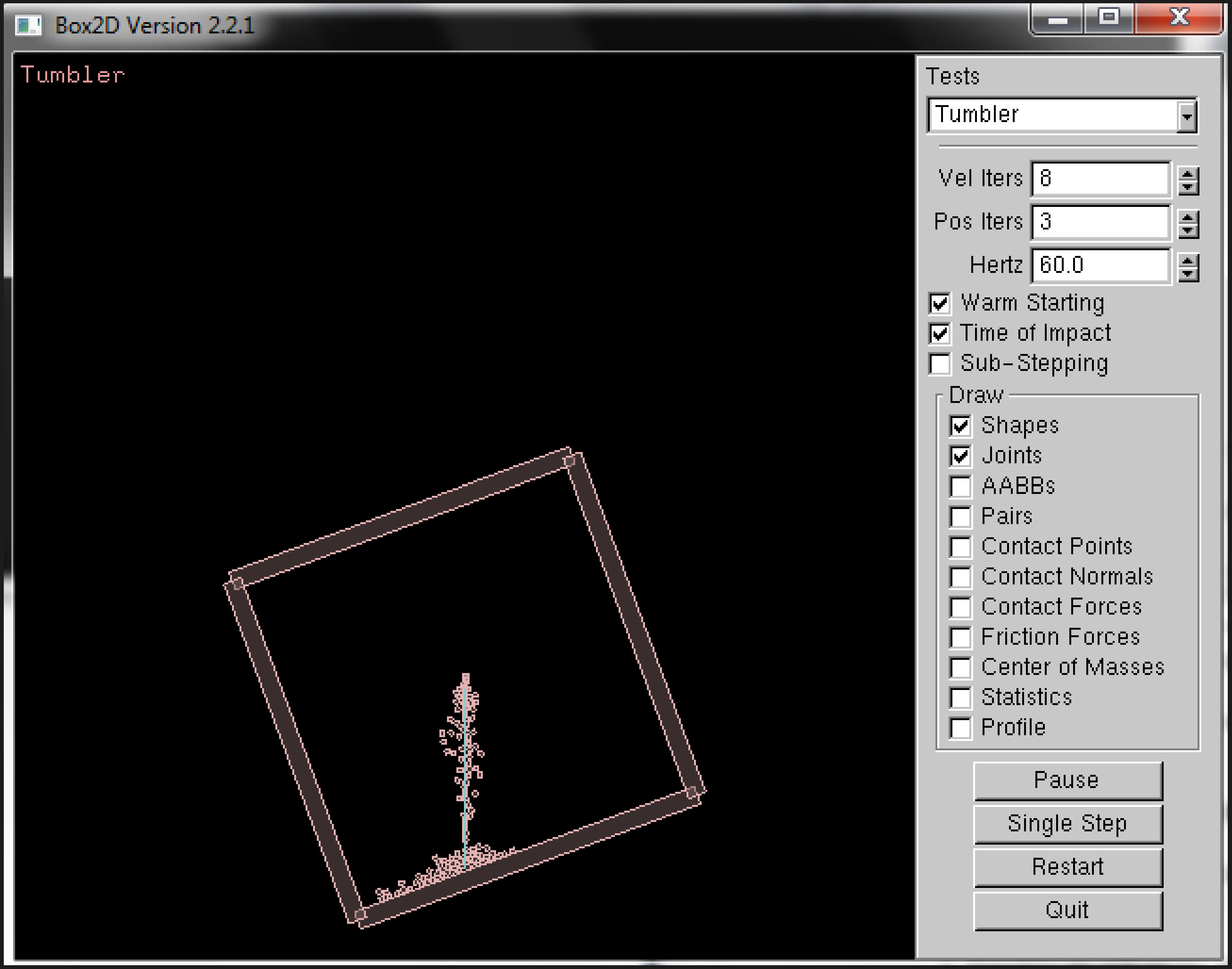
CSE 380 – Computer Game Programming
Box2D

