

Bullet 2.55 Physics User Manual

Last updated by Erwin Coumans on Wednesday, 11 July 2007

Index

Introduction.....	4
Main Features.....	4
Download and supporting physics Forum.....	4
Quickstart.....	5
Integration overview.....	6
Debugging.....	6
Bullet Rigid Body Dynamics.....	7
World Transforms and btMotionState.....	7
Static, Dynamic and Kinematic Objects using btRigidBody.....	7
Simulation frames and interpolation frames.....	8
Bullet Collision Shapes.....	9
Convex Primitives.....	9
Compound Shapes.....	9
Convex and Concave Meshes.....	10
Convex Decomposition.....	10
Height field.....	10
Scaling of Collision Shapes.....	10
Collision Margin.....	11
Bullet Constraints.....	12
btPoint2PointConstraint.....	12
btHingeConstraint.....	12
btConeTwistConstraint.....	12
btGeneric6DofConstraint.....	12
Bullet Vehicle.....	13
btRaycastVehicle.....	13
Bullet Character Controller.....	13
btCharacterController.....	13
Basic Demos.....	14
CCD Physics Demo.....	14
COLLADA Physics Viewer Demo.....	14
BSP Demo.....	15
Vehicle Demo.....	15
Fork Lift Demo.....	15
General Tips for Bullet users.....	16
Avoid very small and very large collision shapes.....	16
Avoid large mass ratios (differences).....	16
Combine multiple static triangle meshes into one.....	16
Use the default internal fixed timestep.....	16
For ragdolls use btConeTwistConstraint.....	16
Don't set the collision margin to zero.....	16
Use less then 100 vertices in a convex mesh.....	16
Avoid huge or degenerate triangles in a triangle mesh.....	16
Advanced Topics.....	17
Per triangle friction and restitution value.....	17
Custom Constraint Solver.....	17
Custom Friction Model.....	17

Collision Filtering (disabling collisions)	18
Collision groups and masks	18
Disable collisions between a pair of instances of objects	18
Collision Matrix	19
Registering custom collision shapes and algorithms	19
Advanced Low Level Technical Demos	20
Collision Interfacing Demo	20
Collision Demo	20
User Collision Algorithm	20
Gjk Convex Cast / Sweep Demo	20
Continuous Convex Collision	21
Raytracer Demo	21
Concave Demo	21
Simplex Demo	21
Bullet Collision Detection and Physics Architecture	22
Bullet Library Module Overview	23
Bullet Collision Detection Library Internals	24
Multi threaded version	25
Cell SPU / SPURS optimized version	25
Unified multi threading	25
Win32 Threads	25
IBM Cell SDK 2.1, libspe2 SPU optimized version	25
Future support for pthreads	25
Contributions / people	26
Contact	26
Further documentation and references	27
Links	27
Books	27

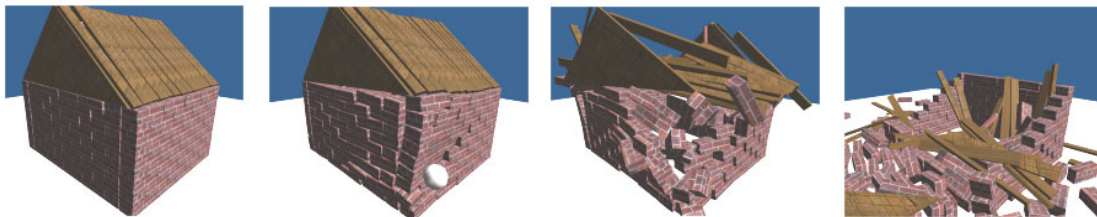
Introduction

Bullet Physics is a professional open source collision detection and physics library, related tools, demos, applications and a community forum at <http://bulletphysics.com> It is free for commercial use under the ZLib license.

Originally started as toy project by Erwin Coumans, ex-Havok employee, and since then many developers have contributed. Target audience for this work are game developers, academics as well as physics enthusiasts who want to play with 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 just the multi threaded collision detection parts. It is under active development and some of the recent new developments are the addition of a universal multi-threaded C++ version and a C# port that supports Windows and Xbox 360 XNA.

Authoring of physics content can be done using the COLLADA Physics specification. 3D modelers like Maya, Max, XSI, Blender and Ageia's CreateDynamics tools support COLLADA physics xml .dae files. Bullet is also integrated in the free Blender 3D modeler, <http://www.blender.org>. The integration allows real-time simulation and also baking the simulation into keyframes for rendering. See the References for other integrations and links.



