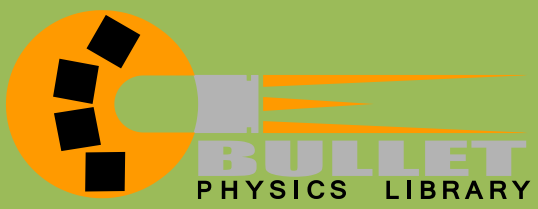
Bullet 2.83 Physics SDK Manual



Also check out the forums and wiki at bulletphysics.org

Table of Contents

BULLET 2.83 PHYSICS SDK MANUAL **1**

1 Introduction *4*
Description of the library 4
Main Features 4
Contact and Support 4
2 Build system, Getting started and What's New *5*
Please see separate BulletQuickstart.pdf guide. 5
3 Library Overview *6*
Introduction 6
Software Design 6
Rigid Body Physics Pipeline 7
Integration overview 7
Basic Data Types and Math Library 9
Memory Management, Alignment, Containers 9
Timing and Performance Profiling 10
Debug Drawing 11
4 Bullet Collision Detection *12*
Collision Detection 12
Collision Shapes 13
Convex Primitives 13
Compound Shapes 14
Convex Hull Shapes 14
Concave Triangle Meshes 14
Convex Decomposition 14
Height field 15
btStaticPlaneShape 15
Scaling of Collision Shapes 15
Collision Margin 15
Collision Matrix 16
Registering custom collision shapes and algorithms 16
5 Collision Filtering (selective collisions) *17*
Filtering collisions using masks 17
Filtering Collisions Using a Broadphase Filter Callback 17
Filtering Collisions Using a Custom NearCallback 18
Deriving your own class from btCollisionDispatcher 18
6 Rigid Body Dynamics *19*
Introduction 19
Static, Dynamic and Kinematic Rigid Bodies 19
Center of mass World Transform 20
What's a MotionState? 20
Interpolation 21
So how do I use one? 21
`DefaultMotionState` 21
Kinematic Bodies 22
Simulation frames and interpolation frames 22
7 Constraints *23*
Point to Point Constraint 23
Hinge Constraint 23
Slider Constraint 24
Cone Twist Constraint 24
Generic 6 Dof Constraint 24
8 Actions: Vehicles & Character Controller *26*

Action Interface 26
Raycast Vehicle 26
Character Controller 26
9 Soft Body Dynamics 27
Introduction 27
Construction from a triangle mesh 27
Collision clusters 27
Applying forces to a Soft body 28
Soft body constraints 28
10 Bullet Example Browser 29
BSP Demo 30
Vehicle Demo 30
Fork Lift Demo 30
11 Advanced Low Level Technical Demos 31
Collision Interfacing Demo 31
Collision Demo 31
User Collision Algorithm 31
Gjk Convex Cast / Sweep Demo 31
Continuous Convex Collision 31
Raytracer Demo 31
Simplex Demo 32
12 Authoring Tools and Serialization 33
Dynamica Maya Plugin 33
Blender 33
Cinema 4D, Lightwave CORE, Houdini 33
Serialization and the Bullet .bullet binary format 34
13 General Tips 35
Avoid very small and very large collision shapes 35
Avoid large mass ratios (differences) 35
Combine multiple static triangle meshes into one 35
Use the default internal fixed timestep 35
For ragdolls use btConeTwistConstraint 35
Don’t set the collision margin to zero 35
Use less then 100 vertices in a convex mesh 36
Avoid huge or degenerate triangles in a triangle mesh 36
The profiling feature btQuickProf bypasses the memory allocator 36
Per triangle friction and restitution value 37
Other MLCP Constraint Solvers 37
Custom Friction Model 37
14 Parallelism using OpenCL 38
OpenCL rigid body and collision detection. 38
15 Further documentation and references 39
Online resources 39
Authoring Tools 39
Books 39
Contributions and people 40

1 Introduction

Description of the library

Bullet Physics is a professional open source collision detection, rigid body and soft body dynamics library written in portable C++. The library is primarily designed for use in games, visual effects and robotic simulation. The library is free for commercial use under the ZLib license.

Main Features

- Discrete and continuous collision detection including ray and convex sweep test. Collision shapes include concave and convex meshes and all basic primitives
- Maximal coordinate 6-degree of freedom rigid bodies (btRigidBody) connected by constraints (btTypedConstraint) as well as generalized coordinate multi-bodies (btMultiBody) connected by mobilizers using the articulated body algorithm.
- Fast and stable rigid body dynamics constraint solver, vehicle dynamics, character controller and slider, hinge, generic 6DOF and cone twist constraint for ragdolls
- Soft Body dynamics for cloth, rope and deformable volumes with two-way interaction with rigid bodies, including constraint support
- Open source C++ code under Zlib license and free for any commercial use on all platforms including PLAYSTATION 3, XBox 360, Wii, PC, Linux, Mac OSX, Android and iPhone
- Maya Dynamica plugin, Blender integration, native binary .bullet serialization and examples how to import URDF, Wavefront .obj and Quake .bsp files.
- Many examples showing how to use the SDK. All examples are easy to browse in the OpenGL 3 example browser. Each example can also be compiled without graphics.
- Quickstart Guide, Doxygen documentation, wiki and forum complement the examples.

Contact and Support

- Public forum for support and feedback is available at http://bulletphysics.org

2 Build system, Getting started and What's New

Please see separate BulletQuickstart.pdf guide.

From Bullet 2.83 onwards there is a separate quickstart guide. This quickstart guide includes the changes and new features of the new SDK versions, as well as how to build the Bullet Physics SDK and the examples.

You can find this quickstart guide in `Bullet/docs/BulletQuickstart.pdf`.

3 Library Overview

Introduction

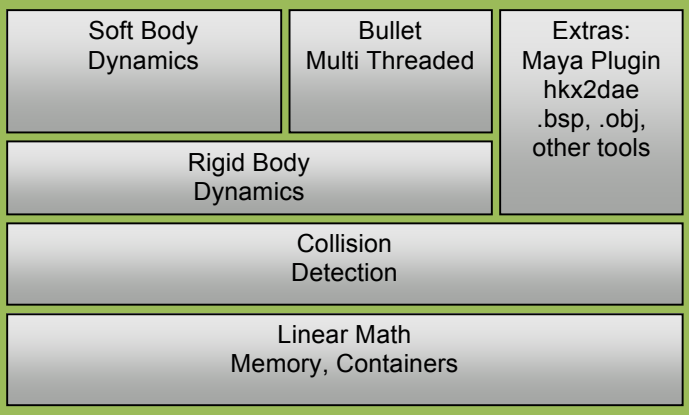
The main task of a physics engine is to perform collision detection, resolve collisions and other constraints, and provide the updated world transform[1] for all the objects. This chapter will give a general overview of the rigid body dynamics pipeline as well as the basic data types and math library shared by all components.

