Open Dynamics Engine Function Overview

CSC200
Spring 2005
Marc Schraffenberger
The ODE library can be built to use either single or double precision floating point numbers. Single precision is faster and uses less memory, but the simulation will have more numerical error that can result in visible problems.

You will get less accuracy and stability with single precision.

The floating point data type is dReal. Other commonly used types are dVector3, dVector4, dMatrix3, dMatrix4.
Objects and IDs

There are various kinds of objects that can be created:

- dWorld - a dynamics world.
- dSpace - a collision space.
- dBody - a rigid body.
- dGeom - geometry (for collision).
- dJoint - a joint
- dJointGroup - a group of joints.

Functions that deal with these objects take and return object IDs.

The object ID types are dWorldID, dBodyID, etc.
The world object is a container for rigid bodies and joints. Objects in different worlds cannot interact, for example rigid bodies from two different worlds cannot collide.

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.
World Functions

```c
dWorldID dWorldCreate();
```
Create a new, empty world and return its ID number.

```c
void dWorldDestroy (dWorldID);
```
Destroy a world and everything in it. This includes all bodies, and all joints that are not part of a joint group. Joints that are part of a joint group will be deactivated, and can be destroyed by calling, for example, `dJointGroupEmpty`. 
World Functions

void dWorldSetGravity (dWorldID, dReal x, dReal y, dReal z);
void dWorldGetGravity (dWorldID, dVector3 gravity);

Set and get the world's global gravity vector. The units are m/s/s, so Earth's gravity vector would be (0,0,-9.81), assuming that +z is up. The default is no gravity, i.e. (0,0,0).

void dWorldSetERP (dWorldID, dReal erp);
dReal dWorldGetERP (dWorldID);

Set and get the global ERP value, that controls how much error correction is performed in each time step. Typical values are in the range 0.1--0.8. The default is 0.2.
void dWorldStep (dWorldID, dReal stepszie);

Step the world. For large systems this will use a lot of memory and can be very slow, but this is currently the most accurate method.

void dWorldQuickStep (dWorldID, dReal stepszie);

Step the world. For large systems this is a lot faster than dWorldStep, but it is less accurate.
Rigid Body Functions

dBodyID dBodyCreate (dWorldID);

Create a body in the given world with default mass parameters at position (0,0,0). Return its ID.

void dBodyDestroy (dBodyID);

Destroy a body. All joints that are attached to this body will be put into limbo (i.e. unattached and not affecting the simulation, but they will NOT be deleted).
Position and Orientation

void dBodySetPosition (dBodyID, dReal x, dReal y, dReal z);
void dBodySetRotation (dBodyID, const dMatrix3 R);
void dBodySetLinearVel (dBodyID, dReal x, dReal y, dReal z);
void dBodySetAngularVel (dBodyID, dReal x, dReal y, dReal z);
const dReal * dBodyGetPosition (dBodyID);
const dReal * dBodyGetRotation (dBodyID);
const dReal * dBodyGetLinearVel (dBodyID);
const dReal * dBodyGetAngularVel (dBodyID);

These functions set and get the position, rotation, linear and angular velocity of the body. After setting a group of bodies, the outcome of the simulation is undefined if the new configuration is inconsistent with the joints/constraints that are present.
Mass and Force

void dBodySetMass (dBodyID, const dMass *mass);
void dBodyGetMass (dBodyID, dMass *mass);

Set/get the mass of the body (see the mass functions).

void dBodyAddForce (dBodyID, dReal fx, dReal fy, dReal fz);
void dBodyAddTorque (dBodyID, dReal fx, dReal fy, dReal fz);

Add forces to bodies. The forces are accumulated on to each body, and the accumulators are zeroed after each time step.

const dReal * dBodyGetForce (dBodyID);
const dReal * dBodyGetTorque (dBodyID);

Return the current accumulated force and torque vector. The returned pointers point to an array of 3 dReals, so the values are only valid until any changes are made to the rigid body system.
void dMassSetSphere (dMass *, dReal density, dReal radius);