Main Features

- ✓ Discrete and Continuous collision detection including ray casting
- ✓ Collision shapes include concave and convex meshes and all basic primitives
- ✓ Rigid body dynamics solver with auto deactivation
- ✓ Generic 6 degree of freedom constraint, hinge etc, for Ragdolls
- ✓ Vehicle simulation with tuning parameters
- ✓ COLLADA physics import/export with tool chain
- ✓ Compiles out-of-the-box for all platforms, including COLLADA support
- ✓ Open source C++ code under Zlib license and free for any commercial use
- ✓ C# port available that runs on XNA for Windows and Xbox 360
- ✓ Cell SPU optimized version available through Sony PS3 Devnet
- ✓ Multi-threaded version for multi core public available

Download and supporting physics Forum

Please visit <http://bulletphysics.com> for download, support and feedback.

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 zipped tar archive.

Step 2: Building

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

Jam: Bullet includes jam-2.5 sources from <http://www.perforce.com/jam/jam.html>. Install jam and run ./configure and then run jam, in the Bullet root directory.

Step 3: Testing demos

Try to run and experiment with CcdPhysicsDemo 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 like the 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: libbulletphysics, 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 ConvexHullDistance demo demonstrates direct use of **GjkPairDetector**.

Integration overview

If you want to use Bullet in your own 3D application, it is best to follow the steps in the CcdPhysicsDemo. In a nutshell:

- ✓ Create a `btDynamicsWorld` implementation like `btDiscreteDynamicsWorld`

This `btDynamicsWorld` is a high level interface that manages your physics objects and constraints. It also implements the update of all objects each frame. A `btContinuousDynamicsWorld` is under development to make use of Bullet's Continuous Collision Detection. This will prevent missing collisions of small and fast moving objects, also known as tunneling. Another solution based on internal variable timesteps called `btFlexibleStepDynamicsWorld` will be added too.

- ✓ 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
- `btMotionState` use to synchronize the World transform to controls the graphics
- Material properties like friction and restitution

- ✓ Update the simulation each frame: `stepSimulation`

Call the `stepSimulation` on the `btDynamicsWorld`. 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 objects by calling the `btMotionState`'s `setWorldTransform`.

There is performance functionality like auto deactivation for objects which motion is below a certain threshold.

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.

Debugging

You can get additional debugging feedback by registering a derived class from `IDebugDrawer`. You just need to hook up 3d line drawing with your graphics renderer. See the CcdPhysicsDemo `OpenGLDebugDrawer` for an example implementation. It can visualize collision shapes, contact points and more. This can help to find problems in the setup. Also the Raytracer demo shows how to visualize a complex collision shape.

Bullet Rigid Body Dynamics

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 Objects using btRigidBody

There are 3 different types of objects in Bullet:

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

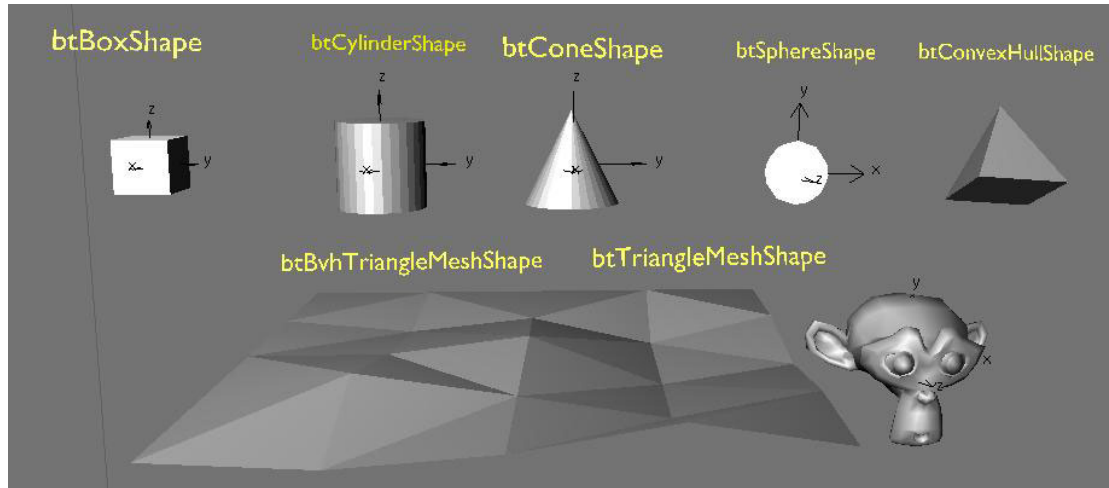
```
body->setCollisionFlags( body->getCollisionFlags() |  
btCollisionObject::CF_KINEMATIC_OBJECT);  
body->setActivationState(DISABLE_DEACTIVATION);
```


