Code Reading Report

Liquid Fun

翁家翌 2016011446

2017.5

目录

1 简介 **2**
1.1 目的 2
1.2 描述 2
1.3 相关材料 2

2 系统概览 **2**

3 Linux 上的编译运行 **3**

4 系统结构 **5**
4.1 测试台结构 5
4.2 物体结构 5
4.3 世界结构 6

5 数据结构 **7**
5.1 b2ParticleDef 数据列表 7
5.2 b2BodyDef 数据列表 7

6 用户界面设计 **9**
6.1 概况 9
6.2 一些操作 9
6.3 屏幕截图 10

7 编程技巧 **12**

1 简介

1.1 目的

这份文档简略描述了软件 *Liquid Fun* 的结构和系统设计。

1.2 描述

Liquid Fun 是由 Google 开发的，基于 Box2D 进行刚体和流体模拟的 C++ 库。它对物理结构提供可视化支持，使其能够实时进行移动和交互。

1.3 相关材料

http://google.github.io/liquidfun/
https://github.com/google/liquidfun
http://www.iforce2d.net/b2dtut/
Inside LiquidFun

2 系统概览

Liquid Fun 基于牛顿三大定律，使用积分器得出物体的位置与速度，对二维刚体进行运动状态的模拟，包括物体与物体之间的相互关系和状态数据（如摩擦）。主要使用的算法有碰撞检测、AABB 等。在一个时间周期内，求解器会进行对力和力矩的若干次迭代计算。它支持模拟许多个物体同时运动时候的状态、支持修改重力、支持自定义位置、阻尼、形状、摩擦、密度等等参数来满足设计者的需要。

Liquid Fun 有若干个基本的类型，分别是:

- 物体（Body）：存储一些特性，以下章节会对其进行描述
- 定制器（Fixture）：设定物体形状、密度、摩擦等一系列参数
- 联合体（Joint）：设定物体与物体相互之间的一些约束
- 联系体（Contact）：设定物体与物体相互之间的一些作用，比如力

Box2D 只能处理刚体。然而在 *Liquid Fun* 中，粒子模拟被扩展到能够模拟液体、砂石和可形变的物体。粒子是一种圆型小颗粒，当它们聚集在一起时，能够通过不同的方法呈现出刚体、弹性体、粉末、拉伸体（具有表面张力）、粘稠体、混色体……

为了让输出更直观，*Liquid Fun* 还实现了测试台功能。测试台是一个单元测试框架和演示环境，它支持视角的平移和缩放、用鼠标操作动态体、可扩展测试集、设置用于选择测试，参数调整和调试绘图选项的 GUI、暂停和单步模拟、文字呈现。

3 Linux 上的编译运行

```
sudo apt install cmake libglapi-mesa libglu1-mesa-dev
git clone https://github.com/google/liquidfun.git
cd liquidfun/liquidfun/Box2D/
cmake .
```

这个时候有可能会报错:

```
CMake Error at CMakeLists.txt:158 (set_target_properties):
  set_target_properties Can not find target to add properties
      to:
  Threads::Threads
```

解决方案则是在该文件夹下的 CMakeList.txt 中，在"project(Box2D)"的下一行添加

```
find_package(Threads)
```

之后运行

```
make -j4
```

会依次得到

```
[ 24%] Built target Box2D
[ 39%] Built target glui
[ 42%] Built target gtest
[ 46%] Built target HelloWorld
[ 52%] Built target gtest_main
[ 65%] Built target SlabAllocatorTests
[ 77%] Built target ColorTests
[ 83%] Built target freeglut
[ 85%] Built target freeglut_static
[ 87%] Built target BlockAllocatorTests
[ 87%] Built target CallbackTests
[ 89%] Built target CommonTests
[ 89%] Built target FunctionTests
[ 90%] Built target ConfinementTests
```

