well as revealing optimization opportunities that specifically benefit the types of situations commonly encountered in physical simulation. The Collision Toolkit's contact generation algorithms can be used with any physically-based response system or with other types of response behavior such as an arcade style response, where the physics is approximated and/or exaggerated.

How Do I Use Contacts to Perform Collision Response?

No collision response takes place between any 2 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 also provided in order, for example, for a collision response algorithm to know how much "overshoot" to compensate for.

Non-Dynamics Related Functionality

The Collision Toolkit knows about the geometrical shape and movement of the models in your application, in order to handle collisions and contact situations. Once this is in place, you can treat the Collision Toolkit as a high-performance geometrical database, which can perform other geometric calculations at the same time that it handles your collision detection needs.

Sensors and Event Handling

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

You do this by inserting additional models into the Collision Toolkit that do not correspond to visually-rendered elements in your application, but define rather a spatial location. These models then function as sensors which you can use to detect which of your moving models are (for a game, say):

- Inside a given room
- Inside a “danger zone”
- Close to a bomb
- Within hearing distance of an enemy

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

You can use the simulation support layer to filter all sensor-related events directly to your event handler, ensuring that all semantic processing occurs in one place.

Rays

The Collision Toolkit's ray functionality can help you with many types of operations, including:

- See
- Occlude
- Shoot

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

You can use the global ray query which intersects a ray with the entire collision space, in which the space's internal representation is used to carry out an efficient determination of the “first hit”.

Far Field Functionality

Pairwise Proximity Culling

As the number n of collision models you use in your system increases, you will find a new problem emerging: 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 far field module. The far field module is a representation of the structure of 3D space and of the region of space occupied by each model present in it. It keeps track of nearby pairs efficiently by exploiting frame-to-frame coherence.

You use the Far Field by creating a collision space and then populating it with the collision models in your scene. When you modify the coordinate systems of the collision models, you need to inform the collision space so that it can update its internal state. At any point you can request the current list of nearby pairs.

Other Spatial-Global Queries

You can also use the Far Field moduli data structures and updating mechanisms to provide fast determination of other properties involving spatial relationships.

It can be used to quickly locate which models satisfy a particular spatial condition, such as being near to a particular region (e.g., sensors) or lying along a particular path through space (e.g., rays).

It is useful to think of a collision space as a container holding your collision models. The main feature of the collision space is that it orders locations in 3D space. Like any ordered container, you can insert and remove elements, and benefit from efficient update-sorting and fast global searches. It is within this broader vision that the Far Field module has been designed.

Alternative Implementations

The Far Field module offers a number of distinct representations for the structure of 3D space, allowing you to choose the one that is optimal for keeping track of the type of activity particular to your scene. The choice depends on the pattern of spatial distribution of your models, and the known constraints on how they can move about in your scene. For example, one of the Far Field's representations specializes for scenarios in which all your models stay close to a relatively flat plane.

Simulation Support Layer

The role of the Collision Toolkit is to provide you with the collision detection information you need. The Collision Toolkit's Simulation Support Layer addresses the problem of what you are going to do with it.

More precisely, it provides high-level support for integrating response solutions with the Collision Toolkit. This makes it easy to plug the Collision Toolkit's intersection data into products such as MathEngine Dynamics Toolkit. Other MathEngine behavioral solutions will be integrated with the Collision Toolkit in the same manner. It also makes it easy for you to attach your own solutions.

Figure 2: Integrating the Dynamics Toolkit into the Collision Toolkit shows how the Simulation Support Layer is supporting one such response solution, the Dynamics Toolkit. *Figure 3: Collision Toolkit supporting multiple product integrations* in the same section is a sketch of how multiple products can all be integrated simultaneously.

The Simulation Support Layer defines a control flow ensuring that operations are performed at the right times, and allows bridges to be written independently of the details of the Collision Toolkit.

Specific Product Integrations

Dynamics Toolkit Bridge

MathEngine's Dynamics Toolkit is a fast and robust rigid-body dynamics package that lets you create physically realistic simulations.

With the Collision Toolkit's Dynamics Toolkit Bridge, you have at your disposal a sophisticated collision response solution that takes full advantage of the Collision Toolkit and the Collision Toolkit's contact generation algorithms.

Figure 2: Integrating the Dynamics Toolkit into the Collision Toolkit shows how the two products fit together via the Collision Toolkit's Dynamics Toolkit Bridge.

The bridge will ensure that all necessary operations are carried out at the right times, and manage all memory needed to carry them out efficiently. These operations include:

- Identifying which collision models and collision events the Bridge is responsible for
- Updating the coordinate system of each collision model from its corresponding rigid body
- Obtaining contact intersection data for each collision event
- Preparing and sending contact data to the Dynamics Toolkit
- Ensuring that the dynamical properties of each Dynamics Toolkit contact are set to the values appropriate for the given pair of models

The bridge automates the entire process, and provides you with high-level control over each aspect of its activity. Use of the Bridge ensures that your combined use of the two packages is efficient and takes full advantage of the features available in both.

You can continue to adjust the Collision Toolkit's low-level state without affecting how the Bridge operates, for example you can:

- Change the geometrical types of individual collision models
- Change which intersection algorithms are used for individual pairs of collision models
- Insert and remove models from spaces
- Freeze/Unfreeze models

Using the bridge does not prevent you from handling other pairs of collision models using other techniques, nor from using the Collision Toolkit for other purposes at the same time. You can attach other product integrations, write your own, or access the Collision Toolkit's functionality directly. You can also re-use the results obtained by the Dynamics Toolkit bridge for additional processing, such as changing visual properties, adding sound effects or applying state transitions to your application event handler.

RenderWare Models

This integrates RenderWare models directly into the Collision Toolkit, allowing you to treat them the same way as the Collision Toolkit's native geometry types. This also means that they can interact transparently with other collision models present in the system.

Chapter 2 • Installing the Collision Toolkit

Overview

The MathEngine Collision Toolkit is installed with the MathEngine Dynamics Toolkit. See the *MathEngine Dynamics Toolkit Developer's Guide*.

Chapter 3 • Getting Started

Overview

This chapter shows you how to write simple, powerful programs using the MathEngine Collision Toolkit. First we show you how to use it alone, and then in combination with the MathEngine Dynamics Toolkit.

Organizing Your Programs

This section describes how to write a simple program using the Collision Toolkit.

Including the Required Header Files

The simplest applications using the Collision Toolkit 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 MathEngine Toolkits distribution.

Linking the Required Libraries

The libraries for the Collision Toolkit are located in the `bin` directory of the MathEngine Toolkits distribution:

- The Collision Toolkit libraries are prefixed with `mcd`.
- You always need `McdFrame.lib`. You will usually also want `McdPrimitives.lib` if you want to use any of the Collision Toolkit's primitive geometry types.
- The `McdDtBridge.lib` library is only required if you are using the bridge to the Dynamics Toolkit.

