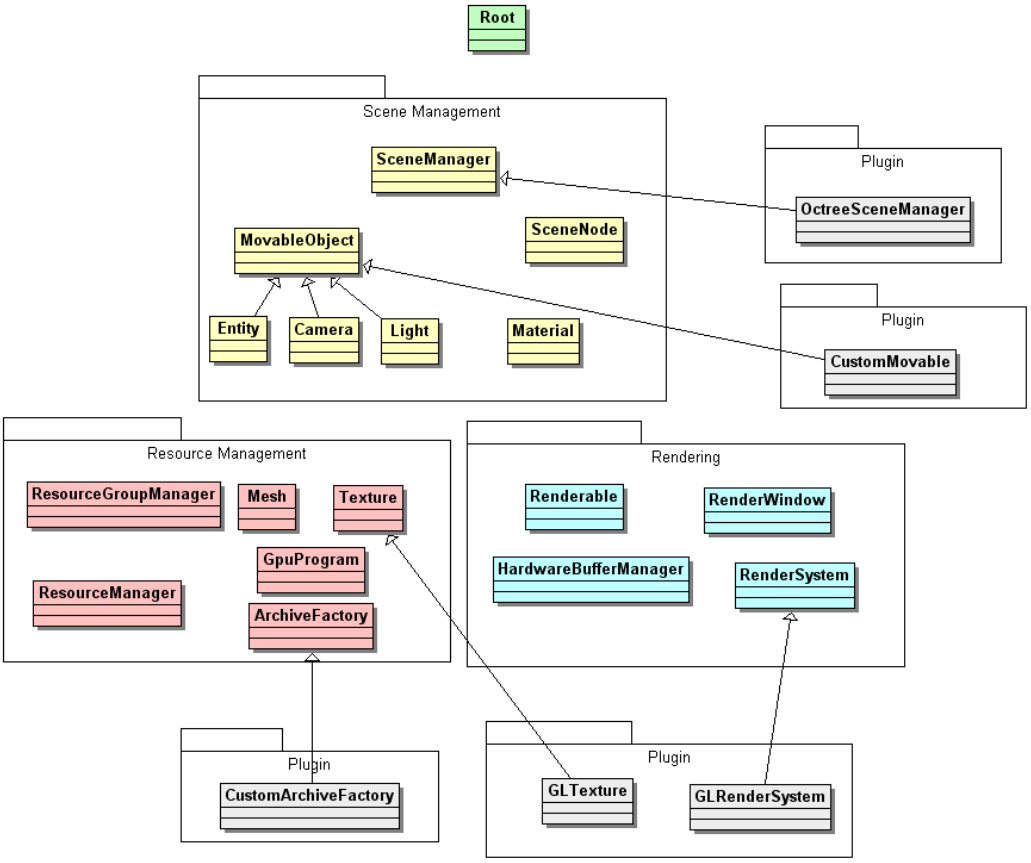
Anexo I: Ogre3D

Ogre3D, cuyo nombre proviene de *Object-Oriented Graphics Rendering Engine* es, como es lógico, un motor de renderizado orientado a objetos. Como característica adicional, está escrito en el lenguaje de programación C++.

No es un motor gráfico como el resto de paquetes de nombres comerciales, como Unity. Ogre3D plantea una alternativa done el usuario puede programar su propio motor gráfico, éste sólo pone las herramientas para hacerlo.

Lo más fácil para comprender Ogre3D es ver su arquitectura desde muy alto nivel. La siguiente figura muestra los objetos más importantes y sobre los que se construye toda la arquitectura.



El objeto Root que se ve en la parte superior de la figura es el punto de acceso del sistema. Aquí es donde se tiende a crear los objetos de más alto nivel con los que tienes que tratar, como los *scene manager*, los sistemas de renderizado o las ventanas de renderización, cargar plugins, y todas los demás componentes fundamentales.

La mayor parte de las demás clases de Ogre3D pertenece a uno de los siguientes tres roles.

Scene Management

Proporciona información sobre el contenido de la escena, de su estructura, de cómo se ve a través de las cámaras, etc. Los objetos que siguen este rol son responsables de ofrecer una interfaz natural para el mundo que estás construyendo. Sirven para no tener que decirle a Ogre3D dónde tiene que poner cada polígono, sino que baste con decirle donde quieres el objeto, con qué materiales y dejarle que se encargue él de lo demás.

Resource Management

Todo el renderizado necesita recursos. La geometría, las texturas, las fuentes, etc. Es importante gestionar a carga, la reutilización y la descarga de los recursos con cuidado. De esto se encargan las clases que siguen este rol.

Rendering

Finalmente, se muestran las imágenes por pantalla. Se trata del nivel más bajo del proceso de renderizado, el sistema específico de la representación de objetos de la API, como búffers, estados de renderizado, etc. Las clases del rol de *Scene Management* utilizan esto para conseguir la información sobre la escena.

Como vemos, Ogre3D tiene la posibilidad casi infinita de conectar plugins al sistema, de manera que no ofrece la solución a un problema determinado, sino que trata de cubrir cualquier problema que pueda surgir y que requiera un motor de renderizado para su solución.

Una vez comprendida la arquitectura general del sistema de Ogre3D y sabiendo de la existencia de los objetos principales que deben der ser construidos, trataremos de mostrar la utilización básica de Ogre3D como medio ilustrativo para dar a entender qué es. Para empezar a comprenderlo, lo mejor es ver un ejemplo donde se crea una escena básica con unas cuantas luces. En el siguiente fragmente de código se expone todo esto comentado de manera que trata de ser comprensible.

```
/**
* Este método se encargará de crear la escena en sí. Antes, hemos
* tenido que configurar todos los objetos fundamentales de Ogre3D,
* como Root, Window, SceneManager, Camera, Viewport y
* ResourceListener.
*
* Esto código sólo pretende demostrar la utilización básica de Ogre3D,
* para información más avanzada, se recomienda visitar la página web
* de Ogre3D (http://www.ogre3d.org/) y más concretamente, los
* tutoriales (http://www.ogre3d.org/tikiwiki/)
*/
void BasicTutorial2::createScene(void)
{
    //Crea una luz ambiental con intensidad 0.5 en RGB
    mSceneMgr->setAmbientLight(Ogre::ColourValue(0.5, 0.5, 0.5));
    //Establece el tipo de sombreado que se utilizará
    mSceneMgr->setShadowTechnique(Ogre::SHADOWTYPE_STENCIL_ADDITIVE);

    //Crea una nueva entidad (objeto que se encarga de la parte
    //visual de cada nodo) a partir de una mesh
    Ogre::Entity* entNinja = mSceneMgr->createEntity("Ninja",
"ninja.mesh");
    //Activa el lanzamiento de sombras por parte de dicha entidad
    entNinja->setCastShadows(true);
    //Crea el nodo (objeto que se encarga del posicionamiento y
    //rotación de los objetos 3D) y lo une con la entidad.
    mSceneMgr->getRootSceneNode()->createChildSceneNode()-
>attachObject(entNinja);
    //Crea un plano
    Ogre::Plane plane(Ogre::Vector3::UNIT_Y, 0);
    //Obtiene la mesh del plano
    Ogre::MeshManager::getSingleton().createPlane("ground",
Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
        plane, 1500, 1500, 20, 20, true, 1, 5, 5,
Ogre::Vector3::UNIT_Z);
    //Crea la entidad del plano
    Ogre::Entity* entGround = mSceneMgr->createEntity("GroundEntity",
"ground");
    //El nodo del plano, y lo une con la entidad.
    mSceneMgr->getRootSceneNode()->createChildSceneNode()-
>attachObject(entGround);
```

```
    //Le aplica un material
    entGround->setMaterialName("Examples/Rockwall");
    //Impide que el plano lance sombras
    entGround->setCastShadows(false);

    //Crea una luz de tipo puntual y la coloca en una posición
    Ogre::Light* pointLight = mSceneMgr->createLight("pointLight");
    pointLight->setType(Ogre::Light::LT_POINT);
    pointLight->setPosition(Ogre::Vector3(0, 150, 250));

    //Establece su color difuso y especular
    pointLight->setDiffuseColour(1.0, 0.0, 0.0);
    pointLight->setSpecularColour(1.0, 0.0, 0.0);

    //Crea de manera similar una luz de tipo direccional
    Ogre::Light* directionalLight = mSceneMgr-
>createLight("directionalLight");
    directionalLight->setType(Ogre::Light::LT_DIRECTIONAL);
    directionalLight->setDiffuseColour(Ogre::ColourValue(.25, .25,
0));
    directionalLight->setSpecularColour(Ogre::ColourValue(.25, .25,
0));

    directionalLight->setDirection(Ogre::Vector3( 0, -1, 1 ));

    //Y por último, una luz de tipo spot
    Ogre::Light* spotLight = mSceneMgr->createLight("spotLight");
    spotLight->setType(Ogre::Light::LT_SPOTLIGHT);
    spotLight->setDiffuseColour(0, 0, 1.0);
    spotLight->setSpecularColour(0, 0, 1.0);

    spotLight->setDirection(-1, -1, 0);
    spotLight->setPosition(Ogre::Vector3(300, 300, 0));

    spotLight->setSpotlightRange(Ogre::Degree(35), Ogre::Degree(50));
}
```

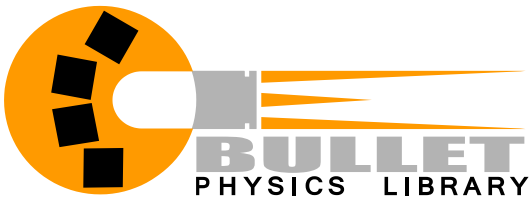
Como vemos, la creación de escenas en Ogre3D es, gracias a los objetos *Scene Manager* muy intuitivo.

Este documento únicamente ha tenido como finalidad mostrar qué es Ogre3D y cuál es la manera más básica de utilizarlo. Para mayor información, lo mejor es acudir:

- Web de Ogre: http://www.ogre3d.org/
- Manual: http://www.ogre3d.org/docs/manual/
- Wiki: http://www.ogre3d.org/tikiwiki/Home
- Foros: http://www.ogre3d.org/forums/
- API: http://www.ogre3d.org/docs/api/1.9/

Anexo II: Manual BulletPhysics

Bullet 2.82 Physics SDK Manual



Also check out the forums and wiki at bulletphysics.org

Table of Contents

1 Introduction

Description 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.

