# Bullet 2.73 Physics SDK Manual

This draft documentation is work-in-progress

Check out the Wiki and forums at http://bulletphysics.com

# Table of Contents

**1 Introduction ..... 6**

Decription of the library ..... 6

History ..... 6

Mission Statement ..... 6

Main Features ..... 7

**2 Library Overview ..... 8**

Software Design ..... 8

Basic Data Types and Math Library ..... 8

Memory Management, Alignment, Containers ..... 9

Timing and Performance Profiling ..... 9

Debug Drawing ..... 10

Collision Detection ..... 10

Rigid Body Dynamics ..... 11

Soft Body Dynamics ..... 11

**3 Quickstart ..... 12**

Step 1: Download ..... 12

Step 2: Building ..... 12

Step 3: Testing demos ..... 12

Step 4: Integrating Bullet physics in your application ..... 12

Step 5 : Integrate only the Collision Detection Library ..... 13

Step 6 : Use snippets only, like the GJK Closest Point calculation. ..... 13

Integration overview ..... 14

**4 Bullet Collision Detection ..... 15**

Bullet Collision Shapes ..... 15

Convex Primitives ..... 15

Compound Shapes ..... 15

Convex Hull Shapes ..... 15

Concave Triangle Meshes ... 16
Convex Decomposition ... 16
Height field ... 16
btStaticPlaneShape ... 16
Scaling of Collision Shapes ... 16
Collision Margin ... 17
Collision Matrix ... 18
Registering custom collision shapes and algorithms ... 18
**5 Collision Filtering (selective collisions) ... 19**
Filtering collisions using masks ... 19
Filtering Collisions Using a Broadphase Filter Callback ... 20
Filtering Collisions Using a Custom NearCallback ... 20
Deriving your own class from btCollisionDispatcher ... 21
**6 Bullet Rigid Body Dynamics ... 22**
World Transforms and btMotionState ... 22
Static, Dynamic and Kinematic Rigid Bodies ... 22
Simulation frames and interpolation frames ... 23
**7 Bullet Constraints ... 24**
btPoint2PointConstraint ... 24
btHingeConstraint ... 24
btConeTwistConstraint ... 24
btGeneric6DofConstraint ... 24
btRaycastVehicle ... 25
Character Controller ... 25
**8 Bullet Demo Description ... 26**
AllBulletDemos ... 26
CCD Physics Demo ... 26
COLLADA Physics Viewer Demo ... 26
BSP Demo ... 26

Vehicle Demo ..... 27
Fork Lift Demo ..... 27
Collision Interfacing Demo ..... 27
Collision Demo ..... 27
User Collision Algorithm ..... 27
Gjk Convex Cast / Sweep Demo ..... 27
Continuous Convex Collision ..... 27
Raytracer Demo ..... 28
Simplex Demo ..... 28

**9 General Tips ..... 29**

Avoid very small and very large collision shapes ..... 29
Avoid large mass ratios (differences) ..... 29
Combine multiple static triangle meshes into one ..... 29
Use the default internal fixed timestep ..... 29
For ragdolls use btConeTwistConstraint ..... 29
Don’t set the collision margin to zero ..... 29
Use less then 100 vertices in a convex mesh ..... 30
Avoid huge or degenerate triangles in a triangle mesh ..... 30
Per triangle friction and restitution value ..... 31
Custom Constraint Solver ..... 31
Custom Friction Model ..... 31

**10 Parallelism: SPU, CUDA, OpenCL ..... 32**

Cell SPU / SPURS optimized version ..... 32
Unified multi threading ..... 32
Win32 Threads, pthreads, sequential thread support ..... 32
IBM Cell SDK 3.1, libspe2 SPU optimized version ..... 32
btCudaBroadphase ..... 32

**11 Further documentation and references ..... 33**

Online resources ..... 33

Authoring Tools ..... 33

Books ..... 33

Contributions / people ..... 34

# 1 Introduction

## Decription of the library

Bullet Physics is a professional open source collision detection, rigid body and soft body dynamics library. The library is free for commercial use under the ZLib license from http://bulletphysics.com

Authoring of physics content can be done using the Dynamica Maya plugin, the free Blender 3D modeler, or any other tool that support the COLLADA Physics specification. Both Dynamica and Blender allow to rapidly prototype the physics simulation and to bake the simulation into keyframes for movie rendering.

