Panda3D 物理引擎分析

2015 级软件工程卓越班

练孙鸿

（3D 游戏引擎架构设计基础课程论文，本课程论文已选择"优秀"）

0 摘要

本文从文档、c++代码及相关文献入手介绍了 Panda3D 的物理引擎 Bullet。Bullet 是一个用于游戏、电影行业的跨平台开源碰撞检测及物理程序库，世界三大物理模拟引擎之一。本文介绍了物理引擎的现状、Bullet 引擎 c++项目的概览、碰撞检测模块、刚体动力学模拟模块。中间有部分算法、数学的简要介绍。由于 Bullet 是一个非常庞大、复杂、技术含量高的库，所以本文虽然有将近两万字，但是却只非常概要性地介绍了 Bullet 相关设计与技术。最后笔者给出了结论，分析了 Bullet 物理库的优缺点。

关键词 Bullet 物理引擎 刚体动力学 碰撞检测

1 引言

1.1 游戏中的物理引擎

在真实世界中，物体的运行是遵循物理规则的。真实世界的物体之间也是存在着遵循物理规则的相互作用，例如两个物体碰撞在一起之后，必定会产生线速度、角速度的变化，这是非常自然而然的事情。但是在游戏中，两个物体（一般情况下都是网格）的物理属性却不是天然存在的，这种物体间按照一定简化的物理规则的交互是需要用程序逻辑来实现。所以程序员其实要花很多经历来保证物体之间不会穿透，甚至有一定程序的物理互动。那么游戏开发就产生了对**物理引擎(physic engine)**的需求。

游戏里面的物理引擎/物理系统可以进行物理实体之间的**刚体动力学(rigid body dynamics)**模拟。[1]刚体(rigid body)是经过理想化的，无论对其施加多少外力，形变均为 0 的物体。刚体上任意两点的距离会一直维持恒定。在这个语境下，[2]刚体动力学模拟是一个模拟过程，它计算了一个或多个刚体在**力(force)**的作用下随时间移动和相互作用。

因为物理引擎需要模拟物体间的接触和相互作用，所以物理引擎高度依赖于**碰撞检测系统(collision detection system)**，并利用碰撞检测正确地模拟物体的多种物理行为（包括碰撞并弹开、在摩擦力作用下滑行并逐渐停止等）。当然许多游戏都没有专门的物理引擎，而只有最基础最重要的碰撞检测系统。但是如果加上了物理引擎，那么游戏作品给玩家的体验也会更上一层。

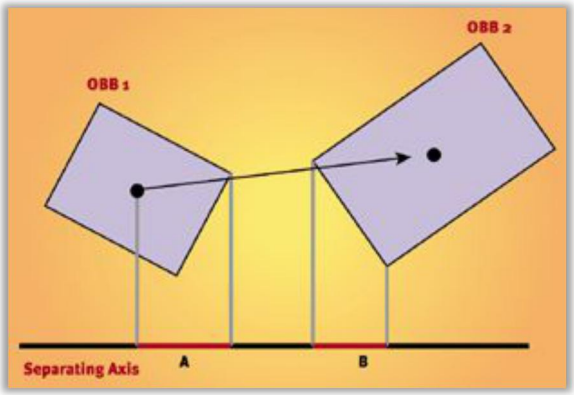


图 1-1：碰撞检测（分离轴定理）

1.2 物理引擎可以做的事

这里的物理引擎指的是一般的刚体动力学模拟引擎，[2]那么它可以实现的效果是：

- 检测动画物体与静态世界几何体之间的碰撞
- 模拟刚体受到引力的影响
- 可破坏的建筑物和结构
- 射线的投射以及求解（例如求解子弹弹道）
- 触发体积(trigger volume)（判断物体进入/离开游戏世界中定义的区域，或者停留在区域内的时间）
- 容许角色捡起刚体
- 复杂机器（起重机、移动平台等，例如 Portal 2 就有具有物理属性的方块）
- 带有逼真悬挂系统(suspension)的可驾驶载具
- 布娃娃系统
- 布料模拟
- 毛发物理模拟
- ……

除了刚体动力学物体引擎以外，还有更高级的实时流体模拟引擎。但是流体模拟的开销比较大，所以一般在游戏里面，只有简单的刚体动力学模拟。

1.3 碰撞/物理中间件

物理模拟是一个火热的研究方向，其技术含量也是非常的高，问题都极具挑战性，实现的工程难度也很大，而且运行时的运算量也非常大。所以为了更加专业化地实现物理模拟效果，物理模拟系统一般都以中间件的形式存在。中间件开发商可以更加专注、更加专业地解决物理模拟这一极具挑战性的问题，而不用所有的游戏程序员专门为了这一个功能去头疼和钻研。

庆幸的是，我们有很多开源免费/商用的物理模拟中间件可以选择，他们都各有千秋，下面来介绍几款可供游戏使用的碰撞检测/物理引擎[2]：

1.ODE：[3]全称 Open Dynamics Engine（开放动力学引擎），这是一个开源碰撞以及刚体动力学 SDK。与商用产品相近，但是却免费和完全开源！作者是 Russell Smith 和几位开源社区贡献者。ODE 主要实现了单摆运动 Panda3D 里面整合了基于 ODE 的物理系统。

2.Bullet：[4]Bullet 是一个用于游戏、电影行业的跨平台开源碰撞检测及物理程序库，世界三大物理模拟引擎之一。它的碰撞引擎和动力学模拟是整合在一起的，但是碰撞引擎也提供了钩子供独立使用或者整合到其他物理引擎上面。[2]Bullet 支持**连续碰撞检测（continuous collision detection, CCD）**，这又叫做**冲击时间（time of impact, TOI）**碰撞检测。这种检测对于检测细小而高速移动的物体很有帮助。

3.NVIDIA PhysX：[5]PhysX 读音与 Physics 相同，由 AGEIA 公司开发的物理模拟引擎，后来 Nvidia 收购了 AGEIA。PhysX 是世界三大物理模拟引擎之一。在 AGEIA 被 Nvidia 收购以后，PhysX 被改造成可以在 GPU 协处理器上使用。PhysX 的 SDK 可以在 Nvidia 的官网上下载到。

4.Havok：[6]Havok 全称 Havok Game Dynamic SDK，它是商业物理 SDK 的绝对标准，世界三大物理引擎之一。它提供了最丰富的功能集，性能在各个平台上都非常棒（毫无疑问是最贵的）。Havok 由一个核心碰撞/物理引擎，加上数个可选的产品构成。可选的产品包括载具物理系统、为可破坏环境建模的系统、全功能动画 SDK（整合了布娃娃系统）

1.4 Panda3D 支持的物理引擎

Panda3D 游戏引擎里面提供了几种物理系统的选择[7]：

(1) Panda 内建(built-in)的物理引擎：可以把力(force)施加在不同的类上。这个引擎可以用力来改变物体的线速度、角速度，并且可以模拟粘性(viscosity)。

(2) ODE：Open Dynamic Engine，前文（1.3）提到过，从 Panda3D 1.5.3 开始支持。

(3) NVIDIA PhysX：前文（1.3）提到过，从 Panda3D 1.7.0 开始支持，但截止至 2018.5.14 为止依旧没有相关文档介绍。

(4) Bullet Physic Engine：前文（1.3）提到过，Panda3D 1.8.0 开始支持。

如果用户是想要非常简单的物理模拟，那么用 Panda 内建的物理引擎就足够了（而且内建的物理引擎会使用 Panda3D 自己的碰撞检测模块，理论上配合会更加好）。但是 Panda 内建的物理引擎可能功能不够强大，那么用户可以切换至 ODE，PhysX，Bullet 等物理系统以寻求更加强大的物理模拟表现。

所以在 Panda3D 实现/集成的物理系统里面，最高配、最完整、开源的物理模拟系统就只有 Bullet Physic Engine 了。为了使本文的论述更加有针对性，本文决定选择 Bullet 物理引擎来进行详细分析。

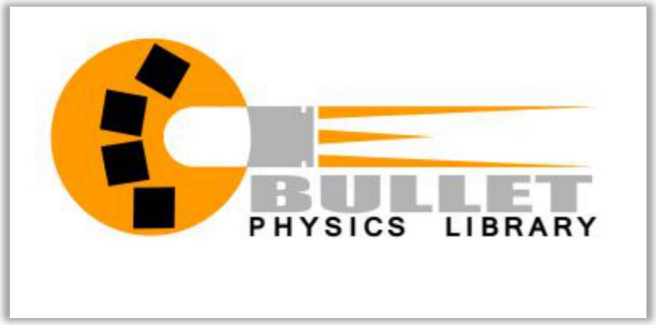


图 1-2：Bullet Physics 图标

Bullet 物理引擎是一个现代的开源物理引擎，多用在游戏和模拟程序里面。Bullet 可以在很多平台上编译，例如 Windows，Linux，MacOSX。Bullet 支持碰撞

检测，刚体物理动力学，柔体动力学，运动角色控制器，可见其功能的强大。

Panda3D 花费了很多的功夫去把 Bullet 集成进来，因为要考虑到 Panda3D 与 Bullet 的交互和合作。所以以目前的情况来说，只能说是把 Panda 与 Bullet 的类整合得比较合理，联系尽量紧密，集成度尽量高。有一点要注意的是，在用集成在 Panda 上的 Bullet 时，碰撞测试和其他物理效果、碰撞节点不能与其他物理系统混用。说得更加具体一点就是，Bullet 的物体不能和 ODE、PhysX 的物理实体进行交互和碰撞。

2 Bullet 物理引擎概览

这一章先从 Bullet 的 Panda3D 集成版的 Hello World in python 引入。然后我们着重分析 Bullet3 开源物理引擎的源码。为了分析的方便，笔者从 https://github.com/bulletphysics/bullet3 下载了 Bullet 的最新开源代码分支，并对 Bullet 物理引擎进行基于 c++代码的独立分析（先排除掉物理中间件和游戏引擎的耦合）。

2.1 Panda3D Bullet 快速入门

2.1.1 BulletWorld：物理世界

在 Panda3D 里面要使用 Bullet 物理引擎，我们首先需要一个 BulletWorld 对象[8]。这里的”world”是 panda 里面的一个属于，意义跟“空间”“场景”是一致的。这个单独的物理世界里面就有很多物体，如刚体(rigid body)、柔体(soft body)、角色控制器(character controller)。BulletWorld 控制了一些全局参数，例如重力、物理模拟状态。

下面的 Python 代码创建了一个 BulletWorld，一个物理模拟空间，然后给这个模拟空间设置了方向向下、值为 9.81 的重力。虽然 Bullet 物理引擎从理论上来讲是不需要单位的，但是为了实现更加真实的模拟，Bullet 建议还是使用 SI 国际单位(kg,m,s 等单位)。

```
from panda3d.bullet import BulletWorld
world = BulletWorld()
world.setGravity(Vec3(0, 0, -9.81))
```

之后我们可以推进物理系统的模拟，在 Panda3D 里面，这最好是用一个每帧都会被调用的 task 来实现。我们把相邻帧流逝的时间 dt 传给 Bullet 物理系统，Bullet 根据流逝的时间进行模拟，具体的模拟由 doPhysics()函数来驱动。

```
def update(task):
    dt = globalClock.getDt()
    world.doPhysics(dt)
    return task.cont
taskMgr.add(update, 'update')
```

其中 doPhysics()方法允许用户进行更加细致的控制。Bullet 内部是把一个时间步长(就是传入的 dt)分为多个时间子步长。我们可以往这个方法里面传入最大模拟步数和子步骤的大小，像下面的代码就设定了最多 10 个子步(substep)，每个都是 1/180 秒。子步越多，子步长越短，运行效率就会越低，这是显而易见的，因为迭代越多效果越好，但是运行更慢。

```
world.doPhysics(dt, 10, 1.0/180.0)
```

2.1.2 静态刚体对象(Static Body)

到目前为止我们只有一个空的物理世界，我们可以继续添加物体。最简单的物体就是静态物体(static body)。Bullet 里面的静态物体不会随着时间改变位置或者朝向，典型的静态物体就是地面、地形、房子等对象。下面的 python 代码可以创建一个形状是平面的静态刚体：

```
from panda3d.bullet import BulletPlaneShape
from panda3d.bullet import BulletRigidBodyNode

shape = BulletPlaneShape(Vec3(0, 0, 1), 1)

node = BulletRigidBodyNode('Ground')
node.addShape(shape)

np = render.attachNewNode(node)
np.setPos(0, 0, -2)

world.attachRigidBody(node)
```