Main Features

- 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
- Discrete and continuous collision detection including ray and convex sweep test. Collision shapes include concave and convex meshes and all basic primitives
- 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
- Maya Dynamica plugin, Blender integration, COLLADA physics import/export support

Contact and Support

- Public forum for support and feedback is available at http://bulletphysics.org
- PLAYSTATION 3 licensed developers can download an optimized version for Cell SPU through Sony PS3 Devnet from https://ps3.scedev.net/projects/spubullet

2 What's New

Preparing for Bullet 3.0 Alpha

- The new Bullet 3.x version is making good progress, and the performance on high-end GPUs such as AMD 7970 and NVIDIA 680 is good. See the github repository at https://github.com/erwincoumans/bullet3

New in Bullet 2.81

- Mostly bug fixes, see https://code.google.com/p/bullet/source/list for details.

New in Bullet 2.81

- Rolling friction, so that spheres and other rounded shapes come to a rest, even when on a sloped surface. See `Demos/RollingFrictionDemo`

- Gear constraint, joint feedback, improved speculative contacts, SIMD and NEON for iOS, improved premake build system support for Linux and Mac OSX.

New in Bullet 2.80

- GPU rigid body pipeline running 100% on the GPU using OpenCL. This is a preview, Bullet 3.x will use this as a GPU backend. See `Bullet/Extras/RigidBodyGpuPipeline`. Requires very recent GPU and drivers (Radeon 5870 or NVIDIA 470 with 290.53 or newer)

- Physics Effects rigid body pipeline, with Android/NEON optimizations. This is a preview, Bullet 3.x will use this as handheld backend. Sony provides a VITA optimized version. See `Bullet/Extras/PhysicsEffects`.

- Option to override the number of constraint solver iterations for individual constraints. See `Bullet/Demos/VoronoiFractureDemo`, `setOverrideNumSolverIterations`

- The open source Dynamica Bullet Maya plugin is now deterministic when restarting the simulation. Also it has preliminary cloth and convex decomposition (HACD) integration. See http://dynamica.googlecode.com for source and precompiled downloads.

- Check the issue tracker at http://bullet.googlecode.com for other improvements.

New in Bullet 2.79

- Bullet 2.79 is mainly a bugfix release. Speculative contacts are disabled, it introduced too many artifacts.

- Hierarchical Approximate Convex Decomposition library, thanks to Khaled Mamou. It can create a convex decomposition from a 3D concave triangle mesh. It replaces John Ratcliff's ConvexDecomposition library. See *Demos/ConvexDecomposition*

- Premake build system to autogenerate Windows Visual Studio project files that can be redistributed. CMake cannot do this properly. We still keep autotools and CMake support up-to-date. See *Bullet/msvc*

New in Bullet 2.78

- Fracture and glueing demo that can break or glue child shapes of a btCompoundShape when certain impulse thresholds are exceeded.
  See *Bullet/Demos/FractureDemo*

- Breakable constraints, based on an applied impulse threshold.
  See *Bullet/Demos/ConstraintDemos*

- Improved binary .bullet file format with soft body serialization and
  See Bullet/Demos/SerializeDemo

- Polyhedral contact clipping and optional separating axis test (SAT) as alternative to GJK/EPA and incremental contact manifold, for btPolyhedralConvexShape derived shapes such as btConvexHullShape
  See *Demos/InternalEdgeDemo*

- OpenCL and DirectCompute cloth simulation improvements: GPU kernels for capsule collision detection and OpenCL-OpenGL interop
  See *Demos/OpenCLClothDemo* and *Demos/DX11ClothDemo*

- Speculative/predictive contact constraints as a method to perform continuous collision response.
  See *Demos/CcdPhysicsDemo*

New in Bullet 2.76

- New binary .bullet file format support, with 32/64bit little/big endian compatibility.
  See Bullet/Demos/SerializeDemo

- New btCollisionWorld::contactTest and btCollisionWorld::contactPairTest query for immediate collision queries with a contact point callback.
  See `Bullet/Demos/CollisionInterfaceDemo`

- New btInternalEdgeUtility to avoid unwanted collisions against internal triangle edges.
  See `Bullet/Demos/InternalEdgeDemo`

- Improved MiniCL support in preparation for Bullet 3.x OpenCL support.
  See `Bullet/Demos/MiniCL_VectorAdd`

- Improved CMake build system support, making Jam and other build systems obsolete.

- Many enhancements and bug fixes, see the issue tracked at bullet.googlecode.com

Maya Dynamica Plugin

- Improved constraint authoring for all constraint types.

- Export to the new .bullet file format.

3 Quickstart

Download

Windows developers can download the zipped sources of Bullet from http://bullet.googlecode.com. Mac OS X, Linux and other developers should download the gzipped tar archive. Bullet should compile out-of-the-box for all platforms, and includes all dependencies.

Building using premake

You can generate projects for most platforms and compilers using premake. Under Windows, you can click on *build/vs2010.bat* to generate Visual Studio projects. Some examples on other platforms are *./premake_osx gmake* or *./premake_osx xcode4* or *./premake_linux64 codeblocks*

Projectfiles for Windows Visual Studio are included in *Bullet/msvc/*/BULLET_PHYSICS.sln*

Building using CMake

CMake adds support for many other build environments and platforms, including Microsoft Visual Studio, XCode for Mac OSX, KDevelop for Linux and Unix Makefiles. Download and install Cmake from http://cmake.org and use the CMake *cmake-gui* tool. See the Appendix for details.
You can also use cmake command-line. Run cmake without arguments to see the list of build system generators for your platform. For Microsoft Visual Studio 8 2005, use

```
cmake . -G "Visual Studio 8 2005"
```

For example to generate Linux or Mac OSX makefiles run

```
cmake . -G "Unix Makefiles"
```
or for Mac OSX Xcode4 use *cmake . -G Xcode*

Building using autotools/automake

Open a shell terminal and go to the Bullet root directory. Then execute

```
./autogen.sh (this is optional and not needed for downloaded packages)
./configure
make
```

Testing demos

Try to run and experiment with the Bullet demos in the *Bullet/Demos* folder.

Integrating Bullet physics in your application

Check out CcdPhysicsDemo how to create a *btDiscreteDynamicsWorld*, *btCollisionShape*, *btMotionState* and *btRigidBody*. Each frame call the *stepSimulation* on the dynamics world, and synchronize the world transform for your graphics object. Requirements:

- *#include "btBulletDynamicsCommon.h"* in your source file
- Required include path: *Bullet/src* folder
- Required libraries: *BulletDynamics*, *BulletCollision*, *LinearMath*

The actual name and of the library might be different. For example the Visual Studio compiler adds the .lib extension, Unix systems usually add a lib prefix and .a extension. When using CMake the library is usually location in the `lib` folder of the build directory.

See the Appendix with details how to setup a Visual Studio and Mac OSX Xcode project.

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 `btCollisionWorld`. Requirements:

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

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*.

4 Library Overview

Introduction

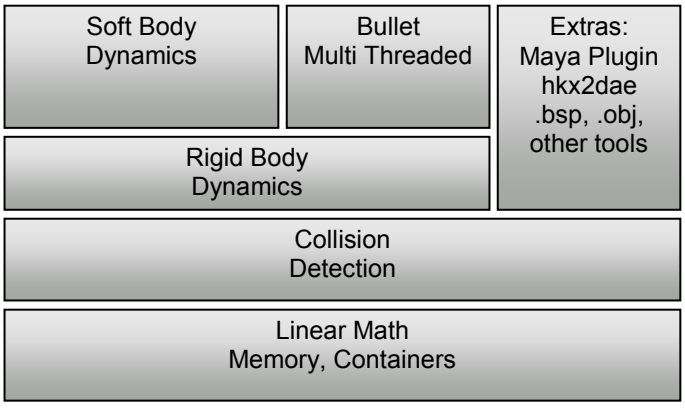
The main task of a physics engine is to perform collision detection, resolve collisions and other constraints, and provide the updated world transform[1] for all the objects. This chapter will give a general overview of the rigid body dynamics pipeline as well as the basic data types and math library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

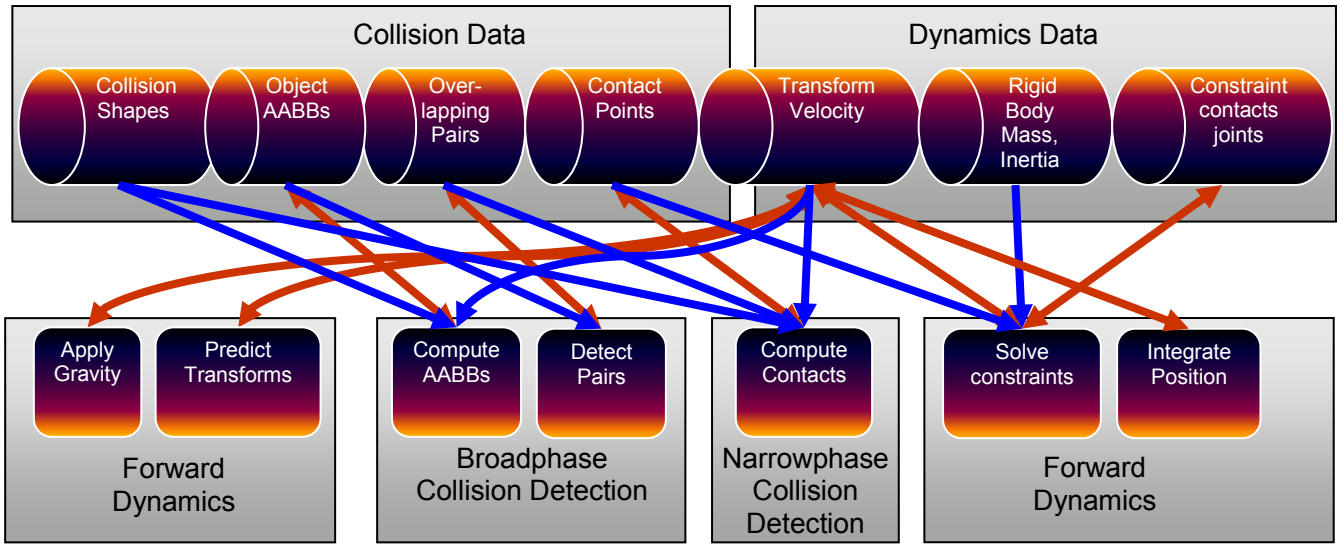
The main components are organized as follows:



[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/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`.

