iiD v1.0 User Manual

Contents

Introduction

iiD is a 2D rigid body simulation library written entirely in portable C++. All files are appended with iiD as to make the library easy to integrate into existing projects. All of the library is within the iiD namespace.

Features

iiD has a few interesting features that are quite desirable for DigiPen game projects:

- LCP solver with Sequential Impulses
  - Highly stable, efficient, and deterministic results
- Many types of joints for interesting rigid body interactions
- Very simple to use API
- Library is not thread-safe, but can easily be placed onto an isolated thread
- Simple communication of events and collisions with callbacks
- Islanding simulation and sleeping
- Warm starting
- Generic 2D polygon and sphere collision detection
- Efficient data oriented implementation
- Math is built-in

Dependencies

iiD itself has no dependencies. The test demo uses GLUT and Windows.h in order to run the demo.

Compiler Dependencies

iiD was written in a portable way, allowing it to compile on both G++ and Visual Studio. All other major compilers should be supported as well. Support for any compiling issues will be provided to all game teams interested in using the iiD library.

Glossary

LCP	- Linear complementary problem. Solving constraints can be thought of as an LCP. See [LCP Wikipedia page]() for more details.
Joint	- A constraint between 2 rigid bodies. Joints make 2 rigid bodies interact with each other in interesting ways, and are very useful for gameplay mechanics.
Warm Starting	- Optimization involving storing old LCP solutions and using them to kick start new solutions.
Sleeping	- When a rigid body is not moving around, it does not need to be solved. Sleeping objects will never move until something awake comes into contact with it. This is an optimization.
Constraint	- A rule that forces a rigid body to behave in a certain way. Constraints are solved with sequential impulses.
Sequential Impulses	- A method of using impulses to solve constraints within a physics engine. Originally proposed by Erin Catto.
Rigid Body	- A physics object is referred to as a body, or rigid body. A rigid body contains a shape (either Polygon or Circle), and never deforms.
Restitution	- How bouncy a rigid body is.
OBB	- Oriented bounding box. Pretty much just a box that can rotate.
AABB	- Axis aligned bounding box. This means a box aligned with the x and y axes, and cannot ever rotate.

Feedback and Reporting Bugs

I'm very interested in hearing feedback from any DigiPen team that uses the physics engine. I also need to know about any potential bugs. Please contact me at: r dot gaul at digipen dot edu

Distance Units

The iiD engine works with an arbitrary distance unit. The standard distance of a medium sized object has a width of one. I recommend zooming in if your simulation feels slow and floaty.

Creating a Scene

All rigid bodies in a simulation belong to a `Scene`. The `Scene` class is one of the only parts of the iiD engine the user interacts with. To create a `Scene` simply create one like so:

```
Vec2 gravity( 0, -9.8f );
const float dt = 1.0f / 60.0f;
Renderer renderer;
iiD::Scene scene( gravity, dt, &renderer, 10 );
```

Rendering iiD

The `Renderer` is a small interface used for debug drawing. All rendering for iiD must be done externally to iiD.

Simulating Rigid Bodies

Once a scene has been constructed, all that is left to do is to add some rigid bodies to it. To create a rigid body the scene expects a `BodyDef` and `Shape` to be provided. Here is an example:

```
iiD::Polygon poly;
iiD::BodyDef def;
def.material.Set( iiD::Material::Wood );
poly.SetAsBox( 1.0f, 1.0f );
iiD::Body *myBody = scene.CreateBody( poly, def );
```

The `BodyDef` is a small struct that contains information about how to create a rigid body. Things like position, the material the body is made of (restitution and friction), orientation, and more are set here. The example code above will default the position (0, 0), and create a rigid body with a polygon shape as an oriented bounding box (OBB).

After bodies have been created, the scene needs to update them. To update the scene call the `Step` function:

```
scene.Step( );
```

Materials

A rigid body has a friction coefficient, density, and a restitution coefficient. Both of these together determine how bouncy and sticky (when sliding) a rigid body behaves. It can be annoying to come up with arbitrary coefficients, so a naming convention was put together. iiD comes packaged with some types of materials, like Wood, Metal and Pillow. To set a material type, see the code example from the previous section.

Alternatively, the friction and restitution can be manually set to custom values.