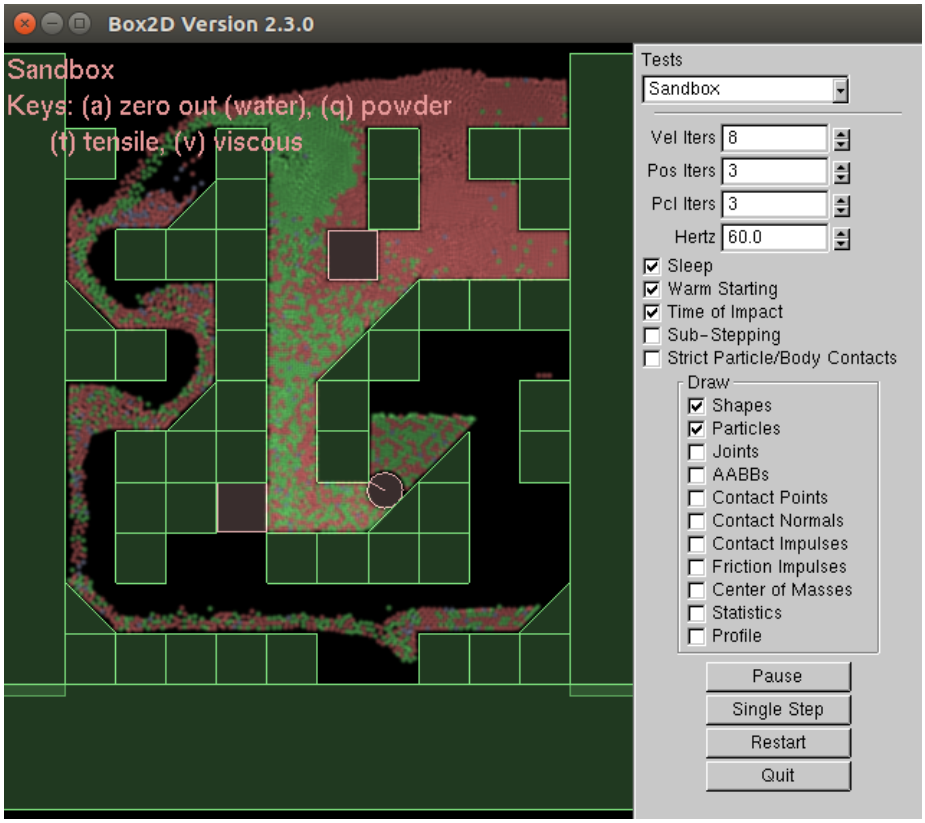
```
[ 92%] Built target IntrusiveListTests
[ 94%] Built target BodyContactsTests
[ 97%] Built target FreeListTests
[100%] Built target Testbed
[100%] Built target HelloWorldTests
[100%] Built target ConservationTests
```

大部分生成的可执行文件位于"./Unittests/Release" 下，用来测试运行时间。

可视化程序位于"./Testbed/Release" 下，运行命令

```
cd Testbed/Release/
./Testbed
```

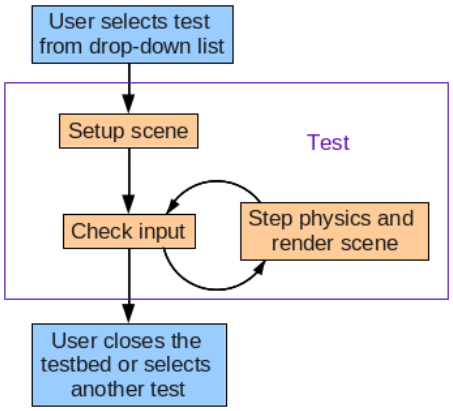
即可看到可视化之后的效果，一张效果图如下：



4 系统结构

4.1 测试台结构

Testbed 框架可以高效地添加新测试。当定义一个新的测试时，对于所有测试（如创建/销毁/重置）以及控制面板复选框和按钮都相同的功能由主程序代码处理，均不需要更改。一个测试的生命周期大致如下图所示：



