Physics Toolkits for VR and Real-Time Graphics

Kenneth Holmlund

HPC2N

Why a Physics Toolkit?

The same advantages as always with modular software.

- Reusability
- Evolving
- Generality
- Scalability
- Portability
- Implementation (mature API's)
- Documentation
- Quality
- Stability
- Precision

Most programmers, designers and developers are not very skilled in computational physics or numerical methods!

Not too cleaver to re-invent the wheel...

Havok

www.havok.com

Games SDK 1.3

MAX Havok plug-in

- Win32 (and Xbox), PS2, MacOS
- Rigid-body dynamics, soft-objects, particles, fluids & flotation, fracture, collisions, networking
- Rather high abstraction level API, C/C++
- Just aquired another SDK/company, ”Ipion”.

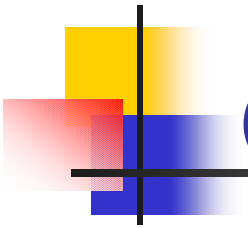
MathEngine

www.mathengine.com

Collision Toolkit 1.0 (alpha 0.0.5)

Dynamics Toolkit 2.0 (alpha 0.0.5)

- Irix, Linux, Win32, PS2
- Semi-implicit rigid-body dynamics for explicit forces, fields, friction, contacts and constraints
- Rather low-level API, mainly for C
- Bridge between Dynamics and Collisions
- Many other features planned
- Plug-in for Softimage



Other SDK's

Research prototypes

Inside game engines

Inside VR engines

3D-modelling software plug-ins (Maya, MAX, ...)