Density and Mass

Mass is calculated on a per-shape basis. Mass can be manually set, but does not need to be. Instead it is recommended to set the density of shapes through the material within the `BodyDef`.

Static Bodies

A rigid body can be static. Static bodies never move, ever. Static bodies usually represent level geometry, and are much more inexpensive to simulate than dynamic bodies. To create a static body, set the material type to `Static`, or set the `is_static` Boolean in the `BodyDef` to true. Static bodies have a density of zero.

Creating a Polygon

Creating a polygon is easy. A polygon requires a list of vertices in world coordinates. Here is an example of creating a triangle:

```
        iiD::Polygon poly;
        iiD::BodyDef def;
        Vec2 v[] = {
          Vec2( -1.0f, 0.0f ),
          Vec2(  1.0f, 0.0f ),
          Vec2(  0.0f, 1.0f ),
        };
        poly.Set( v, 3 );
        def.tx.Set( 5.0f, 0.5f, 0 );
        iiD::Body *triangle = scene.CreateBody( poly, def );
```

The vertices do *not* need to be dynamically allocated.

Creating a Circle

Creating a circle is very simple:

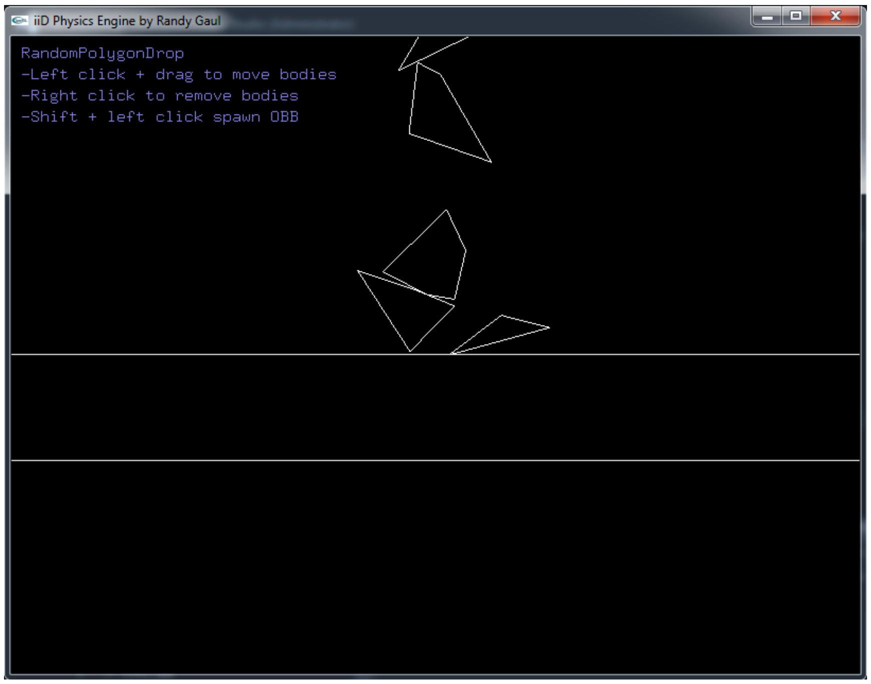
```
        iiD::BodyDef def;
        iiD::Circle circle( 5 );
        iiD::Body *myCircle = scene.CreateBody( circle, def );
```

Creating an OBB

An easy to use function called `SetAsBox` was created to aid in setting up OBBs. Call this function from the `Polygon` itself before creating a rigid body. `SetAsBox` excepts half width and height values.

Demo

The iiD demo has many demo simulations. These simulations provide examples on how to use the iiD engine, and I suggest everyone to refer to the demos regularly as you learn to use the iiD library.



```
RandomPolygonDrop
-Left click + drag to move bodies
-Right click to remove bodies
-Shift + left click spawn OBB
```

Here is a short video of the demo on youtube.

Shapes

There are two types of shapes in iiD: polygons and circles. All polygons **MUST** be convex. Please see the Simulating Rigid Bodies section on how to create these shapes.

Each rigid body must contain one shape in order to be simulated. A future version of the iiD library will allow rigid bodies to contain more than one shape, in order to support convex and interesting compositions.