Software Design

Bullet has been designed to be customizable and modular. The developer can

- use only the collision detection component
- use the rigid body dynamics component without soft body dynamics component
- use only small parts of a the library and extend the library in many ways
- choose to use a single precision or double precision version of the library
- use a custom memory allocator, hook up own performance profiler or debug drawer

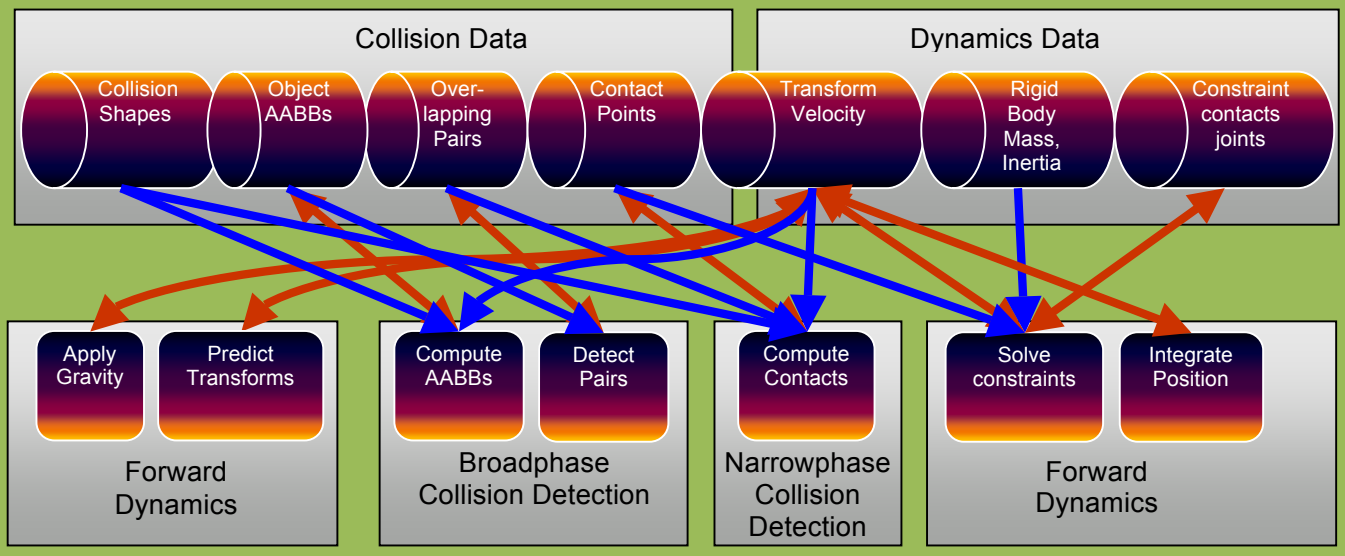
The main components are organized as follows:



[1] World transform of the center of mass for rigid bodies, transformed vertices for soft bodies

Rigid Body Physics Pipeline

Before going into detail, the following diagram shows the most important data structures and computation stages in the Bullet physics pipeline. This pipeline is executed from left to right, starting by applying gravity, and ending by position integration, updating the world transform.



The entire physics pipeline computation and its data structures are represented in Bullet by a dynamics world. When performing '`stepSimulation`' on the dynamics world, all the above stages are executed. The default dynamics world implementation is the *`btDiscreteDynamicsWorld`*.

Bullet lets the developer choose several parts of the dynamics world explicitly, such as broadphase collision detection, narrowphase collision detection (dispatcher) and constraint solver.

Integration overview

If you want to use Bullet in your own 3D application, it is best to follow the steps in the HelloWorld demo, located in `Bullet/examples/HelloWorld`. In a nutshell:

- Create a *`btDiscreteDynamicsWorld`* or *`btSoftRigidDynamicsWorld`*

These classes, derived from *`btDynamicsWorld`*, provide a high level interface that manages your physics objects and constraints. It also implements the update of all objects each frame.

- Create a *`btRigidBody`* and add it to the *`btDynamicsWorld`*

To construct a *`btRigidBody`* or *`btCollisionObject`*, you need to provide:

- Mass, positive for dynamics moving objects and 0 for static objects

- CollisionShape, like a Box, Sphere, Cone, Convex Hull or Triangle Mesh
- Material properties like friction and restitution

Update the simulation each frame:

- `stepSimulation`

Call the *`stepSimulation`* on the dynamics world. The *`btDiscreteDynamicsWorld`* automatically takes into account variable timestep by performing interpolation instead of simulation for small timesteps. It uses an internal fixed timestep of 60 Hertz. *`stepSimulation`* will perform collision detection and physics simulation. It updates the world transform for active objecs by calling the `btMotionState's setWorldTransform`.

The next chapters will provide more information about collision detection and rigid body dynamics. A lot of the details are demonstrated in the `Bullet/examples`. If you can't find certain functionality, please visit the physics forum on the Bullet website at http://bulletphysics.org

Basic Data Types and Math Library

The basic data types, memory management and containers are located in *Bullet/src/LinearMath*.

- *btScalar*

A *btScalar* is a posh word for a floating point number. In order to allow to compile the library in single floating point precision and double precision, we use the btScalar data type throughout the library. By default, *btScalar* is a typedef to *float*. It can be *double* by defining *BT_USE_DOUBLE_PRECISION* either in your build system, or at the top of the file *Bullet/src/LinearMath/btScalar.h*.

- *btVector3*

3D positions and vectors can be represented using btVector3. btVector3 has 3 scalar x,y,z components. It has, however, a 4th unused w component for alignment and SIMD compatibility reasons. Many operations can be performed on a *btVector3*, such as add subtract and taking the length of a vector.

- *btQuaternion* and *btMatrix3x3*

3D orientations and rotations can be represented using either *btQuaternion* or *btMatrix3x3*.

- *btTransform*

btTransform is a combination of a position and an orientation. It can be used to transform points and vectors from one coordinate space into the other. No scaling or shearing is allowed.

Bullet uses a right-handed coordinate system:

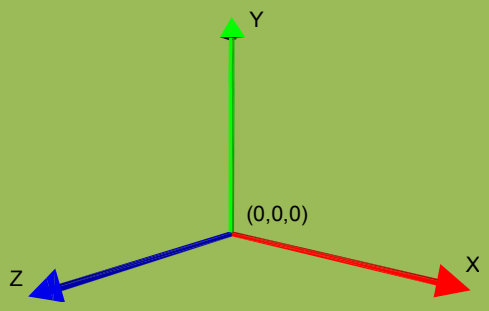


Figure 1 Right-handed coordinate system

btTransformUtil, *btAabbUtil* provide common utility functions for transforms and AABBs.

Memory Management, Alignment, Containers

Often it is important that data is 16-byte aligned, for example when using SIMD or DMA transfers on Cell SPU. Bullet provides default memory allocators that handle alignment, and developers can provide their own memory allocator. All memory allocations in Bullet use:

- *btAlignedAlloc*, which allows to specify size and alignment
- *btAlignedFree*, free the memory allocated by *btAlignedAlloc*.

To override the default memory allocator, you can choose between:

- *btAlignedAllocSetCustom* is used when your custom allocator doesn't support alignment
- *btAlignedAllocSetCustomAligned* can be used to set your custom aligned memory allocator.

To assure that a structure or class will be automatically aligned, you can use this macro:

- *ATTRIBUTE_ALIGNED16(type) variablename* creates a 16-byte aligned variable

Often it is necessary to maintain an array of objects. Originally the Bullet library used a STL std::vector data structure for arrays, but for portability and compatibility reasons we switched to our own array class.

- *btAlignedObjectArray* closely resembles std::vector. It uses the aligned allocator to guarantee alignment. It has methods to sort the array using quick sort or heap sort.

To enable Microsoft Visual Studio Debugger to visualize *btAlignedObjectArray* and *btVector3*, follow the instructions in `Bullet/msvc/autoexp_ext.txt`

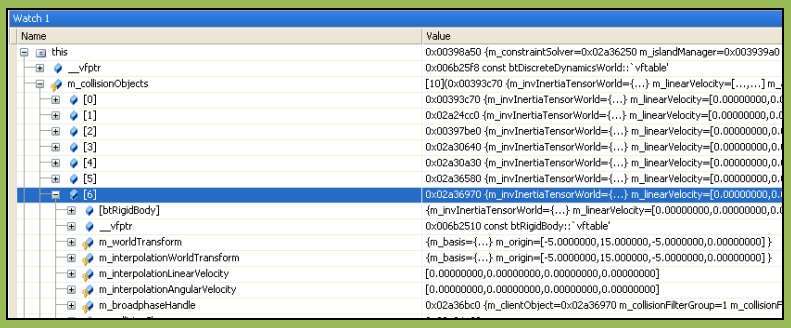


Figure 2 MSVC Debug Visualization

Timing and Performance Profiling

In order to locate bottlenecks in performance, Bullet uses macros for hierarchical performance measurement.

