Contents

Chapter 1
Introduction

The Open Dynamics Engine (ODE) is a free, industrial quality library for simulating articulated rigid body dynamics. For example, it is good for simulating ground vehicles, legged creatures, and moving objects in VR environments. It is fast, flexible and robust, and it has built-in collision detection. ODE is being developed by Russell Smith[1] with help from several contributors[2].

If "rigid body simulation" does not make much sense to you, check out What is a Physics SDK?[3].

This is the user guide for ODE version 0.5. Despite the low version number, ODE is reasonably mature and stable.

1.1 Features

ODE is good for simulating *articulated* rigid body structures. An articulated structure is created when rigid bodies of various shapes are connected together with joints of various kinds. Examples are ground vehicles (where the wheels are connected to the chassis), legged creatures (where the legs are connected to the body), or stacks of objects.

ODE is designed to be used in interactive or real-time simulation. It is particularly good for simulating moving objects in changeable virtual reality environments. This is because it is fast, robust and stable, and the user has complete freedom to change the structure of the system even while the simulation is running.

ODE uses a highly stable integrator, so that the simulation errors should not grow out of control. The physical meaning of this is that the simulated system should not "explode" for no reason (believe me, this happens a lot with other simulators if you are not 5areful). ODE emphasizes speed and stability over physical accuracy.

ODE has

[redacted]

```
cd ode/test
DISPLAY=:0.0 ./test_boxstack
```

CHAPTER 2. HOW TO INSTALL AND USE ODE 4 cd ode/test DISPLAY=:0.0 ./test_boxstack thngs:E

Chapter 3
Concepts

3.1 Background

[Here is where I will write some background information about rigid body dynamics and simulation. But in the meantime, please refer to Baraff's excellent SIGGRAPH tutorial[1]].

3.2 Rigid bodies

A rigid body has various properties from the point of view of the simulation. Some properties change over time:

- Position vector (x,y,z) of the body's point of reference. Currently the point of reference must correspond to the body's center of mass.
- Linear velocity of the point of reference, a vector (vx,vy,vz).
- Orientation of a body, represented by a quatern of,qz) or a 3x3 rotation matrix.
- Angular velocity vector (wx,wy,wz) which describes how the orientation changes over time.

Other body properties are usually constant over time:

- Mass of the body.
- Position of the center of mass with respect to the point of reference. In the current implementation the center of mass and the point of reference must coincide.
- Inertia matrix. This is a 3x3 matrix that describes how the body's mass is distributed around the center of mass.

Conceptually each body has an x-y-z coordinate frame embedded in it, that moves and rotates with the body, as shown in Figure 3.1.

The origin of this coordinate frame is the body's point of reference. Some values in ODE (vectors,

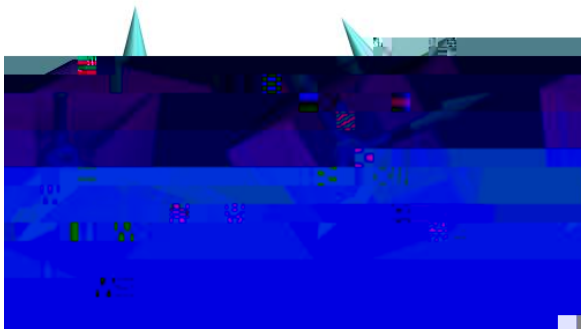


Figure 3.1: *The body coordinate frame.*

3.2.1 Islands and Disabled Bodies

Bodies are connected to each other with joints. An “island” of bodies is a group that can not be pulled apart - in other words each body is connected somehow to every other body in the island.