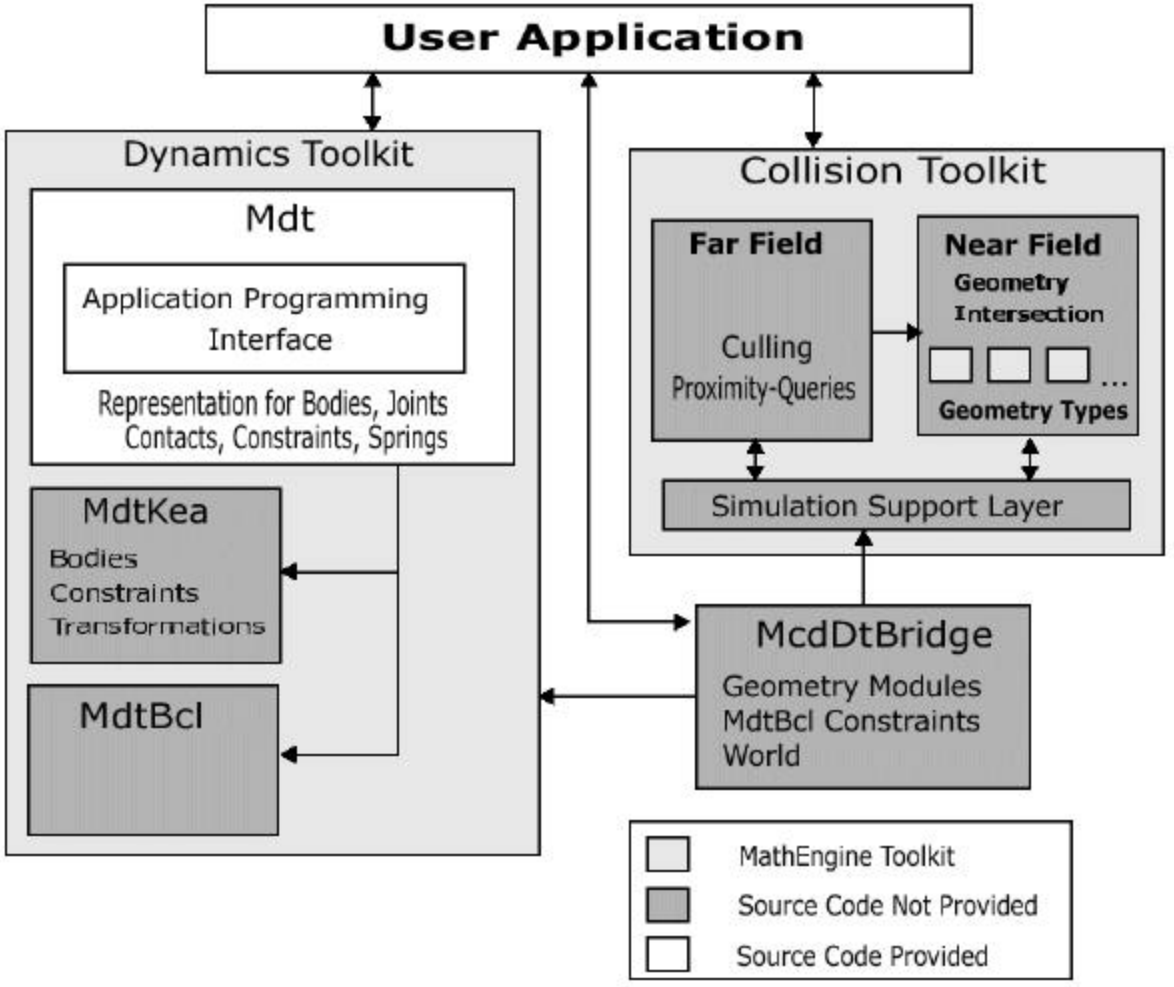
CAD plug-ins

Proprietary in-house products

Modelica, MatLab/FemLab, FEM, etc. (not really real-time, or even interactive...)

No Open Source/GPL project yet!

MathEngine SDK



MathEngine Toolkits: High-Level Architecture

MEDT Modules

Mdt

Implements high-level abstractions: rigid bodies, forces, joint constraints, contact constraints, ”the world”.

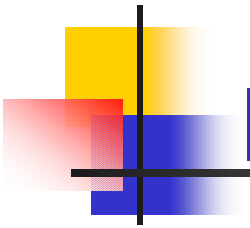
MdtBcl

Constructs the Jacobian constraint matrices that are passed to MdtKea.

MdtKea

The heart. Semi-implicit integrator for the differential algebraic equations of rigid multibody systems. Optimized for speed ”without” loss of precision and stability.
Takes the Jacobian, list of bodies, their transformation matrices, and some additional parameters (e.g. time step).

MdtKea and MdtBcl can be used explicitly without Mdt, but we won’t go into that here.