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

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

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

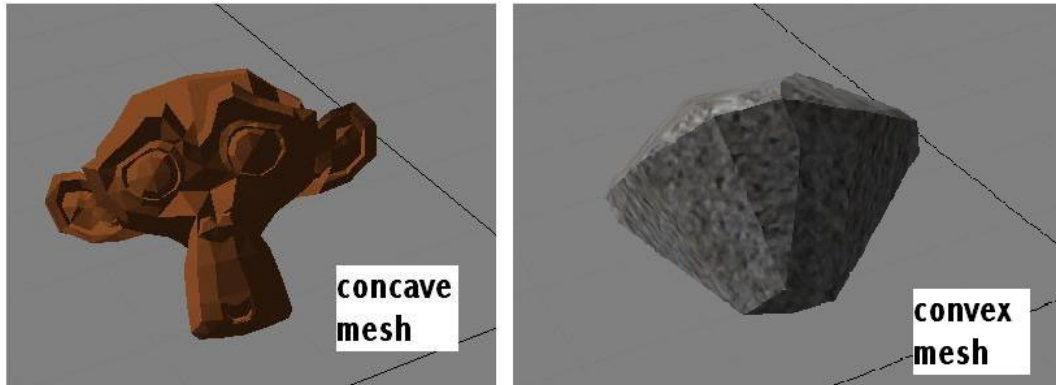
`btMultiSphereShape` : Convex hull of multiple spheres, that can be used to create a Capsule (by passing 2 spheres) or other convex shapes.

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 and Concave Meshes

For moving objects, concave meshes can be passed into `btConvexHullShape`. This automatically collides with the convex hull of the mesh:



General triangle meshes that represent static environment can best be represented in Bullet by using the `btBvhTriangleMeshShape`.

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 `ConvexDecompositionDemo` for an automatic way of doing convex decomposition. The implementation is taken from Ageia CreateDynamics tool, which can do the same with some fancy user interface. CreateDynamics can export to COLLADA Physics, so Bullet can import that data.

A recent contribution called GIMPACT can handle moving concave meshes. See `Demos/MovingConcaveDemo` for its usage.

Height field

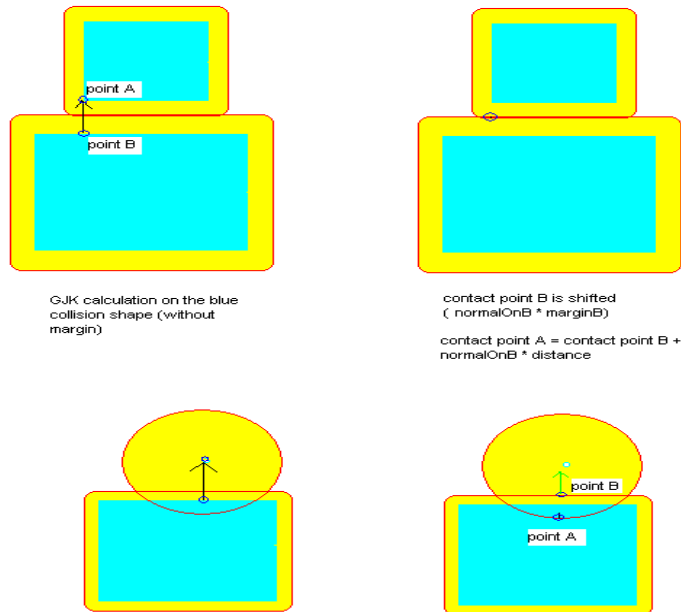
Bullet provides support for the special case of a flat 2D concave terrain through the `btHeightfieldTerrainShape`. See `VehicleDemo` for its usage.

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.

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 `BspDemo` for this advanced use. The yellow in the following picture described the working of collision margin for internal contact generation.