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() );  
McdPrimitivesRegisterTypes();  
McdPrimitivesRegisterInteractions();
```

Defining Collision Models

Now we can create primitive geometries:

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

And create collision models from these shapes:

```
McdModelID ballCM = McdModelCreate(ball_prim);
```

Collision models are the principal objects in the Collision Toolkit. Their position and orientation in 3D space is represented as a 4X4 transformation matrix which you must allocate yourself:

```
MeMatrix4* ballGraphicTransform = (MeMatrix4*) malloc (sizeof (MeMatrix4));
McdModelSetTransformPtr(ballCM, ballGraphicTransform);
```

Creating Collision Spaces

We'll create a collision space to keep track of which of the possible pairs of collision models are nearby to each other. Rather than creating space for the maximum number of pairs, we are leaving space for a smaller number:

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

Finally, we insert the models in the collision space, and build the space:

```
McdSpaceInsertModel(space, ballCM);
McdSpaceBuild(space);
```

Testing for Collisions

You can test for potential collisions between a pair of collision models by calling the Collision Toolkit's dispatcher, which will find an appropriate algorithm for the given pair of collision models, and compute the desired intersection data. We test for collision at every frame refresh:

```
#define MAX_NUM_PAIRS 10
McdSpacePair pairs[MAX_NUM_PAIRS];
int nPairs = McdSpaceGetNewPairs(space, pairs, MAX_NUM_PAIRS);
McdInteractionResult result;

McdIntersect(pairs[0], &result);
```

We can now adjust the velocity of the ball, based on the collision result:

```
MeReal ball_velocity[3];
```

```
/* calculate change of velocity (along normal) scalar */  
float k = -2*Dot(ball_velocity, result.normal);  
/*change velocity*/  
MultiplyAddVec(ball_velocity, ball_velocity, k, result.normal);
```

BallHitsWall1.c: A Sample Program

This example shows the complete program that simulates a ball that bounces off a wall and then bounces off the floor when thrown.

The motion of the ball is handled by the application program using a `move()` function. When the Collision Toolkit 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.

You need two calls to obtain a complete list of all potentially colliding pairs of collision models. Collision space distinguishes between “newly nearby” pairs and pairs that were already nearby at the previous time step.

We use a cube as collision model representing the floor. It allows a simpler and therefore faster intersection algorithm to be applied.

What This Program Shows

The program shows how to use the Collision Toolkit to:

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

The code for the motion algorithm shows you 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 also show how the new positions computed from the motion algorithm are used to update both the collision model and the rendered model.

```
/*
  Copyright MathEngine PLC 2000
  www.mathengine.com

  $Name:  $

  $Id: BallHitsWall1.c,v 1.5 2000/07/17 16:00:31 chris Exp $
*/
/*
  Overview:

  BallHitsWall1 uses the Collision Toolkit alone, without the Dynamics
  Toolkit. The scenario: a ball is thrown at a wall, bounces off the
  wall, bounces off the floor, and then exits the scene. The motion of
  the ball is not being driven by the dynamics engine, rather, it is
```

```

    scripted by hand.

    Note: There is no gravity in this simulation.
*/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "McdFrame.h"
#include "McdPrimitives.h"
#include "MeMath.h"
#include "MeViewer.h"

/* define to get cube collision */
#if 0
#define USE_CUBE
#endif

/*
    Globals
*/
#define MAX_BODIES 10
/*
    render context
*/
RRender *rc;

/*
    Worlds and Spaces
*/
McdSpaceID space;

/*
    Collision reps
*/
McdModelID groundCM;
/*
    collision model geometry representations
*/
McdGeometryID plane_prim, box_prim, box_prim1, ball_prim;

/*
    Graphics reps
*/
RGraphicsDescription *groundG;
RGraphicsDescription *boxG;
RGraphicsDescription *ballG;
/*
    time step
*/

MeReal step = 0.02f;
/*
    ball radius
*/
MeReal radius = 0.5;
/*

```

```

    wall dimensions
*/
MeReal wallDims[3] = { 0.5f, 2.0f, 5.0f };
/*
    floor dimensions
*/
MeReal floorDims[3] = { 6.0f, 0.05f, 5.2f };

/*
    Colors
*/
float orange[3] = { 1.0f, 0.4f, 0.0f };
float green[3] = { 0.0f, 1.0f, 0.0f };
float blue[3] = { 0.0f, 0.598f, 0.797f };

/*
    rotate ground plane collision model 90 about x-axis
*/
MeReal groundTransform[4][4] =
    { {1, 0, 0, 0}, {0, 0, -1, 0}, {0, 1, 0, 0}, {0, 0, 0, 1} };

/*
    help text
*/
char *help[2] =
{
    "Press enter to reset",
    "Press Spacebar to shoot"
};

/*
    Structure that holds information about an object
*/
typedef struct Object
{
    McdModelID m_object;
    MeMatrix4 m_TM;
    MeReal *m_position;
    MeReal m_velocity[3];
}
Object;

/*
    Object instances
*/
Object box;
Object ball;

/*
    Set vector 'a' to be the sum of vector 'b' and the product of scalar
    'x' and vector 'c'. This is mainly used to drive the motion of the
    ball.
*/
void MultiplyAddVec(MeReal *a, const MeReal *b, const MeReal x,
    const MeReal *c)
{
    int i;

```

```

        for (i = 0; i < 3; i++)
            a[i] = b[i] + x * c[i];
    }

    /*
    Shoot the ball
    */
    void shoot(void)
    {
        ball.m_TM[3][0] = 0.0;
        ball.m_TM[3][1] = 4.0;
        ball.m_TM[3][2] = 0.0;
        ball.m_velocity[0] = 2.0;
        ball.m_velocity[1] = -2.0;
    }

    /*
    Reset the scene
    */
    void reset(void)
    {
        MeMatrix4MakeIdentityTM(ball.m_TM);
        ball.m_TM[3][0] = 0.0;
        ball.m_TM[3][1] = 4.0;
        ball.m_TM[3][2] = 0.0;
        ball.m_velocity[0] = 0.0;
        ball.m_velocity[1] = 0.0;
        ball.m_velocity[2] = 0.0;

        MeMatrix4MakeIdentityTM(box.m_TM);
        box.m_TM[3][0] = 3;
        box.m_TM[3][1] = wallDims[1];
        box.m_TM[3][2] = 0;
    }

    /*
    Update the positions of the objects
    */
    void move()
    {
        MultiplyAddVec(box.m_position, box.m_position, step,
            box.m_velocity);
        MultiplyAddVec(ball.m_position, ball.m_position, step,
            ball.m_velocity);
    }

    /*
    Handle collision for a given list of McdModelPairLinks. If a
    collision is detected, change velocity of the ball, given the
    contact information
    */

    void handleCollision(McdModelPair ** pairList, int NPairs)
    {
        static McdContact contacts[10];
        float k;
        int i;

```

```

McdIntersectResult result;

result.maxContactCount = 10;
result.contacts = contacts;

for (i = 0; i < NPairs; i++)
{
    McdModelPair *pair = pairList[i];

    /*
     * get contact info
     */
    McdIntersect(pair, &result);
    printf("\nNumber of contacts: %d", result.contactCount);
    /*
     * calculate change of velocity (along normal) scalar
     */
    k = -2 * MeVector3Dot(ball.m_velocity, result.normal);
    /*
     * change velocity
     */
    MultiplyAddVec(ball.m_velocity, ball.m_velocity, k,
        result.normal);
}
}

/*
 * This function is called every frame by the renderer.
 */
void tick(RRender * rc)
{
    static McdModelPair* pairs[100];
    int NPairs;

    RStopTimer(kRRender);
    RStartTimer(kRDynamics);
    move();
    RStopTimer(kRDynamics);
    RStartTimer(kRCollision);
    /*
     * evolve space
     */
    McdSpaceUpdate(space);
    McdSpacePairIteratorID iter;
    iter = McdSpaceGetHelloPairIterator(space);
    NPairs = McdSpaceGetHelloPairs(space, &iter, pairs, 100);
    handleCollision(pairs, NPairs);
    iter = McdSpaceGetStayingPairIterator(space);
    NPairs = McdSpaceGetStayingPairs(space, &iter, pairs, 100);
    handleCollision(pairs, NPairs);
    RStopTimer(kRCollision);
    RStartTimer(kRRender);
}

void cleanup(void)
{
    /*
     * memory release. Collision objects must be explicitly deleted

```

```

        by the user.
    */
    RDeleteRenderContext(rc);
    McdGeometryDestroy(plane_prim);
    McdGeometryDestroy(box_prim1);
#ifdef USE_CUBE
    McdGeometryDestroy(box_prim);
    McdModelDestroy(box.m_object);
#else
    McdGeometryDestroy(ball_prim);
    McdModelDestroy(ball.m_object);
#endif
    McdModelDestroy(groundCM);
    McdSpaceDestroy(space);
    McdTerm();
}

/*
Main Routine
*/
int main(int argc, const char **argv)
{
    /*
    Initialize renderer
    */
    const RRenderType render = RParseRenderType(&argc, &argv);
    rc = RNewRenderContext(render, kRQualitySmooth);