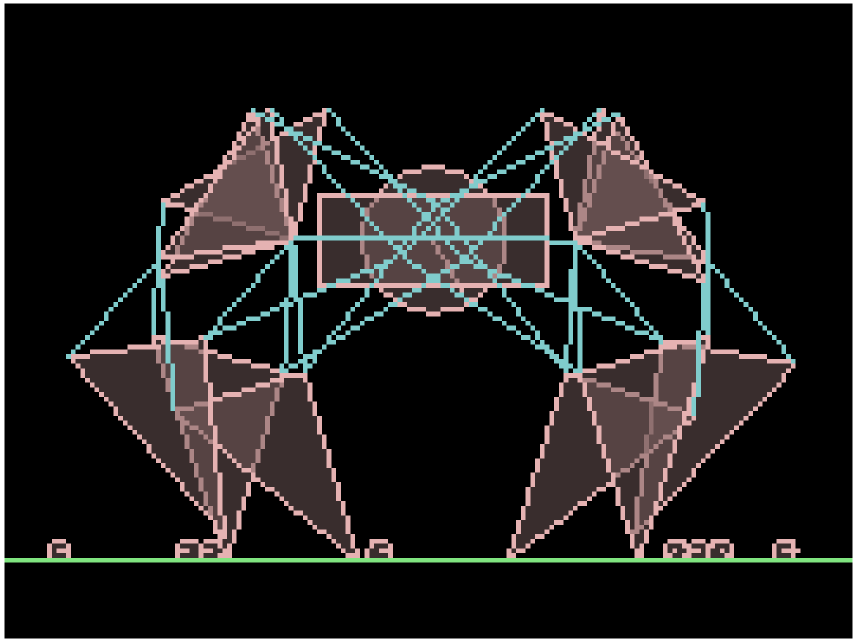


Vel Iters 8
Pos Iters 3
Hertz 60.0
[✓] Warm Starting
[✓] Time of Impact
[ ] Sub-Stepping
Draw
[✓] Shapes
[✓] Joints
[ ] AABBs
[ ] Pairs
[ ] Contact Points
[ ] Contact Normals
[ ] Contact Forces
[ ] Friction Forces
[ ] Center of Masses
[ ] Statistics
[ ] Profile

Box2D TestBed

What is Box2D?

- A 2D rigid body simulation engine
  - i.e. a 2D physics library
- Written in C++ with lots of C



Rigid Body Physics?

- Not Deformable
- Can be translated and rotated
- As opposed to?
  - Soft Body Physics

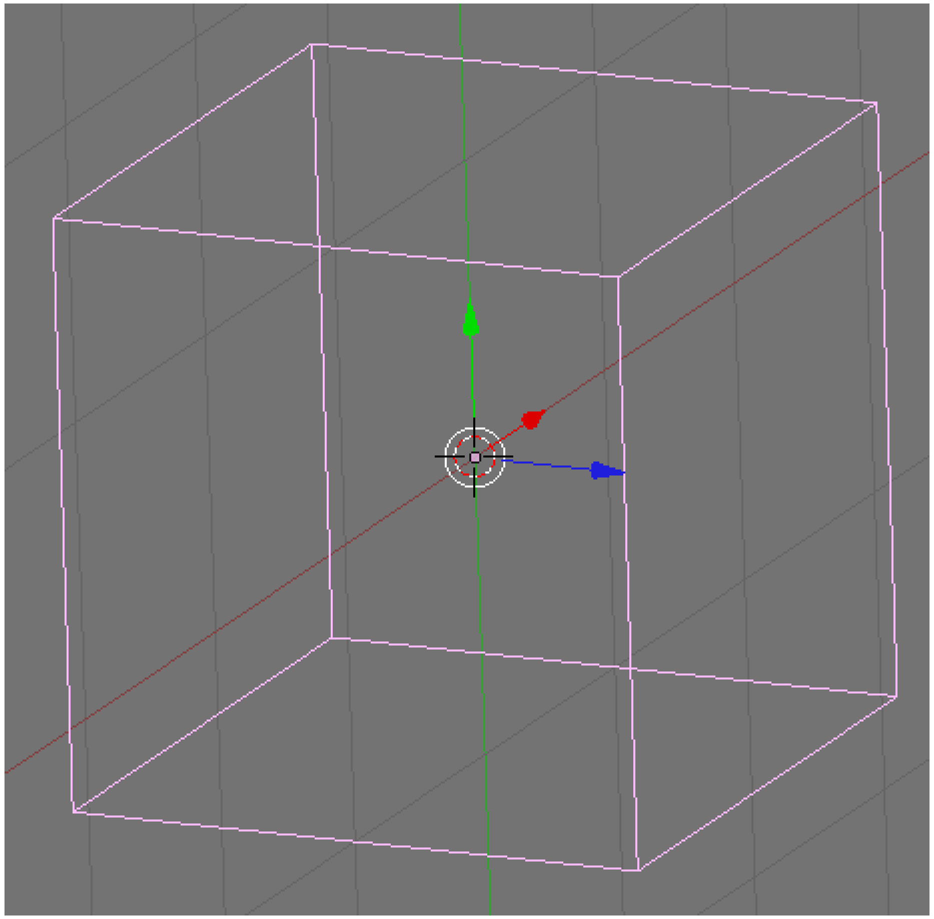
http://en.wikipedia.org/wiki/File:Blender3D_HiddenTextSoftbody.ogg