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 cone 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

The generic D6 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 `btGenericD6Constraint`, 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

Bullet Vehicle

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 VehicleDemo.

Bullet 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.

btCharacterController

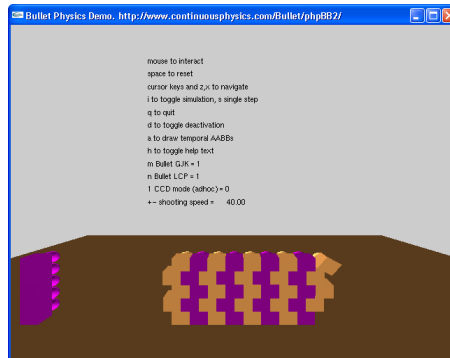
TODO: To get maximum control when moving a player character or NPC through the world `btCharacterController` will be provided. `btCharacterController` is a class derived from `btRigidBody` with special properties that make it easier to create a character that can climb stairs, slide smoothly along walls etc.

Basic Demos

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.

CCD Physics Demo

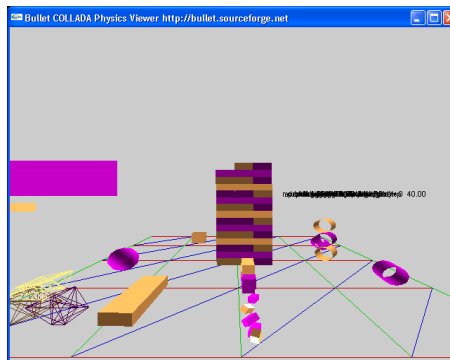
This is the main demo that shows how to setup a physics simulation, add some objects, and step the simulation. It shows stable stacking, and allows mouse picking and shooting boxes to collapse the wall. The shooting speed of the box can be changed, and for high velocities, the CCD feature can be enabled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



COLLADA Physics Viewer Demo

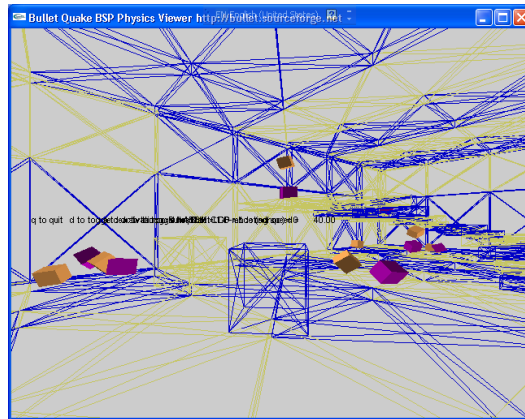
Imports and exports COLLADA Physics files. It uses the included libxml and COLLADA-DOM library.

The COLLADA-DOM imports a .dae xml file that is generated by tools and plugins for popular 3D modelers. ColladaMaya with Nima from FeelingSoftware, Blender, Ageia's free CreateDynamics tool and other software can export/import this standard physics file format. The ColladaConverter class can be used as example for other COLLADA physics integrations.



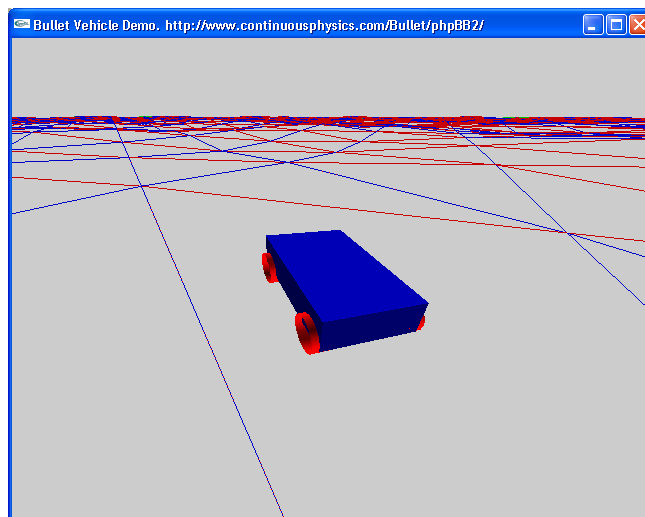
BSP Demo

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. For slow vehicles where the interaction between wheels and environment needs to be more precise the Forklift Demo is more recommended. The landscape is either triangle mesh or a heightfield.