The next chapters will provide more information about collision detection and rigid body dynamics. A lot of the details are demonstrated in the `Bullet/Demos`. 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:

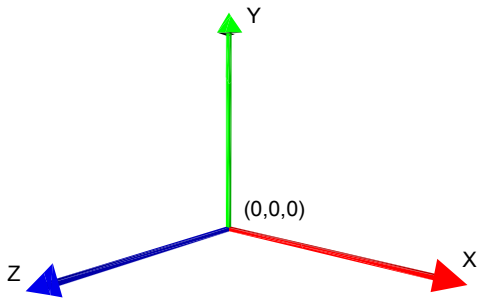


Figure 1 Right-handed coordinate system

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`

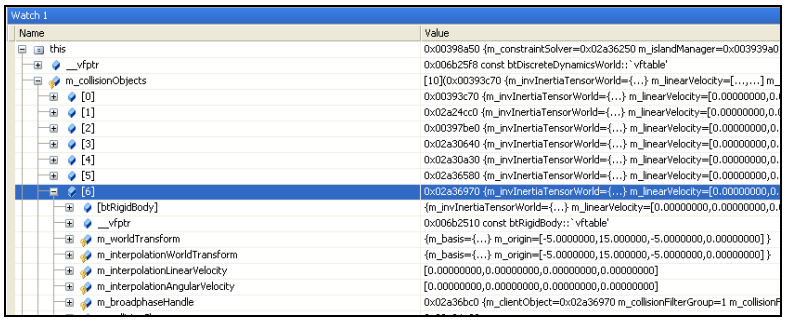


Figure 2 MSVC Debug Visualization

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`

5 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
- *btCudaBroadphase* implements a fast uniform grid using GPU graphics hardware

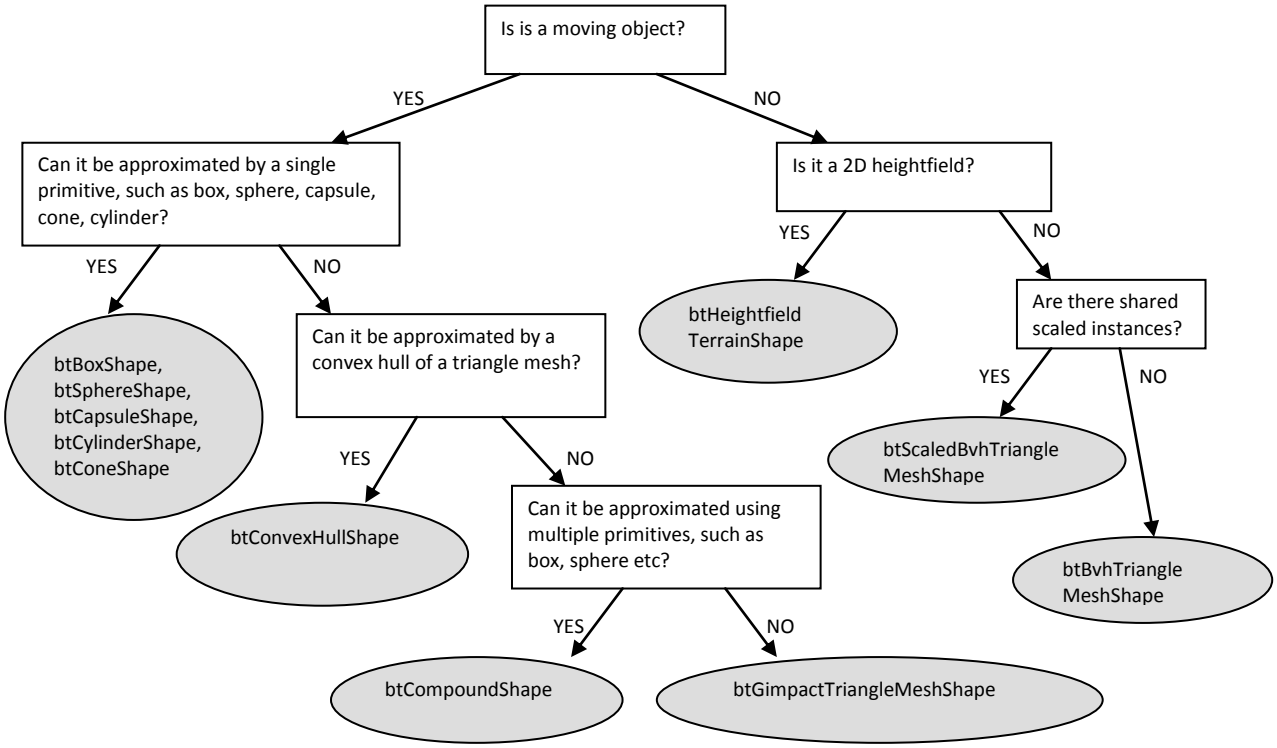
The broadphase adds and removes overlapping pairs from a pair cache. The developer can choose the type of pair cache.

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.

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 *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.* 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 `Demos/TerrainDemo` or `Demos/VehicleDemo` 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 `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 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 `Demos/UserCollisionAlgorithm` for an example, that registers a SphereSphere collision algorithm.

6 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) ;
```

7 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 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`* to proceed.

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.

8 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.

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);
```

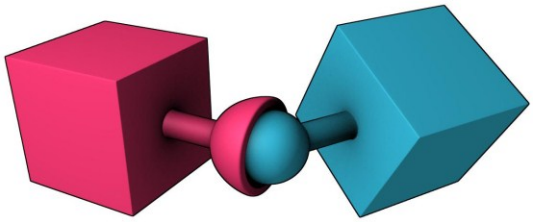


Figure 3 Point to point constraint

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);
```

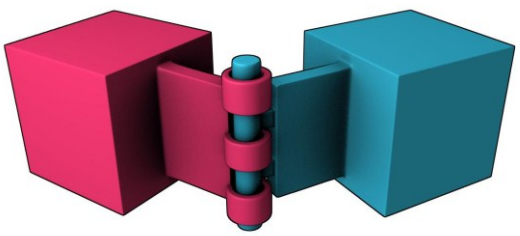


Figure 4 Hinge Constraint

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);
```