Point to Shape Test

All shapes implement an interface that allows the user to see if a point collides with a certain shape.

```
shape->TestPoint( Vec2( 1.0f, 2.0f ) );
```

Scene Queries

Raycasting

Currently no form of raycasting is implemented in iiD. Raycasting is essential for lots of gameplay types, and so raycasting will be implemented in the near future.

AABB Query

It is often times useful to test an AABB with a scene to see if any rigid bodies lay within its boundaries. To do this, examine the following code:

```
bool myCallBack( iiD::Body *body )
{
  // Do something with body
  // ...

  // Return true to search for more hits
  // Return false to end query
  return false;
}

scene.QueryAABB( myCallBack, myAABB );
```

As you can see your callback should be very efficient, as it will be called for each object that is found to be intersecting with the provided `AABB`. Returning true from a callback will continue the query within the scene and look for more intersections. Returning false will end the query right then and there. This allows the user to fine-tune and optimize their own queries.

Point Query

Checking to see if any rigid bodies in a scene overlap a point in world space can be quite useful. To do so is similar to querying an `AABB`, except instead of providing an `AABB` a single point is provided. Here is a code example:

```
bool myCallBack( iiD::Body *body )
{
  // Do something with body
  // ...
```

```
    // Return true to search for more hits
    // Return false to end query
    return false;
  }

  Vec2 myPoint( 0.0f, 0.0f );

  scene.QueryPoint( myCallBack, myPoint );
```

Know when Bodies Collide

A certain callback is provided to the scene, and will be called whenever two rigid bodies collide with one another. Use the `SetContactCallback` function on the scene class in order to set this callback.

Ghosts

Sometimes it is useful to have a rigid body that does not resolve collisions, but only reports them. To create a body like this, simply set the `is_ghost` bool within the `BodyDef`.

Joints

A joint creates some rules on how two specific bodies will interact with one another. Joints are useful for creating interesting levels and game mechanics. Joints can be used to create rag dolls, bridges, breakable compositions of bodies, motors, and more.

By default bodies connected by two joints do not collide, however this can be changed by setting a bool within the JointDef of a joint type.

Soft Joints

Some joints can be *soft*. A soft joint will not be completely rigid, and will give way to strong forces. Soft joints are great for creating springs or squishy things.

A soft joint has two parameters that need to be placed into the JointDef of the joint type. One parameter is called the frequency. The `frequencyHz` of a joint represents how soft or squishy the joint is.

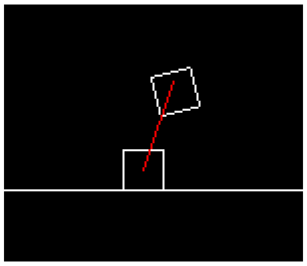
The other parameter is the damping ratio. The damping ratio lowers the velocity of the bodies involved in the joint. This is useful for controlling the stability and behavior of the joint.

JointDef

All joints, just as rigid bodies, require a definition to be created. To create a joint a JointDef must be first provided. Each different type of joint has a different JointDef.

Distance Joint

The distance joint ensures that two points on two rigid bodies maintain a constant distance from another. You can imagine a massless, rigid rod connecting the two points together.



Example of a soft distance joint.

To create a distance joint, examine the following example code:

```
iiD::DistanceJointDef jointDef;
jointDef.Initialize( body1, body2, anchorWorldA, anchorWorldB );
jointDef.frequencyHz = 1.5f;
jointDef.dampingRatio = 0.05f;
scene.CreateJoint( jointDef );
```

See the `DistanceJoints` function in `main.cpp` of the demo for more example code.

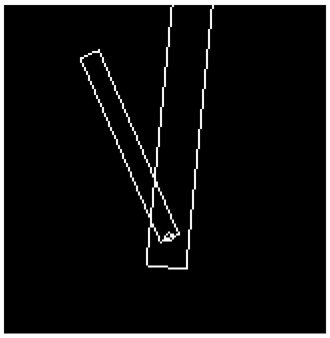
Ropes

The distance joint can be configured to act as a rope. Ropes will not be solved unless the distance between the two bodies is large enough. Setting up a distance joint as a rope is simple:

```
iiD::DistanceJointDef jointDef;
jointDef.Initialize( bodyA, bodyB, anchorWorldA, anchorWorldB );
jointDef.rope = true;
jointDef.length = 15.0f;
jointDef.frequencyHz = 0.5f;
jointDef.dampingRatio = 0.1f;
scene.CreateJoint( jointDef );
```

Revolute Joint

The revolute joint attaches two points, one on each body, together. The bodies can only rotate around the point together, but the point does not separate.