## History

Bullet started in 2003 as a research project by Erwin Coumans, former Havok employee and currently working for Sony Computer Entertainment America. Since 2005 Bullet has been open sourced, and many professional and enthusiast game developers are using, contributing and collaborating in the project. Target audience for this work are professional game developers, physics enthusiasts who are interested in collision detection and rigidbody dynamics.

Bullet is used in several games for Playstation 2 and 3, XBox 360, Nintendo Wii and PC, either fully or only selected parts. It is under active development and there are ports to C# and Java. Some of the recent new developments are CUDA GPGPU acceleration and an iPhone version.

## Mission Statement

- Supporting open standards in real-time physics simulation
- Making the cross-platform Bullet Physics SDK widely adapted by game and movie studios
- Sharing knowledge and experience amongst self-sufficient physics developers and enthusiasts

## Main Features

- Collision detection, Rigid body dynamics and Soft Body dynamics
- Open source C++ code under Zlib license and free for any commercial use
- Discrete and continuous collision detection including ray and convex sweep test
- Collision shapes include concave and convex meshes and all basic primitives
- Rigid body dynamics solver with auto deactivation, pyramid friction and restitution
- Several constraints including slider, hinge, generic 6DOF and cone twist constraint for ragdolls
- Vehicle simulation with tuning parameters
- Soft Body dynamics for cloth and deformable volumes with two-way interaction with rigid bodies
- Soft Body constraints and convex collision clusters
- COLLADA physics import/export with tool chain
- Compiles out-of-the-box for all platforms, including PC, Linux, Mac OSX and iPhone
- C# port available that runs on XNA for Windows and Xbox 360
- Initial CUDA support, to be ported to OpenCL
- Cell SPU optimized version available through Sony PS3 Devnet

# 2 Library Overview

## Software Design

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

- 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
- use the collision detection component without the rigid body dynamics component
- use the rigid body dynamics component without soft body dynamics component
- use only small parts of a the library
- extend the library in many ways

## Basic Data Types and Math Library

- *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.

- *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.

*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. Also it is important for a middleware library to use a memory allocator provided by the developer. For those reasons the Bullet library provides:

- *ATTRIBUTE_ALIGNED16(type) variablename* creates a 16-byte aligned variable
- *btAlignedAlloc*, all memory allocations use *btAlignedAlloc*, which allows to specify size and alignment
- *btAlignedFree*, free the memory allocated by *btAlignedAlloc*.

All internal memory allocations are using *btAlignedAlloc/Free*. This makes it easy for developers to slot in their own 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.

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.

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

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 axis and pivots show up.

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

## 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 providing 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
- *btCudaBroadphase* implements a fast uniform grid using GPU graphics hardware

The broadphase adds and removes overlapping pairs from a pair cache. The developer can choose the type of pair cache.

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.

## Rigid Body Dynamics

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

- `btRigidBody` is the main rigid body object, moving objects have non-zero mass and inertia. 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 derived from *`btCollisionWorld`*, and is a container for rigid bodies and constraints. It provides the stepSimulation.

## Soft Body Dynamics

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 *`Demos/SoftBodyDemo`* how to use soft body simulation.

# 3 Quickstart

## Step 1: Download

Windows developers should download the zipped sources from of Bullet from http://bulletphysics.com.Mac OS X, Linux and other developers should download the gzipped tar archive.

## Step 2: Building

Bullet should compile out-of-the-box for all platforms, and includes all dependencies.

- Windows Visual Studio projectfiles for all versions are available in `Bullet/msvc`. The main Workspace/Solution is located in `Bullet/msvc/8/wksbullet.sln`
- CMake adds support for many other build environments and platforms, including XCode for Mac OSX, KDevelop for Linux and Unix Makefiles. Download and install Cmake from http://www.cmake.org. Run cmake without arguments to see the list of build system generators for your platform. For example, run `cmake . -G Xcode` to auto-generate projectfiles for Mac OSX Xcode.
- Autoconf/automake generates both Makefile and Jamfile. Run `./autogen.sh`, `./configure` and then make or jam.