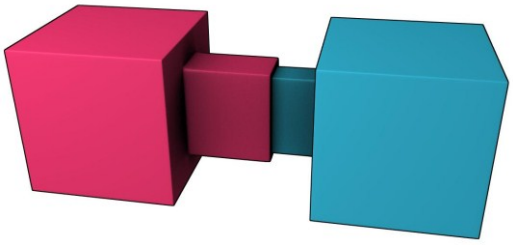


Figure 5 Slider Constraint

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 `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.

```
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);

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

9 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 `Demos/VehicleDemo` 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.

10 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 *Demos/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 and AllBulletDemos 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`.

11 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`

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.

12 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.

13 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 '=' key in most of the Bullet demos to save a 'testFile.bullet'. You can read .bullet files using the btBulletWorldImporter as implemented in the `Bullet/Demos/SerializationDemo`.

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

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

COLLADA Physics format and viewer

COLLADA is rich 3D specification and XML based file format that includes collision and physics information. Dynamica Maya plugin, Blender and other software can export/import this standard physics file format.

The Dynamica source repository at http://dynamica.googlecode.com provides example code to load and save COLLADA Physics files based on the `COLLADA-DOM` and `libxml`. See `Dynamica/Demos/ColladaDemo`.

The `Dynamica/Extras/BulletColladaConverter` class can be used as example for other COLLADA physics integrations.

14 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.

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.

15 Parallelism: OpenCL, Compute, SPU, multi thread

OpenCL and Direct Compute cloth simulation

Parts of the CPU intensive innerloop of the cloth and soft body library have been implemented using OpenCL and DirectX 11 DirectCompute kernels. See *Demos/OpenCLClothDemo* and *Demos/DX11ClothDemo* for examples.

Cell SPU / SPURS optimized version

Bullet collision detection and physics 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. See *Bullet/ThreadingDemo* for an example.

Win32 Threads, pthreads, sequential thread support

Basic Win32 Threads, pthreads and sequential thread support is available to execute the parallel constraint solver and parallel collision dispatcher. See *Demos/BulletMultiThreaded* for an example.

16 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

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 and continues doing so full-time at Advanced Micro Devices.

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

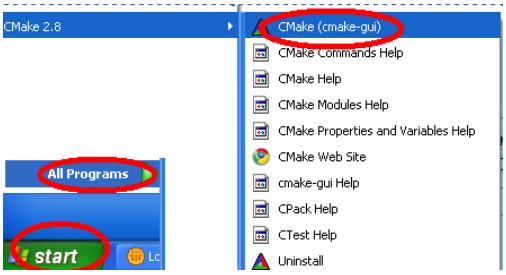
Many more people have contributed to Bullet, thanks to everyone on the Bullet forums.

(please get in touch if your name should be in this list)

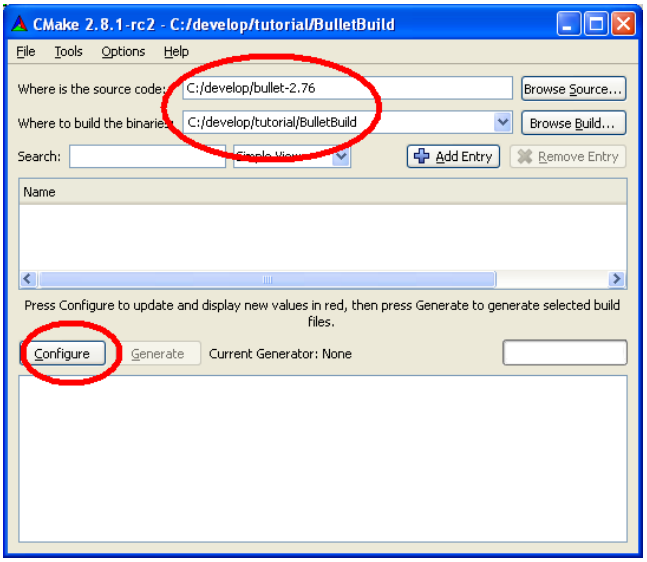
Create Bullet Visual Studio projectfiles using CMake

First, download Bullet from http://bullet.googlecode.com and CMake from http://cmake.org

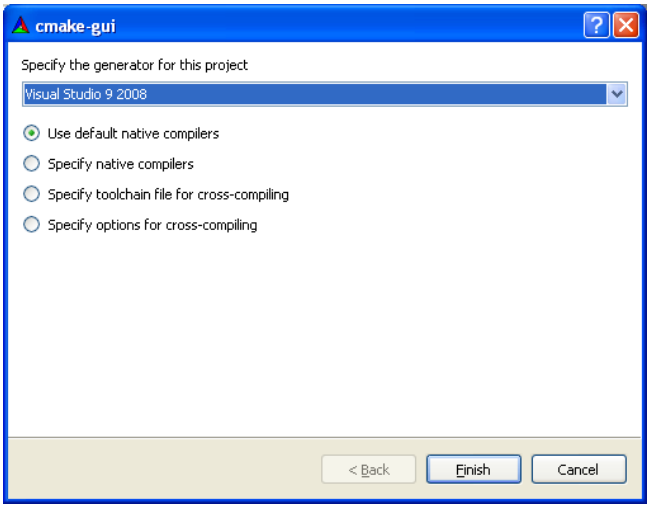
1. Run CMake-gui



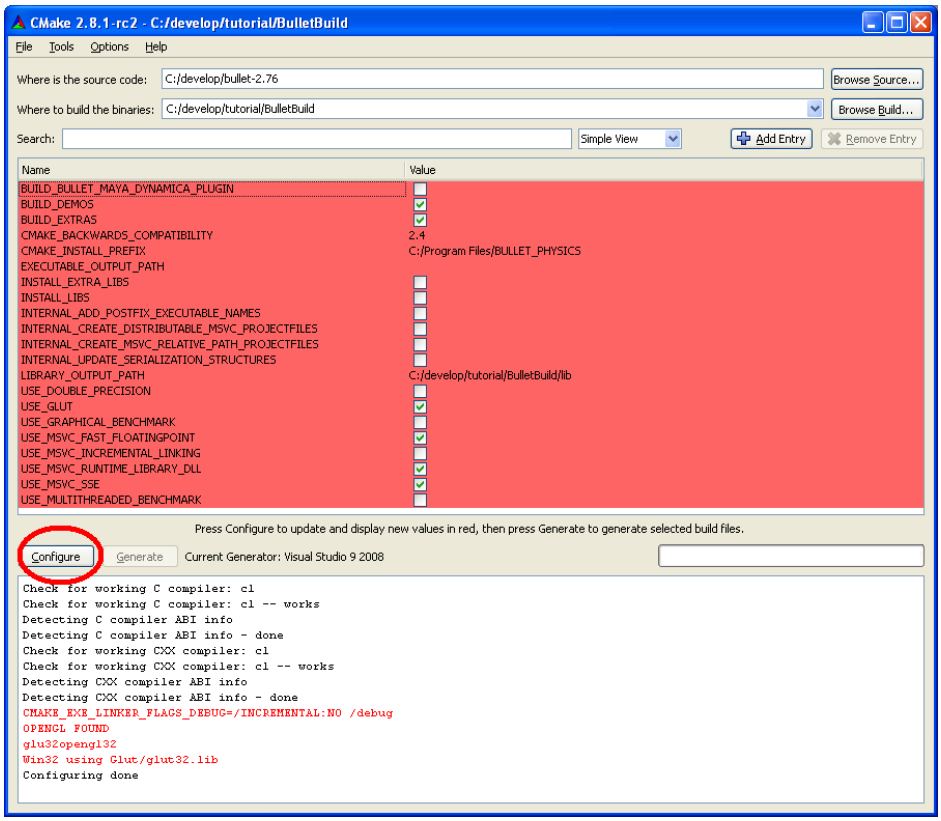
2. Choose the location where you unzipped the Bullet source code, and where you build the binaries for your own project and press Configure



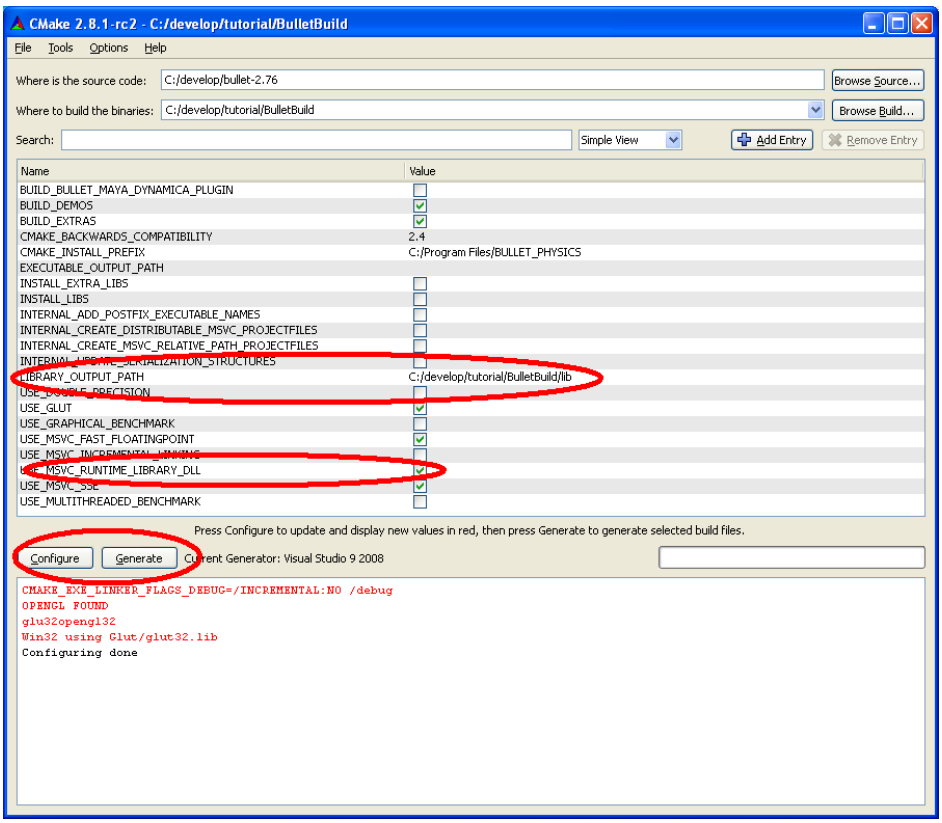
3. Choose the compiler, such as Microsoft Visual Studio 2008



4. Review the settings and press Configure again



5. Make sure the Run-time library is the same as your project (by default it is set to Run-time library DLL) and press Generate



6. Optionally, you could open the generated project solution (in our case `C:\develop\tutorial\BulletBuild\BULLET_PHYSICS.sln`) and build the library and demos, but this is not necessary

See also the Bullet Wiki on how to create your own project using Visual Studio

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

Anexo III: Documentación del código

Ogre3D + BulletPhysics TFG

Generated by Doxygen 1.8.8

Thu Nov 20 2014 18:54:17

Contents

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

Class Documentation

3.1 BaseApplication Class Reference

Main class of the application, this will run it.

`#include <BaseApplication.h>`

Public Member Functions

- BaseApplication (void)
  Empty constructor,.
- virtual ~BaseApplication (void)
  Destructor,.
- virtual void go (void)
  Launches the application.

Protected Member Functions

- virtual bool setup ()
  Initialises main components of the application.
- virtual bool configure (void)
  Show the configuration dialog and initialise the system.
- virtual void chooseSceneManager (void)
  Create and choose the scene manager.
- virtual void createCamera (void)
  Create the camera.
- virtual void createScene (void)
  Create the Scene and all the 3D objects in it.
- virtual void destroyScene (void)
  Destroy the Scene.
- virtual void createViewports (void)
  Create the viewports.
- virtual void setupResources (void)
  Setup the resources to be loaded.
- virtual void createResourceListener (void)
  Create the listener of the resources when they are loaded.
- virtual void loadResources (void)
  Load the external resources such as meshes, images, etc.

Protected Attributes

- Ogre::Root * mRoot
- Ogre::Camera * **mCamera**
- Ogre::SceneManager * **mSceneMgr**
- Ogre::RenderWindow * **mWindow**
- Ogre::String **mResourcesCfg**
- Ogre::String **mPluginsCfg**
- Ogre::OverlaySystem * **mOverlaySystem**
- OgreBites::InputContext **mInputContext**
- OgreBites::ParamsPanel * **mDetailsPanel**
- bool **mCursorWasVisible**
- bool **mShutDown**
- Ogre::String **m_ResourcePath**
- Physics * mPhysicsEngine
- MyFrameListener * mFrameListener
- MyCameraController * myCamera
- Character * mChar

3.1.1 Detailed Description

Main class of the application, this will run it.

Class based in the lass that appears in the tutorials of Ogre3D. That one can be downloaded looking for "Ogre3D Tutorial Framework"

3.1.2 Member Data Documentation

3.1.2.1 Character* BaseApplication::mChar [protected]

Main character object

3.1.2.2 MyFrameListener* BaseApplication::mFrameListener [protected]

Frame listener

3.1.2.3 Physics* BaseApplication::mPhysicsEngine [protected]

Physics engine

3.1.2.4 Ogre::Root* BaseApplication::mRoot [protected]

Entry point of the system

3.1.2.5 MyCameraController* BaseApplication::myCamera [protected]

Camera controller

The documentation for this class was generated from the following files:

- BaseApplication.h
- BaseApplication.cpp

3.2 Character Class Reference

Class in charge of manage all the character stuff, create it, update it, etc.

`#include <Character.h>`

Public Member Functions

- Character (Ogre::String aName, Ogre::SceneManager *aSceneMgr, MyCameraController *aCam, CharacterController *aCController, btPairCachingGhostObject *ghostObject, btConvexShape *convex↩ Shape, btScalar stepHeight, btSoftRigidDynamicsWorld *theWorld, Ogre::Vector3 &origin, int upAxis=1)

  Constructor of the class.
- virtual ~Character ()

  Destructor of the class.
- virtual void setResetSight (bool boolean)

  Setter of the resetSight boolean.
- virtual CharacterController * getCController ()

  Getter of the character animation controller.
- virtual CharacterPhysics * getCPhysics ()

  Getter of the physics engine.
- virtual Ogre::SceneNode * getMainNode ()

  Getter of the character physics controller.
- void injectKeyDown (const OIS::KeyEvent &evt)

  Method used to receive keyboard push events from the frame listener.
- void injectKeyUp (const OIS::KeyEvent &evt)

  Method used to receive keyboard releases events from the frame listener.
- void injectMouseMove (const OIS::MouseEvent &evt)

  Method used to receive mouse movements events from the frame listener.
- void updateCharacter (Ogre::Real dt)

  Method used to update the character given an amount of elapsed time.

Protected Attributes

- MyCameraController * mCam
- Ogre::SceneManager * mSceneMgr
- Ogre::String mName
- bool resetSight
- Ogre::SceneNode * mMainNode
- Ragdoll * mRagdoll
- CharacterPhysics * mPhysics
- CharacterController * mCController
- Ogre::Vector3 mWalkDirection
- Ogre::Vector3 mGoalDirection
- Ogre::Vector3 mKeyDirection
- bool mIsFalling
- bool mJumped

3.2.1 Detailed Description

Class in charge of manage all the character stuff, create it, update it, etc.

I took some ideas from this tutorial (reading strongly recommended): http://codefreax.org/tutorials/view/id/3

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Character::Character (Ogre::String *aName,* Ogre::SceneManager * *aSceneMgr,* MyCameraController * *aCam,* CharacterController * *aCController,* btPairCachingGhostObject * *ghostObject,* btConvexShape * *convexShape,* btScalar *stepHeight,* btSoftRigidDynamicsWorld * *theWorld,* Ogre::Vector3 & *origin,* int *upAxis* = `1`)

Constructor of the class.

Parameters

aName	the name
aSceneMgr	the scene manager
aCam	the camera controller
aCControler	the character animations controller