这里的 BulletRigidBodyNode 是继承自 PandaNode 的，所以创建的物理刚体节点也可以绑定到 Panda3D 的场景图(Scene Graph)上并进行渲染，这个可以算是 Panda3D 对 Bullet 集成的体现。所以要添加一个具有物理属性的静态物体，步骤是：

- 创建物理节点 RigidBodyNode
- 给节点绑定一个形状（同时用于渲染和碰撞）
- 节点绑定到 Panda3D 场景图
- 节点绑定到 Bullet 物理世界

之后可以调用节点的 SetPos()、SetH()等去设置节点的位置。要注意的是，只有绑定到了 BulletWorld 的节点才会在每一步的物体模拟考虑进去。

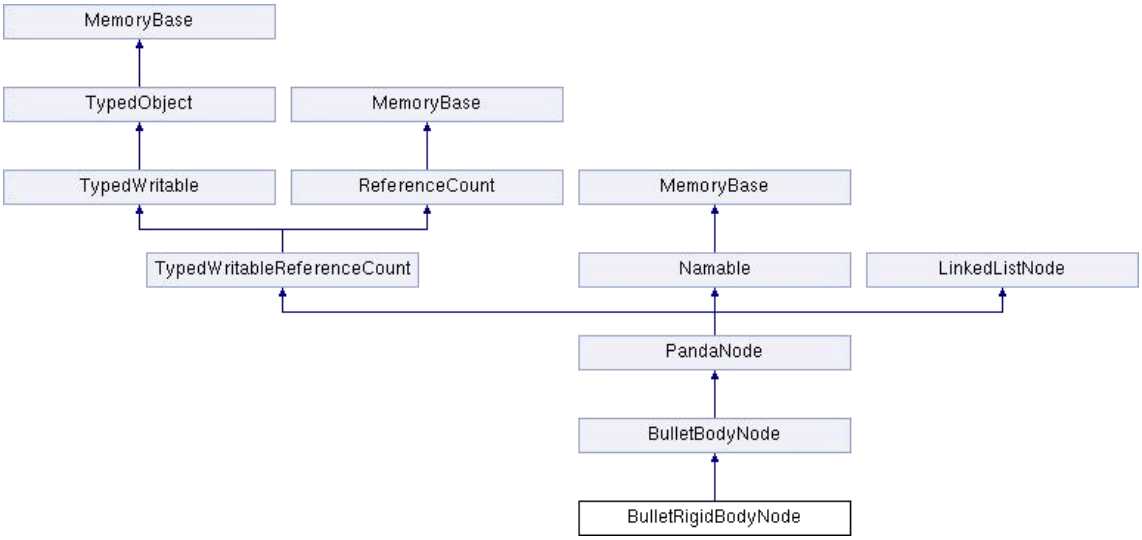


图 2-1：BulletRigidBodyNode 的继承关系图

2.1.3 动态刚体对象(Dynamic Body)

动态刚体对象跟静态刚体对象是比较相似的，不同点在于动态刚体可以通过施加力(force)和力矩(torque)在物理世界里面移动和旋转。而且既然要模拟外力对物体的影响，我们还需要设置物体的质量(mass)。静态和动态对象的创建取决于设置的质量：设置一个有穷质量可以创建一个动态刚体，设置质量为 0 可以创建一个静态刚体。物体被设置成零质量的话，Bullet 会自动将它当成无穷大的质量，而无穷大的质量意味着无法用外力对其动量、角动量造成影响，也就是所谓的“静态刚体”了。下面是创建动态刚体的 python 代码：

```
from panda3d.bullet import BulletBoxShape

shape = BulletBoxShape(Vec3(0.5, 0.5, 0.5))

node = BulletRigidBodyNode('Box')
node.setMass(1.0)
node.addShape(shape)

np = render.attachNewNode(node)
np.setPos(0, 0, 2)

world.attachRigidBody(node)
```

Bullet 会在每一帧 doPhysics()进行物理模拟，并自动更新一个刚体节点(rigid body node)的位置和朝向(orientation)。所以如果你有一个几何节点 GeomNode，那么可以把这个几何节点 reparentTo()这个刚体节点，这个 geomNode 也会跟着刚体物理节点移动和旋转。这样一样，我们就不需要考虑用于渲染的场景和物理世界的同步(synchronize)。这就是 Panda3D 对 Bullet 的集成，给人一种舒适的体验。

2.1.4 panda3d.bullet 示例程序

```
import direct.directbase.DirectStart
from panda3d.core import Vec3
from panda3d.bullet import BulletWorld
from panda3d.bullet import BulletPlaneShape
from panda3d.bullet import BulletRigidBodyNode
from panda3d.bullet import BulletBoxShape

base.cam.setPos(0, -10, 0)
base.cam.lookAt(0, 0, 0)

# World
world = BulletWorld()
world.setGravity(Vec3(0, 0, -9.81))

# Plane
shape = BulletPlaneShape(Vec3(0, 0, 1), 1)
node = BulletRigidBodyNode('Ground')
node.addShape(shape)
np = render.attachNewNode(node)
np.setPos(0, 0, -2)
world.attachRigidBody(node)

# Box
shape = BulletBoxShape(Vec3(0.5, 0.5, 0.5))
node = BulletRigidBodyNode('Box')
node.setMass(1.0)
node.addShape(shape)
np = render.attachNewNode(node)
```

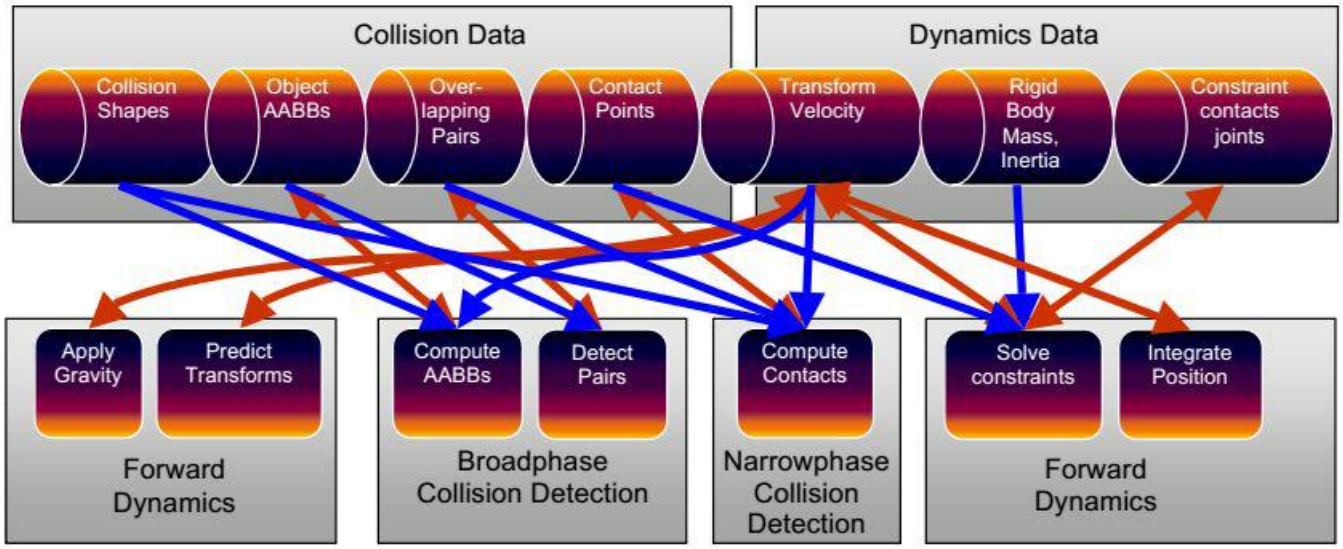


图 2-2：Bullet 刚体物理计算管线

```
np.setPos(0, 0, 2)
world.attachRigidBody(node)
model = loader.loadModel('models/box.egg')
model.flattenLight()
model.reparentTo(np)

# Update
def update(task):
    dt = globalClock.getDt()
    world.doPhysics(dt)
    return task.cont

taskMgr.add(update, 'update')
run()
```

2.2 Bullet 的 C++解决方案概览

为了更加方便地分析物理系统，笔者决定单独分析 Bullet 的非集成版（原生的 SDK）。

github 项目地址：

https://github.com/bulletphysics/bullet3

利用 cmake 生成 visual studio 2015 项目文件之后，我们将根据项目的用户手册、API 手册、c++源码对 bullet 物理引擎进行分析。手册与文档的版本是 bullet 2.83，2015.5。

2.2.1 Bullet 库设计概览

图 2-3 图是 Bullet 库的主要模块与架构[9]，上层的模块会对下层有一定程度的依赖：

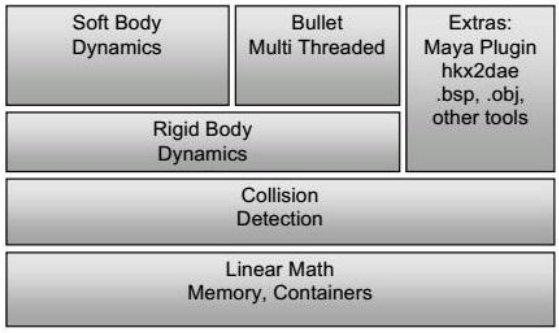


图 2-3：Bullet 主要模块与架构

1. Linearn Math, Memory, Containers：线性数学、内存管理、容器，这一层包含的都是最底层的模块

2. Collision Detection：碰撞检测，也就是各种几何体（射线、圆柱体、立方体、凸包、网格模型）之间的求交，一般需要实现“判断两个物体是否有交集”和“如果两个物体有交集那么相交点是什么”

3. Rigid Body Dynamics：刚体动力学模拟，可以说是 Bullet 物理引擎的核心。

4. Soft Body Dynamics：柔体动力学模拟。在 Bullet 里面例如绳子、布料、布娃娃(rag doll)之类的“柔体”(soft body)可以看作是很多刚体元素用约束绑定在一起之后的模拟结果。

5. Bullet Multi Threaded：Bullet 库的多线程支持，本文不作分析。

6. Extras：其他插件、工具模块，例如.bsp/.obj 文件加载。

图 2-2 是 Bullet 的物理模拟计算管线(Bullet physics pipeline)流程图[9]，图中包含了物理模拟所需的重要数据结构和阶段。整个物理模拟流程是**从左往右执行**的，从一开始的施加重力(gravity)，执行到最后的位置积分(integration)和变换更新。从图中可以

知道物理模拟管线有下面的组成要素：

数据：

- 碰撞数据(Coliision Data)：
  - 用于碰撞的形状池
  - 物体的轴对齐包围盒 AABB(Axis-Aligned Bounding Box)
  - 重合的物体对(Overlapping pairs)
  - 接触点(contact point)(应该指的是 object pair 接触&碰撞的空间点)
- 动力学数据(Dynamics Data)：
  - 变换速度(Transform Velocity)
  - 刚体质量与惯性(Rigid Body Mass & Inertia)
  - 约束关节点(Constraint contact joints)

物理管线阶段：

- 1.前向动力学(Forward Dynamics)：
  - 施加引力(Apply Gravity)
  - 预测刚体变换(Predict Transform)
- 2.粗略的碰撞检测阶段(Boardphase Collision Detection)
  - 计算 AABB
  - 检测碰撞对（与 Overlapping pairs 数据有交互）
- 3.细致的碰撞检测阶段(Narrowphase Collision Detection)
  - 计算精确的物理碰撞/接触点(ompute contact point)
- 4.前向动力学(Forward Dynamics)：
  - 求解约束(Solve constraints)
  - 对位置进行积分(Integrate Position)

注意到物理管线图里面有两种颜色的箭头：蓝色箭头（数据单向流动，只读）；红色箭头（数据双向流动，物理管线的计算会对指向的数据有影响，可读可写）。那么大致可以从图中分析出一些信息：

1. 第一阶段先预测前向动力学变换
2. 粗略的碰撞检测是用 AABB 与分离轴定理来检测可能碰撞的物体对(overlapping pair)。
3. 精确的碰撞检测是基于粗略碰撞检测阶段得到的物体对，进行精确的几何求交，并求出接触点（contact point）
4. 约束(constraints)会影响刚体动力学的模拟结果，需要多一个阶段来求解。