Fork Lift Demo

TODO: A demo that shows how to use constraints like hinge and generic D6 constraint to build a fork lift vehicle. Wheels are approximated by cylinders.

General Tips for Bullet users

Avoid very small and very large collision shapes

The minimum object size for moving objects is about 0.2 units. When using default gravity of 9.8, those units are in meters so don't create objects smaller than 20 centimeter. It is recommended to keep the maximum size of moving objects smaller than 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 than 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. Avoid the `btGenericD6Constraint`, it won't work well.

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 than 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.

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.

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.

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

Collision Filtering (disabling collisions)

Collision groups and masks

To disable collision detection between certain shapes, collision groups and collision filter masks are used. By default, when a rigidbody is added to the `btDynamicsWorld`, the collision group and mask is chosen to prevent collisions between static objects at a very early stage. You can specify the group and mask in

`'btDynamicsWorld::addRigidBody'` and `'btCollisionWorld::addCollisionObject'`.

```
if (body->getCollisionShape())
{
    bool isDynamic = !(body->isStaticObject() || body->isKinematicObject());
    short collisionFilterGroup = isDynamic?
btBroadphaseProxy::DefaultFilter : btBroadphaseProxy::StaticFilter;
    short collisionFilterMask = isDynamic?
btBroadphaseProxy::AllFilter :
btBroadphaseProxy::AllFilter ^ btBroadphaseProxy::StaticFilter;

    addCollisionObject(body, collisionFilterGroup, collisionFilterMask);
}
```

The broadphase checks those filter flags to determine whether collision detection needs to be performed using the following code:

```
inline bool needsBroadphaseCollision(btBroadphaseProxy*
proxy0, btBroadphaseProxy* proxy1) const
{
    bool collides = (proxy0->m_collisionFilterGroup & proxy1->m_collisionFilterMask) != 0;
    collides = collides && (proxy1->m_collisionFilterGroup & proxy0->m_collisionFilterMask);

    return collides;
}
```

You can override this default filtering behaviour after the rigidbody has been added to the dynamics world by assigning new values to `collisionFilterGroup` and `collisionFilterMask`.

Disable collisions between a pair of instances of objects

When two bodies share a constraint, you can optionally disable the collision detection between those instances. Pass true as optional second argument to `btDiscreteDynamicsWorld::addConstraint` (this bool `disableCollisionsBetweenLinkedBodies` defaults to false).

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 Keethi, the people behind this convex distance calculation algorithm. EPA stands for Expanding Polythope Algorithm by Gino van den Bergen. Bullet has its own free implementation of GJK and EPA.

Bullet Collision Matrix (*= optional)

Collision Shape:	Sphere	Box	Convex	Compound	Trianglemesh
Sphere	GjkEpa /*SphereSphere	GjkEpa / *SphereBox	GjkEpa	Compound	ConcaveConvex
Box	GjkEpa /*SphereBox	GjkEpa / *BoxBox	GjkEpa /*SAT	Compound	ConcaveConvex
Convex	GjkEpa	GjkEpa / *SAT	GjkEpa /*SAT	Compound	ConcaveConvex
Compound	Compound	Compound	Compound	Compound	Compound
Trianglemesh	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 `UserCollisionAlgorithm` for an example, that registers SphereSphere collision algorithm.

Advanced Low Level Technical Demos

Collision Interfacing Demo

This demo shows how to use Bullet collision detection without the dynamics. It uses the CollisionWorld class, and fills this with CollisionObjects. performDiscreteCollisionDetection method is called and the demo shows how to gather the contact points.

Collision Demo

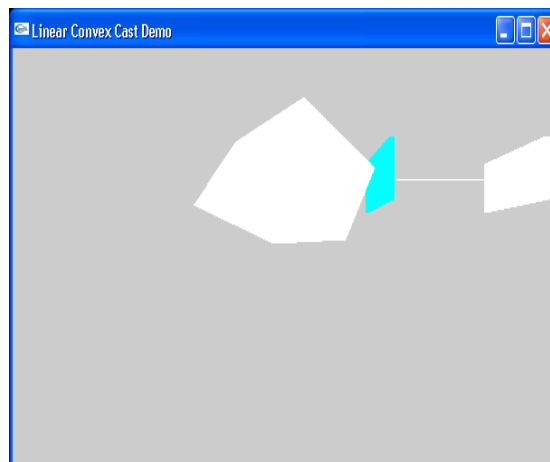
This demo is more low level than previous Collision Interfacing Demo. It directly uses the GJKPairDetector to query the closest points between two objects.

