Open source physics engines

Building believable worlds with open source

M. Tim Jones
Consultant Engineer
独立作家

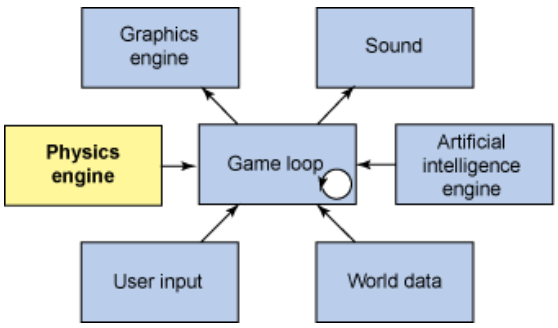
07 July 2011

Graphics give games a visual appeal, but it's the internal physics engine that gives the game's world life. A *physics engine* is a software component that provides a simulation of a physical system. This simulation can include soft- and rigid-body dynamics, fluid dynamics, and collision detection. The open source community has a number of useful physics engines operating in the 2D and 3D domains targeted to games and simulations. This article introduces the use and basics of a physics engine and explores two options that exist: Box2D and Bullet.

A *physics engine* is a simulator used to create a virtual environment that incorporates laws from the physical world. That virtual environment can include objects with accompanying forces applied to them (such as gravity) in addition to interactions between objects, such as collisions. A physics engine simulates Newtonian physics in a simulated environment and manages those forces and interactions.

One of the most advertised applications of a physics engine is in the entertainment and game industry (see Figure 1), where the physics engine provides a real-time simulation of the game environment (including the player and other objects that may be present). Prior to their use in games, physics engines found a number of applications in the scientific domain, from large-scale simulations of celestial bodies, to weather simulations, all the way down to small-scale simulations to visualize the behavior of nanoparticles and their associated forces.

Figure 1. A physics engine in the context of a game application



One key difference between these applications is that although game-focused physics engines focus on real-time approximations, the scientific variety focuses more on precise calculations for increased accuracy. Scientific physics engines can rely on supercomputers for their raw processing capacity, where game physics engines can run on considerably more resource-constrained platforms (such as handheld gaming devices and mobile phones). Game physics engines scale back the simulation by avoiding such things as Brownian motion, which in turn minimizes the processing complexities of the simulation. The range of mathematics and physics concepts built into these engines is outside of the scope of this article, but you can find links to more information in Resources.

Numerous types of game physics exist, depending upon the requirement, though all are variations on the same theme. In games, you can find *ragdoll physics* (which simulate the behavior of a complex articulated system) and *particle systems* (which model the behavior of many small and large particles in response to events such as an explosion). One of the earliest software physics engines was the ENIAC computer, which was used to simulate artillery shells given variables of mass, angle, propulsion, and wind. Wikipedia provides an interesting introduction to this application—see Resources for a link.

Open source options

One of the main uses of physics engines (in particular, the real-time and low-precision variety) is in the development of game run times. Based on the popularity of these software frameworks, there are many open source options to choose from. This article explores some of the available open source physics engines and illustrates their use in simple applications.

Box2D

Box2D is a simple physics engine with a broad use. It was originally designed by Erin Catto as a demonstration engine for a physics presentation given at the Game Developers Conference in 2006. Box2D was originally called *Box2D Lite*, but the engine has been expanded to enhance the API in addition to include continuous collision detection. Box2D is written in `C++`, and its portability is demonstrated by the platforms in which it's used (Adobe® Flash®, Apple iPhone and iPad, Nintendo DS and Wii, and Google Android). Box2D provides the physics behind a number of popular handheld games, including *Angry Birds* and *Crayon Physics Deluxe.*

Box2D provides a rigid-body simulation supporting geometrical shapes like circles or polygons. Box2D can join shapes with joints and even includes joint motors and pulleys. Within Box2D, the engine can apply gravity and friction while managing detection of collisions and the resulting dynamics.

Box2D is defined as a rich API that provides a variety of services. These services permit the definition of a world populated with a number of objects and attributes. With the objects and attributes defined, you next simulate the world in discrete time steps. This sample application (based on Erin Catto's sample application) explores a box hurled into the world with gravity.

Box2D example

Listing 1 illustrates the process of creating a simple world occupied by a box (in the with upward momentum) and a ground plane. You define the world and a gravity vector for the world using the gravity and world functions. The `true` parameter for the world simply says that it's a sleeping body and therefore requires no simulation.

With the world defined, you specify the ground body within that world and its position. The ground is a box that is static, which Box2D knows, because the box has zero mass (by default) and therefore doesn't collide with other objects.

Next, create your dynamic body, which has a position, initial linear velocity, and angle. This setup is similar to the creation of the ground body, except that you define additional attributes for the dynamic body. These attributes include the density of the object and the friction. You add the new dynamic body to the world by creating the fixture with `CreateFixture`.