2.2.2 Bullet Visual Studio 解决方案结构

Bullet 的解决方案大致由如下的项目组成：

1. 名字是 App_XXX 的项目都会在相应的目录下生成.exe 可执行文件。这些 exe 都是 bullet 功能的演示 demo,但不一定有 GUI 和 3D 渲染(GUI 是要另外实现的)。例如 App_HelloWorld 就是生成一个控制台程序，输出所有模拟的中间步骤，包括各个物体的世界坐标、姿态四元数等。这里比较重要的是 App_ExampleBrowser 项目，这个项目驱动了 bullet 的绝大部分例子程序，下文会介绍 example browser。
2. 名字是 BulletXXX 的项目是 Bullet 的核心组件，包含了 bullet 历代版本的一些核心组件，例如碰撞检测（Collision）、动力学（dynamics）、文件加载器（FileLoader）、柔体（Soft Body）等。可以发现一些有些项目名字比较类似，只是项目的前缀不一样。其实他们是相互补充的，例如 BulletCollision 和 Bullet3Collision。
3. 其余的项目都是一些 utility 类的辅助模块、测试模块，如 LinearMath 一些线性数学辅助函数，OpenGLWindow 基于 OpenGL 的渲染窗口辅助模块等。

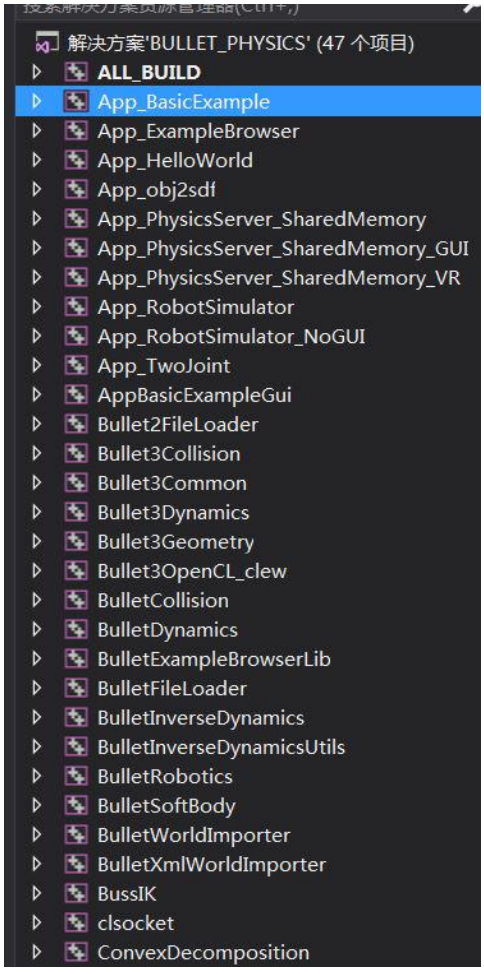
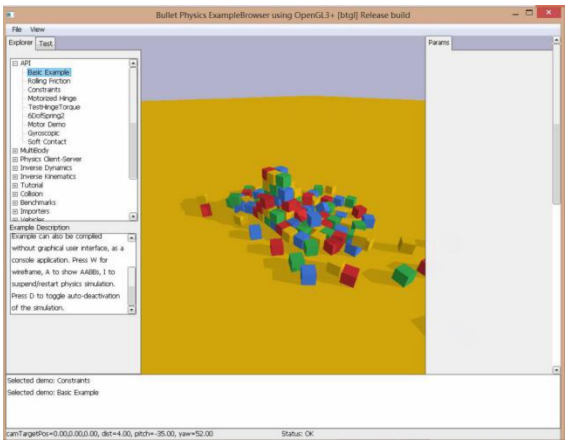


图 2-4：Bulle Visual Studio 解决方案项目浏览器

2.2.3 Bullet 的示例程序浏览器(Example Browser)

在生成了 bullet 解决方案以后，我们会在\example\exampleBrowser\$(Configuration)\目录下得到一个 Example Browser 的可执行程序。这是 bullet 2.8.3 新推出的特性，这个 exe 打包了很多例子程序，用户可以在 Example Browser GUI 左边的列表栏里面选择要运行的例程。这些例子程序都在 App_ExampleBrowser 与 BulletExampleBrowserLib 项目里面进行了实现。



图：Bullet Example Browser 实现

我们可以通过这个 exampleBrowser 来直观地感受 bullet 引擎的功能，然后从这个项目开始自顶向下地分析 Bullet 的源码。

Bullet 源码的注释比起 OGRE 这种优秀开源渲染引擎项目来说真的是太少了，所以分析源码并不容易入手。但是可以看到 App_ExampleBrowser 项目里面有一些.h/.cpp 文件是与例程浏览器里面的条目的名字一样的，一些却不是，我们可以过滤掉其他源码，先从例子程序开始分析。

Bullet 最简单的例子是 Basic Example，Browser GUI 的第一个就是。这一个例子就是一堆小立方体没有任何约束地堆成一个大立方体，然后掉下来散开。对于 Bullet 来说，带约束(constraint)、带动力马达（motor）、带摩擦力（friction）等的场景才是相对比较高级的功能。所以才说 BasicExample 是“基础”的。

在 BasicExample.h 里面，只有一句声明：

```
#ifndef BASIC_EXAMPLE_H
#define BASIC_EXAMPLE_H
class CommonExampleInterface*
BasicExampleCreateFunc(struct CommonExampleOptions&
options);
#endif //BASIC_DEMO_PHYSICS_SETUP_H
```

这是 ExampleBrowser 组织例子程序的方法，要求**每个例子程序都声明并定义一个例子程序创建函数 XXXCreateFunc()**，返回值是一个 CommonExampleInterface*。

然后我们来看相关实现：

BasicExample.cpp:

```
struct BasicExample : public CommonRigidBodyBase{...};
.....

CommonExampleInterface*
BasicExampleCreateFunc(CommonExampleOptions& options)
{
    return new BasicExample(options.m_guiHelper);
}
```

CommonRigidBodyBase.h：

```
struct CommonRigidBodyBase : public
CommonExampleInterface{....}
```

这意思就是每一个物理模拟的 demo 都需要实现 CommonExampleInterface 接口，如果再特别指定到刚体模拟的 demo，就需要实现派生自 CommonExampleInterface 的接口类 CommonRigidBodyBase。虽然我们不需要过多地纠结 ExampleBrowser 的 GUI 和例子程序是怎么组织的，但是从这些接口类我们大致窥探到完成一个物理模拟过程的流程，从而找到分析更深层代码的入口。那么就先来看一下每个例子所需要的实现的接口吧：

```
class CommonExampleInterface
{
public:
    ...
    virtual void initPhysics()=0;
    virtual void exitPhysics()=0;
    virtual void updateGraphics(){}
    virtual void stepSimulation(float deltaTime)=0;
    virtual void renderScene()=0;
    virtual void physicsDebugDraw(int debugFlags)=0;//for now
we reuse the flags in
    virtual void resetCamera(){};
    virtual bool mouseMoveCallback(float x,float y)=0;
    virtual bool mouseButtonCallback(int button, int state, float
x, float y)=0;
    virtual bool keyboardCallback(int key, int state)=0;
    ...
};
```

从接口类的定义来看，每个 demo 程序都需要实现物理世界初始化、物理世界销毁、OpenGL 渲染相关的接口。但因为这一章主要讲 Bullet 解决方案的结构，所以在此不再展开太多。总结一下，App_ExampleBrowser

项目作为一个例子程序浏览器，涵盖了 Bullet 所有的 demo，每个 demo 都需要实现 CommonExampleInterface 接口类的一些接口，那么所有 demo 程序都可以以同样的方式管理。

2.3 使用 Bullet 做一个 C++ demo 的流程

创建一个基于 Bullet 的小程序流程都是大同小异的，因为 Bullet 的物理管线流程就是那样子。用户可以按照 Bullet 的 HelloWorld（在 Bullet\examples\HelloWorld 文件夹下）项目来入手。下面的 HelloWorld.cpp 代码经过精简和修改：

```
//File: HelloWorld.cpp(Simplified Version)

#include "btBulletDynamicsCommon.h"
#include <stdio.h>
/// This is a Hello World program for running a basic Bullet physics simulation
int main(int argc, char** argv)
{
	int i;
	btDefaultCollisionConfiguration* collisionConfiguration = new btDefaultCollisionConfiguration();
	btCollisionDispatcher* dispatcher = new btCollisionDispatcher(collisionConfiguration);
	btBroadphaseInterface* overlappingPairCache = new btDbvtBroadphase();
	btSequentialImpulseConstraintSolver* solver = new btSequentialImpulseConstraintSolver;
	btDiscreteDynamicsWorld* dynamicsWorld = new btDiscreteDynamicsWorld(dispatcher, overlappingPairCache, solver, collisionConfiguration);
	dynamicsWorld->setGravity(btVector3(0, -10, 0));
	btAlignedObjectArray<btCollisionShape*> collisionShapes;

	///create a few basic rigid bodies
	{
		btCollisionShape* groundShape = new btBoxShape(btVector3(btScalar(50.), btScalar(50.), btScalar(50.)));
		collisionShapes.push_back(groundShape);
		btTransform groundTransform;
		groundTransform.setIdentity();
		groundTransform.setOrigin(btVector3(0, -56, 0));
		btScalar mass(0.);
		bool isDynamic = (mass != 0.f);
		btVector3 localInertia(0, 0, 0);
		if (isDynamic)groundShape->calculateLocalInertia(mass, localInertia);

		btDefaultMotionState* myMotionState = new btDefaultMotionState(groundTransform);
		btRigidBody::btRigidBodyConstructionInfo rbInfo(mass, myMotionState, groundShape, localInertia);
		btRigidBody* body = new btRigidBody(rbInfo);

		//add the body to the dynamics world
		dynamicsWorld->addRigidBody(body);
	}

	.....

	/// Do some simulation
```

```
	///-----stepsimulation_start-----
	for (i = 0; i < 150; i++)
	{
		dynamicsWorld->stepSimulation(1.f / 60.f, 10);

		//print positions of all objects
		for (int j = dynamicsWorld->getNumCollisionObjects() - 1; j >= 0; j--)
		{
			btCollisionObject* obj = dynamicsWorld->getCollisionObjectArray()[j];
			btRigidBody* body = btRigidBody::upcast(obj);
			btTransform trans;
			if (body && body->getMotionState())body->getMotionState()->getWorldTransform(trans);
else trans = obj->getWorldTransform();

			printf("world pos object %d = %f,%f,%f\n", j, float(trans.getOrigin().getX()), float(trans.getOrigin().getY()),
float(trans.getOrigin().getZ()));
		}
	}///-----stepsimulation_end-----

	///-----cleanup_start-----
	...（各种销毁工作）
}
```

总结起来就分为几步：

1. 创建一个 btDiscreteDynamicsWorld 或 btSoftRigidDynamicsWorld（这两个类都继承自 btDynamicsWorld 接口类）
2. 创建 btRigidBody 然后把它添加到某个物理世界 btDynamicsWorld 里面。而为了构建 btRigidBody 或者 btCollisionObject，用户需要提供质量(mass)(动态物体质量大于 0，静态物体质量等于 0)、用于碰撞的形状（Collision Shape）、材质参数（摩擦力、碰撞的恢复系数 coefficient of restitution）。
3. 逐步模拟，每帧调用 byDynamicsWorld 的函数 StepSimulation()来推进物理世界的演进。