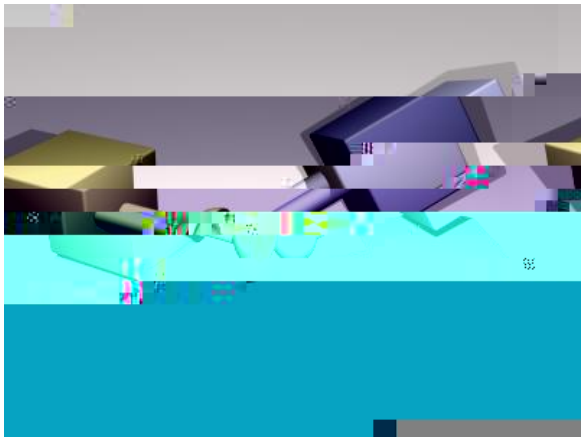


Figure 3.3: *An example of error in a ball and socket joint.*

2. During the simulation, errors can creep in that result in the bodies drifting away from their required positions.

Figure 3.3 shows an example of error in a ball and socket joint (where the ball and socket do not line up).

There is a mechanism to reduce joint error: during each simulation step each joint applies a special force

$$f_m = \mu \quad |f_N| \tag{3.8}$$

and then proceeds to solve for the entire system with these fixed limits (in a manner similar to ap-

[redacted]

Set and get the global CFM (constraint force mixing) value. Typical values are in the range 10^{-9} – 1. The default is 10^{-5} if single precision is being used, or 10^{-10} if double precision is being used.

```
void  dWorldSetAutoDisableFlag (dWorldID, int do_auto_disable);
int   dWorldGetAutoDisableFlag (dWorldID);
void  dWorldSetAutoDisableLinearThreshold (dWorldID, dReal linear_threshold);
dReal dWorldGetAutoDisableLinearThreshold (dWorldID);
void  dWorldSetAutoDisableAngularThreshold (dWorldID, dReal angular_threshold);
dReal dWorldGetAutoDisableAngularThreshold (dWorldID);
void  dWorldSetAutoDisableSteps (dWorldID, int steps);
int   dWorldGetAutoDisableSteps (dWorldID);
void  dWorldSetAutoDisableTime (dWorldID, dReal time);
dReal dWorldGetAutoDisableTime (dWorldID);
```

Set and get the default auto-disable parameters for newly created bodies. See section 6.5 for a description of the auto-disable feature. The default parameters are:

- AutoDisableFlag = disabled
- AutoDisableLinearThreshold = 0.01
- AutoDisableAngularThreshold = 0.01
- AutoDisableSteps = 10
- AutoDisableTime = 0

```
void dWorldImpulseToForce (dWorldID, dReal stepsize,
                           dReal ix, dReal iy, dReal iz, dVector3 force);
```

If you want to apply a linear or angular impulse to a rigid body, instead of a force or a torque, then you can call this function with the desired impulse, and ODE will calculate and return the force that is necessary to apply over a time step in order to produce the desired impulse. Note that the force applied in the next time step may be different, as it depends on the time step and the current velocity.

ection.8188f40260091 -13.549 315e25(vgra5eecat)-250(parameters)-250(tait)-272(arr)-2e(tait)-2suble)-250p315iestion of the w[0

5.1 Stepping Functions

```
void dWorldStep (dWorldID, dReal stepsize);
```

Step the world. This uses a ”big matrix” method that takes time on the order of m^3 and memory on the order of m

Set and get the maximum correcting velocity that contacts are allowed to generate. The default value

6.3 Mass and force

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

Manually enable and disable a body. Note that a disabled body that is connected through a joint to an enabled body will be automatically re-enabled at the next simulation step.

```
int dBodyIsEnabled (dBodyID);
```

Return 1 if a body is currently enabled or 0 if it is disabled.

```
void  dBodySetAutoDisableFlag (dBodyID, int do_auto_disable);
int   dBodyGetAutoDisableFlag (dBodyID);
```

Set and get the auto-disable flag of a body. If the do_

Return a joint attached to this body, given by `index`. Valid indexes are 0 to $n - 1$ where