ME Viewer

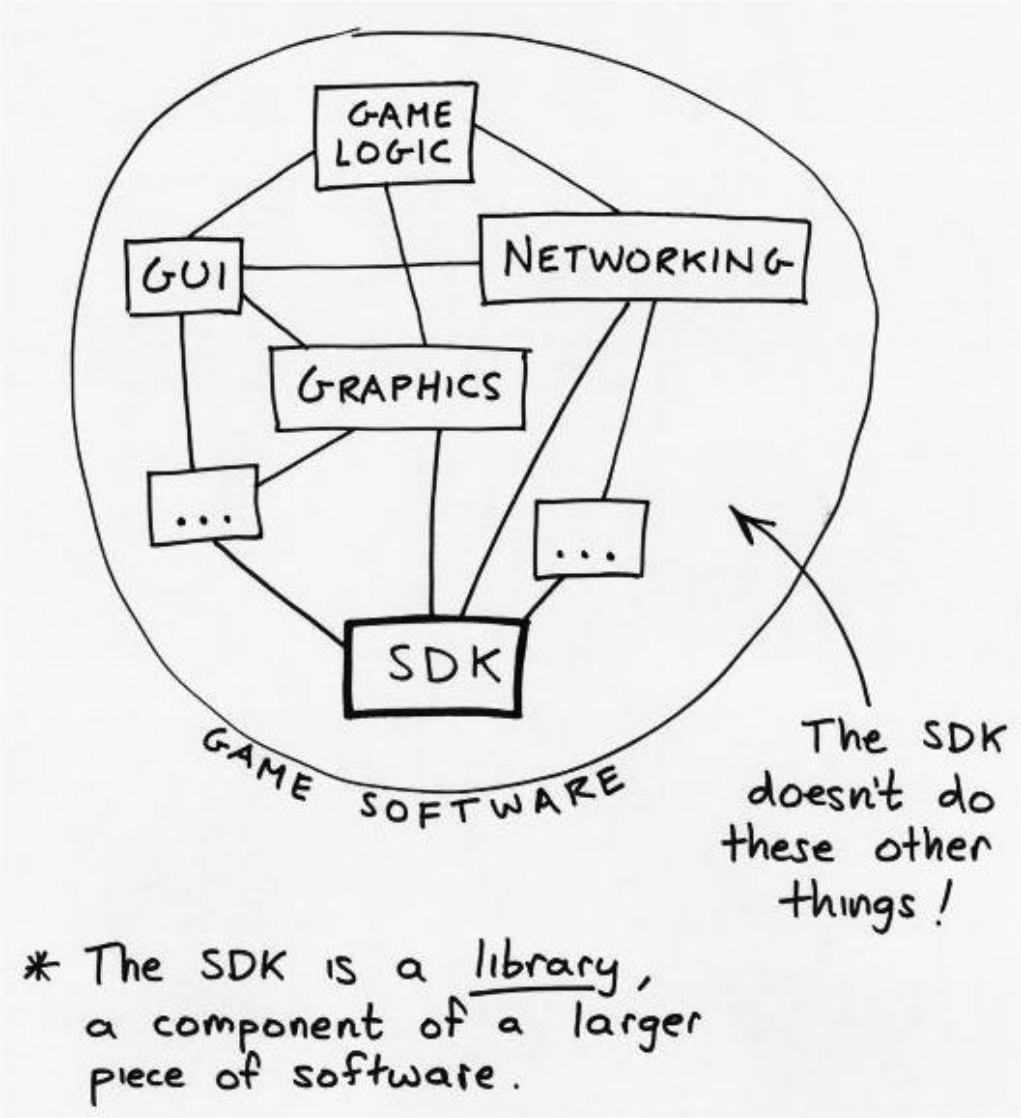
Simple viewer for prototyping and demonstrations.

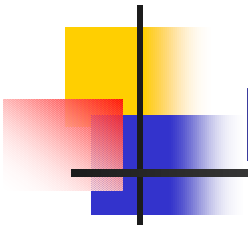
Win32: OpenGL, Direct3D
Linux, Irix: OpenGL
PS2: Renderware

Pan, rotate, zoom, lightning, shading, stop and restart, keyboard and mouse events, ...

Will be used in my examples, but can easily be replaced with e.g. WorldToolkit or other rendering engines.

ME SDK in an Application





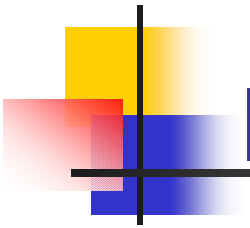
ME Implementation Issues

By tradition the application is driven by the rendering engine.

Typically the SDK is called from a registered callback function, e.g. `Tick()`.

`Tick()` updates the simulation by one (or several) timesteps, and may also do other things that should be done during that tick:

`Tick()`could also add an explicit force/torque or arrange for human-in-the-loop, e.g. add user related forces.



ME Documentation

Dynamics Toolkit Developers Guide

Collision Toolkit Developers Guide

Viewer Developers Guide

MathEngine Overview

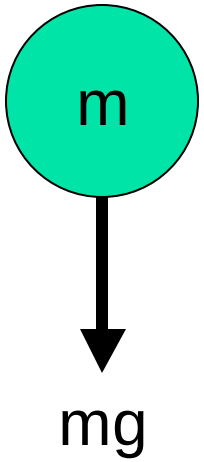
Reference manuals for MECT, MEV, MEDT.