- `btClock` measures time using microsecond accuracy.
- *BT_PROFILE(section_name)* marks the start of a profiling section.

- *CProfileManager::dumpAll();* dumps a hierarchical performance output in the console. Call this after stepping the simulation.

- *CProfileIterator* is a class that lets you iterate through the profiling tree.

Note that the profiler doesn't use the memory allocator, so you might want to disable it when checking for memory leaks, or when creating a final release build of your software.

The profiling feature can be switched off by defining *#define BT_NO_PROFILE 1* in *Bullet/src/LinearMath/btQuickProf.h*

Debug Drawing

Visual debugging the simulation data structures can be helpful. For example, this allows you to verify that the physics simulation data matches the graphics data. Also scaling problems, bad constraint frames and limits show up.

btIDebugDraw is the interface class used for debug drawing. Derive your own class and implement the virtual '*drawLine*' and other methods.

Assign your custom debug drawer to the dynamics world using the `setDebugDrawer` method.

Then you can choose to draw specific debugging features by setting the mode of the debug drawer:

```
dynamicsWorld->getDebugDrawer()->setDebugMode(debugMode);
```

Every frame you can call the debug drawing by calling the

```
world-> debugDrawWorld();
```
[2]

Here are some debug modes

- `btIDebugDraw::DBG_DrawWireframe`
- `btIDebugDraw::DBG_DrawAabb`
- `btIDebugDraw::DBG_DrawConstraints`
- `btIDebugDraw::DBG_DrawConstraintLimits`

By default all objects are visualized for a given debug mode, and when using many objects this can clutter the display. You can disable debug drawing for specific objects by using

```
int f = objects->getCollisionFlags();
ob->setCollisionFlags(f|btCollisionObject::CF_DISABLE_VISUALIZE_OBJECT);
```

[2] This feature is supported for both `btCollisionWorld` and `btDiscreteDynamicsWorld`

4 Bullet Collision Detection

Collision Detection

The collision detection provides algorithms and acceleration structures for closest point (distance and penetration) queries as well as ray and convex sweep tests. The main data structures are:

- *btCollisionObject* is the object that has a world transform and a collision shape.
- *btCollisionShape* describes the collision shape of a collision object, such as box, sphere, convex hull or triangle mesh. A single collision shape can be shared among multiple collision objects.
- *btGhostObject* is a special *btCollisionObject*, useful for fast localized collision queries.
- *btCollisionWorld* stores all *btCollisionObjects* and provides an interface to perform queries.

The broadphase collision detection provides acceleration structure to quickly reject pairs of objects based on axis aligned bounding box (AABB) overlap. Several different broadphase acceleration structures are available:

- `btDbvtBroadphase` uses a fast dynamic bounding volume hierarchy based on AABB tree
- `btAxisSweep3` and `bt32BitAxisSweep3` implement incremental 3d sweep and prune
- *btSimpleBroadphase* is a brute force reference implementation. It is slow but easy to understand and useful for debugging and testing a more advanced broadphase.

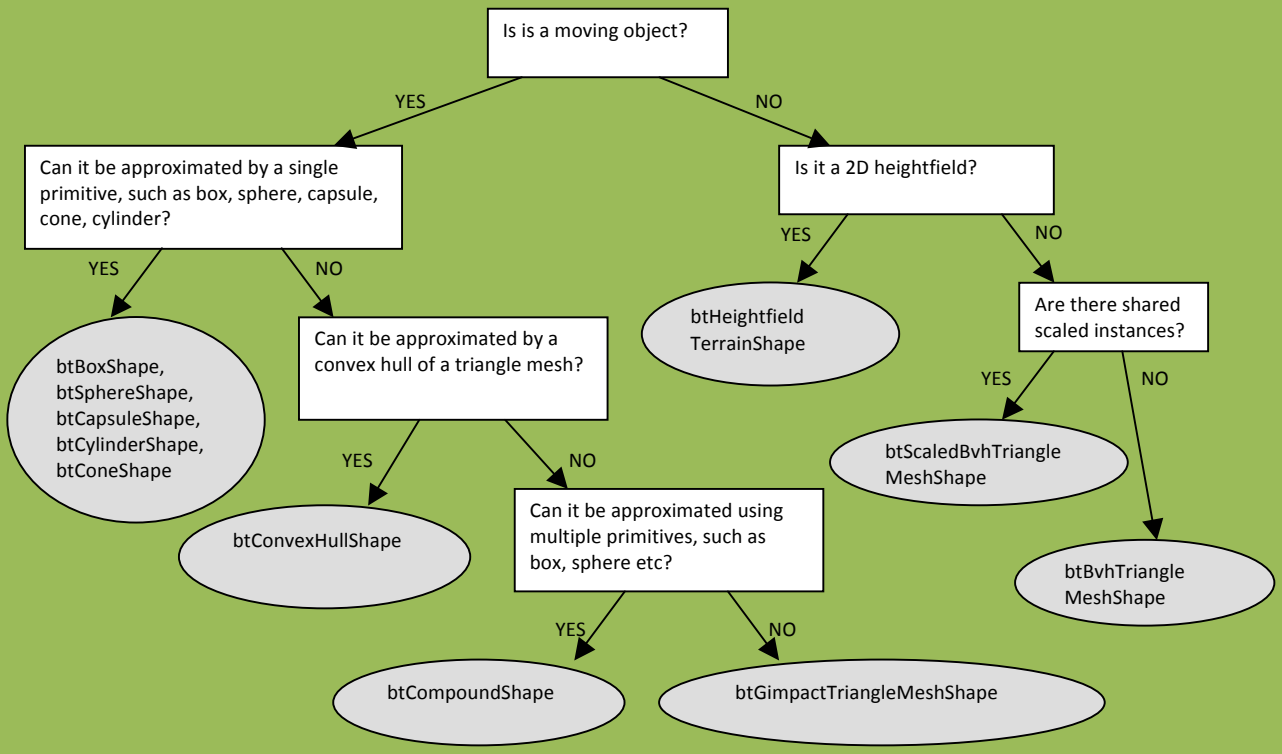
The broadphase adds and removes overlapping pairs from a pair cache. Overlapping pairs are persistent over time and can cache information such as previous contact constraint forces that can be used for 'warmstarting': using the previous solution to converge faster towards constraint solving.

A collision dispatcher iterates over each pair, searches for a matching collision algorithm based on the types of objects involved and executes the collision algorithm computing contact points.

- *btPersistentManifold* is a contact point cache to store contact points for a given pair of objects.

Collision Shapes

Bullet supports a large variety of different collision shapes, and it is possible to add your own. For best performance and quality it is important to choose the collision shape that suits your purpose. The following diagram can help making a decision:



Convex Primitives

Most primitive shapes are centered around the origin of their local coordinate frame:

`btBoxShape` : Box defined by the half extents (half length) of its sides

`btSphereShape` : Sphere defined by its radius

`btCapsuleShape`: Capsule around the Y axis. Also `btCapsuleShapeX/Z`

`btCylinderShape` : Cylinder around the Y axis. Also `btCylinderShapeX/Z`.

`btConeShape` : Cone around the Y axis. Also `btConeShapeX/Z`.

`btMultiSphereShape` : Convex hull of multiple spheres, that can be used to create a Capsule (by passing 2 spheres) or other convex shapes.