    /*
    Initialize toolkit
    */

    McdInit(McdPrimitivesGetTypeCount());

    McdPrimitivesRegisterTypes();
    McdPrimitivesRegisterInteractions();

    space =
        McdSpaceAxisSortCreate(McdAllAxes, MAX_BODIES, 2 * MAX_BODIES);
    McdPairHandlerRegisterSpace(space);
    /*
    ground plane
    */
    groundG =
        RCreateCube(rc, 2 * floorDims[0], 2 * floorDims[1],
            2 * floorDims[2], blue, 0);
    RSetTexture(groundG, "checkerboard");
    /*
    top of plane at y=0
    */
    groundG->m_matrixArray[13] = -floorDims[1];
    plane_prim = McdPlaneCreate();
    groundCM = McdModelCreate(plane_prim);
    McdSpaceInsertModel(space, groundCM);
    McdModelSetTransformPtr(groundCM, groundTransform);
    /*

```

```

    optimize
*/
/* Inform the system that groundCM 's transform will not change value*/
McdModelFreezeInSpace(groundCM);

/*
    box
*/
box_prim1 = McdBoxCreate(2*wallDims[0], 2*wallDims[1], 2*wallDims[2]);
MeMatrix4MakeIdentityTM(box.m_TM);
box.m_TM[3][0] = 3;
/*
    raise above floor
*/
box.m_TM[3][1] = wallDims[1];
box.m_TM[3][2] = 0;
box.m_position = &box.m_TM[3][0];
box.m_object = McdModelCreate(box_prim1);
McdModelSetTransformPtr(box.m_object, box.m_TM);
McdSpaceInsertModel(space, box.m_object);
boxG =
    RCreateCube(rc, 2 * wallDims[0], 2 * wallDims[1],
        2 * wallDims[2], blue,
        (MeReal *) McdModelGetTransformPtr(box.m_object));
#ifdef USE_CUBE
    box_prim = McdBoxCreate(2*radius, 2*radius, 2*radius);
#else
    ball_prim = McdSphereCreate(radius);
#endif

    MeMatrix4MakeIdentityTM(ball.m_TM);
    ball.m_TM[3][0] = 0;
    ball.m_TM[3][1] = 4;
    ball.m_TM[3][2] = 0;
    ball.m_position = &ball.m_TM[3][0];

#ifdef USE_CUBE
    ball.m_object = McdModelCreate(box_prim);
#else
    ball.m_object = McdModelCreate(ball_prim);
#endif

    McdModelSetTransformPtr(ball.m_object, ball.m_TM);
    McdSpaceInsertModel(space, ball.m_object);

#ifdef USE_CUBE
    ballG = RCreateCube(rc, 2*radius, 2*radius, 2*radius, orange,
        (MeReal*)McdModelGetTransformPtr(ball.m_object));
#else
    ballG = RCreateSphere(rc, radius, orange,
        (MeReal *) McdModelGetTransformPtr(ball.m_object));
    RSetTexture(ballG, "ME_ball3");
#endif

/*
    Build Space
*/
McdSpaceBuild(space);

```

```

    /*
    Keyboard callbacks
    */

#ifdef PS2
    RUseKey('\r', reset);
    RUseKey(' ', shoot);
#else
    RUsePadKey(PADsquare, reset);
    RUsePadKey(PADcircle, shoot);
#endif

    /*
    On screen help text
    */
    RCreateUserHelp(help, 2);
    rc->m_title = "BallHitsWall1 v.1.0  tutorial - www.mathengine.com";

    /*
    cleanup after simulation
    */
#ifdef PS2
    atexit(cleanup);
#endif
    /*
    run the simulation loop
    */
    RRun(rc, tick);

    return 0;
}

```

Supporting the Dynamics Toolkit

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

The Dynamics Toolkit's solver uses this contact information so that it can respond to the contact condition, computing a motion that allows them to come into contact with each other: to touch (rather than pass through) each other.

The Collision Toolkit provides a Bridge to the Dynamics Toolkit. This Dynamics Toolkit Bridge manages all control flow and data exchange between the two toolkits, 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 the Dynamics Toolkit instead of in the application's code. To support this, the Dynamics Toolkit Bridge (type McdDtBridge), computes the appropriate contact information for you, and communicates it automatically to the Dynamics Toolkit.

BallHitsWall2.c: Let's Get Dynamical

This example reproduces the identical behavior of the previous example, only using the Dynamics Toolkit to control the motion. Aside from calls to the Dynamics Toolkit, no motion code appears anywhere in the program: it is a 3D simulation involving physics and geometry without any code involving physical nor geometrical computations! In addition, there is no need for you to create or manipulate McdContactData objects at all. The complete code of BallHitsWall2.c is distributed with the Collision Toolkit.

You will need the following key bridging operations:

Initializing the Bridge

You have to initialize both the Collision Toolkit and Dynamics Toolkit packages, of course, but now you must also create the bridge object connecting them together:

```
McdDtBridgeInit();
McdDtBridgeID bridge = McdDtBridgeCreate();
```

Giving shape to an MdtBody

Specify the collision model that will represent the geometrical shape of each MdtBody:

```
MdtBodyID ball = MdtBodyCreate();
MdtBodyEnable(ball);
McdDtBridgeSetBody(ballCM, ball);
```

What This Program Shows

The Bridge to the Dynamics toolkit handles all communication and synchronization between the Collision Toolkit and the Dynamics Toolkit. The only responsibility of the application program is to specify the geometrical and dynamical properties of the simulation (via the Collision Toolkit and the Dynamics Toolkit, respectively).

BallHitsWall3: Complex Behavior

Although the previous example produces very simple behavior, it is written at a very high level of generality: very little of code depends on the actual geometries you chose, and very little code is specific to the kind of behavior you wanted to achieve.

Programming at this level of abstraction opens up many new opportunities. Small changes to your code can result in very rich and complex changes in behavior. Dynamical and geometrical properties can also be varied and tuned without disturbing the rest of your code.

This is in contrast with the first example (`BallHitsWall1.c`) in which we wrote specialized code that handled only the specific collision conditions. Such code must often be rewritten as the scenario changes, and must be constantly re-checked for special boundary conditions. For example, the above `handleCollision()` function does not anticipate the “corner” condition in which the ball hits the wall and the floor at the same time.

Below we will demonstrate a variety of complex behaviors (which you wouldn't want to hand-program each time) that can be produced and experimented with by single-line-of-code changes to the program. We will look individually at changing properties that are controlled by the Collision Toolkit, and by properties that are controlled by the Dynamics Toolkit.

Changing Geometrical Properties

Replace:

```
ballCM = McdModelCreate( McdSphereCreate(radius) );
```

with:

```
ballCM = McdModelCreate( McdBoxCreate(radius, radius, radius) );
```

The ball has become a flying box, complete with realistic spinning and toppling motions. Note how edge-on-wall-edge, box-face-on-wall-edge and all other contact combinations are automatically handled with geometric and physical realism.

Note that you might also want to use the Dynamics Toolkit to adjust the moment of inertia as you switch geometries, to capture the effect of preferred-axis spinning.

Changing Dynamical Properties

Let's return the geometrical specification in its original state (a ball), and try instead playing with some of the dynamical properties.

Try making the ball into a demolition ball—something much heavier than the wall, and that doesn't bounce much:

```
MdtBodySetMass( ball, 100 );
MdtBodySetMass( wall, 10 );

MdtConContactParams *params
    = McdDtBridgeGetContactParams(McdGetDefaultMaterialID(),
                                   McdGetDefaultMaterialID());
params->bouncyness = 0.0f;      /* min bouncyness */
```

Now the ball knocks over the wall, which comes tumbling down onto the ball, coming to rest at some small angle, trapping the ball. Try coding that by hand!

Let's try something different. We'll add a floor hinge at one end of the wall, so it won't fall over:

```
/*create hinge joint and orient in space*/
MdtHingeEnable(hinge, world);
```

Now when the ball hits, the wall acts like a door, swinging on its hinge. It is an unconstrained hinge, so it swings a full 360 degrees, coming around to smack the ball back again!

The complete code of this sample program is distributed with the Collision Toolkit.

Chapter 4 • Advanced Features

Collision Models and the Geometry Library

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 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: If you are integrating with Dynamics Toolkit, the `McdModel` by default shares the `MdtBody`'s transformation matrix, in which case the above step is not required.

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.

Culling Management and Culling Optimizations

The “Freeze” Feature

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.