## Step 3: Testing demos

Try to run and experiment with *`Demos/AllBulletDemos`* executable as a starting point. Bullet can be used in several ways, as full rigid body simulation, as collision detector library or low level / snippets such as GJK closest point calculation. The dependencies can be seen in the doxygen documentation under 'Directories'.

## Step 4: Integrating Bullet physics in your application

Check out CcdPhysicsDemo how to create a *`btDynamicsWorld`*, *`btCollisionShape`*, *`btMotionState`* and *`btRigidBody`*, Stepping the simulation and synchronizing the transform for your graphics object. Requirements:

- *`#include "btBulletDynamicsCommon.h"`* in your source file
- Required include path: *`Bullet /src`* folder
- Required libraries: *`libbulletdynamics`*, *`libbulletcollision`*, *`libbulletmath`*

### Step 5 : Integrate only the Collision Detection Library

Bullet Collision Detection can also be used without the Dynamics/Extras. Check out the low level demo Collision Interface Demo, in particular the class CollisionWorld. Requirements:

- `#include "btBulletCollisionCommon.h"` at the top of your file
- Add include path: *`Bullet /src`* folder
- Add libraries: *`libbulletcollision`*, *`libbulletmath`*

### Step 6 : Use snippets only, like the GJK Closest Point calculation.

Bullet has been designed in a modular way keeping dependencies to a minimum. The *`Demos/ConvexHullDistance`* demo demonstrates direct use of *`btGjkPairDetector`*.

## 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/Demos/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`.

A lot of the details are demonstrated in the Demos. If you can't find certain functionality, please use the FAQ or the physics Forum on the Bullet website.

# 4 Bullet Collision Detection

## Bullet Collision Shapes

Bullet supports a large variety of different collision shapes, and it is possible to add your own.

### 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`.

### Convex Hull Shapes

Bullet supports several ways to collide against convex and concave triangle meshes. One easy way is to create a `btConvexHullShape` and pass in an array of vertices.

Bullet can also represent triangle meshes. The easiest way, but not so efficient, is to use the `btTriangleMesh`. The btTriangleMesh has a method to add triangles, one at the time, and it stores them in an array. A better way to represent triangle meshes is using a `btStridingMeshInterface`. This interface allows to pass pointers to existing index and vertex arrays, similar to OpenGL `glVertexPointer`/`glDrawElements`.

Once we have a triangle mesh, we can choose to collide with its convex hull, using the `btConvexTriangleMeshShape`.

### 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 *Demos/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.*

### 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 *Demos/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.

### 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 `Demos/BspDemo` 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<br>cone,capsule | compound | triangle mesh |
|---|---|---|---|---|---|
| box | boxbox | spherebox | gjk | compound | concaveconvex |
| sphere | spherebox | spheresphere | gjk | compound | concaveconvex |
| convex, cylinder,<br>cone,capsule | gjk | gjk | gjk | 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 `Demos/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.

For example, in a spaceship game, you could have your spaceships ignore collisions with other spaceships [the spaceships would just fly through each other], but always collide with walls [the spaceships always bounce off walls].

Your spaceship needs a callback when it collides with a wall [for example, to produce a “plink” sound], but the walls do nothing when you collide with them so they do not need to receive callbacks.

A third type of object, “powerup”, collides with walls and spaceships. Spaceships do not receive collisions from them, since we don't want the trajectory of the spaceship changed by collecting a powerup. The powerup object modifies the spaceship from its own collision callback.