Source code for MDt

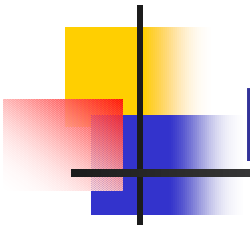
Examples and tutorials.

ME Getting Started - Example

Simulate a ball that falls through space in a gravity field.



Pseudo C code example



Manage and allocate memory

Include Mdt

```
#include ”Mdt.h”
```

Managing and allocating memory for MDt:

```
extern struct MeMemoryAPI MeMemoryAPIMalloc;
struct MeMemoryOptions opts;
```

This is black magic:

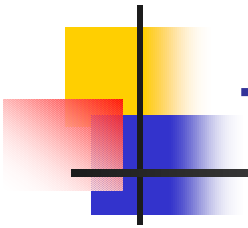
```
void *memory;
memory = malloc(10000);
```

Initialize the world

```
MdtWorldID world;
world = MdtWorldCreate(1, 0, memory, 10000);
```

The four parameters are:

- The maximum number of bodies in the world.
- The maximum number of constraints in the world.
- The block of memory to be used by MdtKea
- The size of that block of memory.



The memory management API

The memory buffer is used for temporary storage in MdtKea.

MdtKea is highly optimized for Win32 and PS2 memory architecture.

It's a matter of allocating *enough* memory.

Get **maximum** memory requirements from MdtKea:

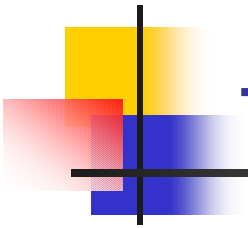
```
int MdtKeaMemoryRequired ( int * rows_in_partition,
                           int   num_partitions );
```

In practice, this maximum is never needed – in fact it is **a waste** of memory! Instead:

```
int MdtWorldGetMemoryPoolSize ( MdtWorldID w );
```

Returns max memory used so far in the simulation.

Also: change memory buffer, change size of it, return the adress of the buffer, register an overflow callback, get a pointer to the callback pointer.



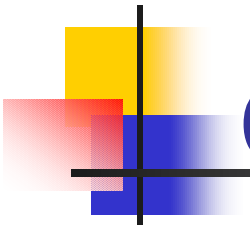
Timestep

```
MeReal step = (MeReal)(0.03);
```

Step is chosen so that the simulation is fast enough (evolution of simulation time is interactive or real-time), but still realistic and stable.

A too large timestep may for example cause a ball to fall straight through a table since it passes the entire table in a single step!

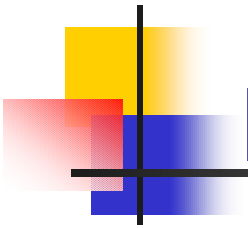
Using too small timesteps one may loose numerical precision (small numbers added to large numbers). However, this is very rarely a problem as such small timesteps often lead to extremely slow evolution of simulation time.



Gravity

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

We define gravity to act in the negative y direction.



Define the ball

Define a body:

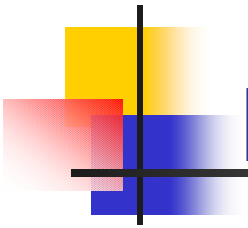
```
MdtBodyID body;

body = MdtBodyCreate(world);
```

The memory for this body came out of the memory allocated by the Memory Manager earlier.

The body has defaults set for mass (1), moment of inertia (identity matrix), and some other properties.