3 Bullet 底层基础设施

Bullet 几个主要模块里面，最底层的都是一些基础设施，例如一些几何数据结构、线性代数数学库、内存管理、容器、计时、debug 模式绘图等。

3.1 基本数据结构与数学库

刚体物理模拟与碰撞检测密切相关，那么自然也就跟线性代数、几何运算密切相关了。Bullet 在 LinearMath 项目里面实现一些基本的数据结构和相应的数学函数：

btScalar：为了可以让 Bullet 库被编译成使用单精度和双精度的浮点数，Bullet 里面的浮点数用 btScalar 来表示（OGRE，OpenCV 等开源库都是这样子，Ogre::Real，cv::Scalar）。默认情况下 btScalar 被 typedef 成 float。

btVector3：三维位置和向量可以用 btVector3 来表示。但是在实现的时候，btVector3 多了一个不用的 w 分量出来，这是为了充分使用 SIMD 加速而作的 16bytes 内存对齐。

```
#if defined (BT_USE_SSE) || defined(BT_USE_NEON) // _WIN32
|| ARM
	union {
		btSimdFloat4 mVec128;
		btScalar	m_floats[4];
	};
.....
```

```
#if defined(BT_USE_SSE_IN_API) && defined (BT_USE_SSE)
	__m128	vrt = _mm_load_ss(&rt);	//	(rt 0 0 0)
	btScalar s = btScalar(1.0) - rt;
	__m128	vs = _mm_load_ss(&s);	//	(S 0 0 0)
	vs = bt_pshufd_ps(vs, 0x80);	//	(S S S 0.0)
	__m128 r0 = _mm_mul_ps(v0.mVec128, vs);
	vrt = bt_pshufd_ps(vrt, 0x80);	//	(rt rt rt 0.0)
	__m128 r1 = _mm_mul_ps(v1.mVec128, vrt);
```

```
    __m128 tmp3 = _mm_add_ps(r0,r1);
    mVec128 = tmp3;
……
```

btVector3 的实现代码里面有很多这种画风的代码，都是直接上 SSE 指令集来进行底层的 SIMD/细粒度并行优化。诸如_mm_load_ss()这类的函数是操作系统(更准确的说是 Intel）提供的底层 SIMD 相关的接口。类似的底层优化可以在很多数学库里面发现，例如 DirectXMath，Eigen。

btQuaternion & btMatrix3x3:Bullet 里面，旋转和朝向(orientation)用四元数或者 3x3 矩阵表示。

btTransform:Bullet 里面的变换包含一个位置和旋转（注意没有缩放，因为 Bullet 是刚体物体引擎）。变换用于把点从一个坐标系转换到另一个坐标系下。

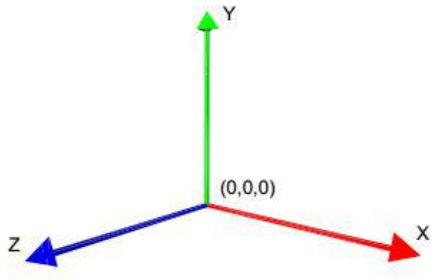


图 3-1：Bullet 使用右手系

还有其他的一些数据结构和工具例如 hash map，serializer，AABB，凸包计算器等不再展开分析。

3.2 内存管理、对齐

在 Bullet 里面，数据的 16 字节对齐(alignment)是很关键的[9]，例如用 SIMD 优化或者要用 DMA(Direct Memory Access)把数据传到 SPU(协处理器，Synergistic Processing Unit)的时候就要求数据对齐。因此，Bullet 提供了默认的、支持 16 字节对齐的内存分配器(Memory Allocator)。用户当然也可以自己实现，但要注意内存对齐。

Bullet 里内存分配相关的函数是：

1. **btAlignedAlloc**，分配对齐的内存，可指定分配的大小和对齐的长度。
2. **btAlignedFree**，释放 btAlignedAlloc 分配的内存。

如果用户要自己重写内存分配器，可以选择重写下面两个类：

1.**btAlignedAllocSetCustom**

2.**btAlignedAllocSetCustomAligned**

3.3 计时与性能分析

Bullet 用宏(macro)来实现层级性能分析（performance profiling）（虽然 Visual Studio IDE 本身也有性能分析器）：

- **btClock** 用于测量毫秒级精度的时间
- **BT_PROFILE(section_name)**用于标记性能分析代码段的开始
- **CProfileManager::dumpAll()**把性能分析结果输出到控制台
- **CProfileIterator** 是用于遍历性能分析树的迭代器类

4 碰撞检测

碰撞检测(Collision Detection)，是物理引擎里面很重要的一个较低层模块，用于各种形状之间的求交测试。一个游戏引擎可能不需要用到物理引擎，但是一定需要碰撞测试库，就例如最简单的子弹飞行击中敌人就可以简化为射线与碰撞几何体的求交运算。

Bullet 的碰撞检测提供了一系列的算法和加速结构来实现：

1. 最近点查询(closest point queries)
2. 射线与凸形的扫掠测试(sweep test)

下面讲一些主要的模块、类、数据结构。

4.1 碰撞物体(Collision Object)

Bullet 里面，碰撞物体用类 **btCollisionObject** 表示。**btCollisionObject** 是一个拥有世界变换(world transform)与碰撞形状的物体。它保存了所有用于实现碰撞检测的相关信息：形状、变换、AABB。

```
ATTRIBUTE_ALIGNED16(class)    btCollisionObject
{
protected:
    btTransform m_worldTransform;
    btTransform m_interpolationWorldTransform;
    btVector3 m_anisotropicFriction;
    int    m_hasAnisotropicFriction;

    btScalar    m_contactProcessingThreshold;
    btBroadphaseProxy* m_broadphaseHandle;
    btCollisionShape*    m_collisionShape;
    btCollisionShape*    m_rootCollisionShape;
   int m_worldArrayIndex;
    btScalar    m_friction;
```

```
    btScalar    m_restitution;
    btScalar    m_rollingFriction;
    btScalar m_spinningFriction;
    btScalar    m_contactDamping;
    btScalar    m_contactStiffness;
    .......
    btScalar    m_hitFraction;
    btScalar    m_ccdSweptSphereRadius;
    btScalar    m_ccdMotionThreshold;
    .......
}
```

上面是 btCollisionObject 类的部分成员变量的生命，为了节省文章的空间删掉了相关注释（虽然源码里面的注释少，而且写的很一般），这里解释部分变量的用途：

- m_worldTransform：世界变换
- m_anisotropicFriction：各向异性摩擦力（一般的摩擦力指的是各向同性的）
- m_collisionShape：碰撞形状，但是这个可能会被临时地替换为子碰撞形状（而不是原碰撞形状）
- m_rootCollisionShape：原始的碰撞形状
- m_worldArrayIndex：当前碰撞体在物理世界碰撞体列表里面的索引。
- m_friction：摩擦力
- m_restitution：弹性恢复系数（高中在讲刚体碰撞的时候有提到过，恢复系数为 1 就是完全弹性碰撞，0 是完全非弹性碰撞，但是都遵循动量守恒定律）
- m_rollingFriction：滚动摩擦力（torsional friction），用于防止一些球类碰撞体一直滚而停不下来
- m_spinningFriction：滚动摩擦力，在实现抓取物体(grasping)功能时很有用
- m_ccdSweptSphereRadius：CCD(Continuous Collision Detection)连续碰撞检测扫掠球的半径，具体用法与相关细节尚未进行分析。
- m_ccdMotionThreshold：如果单步的运动小于此阈值，就不进行 CCD。

简单分析了下 **btCollisionObject** 的成员变量就可以知道，它的功能不只是做几何求交，还包含了非常多有用的动力学参数。这些参数都会在具体的物理模拟里面使用到。

4.2 碰撞形状(Collision Shape)

Bullet 支持非常多的基本碰撞形状（立方体、球形、胶囊体、圆柱体等），还能自定义形状。除此之外，Bullet 碰撞系统还支持凹形的凸分解(convex decomposition)、复合对象(compound object)、高度场(height field)，华丽得无以复加，令人惊讶。接下来分析一下 Bullet 的碰撞形状。

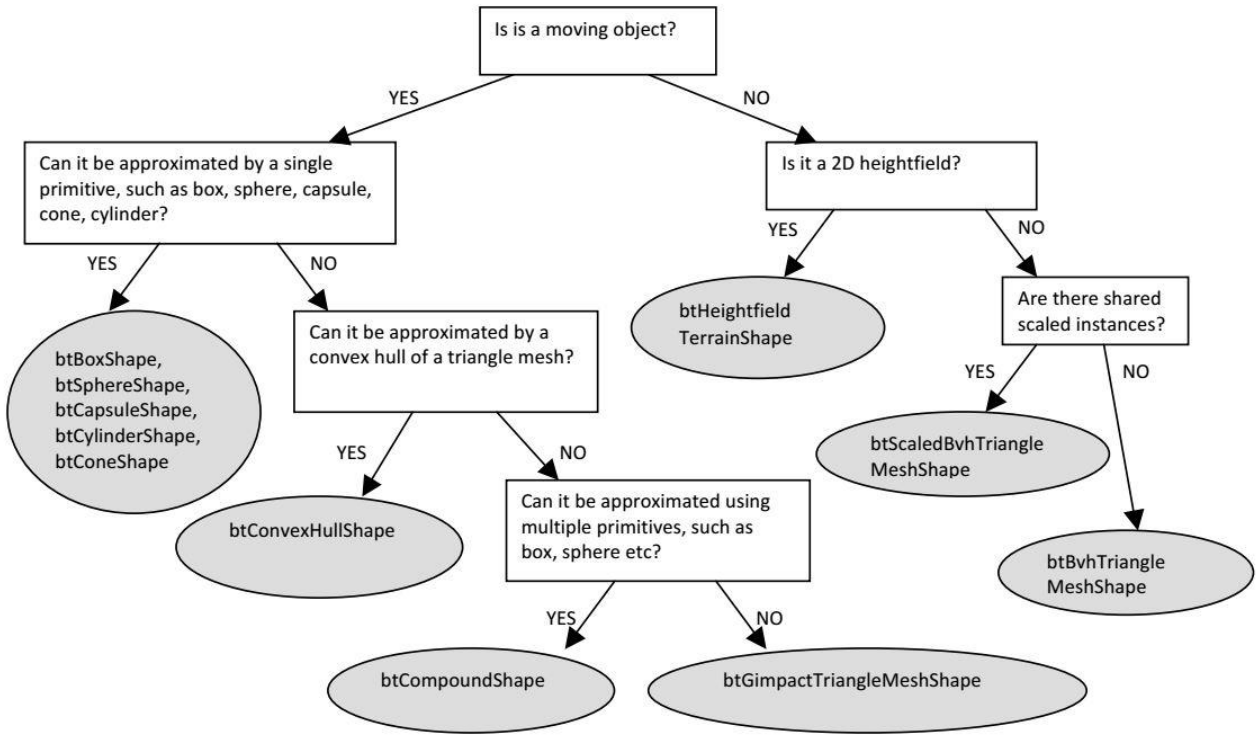


图 4-1:如何选择碰撞形状

4.2.1 如何选择合适的碰撞形状？

图 4-1 的决策树展示了 bullet 的使用者该如何选择合适的碰撞形状。可以看到的是 Bullet 有非常多的碰撞形状可以选择：

1. 基本凸形：btBoxShape，btSphereShape，btCapsuleShape，btCylinderShape，btConeShape
2. 凸包：btConvexHullShape
3. 复合对象：btCompoundShape
4. 可以移动的闭合三角形网格：btGimpactMeshShape (bullet3)
5. 高度场地形：btHeightFieldTerrainShape
6. 静止的三角形网格：btBvhTriangleMeshShape，btScaledBvhTriangleMeshShape

下面介绍一下部分碰撞形状。

4.2.2 基本凸形(Convex Primitive)

基本的凸形就是常见的几种简单几何体，而且都是可以用一到两个参数进行参数化的，在进行碰撞检测的时候可以轻易地进行逻辑上的检测。例如射线与碰撞球求交，就只需判断射线与球心的距离与碰撞球半径的关系。

btBoxShape：用半边长定义的立方体

btSphereShape：用半径定义的球体

btCapsuleShape：沿着 X/Y/Z 轴的胶囊体。由两端半球半径和中间柱形的长度定义。

btCylinderShape：沿着 X/Y/Z 轴的圆柱体。

btConeShape：沿着 X/Y/Z 轴的圆锥体

btMultiSphereShape：多个球体组合的凸包。

4.2.3 复合形状(Compound Shape)

多个凸形可以组成一个复合形状，用 **btCompoundShape** 来表示。

```
ATTRIBUTE_ALIGNED16(class) btCompoundShape	: public btCollisionShape
{
protected:

	btAlignedObjectArray<btCompoundShapeChild> m_children;
	btVector3 m_localAabbMin;
	btVector3 m_localAabbMax;
	btDbvt*	m_dynamicAabbTree;
	int	m_updateRevision;
	btScalar	m_collisionMargin;
	btVector3 m_localScaling;
public:
.....
	void  addChildShape(const btTransform& localTransform,btCollisionShape* shape);

	virtual void removeChildShape(btCollisionShape* shape);
	void removeChildShapeByIndex(int childShapeindex);
.....
}
```

复合对象的 AABB 是跟子碰撞形状相关的，所以复合对象就维护了一个子碰撞对象的 BVH(Bounding Volume Hierarchy)树，其中 btDbvt 就是这样的数据结构(bullet Dynamic bounding volumn tree)，这抽象的名字实在是服气。

4.2.4 凸包(Convex Hull)