ghostObject	the ghost object that will simulate the character physical behaviour
convexShape	the shape of the ghost object
stepHeigth	the height of one step
theWorld	the physics' world
origin	the initial position of the character
upAxis	the axis that are in the "up" direction

3.2.3 Member Function Documentation

3.2.3.1 virtual CharacterController* Character::getCController ( ) [inline], [virtual]

Getter of the character animation controller.

Returns

the character animation controller

3.2.3.2 virtual CharacterPhysics* Character::getCPhysics ( ) [inline], [virtual]

Getter of the physics engine.

Returns

the physics engine

3.2.3.3 virtual Ogre::SceneNode* Character::getMainNode ( ) [inline], [virtual]

Getter of the character physics controller.

Returns

the character physics controller

3.2.3.4 void Character::injectKeyDown (const OIS::KeyEvent & *evt*)

Method used to receive keyboard push events from the frame listener.

Parameters

evt	the event

3.2.3.5 void Character::injectKeyUp (const OIS::KeyEvent & *evt*)

Method used to receive keyboard releases events from the frame listener.

Parameters

evt	the event

3.2.3.6 void Character::injectMouseMove (const OIS::MouseEvent & *evt*)

Method used to receive mouse movements events from the frame listener.

Parameters

evt	the event

3.2.3.7 virtual void Character::setResetSight (bool *boolean*) [inline], [virtual]

Setter of the resetSight boolean.

Parameters

resetSight	the new resetSight boolean

3.2.3.8 void Character::updateCharacter (Ogre::Real *dt*)

Method used to update the character given an amount of elapsed time.

Parameters

dt	time elapsed

3.2.4 Member Data Documentation

3.2.4.1 MyCameraController* Character::mCam [protected]

The Camera Controller

3.2.4.2 CharacterController* Character::mCController [protected]

Character animation controller

3.2.4.3 Ogre::Vector3 Character::mGoalDirection [protected]

vector3 standing of the goal position direction

3.2.4.4 bool Character::mIsFalling [protected]

boolean used for checking if the main character is falling

3.2.4.5 bool Character::mJumped [protected]

boolean used for checking if the main character is jumping

3.2.4.6 Ogre::Vector3 Character::mKeyDirection [protected]

vector3 standing of the key pushed direction

3.2.4.7 Ogre::SceneNode* Character::mMainNode [protected]

Node of the Character

3.2.4.8 Ogre::String Character::mName [protected]

A name of the character

3.2.4.9 CharacterPhysics* Character::mPhysics [protected]

Character physics controller

3.2.4.10 Ragdoll* Character::mRagdoll [protected]

RagDoll of the Character

3.2.4.11 Ogre::SceneManager* Character::mSceneMgr [protected]

The scene manager of the app

3.2.4.12 Ogre::Vector3 Character::mWalkDirection [protected]

vector3 standing of the walk direction

3.2.4.13 bool Character::resetSight [protected]

Boolean used for inner calculations, reset the camera in the next frame when it is true

The documentation for this class was generated from the following files:

- Character.h
- Character.cpp

3.3 CharacterController Class Reference

The class that will manage the animations of the main character.

`#include <CharacterController.h>`

Public Member Functions

- CharacterController (Ogre::SceneManager *sceneMgr, Ogre::Vector3 &origin)
  Constructor of the class.
- void addTime (Ogre::Real deltaTime)
  Method that add time to the current timer.
- Ogre::SceneNode * getBodySceneNode ()

Getter of the main char scene node.

- Ogre::Vector3 getPosition () const

  Getter of the actual main char position.

- void animRunStart ()

  Starts the run animation.

- void animRunEnd ()

  Finish the run animation.

- void animJumpStart ()

  Starts the jump animation.

- void animJumpLoop ()

  Enter in the jump loop animation.

- void animJumpEnd ()

  Finish the jump animation.

- void animSliceStart ()

  Starts the slice animation.

- void animSliceEnd ()

  Finish the slice animation.

- void setIsMoving (bool isMoving)

  Setter of the isMoving boolean.

3.3.1 Detailed Description

The class that will manage the animations of the main character.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 CharacterController::CharacterController (Ogre::SceneManager * *sceneMgr*, Ogre::Vector3 & *origin*)

Constructor of the class.

Parameters

sceneMgr	the Ogre3D scene manager.
origin	the position in which the caracter starts.

3.3.3 Member Function Documentation

3.3.3.1 void CharacterController::addTime (Ogre::Real *deltaTime*)

Method that add time to the current timer.

Parameters

deltaTime	the time to be added.

3.3.3.2 Ogre::SceneNode * CharacterController::getBodySceneNode ( )

Getter of the main char scene node.

Returns

the main char node.

3.3.3.3 Ogre::Vector3 CharacterController::getPosition ( ) const

Getter of the actual main char position.

Returns

vector with the actual position of the main char.

3.3.3.4 void CharacterController::setIsMoving (bool *isMoving*)

Setter of the isMoving boolean.

Parameters

isMoving	the new bool isMoving.

The documentation for this class was generated from the following files:

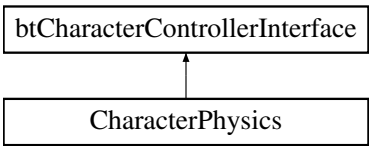
- CharacterController.h
- CharacterController.cpp

3.4 CharacterPhysics Class Reference

This class will manage the physical behaviour of the main character.

`#include <CharacterPhysics.h>`

Inheritance diagram for CharacterPhysics:



Public Member Functions

- btPairCachingGhostObject ∗ getGhostObject ()

  Getter of the ghost object.
- CharacterPhysics (btPairCachingGhostObject ∗ghostObject, btConvexShape ∗convexShape, btScalar step↩ Height, btCollisionWorld ∗collisionWorld, int upAxis=1)

  Constructor of the class.
- void **setDuckingConvexShape** (btConvexShape ∗shape)
- bool **recoverFromPenetration** (btCollisionWorld ∗collisionWorld)
- void **stepUp** (btCollisionWorld ∗collisionWorld)
- void **setRBForceImpulseBasedOnCollision** ()
- void **updateTargetPositionBasedOnCollision** (const btVector3 &hitNormal, btScalar tangentMag=0, bt↩ Scalar normalMag=1)
- void **stepForwardAndStrafe** (btCollisionWorld ∗collisionWorld, const btVector3 &walkMove)
- void **stepDown** (btCollisionWorld ∗collisionWorld, btScalar dt)
- void setVelocityForTimeInterval (const btVector3 &velocity, btScalar timeInterval)

  Inherited function.
- void reset (btCollisionWorld ∗collisionWorld)

  Inherited function.

- void warp (const btVector3 &origin)

  Inherited function.
- void preStep (btCollisionWorld *collisionWorld)

  Setup everything in the world to make a main char movement collisionWorld The collision physic world.
- void playerStep (btCollisionWorld *collisionWorld, btScalar dt)

  Actually makes the character move.
- void setFallSpeed (btScalar fallSpeed)

  Setter of the fall speed.
- void setJumpSpeed (btScalar jumpSpeed)

  Setter of the jump speed.
- void setMaxJumpHeight (btScalar maxJumpHeight)

  Setter of the max jump height.
- bool canJump () const

  Checks if the main char can jump.
- void jump ()

  Makes the character jump.
- void duck ()

  Deprecated. Makes the character duck.
- void stand ()

  If the char is standing of the floor, makes it stand.
- bool canStand ()

  Check if the character is already on the floor.
- void move (bool forward, bool backward, bool left, bool right)

  Move the char in one or more directions.
- void setGravity (const btScalar gravity)

  Set the gravity value.
- btScalar getGravity () const

  Get the gravity value.
- void **setMaxSlope** (btScalar slopeRadians)
- btScalar **getMaxSlope** () const
- bool **onGround** () const
- void **setWalkDirection** (const btVector3 &walkDirection)
- void **setWalkDirection** (const btScalar x, const btScalar y, const btScalar z)
- void **setOrientation** (const btQuaternion &orientation)
- btVector3 **getWalkDirection** () const
- btVector3 **getPosition** () const
- void **debugDraw** (btIDebugDraw *debugDrawer)
- void **setUpInterpolate** (bool value)
- void **updateAction** (btCollisionWorld *collisionWorld, btScalar dt)

3.4.1 Detailed Description

This class will manage the physical behaviour of the main character.

Hardly based in the class that shows how to implement kinematic character controllers in the Bullet Physics demos.

Link: "http://bulletphysics.org/Bullet/BulletFull/classbtKinematicCharacterController.html"

3.4.2 Constructor & Destructor Documentation

3.4.2.1 CharacterPhysics::CharacterPhysics (btPairCachingGhostObject * *ghostObject*, btConvexShape * *convexShape*, btScalar *stepHeight*, btCollisionWorld * *collisionWorld*, int *upAxis* = `1`)

Constructor of the class.

Parameters

ghostObject	the ghostObject that will simulate the physical behaviour of the character
convesShape	the shape of the ghostObject
stepHeight	the height of a single step
collisionWorld	the physics collision world
upAxis	the axis that are in the "up" direction

3.4.3 Member Function Documentation

3.4.3.1 bool CharacterPhysics::canJump ( ) const

Checks if the main char can jump.

Returns

the result of the check

3.4.3.2 btPairCachingGhostObject * CharacterPhysics::getGhostObject ( )

Getter of the ghost object.