Enable it:

MdtBodyEnable (body);

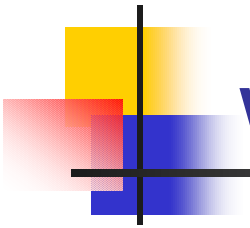


Reset body parameters

Set up a reset function:

```
void reset(void)
{
      MdtBodySetPosition ( body, 0, 10, 0 );
      MdtBodySetLinearVelocity ( body, 0, 0, 0 );
      MdtBodySetAngularVelocity ( body, 0, 0, 0 );
      MdtBodySetQuaternion ( body, 1, 0, 0, 0 );
}
```

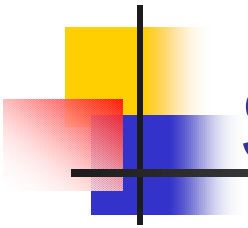
This function is called before we start the simulation and can also be called during the simulation to reset it.



What about body geometry?

The ball doesn’t have a geometry!

This is an approximation that is exact in this simple example, since no other forces than gravity acts on the body, and it can’t collide with something.



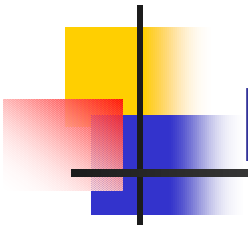
Stepping through time

Set up a tick callback function:

```
void Tick ( RRender * rc)
{
      MdtWorldStep ( world, step);
}
```

In our main program we may e.g. start the renderer and register the Tick function as a callback:

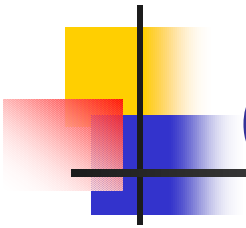
```
RRun(rc, Tick);
```



Pseudo renderer...

Pseudo-code for the MathEngine viewer's RRun:

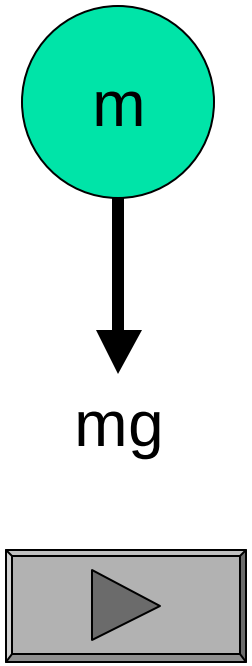
```
while no exit-request
{
  Handle user input;
  call Tick() to evolve the simulation
  and update graphics transforms;
  Get the body transformations e.g.
  sphereG->m_matrix = MdtBodyGetTransformPtr(body);
  Draw graphics;
}
```



Cleaning up

```
voind cleanup ( void )
{
      MdtWorldDestroy(world)
      free ( memory );
      RDeleteRenderContext ( rc );
}
```

Try it...



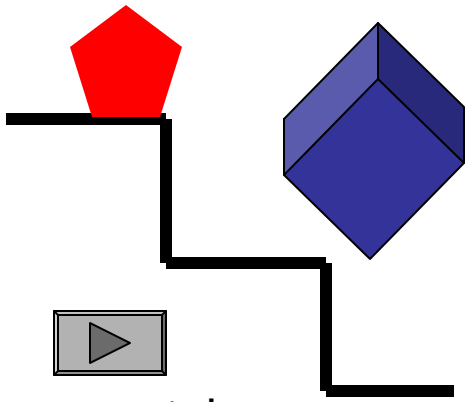
Contacts and Collisions

When and why?

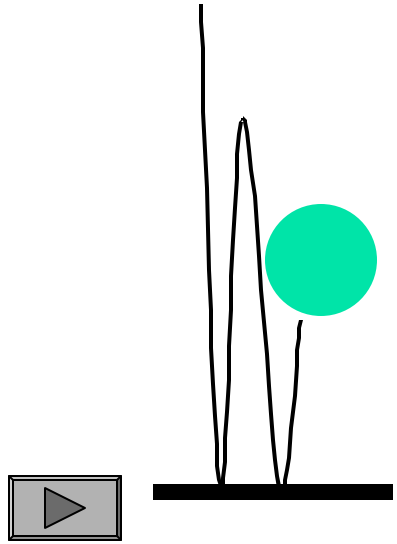
For very simple simulations, e.g., sphere-sphere, sphere-plane, box-plane, collisions can be handled with a few lines of code.