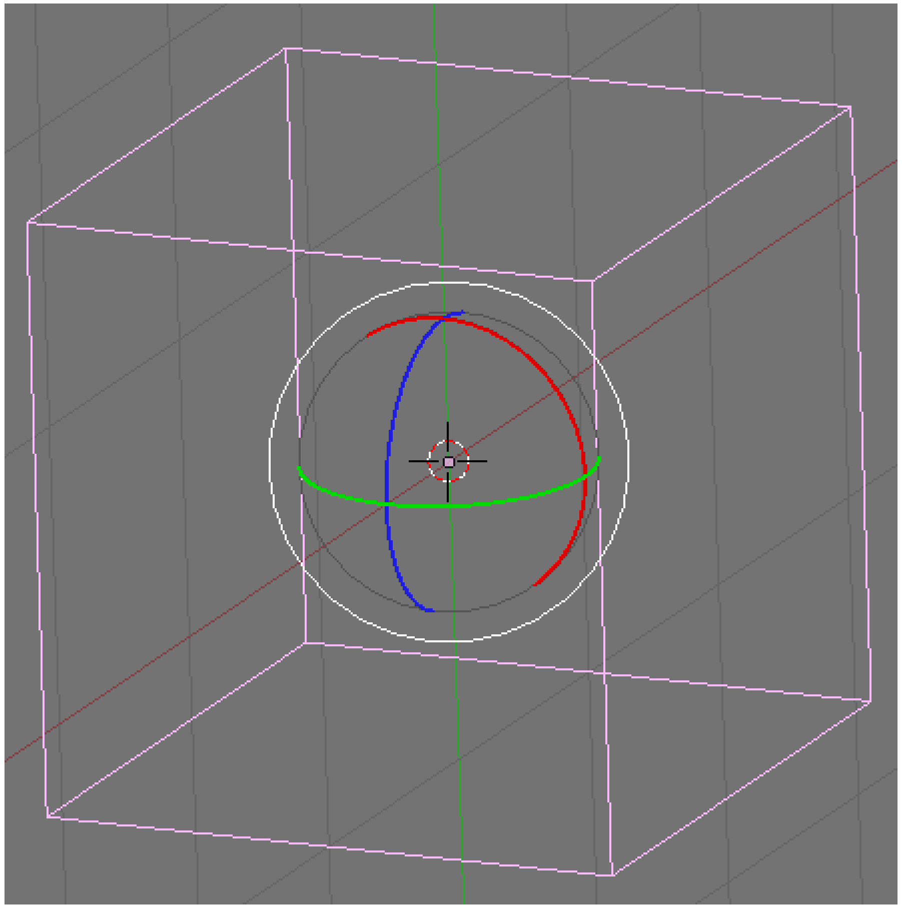
Fundamental Objects and Terms

- The core objects used by the system:
  - **shape** (i.e. circle, polygon, etc.)
  - **rigid body** (we know what that is)
  - **fixture** (gives physical properties to shape)
    - density, friction, restitution, etc.
  - **constraint** (limits a degree of freedom)
  - **contact constraint** (prevents penetration)
  - **joint** (holds 2 ore more rigid bodies together)
  - **joint limit** (restricts range of motion of joint)
  - **joint motor** (drives motion of connected bodies)
  - **world** (collection of bodies, fixtures, & constraints)
  - **solver** (advances time and resolves constraints)
  - **First TOI** (time of impact)

Degrees of Freedom?