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 overrides the default GJK detection.

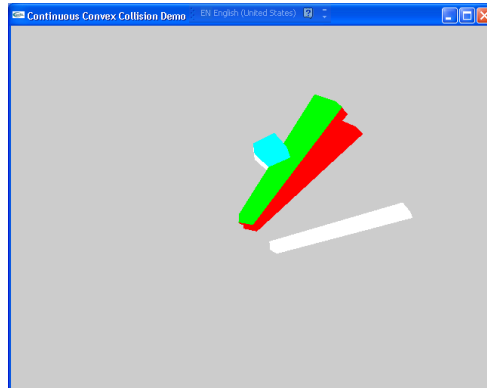
Gjk Convex Cast / Sweep Demo

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



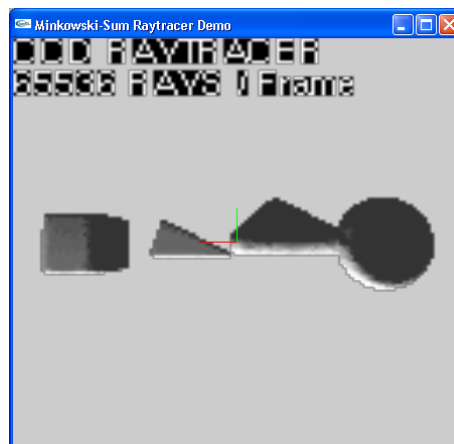
Continuous Convex Collision

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.



Concave Demo

This advanced demo shows how to implement user defined per-triangle restitution and friction in a static triangle mesh. A callback can be registered and triangle identifiers can be used to modify the friction in each reported contact point.

