VERSION 1.0

Here is my very first tutorial that I have ever written and I hope you find it useful. If there's any mistake (and there should be heaps ☺) please don't hesitate to tell me at

Mdobele@primedgames.com

With the subject TUTORIAL

This tutorial is aimed at helping people get ODE into a very small OGRE application.

Also you can talk directly to me at my forum at

www.PrimedGames.com/Forum

Special Thanks to…

Kojack for the Renderable Line classes and his constant help at 3am on MSN ☺

Installation

Extract the code into a folder under this path within your copy of OGRE

e.g **Ogrenew \ Samples \ Tute1 ** *sln file must reside here*

Make sure this is correct otherwise it will stuff up the path settings for all the dependencies.

Also you may have to copy the ode.dll files directly into your ogre's bin directories in order to run it.

These have been included in the folded Needed Dlls. These are placed in the

C:\ogrenew\Samples\Common\bin\Release
C:\ogrenew\Samples\Common\bin\Debug

Now just open the project, build and away you go.

Part 1
Understanding ODE

ODE is a free open source physics SDK that is obtained from the following website.
http://ode.org/

In their words

ODE is an open source, high performance library for simulating rigid body dynamics. It is fully featured, stable, mature and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools.

In My words

When trying to understand how ODE works just think of it as Physics Simulation SDK that runs invisibly and constantly behind your application. You create an invisible physics object to match a visible object on screen. ODE then works out all of the maths behind affecting the invisible object when it bounces around your world when it is affected by things like gravity and friction, and all you have to do is each update tell your visible object to be in the same position and orientation as its invisible ode counterpart. Simple hey!

- Download the source code for this tutorial. It was created in Visual Studio .NET2003 and as that's is the only environment I have that's all I am going to support sorry.

- Including ODE

Hopefully you have already successfully compiled ODE. I generally just download the pre-compiled binaries as I don't tend to change any of their source code anyway.

You can download the precompiled ode.0.5 windows lib's from the ODE website. They are already included in my Source Code anyway.

Using ODE is very similar to using OGRE in the fact that you simply only have to include one header file

```
#include <ode/ode.h>
```

in order to gain access to all of ODE's functionality.

Now ODE and OGRE tend to have some memory allocation fights with each other so If you get a whole bunch of Malloc.h errors simply make sure that Ode.h is included after OGRE.h and wrap the include in this

```
#include "OgreNoMemoryMacros.h"
#include "ode/ode.h"
#include "OgreMemoryMacros.h"
```

Hope fully that will fix any problems.

- **INIT**

Create an ode physics world

This is all found in the `Physics::CreatePhysicsWorld()`

All the objects in a world exist at the same point in time, thus one reason to use separate worlds is to simulate systems at different rates.

Most applications will only need one world.

```
dWorldID m_world;

m_worldID = dWorldCreate();
```

Setting a global gravity for your world. This will affect all movable ODE objects.

```
dWorldSetGravity (m_worldID, 0.0f, -9.8f , 0.0f);
```

Creating our collision Space

Whenever you create an ODE physics object you add it into your collision space. You can create multiple collision spaces if you wish so that only objects contained within the same collision space will collide against each other.

Another benefit of using Collision Spaces is when we perform our collision checking we are able to use ODE `dSpaceCollide()` which will check to see which ODE geometries

may potentially collide this update and then only performs the collision checks on those geoms.

Creating a collision space.

```
dSpaceID m_CollisionSpace;

m_CollisionSpace = dHashSpaceCreate(m_CollisionSpace);
```

Creating A Physics Object (A box to start with)

This is all found within the Physics::CreateOdeBox()

Create an ODE body. Think of it as being the same thing as an OGRE SceneNode, you pass in the worldId that you want this OdeBody to be created in.

```
m_OdeBody = dBodyCreate(m_worldID);
```

Create an ODE collision Geometry. This is the shape that the collision for this object will mimic. So here we are creating a BOX that will act and collide as a BOX would in real life. You pass in your ODE collision space and the size you want the box to be.

```
m_OdeGeom = dCreateBox( m_CollisionSpace , X, Y, Z);
```

Now you have to Link the collision geometry to the ode body. Same as you would do for an ogre entity and an ogre scene node.

```
dGeomSetBody(m_OdeGeom , m_OdeBody);
```

At this stage its always a good idea to set your ODE body to be at the same position as the OGRE SceneNode that you want it to effect. Otherwise your collisions will be offset if both the ode and ogre counterpart are not sitting on 0,0,0

```
dBodySetPosition(m_OdeBody,  pSceneNode->getPosition().x,
            pSceneNode->getPosition().y, pSceneNode->getPosition().z );
```

Now I simply add my two objects into a MAP. This is so later on during the update phase we can get the position of our invisible ODE object and set the position of our visible OGRE object to the same position / orientation.

```
m_objects.insert(std::pair<dBodyID,Ogre::SceneNode
            *>(m_OdeBody,pSceneNode));
```

- **UPDATE**

Now that we have a physics world we have to start updating it.

This is all found in Physics::Update(deltaT);

The first thing we call in our update loop is

```
dSpaceCollide(m_CollisionSpace,0, CollisionCallback );
```

This determines which pairs of geom in a space may potentially intersect, and calls the callback function with each candidate pair.

CollisionCallback is simply a global function that can be found at the top of the Physics.cpp file. As you can see it simply returns back into our own Physics::Collision() function.

Physics::Collision explained.

After finding two objects that will collide with each other its time to tell ODE what it should do to those objects.

The first thing we check is to see if any objects are connected by a joint. If they were then simply exit as we want these objects to be colliding and don't want to do anything else.

If we pass that step then its time to tell ODE what values to set for each object that collides. These kinds of values are things like

```
// The amount of "Bounce" that occurs when objects collide

contact[i].surface.bounce = (dReal)0.1;

// The amount that objects can "Sink" into each other.

contact[i].surface.soft_cfm = (dReal)0.01;
```