Code Reading Report V1

2015011308 唐适之

1 Building

The official *LiquidFun Build and Run Instructinos* gives a specific building guide on different platforms. Here is a brief repeat on how to build it on a Linux, as well as something that is not mentioned in the official guide.

1.1 Dependencies

The official guide gives 3 minimum dependencies:

- OpenGL: libglapi-mesa 8.0.4
- GLU: libglu1-mesa-dev 8.0.4
- cmake: 2.8.12.1

In a Debian origined Linux, like Ubuntu, they can be installed as below:

- sudo apt-get install cmake
- sudo apt-get install libglapi-mesa
- sudo apt-get install libglu1-mesa-dev

There might be two missing dependencies: *X11 client-side library (Xlib)*, which provides an API to the basic X Window System, and *X11 Input extension library (libXi)*, which provides an API to the XINPUT extension to the X protocol. In a Ubuntu system, it can be installed via *sudo apt-get install libx11-dev libxi-dev*. If compiling without them, the compiler will report that it cannot find *X11/Xlib.h* or *X11/extensions/XInput2.h* especially.

1.2 Compiling

As described in the guide, we use *cmake* to compile the project as below.

- *cd liquidfun/Box2D* # switch to the corresponding directory
- *cmake -G'Unix Makefiles'* # generate Makefile using cmake

- *make*

In the ideal case, it should have been done, but there is a known issue in the CMakeLists.txt, which is used by cmake, even in the stable version. The CMakeLists.txt should have load the *Thread* package to load a multithread library for the corresponding platform, but the loading instruction is simply missing for some platforms however. In this situation, try adding *find_package(Threads)* in the CMakeLists.txt. This rough patch may not work for every platform because the instruction might not needed on some platform, but it should resolve the issue when the problem truly occurs.

1.3 Run to test

Under a full building, to determine whether we have complete a successful building, execute

./liquidfun/Box2D/Testbed/Release/Testbed

to run a demo, or execute

./liquidfun/Box2D/Unittests/run_tests.sh

to run unit tests.

2 Usage

LiquidFun is a library to calculate 2D rigid body and liquid physics, extended from *Box2D*. It just does the math, but does not include the displaying function. We have to implement our own programs that makes use of *LiquidFun*.

2.1 Using the Framework and Linking the Libraries

Under a full building, the following parts will be built to their respective directives. (On a Linux platform)

- *Box2D*. It's *LiquidFun* itself, the core library. Modules below are not necessarily part of *LiquidFun*.

- *HelloWorld*. A minimum demo consisting no GUI, just displying the calculated digits.

- *freeglut* and *glui*. They are APIs to access OpenGL (a 3D graph library) easily, providing a basic UI library. They help to build a program that can display the result on screen as it is.

- *Testbed*. It's a demo or a UI program built to display the result, making use of *freeglut* and *glui*, so when we are working with *LiquidFun*, it's no need to implement the display program by ourselves, even we have *freeglut* or *glui*. As the name indicates, *Testbed* can also help do some debugging, such as printing debug info or doing step-by-step executing.

- *googletest*. It's a framework that help building unit tests.

- *Unittests*. Unit tests for *LiquidFun*.

If we tend to ignore the GUI, or to implement the UI by ourselves, we only need to include the headers and link the libraries in *Box2D* directive, Or we can put our code in the *Testbed* and compile it together with the *Testbed*.

2.2 Run the Physics Wolrd

To make a brief explaination, there are roughly three steps to run the *LiquidFun*:

- Create a World. A *b2World* object handles all infrastructural work, including memory allocating and objects management, so we should create a *b2World* object first before adding objects to it.
- Adding Objects. Several objects shold be added to *b2World*, including *b2Body*(s), which act as priticles, *b2Fixture*(s), which give *b2Body* shape and physical parameters, and other additional objects.
- Call *b2World::Step* to simulate the physics step by step with a discret time interval. Between each simulation, we can get access to the objects to get their parameters or make operations on them.

More details can be referred to the document and is no need to be repeated here.

3 Basic OOP Design

3.1 Polymorphism

Polymorphism doesn't play a main role in *LiquidFun*, but there are still a few. Polymorphism is used to handle different types of shapes, contacts between shapes and joints. We have similar queries and operations to these objects, for example, getting the center of gravity of a circle, a polygon or an edge. Therefore, there is a base class *b2Shape*, which has subclasses *b2ChainShap*, *b2CircleShape*, *b2EdgeShape*, and *b2PolygonShape*.