Compound Shapes

Multiple convex shapes can be combined into a composite or compound shape, using the `btCompoundShape`. This is a concave shape made out of convex sub parts, called child shapes. Each child shape has its own local offset transform, relative to the `btCompoundShape`. It is a good idea to approximate concave shapes using a collection of convex hulls, and store them in a `btCompoundShape`. You can adjust the center of mass using a utility method `btCompoundShape ::calculatePrincipalAxisTransform.`

Convex Hull Shapes

Bullet supports several ways to represent a convex triangle meshes. The easiest way is to create a `btConvexHullShape` and pass in an array of vertices. In some cases the graphics mesh contains too many vertices to be used directly as `btConvexHullShape`. In that case, try to reduce the number of vertices.

Concave Triangle Meshes

For static world environment, a very efficient way to represent static triangle meshes is to use a `btBvhTriangleMeshShape`. This collision shape builds an internal acceleration structure from a `btTriangleMesh` or `btStridingMeshInterface`. Instead of building the tree at run-time, it is also possible to serialize the binary tree to disc. See `examples/ConcaveDemo` how to save and load this `btOptimizedBvh` tree acceleration structure. When you have several instances of the same triangle mesh, but with different scaling, you can instance a `btBvhTriangleMeshShape` multiple times using the `btScaledBvhTriangleMeshShape.` The `btBvhTriangleMeshShape` can store multiple mesh parts. It keeps a triangle index and part index in a 32bit structure, reserving 10 bits for the part Id and the remaining 22 bits for triangle index. If you need more than 2 million triangles, either split the the triangle mesh into multiple sub meshes, or change the default in `#define MAX_NUM_PARTS_IN_BITS` in the file `src\BulletCollision\BroadphaseCollision\btQuantizedBvh.h`

Convex Decomposition

Ideally, concave meshes should only be used for static artwork. Otherwise its convex hull should be used by passing the mesh to `btConvexHullShape`. If a single convex shape is not detailed enough, multiple convex parts can be combined into a composite object called `btCompoundShape`. Convex decomposition can be used to decompose the concave mesh into several convex parts. See the `Demos/ConvexDecompositionDemo` for an automatic way of doing convex decomposition.

Height field

Bullet provides support for the special case of a flat 2D concave terrain through the `btHeightfieldTerrainShape`. See `examples/TerrainDemo` for its usage.

btStaticPlaneShape

As the name suggests, the `btStaticPlaneShape` can represent an infinite plane or half space. This shape can only be used for static, non-moving objects. This shape has been introduced mainly for demo purposes.

Scaling of Collision Shapes

Some collision shapes can have local scaling applied. Use `btCollisionShape::setScaling`(vector3). Non uniform scaling with different scaling values for each axis, can be used for `btBoxShape`, `btMultiSphereShape`, `btConvexShape`, `btTriangleMeshShape`. Uniform scaling, using x value for all axis, can be used for `btSphereShape`. Note that a non-uniform scaled sphere can be created by using a `btMultiSphereShape` with 1 sphere. As mentioned before, the `btScaledBvhTriangleMeshShape` allows to instantiate a btBvhTriangleMeshShape at different non-uniform scale factors. The `btUniformScalingShape` allows to instantiate convex shapes at different scales, reducing the amount of memory.

Collision Margin

Bullet uses a small collision margin for collision shapes, to improve performance and reliability of the collision detection. It is best not to modify the default collision margin, and if you do use a positive value: zero margin might introduce problems. By default this collision margin is set to 0.04, which is 4 centimeter if your units are in meters (recommended).

Dependent on which collision shapes, the margin has different meaning. Generally the collision margin will expand the object. This will create a small gap. To compensate for this, some shapes will subtract the margin from the actual size. For example, the `btBoxShape` subtracts the collision margin from the half extents. For a `btSphereShape`, the entire radius is collision margin so no gap will occur. Don't override the collision margin for spheres. For convex hulls, cylinders and cones, the margin is added to the extents of the object, so a gap will occur, unless you adjust the graphics mesh or collision size. For convex hull objects, there is a method to remove the gap introduced by the margin, by shrinking the object. See the `examples/Importers/ImportBsp` for this advanced use.

Collision Matrix

For each pair of shape types, Bullet will dispatch a certain collision algorithm, by using the dispatcher. By default, the entire matrix is filled with the following algorithms. Note that Convex represents convex polyhedron, cylinder, cone and capsule and other GJK compatible primitives. GJK stands for Gilbert, Johnson and Keerthi, the people behind this convex distance calculation algorithm. It is combined with EPA for penetration depth calculation. EPA stands for Expanding Polythope Algorithm by Gino van den Bergen. Bullet has its own free implementation of GJK and EPA.

	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

Registering custom collision shapes and algorithms

The user can register a custom collision detection algorithm and override any entry in this Collision Matrix by using the `btDispatcher::registerCollisionAlgorithm`. See `examples/UserCollisionAlgorithm` for an example, that registers a SphereSphere collision algorithm.

5 Collision Filtering (selective collisions)

Bullet provides three easy ways to ensure that only certain objects collide with each other: masks, broadphase filter callbacks and nearcallbacks. It is worth noting that mask-based collision selection happens a lot further up the toolchain than the callback do. In short, if masks are sufficient for your purposes, use them; they perform better and are a lot simpler to use.

Of course, don't try to shoehorn something into a mask-based selection system that clearly doesn't fit there just because performance may be a little better.

Filtering collisions using masks

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

```
int myGroup = 1;

int collideMask = 4;

world->addCollisionObject(object,myGroup,collideMask);
```

During broadphase collision detection overlapping pairs are added to a pair cache, only when the mask matches the group of the other objects (in `needsBroadphaseCollision`)

```
bool collides = (proxy0->m_collisionFilterGroup & proxy1->m_collisionFilterMask) != 0;

collides = collides && (proxy1->m_collisionFilterGroup & proxy0->m_collisionFilterMask);
```

If you have more types of objects than the 32 bits available to you in the masks, or some collisions are enabled or disabled based on other factors, then there are several ways to register callbacks to that implements custom logic and only passes on collisions that are the ones you want:

Filtering Collisions Using a Broadphase Filter Callback

One efficient way is to register a broadphase filter callback. This callback is called at a very early stage in the collision pipeline, and prevents collision pairs from being generated.

```
struct YourOwnFilterCallback : public btOverlapFilterCallback
{
        // return true when pairs need collision
        virtual bool        needBroadphaseCollision(btBroadphaseProxy* proxy0,btBroadphaseProxy* proxy1) const
        {
                bool collides = (proxy0->m_collisionFilterGroup & proxy1->m_collisionFilterMask) != 0;
                collides = collides && (proxy1->m_collisionFilterGroup & proxy0->m_collisionFilterMask);

                //add some additional logic here that modified 'collides'
                return collides;
        }
};
```

And then create an object of this class and register this callback using:

```
btOverlapFilterCallback * filterCallback = new YourOwnFilterCallback();
dynamicsWorld->getPairCache()->setOverlapFilterCallback(filterCallback);
```

Filtering Collisions Using a Custom NearCallback

Another callback can be registered during the narrowphase, when all pairs are generated by the broadphase. The *btCollisionDispatcher::dispatchAllCollisionPairs* calls this narrowphase nearcallback for each pair that passes the *'btCollisionDispatcher::needsCollision'* test. You can customize this nearcallback:

```
void MyNearCallback(btBroadphasePair& collisionPair,
  btCollisionDispatcher& dispatcher, btDispatcherInfo& dispatchInfo) {

    // Do your collision logic here
    // Only dispatch the Bullet collision information if you want the physics to continue
    dispatcher.defaultNearCallback(collisionPair, dispatcher, dispatchInfo);
}

mDispatcher->setNearCallback(MyNearCallback);
```

Deriving your own class from btCollisionDispatcher

For even more fine grain control over the collision dispatch, you can derive your own class from *btCollisionDispatcher* and override one or more of the following methods:

```
        virtual bool      needsCollision(btCollisionObject* body0,btCollisionObject* body1);

        virtual bool      needsResponse(btCollisionObject* body0,btCollisionObject* body1);

        virtual void      dispatchAllCollisionPairs(btOverlappingPairCache* pairCache,const
btDispatcherInfo& dispatchInfo,btDispatcher* dispatcher) ;
```

6 Rigid Body Dynamics

Introduction

The rigid body dynamics is implemented on top of the collision detection module. It adds forces, mass, inertia, velocity and constraints.