Set the mass parameters to represent a sphere of the given radius and density.

void dMassSetBox (dMass *, dReal density, dReal lx, dReal ly, dReal lz);

Set the mass parameters to represent a box of the given dimensions and density. The side lengths of the box along the x, y and z axes are lx, ly and lz.
# Material Density Reference

<table>
<thead>
<tr>
<th>Material</th>
<th>Density (kg/m(^3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balsa wood</td>
<td>120</td>
</tr>
<tr>
<td>Brick</td>
<td>2000</td>
</tr>
<tr>
<td>Copper</td>
<td>8900</td>
</tr>
<tr>
<td>Cork</td>
<td>250</td>
</tr>
<tr>
<td>Diamond</td>
<td>3300</td>
</tr>
<tr>
<td>Glass</td>
<td>2500</td>
</tr>
<tr>
<td>Gold</td>
<td>19300</td>
</tr>
<tr>
<td>Iron</td>
<td>7900</td>
</tr>
<tr>
<td>Lead</td>
<td>11300</td>
</tr>
<tr>
<td>Styrofoam</td>
<td>100</td>
</tr>
</tbody>
</table>


Enabling and Disabling Bodies

Every body can be enabled or disabled. Enabled bodies participate in the simulation, while disabled bodies are turned off and do not get updated during a simulation step. New bodies are always created in the enabled state.

A disabled body that is connected through a joint to an enabled body will be automatically re-enabled at the next simulation step.

Disabled bodies do not consume CPU time, therefore to speed up the simulation bodies should be disabled when they come to rest.

    void dBodyEnable (dBodyID);
    void dBodyDisable (dBodyID);

Manually enable and disable a body.
Joint Create Functions

```cpp
    dJointID dJointCreateBall (dWorldID, dJointGroupID);
    dJointID dJointCreateHinge (dWorldID, dJointGroupID);
    dJointID dJointCreateSlider (dWorldID, dJointGroupID);
    dJointID dJointCreateContact (dWorldID, dJointGroupID, const dContact *);
    dJointID dJointCreateUniversal (dWorldID, dJointGroupID);
    dJointID dJointCreateHinge2 (dWorldID, dJointGroupID);
    dJointID dJointCreateFixed (dWorldID, dJointGroupID);
    dJointID dJointCreateAMotor (dWorldID, dJointGroupID);
```

Create a new joint of a given type. The joint is initially in "limbo" (i.e. it has no effect on the simulation) because it does not connect to any bodies. The joint group ID is 0 to allocate the joint normally. If it is nonzero the joint is allocated in the given joint group. The contact joint will be initialized with the given dContact structure.
Joint Functions

void dJointDestroy (dJointID);

Destroy a joint, removing it from the world. However, if the joint is a member of a group then this function has no effect.

void dJointAttach (dJointID, dBodyID body1, dBodyID body2);

Attach the joint to some new bodies. If the joint is already attached, it will be detached from the old bodies first.
Joint Group Functions

\[ \text{dJointGroupID dJointGroupCreate (int max\_size);} \]

Create a joint group. The max\_size should be always set to 0.

\[ \text{void dJointGroupDestroy (dJointGroupID);} \]

Destroy a joint group. All joints in the joint group will be destroyed.

\[ \text{void dJointGroupEmpty (dJointGroupID);} \]

Empty a joint group. All joints in the joint group will be destroyed, but the joint group itself will not be destroyed.
Ball and Socket Joint

void dJointSetBallAnchor (dJointID, dReal x, dReal y, dReal z);

Set the joint anchor point. The joint will try to keep this point on each body together. The input is specified in world coordinates.
Hinge Joint

void dJointSetHingeAnchor (dJointID, dReal x, dReal y, dReal z);
void dJointSetHingeAxis (dJointID, dReal x, dReal y, dReal z);

Set hinge anchor and axis parameters.
void dJointSetSliderAxis (dJointID, dReal x, dReal y, dReal z);

Set the slider axis parameter.
A universal joint is like a ball and socket joint that constrains an extra degree of rotational freedom. Given axis 1 on body 1, and axis 2 on body 2 that is perpendicular to axis 1, it keeps them perpendicular.

