Bullet Things

May 26, 2008

Contents

1 Stepping the World

1.1 What do the parameters to btDynamicsWorld::stepSimulation mean?

Here's the prototype:

```
btDynamicsWorld::stepSimulation(
    btScalar timeStep,
    int maxSubSteps=1,
    btScalar fixedTimeStep=btScalar(1.)/btScalar(60.));
```

The first parameter is the easy one. It's simply the amount of time to step the simulation by. Typically you're going to be passing it the time since you last called it

Bullet maintains an internal clock, in order to keep the actual length of ticks constant. This is pivotally important for framerate independance. The third parameter is the size of that internal step.

The second parameter is the maximum number of steps that bullet is allowed to take each time you call it. If you pass a very large timestep as the first parameter [say, five times the size of the fixed internal time step], then you must increase the number of maxSubSteps to compensate for this, otherwise your simulation is "losing" time.

1.2 How do I use this?

It's important that timeStep is always less than maxSubSteps*fixedTimeStep, otherwise you are losing time. Mathematically,

timeStep < *maxSubSteps * fixedTimeStep*

When you are calculating what figures you need for your game, start by picking the maximum and minimum framerates you expect to see. For example, I cap my game framerate at 120fps, I guess it might conceivably go as low as 12fps.

At 120fps, the timeStep I'm passing will be roughly 1/120th of a second, or 0.0083. The default fixedTimeStep is 1/60th of a second, or 0.017. In order to meed the equation above, timestep doesn't need to be greater than 1. At 120fps, with 1/60th of a second a tick, you're looking at interpolating [as opposed to more accurately simulating] one in every two ticks.

At 12fps, the timeStep will be roughly 1/12th of a second, or 0.083. In order to meet the equation above, maxSubSteps would need to be at least 5. Every time the game spikes a little and the framerate drops lower, I'm still losing time. So run with 6 or 7. At 12fps, with 1/60th of a second per tick, you're going to be getting maybe five genuine simulation steps every tick.

My call to btDynamicsWorld::stepSimulation here would therefore be

```
mWorld->stepSimulation(time-since-last-call, 7);
```

1.3 Any other important things to know?

1.3.1 fixedTimeStep resolution

By decreasing the size of fixedTimeStep, you are increasing the "resolution" of the simulation.

If you are finding that your objects are moving very fast and escaping from your walls instead of colliding with them, then one way to help fix this problem is by decreasing fixedTimeStep. If you do this, then you will need to increase maxSubSteps to ensure the equation listed above is still satisfied.

The issue with this is that each internal "tick" takes an amount of computation. More of them means your CPU will be spending more time on physics and therefore less time on other stuff. Say you want twice the resolution, you'll need twice the maxSubSteps, which could chew up twice as much CPU for the same amount of simulation time.

1.3.2 maxSubSteps == 0 ?

If you pass maxSubSteps=0 to the function, then it will assume a variable tick rate. Every tick, it will move the simulation along by exactly the timeStep you pass, in a single tick, instead of a number of ticks equal to fixedTimeStep.

This is not officially supported, and the death of determinism and framerate independance. Don't do it.

1.3.3 Bullet interpolates stuff, aka, maxSubSteps == 1 ?

When you pass bullet maxSubSteps > 1, it will interpolate movement for you. This means that if your fixedTimeStep is 3 units, and you pass a timeStep of 4, then it will do exactly one tick, and estimate the remaining movement by 1/3. This saves you having to do interpolation yourself, but keep in mind that maxSubSteps needs to be greater than 1.

1.4 Callbacks

Every time that bullet does a complete internal tick, it has the ability to call a callback of your choosing. This is useful, for example, if you are building a spaceship and you want each spaceship to have an individual speed cap. No matter how many substeps bullet will do, your spaceship can have its speed limited at the end of every tick, which helps framerate independance.

The prototype function for the callback is this:

```
typedef void (*btInternalTickCallback)(const btDynamicsWorld *world, btScalar timeStep);
```

and the appropriate call to add it to the world is

```
void btDynamicsWorld::setInternalTickCallback(btInternalTickCallback cb);
```

Your code to use it might look something like this:

```
void myTickCallback(const btDynamicsWorld *world, btScalar timeStep) {
    printf("The world just ticked by %f seconds\n", (float)timeStep);
}
// And then somewhere after you construct the world:
mWorld->setInternalTickCallback(myTickCallback);
```

2 Collision Things

2.1 Selective collisions using masks

Bullet supports bitwise masks as a way of deciding whether or not things should collide with other things, or receive collisions.

For example, in a spaceship game, you could have your spaceships ignore collisions with other spaceships [the spaceships would just fly through each other], but always collide with walls [the spaceships always bounce off walls].

Your spaceship needs a callback when it collides with a wall [for example, to produce a "plink" sound], but the walls do nothing when you collide with them so they do not need to receive callbacks.

A third type of object, "powerup", collides with walls and spaceships. Spaceships do not receive collisions from them, since we don't want the trajectory of the spaceship changed by collecting a powerup. The powerup object modifies the spaceship from its own collision callback.