图中橙色的部分是我们将添加或者更改代码以进行自己的测试。具体而言，我们将创建一个 Test 类的子类 myTest，它处理测试的所有常见特征，并覆盖我们的子类中的一些函数。Test 类中的一些虚函数如下所示：

```
class Test {
    virtual void Step(Settings* settings);
    virtual void Keyboard(unsigned char key);
    virtual void MouseDown(const b2Vec2& p);
    virtual void MouseUp(const b2Vec2& p);
}
```

第一个函数 Step() 实现了物理引擎中的单步迭代计算以及场景渲染。其它三个函数 Keyboard()，MouseDown() 和 MouseUp() 允许我们获取用户当前的操作信息。这里没有覆盖设置场景的方法，因为这些操作已经在子类的构造函数中完成。

4.2 物体结构

可以把物体想象成是一种看不见摸不着的实物的属性，大致有：

- 质量 (mass)：实物到底有多重
- 速度 (velocity)：某方向上实物到底运动多快

- 转动惯量 (rotational inertia): 开始或停止转动需要多大的力
- 角速度 (angular velocity): 某方向实物转动的速度有多快
- 位置 (location): 实物在哪
- 角度 (angle): 实物面向哪个方向

有三种类型的物体: 静态物体 (static), 动态物体 (dynamic) 以及运动学物体 (kinematic)。前两个顾名思义看起来更好理解一些, 最后一个看起来并不是那么直观。

在 myTest 类 (继承自 Test 类) 的构造函数里, 添加以下代码即可创建物体:

```
b2BodyDef myBodyDef;
myBodyDef.type = b2_dynamicBody; //this will be a dynamic body
myBodyDef.position.Set(0, 23); //set the starting position
myBodyDef.angle = 0; //set the starting angle
```

4.3 世界结构

世界 (Worlds) 是 Box2D 世界里主要的实体。当创建或者删除物体的时候, 可以调用 Worlds 里的函数来完成这些功能, 所以世界也管理着所有对象的空间分配。世界的主要功能有:

- 定义重力加速度
- 调用物理模拟
- 发现定制器的作用域
- 切断射线并找到相交的定制器

世界的创建像其它普通类型一样, 在构造函数里进行基本的设置。

```
b2Vec2 gravity(0, -9.8); //normal earth gravity, 9.8 m/s^2
    straight down
b2World* myWorld = new b2World(gravity);
```

当然可以把-9.8 改为 0, 这样的话就会生成出一个零重力的环境。像这样:

```
myWorld->SetGravity( b2Vec2(0,0) );
```

一旦像上面那样创建了一个世界, 就可以像我们之前做的那样往世界里添加物体。为了让好玩的事情发生, 我们需要不停的调用 Step() 函数来模拟物理世界的运动。就像下面这样:

```
float32 timeStep = 1/20.0;  //the length of time passed to
    simulate (seconds)
int32 velocityIterations = 8;  //how strongly to correct
    velocity
int32 positionIterations = 3; //how strongly to correct
    position
myWorld->Step( timeStep, velocityIterations, position
    iterations);
```

当世界对象完成了所有工作的时候，就可以将其简单的删除：

```
delete myWorld;
```

当世界被删除之后，它会将它所关联的所有连接器和物体都删除掉。

5 数据结构

Liquid Fun 中最为重要的数据结构便是针对物体的数据结构 b2BodyDef 和针对粒子的数据结构 b2ParticleDef，以下篇幅将对其进行描述。

5.1 b2ParticleDef 数据列表

以下是 b2ParticleDef 的类型定义[1]

```
struct b2ParticleDef {
        uint32 flags; //18种粒子的子类型可组合，比如
        // pd.flags = b2_elasticParticle | b2_viscousParticle
        b2Vec2 position; //粒子的方向
        b2Vec2 velocity; //粒子的线速度
        b2ParticleColor color; //粒子颜色
        float32 lifetime; //粒子的生命周期，如果该值小于等于0，
            那么就意味着生命周期无限长
        void* userData; //存储用户对于该粒子的附加数据
        b2ParticleGroup* group; //这个粒子属于哪一个群
};
```