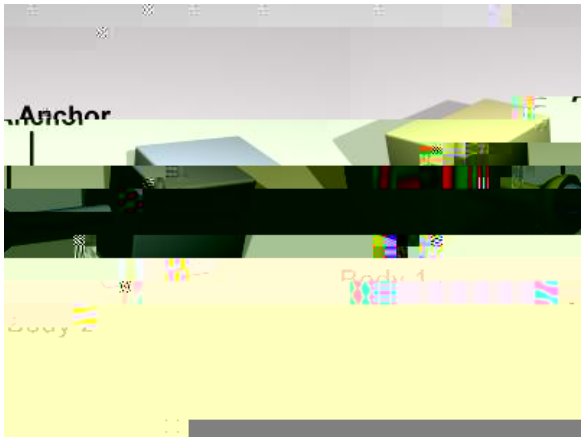
Chapter 7

Joint Types and Joint Functions

7.1 Creating and Destroying Joints

```
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)


Anchor

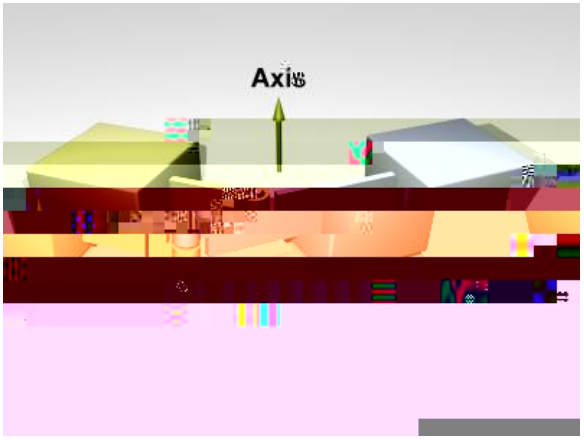


Figure 7.2:

A Universal joint is equivalent to a hinge-2 joint where the hinge-2's axes are perpendicular to each other, and with a perfectly rigid connection in place of the suspension.

Universal joints show up in cars, where the engine causes a shaft, the drive shaft, to rotate along its own axis. At some point you'd like to change the direction of the shaft. The problem is, if you just bend the shaft, then the part after the bend won't rotate about its own axis. So if you cut it at the bend location and insert a universal joint, you can use the constraint to force the second shaft to rotate about the same angle as the first shaft.

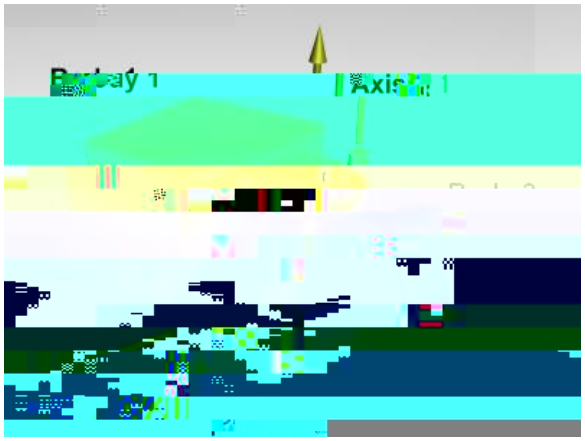


Figure 7.5: *A hinge-2 joint.*

```
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);
```

`surface` is a substructure that is set by the user. Its members define the properties of the colliding surfaces. It has the following members:

- `int mode` - Contact flags. This must always be set. This is a combination of one or more of the following flags:

`dContactMu2`	If not set, use `mu` for both friction directions. If set, use `mu` for friction direction 1, use `mu2` for friction direction 2.
`dContactFDir1`	If set, take `fdir1` as friction direction 1, otherwise automatically compute friction direction 1 to be perpendicular to the contact normal (in which case its resulting orientation is unpredictable).
`dContactBounce`	If set, the contact surface is bouncy, in other words the bodies will bounce off each other. The exact amount of bouncyness is controlled by the `bounce` parameter.
`dContactSoftERP`	If set, the error redue

- `dReal soft_erp` : Contact normal “softness” parameter. This is only set if the corresponding flag is set in `mode`.
- `dReal soft_cfm` : Contact normal “softness” parameter. This is only set if the corresponding flag is set in `mode`.
- `dReal motion1,motion2` : Surface velocity in friction directions 1 and 2 (in m/s). These are

Figure 7.7