In order to do this, you need a bit mask for the walls, spaceships, and powerups:

```
#define BIT(x) (1<<(x))
enum collisiontypes {
    COL_NOTHING = 0, //<Collide with nothing
    COL_SHIP = BIT(1), //<Collide with ships
    COL_WALL = BIT(2), //<Collide with walls
    COL_POWERUP = BIT(3) //<Collide with powerups
}

int shipCollidesWith = COL_WALL;
int wallCollidesWith = COL_NOTHING;
int powerupCollidesWith = COL_SHIP | COL_WALL;
```

After setting these up, simply add your body objects to the world as normal, except as the second and third parameters pass your collision type for that body, and the collision mask.

```
btRigidBody ship; // Set up the other ship stuff
btRigidBody wall; // Set up the other wall stuff
btRigidBody powerup; // Set up the other powerup stuff

mWorld->addRigidBody(ship, COL_SHIP, shipCollidesWith);
```

```
mWorld->addRigidBody(wall, COL_WALL, wallCollidesWith);
mWorld->addRigidBody(powerup, COL_POWERUP, powerupCollidesWith);
```

It's worth noting that if you are using masks, and they're sufficient for your needs, then you do not need a custom collision filtering.

If you have more types of objects than bits available to you in the masks above, 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 Bullet Rigid Body Dynamics

## CHAPTER IS WORK-IN-PROGRESS

### World Transforms and btMotionState

The main purpose of rigid body simulation is calculating the new world transform, position and orientation, of dynamic bodies. Usually each rigidbody is connected to a user object, like graphics object. It is a good idea to derive your own version of btMotionState class.

Each frame, Bullet dynamics will update the world transform for active bodies, by calling the `btMotionState::setWorldTransform`. Also, the initial center of mass worldtransform is retrieved, using btMotionState::getWorldTransform, to initialize the btRigidBody. If you want to offset the rigidbody center of mass world transform, relative to the graphics world transform, it is best to do this only in one place. You can use `btDefaultMotionState` as start implementation.

### Static, Dynamic and Kinematic Rigid Bodies

There are 3 different types of objects in Bullet:

- Dynamic rigidbodies
  - positive mass
  - User should only use apply impulse, constraints or setLinearVelocity/setAngularVelocity and let the dynamics calculate the new world transform
  - every simulation frame and interpolation frame, the dynamics world will write the new world transform using btMotionState::setWorldTransform
- Static rigidbodies
  - cannot move but just collide
  - zero mass
- Kinematic rigidbodies
  - animated by the user
  - only one-way interaction: dynamic objects will be pushed away but there is no influence from dynamics objects
  - every simulation frame, dynamics world will get new world transform using btMotionState::getWorldTransform

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.