5.2 b2BodyDef 数据列表

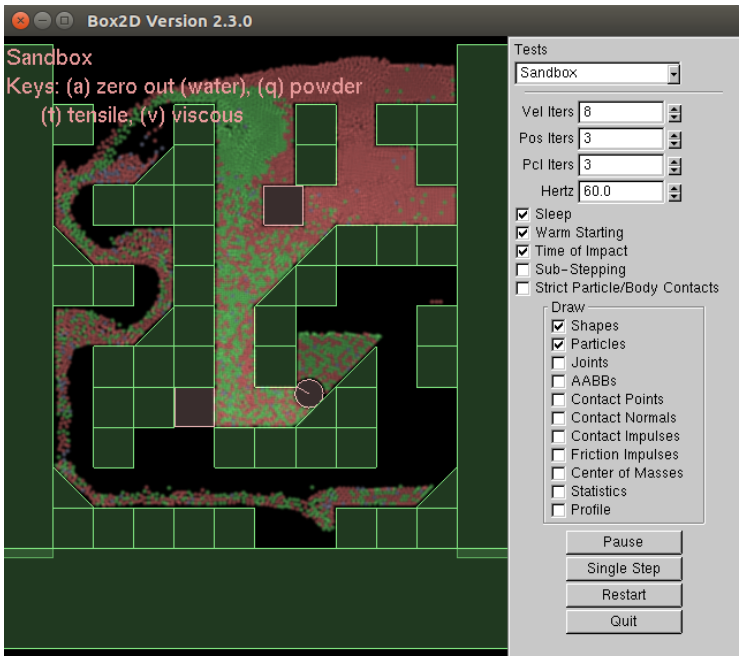
以下是 b2BodyDef 的类型定义[2]

[1]摘自文件"b2Particle.h"
[2]摘自文件"b2Body.h"

```
struct b2BodyDef {
        b2BodyType type; //三种类型，分别为b2_staticBody，
            b2_kinematicBody和b2_dynamicBody。如果一个动态物体
            质量为0，那么程序在运行过程中会将其当作质量为1进行
            处理
        b2Vec2 position; //物体质心在这个虚拟世界中的位置。尽量
            避免在原点创建物体，因为默认构造函数就是设置为原
            点，由此可能会造成许多物体的相互覆盖
        float32 angle; //物体相对于初始状态逆时针旋转的角度
        b2Vec2 linearVelocity; //质心的线速度矢量
        float32 angularVelocity; //质心的角速度
        float32 linearDamping; //线性阻尼，用于降低线速度。如果
            该数值大于1.0f，那么阻尼影响将会对时间模拟间隔敏感
        float32 angularDamping; //角阻尼，用于降低角速度。如果
            该数值大于1.0f，那么阻尼影响将会对时间模拟间隔敏感
        bool allowSleep; //FALSE:在每个时刻都进行对该物体的模拟
            计算。这将会增大CPU的开销。反之则有利于提高运行效率
        bool awake; //在当前时刻下，该物体是否被唤醒
        bool fixedRotation; //TRUE:物体将不能够进行旋转操作
        bool bullet; //在物体快速运动的情况下，有可能会产生隧道
            效应。该选项只针对于动态体，并且开启该选项则意味着
            需要增加计算时间以避免产生隧道效应
        bool active; //物体是否被激活（是否需要对某些身边的事件
            进行响应，从而进行相应的模拟计算）
        void* userData; //存储用户对于该物体的附加数据
        float32 gravityScale; //作用于该物体的重力的影响因子。
};
```