Bullet 支持多种方法来表示凸包。所以创建 **btConvexHullShape** 最简单的方法是传入一系列的顶点。但是有时用于绘制的网格模型顶点太多了，这要是再把它直接传给 bulllet 当凸包然后进行碰撞检测的话，开销就太大了。这时候需要自行删减一些顶点。

```
///The btConvexHullShape implements an implicit convex hull of an array of vertices.
///Bullet provides a general and fast collision detector for convex shapes based on GJK and EPA using localGetSupportingVertex.
ATTRIBUTE_ALIGNED16(class) btConvexHullShape : public btPolyhedralConvexAabbCachingShape
{
  btAlignedObjectArray<btVector3>	m_unscaledPoints;
.....
}
```

btConvexHullShape 实现了一组顶点的隐式凸包。Bullet 提供了一个基于 GJK 和 EPA 算法的通用而快速的碰撞检测器。这两个算法会在下文进行介绍。

4.2.5 三角形网格(Triangle Mesh)

当简单的形状不足以近似描述碰撞体的话，就需要用更加复杂的三角形网格来描述了。**btBvhTriangleMeshShape** 和 **btScaledBvhTriangleMeshShape** 可以用来表示静态的三角形网格碰撞形状。可以用 **btTriangleMesh** 类给他们提供数据。

4.2.6 高度场地形(Height Field Terrain)

Bullet 提供了 btHeightFieldTerrainShape 来模拟地形，但是高度场的数据却不是由这个类来管理的。使用这个类要注意的一点是，它的坐标中心始终是 AABB 的

中心。例如，高度场的局部 Y 坐标范围是[-100,500]，那么 Bullet 就会根据其 AABB 重新确定坐标原点，于是局部 Y 坐标的范围就会变成[-300,300]。这可能对渲染物体不是很友好，但是却对物理模拟很有用。

```
ATTRIBUTE_ALIGNED16(class) btHeightfieldTerrainShape : public
btConcaveShape
{
protected:
……
    ///terrain data
    union
    {
    const unsigned char* m_heightfieldDataUnsignedChar;
    const short*     m_heightfieldDataShort;
    const btScalar*  m_heightfieldDataFloat;
    const void*      m_heightfieldDataUnknown;
    };
……
```

可以看到 btHeightFieldTerrainShape 并不实际包含高度场的数据。

4.3 碰撞检测策略

Bullet 之所以设计这么多不同的碰撞体，是因为不同的碰撞体都可以有不同程度的优化。很明显，越简单的形状计算起来越方便，基本几何体是最简单的，凹的三角形网格是最复杂的。

4.3.1 碰撞矩阵(Collision Matrix)

图 4-2 的碰撞矩阵是一个表格，里面列出了不同碰撞体之间的碰撞策略。其中，有一些特殊的碰撞检测策略：

- SAT：Seperating Axis Thereom，分离轴定理，快速检测两个凸形是否相交。
- GJK 算法：由 Gilbert, Johnson, Keerthi 开发的凸距离计算算法(convex calculation algorithm)。
- EPA：Expanding Polytope Algorithm，扩展多面体算法，作者 Gino van den Bergen。

Bullet 实现了自己的 GJK 和 EPA 算法。这些碰撞检测算法在很多其他地方都有可能使用到，例如视锥剔除、光线跟踪器等。所以下文将会简要介绍这些特殊的策略。

	box	sphere	convex,cylinder cone,capsule	compound	triangle mesh
box	boxbox	spherebox	gjk	compound	concaveconvex
sphere	spherebox	spheresphere	gjk	compound	concaveconvex
convex, cylinder, cone, capsule	gjk	gjk	gjk or SAT	compound	concaveconvex
compound	compound	compound	compound	compound	compound
triangle mesh	concaveconvex	concaveconvex	concaveconvex	compound	gimpact

图 4-2：碰撞矩阵(Collision Matrix)，不同的碰形状之间有不同的策略

4.3.2 分离轴定理

分离轴定理（SAT，Seperating Axis Theorem），是一个判断两个凸多边形是否碰撞的理论[10]。此理论可以用于找到最小的渗透向量（感觉应该是模最小的），此向量在物理模拟和其他很多应用中很有用。SAT 是一种高效的算法，能够处理每种形状对（譬如圆和圆，圆和多边形，多边形和线段）对碰撞检测代码的需求从而减少代码减轻维护压力。如果形状对里面有凹形，可以先进行凸形分解（convex decomposition）。

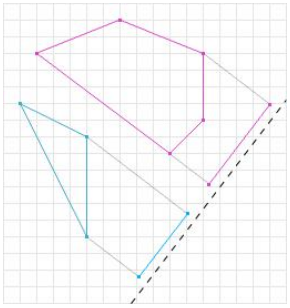


图 4-3：分离轴与投影

SAT 可能会测试多个轴来判断是否重叠，**如果找到一个轴两个多边形对应的投影没有重叠，那么这个算法立马可以得出结论这两个多边形没有相交**。因为这个前提，SAT 对于一些物体很多碰撞很少的应用（游戏，模拟，等等）都是非常理想的。

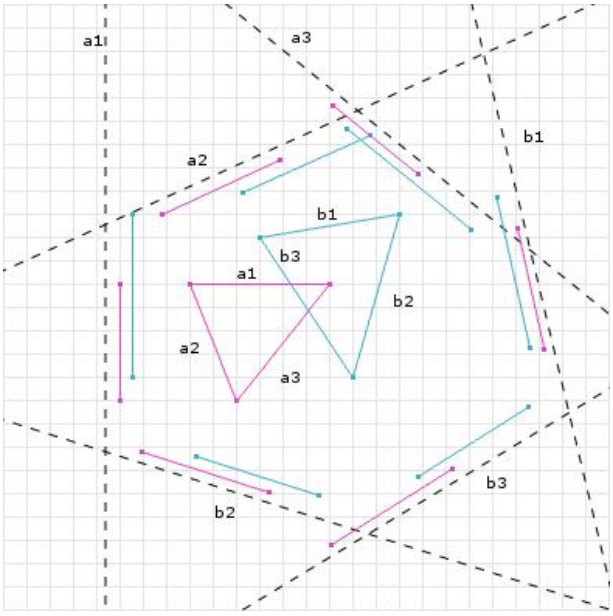


图 4-4：分离轴的选取

分离轴的选取是按照多边形的边及其法线决定：也就是所有需要测试的分离轴是平行于某条边的法线的，或者说分离轴垂直于某条多边形的边。2D-SAT 的例子如 4-4 所示。

SAT 算法有几点值得注意：

- SAT 可以找到一个“最小分离向量”（Minimum Translation Vector），沿着这个向量可以用最短的移动距离分开两个凸形。
- 被测试轴的数量可以通过不测试平行轴的方式减少。这就是为什么一个长方形可以只测试两个轴。
- SAT 推广到三维空间并不平凡，这里不展开分析，有待日后研究。

4.3.3 GJK 算法

GJK 算法是由 Gilbert,Johnson,Keerthi 开发的凸距离计算算法(convex calculation algorithm)。GJK 和 SAT 一样，只适用于凸多边形。GJK 更为强大的地方在于，它可以通过“支持函数”(稍后讨论)支持所有的形状。因此，和 SAT 不同，你不用对曲边型使用特殊的代码或者方法进行特殊处理，就可以使用 GJK。

GJK 是一个迭代式的方法但是收敛的非常快，如果使用最新的渗透（分离）向量，它几乎是在一个很短的时间内得出结果。它是 SAT 在 3D 环境下的一个很好的替代，就因为 SAT 要测试的轴的数量非常多。

GJK 的最初始的目的是为了计算两个凸边形的距离。GJK 同样可以被用作在渗透很短的情况下的碰撞信息,还可以在其他的计算很大的渗透的方法中使用。

在 GJK 算法里面，有个很重要的概念叫做闵可夫斯基差(Minkowski Difference)，对于形状 A、B，它们的闵可夫斯基差定义为：

$$A \ominus B = \{a - b | a \in A, b \in B\}$$

所以 Minkowski 差是按上式构建的点集。要注意的是我们的最终目的并不是求出 A 和 B 的 Minkowski 差，而是**尝试在这个 Minkowski 差里面找到并构建一个多边形，使得这个 Minkowski 空间中的多边形包含原点，从而推断出形状 A、B 相交**。GJK 算法迭代地在 Minkowski Difference 里面选取顶点并构建多边形，尝试包含原点。这里的多边形是**单纯形（Simplex）**。

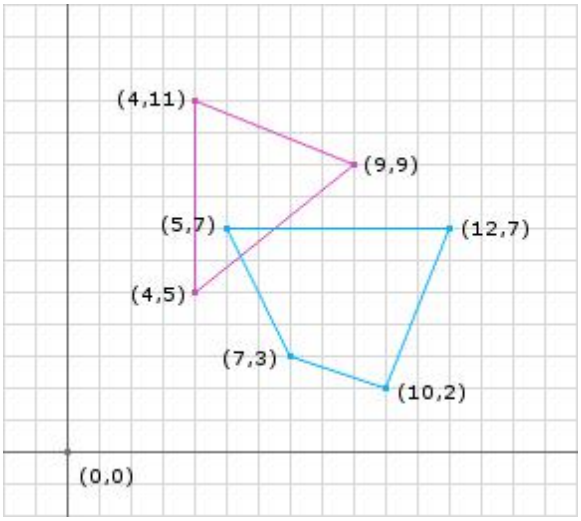


图 4-5：2D 平面上，两个相交的多边形

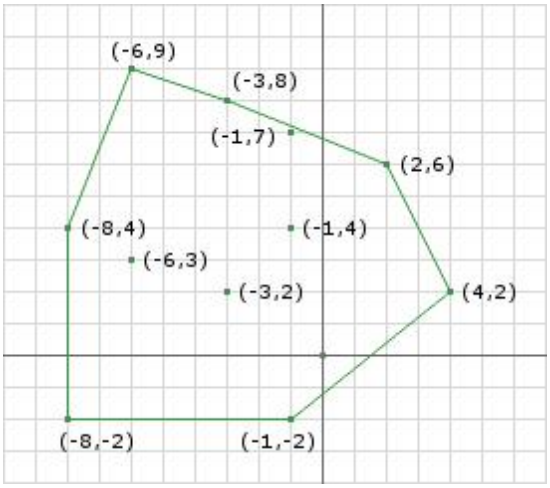


图 4-6：图 4-5 中两个多边形的 minkowski 差，可以发现它包含原点，从而可以判定两个形状相交

而在构建单纯形的时候，需要有一个辅助的支持函数(the support function)，输入一个向量，输出一个在 Minkowski 差里面的点。这个支持函数是致力于用最快的速度构建单纯形，尽快地尝试去覆盖原点来判断两个多边形是否碰撞。具体参考[13]。

4.3.4 EPA 算法

上一小节里面讨论的 GJK 算法只是用来判断两个凸形是否相交，返回值是 true/false。但是对于碰撞测试而言我们可能需要更多有用的信息，例如穿透深度(penetration depth)、穿透向量(penetration vector)，这些可以通过 EPA 算法来实现[14]。GJK 算法可以和 EPA 算法结合在一起使用，也就是当 GJK 算法检测出凸形对碰撞以后，可以用 EPA 算法计算这些信息。

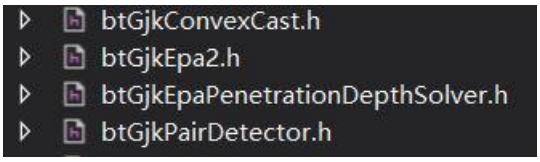


图 4-7：BulletCollision 里面 GJK-EPA 相关算法头文件

```
///btGjkEpaSolver contributed under zlib by Nathanael Presson
struct btGjkEpaSolver2
{
struct sResults
	{
	enum eStatus
		{
		Separated,
		Penetrating,
		GJK_Failed,
		EPA_Failed
		}		status;
	btVector3	witnesses[2];
	btVector3	normal;
	btScalar	distance;
	};

static int	StackSizeRequirement();
static bool	Distance(....);
static bool	Penetration(....);

#ifndef __SPU__
static btScalar	SignedDistance(....);
static bool	SignedDistance(....);
#endif //__SPU__
};
```

```
class btGjkEpaPenetrationDepthSolver : public
btConvexPenetrationDepthSolver
{
	public :
	btGjkEpaPenetrationDepthSolver(){}
	bool calcPenDepth(...);
};
```