Templates can be used to help the base classes get to know the infomation about the subclasses. For example, *b2TypedIntrusiveListNode<T>* is a node of an intrusive double linked list. The subclass *b2ParticleHandle* extends *b2TypedIntrusiveListNode<b2ParticleHandle>*, while subclass *b2TrackedBlock* extends *b2TypedIntrusiveListNode<b2TrackedBlock>*. So that the base class can determine which subclass extends it, from the template parameter, during compile time and make corresponding instructions. In this case of a node of a linked list, the compiler can allocate the data variable of each node of the corresponding type.

However, there is also what can't be done by C++'s polymorphis mechanism. Virtual functinos can handle most of the polymorphism, but when we query the size of a subclass during memory management, *LiquidFun* have to use a *switch* statement and do it manually.

3.2 Listener Classes

End users may want to make queries or operations during a step simulation, and therefore *LiquidFun* provides an interface: listeners. End users can implement a listener class which inherits from the given base class, and then register it to *LiquidFun*. *LiquidFun* will call back the listener in a certain situations.

For example, a Contact Listener aims to get call-back during different stages when *LiquidFun* is processing collisions. *LiquidFun* provides a class *b2ContactListener* as a base class. *b2ContactListener* provides member functions like *BeginContact* or *PreSolve*, the first of which will be called when a contact has first been detected, while the second will be called before a contact to be solved. Calling the base class will result in nothing. A User should implement his own class which inherits from *b2ContactListener*, and override those member functions to do what he wants. To make *LiquidFun* know about the listener objects, the user should register it to *LiquidFun* using *b2World::SetContactListener*. The *b2World* object will therefore refer to it with a *b2ContactListener** pointer.

C++11's lambda expression can also play the role as a listener, but defining a listener base class to be inherited from, like what *LiquidFun* do, may make the code more structualized and reduce errors.

3.3 Factories

Classes like *b2Body*, *b2Fixture* are created by factory functions but not directly by constructors. Because these classes should bind to a parent class to play its role, their factory class is just their parent class. For example, class *b2Body* has a member function

b2Fixture CreateFixture(const b2FixtureDef *def)*

to create a *b2Fixture* object and bind the newly created object to it.

Because *LiquidFun* will place a big amount of reletively small objects into the heap memory, which will reduce the performance, *LiquidFun* implements its own heap memory allocator. Creating objects in the factory functions make calling the allocator organized, and therefore the end user don't have to be cencerned about the allocator.

Also because of the allocator, these classes seperates the initializing and cleaning procedures from constructors and destructors, but defines functions like *b2Fixture::Create* or *b2Fixture::Destroy*, for the reason that we can pass the allocator to a normal member function but not a destructor.

3.4 Definition Classes

Creating an object like *b2Body* or *b2Fixture* may require a great amount of parameters, which make passing them one by one as parameters of creater functions (not sufficiently constructors here) impossible. *LiquidFun* introduces definition classes, for instance, *b2BodyDef* and *b2FixtureDef*. Users construct a definition object first before passing it into the creater function.

The creater functions won't keep references to the definition objects, and therefore the definition objects can be reused. Comparing to creating an empty object first, for example, an empty *b2Body*, and then use the setter to set the parameters one by one, users can take advantage of the definition object reusing. One can stores a definition class and apply it to each creating procedures after minor modifications. Notice that *b2Body* can not even be copied.

4 Specific Coding Skills

4.1 *B2_NOT_USED*

B2_NOT_USED is a macro defined as below:

#define B2_NOT_USED(x) ((void)(x))

Suppose there is a variable or a parameter *var* that won't be used, "*B2_NOT_USED(var);*" is called in *LiquidFun*. This will prevent the compiler to give the warning that *var* is not used. This statement has no side effect to the instructions and is safe to use. Converting *x* to *void* prevents the macro to be called for other purpose by mistake.

4.2 *default: b2Assert(false);*

In *switch* branch statement, *LiquidFun* handles *default* case as below:

default: b2Assert(false); break;

b2Assert above is a specialized *assert* statement defined in *LiquidFun*. This can prevent encountering situations that are not expected by the *switch*.

4.3 Destruction Listener

One of *LiquidFun*'s listeners (described above) is Destruction Listener: *b2DestructionListener*. End users can register destruction listeners. When an object is destroied by *LiquidFun*, *LiquidFun* will call the corresponding listener, and users can therefore do some cleaning procedures, such as nullify the pointers. This prevent accessing to an invalid object.