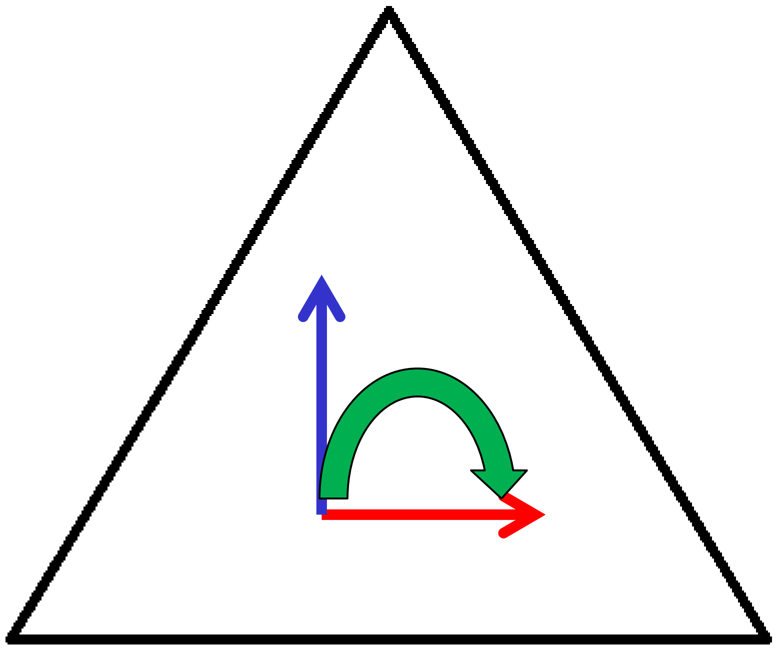
- How many degrees of freedom does a 3D object have?
  - 6 (3 translate, 3 rotate)





How about a 2D object?

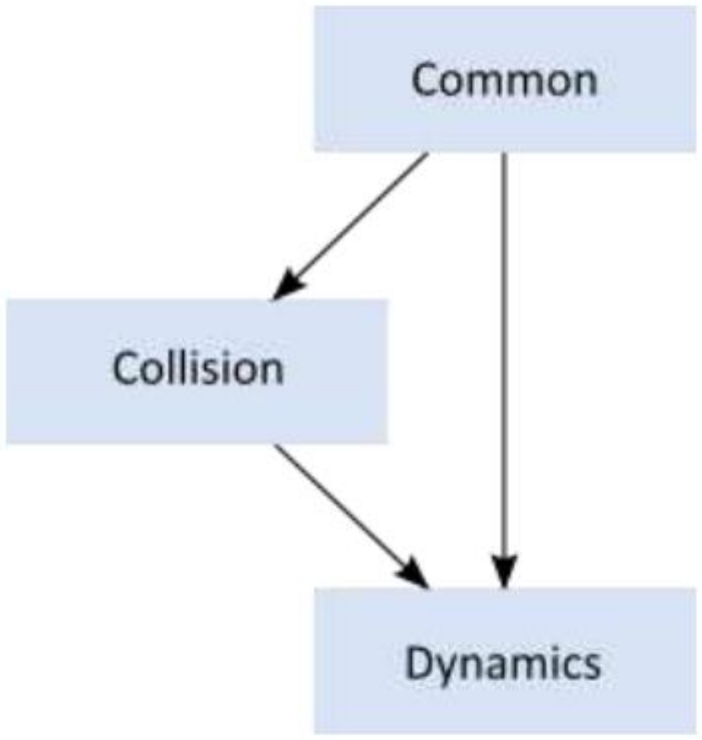
- 3 (2 translate, 1 rotate



- Constrain a degree of freedom?
  - Ex: pin object to wall
    - it can rotate but not translate

Box2D Modules

- **Common**
  - memory allocation
  - math
  - settings
- **Collision**
  - defines shapes
  - broad phase
  - collision functions & queries
- **Dynamics**
  - simulation world
  - bodies, fixtures, & joints