可见 GJK-EPA 算法实现起来还是比较麻烦的。限于时间和篇幅，此处不展开，留待以后继续研究。

4.4 凸分解(convex decomposition)

在理想的情况下，凹形（concave object）只应该当作是静态的物体。动态物体的碰撞一般都是用凸形或者凹形的凸包[9]。但是如果凸包不够精确的话，我们就最好把凹形分解成凸形的组合(composite object of convex objects)。这个过程叫凸分解（convex decomposition），可以参考文献[18]。Bullet 实现的是 John Ratcliff 实现的层级近似凸分解 HACD（Hierarchical Approximate Convex Decomposition）。除了 Bullet 源码，网上还有其他 HACD 的实现[15]。

HACD 的大致思路是[16]：

1. 针对一个 3D mesh，建立一个所谓的 dual graph，按照我的理解，就是用来记录 mesh 上顶点、三角形等的拓扑关系。
2. 然后，对 mesh 的顶点迭代进行半边坍缩(half-edge collapse)操作，然后对 mesh 顶点进行聚类。在这个过程中用一个能量方程来进行控制，这个能量方程中由于考虑到了多边形的凹性(concavity)，所以，最后的结果对于原 mesh，有较好地匹配。
3. 最后，对聚类得到的每一组顶点，计算其 convex-hull，也就是包围体。

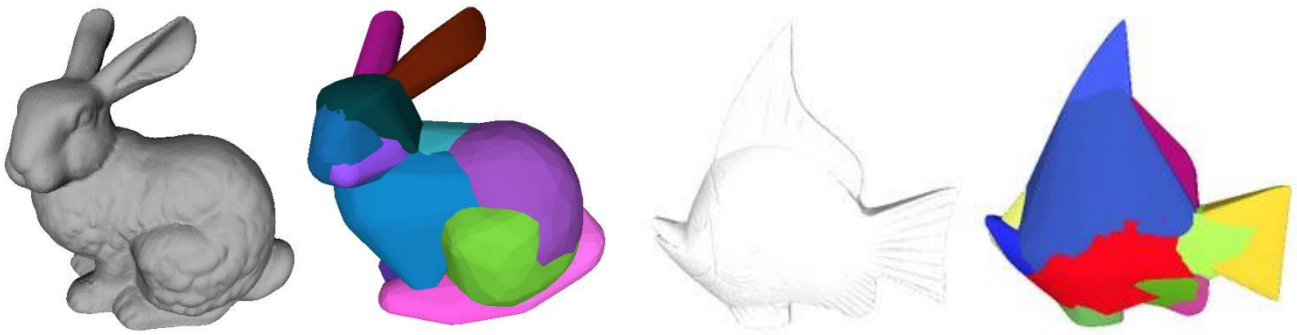


图 4-8：HACD 算法运行效果图

5 刚体动力学模拟

力学研究力（force）怎样影响物体的行为。在游戏引擎中，我们通常特别关注物体的动力学[17]（dynamics），即它人如何随时间移动。游戏中几乎只关注动力学中的经典刚体动力学（classical rigid body dynamics）。模拟中的刚体是完美的固体，不会变形。物理引擎也确保游戏世界中的刚体运动符合多个约束（constraint）。最常见的约束是**非穿透性（non-penetration）**，即物体不能互相穿透。当发现物体互相穿透时，物理系统要尝试提供真实的碰撞响应（collision response）。这是物理引擎与碰撞系统紧密关联的主因之一。

刚体动力学(Rigid Body Dynamics)模拟是物理引擎里面最重要的一个模块，(毕竟 Bullet 是个物理引擎)，前面介绍的碰撞检测等底层模块都是为了刚体动力学模拟做的铺垫。

可以参考一些经典的论文，如[20]：基于冲量的刚体动力学模拟。

5.1 Bullet 的刚体

Bullet 的刚体动力学模拟会模拟一些物理要素如：力(force)，质量(mass)，惯性(inertia)，约束(constraint)。

5.1.1 一些刚体动力学模拟相关的类

- **btRigidBody**：用于模拟单个 6-DOF 的自由移动物体。**btRigidBody** 继承自 **btCollisionObject**，所以它也继承了变换(transform)、摩擦力(friction)、恢复系数(restitution)，并且可以累加速度和角速度。

- **btTypedConstraint**：这是刚体物理约束的基类，有多种具体的约束，例如 **btHingeConstraint**, **btPoint2PointConstraint**, **btConeTwistConstraint**, **btSliderConstraint**, **btGeneric6DofConstraint**。

- **btDiscreteDynamicWorld**：这个类继承自动力学世界接口类 btDynamicsWorld，而 btDynamicsWorld 又继承自 btCollisionWorld，其中 btCollisionWorld 储存了所有的 btCollisionObject（详见第四章-碰撞检测）并且提供碰撞检测的查询接口。

```
class btCollisionWorld
{
protected:
    btAlignedObjectArray<btCollisionObject*>
m_collisionObjects;
    btDispatcher*  m_dispatcher1;
    btDispatcherInfo    m_dispatchInfo;
    btBroadphaseInterface*    m_broadphasePairCache;
    btIDebugDraw* m_debugDrawer;
    bool m_forceUpdateAllAabbs;
.....
}
```

```
class btDynamicsWorld : public btCollisionWorld
{.....
```

```
ATTRIBUTE_ALIGNED16(class) btDiscreteDynamicsWorld : public
btDynamicsWorld
{
protected:
    btAlignedObjectArray<btTypedConstraint*>
    m_sortedConstraints;
    InplaceSolverIslandCallback*    m_solverIslandCallback;
    btConstraintSolver*  m_constraintSolver;
    btSimulationIslandManager*    m_islandManager;
    btAlignedObjectArray<btTypedConstraint*> m_constraints;
    btAlignedObjectArray<btRigidBody*>
m_nonStaticRigidBodies;
    btVector3 m_gravity;
```

```
btScalar    m_localTime;
btScalar    m_fixedTimeStep;
bool m_ownsIslandManager;
bool m_ownsConstraintSolver;
bool m_synchronizeAllMotionStates;
bool m_applySpeculativeContactRestitution;
.....
```

- **btMultiBody**：这个类是比较新近才加入的类，是多级刚体的替代表示形式，相关算法可以参考 Roy Featherstone 的《Rigid Body Dynamics Algorithm》[19]。整个刚体的树状层级结构的根是由：
  1. 一个固定的或者移动的基座(base)
  2. 子刚体(child bodies)

  组成。他们之间用关节(joints)来连接:1-DOF 的旋转关节(revolute joint)(与 hinge constraint 类似)和 1-DOF 移动关节(prismatic joint)(与 slider constraint 类似)。

5.1.2 静态、动态、运动刚体

在 Bullet 里面有三种刚体：

1. 动态刚体(Dynamic bodies)：拥有正质量(positive mass)，每个模拟帧都会更新其世界变换。
2. 静态刚体(static bodies)：在 Bullet 里面设置的时候设置为 0 质量，不能移动，但是可以碰撞并与其他刚体交互。
3. 运动的刚体(kinematic bodies)：这个与动态刚体不是一个概念。在 Bullet 里面设置的时候设置为 0 质量，但是它可以**被用户控制着运动**，而且动力学交互也是单向的：运动刚体可以推动动态刚体，但是运动刚体本身却不会被影响。这个其实比较有用，运动刚体可以是制作好了骨骼动画的角色，他们可以对其他动态刚体施加影响。GTA5 里面的角色就可以直接撞到门上并推开门，这个角色就是运动刚体，门就是动态刚体。

5.1.3 质心的世界变换

Bullet 刚体的世界变换总是等于它的质心的变换[9]。刚体的局部惯性张量(local inertia tensor)取决于刚体形状，而 btCollisionShape 就提供了接口计算局部惯性(local inertia)。注意 Bullet 会自动计算刚体的质心，然后把局部坐标系的原点对齐至质心。

5.1.4 惯性张量(inertia tensor)

刚体动力学里面，刚体的运动一般分为平移和旋转。平动我们是比较熟悉的，计算起来相对也比较 trivial。旋转部分可以稍微科普一点概念，参考[21]。

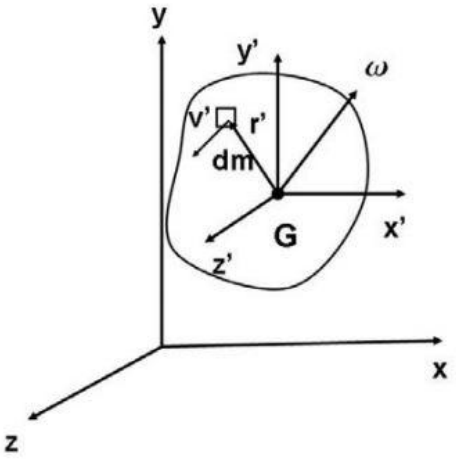


图 5-1：物体的角动量

首先，对于一个绕点 G 旋转的物体而言，其角动量(angular momentum) H_G 可以通过下面的公式计算：

$$H_G = \int_m \mathbf{r}' \times \mathbf{v}' dm$$

其中 ω' 为考察点的局部角速度，v' 为考察点的局部线速度。我们继续在质心坐标系的局部空间中考察问题，则有：

$$\mathbf{v}' = \boldsymbol{\omega} \times \mathbf{r}'$$

然后：

$$H_G = \int_m \mathbf{r}' \times (\boldsymbol{\omega} \times \mathbf{r}') dm = \int_m [(\mathbf{r}' \cdot \mathbf{r}')\boldsymbol{\omega} - (\mathbf{r}' \cdot \boldsymbol{\omega})\mathbf{r}'] dm$$

这里用到了向量恒等式：

$$A \times (B \times C) = (A \times C)B - (A \times B)C$$

之后我们如果用笛卡尔坐标系来表示 ω 和 r'，即：

$$\mathbf{r}' = x'\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

$$\boldsymbol{\omega}' = \omega_x \mathbf{i} + \omega_y \mathbf{j} + \omega_z \mathbf{k}$$

之后角动量可以被化为：

$$H_G = (I_{xx}\omega_x - I_{xy}\omega_y - I_{xz}\omega_z)\mathbf{i}$$

$$+ (I_{yx}\omega_x - I_{yy}\omega_y - I_{yz}\omega_z)\mathbf{j}$$

$$+ (I_{zx}\omega_x - I_{zy}\omega_y - I_{zz}\omega_z)\mathbf{k}$$

其中 I_{xx}, I_{yy}, I_{zz} 为惯性矩(moments of inertia)，就是绕 x,y,z 轴旋转的转动惯量。

$I_{xy}, I_{xz}, I_{yx}, I_{yz}, I_{zx}, I_{zy}$ 为惯性积（product of inertia），可正可负，用于衡量物理质量分布的不平衡性。

惯性张量(inertia tensor)是基于上面的量定义的，可以写成 3x3 的矩阵：

$$\mathbf{I}_G = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{bmatrix}$$

惯性张量从某些角度给了我们一些刚体质量分布的信息。参考[21]，主要到惯性张量矩阵的对称位置的惯性积相等，所以这个 3x3 矩阵是一个实对称矩阵，可以用特征分解(eigen decomposition)对它进行对角化，从而找到惯性主轴(principal axis)。

5.1.5 MotionState

为了把物体的世界变换同步到渲染节点，一般的游戏循环的流程是：

1. 遍历所有的物理刚体
2. 同步物理的世界变换到相应渲染节点

但是 Bullet 用一个叫 MotionState 的东西来节省了用户的工作量，省了物理模拟与变换同步的工作，而直接可以获取到相应物体的世界变换。

```
class btMotionState
{
public:
	virtual ~btMotionState(){}
	virtual void getWorldTransform(btTransform& worldTrans )
const =0;
	virtual void setWorldTransform(const btTransform&
worldTrans)=0;

};
```

这是 **btMotionState** 接口类，让 **btDynamicsWorld** 对物体的世界变换进行同步(synchronize)和插值(interpolation)，使得渲染时能很方便地更新物理模拟。为了优化性能，Bullet 仅对潜在移动的物体进行更新和同步。