- `btRigidBody` is used to simulate single 6-degree of freedom moving objects. btRigidBody is derived from btCollisionObject, so it inherits its world transform, friction and restitution and adds linear and angular velocity.
- `btTypedConstraint` is the base class for rigid body constraints, including *`btHingeConstraint`*, *`btPoint2PointConstraint`*, *`btConeTwistConstraint`*, *`btSliderConstraint`* and *`btGeneric6DOFconstraint`*.
- *`btDiscreteDynamicsWorld`* is *`UserCollisionAlgorithm`* *`btCollisionWorld`*, and is a container for rigid bodies and constraints. It provides the *`stepSimulation`* to proceed.
- *`btMultiBody`* is an alternative representation of a rigid body hierarchy using generalized (or reduced) coordinates, using the articulated body algorithm, as discussed by Roy Featherstone. The tree hierarchy starts with a fixed or floating base and child bodies, also called links, are connected by joints: 1-DOF revolute joint (similar to the btHingeConstraint for btRigidBody), 1-DOF prismatic joint (similar to btSliderConstraint)

Note that btMultiBody is introduced in the Bullet Physics SDK fairly recently, and is still work-in-progress. In this document, only the maximal coordinates based btRigidBody and btTypedConstraints is discussed. In a future revision, a chapter about btMultiBody will be added. For now, if you are interested in btMultiBody, please see the example browser and its source code in examples/MultiBody and examples/ImportURDF.

Static, Dynamic and Kinematic Rigid Bodies

There are 3 different types of objects in Bullet:

- Dynamic (moving) rigidbodies
  - positive mass
  - every simulation frame the dynamics will update its world transform
- Static rigidbodies
  - zero mass

    - cannot move but just collide

- Kinematic rigidbodies
    - zero mass
    - can be animated by the user, but there will be only one-way interaction: dynamic objects will be pushed away but there is no influence from dynamics objects

All of them need to be added to the dynamics world. The rigid body can be assigned a collision shape. This shape can be used to calculate the distribution of mass, also called inertia tensor.

Center of mass World Transform

The world transform of a rigid body is in Bullet always equal to its center of mass, and its basis also defines its local frame for inertia. The local inertia tensor depends on the shape, and the `btCollisionShape` class provides a method to calculate the local inertia, given a mass.

This world transform has to be a rigid body transform, which means it should contain no scaling, shear etc. If you want an object to be scaled, you can scale the collision shape. Other transformation, such as shear, can be applied (baked) into the vertices of a triangle mesh if necessary.

In case the collision shape is not aligned with the center of mass transform, it can be shifted to match. For this, you can use a `btCompoundShape`, and use the child transform to shift the child collision shape.

What's a MotionState?

MotionStates are a way for Bullet to do all the hard work for you getting the world transform of objects being simulated into the rendering part of your program.

In most situations, your game loop would iterate through all the objects you're simulating before each frame rander. For each object, you would update the position of the render object from the physics body. Bullet uses something called MotionStates to save you this effort.

There are multiple other benefits of MotionStates:

- Computation involved in moving bodies around is only done for bodies that have moved; no point updating the position of a render object every frame if it isn't moving.
- You don't just have to do render stuff in them. They could be effective for notifying network code that a body has moved and needs to be updated across the network.
- Interpolation is usually only meaningful in the context of something visible on-screen. Bullet manages body interpolation through MotionStates.

- You can keep track of a shift between graphics object and center of mass transform.
- They're easy

Interpolation

Bullet knows how to interpolate body movement for you. As mentioned, implemention of interpolation is handled through MotionStates.

If you attempt to ask a body for its position through `btCollisionObject::getWorldTransform` or `btRigidBody::getCenterOfMassTransform`, it will return the position at the end of the last physics tick. That's useful for many things, but for rendering you will want some interpolation. Bullet interpolates the transform of the body before passing the value to setWorldTransform.

If you want the non-interpolated position of a body [which will be the position as it was calculated at the end of the last physics tick], use btRigidBody::getWorldTransform() and query the body directly.

So how do I use one?

MotionStates are used in two places in Bullet.

The first is when the body is first created. Bullet grabs the initial position of the body from the motionstate when the body enters the simulation

Bullet calls `getWorldTransform` with a reference to the variable it wants you to fill with transform information

Bullet also calls `getWorldTransform` on kinematic bodies. Please see the section below

After the first update, during simulation Bullet will call the motion state for a body to move that body around

Bullet calls setWorldTransform with the transform of the body, for you to update your object appropriately

To implement one, simply inherit btMotionState and override getWorldTransform and setWorldTransform.

DefaultMotionState

Although recommended, it is not necessary to derive your own motionstate from btMotionState interface. Bullet provides a default motionstate that you can use for this. Simply construct it with the default transform of your body:

```
btDefaultMotionState* ms =new btDefaultMotionState();
```

There is an example for an Ogre3D Motion State in an Appendix.

Kinematic Bodies

If you plan to animate or move static objects, you should flag them as kinematic. Also disable the sleeping/deactivation for them during the animation. This means Bullet dynamics world will get the new worldtransform from the btMotionState every simulation frame.

```
body->setCollisionFlags( body->getCollisionFlags() |
btCollisionObject::CF_KINEMATIC_OBJECT);
body->setActivationState(DISABLE_DEACTIVATION);
```

If you are using kinematic bodies, then getWorldTransform is called every simulation step. This means that your kinematic body's motionstate should have a mechanism to push the current position of the kinematic body into the motionstate.

Simulation frames and interpolation frames

By default, Bullet physics simulation runs at an internal fixed framerate of 60 Hertz (0.01666). The game or application might have a different or even variable framerate. To decouple the application framerate from the simulation framerate, an automatic interpolation method is built into stepSimulation: when the application delta time, is smaller then the internal fixed timestep, Bullet will interpolate the world transform, and send the interpolated worldtransform to the btMotionState, without performing physics simulation. If the application timestep is larger then 60 hertz, more then 1 simulation step can be performed during each 'stepSimulation' call. The user can limit the maximum number of simulation steps by passing a maximum value as second argument.

When rigidbodies are created, they will retrieve the initial worldtransform from the btMotionState, using `btMotionState::getWorldTransform`. When the simulation is running, using stepSimulation, the new worldtransform is updated for active rigidbodies using the `btMotionState::setWorldTransform`.

Dynamic rigidbodies have a positive mass, and their motion is determined by the simulation. Static and kinematic rigidbodies have zero mass. Static objects should never be moved by the user.

7 Constraints

There are several constraints implemented in Bullet. See examples/ConstraintDemo for an example of each of them. All constraints including the btRaycastVehicle are derived from `btTypedConstraint`. Constraint act between two rigidbodies, where at least one of them needs to be dynamic.