6 用户界面设计

6.1 概况



如图所示，界面分为两个部分，左部为演示区，右部为功能区。自上到下依次为：

- Test：不同的测试类别，详见截图部分
- iters：设置迭代次数，数值越大，模拟越精确，相应计算时间越长
- Hertz：计算频率，默认 60Hz，即一秒内总共计算 60 次所有物体的状态。数值越大，模拟越精确，相应的计算时间越长
- 5 个计算开关：调整计算策略
- 11 个显示开关：在左侧屏幕中显示相关的信息
- 4 个按钮：浅显易懂就解释了

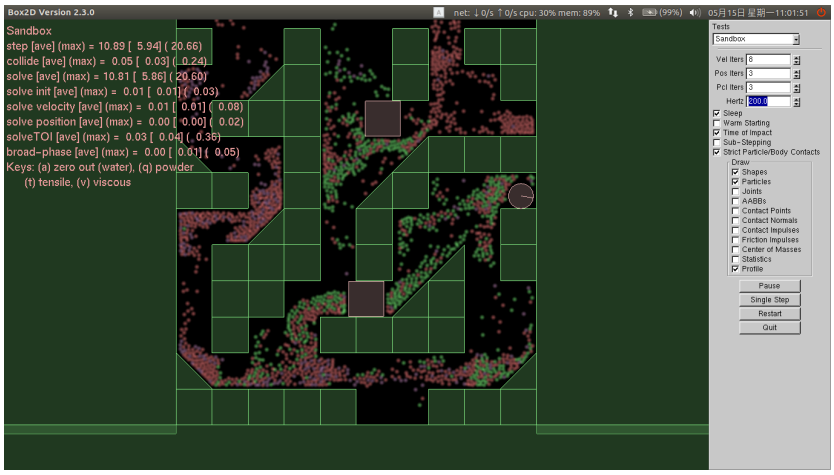
6.2 一些操作

- r：回到初始状态进行模拟
- p：暂停，但是对于生成粒子的粒子源无效
- 上下左右键/鼠标右键拖动：调整左侧屏幕的显示位置

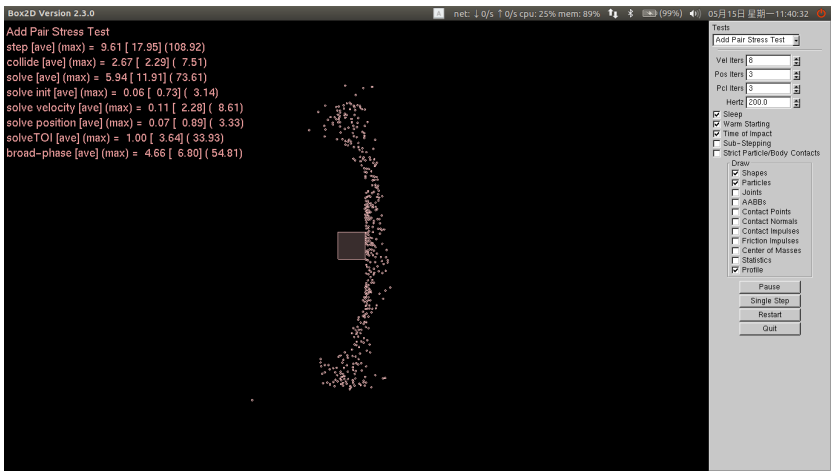
- 空格：随机在屏幕中的某个位置生成一个半径很小的圆，并具有一定的初速度
- "[" 和"]"：切换各个 Test
- 鼠标左键点击：点击并拖动物体，表现为施加一个作用力；在非物体处（如一群小颗粒）点击，则会达到让它们停止运动的效果