```
unsigned int McdModelFreezeInSpace (McdModelID cm);
```

Alternative Implementations and Specializations

The Far Field module offers a coordinate sorting implementation. You can vary the number and combination of axes that are used for sorting.

For example, if all your models stay close to a z-plane, then you can benefit from sorting on only the x- and y-axes.

Chapter 5 • Integrating with the Dynamics Toolkit

Overview

This chapter shows you how to use the MathEngine Collision Toolkit with the MathEngine Dynamics Toolkit.

An `McDdtBridge` object is responsible for all operations and communication between the Collision Toolkit and the Dynamics Toolkit. In this chapter, we will often refer to this object simply as the Bridge.

Relative Transforms

Both `MdtBody` objects and `McdModel` objects use the same `MeMatrix` type to represent their coordinate system. When you associate a collision model with a Dynamics Toolkit body via:

```
McdDtBridgeSetBody( McdDtBridgeID, McdModelID, MdtBodyID );
```

the collision model shares the transformation matrix already allocated in the `MdtBody` object: subsequent calls to `McdModelGetTransformPtr()` will return a pointer to the `MdtBody`'s transform. There is no need to allocate a transform for that `McdModel`. If you attempt to use `McdModelSetTransformPtr()` after this point, you will get an error message. You will also get an error message if you use `McdDtBridgeSetBody()` on a `McdModel` for which a transform has already been set.

Sometimes the default behavior of `McdDtBridgeSetBody()` is not appropriate: you might need an offset between the rigid body's coordinate system and that of the collision model. In this case you can follow that call with:

```
McdDtBridgeSetRelativeTransformToBodyPtr( McdDtBridgeID, McdModelID,
                                           MeMatrix* relTM,
                                           MeMatrix* compoundTM );
```

where `relTM` and `compoundTM` each point to a `MeMatrix` that you have allocated. Each time step, the `MdtBody`'s transform will be compounded by `relTM`, and the result written to `compoundTM`. Subsequent calls to `McdModelGetTransformPtr()` will return `compoundTM`.

When you perform your application-specific updating, you may also need an additional alignment transform to account for any offset between the collision model's and the graphics model's coordinate systems, that is, when the center of mass of the model does not correspond with its local origin. This is useful, for example, for insuring that the `MdtBody`'s origin lies at the center of mass of the `McdModel`.

Coordinate Systems and Implied Geometry

The Dynamics Toolkit handles the dynamical properties of a system, having no explicit representation for geometrical properties. The Collision Toolkit informs the Dynamics Toolkit about geometry indirectly, by giving it a set of `MdtContact` objects describing the contact conditions between two `MdtBody` structures.

The Dynamics Toolkit is not completely geometry-free, however. Some properties associated with a body can imply certain assumptions about its geometry. These properties are:

- Constraints
- Constraint limits
- Inertia matrix

These properties can imply a particular position or orientation to the body coordinate system, which may be different from that of the geometry types in the Collision Toolkit. For example, the constraints may be based on the assumption of a long bar with one end at the origin, but the Collision Toolkit's box geometry defines a bar with the origin in the middle of the bar. Be careful that these properties are consistent with the geometrical shape and relative orientation you assign to the associated collision model.

Inertia Matrix

The values in the rigid body's inertia matrix also imply a certain orientation of its coordinate system. Make sure this aligns properly with that of the collision model. Be careful when you change the geometrical shape of a collision model that the corresponding rigid body's inertia matrix reflects this. In some situations this discrepancy may have a minor effect, but in other situations it could be quite apparent.

The effect is most important when you create a box with one axis significantly longer or shorter than the other ones.

Composite Geometries

Currently, the Collision Toolkit offers only single primitive geometry types. You can nevertheless create a composite geometry for a body by associating multiple collision models with it:

```
McdModelID model1 = McdModelCreate( McdSphereCreate(1) );
McdModelID model2 = McdModelCreate( McdSphereCreate(2) );
MeMatrix *m2Rel, m2Compound;

MdtBodyID b;

McdDtBridgeSetBody( bridge, model1, body );
McdDtBridgeSetBody( bridge, model2, body );
McdDtBridgeSetRelativeTransformPtr( bridge, model2, m2Rel, m2Compound );
```

Now `model1` is sharing body's coordinate system, and `model2`'s is offset by `m2Rel`. You must provide the value of `m2Rel`, and you can modify it at any point in the simulation. `m2Compound`'s value will be written anew every time step, by compounding `m2Rel` with `b`'s transform.

If the two `McdModel` "pieces" in the composite are arranged such that they are touching each other, you must tell the Collision Toolkit to ignore that interaction, otherwise undesired contacts will be produced:

```
McdModelSetInteractionDisabled( model1, model2);
```

For an example program, see `Chair.c`, which shows how to build a "chair" rigid body composed of multiple collision model pieces.

Material Properties

A Dynamics Toolkit `MdtContact` object contains both geometrical and dynamical properties. The geometrical properties can be obtained from the Collision Toolkit's computations.

The Dynamic Toolkit `McdDtBridge` offers a material identifier for individual bodies, and a material table where you can specify the values of the dynamical properties for each pair of ID values. There is also a callback for additional modifications.

Interactions with the Static Environment

The Bridge will ensure that contacts are generated not only when rigid bodies touch other rigid bodies, but also when a rigid body touches collision models that are **not** associated with a rigid body. Must call

```
McdBridgeSetBody(bridge, model, 0);
```

on non-physical models to activate this functionality.

Good examples are the non-moving elements of the environment, such as walls, floors, obstacles, etc. These are described by collision models, but do not need to be associated with rigid bodies. They have to be inserted into a collision space in order for collisions with them to be detected, but the Bridge does not need to be informed of their existence directly. When encountered, the Bridge considers them as part of the Newtonian *reference frame*.

Optimizations

Try to choose intersection algorithms that generate the least number of contacts, since a low number of contact increases the speed. Using sphere geometries is sometime a reliable way of achieving this, but is not always appropriate.

For example a box resting on a surface (e.g., a table) would need at lest three points of contacts: with only two points of contact, the box could rotate about the axis passing by these two points and sink into the surface. Hence, the trade-off is that the accuracy may decrease and that objects may become less stable.

Using relative transforms involves the overhead of allocating (memory) and multiplying (performance) two `MeMatrix4` objects.

Chapter 6 • Geometrical Types and Their Interactions

Overview

The Collision Toolkit 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 imply less memory to hold the representation, and simpler algorithms that operate on them. Having a wide selection of geometry types available allows you a lot of flexibility in choosing tradeoffs 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.

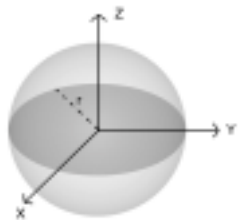
Geometrical Primitives

Primitive geometrical types are shapes defined by a small and fixed number of parameters. They are lightweight and fast. 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:

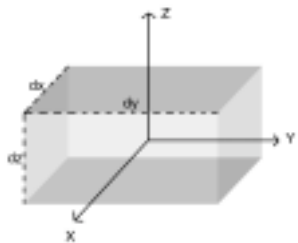
Sphere

Ball
Particle
Planet
Head



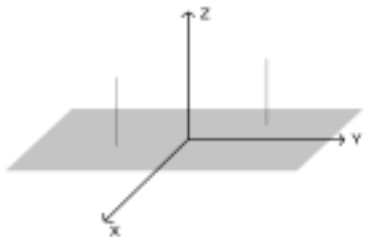
Box

Book
Skyscraper
Gambling Dice
MetalBar



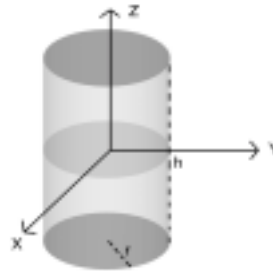
Plane

Floor
Wall
Ceiling
Ship-Deck



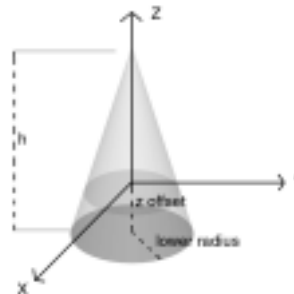
Cylinder

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 indicates the parameters used to specify each. There are two conventions common to all primitives:

- z is the special axis¹, 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.

See *Intersection Functions on page 57* for a figure indicating which pairs of primitive types can interact with each other. The missing ones are not implemented yet; the plane-plane case is never expected to be needed. The others will be implemented in the coming releases.

Registering Geometrical Types and Intersection

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