Point to Point Constraint

Point to point constraint limits the translation so that the local pivot points of 2 rigidbodies match in worldspace. A chain of rigidbodies can be connected using this constraint.

```
btPoint2PointConstraint(btRigidBody& rbA,const btVector3& pivotInA);
btPoint2PointConstraint(btRigidBody& rbA,btRigidBody& rbB, const btVector3& pivotInA,const btVector3& pivotInB);
```

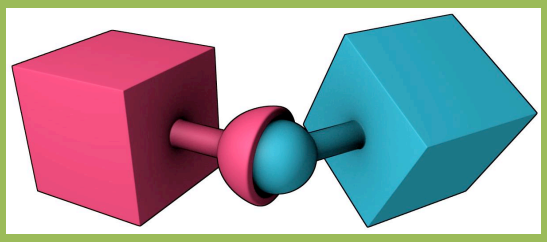


Figure 3 Point to point constraint

Hinge Constraint

Hinge constraint, or revolute joint restricts two additional angular degrees of freedom, so the body can only rotate around one axis, the hinge axis. This can be useful to represent doors or wheels rotating around one axis. The user can specify limits and motor for the hinge.

```
btHingeConstraint(btRigidBody& rbA,const btTransform& rbAFrame, bool useReferenceFrameA = false);
btHingeConstraint(btRigidBody& rbA,const btVector3& pivotInA,btVector3& axisInA, bool useReferenceFrameA = false);
btHingeConstraint(btRigidBody& rbA,btRigidBody& rbB, const btVector3& pivotInA,const btVector3&
pivotInB, btVector3& axisInA,btVector3& axisInB, bool useReferenceFrameA = false);
btHingeConstraint(btRigidBody& rbA,btRigidBody& rbB, const btTransform& rbAFrame, const btTransform& rbBFrame, bool
useReferenceFrameA = false);
```

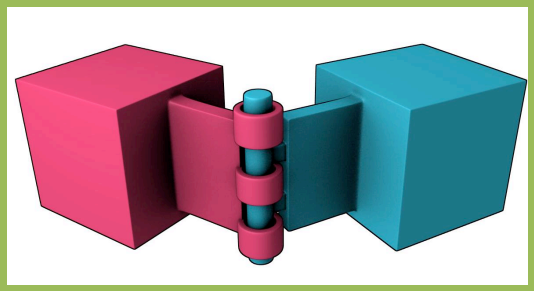


Figure 4 Hinge Constraint

Slider Constraint

The slider constraint allows the body to rotate around one axis and translate along this axis.

```
btSliderConstraint(btRigidBody& rbA, btRigidBody& rbB, const btTransform& frameInA, const btTransform& frameInB ,bool
useLinearReferenceFrameA);
```

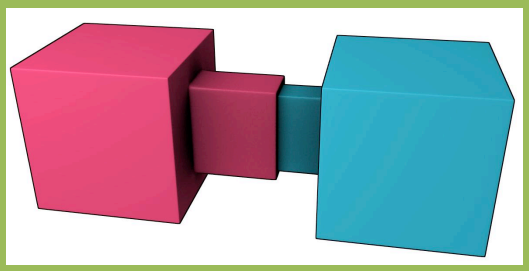


Figure 5 Slider Constraint

Cone Twist Constraint

To create ragdolls, the conve twist constraint is very useful for limbs like the upper arm. It is a special point to point constraint that adds cone and twist axis limits. The x-axis serves as twist axis.

```
btConeTwistConstraint(btRigidBody& rbA,const btTransform& rbAFrame);

btConeTwistConstraint(btRigidBody& rbA,btRigidBody& rbB,const btTransform& rbAFrame, const btTransform& rbBFrame);
```

Generic 6 Dof Constraint

This generic constraint can emulate a variety of standard constraints, by configuring each of the 6 degrees of freedom (dof). The first 3 dof axis are linear axis, which represent translation of rigidbodies, and the latter 3 dof axis represent the angular motion. Each axis can be either locked, free or limited. On construction of a new *btGeneric6DofSpring2Constraint*, all axis are locked. Afterwards the axis can be reconfigured. Note that several combinations that include free and/or limited angular degrees of freedom are undefined. See the Bullet/examples/Dof6SpringSetup.cpp.

```
btGeneric6DofConstraint(btRigidBody& rbA, btRigidBody& rbB, const btTransform& frameInA, const btTransform& frameInB
,bool useLinearReferenceFrameA);
```

Following is convention:

```
btVector3 lowerSliderLimit = btVector3(-10,0,0);
btVector3 hiSliderLimit = btVector3(10,0,0);

btGeneric6DofSpring2Constraint* slider = new btGeneric6DofSpring2Constraint(*d6body0,*fixedBody1,frameInA,frameInB);
slider->setLinearLowerLimit(lowerSliderLimit);
slider->setLinearUpperLimit(hiSliderLimit);
```

For each axis:

- Lowerlimit == Upperlimit -> axis is locked.
- Lowerlimit > Upperlimit -> axis is free
- Lowerlimit < Upperlimit -> axis it limited in that range

It is recommended to use the *btGeneric6DofSpring2Constraint,* it has some improvements over the original *btGeneric6Dof(Spring)Constraint.*

8 Actions: Vehicles & Character Controller

Action Interface

In certain cases it is useful to process some custom physics game code inside the physics pipeline. Although it is possible to use a tick callback, when there are several objects to be updated, it can be more convenient to derive your custom class from `btActionInterface`. And implement the `btActionInterface::updateAction(btCollisionWorld* world, btScalar deltaTime);` There are built-in examples, `btRaycastVehicle` and `btKinematicCharacterController`, that are using this `btActionInterace`.

Raycast Vehicle

For arcade style vehicle simulations, it is recommended to use the simplified Bullet vehicle model as provided in btRaycastVehicle. Instead of simulation each wheel and chassis as separate rigid bodies, connected by constraints, it uses a simplified model. This simplified model has many benefits, and is widely used in commercial driving games.

The entire vehicle is represented as a single rigidbody, the chassis. The collision detection of the wheels is approximated by ray casts, and the tire friction is a basic anisotropic friction model.

See `src/BulletDynamics/Vehicle` and `examples/ForkLiftDemo` for more details, or check the Bullet forums.

Kester Maddock shared an interesting document about Bullet vehicle simulation here:

http://tinyurl.com/ydfb7lm

Character Controller

A player or NPC character can be constructed using a capsule shape, sphere or other shape. To avoid rotation, you can set the 'angular factor' to zero, which disables the angular rotation effect during collisions and other constraints. See `btRigidBody::setAngularFactor`. Other options (that are less recommended) include setting the inverse inertia tensor to zero for the up axis, or using a angular-only hinge constraint.

There is also an experimental[3] `btKinematicCharacterController` as an example a non-physical character controller. It uses a `btGhostShape` to perform collision queries to create a character that can climb stairs, slide smoothly along walls etc. See `src/BulletDynamics/Character` and `Demos/CharacterDemo` for its usage.

[3] `btKinematicCharacterController` has several outstanding issues.

9 Soft Body Dynamics

Preliminary documentation

Introduction

The soft body dynamics provides rope, cloth simulation and volumetric soft bodies, on top of the existing rigid body dynamics. There is two-way interaction between soft bodies, rigid bodies and collision objects.

- `btSoftBody` is the main soft body object. It is derived from `btCollisionObject`. Unlike rigid bodies, soft bodies don't have a single world transform: each node/vertex is specified in world coordinate.
- `btSoftRigidDynamicsWorld` is the container for soft bodies, rigid bodies and collision objects.

It is best to learn from `examples/SoftBodyDemo` how to use soft body simulation.

Here are some basic guidelines in a nutshell:

Construction from a triangle mesh

The `btSoftBodyHelpers::CreateFromTriMesh` can automatically create a soft body from a triangle mesh.

Collision clusters

By default, soft bodies perform collision detection using between vertices (nodes) and triangles (faces). This requires a dense tessellation, otherwise collisions might be missed. An improved method uses automatic decomposition into convex deformable clusters. To enable collision clusters, use:

```
psb->generateClusters(numSubdivisions);

//enable cluster collision between soft body and rigid body

psb->m_cfg.collisions         +=        btSoftBody::fCollision::CL_RS;

//enable cluster collision between soft body and soft body

psb->m_cfg.collisions += btSoftBody::fCollision::CL_SS;
```

The Softbody of the ExampleBrowser has a debug option to visualize the convex collision clusters.