With your world and its objects defined, you can move on to the simulation. Begin by defining the time step of your simulation (in this case, 60Hz). You also define the number of iterations to run, which determines how many times to iterate over velocity and position calculations (because solving for one modifies others). The more iterations, the more accuracy is achieved— and the more time is spent in the calculations.

Finally, you run the simulation, which involves performing a step in the simulation through a call to the `Step` method for the world. Once the call returns for the current time step, you clear the forces applied to the objects from the last step, and then get the current position and angle of your dynamic body. Those returned variables are emitted to standard output (stdout) for viewing. You continue the simulation until your dynamic body has come to rest (that is, sleeping).

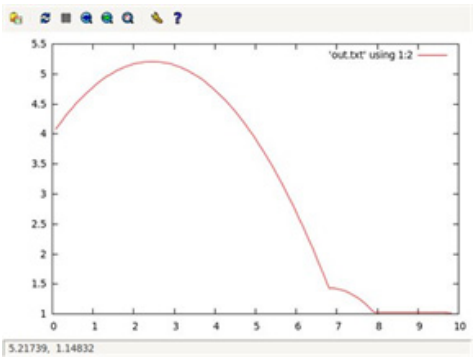
Listing 1. Simple application using Box2D (adapted from Erin Catto's HelloWorld)

```
#include <Box2D/Box2D.h>

#include <cstdio>

int main()
{
 // Define the gravity vector.
 b2Vec2 gravity(0.0f, -10.0f);

 // Construct a world object, which will hold and simulate the rigid bodies.
 // Allow bodies to sleep.
 b2World world(gravity, true);

 // Define the ground body.
 b2BodyDef groundBodyDef;
 groundBodyDef.position.Set(0.0f, -10.0f);
 b2Body* groundBody = world.CreateBody(&groundBodyDef);

 // Define the ground box shape.
 b2PolygonShape groundBox;
 groundBox.SetAsBox(50.0f, 10.0f);

 // Add the ground fixture to the ground body.
 groundBody->CreateFixture(&groundBox, 0.0f);
```

```
// Define the dynamic body. Set its position and call the body factory.
b2BodyDef bodyDef;
bodyDef.type = b2_dynamicBody;
bodyDef.position.Set(0.0f, 4.0f);
bodyDef.linearVelocity.Set(5.0f, 5.0f);
bodyDef.angle = 0.25f * b2_pi;

b2Body* body = world.CreateBody(&bodyDef);

// Define another box shape for your dynamic body.
b2PolygonShape dynamicBox;
dynamicBox.SetAsBox(1.0f, 1.0f);

// Define the dynamic body fixture.
b2FixtureDef fixtureDef;
fixtureDef.shape = &dynamicBox;
fixtureDef.density = 1.0f;
fixtureDef.friction = 0.3f;

// Add the shape to the body.
body->CreateFixture(&fixtureDef);

float32 timeStep = 1.0f / 60.0f;
int32 velocityIterations = 6;
int32 positionIterations = 2;

do {
 world.Step(timeStep, velocityIterations, positionIterations);

 world.ClearForces();

 b2Vec2 position = body->GetPosition();
 float32 angle = body->GetAngle();

 printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle);

} while (body->IsAwake());

 return 0;
}
```

Box2D is meant to be agnostic of the renderer (graphical visualization). A simple rendering of the box's position (from Listing 1) is shown in Figure 2. Note that behavior of the box's position as gravity pulls it to the ground and it comes to rest and the collision.

Figure 2. Simple rendering of the box position from Listing 1



Bullet

Bullet is a 3D open source physics engine that supports rigid- and soft-body dynamics and collision detection in 3D. Bullet was developed by Erwin Coumans while he was at Sony Computer Entertainment. The engine is supported on a large number of platforms, such as Sony Playstation 3, Xbox 360®, iPhone, and Wii. It includes operating system support for Windows®, Linux®, and Mac OS, as well as a number of optimizations targeted for the Cell Synergistic Processing Unit in Playstation 3 and the OpenCL framework on the PC.

Bullet is a production physics engine that has wide support both in games and in movies. Some of the games that have used Bullet include Rockstar's *Red Dead Redemption* and Sony's *Free Realms* (MMORPG). Bullet has also been used for special effects in a number of commercial films, including "The A-team" (Weta Digital) and "Shrek 4" (DreamWorks).

Bullet includes rigid-body simulation with both discrete and continuous collision detection, including support for soft bodies (such as cloth or other deformable objects). As a production engine, Bullet includes a rich API and SDK.

Bullet example

The Bullet example shown in Listing 2 is the "Hello World" program from the Bullet distribution. It implements a similar simulation as that demonstrated with the Box2D example (but in this case, instead of a box, a sphere is used as the falling object). As you would expect, this implementation is quite a bit more complex than the prior example because of the increased richness and variety of the API.