```
void dJointSetAMotorAxis (dJointID, int anum, int rel,
                          dReal x, dReal y, dReal z);
void dJointGetAMotorAxis (dJointID, int anum, dVector3 result);
int dJointGetAMotorAxisRel (dJointID, int anum);
```

Set (and get) the AMotor axes. The `anum` argument selects the axis to change (0,1 or 2). Each axis can have one of three "relative orientation" modes, selected by `rel`:

- 0: The axis is anchored to the global frame.
- 1: The axis is anchored to the first body.
- 2: The axis is anchored to the second body.

The default anchor for all joints is (0,0,0). The default axis for all joints is (1,0,0).

When an axis is set it will be normalized to unit length. The adjusted axis is what the axis getting functions will return.

7.5.1 Parameter Functions

Here are the functions that set stop and motor parameters (as well as other kinds of parameters) on a joint:

```
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);
dReal dJointGetHingeParam (dJointID, int parameter);
dReal dJointGetSliderParam (dJointID, int parameter);
dReal dJointGetHinge2Param (dJointID, int parameter);
dReal dJointGetUniversalParam (dJointID, int parameter);
dReal dJointGetAMotorParam (dJointID, int parameter);
```

dParamSuspensionERP	Suspension error reduction parameter (ERP). Currently this is only implemented on the hinge-2 joint.
dParamSuspensionCFM	Suspension constraint force mixing (CFM) value. Currently this is only implemented on the hinge-2 joint.

If a particular parameter is not implemented by a given joint, setting it will have no effect.

Chapter 8

StepFast

NOTE: The StepFast algorithm has been superseded by the QuickStep algorithm: see the dWorldQuickStep()

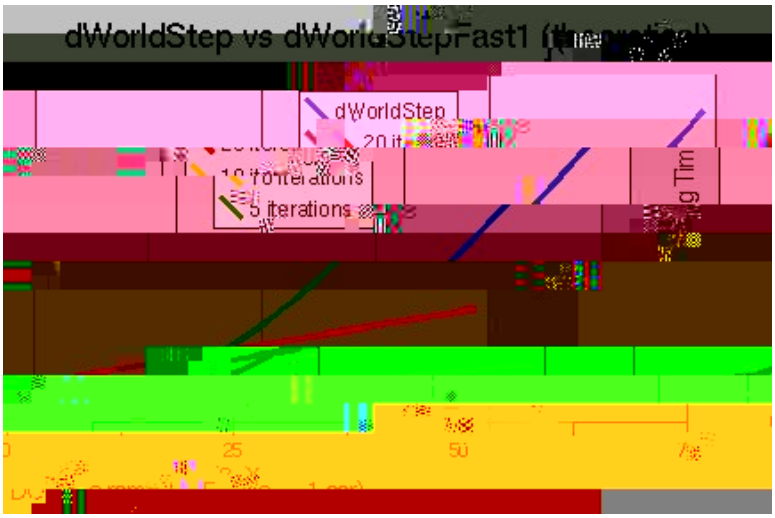


Figure 8.1: *Speed advantage of StepFast.*

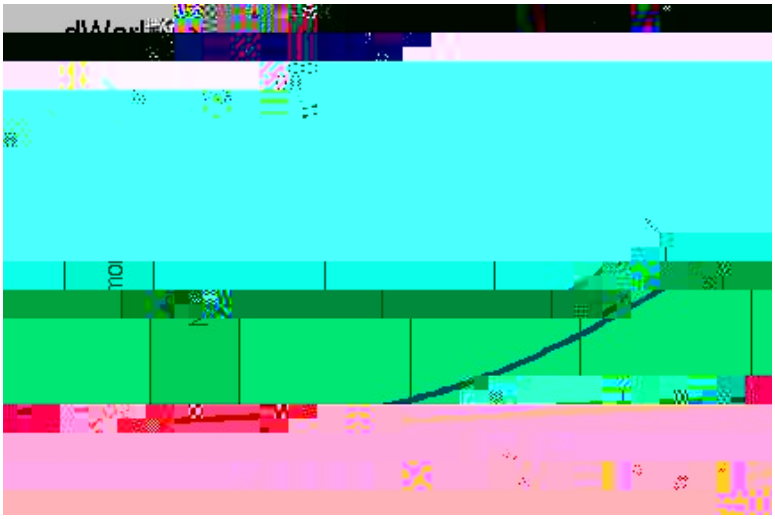


Figure 8.2: *Memory advantage of StepFast.*