A use of this joint is to attach the arms of a simple virtual creature to its body. Imagine a person holding their arms straight out. You may want the arm to be able to move up and down, and forward and back, but not to rotate about its own axis.
Universal Joint Functions

void dJointSetUniversalAnchor (dJointID, dReal x, dReal y, dReal z);
void dJointSetUniversalAxis1 (dJointID, dReal x, dReal y, dReal z);
void dJointSetUniversalAxis2 (dJointID, dReal x, dReal y, dReal z);

Set universal anchor and axis parameters. Axis 1 and axis 2 should be perpendicular to each other.
Hinge-2 Joint

The hinge-2 joint is the same as two hinges connected in series, with different hinge axes. An example, shown in the picture below is the steering wheel of a car, where one axis allows the wheel to be steered and the other axis allows the wheel to rotate.

Axis 1 can have joint limits and a motor, axis 2 can only have a motor.

Axis 1 can function as a suspension axis. (i.e. the constraint can be compressible along that axis)
Hinge-2 Joint Functions

void dJointSetHinge2Anchor (dJointID, dReal x, dReal y, dReal z);
void dJointSetHinge2Axis1 (dJointID, dReal x, dReal y, dReal z);
void dJointSetHinge2Axis2 (dJointID, dReal x, dReal y, dReal z);

Set hinge-2 anchor and axis parameters. Axis 1 and axis 2 must not lie along the same line.
Contact Joint

The contact joint prevents body 1 and body 2 from inter-penetrating at the contact point. It does this by only allowing the bodies to have an "outgoing" velocity in the direction of the contact normal. Contact joints typically have a lifetime of one time step. They are created and deleted in response to collision detection.

Contact joints can simulate friction at the contact by applying special forces in the two friction directions that are perpendicular to the normal.
Joint Stop and Motor Parameters

When a joint is first created there is nothing to prevent it from moving through its entire range of motion. For example a hinge will be able to move through its entire angle, and a slider will slide to any length.

This range of motion can be limited by setting stops on the joint. The joint angle (or position) will be prevented from going below the low stop value, or from going above the high stop value. Note that a joint angle (or position) of zero corresponds to the initial body positions.

As well as stops, many joint types can have motors. A motor applies a torque (or force) to a joint's degree(s) of freedom to get it to pivot (or slide) at a desired speed. Motors have force limits, which means they can apply no more than a given maximum force/torque to the joint.
Joint Parameter Functions

void dJointSetHingeParam (dJointID, int parameter, dReal value);
void dJointSetSliderParam (dJointID, int parameter, dReal value);
void dJointSetHinge2Param (dJointID, int parameter, dReal value);
void dJointSetUniversalParam (dJointID, int parameter, dReal value);
void dJointSetAMotorParam (dJointID, int parameter, dReal value);
Joint Parameter Functions

**dParamLoStop**  Low stop angle or position. Setting this to -dInfinity (the default value) turns off the low stop. For rotational joints, this stop must be greater than - pi to be effective.

**dParamHiStop**  High stop angle or position. Setting this to dInfinity (the default value) turns off the high stop. For rotational joints, this stop must be less than pi to be effective.

**dParamVel**  Desired motor velocity (this will be an angular or linear velocity).

**dParamFMax**  The maximum force or torque that the motor will use to achieve the desired velocity. This must always be greater than or equal to zero. Setting this to zero (the default value) turns off the motor.