Returns

the ghost object

3.4.3.3 btScalar CharacterPhysics::getGravity ( ) const

Get the gravity value.

Returns

the gravity value

3.4.3.4 void CharacterPhysics::move (bool *forward,* bool *backward,* bool *left,* bool *right*)

Move the char in one or more directions.

Parameters

forward	move the char fowards
backward	move the char backwards
left	move the char to the left
right	move the char to the right

3.4.3.5 void CharacterPhysics::playerStep (btCollisionWorld * *collisionWorld,* btScalar *dt*)

Actually makes the character move.

Parameters

collisionWorld	The collision physic world
dt	Time elapsed

3.4.3.6 void CharacterPhysics::setFallSpeed (btScalar *fallSpeed*)

Setter of the fall speed.

Parameters

fallSpeed	the new fall speed

3.4.3.7 void CharacterPhysics::setGravity (const btScalar *gravity*)

Set the gravity value.

Parameters

gravity	the new gravity value

3.4.3.8 void CharacterPhysics::setJumpSpeed (btScalar *jumpSpeed*)

Setter of the jump speed.

Parameters

jumpSpeed	the new jump speed

3.4.3.9 void CharacterPhysics::setMaxJumpHeight (btScalar *maxJumpHeight*)

Setter of the max jump height.

Parameters

maxJumpHeight	the new jump height

The documentation for this class was generated from the following files:

- CharacterPhysics.h
- CharacterPhysics.cpp

3.5 MyCameraController Class Reference

Class that will manage the camera movements and updates.

`#include <MyCameraController.h>`

Public Member Functions

- MyCameraController (Ogre::Camera *cam=NULL, Ogre::SceneManager *scnMgr=NULL, Ogre::String name="", CharacterController *aCController=NULL)

  Constructor of the class.
- virtual ~MyCameraController ()

Destructor of the class.

- virtual void update (const Ogre::Real &timeSinceLastFrame)

  This will update the camera in function of the time elapsed.
- virtual void reset ()

  Reset the camera to it's original position.
- virtual void injectMouseMove (const OIS::MouseEvent &arg, bool moveChar)

  Used to receive mouse movements events from the frame listener.
- virtual void adjustZoom (const OIS::MouseEvent &arg)

  Used to receive mouse scroll events from the frame listener.
- virtual Ogre::Camera * getCamera ()

  Getter of the camera.
- virtual bool getMoveCam ()

  Getter of the moveCam bool.
- virtual void setMoveCam (bool move)

  setter of the moveCam bool.
- virtual Ogre::SceneNode * getCameraNode ()

  Getter of the camera node.
- virtual Ogre::SceneNode * getSightNode ()

  Getter of the sight camera node.
- virtual Ogre::SceneNode * getDesiredCameraNode ()

  Getter of the desired camera node.

Protected Attributes

- Ogre::Camera * mCamera
- Ogre::SceneNode * mCameraNode
- CharacterController * mCController
- Ogre::SceneNode * mMainNode
- Ogre::SceneNode * mSightNode
- Ogre::SceneNode * mDesiredCameraNode
- Ogre::Real mPivotYRot
- Ogre::SceneManager * mSceneMgr
- Ogre::String mName
- Ogre::Real mZoomFactor
- Ogre::Real mRotationFactor
- Ogre::Real **mTightness**
- bool **moveCam**

3.5.1 Detailed Description

Class that will manage the camera movements and updates.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 MyCameraController::MyCameraController (Ogre::Camera * *cam* = NULL, Ogre::SceneManager * *scnMgr* = NULL, Ogre::String *name* = " ", CharacterController * *aCController* = NULL)

Constructor of the class.

Parameters

cam	The Ogre3D camera node.
scnMgr	The ogre3D scene manager, for managing purposes.
name	Name for the Camera Controller.
aCControler	A character controller is necessary, for managing purposes.

3.5.3 Member Function Documentation

3.5.3.1 void MyCameraController::adjustZoom (const OIS::MouseEvent & *arg*) [virtual]

Used to receive mouse scroll events from the frame listener.

Parameters

arg	the mouse event.

3.5.3.2 virtual Ogre::Camera* MyCameraController::getCamera ( ) [inline], [virtual]

Getter of the camera.

Returns

the Ogre3D camera.

3.5.3.3 virtual Ogre::SceneNode* MyCameraController::getCameraNode ( ) [inline], [virtual]

Getter of the camera node.

Returns

the Ogre3D node of the camera.

3.5.3.4 virtual Ogre::SceneNode* MyCameraController::getDesiredCameraNode ( ) [inline], [virtual]

Getter of the desired camera node.

Returns

the Ogre3D node of the desired camera.

3.5.3.5 virtual bool MyCameraController::getMoveCam ( ) [inline], [virtual]

Getter of the moveCam bool.

Returns

the moveCam bool.

3.5.3.6 virtual Ogre::SceneNode* MyCameraController::getSightNode ( ) [inline], [virtual]

Getter of the sight camera node.

Returns

the Ogre3D node of the sight camera.

3.5.3.7 void MyCameraController::injectMouseMove (const OIS::MouseEvent & *arg*, bool *moveChar*) [virtual]

Used to receive mouse movements events from the frame listener.

Parameters

arg	the mouse event.
moveChar	boolean used to check if not only the camera should be rotated but also the character.

3.5.3.8 virtual void MyCameraController::setMoveCam (bool *move*) [inline], [virtual]

setter of the moveCam bool.

Parameters

move	the new moveCam bool.

3.5.3.9 void MyCameraController::update (const Ogre::Real & *timeSinceLastFrame*) [virtual]

This will update the camera in function of the time elapsed.

Parameters

timeSinceLast↩ Frame	the time elapsed.

3.5.4 Member Data Documentation

3.5.4.1 Ogre::Camera* MyCameraController::mCamera [protected]

Ogre3D camera

3.5.4.2 Ogre::SceneNode* MyCameraController::mCameraNode [protected]

The camera itself

3.5.4.3 CharacterController* MyCameraController::mCController [protected]

The Character controller, used to move it if it's necessary

3.5.4.4 Ogre::SceneNode* MyCameraController::mDesiredCameraNode [protected]

Node where the camera wants to be

3.5.4.5 Ogre::SceneNode* MyCameraController::mMainNode [protected]

Node where the character is

3.5.4.6 Ogre::String MyCameraController::mName [protected]

The name of the CameraController

3.5.4.7 Ogre::Real MyCameraController::mPivotYRot [protected]

Inner calculations for rotation

3.5.4.8 Ogre::Real MyCameraController::mRotationFactor [protected]

Factor the movement of the mouse will be multiplied for to adjust the rotation

3.5.4.9 Ogre::SceneManager* MyCameraController::mSceneMgr [protected]

The SceneManager, used to managing purposes

3.5.4.10 Ogre::SceneNode* MyCameraController::mSightNode [protected]

Node where the character is looking at (pivot point)

3.5.4.11 Ogre::Real MyCameraController::mZoomFactor [protected]

Factor the scroll of the mouse will be multiplied for to adjust the zoom

The documentation for this class was generated from the following files:

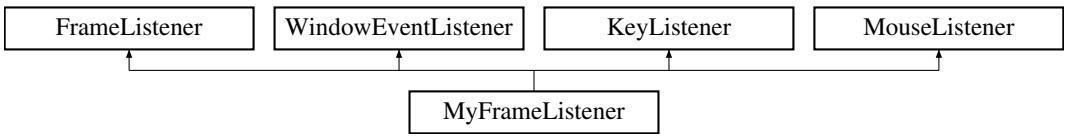
- MyCameraController.h
- MyCameraController.cpp

3.6 MyFrameListener Class Reference

Class that will be in charge of managing everything that happens every frame. From reading inputs to update the main character, the camera and the physical world.

`#include <MyFrameListener.h>`

Inheritance diagram for MyFrameListener:



Public Member Functions

- MyFrameListener (Ogre::RenderWindow *win, Ogre::SceneManager *mgr, Physics *pEngine, Character *aChar, MyCameraController *aCam)

  Constructor of the class.
- ~MyFrameListener ()

  Destructor of the class.
- bool frameStarted (const Ogre::FrameEvent &evt)

  Overwritten method that will be called every time a frame rendering is started.
- bool frameRenderingQueued (const Ogre::FrameEvent &evt)

  Overwritten method that will be called every time a frame rendering is beeing done.
- bool frameEnded (const Ogre::FrameEvent &evt)

Overwritten method that will be called every time a frame rendering is finished.

- bool keyPressed (const OIS::KeyEvent &evt)

  Overwritten method that will be called every time a keyboard key is pressed.
- bool mouseMoved (const OIS::MouseEvent &arg)

  Overwritten method that will be called every time the mouse is moved.
- bool mousePressed (const OIS::MouseEvent &arg, OIS::MouseButtonID id)

  Overwritten method that will be called every time a mouse button is pressed.
- bool mouseReleased (const OIS::MouseEvent &arg, OIS::MouseButtonID id)

  Overwritten method that will be called every time a mouse button is released.
- bool keyReleased (const OIS::KeyEvent &arg)

  Overwritten method that will be called every time a keyboard key is released.
- void windowResized (Ogre::RenderWindow *rw)

  Overwritten method that will be called every time the main rendered window is resized.
- void windowClosed (Ogre::RenderWindow *rw)

  Overwritten method that will be called every time the main rendered window is closed.
- virtual void setDebugDrawer (CDebugDraw *aD)

  Setter of the Debug System.

3.6.1 Detailed Description

Class that will be in charge of managing everything that happens every frame. From reading inputs to update the main character, the camera and the physical world.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 MyFrameListener::MyFrameListener (Ogre::RenderWindow * *win*, Ogre::SceneManager * *mgr*, Physics * *pEngine*, Character * *aChar*, MyCameraController * *aCam*)

Constructor of the class.

Parameters

win	the render window that will be updated