## 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.

If you plan to animate or move static objects, you should flag them as kinematic. Also disable the sleeping/deactivation for them. 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);
```

# 7 Bullet Constraints

There are several constraints implemented in Bullet. See Demos/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.

### btPoint2PointConstraint

Point to point constraint, also known as ball socket joint 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.

### btHingeConstraint

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.

### btConeTwistConstraint

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.

### btGeneric6DofConstraint

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 *btGeneric6DofConstraint*, 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.

Following is convention:

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

btGeneric6DofConstraint* slider = new btGeneric6DofConstraint(*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

### btRaycastVehicle

For most 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 `Demos/VehicleDemo` for more details, or check the Bullet forums.

Changing the up axis of a vehicle., see `#define FORCE_ZAXIS_UP` in VehiceDemo.

### Character Controller

A basic 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.

`btKinematicCharacterController` is a class dedicated to character control. It uses a `btGhostShape` to perform collision queries to create a character that can climb stairs, slide smoothly along walls etc. See `Demos/CharacterDemo` for its usage.

# 8 Bullet Demo Description

Bullet includes several demos. They are tested on several platforms and use OpenGL graphics and Glut. Some shared functionality like mouse picking and text rendering is provided in the `Demos/OpenGL` support folder. This is implemented in the `DemoApplication` class. Each demo derives a class from `DemoApplication` and implements its own initialization of the physics in the '`initPhysics`' method.

### AllBulletDemos

This is a combination of several demos. It includes demonstrations of a fork lift, ragdolls, cloth and soft bodies and several performance benchmarks.

### CCD Physics Demo

This is a that shows how to setup a physics simulation, add some objects, and step the simulation. It shows stable stacking, and allows mouse picking and shooting boxes to collapse the wall. The shooting speed of the box can be changed, and for high velocities, the CCD feature can be enabled to avoid missing collisions. Try out advanced features using the #defines at the top of `Demos/CcdPhysicsDemo/CcdPhysicsDemo.cpp`

### COLLADA Physics Viewer Demo

Imports and exports COLLADA Physics files. It uses the included libxml and COLLADA-DOM library.

The COLLADA-DOM imports a .dae xml file that is generated by tools and plugins for popular 3D modelers. Dynamica Maya plugin, Blender and other software can export/import this standard physics file format. The `Extras/BulletColladaConverter` class can be used as example for other COLLADA physics integrations.

### 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.

## 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.

# 9 General Tips

### Avoid very small and very large collision shapes

The minimum object size for moving objects is about 0.1 units. When using default gravity of 9.8, those units are in meters so don't create objects smaller then 10 centimeter. 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.

# 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.

## Custom Constraint Solver

Bullet uses its *btSequentialImpulseConstraintSolver* by default. You can use a different constraint solver, by passing it into the constructor of your btDynamicsWorld. For comparison you can use the *Extras/quickstep* solver from ODE.

## 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.

# 10 Parallelism: SPU, CUDA, OpenCL

### Cell SPU / SPURS optimized version

Bullet collision detection and other parts have been optimized for Cell SPU. This means collision code has been refactored to run on multiple parallel SPU processors. The collision detection code and data have been refactored to make it suitable for 256kb local store SPU memory. The user can activate the parallel optimizations by using a special collision dispatcher (*SpuGatheringCollisionDispatcher*) that dispatches the work to SPU. The shared public implementation is located in *Bullet/src/BulletMultiThreaded*.

Please contact Sony developer support on PS3 Devnet for a Playstation 3 optimized version of Bullet.

### Unified multi threading

Efforts have been made to make it possible to re-use the SPU parallel version in other multi threading environments, including multi core processors. This allows more effective debugging of SPU code under Windows, as well as utilizing multi core processors. For non-SPU multi threading, the implementation performs fake DMA transfers using a memcpy, and each thread gets its own 256kb 'local store' working memory allocated.

### Win32 Threads, pthreads, sequential thread support

Basic Win32 Threads, pthreads and sequential thread support is available to execute the SPU tasks. Some demos show this preliminary work in action. See *#define USE_PARALLEL_DISPATCHER* in *Demos/ConcaveDemo* and *ConvexDecompositionDemo*.

### IBM Cell SDK 3.1, libspe2 SPU optimized version

IBM provides a Cell SDK with access to SPU through libspe2 for Cell Blade and PS3 Linux platforms. Libspe2 thread support is available through *SpuLibspe2Support*.
To compile the libspe2 version, first run *make* in *the Bullet/src/ibmsdk* directory. Then run *make –f Makefile.original spu ppu* in the *Bullet/src/BulletMultiThreaded* directory, and then run *make* in *the Bullet/Demos/CellSpuDemo/ibmsdk* directory.

### btCudaBroadphase

We are doing some research and development in using CUDA and OpenCL to accelerate parts of the physics pipeline. You can check *Extras/CUDA* and *Extras/CDTestFramework* for early results.

# 11 Further documentation and references

## Online resources

Visit the Bullet Physics website at http://bulletphysics.com for a discussion forum, a wiki with frequently asked questions and tips and download of the most recent version

## Authoring Tools

Dynamica Maya plugin, COLLADA-DOM, libxml are included in Bullet/Extras folder.

Nima Maya plugin, COLLADA export: http://sourceforge.net/projects/nimaplugin

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

## Contributions / people

Thanks everyone on the Bullet forum for feedback.

Some people that contributed source code to Bullet in random order:

Erwin Coumans, SCEA: main author, project lead
Roman Ponomarev, SCEA, constraints and CUDA implementation
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
Simon Hobbs, SCEE: 3d axis sweep and prune: and Extras/SATCollision
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
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.

Many more people have contributed to Bullet, thanks for that.
(please get in touch if your name should be in this list)