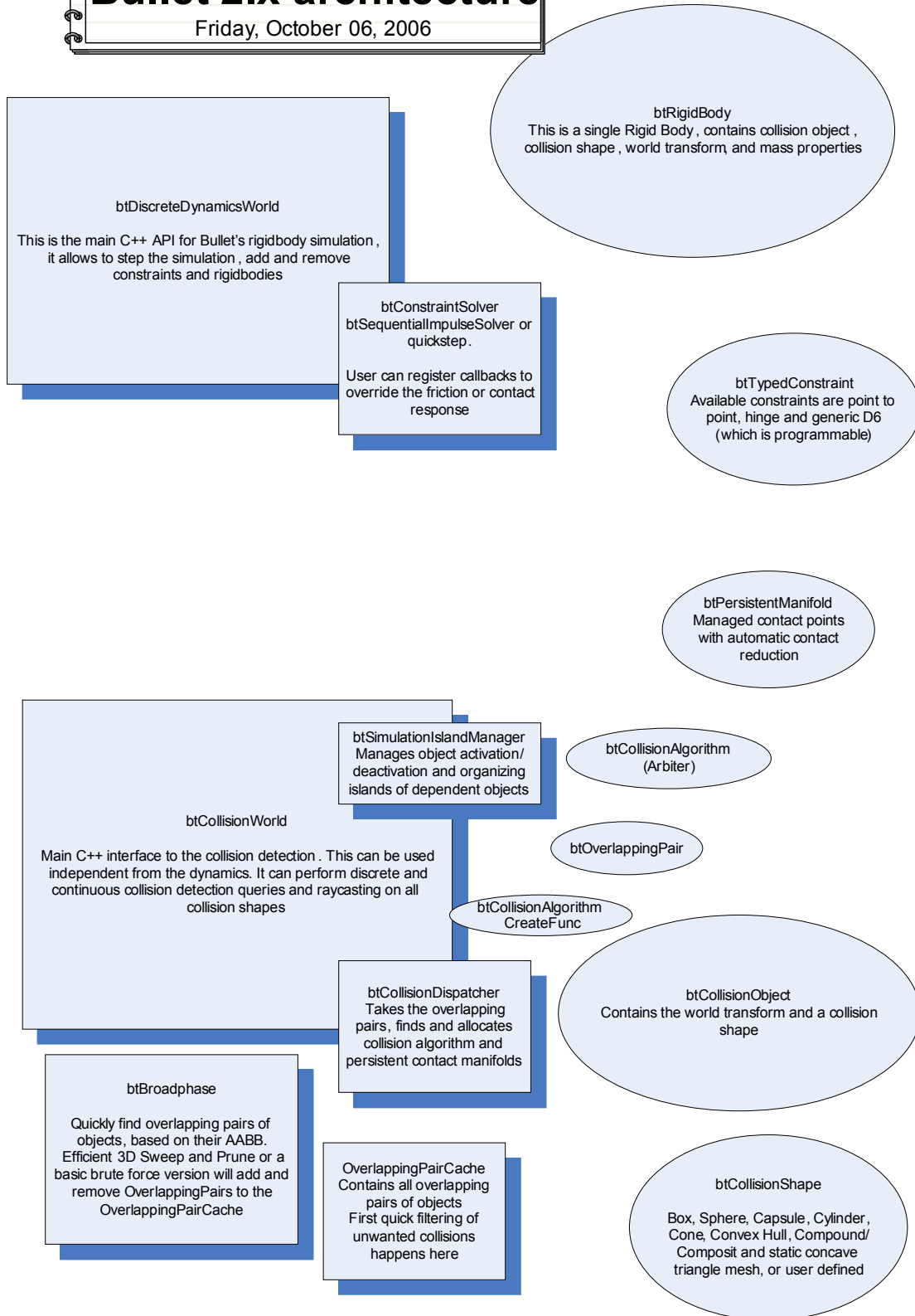
Simplex Demo

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.

Bullet Collision Detection and Physics Architecture

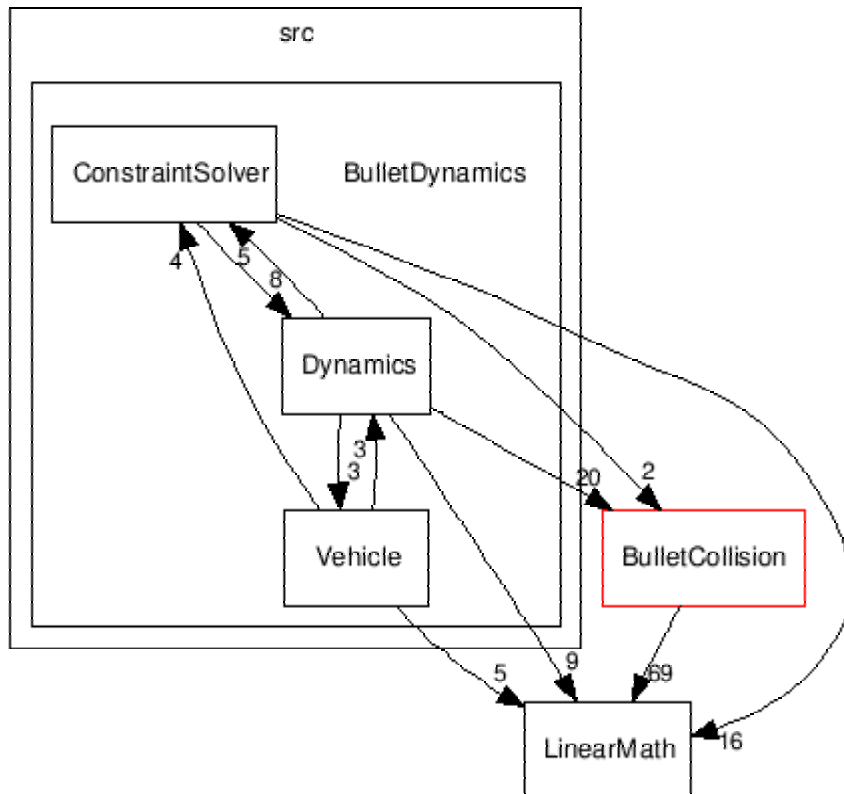
Bullet 2.x architecture

Friday, October 06, 2006



Bullet Library Module Overview

Bullet provides Collision Detection and Rigid Body dynamics. The C++ software is divided into several sub modules with clean dependencies. The division of those modules is reflected in Bullet's directory structure, and further subdirectories are provided per module. This means that the Collision Detection module can be used without using the BulletDynamics module.