mgr	the scene manager to update everything that has to be.
pEnging	the physics engine that wants to be updated in each frame.
aChar	the main character that is wanted to be updated in each frame.
aCam	the camera controller that is wanted to be updated in each fram.

3.6.3 Member Function Documentation

3.6.3.1 bool MyFrameListener::frameEnded (const Ogre::FrameEvent & *evt*)

Overwritten method that will be called every time a frame rendering is finished.

Parameters

evt	the event that called the method.

3.6.3.2 bool MyFrameListener::frameRenderingQueued (const Ogre::FrameEvent & *evt*)

Overwritten method that will be called every time a frame rendering is beeing done.

Parameters

evt	the event that called the method.

3.6.3.3 bool MyFrameListener::frameStarted (const Ogre::FrameEvent & *evt*)

Overwritten method that will be called every time a frame rendering is started.

Parameters

evt	the event that called the method.

3.6.3.4 bool MyFrameListener::keyPressed (const OIS::KeyEvent & *evt*)

Overwritten method that will be called every time a keyboard key is pressed.

Parameters

evt	the event that called the method.

3.6.3.5 bool MyFrameListener::keyReleased (const OIS::KeyEvent & *arg*)

Overwritten method that will be called every time a keyboard key is released.

Parameters

arg	the event that called the method.

3.6.3.6 bool MyFrameListener::mouseMoved (const OIS::MouseEvent & *arg*)

Overwritten method that will be called every time the mouse is moved.

Parameters

arg	the event that called the method.

3.6.3.7 bool MyFrameListener::mousePressed (const OIS::MouseEvent & *arg,* OIS::MouseButtonID *id*)

Overwritten method that will be called every time a mouse button is pressed.

Parameters

arg	the event that called the method.
id	the id of the pressed button.

3.6.3.8 bool MyFrameListener::mouseReleased (const OIS::MouseEvent & *arg,* OIS::MouseButtonID *id*)

Overwritten method that will be called every time a mouse button is released.

Parameters

arg	the event that called the method.
id	the id of the pressed button.

3.6.3.9 virtual void MyFrameListener::setDebugDrawer (CDebugDraw * *aD*) [inline], [virtual]

Setter of the Debug System.

Parameters

aD	the debug system that is wanted to be activated.

3.6.3.10 void MyFrameListener::windowClosed (Ogre::RenderWindow * *rw*)

Overwritten method that will be called every time the main rendered window is closed.

Parameters

rw	the rendendered window that was closed.

3.6.3.11 void MyFrameListener::windowResized (Ogre::RenderWindow * *rw*)

Overwritten method that will be called every time the main rendered window is resized.

Parameters

rw	the rendendered window that was resized

The documentation for this class was generated from the following files:

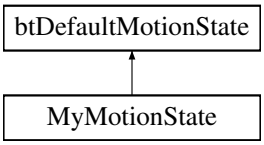
- MyFrameListener.h
- MyFrameListener.cpp

3.7 MyMotionState Class Reference

Class that will update automatically every object in the world in every step of the physics simulation. This is done by encapsulating the physical and the visual object into an object that will receive the information of the transform in each step.

`#include <Physics.h>`

Inheritance diagram for MyMotionState:



Public Member Functions

- MyMotionState (const btTransform &initialPos, Ogre::SceneNode *node, const btTransform &offset=bt↩ Transform::getIdentity())

  Constructor of the class.
- MyMotionState (Ogre::SceneNode *node)

Constructor of the class.

- ~MyMotionState ()

  Destructor of the class.
- void setNode (Ogre::SceneNode *node)

  setter of the Ogre3D node variable
- btTransform getWorldTransform () const

  getter of the actual Transformation.
- void getWorldTransform (btTransform &worldTrans) const

  getter of the actual Transformation.
- void setWorldTransform (const btTransform &worldTrans)

  setter of the actual Transformation.

Protected Attributes

- Ogre::SceneNode * mVisibleObj
- btTransform mTransform
- btTransform **mCOM**

3.7.1 Detailed Description

Class that will update automatically every object in the world in every step of the physics simulation. This is done by encapsulating the physical and the visual object into an object that will receive the information of the transform in each step.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 MyMotionState::MyMotionState (const btTransform & *initialPos*, Ogre::SceneNode * *node*, const btTransform & *offset* = `btTransform::getIdentity()`)

Constructor of the class.

Parameters

initialPos	the initial position of the physical node.
node	the Ogre3D node that wants to be updated every frame.
offset	an offset between the Ogre3D node and the physical node

3.7.2.2 MyMotionState::MyMotionState (Ogre::SceneNode * *node*)

Constructor of the class.

Parameters

node	the Ogre3D node that wants to be updated every frame.

3.7.3 Member Function Documentation

3.7.3.1 btTransform MyMotionState::getWorldTransform ( ) const

getter of the actual Transformation.

Returns

the actual transform.

3.7.3.2 void MyMotionState::getWorldTransform (btTransform & *worldTrans*) const

getter of the actual Transformation.

Parameters

[OUT]	the actual transform.

3.7.4 Member Data Documentation

3.7.4.1 btTransform MyMotionState::mTransform [protected]

actual Transform of the object

3.7.4.2 Ogre::SceneNode* MyMotionState::mVisibleObj [protected]

the ogre3D node that is linked to the object

The documentation for this class was generated from the following files:

- Physics.h
- Physics.cpp

3.8 MySoftBody Class Reference

Class that will encapsulate SoftBodies with their associated Ogre3D nodes.

`#include <Physics.h>`

Public Member Functions

- MySoftBody (Ogre::SceneNode *node, btSoftBody *aSoftBody)

  Constructor of the class.
- virtual ~MySoftBody ()

  Destructor of the class.
- void setNode (Ogre::SceneNode *node)

  setter of the Ogre3D node variable
- int getBulletIndex (int idx)

  Function used to make some inner calculations (it's not supposed to be called by the user)
- void updateOgreMesh ()

  Method that have to be called whenever the mesh of the Ogre3D node is wanted to be updated with the SoftBody deformations.

Protected Attributes

- Ogre::SceneNode * mVisibleObj
- btSoftBody * mSoftBody

3.8.1 Detailed Description

Class that will encapsulate SoftBodies with their associated Ogre3D nodes.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 MySoftBody::MySoftBody (Ogre::SceneNode * *node,* btSoftBody * *aSoftBody*)

Constructor of the class.

Parameters

node	the Ogre3D node that wants to be updated every frame.
aSoftBody	the softbody that will be updated in the physical simulation.

3.8.3 Member Data Documentation

3.8.3.1 btSoftBody* MySoftBody::mSoftBody [protected]

The SoftBody.

3.8.3.2 Ogre::SceneNode* MySoftBody::mVisibleObj [protected]

The Ogre3D node.

The documentation for this class was generated from the following files:

- Physics.h
- Physics.cpp

3.9 OgreBulletUtils Class Reference

Class that will contain every useful function or method that will make communication between Ogre3D and Bullet↩ Physics much easier.

`#include <OgreBulletUtils.h>`

Static Public Member Functions

- static btQuaternion toBullet (const Ogre::Quaternion &q)

  Transform an Ogre3D quaternion into a BulletPhysics quaternion.
- static btVector3 toBullet (const Ogre::Vector3 &v)

  Transform an Ogre3D vector into a BulletPhysics vector.
- static Ogre::Quaternion toOgre (const btQuaternion &q)

  Transform an BulletPhysics quaternion into a Ogre3D quaternion.
- static Ogre::Vector3 toOgre (const btVector3 &v)

  Transform an BulletPhysics vector into a Ogre3D vector.

3.9.1 Detailed Description

Class that will contain every useful function or method that will make communication between Ogre3D and Bullet↩ Physics much easier.

3.9.2 Member Function Documentation

3.9.2.1 static btQuaternion OgreBulletUtils::toBullet (const Ogre::Quaternion & *q*) [inline], [static]

Transform an Ogre3D quaternion into a BulletPhysics quaternion.

Parameters

q	the Ogre3D quaternion to be transformated.

Returns

the BulletPhysics quaternion.

3.9.2.2 static btVector3 OgreBulletUtils::toBullet (const Ogre::Vector3 & *v*) [inline], [static]

Transform an Ogre3D vector into a BulletPhysics vector.

Parameters

v	the Ogre3D vector to be transformated.

Returns

the BulletPhysics vector.

3.9.2.3 static Ogre::Quaternion OgreBulletUtils::toOgre (const btQuaternion & *q*) [inline], [static]

Transform an BulletPhysics quaternion into a Ogre3D quaternion.

Parameters

q	the BulletPhysics quaternion to be transformated.

Returns

the Ogre3D quaternion.

3.9.2.4 static Ogre::Vector3 OgreBulletUtils::toOgre (const btVector3 & *v*) [inline], [static]

Transform an BulletPhysics vector into a Ogre3D vector.

Parameters

v	the BulletPhysics vector to be transformated.

Returns

the Ogre3D vector.

The documentation for this class was generated from the following file:

- OgreBulletUtils.h

3.10 Physics Class Reference

Class in charge of managing the pyshics world by adding new objects and updating them each frame.

`#include <Physics.h>`

Public Types

- enum collisionTypes {