```
void McdInit( McdPrimitivesGetTypeCount());  
void McdPrimitivesRegisterTypes();  
void McdPrimitivesRegisterInteractions();
```

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 McdConeRegisterType();  
void McdConeConeRegisterInteraction();
```

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 tradeoff between memory use and geometrical accuracy. This flexibility is due to the fact that the Collision Toolkit non-primitive geometry types do not represent curved surfaces exactly, but approximate them by a set of discretized surface elements.

ConvexMesh

The ConvexMesh geometry type allows you to specify closed surfaces as a set of convex polygonal meshes. A restriction is that the resulting shape must be a surface which is everywhere convex.

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:

```
McdConvexMeshCreateHull(MeVector3* vertices, int vertexCount,
                        MeReal fatnessRadius );
```

You must first register the convex mesh geometry type with the system, and 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:

```
Int geoTypeCount = McdPrimitivesGetTypeCount() + 1;
McdInit( geoTypeCount );
McdPrimitivesRegisterTypes();
McdConvexMeshRegisterType();
McdPrimitivesRegisterInteractions();
McdConvexMeshRegisterInteractions();
```

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. Consult the Collision Toolkit's reference manual for an exact list.

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 McdRGHeightFieldCreate( McdReal* heightArray,
                                           int xGridCount, int yGridCount, cdReal xIncrement, McdReal yIncrement,
                                           McdReal xOrigin, McdReal yOrigin );
```

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:

```
Int geoTypeCount = McdPrimitivesGetTypeCount() + 2;  
McdInit( geoTypeCount );  
McdPrimitivesRegisterTypes();  
McdConvexMeshRegisterType();  
McdRGHeightFieldRegisterType();  
McdPrimitivesRegisterInteractions();  
McdConvexMeshRegisterInteractions();  
McdRGHeightFieldRegisterInteractions();
```

As with the convex mesh type, the height field type currently interacts with some, but not all of the other types. Consult the Collision Toolkit's reference manual for an exact list.

Intersection Functions

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

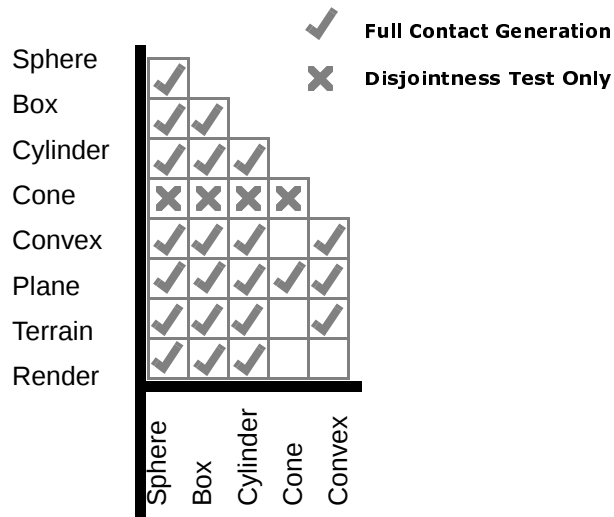


Figure 1: Development State of the Intersection Functions of All Pairs of Geometrical Types

MathEngine Toolkits Glossary

Acceleration	Time rate of change of velocity, or how fast velocity is increasing.
Actuator	A mechanical device for moving or controlling something, such as a motor, a lever, a piston, and so on. Actuators can be implemented with forces and torques or with velocity constraints.
Angular Velocity	The rate of rotation about an axis. This is a vector quantity whose direction points along the instantaneous axis preserved by the rotation and the magnitude is the rate of rotation about this axis. It is usually expressed in radians or revolutions per second, but can be expressed in any unit of time.
Ballistics	Ballistics is the science of the motion and behavior of projectiles, missiles, bullets and other similar objects in flight submitted to a gravitational field. For MathEngine, a projectile is a SRB (single rigid body) and is not <i>in contact</i> with anything – including the atmosphere. We do not calculate the effect of air drag on a projectile's flight. Projectiles, such as bullets, travel in a parabolic trajectory when submitted to a gravitational field or one that becomes more and more curved as range increases and velocity drops off.
Ballistic Objects	In a games development environment, ballistics objects are usually single rigid body projectiles that are subject to ballistic calculations without contacts or impulses.
BSP Tree	Binary Space Partitioning (BSP) tree. It is a method for describing the geometry of objects in 3D graphics. BSP trees are tools for organizing this geometry and extracting information about geometrical relationships. In 3D worlds they are often used for visibility determination, and hidden surface removal.
Cartesian Coordinates	A coordinate system which is based on orthogonal axes. This is the standard x, y and z coordinate system, in contrast to polar coordinates on a sphere. In many 3D graphics applications, x usually stands for left-right position, y describes vertical position and z gives the position along the line of sight, (i.e., depth). Traditionally, graphics viewer use Left Handed system.
Center of Mass	The point in a body or system of bodies at which the whole mass may be considered as concentrated. For the purpose of dynamics calculations, a body or system of bodies can be abstracted to a point mass located at its center of mass coordinates, simplifying the computation greatly.
Collision Detection	A software process that determines if two objects are geometrically interpenetrating or touching. If a collision took place it then determines the location and the local geometry of the object near the point of contact. Collision detection also determines the spatial relationships

between objects, such as the approximate separation distance between them.

Collision Model

A geometrical representation of an object used for collision detection. MathEngine supports a variety of primitives such as sphere, box, plane, cylinder and cone. The collision model may be identical to the model used for rendering or maybe an approximation that allows for faster collision detection. Collision models can also be made up of a combination of collision primitives

Collision Response

The change of motion of an object which results from a collision with another object of the world, or when a joint limit is reached. Note that collision response is related to the dynamics properties of the object in contacts.

Constraint

An external restriction on the motion of an object or body. MathEngine uses two kinds of constraints: joints and contact. An *equality* restriction could be that a joint attached to a specific location of a body should not be allowed to wander off this point, that is, their attachment coordinates should be the same. An *inequality* restriction could be a ball placed in a room: allowed to move freely on the floor as long as it stays inside the area bordered by the walls.

These restrictions are used to connect objects together, such as connecting two rigid bodies with a hinge, or to impose motion in a joint by applying a force limited actuator. Constraints can be equality restrictions as is the case for joints between bodies, or inequality restriction as is the case with contact constraints and joint limits. Constraints are called ideal or non-ideal according to whether or not they dissipate energy. A ball and socket joint is an ideal constraint but a box friction contact constraint is non-ideal.

Constraint Equations

A mathematical relation between the coordinates of several bodies..

Culling

Culling is used to describe the process of selecting items for removal from a list. For example, the process of eliminating certain polygons that shouldn't be seen before drawing a scene.

Cycle

A cycle is used to describe the effect of the force applied to a rigid body in an articulated chain of bodies. The force is relayed through each body in the chain and is eventually applied, at least in part, back to the original body.

Convex Object

A geometrical object with no holes in it, such as a sphere or a cube. More precisely, an object is convex if given any two points of the interior or the surface of the object, all the points that lie on the line segment between those two points are also inside or on the surface of the object. A doughnut is not convex for instance.

Concave Object	Nonconvex object.
Damping	This is a resistance to the motion of a body. It results in the body slowing down and being gradually brought to rest.
Differential Index	The number of times a DAE must be differentiated to obtain a standard set of ODEs, is the index of the DAE. If the index is one, the system may be solved nearly as simply as if it were an ODE.
Dynamics	Dynamics is the part of mechanics (which, of course is a branch of physics) that is concerned with the causes of accelerating motion. It describes the change in motion of objects or particles using the concepts of force and mass. It is governed by the three basic laws of motion, called Newton's laws. See Newton's Laws
Dynamic Simulation	Dynamics simulation is achieved by attaching physical properties and applying forces to objects, and computing the resulting behavior according to the physical laws of motion.
Force	A force is most easily described as a push or a pull – it is what causes the state of motion (momentum) of an object to change. If you want something to change the way it's moving, (change its direction or make it go faster, for example) you must apply a force to it. Forces are vector quantities with orientation and magnitude. The bigger the force, the faster the state of motion will change. In 3D games' worlds, forces typically include gravity and wind.
Force-limited Actuator	This is a special model of actuator provided by the MdtKea Library. It allows you to control the relative velocity of two bodies connected by a prismatic joint using an external source such as mouse or keyboard input. You set the desired target velocity, which will be achieved provided that the force required to do so is less than the limit specified, otherwise, the maximum force is applied. Force limited actuators respond very quickly and are inherently stable; they are the best way to introduce user controlled motion in a joint in the Kea library.
Frame	A single rendered screen of graphics,
Frame Rate	The number of screen images displayed per second, usually abbreviated by <i>fps</i> (frame per second). Common frame rates are 60fps for games consoles, 25fps for PAL video, and 30fps for NTSC video, although other speeds may be used
Friction Force	The force that resists relative motion between two bodies in contact. When bodies are sliding, this force opposes the relative motion and dissipates energy. Otherwise, this force prevents the bodies from

sliding, in which case it does not dissipate energy. In both case, friction force is related to normal force, that is, the force that prevents inter-penetration of the two bodies

GJK Algorithm

A fast method of doing collisions between a set of convex objects.

GJK Engine

The part of the collision detection software that implements the GJK algorithm.

Gravity

In a Newtonian world, gravity is a force that attracts all objects to each other. Gravitational force between any two given objects is proportional to the products of their masses and inversely proportional to the square of the distance that separates them (things are a bit different at velocities near the velocity of light and for very large masses).

However, we are mostly interested in constant and uniform gravity which is the force that keeps your chair on the floor. For objects near the surface of the earth, gravity causes objects to accelerate downward at a constant rate of 9.81 m/sec^2 , independent of their mass. This means that gravity causes an object's velocity to change by 9.81 m/sec^2 downward for every second the object is under the influence of gravity.

Hooke's Law

This law states that the force on a object displaced from a point of equilibrium is proportional to the displacement and in the direction opposite to the displacement.

Ill Conditioning

In a system of linear equations, this happens when some of the equations are close to being redundant (e.g., the linear equations are almost linearly dependant) and therefore, the matrix of the linear system is close to being non-invertible . If an ill-conditioned linear system is solved using a standard method, there can be very large errors in the answer.

Impulse

A change in momentum that happens over a time interval, causing a finite change in the velocity of the object on which the impulse is applied.

In Kea, this time interval is divided into a number of time step and the change of velocity is computed successively for each of them.

Inertia

The tendency of an object to maintain its state of rest or of uniform motion. The resistance to change in the quantity of linear motion (linear momentum) is given by the mass of the object. Resistance to change in the quantity of angular motion (angular momentum) about an axis is a moment of inertia about that axis.

Inertia Tensor	The set of numbers that describe the resistance to change in angular motion. This consists of several numbers because, in any given orientation, a body may have different inertia to rotation about all three axes, x, y and z. This quantity also changes with the orientation of the body, which is why it is a tensor. The inertia tensor can be represented as a 3x3 matrix which is symmetric and positive definite.