5.1.6 模拟帧与插值帧

5.1.5 提到刚体运动状态的插值会通过 **btMotionState** 接口来进行。之所以会有插值这种操作，是因为 Bullet 是默认运行在固定的 60Hz（帧间间隔时间 0.0166666s，这个固定的模拟频率可以更改），但是游戏本身运行的帧率可能会不断变化。这是一开始有点出乎笔者的意料的，但是仔细想想又有点道理，毕竟还是要内部定一个足够细的模拟时间步长才可以较为真实地模拟刚体动力学。不然万一外部传入的模拟时间步长太过大，例如 1000ms，那这 1000ms 中间可能会发生的很多细节物理交互可能会被忽略，毕竟这个是离散的动力学模拟。所以用户 stepSimulation()传入一个 dt 以后，Bullet 会适当地对物理模拟帧的刚体世界变换进行插值，而不是重新模拟一次。

在刚体一开始被创建之后，初始的世界变换可以就可以通过 `virtual void btMotionState::getWorldTransform (btTransform& worldTrans)`获得。

5.2 Bullet 刚体动力学模拟代码概要分析

这个小主题其实是一个巨坑，因为 Bullet 涉及了太多的论文里的算法/解算器。所以笔者在这一小节就从 **btDiscreteDynamicsWorld** 的 stepSimulation()出发尝试进行一定程度上的分析。下面贴上 `btDiscreteDynamicsWorld::stepSimulation(...)`的代码，这是单步模拟的驱动函数，我们尝试对其进行稍微深入的分析。

5.2.1 单步模拟 stepSimulation

```
int	btDiscreteDynamicsWorld::stepSimulation( btScalar timeStep,int maxSubSteps, btScalar fixedTimeStep)
{
	startProfiling(timeStep);
	int numSimulationSubSteps = 0;
	if (maxSubSteps)
	{
		//fixed timestep with interpolation
		m_fixedTimeStep = fixedTimeStep;
		m_localTime += timeStep;
		if (m_localTime >= fixedTimeStep)
		{
```

```
                numSimulationSubSteps = int( m_localTime / fixedTimeStep);
                m_localTime -= numSimulationSubSteps * fixedTimeStep;
            }
        } else
        {
            //variable timestep
            fixedTimeStep = timeStep;
            m_localTime = m_latencyMotionStateInterpolation ? 0 : timeStep;
            m_fixedTimeStep = 0;
            if (btFuzzyZero(timeStep))
            {
                numSimulationSubSteps = 0;
                maxSubSteps = 0;
            } else
            {
                numSimulationSubSteps = 1;
                maxSubSteps = 1;
            }
        }
        //process some debugging flags
        if (getDebugDrawer())
        {
            btIDebugDraw* debugDrawer = getDebugDrawer ();
            gDisableDeactivation = (debugDrawer->getDebugMode() & btIDebugDraw::DBG_NoDeactivation) != 0;
        }
        if (numSimulationSubSteps)
        {
            //clamp the number of substeps, to prevent simulation grinding spiralling down to a halt
            int clampedSimulationSteps = (numSimulationSubSteps > maxSubSteps)? maxSubSteps : numSimulationSubSteps;
            saveKinematicState(fixedTimeStep*clampedSimulationSteps);
            applyGravity();
            for (int i=0;i<clampedSimulationSteps;i++)
            {
                internalSingleStepSimulation(fixedTimeStep);
                synchronizeMotionStates();
            }
        } else
        {
            synchronizeMotionStates();
        }
        clearForces();
#ifndef BT_NO_PROFILE
        CProfileManager::Increment_Frame_Counter();
#endif //BT_NO_PROFILE
        return numSimulationSubSteps;
}
```

从 stepSimulation()的第一层代码来看，它大致干了下面这些事：

- 设置性能分析(profiling)起始标记
- 累加 m_localTime，计算当前步要模拟的子步数(sub-step)。注意子步数会被传入的参数 int maxSubSteps 限制/截断(clamp)。笔者在这有个小疑问，这意味着如果传入 stepSimulation 的时间步长太大的话，物理世界的时间流逝会被硬生生砍掉一部分？
- 储存运动状态(kinematic state)，即计算下一帧的物体运动状态
- 对刚体施加重力
- 对于每一个 Bullet 内部固定步长的子步，执行：
  - 单步模拟
  - 运动状态同步
- 返回实际已模拟的步数，退出

但是仔细分析一下发现其实更多细节工作是藏在调用链的更底层的。那么我们来看看每一个物理模拟子步做的事。先看看单步模拟：

```
void    btDiscreteDynamicsWorld::
internalSingleStepSimulation(btScalar timeStep)
{
    BT_PROFILE("internalSingleStepSimulation");
    ...
    ///apply gravity, predict motion
    predictUnconstraintMotion(timeStep);
    btDispatcherInfo& dispatchInfo = getDispatchInfo();
    dispatchInfo.m_timeStep = timeStep;
    dispatchInfo.m_stepCount = 0;
    dispatchInfo.m_debugDraw = getDebugDrawer();
   createPredictiveContacts(timeStep);
    ///perform collision detection
    performDiscreteCollisionDetection();
    calculateSimulationIslands();
    getSolverInfo().m_timeStep = timeStep;
    ///solve contact and other joint constraints
    solveConstraints(getSolverInfo());
    ///CallbackTriggers();
    ///integrate transforms
    integrateTransforms(timeStep);
    ///update vehicle simulation
    updateActions(timeStep);
    updateActivationState( timeStep );
   ...
}
```

每一个 Bullet 内部固定步长的子步的物理模拟过程大致干了下面这些事：

- 设置性能分析(profiling)起始标记
- 根据时间步长 dt 预测**无约束**的、非静态的刚体运动状态。注意 Bullet 默认给动态物体加了阻尼(damping)，所以预测刚体下一帧世界变换的时候是先施加阻尼的。最后累加预测的变换(integrate transformation)(线速度累加到位移，角速度累加到旋转姿态)
- 创建预测性的接触信息(?)(create predictive contact)。这是根据当前帧到下一帧的变换对凸体进行了粗略阶段的(broad phase)的连续碰撞检测(CCD，Continuous Collision Detection)，或者说是扫掠测试(sweep test)。
- 执行离散碰撞检测(discrete collision detection)。更新所有刚体的 AABB，用分离轴定理寻找重叠对(overlapping pair)，在粗略阶段找出可能相交的刚体对。然后用不同的碰撞检测策略执行碰撞检测（详见第 4.3）。
- 计算模拟岛(?)(calculate simulation islands)。模拟岛的概念在 Bullet 文档与注释中并没有解释（实在是服了 Bullet 的稀缺文档）。代码里：

```
btAlignedObjectArray<btPersistentManifold*>
    m_predictiveManifolds;
```

的每一个“预测流形”都被遍历了一遍。这些 predictive manifold 里面都缓存着至多 4 个接触点(contact point)，以及产生这个接触的碰撞物体指针 btCollisionObject*。
- 求解约束(constraint)
- 累加变换(integrate transformation)
- 更新一些高阶模块的模拟，如汽车(vehicle)

5.2.2 阻尼(damping)

在单步模拟里面，Bullet 先预测了在无约束、有阻尼的情况下刚体的运动状态。

阻尼（damping）是指任何震动系统在振动中，由于外界作用或系统本身固有的原因引起的震动幅度逐渐下降的特性，以及此一特性的量化表征。

阻尼的物理意义是**力的衰减**,或**物体在运动中的能量耗散**。通俗地讲，就是阻止物体继续运动。当物体受到外力作用而振动时,会产生一种使外力衰减的反力,称为阻尼力(或减震力）。它和作用力的比被称为阻尼系数。通常阻尼力的方向总是和运动的速度方向相反。

在 Bullet 里面，增加阻尼是为了防止物理无休止地运动下去，虽然阻尼系数是可以调整的，阻尼系数越小，能量衰减得越慢。Bullet 里面的阻尼效果分为**指数衰减**与**线性衰减**部分。

```
void  btRigidBody::applyDamping(btScalar timeStep){
.....
      m_linearVelocity *= btPow(btScalar(1)-m_linearDamping,
timeStep);
      m_angularVelocity *=
btPow(btScalar(1)-m_angularDamping, timeStep);
.....
}
```

如上面代码所示，在 Bullet 里面，阻尼的指数衰减部分会被施加在刚体的线速度和角速度上，即：

$$v_{t+1} = (1-\alpha)^{\Delta t} v_t$$
$$\omega_{t+1} = (1-\beta)^{\Delta t} \omega_t$$

其中 α, β 分别为线速度和角速度的阻尼系数。

```
void  btRigidBody::applyDamping(btScalar timeStep){
.....
btScalar speed = m_linearVelocity.length();
if (speed < m_linearDamping)
{
btScalar dampVel = btScalar(0.005);
if (speed > dampVel)
{
      btVector3 dir =m_linearVelocity.normalized();
      m_linearVelocity -=    dir * dampVel;
} else
{
      m_linearVelocity.setValue(btScalar(0.),btScalar(0.),btScalar(
0.));
      }
}
...
}
```

Bullet 线性衰减阻尼是这样子工作的：如果线速度在区间(dampVel，m_linearDamping)里面，那么就给线速度降低一个常量。另外，如果线速度绝对值小于阈值(dampVel)，就设为 0，让刚体静止下来。

角速度阻尼的线性衰减部分同理。

5.2.3 旋转的参数化：四元数的指数映射

在累加变换那一步里，Bullet 的实现采用了一种笔者不太熟悉的旋转参数化算法**四元数指数映射参数化**(Exponential Map)，参考[22]。

平移变换的累加是比较简单的，所以平移变换 Bullet 直接就：

$$s += v \cdot dt$$

旋转的累加就稍微复杂一点点，因为 Bullet 用了一种比较有趣的旋转参数化方法——四元数指数映射。其实用什么参数化的方法取决于具体应用的性质。包括欧拉角在内，不少旋转的参数化形式是非欧的[22](non-Euclidean)，通俗来说就是往某个方向走足够长的距离，总会走回到原点。所有对一个 3-DOF 旋转的全集用一个欧式空间开子集来参数化的尝试，都是徒劳的。例如欧拉角就是这么个例子，万向锁(gimbal lock)就是其参数空间中的奇点(singularity)。

在 R^3 中每个非零向量都由方向和长度组成。 而旋转是由物体沿着一个旋转轴的旋转量组成的。 如果我们扩展零向量为 0 模长，单位旋转的话，整个指数映射就成为了连续空间的映射。 与四元数参数化不同的是，由于指数映射参照欧几里德空间，所以也存在万向节死锁的奇点空间，但这个奇点的坐标区间非常远离我们工作区间，并且参数化结果包含了大多数四元数参数结果的特性，并且无需担心会掉进非欧几里德流形之中。

在介绍四元数的指数映射之前简单说说四元数的一些几何意义的对应。

1. $\mathbf{q} = [0,0,0,1]^T$ 是单位旋转四元数：

2. $\mathbf{q} = [q_x, q_y, q_z, q_w]^T$ 表示绕单位轴：

$$\mathbf{v}_{unit} = \frac{1}{\sin(\cos^{-1}\theta)}[q_x, q_y, q_z]^T$$

旋转 $\theta = 2\cos^{-1} q_w$。

3. 绕单位轴 $\mathbf{v}_{unit}$ 旋转角度 θ 的单位四元数可以这样构造：

$$\mathbf{q} = [q_x, q_y, q_z, q_w]^T = \left[\sin(\frac{\theta}{2})\mathbf{v}_{unit}, \cos(\frac{\theta}{2})\right]^T$$

4. 引入四元数的指数映射，我们可以把一个非单位长度的三维向量 R^3 映射到 S^3 [22]，公式如下：

$$\mathbf{q} = e^{\mathbf{v}} = \left[\frac{\sin(\frac{\theta}{2})}{\theta}\mathbf{v}, \cos(\frac{\theta}{2})\right]^T$$

[22]中为了避免除 0 的情况，在考虑了计算机浮点精度之后，把四元数的向量部分进行了泰勒展开(Taylor Expansion)，取了最前面两项，即在 $\theta < \sqrt[4]{machine\_precision}$ 的情况下：

$$\frac{\sin(\frac{\theta}{2})}{\theta}=\frac{1}{\theta}\left(\frac{\theta}{2}+\frac{\left(\frac{\theta}{2}\right)^3}{3!}+O(\theta^4)\right)$$
$$\approx\frac{1}{\theta}\left(\frac{\theta}{2}+\frac{\theta^3}{48}\right)$$