Applying forces to a Soft body

There are methods to apply a force to each vertex (node) or at an individual node:

```
softbody ->addForce(const btVector3& forceVector);

softbody ->addForce(const btVector3& forceVector,int node);
```

Soft body constraints

It is possible to fix one or more vertices (nodes), making it immovable:

```
softbody->setMass(node,0.f);
```

or to attach one or more vertices of a soft body to a rigid body:

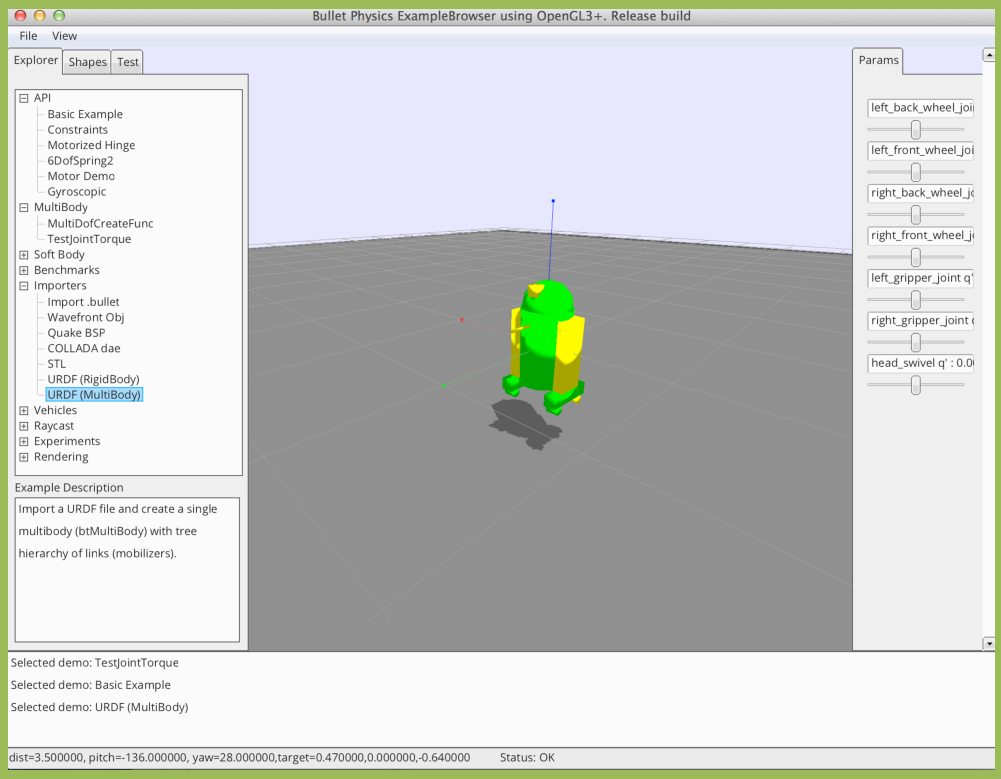
```
softbody->appendAnchor(int node,btRigidBody* rigidbody, bool
disableCollisionBetweenLinkedBodies=false);
```

It is also possible to attach two soft bodies using constraints, see *Bullet/Demos/SoftBody*.

10 Bullet Example Browser

Bullet 2.83 introduces a new example browser based on OpenGL 3+, replacing the previous Glut demos. There is a command-line option with limited support for OpenGL 2: `--opengl2`

The example browser is tested on Windows, Linux and Mac OSX. It has Retina support, so it looks best on Mac OSX. Here is a screenshot:



Each example can also be compiled stand-alone without graphics. See the `examples/BasicDemo/main.cpp` how to do this.

BSP Demo

Import a Quake .bsp files and convert the brushes into convex objects. This performs better then using triangles.

Vehicle Demo

This demo shows the use of the build-in vehicle. The wheels are approximated by ray casts. This approximation works very well for fast moving vehicles.

Fork Lift Demo

A demo that shows how to use constraints like hinge and slider constraint to build a fork lift vehicle.

11 Advanced Low Level Technical Demos

Collision Interfacing Demo

This demo shows how to use Bullet collision detection without the dynamics. It uses the `btCollisionWorld` class, and fills this will `btCollisionObjects`. The `performDiscreteCollisionDetection` method is called and the demo shows how to gather the contact points.

Collision Demo

This demo is more low level then previous Collision Interfacing Demo. It directly uses the `btGJKPairDetector` to query the closest points between two objects.

User Collision Algorithm

Shows how you can register your own collision detection algorithm that handles the collision detection for a certain pair of collision types. A simple sphere-sphere case overides the default GJK detection.

Gjk Convex Cast / Sweep Demo

This demo show how to performs a linear sweep between to collision objects and returns the time of impact. This can be useful to avoid penetrations in camera and character control.

Continuous Convex Collision

Shows time of impact query using continuous collision detection, between two rotating and translating objects. It uses Bullet's implementation of Conservative Advancement.

Raytracer Demo

This shows the use of CCD ray casting on collision shapes. It implements a ray tracer that can accurately visualize the implicit representation of collision shapes. This includes the collision margin, convex hulls of implicit objects, minkowski sums and other shapes that are hard to visualize otherwise.

Simplex Demo

This is a very low level demo testing the inner workings of the GJK sub distance algorithm. This calculates the distance between a simplex and the origin, which is drawn with a red line. A simplex contains 1 up to 4 points, the demo shows the 4 point case, a tetrahedron. The Voronoi simplex solver is used, as described by Christer Ericson in his collision detection book.

12 Authoring Tools and Serialization

Collision shapes, rigid body and constraints can be created in a 3D authoring tool and exported to a file format that Bullet can read.

Dynamica Maya Plugin

Walt Disney Animation Studios contributed their in-house Maya plugin to author Bullet collision shapes and rigid bodies as open source. Dynamica can simulation rigid body dynamica within Maya, and it can export to Bullet .bullet physics files and COLLADA Physics. The latest version has preliminary support for cloth/soft body.

There is more information in the Bullet wiki page. You can download a precompiled version of the Dynamica plugin for Windows or Mac OSX from http://bullet.googlecode.com.
The source code repository of Dynamica is under http://dynamica.googlecode.com

Blender

The open source 3D production suite Blender uses Bullet physics for animations and its internal game engine. See `http://blender.org`
Blender has an option to export COLLADA Physics files. There is also a project that can directly read any information from a Blender .blend file, including collision shape, rigid body and constraint information. See `http://gamekit.googlecode.com`
Blender 2.57 and later has an option to export to .bullet files directly from the game engine. This can be done using the *`exportBulletFile("name.bullet")`* Python command in the *`PhysicsConstraints`* module.

Cinema 4D, Lightwave CORE, Houdini

Cinema 4D 11.5 uses Bullet for the rigid body simulation, and there is a report that Lightwave CORE also plans to use Bullet.

For Houdini there is a DOP/plugin, see
`http://code.google.com/p/bullet-physics-solver/`

Serialization and the Bullet .bullet binary format

Bullet 2.76 onwards has the capability to save the dynamics world to a binary dump. Saving the objects and shapes into a buffer is built-in, so no additional libraries are necessary. Here is an example how to save the dynamics world to a binary .bullet file:

```
btDefaultSerializer* serializer = new btDefaultSerializer();

dynamicsWorld->serialize(serializer);

FILE* file = fopen("testFile.bullet","wb");

fwrite(serializer->getBufferPointer(),serializer->getCurrentBufferSize(),1, file);

fclose(file);
```

You can press the F3 key in most of the Bullet examples to save a 'testFile.bullet'. You can read .bullet files using the btBulletWorldImporter as implemented in the `Bullet/examples/Importers/ImportBullet`.