Box2D module dependencies

Box2D Units

- Uses MKS units system
  - meters-kilograms-seconds
  - radians for angles
  - floating point numbers w/ tolerances
- Recommended sizes:
  - moving objects: 0.1 to 10 meters
  - static objects: 50 meters
- But what if you want to use pixels?
  - render using your own scaling system

The Memory Factory

- Box2D manages all memory
- What if you want a new body or joint?
  - ask the world, it uses the factory pattern

```
b2Body* b2World::CreateBody(const b2BodyDef* def)
b2Joint* b2World::CreateJoint(const b2JointDef* def)
void b2World::DestroyBody(b2Body* body)
void b2World::DestroyJoint(b2Joint* joint)
```

- b2Body objects can make b2Fixture objects

```
b2Fixture* b2Body::CreateFixture(const b2FixtureDef* def)
void b2Body::DestroyFixture(b2Fixture* fixture)
```

Box2D's SOA

- SOA?
  - Small Object Allocator
- Problem: Allocating lots of small objects via malloc or new is bad. Why?
  - memory fragmentation
  - memory search cost
- So, never new or malloc, ask the SOA
  - b2BlockAllocator

b2BlockAllocator

- The b2World has one
- Maintains growable pools of memory blocks
- What does it do:
  - when a request is made for a block?
    - The SOA returns a best-fit block
  - when a block is freed?
    - The SOA returns it to the pool
- Both operations are fast and cause little heap traffic

What if you want to attach data?

- We have the **`void*`**
- **`b2Fixture`**, **`b2Body`**, **`b2Joint`**
  - all have **`void*`** instance variables
- So what? What can it hold?
  - the address of anything

Why would you want to attach data?

- Think AI and your Bots. Ex:
  - apply damage to a bot using a collision result
  - play a scripted event if the player is inside an AABB
  - access a game structure when Box2D notifies you that a joint is going to be destroyed

- Attaching data ex:

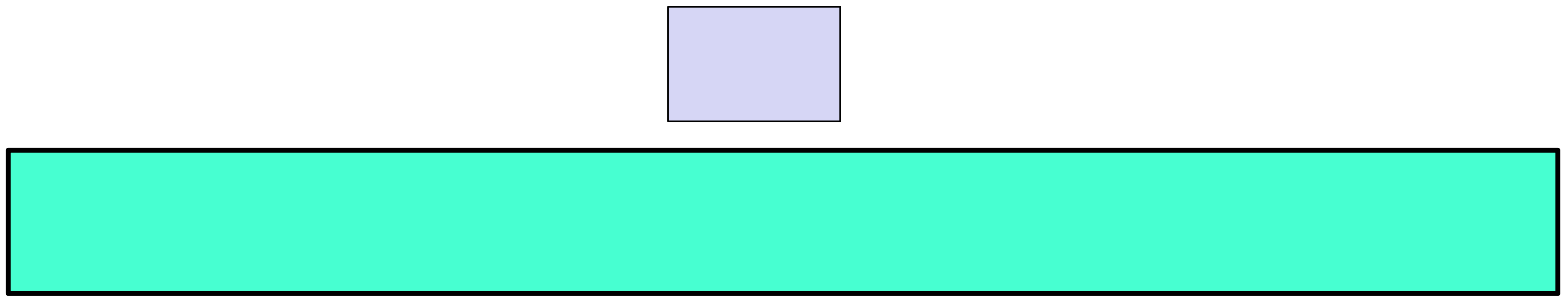
```
MyBot *bot = new MyBot();
b2BodyDef bodyDef;
bodyDef.userData = bot;
bot->body = box2Dworld->CreateBody(&bodyDef);
```

Getting up and Running

- Look at HelloWorld

- This example creates two objects:
  - large ground box
  - small dynamic box

- Note that it doesn't render them



Steps for getting started

1. Define a gravity vector
2. Create a world
3. Create a Ground Box
4. Create a Dynamic Body
5. Setup the integrator
6. Setup the constraint solver