For more complicated cases we use a collision engine (e.g. MECT).

Examples:



convexstairs.c



bounce.c

MathEngine Collision Toolkit

If using MECT, Mdt can also handle MdtContact structures for MECT.

MECT detects collisions in two modes:

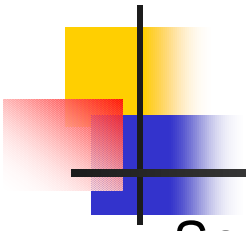
Farfield module:

Monitors the separation of object pairs

Nearfield module:

Geometrical types for collision models
Intersection tests between each pair

MECT can also be used with other dynamics engines that are triggered through events.



MECT: Can also do...

Sensors, triggers and event handling

- Inside a given room
- Inside a ”danger zone”
- Within hearing distance

Rays (do not report intersection data, but instead a special ray data structure that describes the surface hit by the ray)

- See
- Occlude
- Shoot

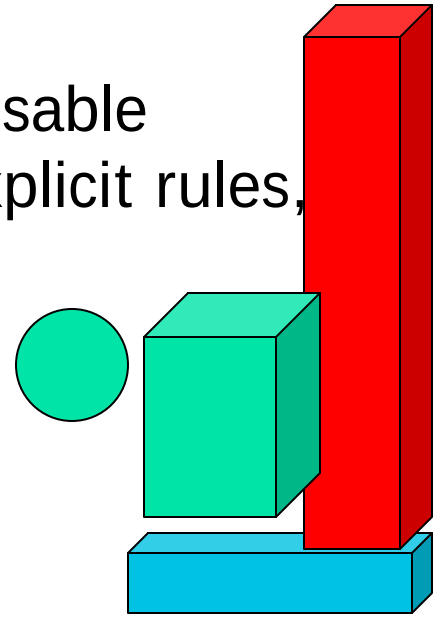
MECT: Enabling, disabling

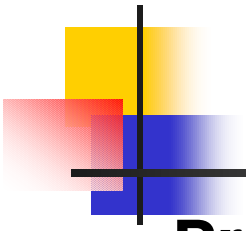
Collision detection is extremely important for enabling and disabling the rigid bodies.

In a world consisting of 10.000+ bodies is *important* that the dynamic handles only the active bodies.

In ME, treshold parameters are used to disable objects while the collision detection, or explicit rules, are used to enable a body.

topple.c





MECT: Geometrical types

Primitives:

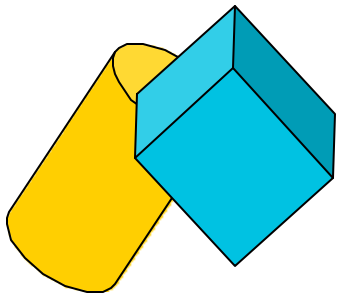
- Sphere
- Box
- Plane
- Cylinder
- Cone

Non-primitives:

- Convex mesh
- Height Field
- Particle System

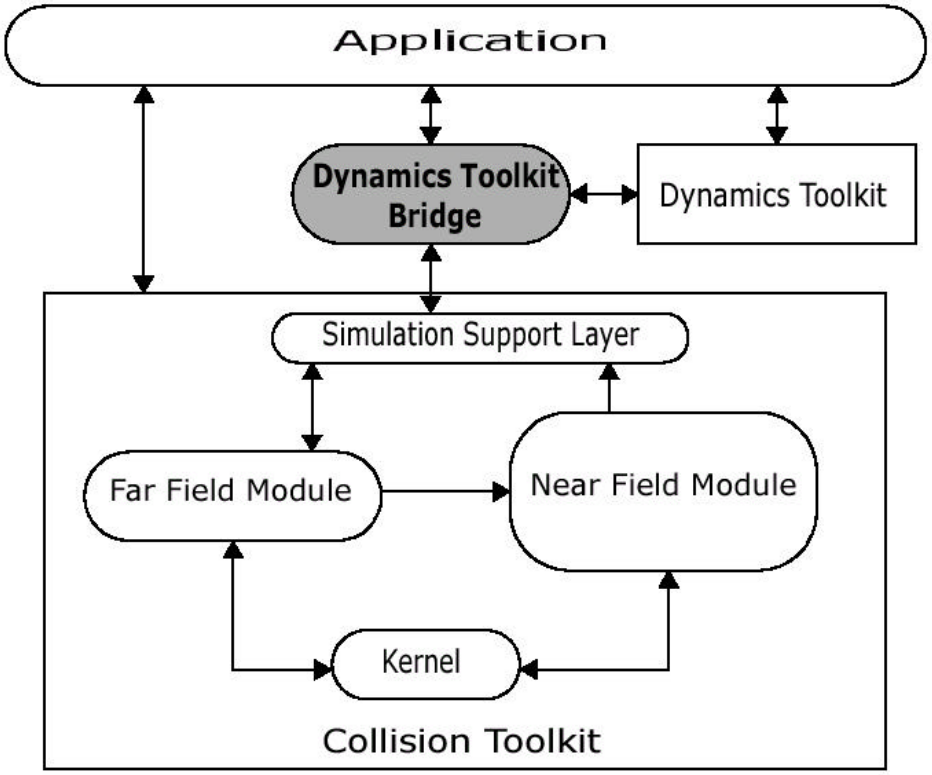
What is a contact?

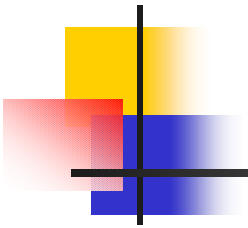
- When two objects have a point in common
- The location of the contact
- The normal
- An estimate of the penetration depth



MECT bridged to MEDT

The bridge manages memory, keeps track of collision events and prepares contact data which it sends to MEDT.





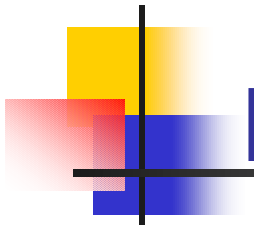
Initialize the CT and the bridge

Initiate and load *all* primitives and *all* models:

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

Initialize the bridge and compile the collision space:

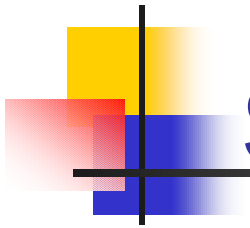
```
McdDtBridgeInit();
McdDtBridgeID bridge = McdDtBridgeCreate();

space = McdSpaceAxisSortCreate(McdAllAxes,
                  MAX_BODIES, 2 * MAX_BODIES);
McdPairHandlerRegisterSpace(space);
cdHandler = McdDtBridgeCreate();
```



Register all objects and models

Specify the collision model for each shape of the MdtBody, e.g.

```
MdtBodyID ball = MdtBodyCreate();
MdtBodyEnable(ball);

McdModel ballCM;

ballCM = McdModelCreate(McdSphereCreate(radius));
McdDtBridgeSetBody(cdHandler, ballCM, ball);
```



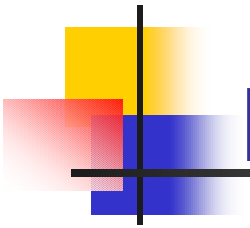
Set contact/collision parameters

```
material0 = McdDtBridgeGetDefaultMaterialID();
params = McdDtBridgeGetContactParams(material0,
                                     material0);

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

We may of course define explicit materials for each type of object if we want to.

Create the collision space:

```
McdSpaceBuild(space);
```

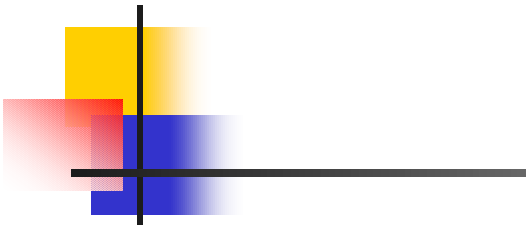


Dynamics of a Collision

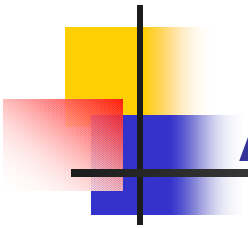
In the tick function:

```
McdPairHandlerUpdate();
```

Registered spaces are updated, and overlap status of pairs in all spaces is sent to various response modules.



More about dynamics



Adding forces and other

An explicit force can be accumulated to a body and is used during MdtWorldStep().

Note: To apply a constant force to a body, you must add it on each timestep!

```
MdtAddForce()

MdtAddForceAtPosition()

MdtAddTorque()

MdtAddImpulse()

MdtAddImpulseAtPosition()
```

…

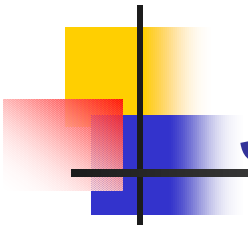
Friction

There are three main friction modes:

```
typedef enum
{
MdtContactTypeFrictionZero, //frictionless contact
MdtContactTypeFriction1D, //friction only along
                            primary direction
MdtContactTypeFriction2D, // friction in both
                             directions
MdtContactTypeUnknown = -1 //invalid contact type
}
```

Setting the parameters:

`MdtContactParamsSetFriction()`
`MdtContactParamsSetMaxAdhesiveForce()`



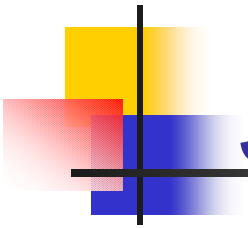
Joints and Constraints

Each rigid body has 6 degrees of freedom (DOF)

Joints and constraints reduce the degrees of freedom.

Joints and constraints are ”stiff”, and therefore cannot be efficiently treated using explicit integration. One integrates them in an implicit space.

The differential-algebraic and numerical details are beyond the scope of this course.



Joints & Constraints in MEDT

Ball-and-socket: MdtBSJoint

Hinge: MdtHinge

Prismatic (piston): MdtPrismatic

Fixed-Path: MdtFixedPath

Fixed-Path-Fixed-Orientation: MdtFPFOJoint

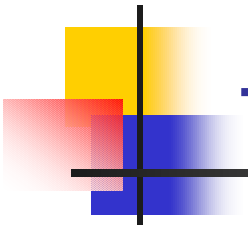
Universal: MdtUniversal

Linear1: MdtLinear1

Linear2: MdtLinear2

Car Wheel: MdtBclCarWheel (structure)

These can be created, enabled, disabled, limited and actuated (driven). The limits may be stiff or soft.

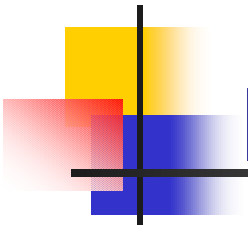


Tricks of the trade

In large simulations, make sure that inactive object are disabled.

When adding forces from user interaction, make sure the magnitudes are limited. Suggestion: connect the user and the simulated objects via a spring or a joint (example: Havok's Soft Ball)

If the physical simulation doesn't behave properly, energy can be monitored for debugging.



Networked Physics

What should be sent to optimize synchronization of simulations?

- State information
- Problem dependent

What if we loose network data (e.g. using UDP)?

- *Dead reckoning algorithms*
  - *Prediction*: Intelligent interpolation forward in time
  - *Convergence*: Adjut the predicted state to the updated state
  - Always a competition between complexity/cpu-time and real-time aspects.