Futher information about .bullet serialization is at the Bullet wiki at

http://bulletphysics.org/mediawiki-1.5.8/index.php/Bullet_binary_serialization

13 General Tips

Avoid very small and very large collision shapes

The minimum object size for moving objects is about 0.2 units, 20 centimeters for Earth gravity. If smaller objects or bigger gravity are manipulated, reduce the internal simulation frequency accordingly, using the third argument of `btDiscreteDynamicsWorld::stepSimulation`. By default it is 60Hz. For instance, simulating a dice throw (1cm-wide box with a gravity of 9.8m/s2) requires a frequency of at least 300Hz (1./300.). It is recommended to keep the maximum size of moving objects smaller then about 5 units/meters.

Avoid large mass ratios (differences)

Simulation becomes unstable when a heavy object is resting on a very light object. It is best to keep the mass around 1. This means accurate interaction between a tank and a very light object is not realistic.

Combine multiple static triangle meshes into one

Many small `btBvhTriangleMeshShape` pollute the broadphase. Better combine them.

Use the default internal fixed timestep

Bullet works best with a fixed internal timestep of at least 60 hertz (1/60 second).

For safety and stability, Bullet will automatically subdivide the variable timestep into fixed internal simulation substeps, up to a maximum number of substeps specified as second argument to stepSimulation. When the timestep is smaller then the internal substep, Bullet will interpolate the motion.

This safety mechanism can be disabled by passing 0 as maximum number of substeps (second argument to `stepSimulation`): the internal timestep and substeps are disabled, and the actual timestep is simulated. It is not recommended to disable this safety mechanism.

For ragdolls use btConeTwistConstraint

It is better to build a ragdoll out of `btHingeConstraint` and/or `btConeTwistLimit` for knees, elbows and arms.

Don’t set the collision margin to zero

Collision detection system needs some margin for performance and stability. If the gap is noticeable, please compensate the graphics representation.

Use less then 100 vertices in a convex mesh

It is best to keep the number of vertices in a *btConvexHullShape* limited. It is better for performance, and too many vertices might cause instability. Use the *btShapeHull* utility to simplify convex hulls.

Avoid huge or degenerate triangles in a triangle mesh

Keep the size of triangles reasonable, say below 10 units/meters. Also degenerate triangles with large size ratios between each sides or close to zero area can better be avoided.

The profiling feature btQuickProf bypasses the memory allocator

If necessary, disable the profiler when checking for memory leaks, or when creating the final version of your software release. The profiling feature can be switched off by defining *#define BT_NO_PROFILE 1* in *Bullet/src/LinearMath/btQuickProf.h*

Advanced Topics

Per triangle friction and restitution value

By default, there is only one friction value for one rigidbody. You can achieve per shape or per triangle friction for more detail. See the *Demos/ConcaveDemo* how to set the friction per triangle. Basically, add CF_CUSTOM_MATERIAL_CALLBACK to the collision flags or the rigidbody, and register a global material callback function. To identify the triangle in the mesh, both triangleID and partId of the mesh is passed to the material callback. This matches the triangleId/partId of the striding mesh interface.

An easier way is to use the *btMultimaterialTriangleMeshShape*. See the *Demos/MultiMaterialDemo* for usage.

Other MLCP Constraint Solvers

Bullet uses its *btSequentialImpulseConstraintSolver* by default. You can use a different constraint solver, by passing it into the constructor of your btDynamicsWorld. Those alternative MLCP constraint solvers are in Bullet/src/BulletDynamics/MLCPSolvers. See the source code of examples/vehicles/VehicleDemo how to use a different constraint solver.

Custom Friction Model

If you want to have a different friction model for certain types of objects, you can register a friction function in the constraint solver for certain body types. This feature is not compatible with the cache friendly constraint solver setting.

See #define `USER_DEFINED_FRICTION_MODEL` in Demos/CcdPhysicsDemo.cpp.

14 Parallelism using OpenCL

OpenCL rigid body and collision detection.

We implement from scratch a rigid body and collision detection pipeline that runs 100% using OpenCL kernels. It works best on high-end discrete desktop GPUs such as AMD 7970 or newer or NVIDIA GTX 680 or newer.

A simple OpenCL example is disabled by default in the example browser. If you have the right GPU hardware and up-to-date OpenCL driver/compiler you can use the following command-line option:

--enable_experimental_opencl

Note that there are many reasons why the OpenCL kernels fail, and you will need to become familiar with OpenCL to deal with those issues.

Please see the separate pdf document with more background about the OpenCL collision detection and rigid body pipeline in the Bullet/docs folder. There is also a book chapter about the OpenCL rigid body pipeline as part of 'Multithreading in Visual Effects' by CRC Press. This book is also available from Amazon.

15 Further documentation and references

Online resources

Visit the Bullet Physics website at http://bulletphysics.org for a discussion forum, a wiki with frequently asked questions and tips and download of the most recent version. The Wikipedia page lists some games and films using Bullet at http://en.wikipedia.org/wiki/Bullet_(software)

Authoring Tools

- Dynamica Maya plugin and Bullet COLLADA Physics support at http://dynamica.googlecode.com
- Blender 3D modeler includes Bullet and COLLADA physics support: http://www.blender.org
- COLLADA physics standard: http://www.khronos.org/collada

Books

- Realtime Collision Detection, Christer Ericson
  http://www.realtimecollisiondetection.net/
  Bullet uses the discussed voronoi simplex solver for GJK
- Collision Detection in Interactive 3D Environments, Gino van den Bergen
  http://www.dtecta.com also website for Solid collision detection library
  Discusses GJK and other algorithms, very useful to understand Bullet
- Physics Based Animation, Kenny Erleben
  http://www.diku.dk/~kenny/
  Very useful to understand Bullet Dynamics and constraints
- Multithreading in Visual Effects
  Discussed the OpenCL rigid body work for Bullet Physics.

Contributions and people

The Bullet Physics library is under active development in collaboration with many professional game developers, movie studios, as well as academia, students and enthusiasts.

Main author and project lead is Erwin Coumans, who started the project at Sony Computer Entertainment America US R&D then at Advanced Micro Devices and now at Google.

Some people that contributed source code to Bullet:

Roman Ponomarev, SCEA, constraints, CUDA and OpenCL research
John McCutchan, SCEA, ray cast, character control, several improvements
Nathanael Presson, Havok: initial author of Bullet soft body dynamics and EPA
Gino van den Bergen, Dtecta: LinearMath classes, various collision detection ideas
Christer Ericson, SCEA: voronoi simplex solver
Phil Knight, Disney Avalanche Studios: multiplatform compatibility, BVH serialization
Ole Kniemeyer, Maxon: various general patches, btConvexHullComputer
Simon Hobbs, SCEE: 3d axis sweep and prune and parts of btPolyhedralContactClipping
Pierre Terdiman, NVIDIA: various work related to separating axis test, sweep and prune
Dirk Gregorius, Factor 5 : discussion and assistance with constraints
Erin Catto, Blizzard: accumulated impulse in sequential impulse
Francisco Leon : GIMPACT Concave Concave collision
Eric Sunshine: jam + msvcgen buildsystem (replaced by cmake since Bullet 2.76)
Steve Baker: GPU physics and general implementation improvements
Jay Lee, TrionWorld: double precision support
KleMiX, aka Vsevolod Klementjev, managed version, C# port to XNA
Marten Svanfeldt, Starbreeze: parallel constraint solver and other improvements and optimizations
Marcus Hennix, Starbreeze: btConeTwistConstaint etc.
Arthur Shek, Nicola Candussi, Lawrence Chai, Disney Animation: Dynamica Maya Plugin

Many more people have contributed to Bullet, thanks to everyone on the Bullet forums.