6.3 屏幕截图

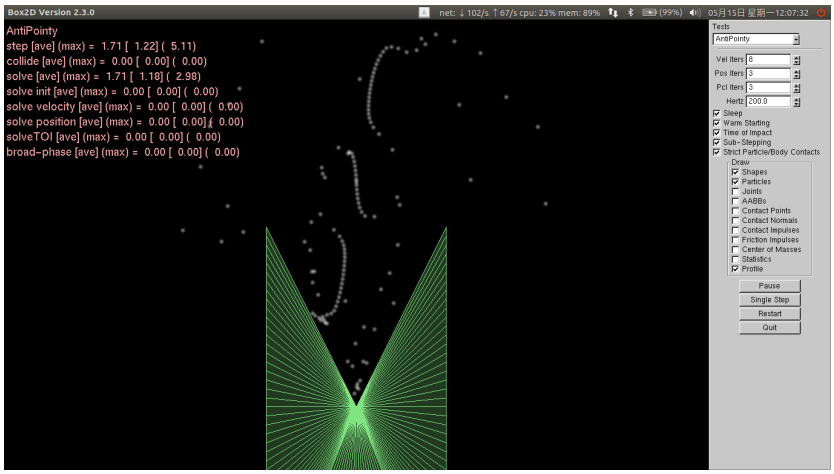
以下是 Testbed 中一些实例的屏幕截图：



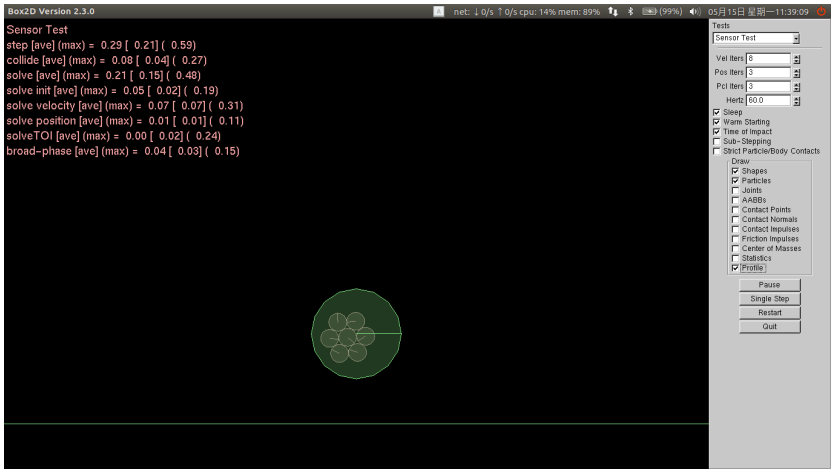
Sandbox 模拟重力环境下小颗粒的运动轨迹



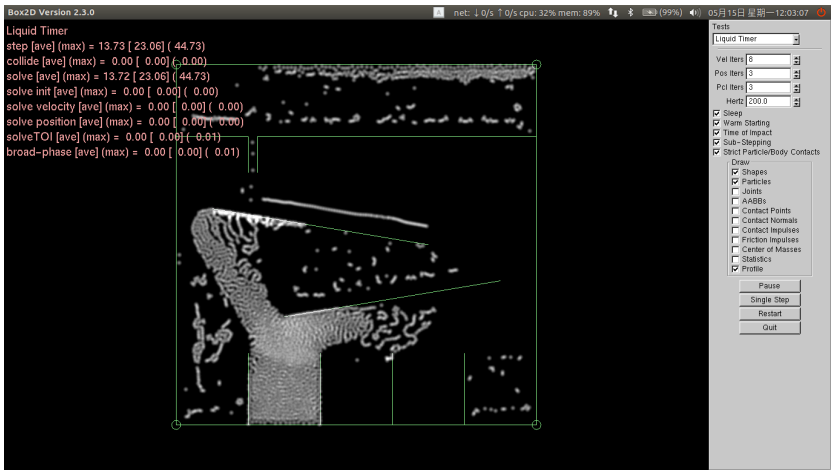
Stress Test 模拟压力测试



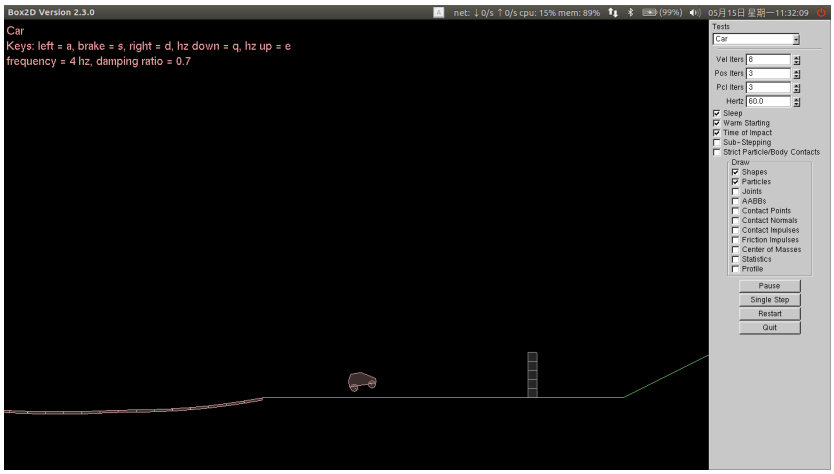
Pointy 模拟尖端现象



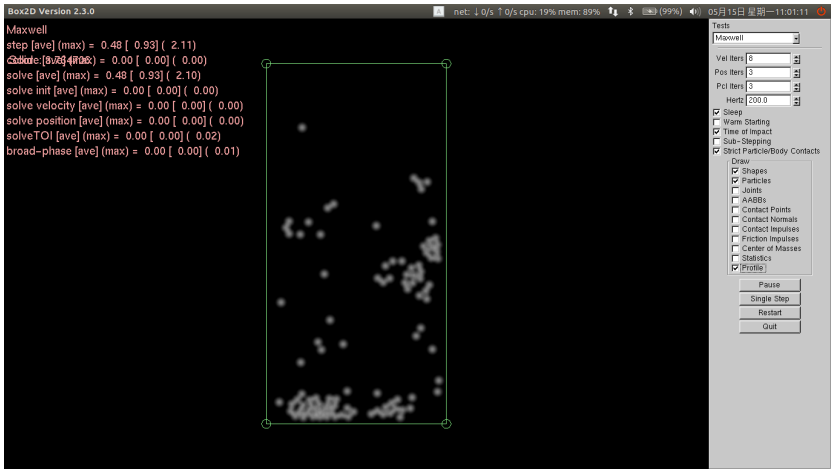
Sensor 在重力和新的引力源叠加下模拟小球的运动