This sample application is split into three segments: setup, simulation, and cleanup. The setup phase creates the world that the simulation phase works with. The cleanup phase simply deallocates the various objects in the world.

To create the world, you need the definition of a broad-phase algorithm (an optimization for identifying objects that should not collide), a collision configuration, and a constraint solver (which incorporates gravity and other forces as well as collisions and defines how objects interact). You also define gravity as the y-axis through a call to `setGravity`. With elements these defined, you create your world. The next two segments in the setup phase define the static ground body and the dynamic sphere body.

The simulation is performed through a call to method `stepSimulation`. This method defines an interval of 60Hz and simulates the physics behind the sphere falling to the ground under the influence of gravity. After each simulation step, the sphere's height (the `y` parameter) is emitted. Looping through the simulation allows the sphere to collide with the ground, and then come to rest.

The final phase is simply cleanup, which frees the objects and other elements from memory.

As shown, although there's a considerable amount of setup required for the simulation, once you've defined the environment of the simulation, the engine does all of the heavy lifting behind the scenes for you. Bullet includes a massive API, permitting fine-tuning of the environment and its

behavior as well as a large number of callbacks for events that occur within the simulation (such as collision and overlap).

Listing 2. Simple falling sphere simulation using Bullet

```
#include <iostream>

#include <btBulletDynamicsCommon.h>

int main (void)
{

  // Setup

  btBroadphaseInterface* broadphase = new btDbvtBroadphase();

  btDefaultCollisionConfiguration* collisionConfiguration =
     new btDefaultCollisionConfiguration();
  btCollisionDispatcher* dispatcher = new btCollisionDispatcher(collisionConfiguration);

  btSequentialImpulseConstraintSolver* solver = new btSequentialImpulseConstraintSolver;

  btDiscreteDynamicsWorld* dynamicsWorld = new btDiscreteDynamicsWorld(
    dispatcher,broadphase,solver,collisionConfiguration);

  dynamicsWorld->setGravity(btVector3(0,-10,0));


  btCollisionShape* groundShape = new btStaticPlaneShape(btVector3(0,1,0),1);

  btCollisionShape* fallShape = new btSphereShape(1);


  btDefaultMotionState* groundMotionState = new
 btDefaultMotionState(btTransform(btQuaternion(0,0,0,1),btVector3(0,-1,0)));
  btRigidBody::btRigidBodyConstructionInfo
               groundRigidBodyCI(0,groundMotionState,groundShape,btVector3(0,0,0));
  btRigidBody* groundRigidBody = new btRigidBody(groundRigidBodyCI);
  dynamicsWorld->addRigidBody(groundRigidBody);


  btDefaultMotionState* fallMotionState =
       new btDefaultMotionState(btTransform(btQuaternion(0,0,0,1),btVector3(0,50,0)));
  btScalar mass = 1;
  btVector3 fallInertia(0,0,0);
  fallShape->calculateLocalInertia(mass,fallInertia);
  btRigidBody::btRigidBodyConstructionInfo
  fallRigidBodyCI(mass,fallMotionState,fallShape,fallInertia);
  btRigidBody* fallRigidBody = new btRigidBody(fallRigidBodyCI);
  dynamicsWorld->addRigidBody(fallRigidBody);

  // Simulation

  for (int i=0 ; i<300 ; i++) {
    dynamicsWorld->stepSimulation(1/60.f,10);

    btTransform trans;
    fallRigidBody->getMotionState()->getWorldTransform(trans);

    std::cout << "sphere height: " << trans.getOrigin().getY() << std::endl;
  }

  // Cleanup

  dynamicsWorld->removeRigidBody(fallRigidBody);
  delete fallRigidBody->getMotionState();
```

```
  delete fallRigidBody;

  dynamicsWorld->removeRigidBody(groundRigidBody);
  delete groundRigidBody->getMotionState();
  delete groundRigidBody;

  delete fallShape;

  delete groundShape;

  delete dynamicsWorld;
  delete solver;
  delete collisionConfiguration;
  delete dispatcher;
  delete broadphase;

  return 0;
}
```

Open source physics engines list

Box2D and Bullet are two examples of useful and widely used physics engines. But there are many other examples that focus on different aspects of physics simulation (performance or accuracy) in addition to using many different licenses. Box2D and Bullet both use the Zlib license (which supports their use in commercial applications). Table 1 provides a list of some of the more common open source physics engines along with the licenses that they use. In addition, although most of the engines support `c++` or `c`, many also support bindings to other languages, such as Ruby or Python.