```
void dQSetIdentity (dQuaternion q);
```

Set `q` to the identity rotation (i.e. no rotation).

```
void dQFromAxisAndAngle (dQuaternion q, dReal ax, dReal ay, dReal az,
                         dReal angle);
```

Compute `q` as a rotation of `angle`

```
void dMassSetZero (dMass *);
```

Set all the mass parameters to zero.

```
void dMassSetParameters (dMass *, dReal themass,
                         dReal cgx, dReal cgy, dReal cgz,
                         dReal I11, dReal I22, dReal I33,
                         dReal I12, dReal I13, dReal I23);
```

Set the mass parameters to the given values. `themass` is the mass of the body. (`cx`,`cy`,`cz`) is the center of gravity position in the body frame. The `Ixx` values are the elements of the inertia matrix:

```
[ I11 I12 I13 ]
[ I12 I22 I23 ]
[ I13 I23 I33 ]
```

```
void dMassSetSphere (dMass *, dReal density, dReal radius);
void dMassSetSphereTotal (dMass *, dReal total_mass, dReal radius);
```

Set the mass parameters to represent a sphere of the given radius and density, with the center of mass at (0,0,0) relative to the body. The first function accepts the density of the sphere, the second accepts the total mass of the sphere.

```
void dMassSetCappedCylinder (dMass *, dReal density, int direction,
                             dReal radius, dReal length);
void dMassSetCappedCylinderTotal (dMass *, dReal total_mass,
                             int direction, dReal radius, dReal length);
```

Set the mass parameters to represent a capped cylinder of the given parameters and density, with the center of mass at (0,0,0) relative to the body. The radius of the cylinder (and the spherical cap) is `radius`

10.2 Geoms

10.5 Collision detection

A collision detection “world” is created by making a space and then adding geoms to that space. At every time step we want to generate a list of contacts for all the geoms that intersect each other. Three functions are used to do this:

- dCollide() intersects two geoms and generates contact points.
- dSpaceCollide() determines which pairs of geoms in a space may potentially intersect, and calls a callback function with each candidate pair. This does not generate contact points directly, because the user may want to handle some pairs specially - for example by ignoring them or using different contact generating strategies. Such decisions are made in the callback function, which can choose whether or not to call dCollide() for each pair.
- dSpaceCollide2() determines which geoms from one space may potentially intersect with geoms from another space, and calls a callback function with each candidate pair. It can also test a single non-space geom against a space. This function is useful when there is a collision hierarchy, i.e. when there are spaces that contain other spaces.

The collision system has been designed to give the user maximum flexibility to decide which objects will be tested against each other. This is w49 Td[7 -13.5225 -13.5oms aThreebe for non-

Sometimes the category field will contain multiple bits set, e.g. if the geom is a space them you may want to , the category to the union of all the geom categories tha- are contained.

The `data`

```
void dGeomSphereSetRadius (dGeomID sphere, dReal radius);
```

Set the radius of the given sphere.

```
dReal dGeomSphereGetRadius (dGeomID sphere);
```

Return the radius of the given sphere.