**dParamBounce**  The bouncyness of the stops. This is a restitution parameter in the range 0..1. 0 means the stops are not bouncy at all, 1 means maximum bouncyness.
Collision Detection

- Defines shapes attached to rigid bodies.
- At each step it figures out which bodies touch each other.
- Create contact joints for each point at which bodies touch.
- The collision detection is broken up into two different types
  - **Geometry** – represents a single rigid shape (ex. box or sphere).
  - **Space** – contains other geometries. It is similar to the rigid bodies concept of a “world”.
Space Functions

dSpaceID dHashSpaceCreate(dSpaceID space);

Creates a space. If space is not zero, then it will insert the space inside another space. Typically we will just pass in a zero for the space argument.

void dSpaceDestroy(dSpaceID space);

This destroys a space. All the geometries in the space are also destroyed.
Geometry Functions

dGeomID dCreateSphere(dSpaceID space, dReal radius);

Creates a sphere geometry of the given radius.

dReal dGeomSphereGetRadius(dGeomID geom);

Gets the radius of the given sphere.

dGeomID dCreateBox(dSpaceID space, dReal lx, dReal ly, dReal lz);

Creates a box geometry of the given x,y,z lengths.

void dGeomBoxGetLengths(dGeomID geom, dReal* sides);

Gets the side lengths of the box.

dGeomID dCreatePlane(dSpaceID space, dReal a, dReal b, dReal c, dReal d);

Create a plane geometry where (a,b,c) are the normal vector of length 1, such that a*x+b*y+c*z = d.
void dGeomSetBody(dGeomID geom, dBodyID body);

Connect a geometry to a body. Combines the position and rotation.

void dGeomSetPosition(dGeomID geom, dReal x, dReal y, dReal z);
void dGeomSetRotation(dGeomID geom, dReal* rotation);
  dReal* dGeomGetPosition(dGeomID geom);
  dReal* dGeomGetRotation(dGeomID geom);

Gets and sets the position and rotation of a geometry. Since a geometry is connected to a rigid body, these functions are equivalent to the rigid body get and set functions.
Collision Detection Functions

- A collision detection “world” is created by making a space and then adding geometries to that space. At every step we want to generate a list of contacts for all the geometries that intersect each other. There are two functions which we will use for this:
  - dCollide – intersects two geometries and generates a contact point
  - dSpaceCollide – determines which pairs of geometries may potentially intersect, and calls a callback function *(using a function pointer)* with each candidate pair. The user can make the decision in the callback function whether or not to call dCollide for each pair.
dCollide

```c
int dCollide(dGeom1 geom1, dGeom2 geom2, int max, dContactGeom* contact, int skip);
```

Given two geometries which potentially intersect, generate contact information for them.

`max` specifies the maximum number of contact points to generate contact points to an array of dContactGeom types. The array must be able to hold at least the maximum number of contacts.

`skip` should always be `sizeof(dContactGeom)`.

The function returns the number of contact points generated and stored in the given array.
dSpaceCollide

void dSpaceCollide(dSpaceID space, void* data, dNearCallback* callback);

This determines which pairs of geometries in the space may potentially intersect, and calls the callback function with each candidate pair. The callback function needs to be defined as

void functionname(void* data, dGeomID geom1, dGeom2 geom2);

The data argument is passed from the dSpaceCollide function call. We will not use this, so you can just set it to zero. The geom1 and geom2 arguments are the pair of geometries which are near each other.
Near Callback Function

The following is a typical callback function which can be given to the `dSpaceCollide` function:

```c
void near(void* data, dGeomID geom1, dGeomID geom2)
{
    dContact contact[maxContacts];

    for(int i = 0; i < maxContacts; i++)
    {
        contact[i].surface.mode = dContactBounce | dContactSoftCFM;
        contact[i].surface.bounce = 0.1;
        contact[i].surface.soft_cfm = 0.01;
    }

    int contacts = dCollide(geom1, geom2, maxContacts, &contact[0].geom, sizeof(dContact));

    for( int i = 0; i < contacts; i++ )
    {
        dJointID c = dJointCreateContact(world, contactGroup, &(contact[i]));
        dJointAttach(c, dGeomGetBody(geom1), dGeomGetBody(geom2));
    }
}
```