Inertial Reference Frame	A reference system in which the Newton's laws of motion holds. See <i>Newton's Law</i>.
Integrator	This is the term describing the algorithms and resulting code implementation that calculates the position and orientation of physical objects in a MathEngine world. MathEngine provides both an explicit and a semi-implicit integrator. An explicit integrator works by directly adding small quantities to the current state of the system, while an implicit system involves solving for the root or minimal point of an algebraic equation. Explicit integrators are usually faster than implicit ones but are also prone to larger integration errors. The MathEngine semi-implicit integrator combines the qualities of both implicit and explicit integrators by being fast and precise at one and the same time.
Joint	An equality constraint or connection between objects or bodies. When two or more bodies or objects are connected by joints, they form a multibody system which is also called a jointed body or an articulated body.
Kinematics	Kinematics is the study of how things move in relation with time. It describes the different types of motion a body can undergo: translational (change of position), rotational (change of orientation), vibrational (change of shape and size).
Forward Kinematics	Allows you to position a complex jointed object quickly by positioning the first body in a jointed body or an articulated chain of bodies.

Linear Complementarity Problem

This is much like solving a linear $Mz + q = 0$. However, in the LCP, we require the solution vector z to have positive components. For example, to solve for the normal forces at points of contact, in which case the normal force must be positive to push things apart and prevent inter-penetration (a negative normal force corresponds to sticking). Since we restrict z to be positive, we won't be able to solve for the linear problem and there will be some left over, say $w = Mz + q$. We further specify the problem by requiring that w be also positive. For instance in Kea, w corresponds to the velocity in the direction normal to the contact plane at a contact point. Requiring that this be positive means that we don't allow the velocity to be in the direction that violates the non-penetration condition.

The complementarity condition means that for each variable w_i, z_i , either $z_i > 0$ and $w_i = 0$, in which case we say that the constraint is active or tight, or $z_i = 0$ and $w_i > 0$, in which case we say that constraint i is free. In other words, the product $z_i * w_i = 0$ for each i . What this means is that at a contact point, either the normal force is positive and the normal velocity is zero (hard contact), or the normal force is zero and the velocity is positive in which case we have a grazing contact and the objects are coming apart. In a multibody situation where the rigid bodies in contact are also connected together with joints, this problem becomes difficult since we must produce a consistent assignment of touch and graze conditions which might be conflicting with each other.

The LCP is related to several other optimization problems like quadratic programs, bimatrix games and so on. However, contrary to standard optimization problem, LCPs don't have a cost function and the problem is almost purely combinatorial which makes it NP hard.

Local Coordinates

The origin of the local coordinate system of a body is usually located at its center of mass.

Jacobian Matrix

Given any vector function of several variables with components

$$F_1(x_1, x_2, \dots, x_n),$$

$$F_2(x_1, x_2, \dots, x_n),$$

....

$$F_n(x_1, x_2, \dots, x_n),$$

the Jacobian matrix is defined as:

$$J_{ij} = \frac{\partial F_i}{\partial x_j}$$

In the case where the functions F_j represents a coordinate transformation between x_i and $q_j = F_j$, then, the Jacobian matrix the linear approximation of the coordinate transformation in the neighborhood of the point where the derivatives of F_j are evaluated.

In a more general way, the Jacobian matrix is the local linear approximation of a non-linear function in the neighborhood of a point x_0 :

$$F(x) = F(x_0) + J(x-x_0)$$

In Kea, we use the Jacobians of the constraint functions in the computation of the constraint forces i.e., the forces required to maintain the constraint.

Mass

Mass is the measure of the amount of material contained within a body. It is independant of the location of the body.

Mechanics

A branch of physics that deals with the analysis of motion of physical objects. This includes the study of the equations of motion derived in dynamics.

Momentum

This is a quantitative measure of the state of motion of an object. Linear momentum is the product of mass and velocity: $\vec{p} = m\vec{v}$ and is a vector quantity. Angular momentum is a measure of the rotational motion of an object about some point. For a point mass m , this is defined as $\vec{L} = m\vec{r} \times \vec{v}$ or $\vec{L} = I\vec{\omega}$ where is $\vec{r}$ a vector from the point about which we are measuring the rotational movement, $\vec{r} \times \vec{v}$ is the cross product and $I = mr^2$ is the corresponding inertia tensor.

For a rigid body, the spin angular momentum is also defined as $\vec{L} = I\vec{\omega}$ where $\vec{\omega}$ is the angular velocity but the inertia tensor may be more complex than the mass point scalar value.

For a collection of objects, the total momentum is composed of three parts: linear momentum of the center of mass; angular momentum of the center of mass about some reference point; and spin angular momentum (i.e., rotational motion of the object about the center of mass).

Note that you can always find an inertial frame of reference in which both part 1-) and 2-) vanish at some point in time, but it may not be possible to find a frame of reference in which the spin angular momentum vanishes.

Note that Newton's second law is really a relation between the time rate of change of momentum and applied forces. The simplification to $F = ma$ is only good for point masses that have constant mass.

Multistep Integrator

This is an integrator that requires information from several preceding time steps. Multistep integrators assume a degree of smoothness about the solution (i.e., of continuity of its function and derivatives) which is not satisfied in a simulation of rigid bodies with collisions and friction. This is because collision and friction functions naturally tend to have discontinuity (i.e., abrupt change in their values).

Newton

The SI unit of force, derived from 3 basic units: mass, length and time.

Newtons' Laws

Newton's first law of motion: In an inertial frame, an object that is free of interaction has constant momentum. An object at rest remains at rest, and if it is in motion, that motion will be uniform and constant, that is, the center of mass will move on a straight line and the motion about the center of mass will be uniform. The tendency of an object to resist any change in motion is called **inertia** of the object. **Mass** is a measure of inertia.

Newton's second law states the relationship between an acting force, inertia of a body, and its acceleration as follows:

$$\frac{d\vec{p}}{dt} = \vec{F}$$

where p is the momentum of the object and F is the applied force. In words, this means that the force exerted on a body is equal to the rate of change of momentum of that body. For point masses with constant mass, this simplifies to:

$$m \frac{d\vec{v}}{dt} = \vec{F}$$

the familiar $F = ma$. As such, Newton's second law only applies to point masses and further analysis is required to derive the equations of motion for rigid bodies and composite objects.

Newton's third law states that for every action there is an equal and opposite reaction. If two bodies interact, the magnitude of the force exerted by body 1 on body 2 is equal to the magnitude of the force exerted on body 1 by body 2. These two forces are also opposite in direction.

Normals	Normals are unit vectors. The normal to a plane is a unit vector perpendicular to the plane. In 3D graphics, they indicate the visible side of an object.
Object	We use this term to describe virtually any entity that can be move—a point mass, rigid body, articulated body, multi-body system, and so on. We usually use it as a general term when what is being discussed is not restricted to rigid bodies. Object does not refer to object-oriented programming constructs.
Ordinary Differential Equation	A differential equation is a relationship between the rate of change for the variables of a system and the state of that system. The term <i>ordinary</i> means that the derivatives are taken with respect to only one variable, by contrast with <i>partial differential equations</i> .
Orientation	Describes a body's direction relative to the world or reference frame coordinates.
Power	The time rate at which work is performed (energy is transformed) by a system.
Quaternion	Quaternions form a four-dimensional algebra which is an extension to normal complex numbers. They can be used to represent rotations of coordinate axes in 3D worlds in a way that is free of singularity, in contrast to Euler angles, for example. With Euler angles there are difficulties when the pitch angle reaches 90 degrees - a problem that never occurs when quaternions are used. The four components of a unit quaternion which represents a 3D rotation can be thought of as a set of parameters for 3D rotations, as an alternative to Euler angles. Because they lack the singularities that Euler systems have, quaternions allow for smooth interpolations between rotations.
Restitution (coefficient of)	When designing your game, think of restitution as bounciness. In physics terms it is the ratio of the relative speeds of two bodies just after and just before a collision between them along a mutual straight line; it ranges from 0, for perfectly inelastic collisions (no bounce), to 1 (maximum bounce), for completely elastic ones. If you increase the value to a number greater than one, you'll get an effect like a pinball machine. It is a constant at moderate speeds.
Right—Hand Rule	A useful visual aid to represent a right-handed coordinate systems. Here's how it works: orient your right hand so that your thumb is perpendicular to the plane defined by the x- and y- axes, then curl your

fingers. If you can curl your fingers in the direction from the x axis towards the y axis, then your thumb will be pointing in the direction of the z axis.

Rigid Body	A solid object that doesn't change its shape or size, despite any forces being applied to it.
Rotation	In a 3D world, rotation of an object is its motion about an internal axis of symmetry relative to an internal coordinate system.
SI Units and Prefixes	SI stands for <i>Systeme International</i> . It is a standardized set of units for scientific measurement. Examples of SI units include: the Newton, metre, Hertz, second and Watt. Examples of SI prefixes include: mega, milli, tera and giga.