这就是 Bullet 里面一段匪夷所思的代码的出处：

```
class btTransformUtil
{
public:
    static void integrateTransform(const btTransform&
curTrans,const btVector3& linvel,const btVector3&
angvel,btScalar timeStep,btTransform& predictedTransform)
{
....
if ( fAngle < btScalar(0.001) )
{
// use Taylor's expansions of sync function
axis = angvel* ( btScalar(0.5) *timeStep
-(timeStep*timeStep*timeStep)
*(btScalar(0.020833333333))*fAngle*fAngle );
}
else
{
axis = angvel* ( btSin(btScalar(0.5)*fAngle*timeStep)/fAngle );
}
btQuaternion dorn (axis.x(),axis.y(),axis.z(),
btCos( fAngle*timeStep*btScalar(0.5) ));
btQuaternion orn0 = curTrans.getRotation();
btQuaternion predictedOrn = dorn * orn0;
.....
}
```

那个奇怪的 0.2083333 就是四十八分之一，泰勒级数第二项的系数。于是这段 integrateTransform()里面的代码的做的事就比较明显了：把角速度用四元数指数映射转为四元数，然后乘到原来的姿态四元数上施加旋转变换。这个小数学计较，Bullet 的文档如下：

```
//Exponential map
//google for "Practical Parameterization of Rotations Using the
Exponential Map", F. Sebastian Grassia
```

呵呵:)。

5.3 约束(Constraint)

约束可以说是 Bullet 刚体动力学模拟里面的非常重要的一环了。Bullet 实现了很多种的约束，每种**约束都限制了刚体的某一些自由度**，这取决于具体的实现。而且约束会影响刚体的下一帧的预测的世界变换，所以在 stepSimulation()里面，Bullet 先预测了下一帧的刚体世界变换，再进行约束的解算(solve constraint)。每一个 Bullet 的约束都继承自 **btTypedConstraint**。

由于约束是作用在两个物体之间的，所以至少要有一个物体是动态物体。

下面先简单介绍一下 Bullet 实现的几种常见约束。

5.3.1 点到点约束

点到点约束(point to point constraint)为了把两个刚体各自的锚点(pivot poibt)在世界空间中对齐而对位移作出了限制。可以用这类型的约束来把一系列的刚体连成一条链子。

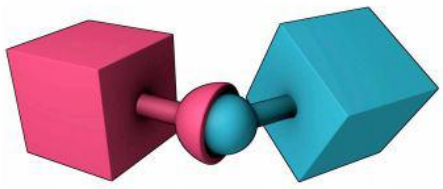


图 5-2：点到点约束

```
ATTRIBUTE_ALIGNED16(class) btPoint2PointConstraint : public
btTypedConstraint
{.....}
```

5.3.2 铰链约束

铰链约束(hinge constraint)，又叫旋转关节约束(revolute joint constraint)，限制了两个角度自由度，添加了铰链约束的刚体只能绕着铰链轴旋转（hinge axis）。铰链约束可以用来模拟门、轮子等对象（！！）。用户可以指定铰链约束的限制量和动力。

```
ATTRIBUTE_ALIGNED16(class) btHingeConstraint : public
btTypedConstraint{.....}
```

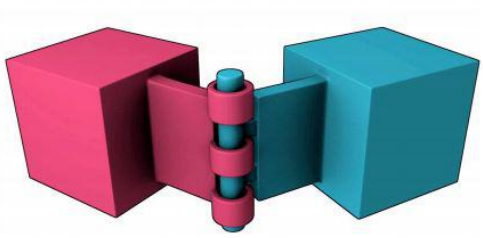


图 5-3：铰链约束

5.3.3 滑动约束

滑动约束(slider constraint)允许刚体沿着某条轴旋转(1-DOF)及平移。

```
ATTRIBUTE_ALIGNED16(class) btSliderConstraint : public btTypedConstraint{.....}
```

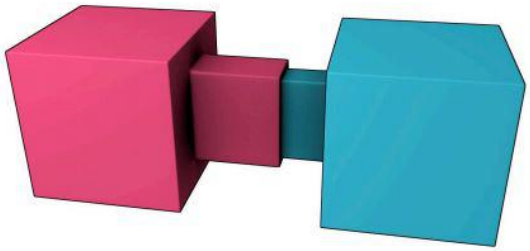


图 5-4：滑动约束

5.3.4 通用 6-DOF 约束

Bullet 提供一种通用的 6-DOF 约束，理论上用户可以用这一种通用的约束来模拟其他约束。前 3 个 DOF 是平移的自由度，后 3 个 DOF 是旋转的自由度，各用一条轴来表示。两条轴（6-DOF）中，每条轴都有三种状态：

1. 锁定(locked)
2. 自由(free)
3. 被限制的(limited)

```
ATTRIBUTE_ALIGNED16(class) btGeneric6DofConstraint : public btTypedConstraint{
.....
	int setAngularLimits(btConstraintInfo2 *info, int row_offset,const btTransform& transA,const btTransform& transB,const btVector3& linVelA,const btVector3& linVelB,const btVector3& angVelA,const btVector3& angVelB);

	int setLinearLimits(btConstraintInfo2 *info, int row, const btTransform& transA,const btTransform& transB,const btVector3& linVelA,const btVector3& linVelB,const btVector3& angVelA,const btVector3& angVelB);
.....
}
```

5.3.5 约束解算

这一部分其实也是比较有技术含量的一部分。但是限于时间与篇幅，笔者无法展开过多，望见谅。

下面是 stepSimulation()里面调用的解算约束的函数：

```
Void btDiscreteDynamicsWorld::solveConstraints (btContactSolverInfo& solverInfo)
{
....
	m_sortedConstraints.quickSort(btSortConstraintOnIslandPredicate());

	btTypedConstraint** constraintsPtr = getNumConstraints() ? &m_sortedConstraints[0] : 0;

	m_solverIslandCallback->setup(&solverInfo,constraintsPtr, m_sortedConstraints.size(),getDebugDrawer());

	m_constraintSolver->prepareSolve(getCollisionWorld()->getNumCollisionObjects(), getCollisionWorld()->getDispatcher()->getNumManifolds());
	/// solve all the constraints for this island
	m_islandManager->buildAndProcessIslands(getCollisionWorld()->getDispatcher(),getCollisionWorld(),m_solverIslandCallback);

	m_solverIslandCallback->processConstraints();

	m_constraintSolver->allSolved(solverInfo, m_debugDrawer);
}
```

最后发现 Bullet 的动力学模拟的精华都在约束解算器(constraint solver)里面了。Bullet 实现了很多的约束解算器，但是文档极其稀缺,注释也极少，如果没有原始论文或者文档支撑的话，代码可读性会变得非常差，工作量将会及其庞大。

Bullet 的约束解算器都要继承自 **btConstraintSolver** 接口：

```
class btConstraintSolver
{
public:
	virtual ~btConstraintSolver() {}
	virtual void prepareSolve (int /* numBodies */, int /* numManifolds */) {;}

	///solve a group of constraints
	virtual btScalar solveGroup(...) = 0;
	virtual void allSolved (const btContactSolverInfo& /* info */,class btIDebugDraw* /* debugDrawer */) {;}
	///clear internal cached data and reset random seed
	virtual	void	reset() = 0;
	virtual btConstraintSolverType	getSolverType() const=0;
};
```

在 Bullet 里面，约束解算是通过先求解每个刚体的约束力，再分别预测每个刚体的线速度和角速度，来实现“约束”的效果[25]。其中约束使用线性方程组的形式来表示的。由于这个一个大坑，这里无法继续展开，

留待日后研究。相关问题与概念：

- 雅克比矩阵(Jacobian Matrix)
- 线性互补问题(Linear Complementary Problem,LCP)
- 混合线性互补问题 MLCP[26]
- 用于求解 LCP 的 Lemke 算法[24]
- Gauss-Seidel 迭代法求解线性方程组[28]

更全面的刚体动力学模拟系统的全面介绍文献可以参考[20][27]。

6 结论

Bullet 是个好东西，作为世界三大物理引擎之一居然是开源的，这种精神值得称赞。由于 Bullet 是一个功能强大，代码库也非常庞大的刚体物理动力学模拟引擎，本文只能简单地分析了一下 Bullet 的设计、组成以及部分细节的算法与数学，如果分析有不周到，望读者见谅。但是经过这么一轮分析，笔者对 Bullet 物理引擎有了一定的认识。

Bullet 有它的优点：

1. Bullet 的技术含量高，无论是几何数学库、碰撞检测库、约束解算器、IK 解算器等都涉及了非常多的数学和算法，不少算法都有其原始论文，这是分析代码工作量巨大的原因之一。碰撞检测库就涉及到了 GJK-EPA 凸体求交算法、约束解算器里面涉及到很多线性代数与数值计算的方法。

2. Bullet 的项目 build 非常方便，没什么依赖库。而且有 ExampleBrowser，demo 运行起来比较舒服。

但是 Bullet 也有它的缺点：

1. 文档和注释实在是非常少。很多很有技术含量、涉及数学的算法里面，情况好的就给出论文的引用，不好的甚至就不解释了。因为注释也是非常的少，所以对于复杂算法的阅读比较吃力。

2. 多态与继承应用广泛，但文档却跟不上。这本质上也是上一个问题。很多类继承链非常长，这个可以通过看 doxygen 生成的文档缓解一下。但是有很多地方都用了接口类与多态，用 IDE 的“转到定义”功能只能直接跳到接口的定义，而具体的实现还要自己另外分析（唔其实这算不算确定还有待商榷，毕竟接口类在一些方面来讲也是比较清晰的）

3. 代码风格：注释少真的是很严重的问题！其次是 bullet 解决方案里面有很多项目，有些项目的引用了一些公用的、其他项目的.h/.cpp，这个感觉比较奇怪。

总之，Bullet 库还是很不错的，但是笔者感觉物理系统这是个游戏开发里面的大坑，要填上并不容易，这得日后继续加油了！

引用

[1] 维基百科:rigid body - https://en.wikipedia.org/wiki/Rigid_body
[2] Jason Gregory 著,叶劲峰译 - 游戏引擎架构[M]. 北京：电子工业出版社，2014.2
[3] 百度百科:ODE - https://baike.baidu.com/item/ODE/2510406?fr=aladdin
[4] 百度百科:Bullet - https://baike.baidu.com/item/Bullet/8198766
[5] 百度百科:PhysX - https://baike.baidu.com/item/PhysX/9272519?fr=aladdin
[6] 百度百科:Havok - https://baike.baidu.com/item/Havok/5570248?fr=aladdin
[7] Panda3D 用户手册- http://www.panda3d.org/manual/index.php/Physics
[8] Panda3D 用户手册- http://www.panda3d.org/manual/index.php/Bullet_Hello_World
[9] Bullet 2.83 Physics SDK Manual
[10] csdn:分离轴理论(一) - https://blog.csdn.net/u011373710/article/details/54773171
[11] twisted oak:minkowski sums and difference - http://twistedoakstudios.com/blog/Post554_minkowski-sums-and-differences
[12] csdn:判断两个形状是否相交(二)-GJK - https://blog.csdn.net/u011373710/article/details/54773174
[13] http://www.dyn4j.org/2010/04/gjk-gilbert-johnson-keerthi/#gjk-top
[14] http://www.dyn4j.org/2010/05/epa-expanding-polytope-algorithm/
[15] http://khaledmammou.com/hacd.html
[16] http://blog.sina.com.cn/s/blog_559ae0ae0100y4v9.html
[17] cnblog:刚体动力学 - https://www.cnblogs.com/tekkaman/p/3647566.html

[18] Lien J M, Amato N M. Approximate convex decomposition of polygons[J]. Computer Aided Geometric Design, 2008, 25(7):503-522.
[19] Roy Featherstone, Rigid Body Dynamics Algorithm[M]. Springer US, 2008
[20] Mirtich B. Impulse-based Dynamic Simulation of Rigid Body Systems[C]// Symposium on Interactive 3d Graphics. 1996.
[21]https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-07-dynamics-fall-2009/lecture-notes/MIT16_07F09_Lec26.pdf
[22] Grassia F S. Practical Parameterization of Rotations Using the Exponential Map[J]. Journal of Graphics Tools, 1998, 3(3):29-48.
[23] https://en.wikipedia.org/wiki/Quaternion#Exponential.2C_logarithm.2C_and_power
[24] Lloyd J E. Fast Implementation of Lemke’s Algorithm for Rigid Body Contact Simulation[C]// IEEE International Conference on Robotics and Automation. IEEE, 2006:4538-4543.
[25] https://www.zhihu.com/question/40135989/answer/84937500
[26] MLCP - http://chrishecker.com/The_Mixed_Linear_Complementarity_Problem
[27] Catto E. Iterative dynamics with temporal coherence[J]. 2005.
[28] https://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel_method