Steps for getting started

1. Define a gravity vector. Ex:

```
b2Vec2 gravity(0.0f, -10.0f);
```

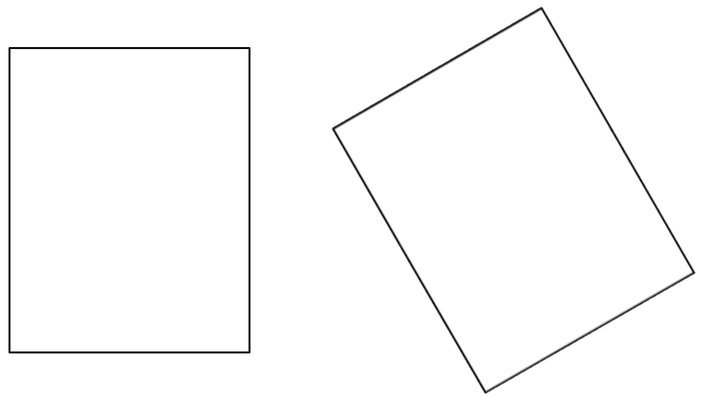
2. Create a world (i.e. b2World)
   - the physics but that manages memory
   - put it where you like (stack, heap, global). Ex:

```
bool doSleep = true;
b2World world(gravity, doSleep);
```

- Now let's add stuff to the world. What? Bodies

Adding bodies to the world

- What's a body?
  - a rigid body
  - i.e. a bounding volume



- To do so:
  - Define a body with position, damping, etc.
  - Use the world object to create the body.
  - Define fixtures with a shape, friction, density, etc.
  - Create fixtures on the body.

So let's continue

3. Create a Ground Box

```
b2BodyDef groundBodyDef;
groundBodyDef.position.Set(0.0f, -10.0f);
b2Body* groundBody = world.CreateBody(&groundBodyDef);
b2PolygonShape groundBox;
groundBox.SetAsBox(50.0f, 10.0f);
groundBody->CreateFixture(&groundBox, 0.0f);
```

- Note that
- What's the width and height of the ground box?
  - 100 x 20
  - note that by default, bodies:
    - are static
    - have no mass

And let's add a dynamic body

4. Create a Dynamic Body
   - set the body type to b2_dynamicBody if you want the body to move in response to forces

```
b2BodyDef bodyDef;
bodyDef.type = b2_dynamicBody;
bodyDef.position.Set(0.0f, 4.0f);
b2Body* body = world.CreateBody(&bodyDef);
b2PolygonShape dynamicBox;
dynamicBox.SetAsBox(1.0f, 1.0f);
b2FixtureDef fixtureDef;
fixtureDef.shape = &dynamicBox;
fixtureDef.density = 1.0f;
fixtureDef.friction = 0.3f;
body->CreateFixture(&fixtureDef);
```

Setup the integrator and constraint solver

- What's the integrator?
  - simulates the physics equations
  - does so at timed intervals
    - i.e. your frame rate

- What do you need to do?
  - specify the time step
  - recommendation: at least 60 Hz for physics systems

And the constraint solver?

- **constraint** (limits a degree of freedom)
- The constraint solver solves all the constraints in the simulation, one at a time.
- A single constraint can be solved perfectly.
- When we solve one constraint, we slightly disrupt other constraints.
- To get a good solution, we need to iterate over all constraints a number of times.

Done in 2 phases

- Velocity phase:
  - Solver computes impulses necessary for bodies to move correctly
- Position phase:
  - Solver adjusts positions of the bodies to reduce overlap and joint detachment
- Each phase has its own iteration count
  - Recommendation: 8 for velocity, 3 for position

Finishing up our steps

5. Setup the integrator

```
float32 timeStep = 1.0f / 60.0f;
```

6. Setup the constraint solver

```
int32 velIterations = 6;
int32 posIterations = 2;
…
// EACH FRAME
world.Step(timeStep, velIterations, posIterations);
```

Hello World

- Here’s our game loop

```
for (int32 i = 0; i < 60; ++i)
{
   world.Step(timeStep, velIterations, posIterations);
   b2Vec2 position = body->GetPosition();
   float32 angle = body->GetAngle();
   printf("%4.2f %4.2f %4.2f\n",
              position.x, position.y, angle);
}
```

References

- Box2D v2.2.1 User Manual
  - *Copyright © 2007-2011 Erin Catto*