Single-Step Integrator	Unlike multistep integrators, a single-step integrator uses information only from the current time step (n, n+1) to compute the next position from the derivatives. This is the case in Runge Kutta integrators, for example. Single-step integrators can be explicit or implicit. They can also be multistage, in which case they need several derivative evaluations at different intermediate times spread over the time step interval to compute the position at the next time step.
Stiff Springs	Stiffness is a measure of how a spring resists stretching. A very stiff spring will hardly move at all—even if you pull very hard, whereas a spring with very little stiffness will stretch easily even if you pull it gently.

The stiffness is related to the 'spring constant' usually represented by the letter *K*. Writing this down as a formula called Hooke's law for the extension of springs we get;

$$\text{Force} = - (\text{spring constant}) * (\text{length increase})$$

Stiff springs create very large forces and this can be a problem for the integrator. The reason is that the force is never very large for very long. For a small mass attached on a stiff spring, the resulting motion for the mass is to stay very near the point of equilibrium. The force from the spring will change very rapidly from large to small to keep the mass in place. An integrator that is using a time step larger than half the period of the spring might overestimate the magnitude of the force and this can lead to an "explosion" i.e., a situation where the mass moves further and further away from the equilibrium point.

Note also that the frequency of a spring is dependent on the mass attached on it and if you drop a small mass on a stiff spring, that will

produce a very high frequency which might be difficult for the integrator.

The constraint solver uses a first order integrator for the forces which doesn't check if forces are getting very large. This means that you should avoid stiff springs altogether. If you want to simulate a stiff spring, you should use a "relaxed" constraint instead i.e., a constraint that is allowed to be violated.

Time-Step

Time, like mass, is another of those rather undefined measures in physics. Everybody knows what it is, and agrees to measure it in seconds.

A time step is an interval of time, typically of the order of the frame rate. If you run a simulation at 60 fps, your time step should be $1/60$ s or shorter; if it is shorter, you will need to perform several steps between frames if you want to create a real-time simulation in which everything seems to move at the same rate as in the real world.

The simulation computes what the state of the system will be after an interval of time equal to the time step has elapsed, using current state and derivative information. Essentially, the simulator uses current velocities and forces and extrapolates them to compute an approximation of the state into the future. The correctness of the approximation is related to the time step: the bigger the time step the worse the approximation.

Tracking Problem

The problem of keeping track of the distance between objects as simulated time is stepped forward. This is useful to estimate when objects will collide.

Torque

A torque can be described as a turning or twisting force. It's the measure of a force's tendency to produce a rotation about an axis, which also produces a change in the angular momentum of a body. For an extended body—one that is not just a single point mass - a torque is generated by applying a force at a point other than at the center of mass. The distance between the point where the force is applied and the center of mass acts like a lever: the greater the distance, the bigger the resulting change in angular momentum.

Velocity

The rate of change of position along a straight line with respect to time. A velocity has speed and direction; it is a vector quantity whose magnitude is expressed in units of distance over time, such as kilometers per hour.

Work

Work is the product of force, and the distance moved by the point of application of the force in the direction that the force was applied. Work is directly related to the change in the energy of the system.

World Reference Frame	A coordinate system that is used to locate an object in relation to a world origin. This is sometimes referred to as a Global Coordinate system.
XYZ axes	In a Cartesian coordinate system, these are the three orthogonal axes which represent three-dimensional space.

- A
 - acceleration
 - glossary entry 60
 - actuator
 - glossary entry 60
 - advanced features 35
 - angular velocity
 - glossary entry 60
- B
 - BallHitsWall1.c sample program 23
 - BallHitsWall2.c sample program 31
 - BallHitsWall3.c sample program 32
 - ballistic objects
 - glossary entry 60
 - ballistics
 - glossary entry 60
 - box 52
 - Bridge
 - and the static environment 46
 - Dynamics Toolkit 15
 - BSP tree
 - glossary entry 60
- C
 - cartesian coordinates
 - glossary entry 60
 - center of mass
 - glossary entry 60
 - code size 9
 - collision
 - detection 5
 - response 5
 - collision detection
 - glossary entry 60
 - collision model
 - glossary entry 61
 - collision models
 - and the geometry library 36
 - defining 21
 - collision response
 - glossary entry 61
 - using contacts to perform 11
 - collision space
 - freezing models in 37
 - collision spaces
 - creating 21
 - Collision Toolkit
 - advanced features 35
 - and your application 3
 - getting started 19
 - installing 17
 - overview 2
 - writing simple programs 20
 - collisions
 - testing 21
 - composite geometries 44
 - cone 53
 - configuration control 9
 - constraint
 - glossary entry 61
 - constraint equations
 - glossary entry 61
 - contact
 - definition of 10
 - contact generation 10
 - contacts
 - using 11
 - conventions xii
 - naming x
 - type ix
 - typographical ix
 - convex object
 - glossary entry 61
 - ConvexMesh geometry type 55
 - coordinate sorting 37

- coordinate systems 42
- culling
 - glossary entry 61
 - management 37
- culling optimizations 37
- cycle
 - glossary entry 61
- cylinder 53
- D
- damping
 - glossary entry 62
- detection
 - collision 5
- differential index
 - glossary entry 62
- documentation
 - related viii
- dynamic simulation
 - glossary entry 62
- dynamics
 - glossary entry 62
- Dynamics Toolkit
 - Bridge 15
 - integrating with 5, 39
 - supporting 31
- E
- event handling 12
- F
- far field functionality 13
- far field view 4
- features
 - advanced 35
- force
 - glossary entry 62
- force-limited actuator
 - glossary entry 62
- forward kinematics
 - glossary entry 64
- frame
 - glossary entry 62
- frame rate
 - glossary entry 62
- freeze feature 37
- friction
 - glossary entry 62
- functions
 - intersection 57
- G
- geometrical primitives 51
- geometrical types 8
 - and interactions 49
 - non-primitive 8, 55
 - registering and intersection 54
- geometry
 - composite 44
 - implied 42
- geometry library 36
- geometry type
 - ConvexMesh 55
 - RGHeightField 55
- geometry types
 - loading primitive 20
- Gilbert-Johnson-Keerthi algorithm
 - glossary entry 63
- GJK engine
 - glossary entry 63
- glossary
 - acceleration 60
 - actuator 60
 - angular velocity 60
 - ballistic objects 60
 - ballistics 60
 - BSP tree 60
 - cartesian coordinates 60

- center of mass 60
- collision model 61
- collision response 61
- collision detection 60
- constraint 61
- constraint equations 61
- convex object 61
- culling 61
- cycle 61
- damping 62
- differential index 62
- dynamic simulation 62
- dynamics 62
- force 62
- force-limited actuator 62
- forward kinematics 64
- frame 62
- frame rate 62
- friction 62
- Gilbert-Johnson-Keerthi algorithm 63
- GJK engine 63
- gravity 63
- Hooke's law 63
- ill conditioning 63
- impulse 63
- inertia 63
- inertia tensor 64
- inertial reference frame 64
- integrator 64
- Jacobian matrix 65
- joint 64
- kinematics 64
- linear complementarity problem 65
- local coordinates 65
- mass 66
- mechanics 66
- momentum 66
- multistep integrator 67
- Newton 67
- Newton's laws 67
- normals 68
- object 68
- ordinary differential equation 68
- orientation 68
- power 68
- quaternion 68
- restitution, coefficient of 68
- right-hand rule 68
- rigid body 69
- rotation 69
- SI units and prefixes 69
- single-step integrator 69
- stiff springs 69
- time-step 70
- torque 70
- tracking problem 70
- velocity 70
- work 70
- world reference frame 71
- XYZ axes 71
- gravity
 - glossary entry 63
- H
 - header files
 - including 20
 - Hooke's law
 - glossary entry 63
- I
 - ill conditioning
 - glossary entry 63
 - implied geometry 42
 - impulse
 - glossary entry 63
 - inertia

- glossary entry 63
- inertia matrix 43
- inertia tensor
 - glossary entry 64
- inertial reference frame
 - glossary entry 64
- installation 17
- integrator
 - glossary entry 64
- interactions 8
 - and geometrical types 49
- intersection functions 57
- J
- Jacobian matrix
 - glossary entry 65
- joint
 - glossary entry 64
- K
- kinematics
 - glossary entry 64
- L
- libraries
 - linking required 20
- linear complementarity problem
 - glossary entry 65
- linking
 - required libraries 20
- local coordinates
 - glossary entry 65
- M
- mass
 - glossary entry 66
- material properties 45
- MathEngine
 - contacting xii
- matrix
 - inertia 43

- McdModelSetTransformPtr, assigned to McModel 36
- mechanics
 - glossary entry 66
- momentum
 - glossary entry 66
- multistep integrator
 - glossary entry 67
- N
- near field
 - view 4
- Newton
 - glossary entry 67
- Newtons' laws
 - glossary entry 67
- normals
 - glossary entry 68
- O
- object
 - glossary entry 68
- optimizations 36, 47
- ordinary differential equation
 - glossary entry 68
- orientation
 - glossary entry 68
- P
- pairwise proximity culling 13
- particle system 57
- physical simulation 10
- plane 52
- power
 - glossary entry 68
- primitive types
 - box 52
 - cone 53
 - cylinder 53
 - plane 52

- sphere 52
- primitives
 - geometrical 51
- programs
 - sample 23, 31, 32
- properties
 - material 45
- Q
- quaternion
 - glossary entry 68
- R
- rays 12
- relative transforms 41
- RenderWare models 16
- required libraries
 - linking 20
- response
 - collision 5
- restitution, coefficient of
 - glossary entry 68
- RGHeightField geometry type 55
- right-hand rule
 - glossary entry 68
- rigid bodies
 - inertia matrix 43
- rigid body
 - glossary entry 69
- rotation
 - glossary entry 69
- S
- sample programs
 - BallHitsWall1.c 23
 - BallHitsWall2.c 31
 - BallHitsWall3.c 32
- sensors 12
- SI units and prefixes
 - glossary entry 69

- simulation support layer 14
- single-step integrator
 - glossary entry 69
- software
 - related xi
- sorting
 - options 37
- spatial-global queries 13
- sphere 52
- static environment
 - interactions with 46
- stiff springs
 - glossary entry 69
- support layer
 - simulation 14
- synchronization
 - with graphics 36
- system
 - particle 57
- systems
 - coordinate 42
- T
- testing
 - for collisions 21
- third-party tools 6
- time-step
 - glossary entry 70
- tools
 - third-party 6
- torque
 - glossary entry 70
- tracking problem
 - glossary entry 70
- transform
 - collision model's 36
- transforms
 - relative 41

- types
 - geometrical 8
- U
- units ix
- V
- velocity
 - glossary entry 70
- views
 - far field 4
 - near field 4
- W
- work
 - glossary entry 70
- world reference frame
 - glossary entry 71
- X
- XYZ axes
 - glossary entry 71