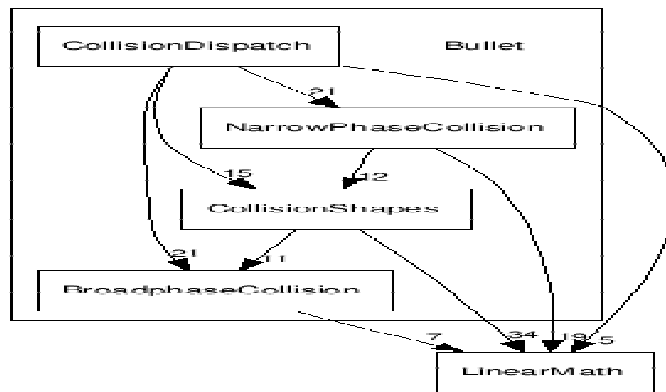


Bullet Collision Detection Library Internals

The main queries provided by the Collision Detection:

- ✓ Closest Distance and closest points
- ✓ Penetration depth calculation
- ✓ Ray cast
- ✓ Sweep API for casting shapes to find Time of Impact (TOI) along a linear path
- ✓ Time of Impact for Continuous Collision Detection including rotations

Supported Collision Shapes include Box, Sphere, Cylinder, Capsule, Minkowski Sum, Convex Hull, (Concave) Triangle Mesh and Compound Shapes and more.



Additional functionality are related to performance and to provide more detail and information useful for the usage in rigid body dynamics and for AI queries in games. The collision pipeline includes 3 stages: Broadphase, Midphase and Narrowphase.

- ✓ Broadphase

Broadphase provides all overlapping pairs based on axis aligned bounding box (AABB). It includes an efficient culling of all potential pairs using the incremental sweep and prune algorithm.

- ✓ Midphase

The midphase performs additional culling for complex collision shapes like compound shapes and static concave triangle meshes. Bullet uses an optimized Bounding Volume Hierarchy, based on a AABB tree and stackless tree traversal. This traversal provides primitives that need to be tested by the Narrowphase.

- ✓ Narrowphase

The Narrowphase perform the actual distance, penetration or time of impact query. Contact points are collected and maintained over several frames in a persistent way. This means that additional information useful for rigid body simulation can be stored in each contact point. Also this means that algorithms that only provide one contact point at a time can still be used, by gathering additional contact points and performing contact point reduction.

Multi threaded version

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/Extras/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

Basic Win32 Threads support has been implemented. Some demos show this preliminary work in action. See #define USE_PARALLEL_DISPATCHER in Demos/BasicDemo, ConcaveDemo and ConvexDecompositionDemo.

IBM Cell SDK 2.1, libspe2 SPU optimized version

IBM also provides a Cell SDK with access to SPU through libspe2 for Cell Blade and PS3 Linux platforms. Libspe2 thread support is currently under development. By providing libspe2 thread support, Bullet can run certain parts on SPU using the same **SpuGatheringCollisionDispatcher**. This will all be available under the ZLib license in Bullet/Extras/BulletMultiThreaded.

Future support for pthreads

If there is interest, pthreads support can be added to support multi threading on various other platforms.

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
Gino van den Bergen, Dtectra: 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
Nathanael Presson, NCSoft : EPA penetration depth calculation
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: several improvements and optimizations
Marcus Hennix, Starbreeze: btConeTwistConstaint etc.

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

Contact

Use either public message or private message (PM) on the Bullet forum at <http://bulletphysics.com>

Or email to bullet <at> erwincoumans.com

Further documentation and references

Bullet Physics website provides most information:

Visit <http://bulletphysics.com> which points to <http://www.continuousphysics.com>

On this website there is online doxygen documentation, a wiki with frequently asked questions and tips, and most important a discussion forum.

A paper describing the Bullet's Continuous Collision Detection method is available online at <http://continuousphysics.com/BulletContinuousCollisionDetection.pdf>

For physics tools and COLLADA physics visit <http://www.khronos.org/collada>
You can find the latest plugin versions and other information at the COLLADA forum at https://collada.org/public_forum/

Links

COLLADA-DOM included in Bullet: <http://colladamaya.sourceforge.net>

ColladaMaya plugin <http://www.feelingsoftware.com>

Blender 3D modeler includes Bullet and COLLADA physics: <http://www.blender.org>

Ageia CreateDynamics tool <http://www.amillionpixels.us/CreateDynamics.zip>

This great tool can perform automatic convex decomposition and create ragdolls from graphics skeletons. It is also available from Ageia support forums at <http://devsupport.ageia.com>

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://image.diku.dk/~kenny>

Very useful to understand Bullet Dynamics and constraints