In order to do this, you need a bit mask for the walls, spaceships, and powerups:

```
#define BIT(x) (1<<(x))
enum collisiontypes {
    COL_NOTHING = 0, //<Collide with nothing
    COL_SHIP = BIT(1), //<Collide with ships
    COL_WALL = BIT(2), //<Collide with walls
    COL_POWERUP = BIT(3) //<Collide with powerups
}
int shipCollidesWith = COL_WALL;
int wallCollidesWith = COL_NOTHING;
int powerupCollidesWith = COL_SHIP | COL_WALL;
```

After setting these up, simply add your body objects to the world using btDynamicsWorld::addCollisionObject instead of the more usual btDynamicsWorld::addRigidBody, and as the second and third parameters pass your collision type for that body, and the collision mask:

```
btRigidBody ship; // Set up the other ship stuff
btRigidBody wall; // Set up the other wall stuff
btRigidBody powerup; // Set up the other powerup stuff
mWorld->addCollisionObject(ship, COL_SHIP, shipCollidesWith);
mWorld->addCollisionObject(wall, COL_WALL, wallCollidesWith);
mWorld->addCollisionObject(powerup, COL_POWERUP, powerupCollidesWith);
```

2.2 Selective Collisions Using a Custom NearCallback

If you have more types of objects than bits available to you in the masks above, or some collisions are enabled or disabled based on other factors, then you need to add a different nearCallback that implements this logic and only passes on collisions that are the ones you want:

```
void MyNearCallback(btBroadphasePair& collisionPair,
    btCollisionDispatcher& dispatcher,
    btDispatcherInfo& dispatchInfo) {
    // Do your collision logic here
    // Only dispatch the bullet collision information if you want the physics to continue
    dispatcher.defaultNearCallback(collisionPair, dispatcher, dispatchInfo);
}
mDispatcher ->setNearCallback(MyNearCallback);
```

2.3 Callbacks

Bullet supports custom callbacks at various points in the collision system. The callbacks themselves are very simply implemented as global variables that you set to point at appropriate functions. Before you can expect them to be called you must set an appropriate flag in your rigid body:

```
mBody->setCollisionFlags(mShipBody->getCollisionFlags() |
    btCollisionObject::CF_CUSTOM_MATERIAL_CALLBACK);
```

There are three collision callbacks:

2.3.1 gContactAddedCallback

This is called whenever a contact is added. From here, you can modify some properties [eg friction] of the contact point

typedef bool (*ContactAddedCallback)(btManifoldPoint& cp, const btCollisionObject* colObj0,int partId0,int index0,const btCollisionObject* colObj1,int partId1,int index1);

If your function returns false, then bullet will assume that you did not modify the contact point properties at all.

2.3.2 gContactProcessedCallback

This is called immediately after the collision has been actually processed

typedef bool (*ContactProcessedCallback)(btManifoldPoint& cp,void* body0,void* body1);

2.3.3 gContactDestroyedCallback

This is called immediately after the contact point is destroyed.

typedef bool (*ContactDestroyedCallback)(void* userPersistentData);

3 Code Snippets

3.1 Simple Triangle Meshes

I don't want anything complicated, I just want a simple triangle mesh that stuff can collide with! What's the simplest possible code to "just make a triangle mesh"?

```
btTriangleMesh *mTriMesh = new btTriangleMesh();
while(!done) {
    // For whatever your source of triangles is
    //   give the three points of each triangle:
    btVector3 v0(x0,y0,z0);
    btVector3 v1(x1,y1,z1);
    btVector3 v2(x2,y2,z2);
    // Then add the triangle to the mesh:
    mTriMesh->addTriangle(v0,v1,v2);
}
btCollisionShape *mTriMeshShape = new btBvhTriangleMeshShape(mTriMesh,false);
// Now use mTriMeshShape as your collision shape.
// Everything else is like a normal rigid body
```

3.2 I want to constrain an object to two dimensional movement, skipping one of the cardinal axes

There are a lot of ways of doing this. Here's the one that's working for me.

1. Create a body in space. It doesn't need a position, size, collision object. Just a body *e nihil* that you can attach other bodies to
   (a) It does need its mass to be zero, though, in order to effectively disable it from moving
2. Create a generic6dof joint that restrains movement along the axis you want to restrain movement along
3. Attach your body you want to restrict movement on, to the body you created in the first step

In the case you see here, I have constrained movement along the Z axis. My spaceship travels only along the X and Y axes.

```
btRigidBody zerobody(0,NULL,NULL); // Create the body that we attach things to
btRigidBody spaceship; // Construct your body that will only move in two dimensions
btGeneric6DofConstraint constrict(spaceship, zeroBody,
    btTransform::getIdentity(), btTransform::getIdentity(), false);
// Use limit(axis, a, b) where a>b to disable limits on that axis
constrict->setLimit(0,1,0); // Disable X axis limits
constrict->setLimit(1,1,0); // Disable Y axis limits
constrict->setLimit(2,0,0); // Set the Z axis to always be equal to zero
constrict->setLimit(3,1,0); // Uncap the rotational axes
constrict->setLimit(4,1,0); // Uncap the rotational axes
constrict->setLimit(5,1,0); // Uncap the rotational axes
mWorld->addConstraint(constrict);
```

3.3 I want to cap the speed of my spaceship

What's important here is two things: Firstly, doing it in a manner that doesn't go against general physics coding karma. Secondly, doing it in a way that is framerate-independant.

To avoid messing with general physics coding karma, you should make sure you don't accidentally do this while the physics is actually stepping, and while you'd ideally like to avoid setting properties directly on the body, in this particular case you don't have a choice.

To make sure that this is done in a manner that leads to framerate-indepedance, you need to do it *every* time that bullet ticks internally. Just waiting for stepSimulation to return is insufficient since your spaceship might be above the max velocity for multiple internal ticks.

In short, the best way to do this is by setting the velocity of your spaceship in the physics tick callback.

```
void myTickCallback(const btDynamicsWorld *world, btScalar timeStep) {
    // mShipBody is the spaceship's btRigidBody
    btVector3 velocity = mShipBody->getLinearVelocity();
    btScalar speed = velocity.length();
    if(speed > mMaxSpeed) velocity *= mMaxSpeed/speed;
    mShipBody->setLinearVelocity(velocity);
}
```