Table 1. Open source physics engines

Engine	Type	License
Box2D	2D	Zlib
Bullet	3D	Zlib
Chipmunk	2D	Massachusetts Institute of Technology (MIT)
Chrono::Engine	3D	GPLv3
DynaMo	3D	GPL
Moby (Physsim)	3D	GPLv2
Newton Game Dynamics	3D	Zlib
Open Dynamics Engine	3D	BSD
Open Physics Abstraction Layer	N/A	BSD/LGPL
OpenTissue	3D	Zlib
Physics Abstraction Layer (PAL)	N/A	BSD
Tokamak	3D	BSD/Zlib

Chipmunk, developed by Scott Lembcke based on Box2D, includes several features for 2D physics, including direct support for `c` as well as an Objective-Chipmunk to support iPhone bindings. Other bindings include Ruby, Python, and Haskell.

Tokamak is a 3D physics engine SDK written in `c++` by David Lam. It includes a number of optimizations that minimize memory bandwidth and therefore make it ideal for smaller, portable devices. One interesting feature of Tokamak is support for model breakage, where composite objects can break on collision, then creating multiple objects in the simulation.

Although listed under physics engines, the abstraction layers provide an interesting capability that should not be ignored. The PAL provides a unified interface over multiple physics engines within a single application, which allows a developer to easily use the right physics engine for the particular application without the porting effort. PAL's plug-in architecture supports several leading open source physics engines, such as Box2D, Bullet, Newton Game Dynamics, OpenTissue, Tokamak, and numerous others. It also supports commercial physics engines like Havok, which is popular in games development. The downside of PAL is that it can restrict functionality offered by a particular physics engine, because its focus is on a common abstraction.

Hardware acceleration

Hardware acceleration for physics has been evolving over the past few years, following a trend in graphics processing units (GPU). A *GPU* is a hardware coprocessor that accelerates computations for computer graphics applications. GPUs have evolved into general-purpose computing on graphics processing units (GPGPU), permitting them to be used in more general-purpose acceleration tasks. A movement for a physics processing unit (PPU) to accelerate physics engine computations has potentially been diverted by the use of more accessible GPGPUs. Examples of GPGPUs include ATI's Stream technology and NVIDIA's Common Unified Device Architecture (CUDA) architecture.

Going further

Physics engines free you from having to develop the complex software to implement physics and collision detection in software. Instead, you can invest your time in your particular application (game or simulation). Although it helps to understand the math behind these engines, it's not necessary to use and enjoy them. The Resources section includes links to a number of open source physics engines that are easily usable on both Linux and Windows systems. Each includes sample illustrative demos to help you understand their APIs and concepts so that you can bring physics into your application.

Resources

Learn

- Wikipedia: Physics engines: Learn about how physics engines are being segregated into classes of engines for their particular problem domain. Wikipedia is a great resource for learning more about physics engines, ragdoll physics, particle systems, and computer simulation. You can also learn about the physics engine used in Second Life in addition to some of the trade-offs for the virtual environment.
- Rigid Body Dynamics: Read this series by Chris Hecker if you're interested in the nuts and bolts within a physics engine. This series appeared first in Game Developer Magazine.
- John Van Der Burg's Building an Advanced Particle System at Gamasutra: Learn more about the development of a particle system. Particle systems are an interesting application of a physics engine, and particles are commonly a difficult problem based on the scale required.
- Real-Time Collision Detection: To learn more about collision detection, check out Christer Ericson's book, which covers the details of building believable simulations.
- developerWorks on-demand demos: Watch video demos ranging from product installation and setup demos for beginners to advanced functionality for experienced developers.
- Open source zone: Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- developerWorks technical events and Webcasts: Stay current with the latest technology.
- developerWorks podcasts: Listen to interesting interviews and discussions for software developers.

Get products and technologies

- A number of physics engines focused on 2D can be found in the open source domain. Options include Box2D from Erin Catto (available under the Zlib license) as well as Chipmunk from Scott Lembcke (available under the MIT license). To understand their APIs, check out the Box2D documentation and Chipmunk documentation.
- A large number of physics engines focused on 3D are available as open source. Options include DynaMo, OpenTissue, Chrono::Engine, and Tokamak. You can also take advantage of an abstraction layer (such as the PAL) to support more than one physics engine without a porting effort.
- Hardware acceleration is a growing area of research and development. From GPUs to GPGPUs (and PPUs), there's a considerable amount of effort to increase both the accuracy and speed of physics simulations. You can find commercial acceleration in NVIDIA's PhysX, CUDA, and AMD/ATI's Stream technology.
- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement service-oriented architecture efficiently.

Discuss

- Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

M. Tim Jones

M. Tim Jones is an embedded firmware architect and the author of Artificial Intelligence: A Systems Approach, GNU/Linux Application Programming (now in its second edition), AI Application Programming (in its second edition), and BSD Sockets Programming from a Multilanguage Perspective. His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a platform architect with Intel and author in Longmont, Colo.