```
dReal dGeomSpherePointDepth (dGeomID sphere, dReal x, dReal y, dReal z);
```

Return the depth of the point (x,y,z) in the given sphere. Points inside the geom will have positive depth, points outside it will have negative depth, and points on the surface will have zero depth.

10.7.2 Box Class

10.7.6 Triangle Mesh Class

A triangle mesh (TriMesh) represents an arbitrary collection of triangles. The triangle mesh collision system has the following features:

Optional per triangle callback. Allows the user to say if collision with a particular triangle is wanted. If the return value is zero no contact will be generated.

```
typedef void dTriArrayCallback (dGeomID TriMesh, dGeomID RefObject,
                                const int* TriIndices, int TriCount);
void dGeomTriMeshSetArrayCallback (dGeomID g, dTriArrayCallback* ArrayCallback);
dTriArrayCallback *dGeomTriMeshGetArrayCallback (dGeomID g);
```

```
void dGeomTriMeshEnableTC(dGeomID g, int geomClass, int enable);
int dGeomTriMeshIsTCEnabled(dGeomID g, int geomClass);
```

These functions can be used to enable/disable the use of temporal coherence during tri-mesh collision checks. Temporal coherence can be enabled/disabled per tri-mesh instance/geom class pair, currently it works for spheres and boxes. lault for spheres and boxes is 'false'.

The 'enable' param should be 1 for true, 0 for false.

Temporal coherence is optional because allowing it can cause subtle lfficiency problems in situations where a tri-mesh may collide with many different geoms during its lifespan. If you enable temporal coherence on a tri-mesh then these problems can be eased by intermittently calling dGeomTriMesh-ClearTCCache()

Set and get the clean-up mode of geometry transform g

This function takes a class number (

```
struct dGeomClass {
  int bytes;                  // bytes of custom data needed
```

10.10 Utility functions

```
void dClosestLineSegmentPoints (const dVector3 a1, const dVector3 a2,
                                const dVector3 b1, const dVector3 b2,
                                dVector3 cp1, dVector3 cp2);
```

Given two line segments A and B with endpoints `a1-a2` and `b1-b2`, return the points on A and B

ODE currently relies on factorization of a “system” matrix that has one row/column for each DOF removed (this is where the $O(m_2^3)$

11.6 Avoiding singularities

-

- Increase ERP to make the problem less visible.
- Do your own variable sized time stepping somehow.

12.4 How can an immovable body be created?

12.8 Should I scale my units to be around 1.0 ?

Say you need to simulate some behavior on the scale of a few millimeters and a few grams. These small lengths and masses will usually work in ODE with no problem. However occasionally you may experience

12.12 My simple rotating bodies are unstable!

If you have a box whose sides have different lengths, and you start it rotating in free space, you should observe that it just tumbles at the same speed forever. But sometimes in ODE the box will gain speed by itself, spinning faster and faster until it "explodes" (disappears off to infinity). Here is the explanation:

ODE uses a first order semi-implicit integrator. The "semi implicit" means that some forces are calculated as though an implicit integrator is being used, and other forces are calculated as though the integrator is explicit. The constraint forces (applied to bodies to keep the constraints together) are implicit, and the

Thus it is still possible that you may encounter some examples of non-physical behavior.

Chapter 14
ODE Internals

[only notes for now]

- Internally, all 6x1 spatial velocities and accelerations are split into 3x1 position and angular compo-

14.2 Internals FAQ

14.2.1 Why do some structures have a dx prefix and some have a d prefix?

The dx prefix is used for internal structures that should never be visible externally. The d prefix is used for structures that are part of the public interface.

14.2.2 Returned Vectors

There seem to be 2 ways of returning vectors in ODE, e.g.:

```
const dReal* dBodyGetPosition (dxBodyID);
void dWorldGetGravity (dxWorldID, dVector3);
```

Why? The second way is the 'official' way. The first way returns pointers to volatile internal data structures and is less clean API-wise. For a stable API I feel that filling in vectors is cleaner than returning pointers to vectors, for two reasons:

1.