Timer 一个计时器，类似沙漏



Car 开车游戏



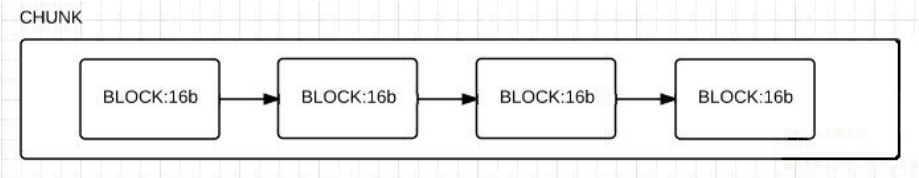
Maxwell 模拟分子无序运动

7 编程技巧

以 *Liquid Fun* 下 Box2D 中的小型对象分配器[3](b2BlockAllocator)为例。在某些场合下，我们会使用到非常小的对象（甚至可能只有几个 byte），其生命周期可能也很短。如果每次都通过 malloc 或者 new 在堆上分配内存，用后销毁，效率太低。于是维护一些不定尺寸并可扩展的内存池，当小对象需要内存分配时，直接从早已准备好的内存池上返回一块大小合适的内存。注意，当内存使用完毕后，我们需要返回其内存到内存池中，而不是销毁它。这样做能够以定程度上提高程序运行效率。