Revolute joint connecting two rods together.

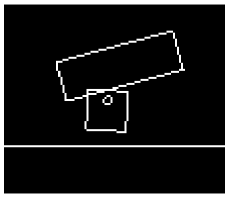
Creating a revolute joint involves setting an anchor point specified in world space:

```
iiD::RevoluteJointDef rdef;
rdef.Initialize( chassis, frontWheel, Vec2( 1.5f, 1.0f ) );
rdef.enableMotor = true;
rdef.motorSpeed = 5.0f;
rdef.maxTorque = 2.0f;
scene.CreateJoint( rdef );
```

The revolute joint can also limit the angles in which the bodies can rotate. Additionally, the revolute joint can be setup to simulate a rotational motor.

Weld Joint

The weld joint connects two rigid bodies together by a single point. The two connected bodies must move with constant relative position and orientation. Weld joints are cool, but be warned: constraints are not 100% rigid, and weld joints will bend under high stress.



Weld joint connecting two rectangles together.

Here is an example of creating a weld joint:

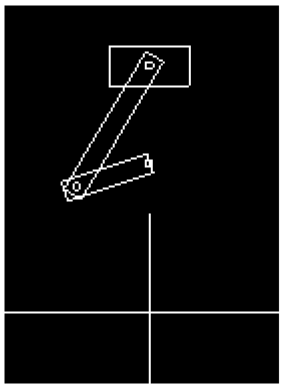
```
iiD::WeldJointDef wdef;
wdef.Initialize( b1, b2, Vec2( 1.0f, 6.5f ) );
scene.CreateJoint( wdef );
```

Angle Joint

The angle joint constraints the angle to a constant between two rigid bodies. This joint was created mostly for debugging purposes, though can be useful on occasion:

Prismatic Joint

The prismatic joint constraints two rigid bodies so their relative motion must be along an axis, of which is relative to the two bodies. You can think of this as the bodies must travel along a line. This is create for creating sliding doors and other mechanical systems.



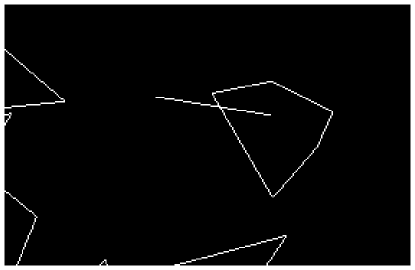
Prismatic joint constraining a box along a line. Revolute motors power a crankshaft.

Here is example code for setting up a prismatic joint:

```
iiD::PrismaticJointDef pdef;
pdef.collisionEnabled = true;
pdef.Initialize( b1, plat, Vec2( 0.0f, 0.0f ), Vec2( 0.0f, 1.0f ) );
scene.CreateJoint( pdef );
```

Mouse Joint

The mouse joint is actually a soft revolute joint, made specifically to be used with the mouse for grabbing and moving rigid bodies around. The mouse joint is harder to setup than the rest of the joint types, so I recommend carefully examining the mouse-related code in `main.cpp` of the demo to learn more about this joint.



Mouse joint pulling a rigid body to the left.

Limitations

There are a few limitations to be aware of when using iiD in order to have a stable simulation:

- Fast moving objects can tunnel (teleport) through each other. Limit the maximum velocity of all objects, or create level geometry with thicker walls to prevent this. Also try to avoid tiny or very skinny rigid bodies.
- Very heavy objects on top of very light objects can cause sinking.
- N^2 broadphase (currently).
- No raycasting (currently).

Feature To-Do List

Here is a short list of features to be added in the near future to the iiD library (listed in arbitrary order):

- Body compositing. Bodies can hold multiple shapes. Shapes within a single body do not collide with one another. Useful for creating compositions of shapes.
- Raycast queries upon the world.
- Dynamic AABB tree Broadphase.
- Prismatic motor and limit.
- Soft revolute joint.
- Post projection positional correction.
- Object slicing and fracturing.
- Buoyancy physics for water simulation.

Contact

r dot gaul at digipen dot edu