首先对 Box2D 中 b2Chunk（大块内存）和 b2Block（区块）进行说明

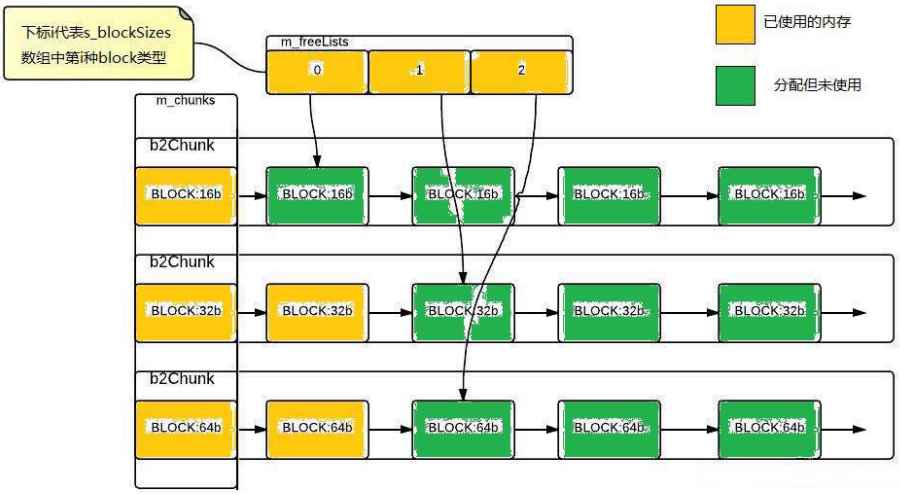
[3]http://www.codeproject.com/useritems/Small_Block_Allocator.asp



如图所示，一个由多个 16byte 大小 block 组成的 chunk，实际上就是一个链表。chunk 是一个由 n 个 block 组成的固定大小的大块内存，多个大块内存形成内存池。以下代码来自 b2BlockAllocator.h:

```
const int32 b2_chunkSize = 16 * 1024; //一个chunk大小固定为16k
const int32 b2_maxBlockSize = 640; //block最大的大小640b
const int32 b2_blockSizes = 14; //block大小的种类，14种
const int32 b2_chunkArrayIncrement = 128; //所有chunk用完时每次增加的chunk数量
struct b2Block; //block结构，即链表节点结构
struct b2Chunk; //chunk结构
class b2BlockAllocator {
public:
        b2BlockAllocator(); //构造函数,对私有成员变量进行初始化
        ~b2BlockAllocator(); //析构函数,销毁内存
        void* Allocate(int32 size); //在内存池中分配内存
        void Free(void* p, int32 size); //只须返回内存到内存池即可
        void Clear(); //清空内存池，重新初始化
private:
        b2Chunk* m_chunks; //chunk数组，即多个大块内存的集合
        int32 m_chunkCount; //已使用的chunk数，用于判断m_chunks数组是否使用完毕，还可以用来获取已使用的最后一个chunk来求得新建的chunk应该在的位置
        int32 m_chunkSpace; //可以存放的chunk数，可理解为m_chunks数组长度
        b2Block* m_freeLists[b2_blockSizes]; //i类型（block类型）的chunk中空闲节点(block)
        static int32 s_blockSizes[b2_blockSizes]; //block的b2_blockSizes种类型，一次赋值，终身受用
        static uint8 s_blockSizeLookup[b2_maxBlockSize + 1]; //存放i（0~b2_maxBlockSize）大小在s_blockSizes数组中类型位置索引
        static bool s_blockSizeLookupInitialized; //判断lookup数组是否初始化过
};
```

关于 m_freeLists 和 chunk 的关系可以参考下图：



若某类型 chunk 使用完毕则在 m_chunks 数组中建一个该类型数组，同时改变相应 m_freeLists 的指向。

若 m_chunks 数组使用完毕,则建新数组,新数组比旧数组大 b2_chunkArrayIncrement，并将旧数组 copy 到新数组，销毁旧数组。