  COL_NOTHING = 0, COL_WORLD = BIT(0), COL_PJ = BIT(1), COL_SOFT = BIT(2),
  COL_CAPSULE = BIT(3) }

  Enum type of collisions. This will make collision filtering possible.

Public Member Functions

- Physics ()

  Empty constructor of the class.
- virtual ~Physics ()

  Destructor of the class.
- virtual btRigidBody * addRigidBody (btTransform transform, btCollisionShape *shape, btScalar mass, Ogre::SceneNode *node=NULL)

  Add a new generic rigid body to the physics world.
- virtual btRigidBody * addStaticPlane (Ogre::SceneNode *node)

  Add a static plane to the world. This is done by adding a new plane collision shape.
- virtual btRigidBody * addCube (Ogre::SceneNode *node, float m=10.0f)

  Add a physic cube to the world. This is done by adding a new cube collision shape.
- virtual btRigidBody * addMeshFromEntity (Ogre::SceneNode *node, Ogre::Entity *ent)

  Add a complex mesh rigid body to the world. This is done by transforming the mess into a complex collision shape. Code of this method is copied and adapted from btOgre.
- virtual btSoftBody * addSoftFromEntity (Ogre::SceneNode *node, Ogre::Entity *ent)

  Add a soft body with a complex collision shape obtained by transforming the mesh.
- virtual btSoftBody * addTrampoline (Ogre::SceneNode *node, Ogre::Entity *ent, int ResX, int ResY)

  Deprecated. Add a soft body with the shape of cloak that will lie as a trampoline.
- virtual btSoftBody * addSoftSphere (Ogre::SceneNode *node, Ogre::Entity *ent)

  Add a soft body with a sphere collision shape and some pressure inside.
- virtual void addImpulsedCube (Ogre::SceneNode *node, Ogre::Quaternion camPos)

  Add an impulsed cube beeing launched from the camera position.
- virtual void addCollisionShape (btCollisionShape *colShape)

  Add a collision shape to the world.
- virtual btSoftRigidDynamicsWorld * getDynamicsWorld ()

  Returns the world.
- virtual btCollisionWorld * getCollisionWorld ()

  Returns the collision world.
- virtual btBroadphaseInterface * getBroadphase ()

  Returns the broadphase algorithm.
- virtual void updateSoftBodies ()

  update all the softbodies' meshes that are in the world

Public Attributes

- int everythingCollidesWith = COL_PJ | COL_CAPSULE | COL_WORLD | COL_SOFT
- int softCollidesWith = COL_WORLD | COL_PJ
- int pjCollidesWith = COL_WORLD | COL_SOFT
- int capsuleCollidesWith = COL_WORLD
- btDefaultCollisionConfiguration * mCollisionConfiguration
- btCollisionDispatcher * mDispatcher
- btBroadphaseInterface * mBroadphase
- btConstraintSolver * mSolver

- btSoftRigidDynamicsWorld * mWorld
- std::vector< btCollisionShape * > mCollisionShapes
- std::vector< MySoftBody * > mSoftBodies
- int numCollisionObjects

3.10.1 Detailed Description

Class in charge of managing the pyshics world by adding new objects and updating them each frame.

3.10.2 Member Enumeration Documentation

3.10.2.1 enum Physics::collisionTypes

Enum type of collisions. This will make collision filtering possible.

Enumerator

COL_NOTHING enum val of things that don't collide with anything

COL_WORLD enum val of things that collide with non-special things

COL_PJ enum val of things that collide with the RagDoll of the main char

COL_SOFT enum val of things that collide with the SoftBody

COL_CAPSULE enum val of things that collide with main capsule of the character

3.10.3 Constructor & Destructor Documentation

3.10.3.1 Physics::Physics ( )

Empty constructor of the class.

3.10.3.2 Physics::~Physics ( ) [virtual]

Destructor of the class.

3.10.4 Member Function Documentation

3.10.4.1 virtual void Physics::addCollisionShape (btCollisionShape * *colShape*) [inline], [virtual]

Add a collision shape to the world.

Parameters

colShape	the shape to be added.

Returns

the soft body added.

3.10.4.2 btRigidBody * Physics::addCube (Ogre::SceneNode * *node*, float *m* = 10.0f) [virtual]

Add a physic cube to the world. This is done by adding a new cube collision shape.

Parameters

node	the Ogre3D node that will be linked to the cube.
mass	the mass that the cube will have

Returns

the rigid body added.

3.10.4.3 void Physics::addImpulsedCube (Ogre::SceneNode * *node*, Ogre::Quaternion *camPos*) [virtual]

Add an impulsed cube beeing launched from the camera position.

Parameters

node	the Ogre3D node of the cube.
camPos	the actual camera position.

Returns

the soft body added.

3.10.4.4 btRigidBody * Physics::addMeshFromEntity (Ogre::SceneNode * *node*, Ogre::Entity * *ent*) [virtual]

Add a complex mesh rigid body to the world. This is done by transforming the mess into a complex collision shape. Code of this method is copied and adapted from btOgre.

Parameters

node	the Ogre3D node that will be linked to the mesh.
ent	the entity containing the mesh that it's wanted to be transformed into a rigid body.

Returns

the rigid body added.

3.10.4.5 btRigidBody * Physics::addRigidBody (btTransform *transform*, btCollisionShape * *shape*, btScalar *mass*, Ogre::SceneNode * *node* = NULL) [virtual]

Add a new generic rigid body to the physics world.

Parameters

transform	the initial transform of the rigid body.
shape	the shape that must be linked to the rigid body.
mass	the mass that the rigid body will have.
node	the Ogre3D node that will be linked to the rigid body.

3.10.4.6 btSoftBody * Physics::addSoftFromEntity (Ogre::SceneNode * *node*, Ogre::Entity * *ent*) [virtual]

Add a soft body with a complex collision shape obtained by transforming the mesh.

Code of this method is copied and adapted from btOgre. Discussion of this in the oficial forum: http://www.↩ bulletphysics.org/Bullet/phpBB3/viewtopic.php?f=9&t=3428

Parameters

node	the Ogre3D node that will be linked to the mesh.
ent	the entity containing the mesh that it's wanted to be transformed into a soft body.

Returns

the rigid body added.

3.10.4.7 btSoftBody * Physics::addSoftSphere (Ogre::SceneNode * *node*, Ogre::Entity * *ent*) [virtual]

Add a soft body with a sphere collision shape and some pressure inside.

Parameters

node	the Ogre3D node that will be linked to the mesh.
ent	the entity containing the mesh that it's wanted to be transformed into a soft sphere.

Returns

the soft body added.

3.10.4.8 btRigidBody * Physics::addStaticPlane (Ogre::SceneNode * *node*) [virtual]

Add a static plane to the world. This is done by adding a new plane collision shape.

Parameters

node	the Ogre3D node that will be linked to the plane.

Returns

the rigid body added.

3.10.4.9 btSoftBody * Physics::addTrampoline (Ogre::SceneNode * *node*, Ogre::Entity * *ent*, int *ResX*, int *ResY*) [virtual]

Deprecated. Add a soft body with the shape of cloak that will lie as a trampoline.

Parameters

node	the Ogre3D node that will be linked to the SoftBody.
ent	the entity containing the mesh that will be transformed into a cloak.
ResX	Resolution in the X-axis of the plane that will be transformed into a cloak.
ResY	Resolution in the Y-axis of the plane that will be transformed into a cloak.

Returns

the soft body added.

3.10.4.10 virtual btBroadphaseInterface* Physics::getBroadphase ( ) [inline], [virtual]

Returns the broadphase algorithm.

Returns

the broadphase algorithm.

3.10.4.11 virtual btCollisionWorld* Physics::getCollisionWorld ( ) [inline], [virtual]

Returns the collision world.

Returns

the collision world.

3.10.4.12 virtual btSoftRigidDynamicsWorld* Physics::getDynamicsWorld ( ) [inline], [virtual]

Returns the world.

Returns

the world.

3.10.5 Member Data Documentation

3.10.5.1 int Physics::capsuleCollidesWith = COL_WORLD

Mask for filtering collisions of the main capsule.

3.10.5.2 int Physics::everythingCollidesWith = COL_PJ | COL_CAPSULE | COL_WORLD | COL_SOFT

Mask for filtering collisions.

3.10.5.3 btBroadphaseInterface* Physics::mBroadphase

BroadPhase algorithm of the world.

3.10.5.4 btDefaultCollisionConfiguration* Physics::mCollisionConfiguration

Collision configuration of the world.

3.10.5.5 std::vector<btCollisionShape *> Physics::mCollisionShapes

Array to store all the collision shapes that are in the world.

3.10.5.6 btCollisionDispatcher* Physics::mDispatcher

Collision Dispatcher of the world.

3.10.5.7 std::vector<MySoftBody *> Physics::mSoftBodies

Array to store all the soft bodies that are in the world.

3.10.5.8 btConstraintSolver* Physics::mSolver

Collision solver of the world.

3.10.5.9 btSoftRigidDynamicsWorld* Physics::mWorld

The physics world.

3.10.5.10 int Physics::numCollisionObjects

Quantity of collision objects that are in the world.

3.10.5.11 int Physics::pjCollidesWith = COL_WORLD | COL_SOFT

Mask for filtering collisions of the RagDoll.

3.10.5.12 int Physics::softCollidesWith = COL_WORLD | COL_PJ

Mask for filtering collisions of SoftBodies.

The documentation for this class was generated from the following files:

- Physics.h
- Physics.cpp

3.11 Ragdoll Class Reference

This class will modulate the physic body of the character accurately, creating a bunch of capsules that will wrap his body.

`#include <Ragdoll.h>`

Public Types

- enum BonesID {
  BODYPART_HIPS =0, **BODYPART_SPINE**, **BODYPART_HEAD**, **BODYPART_LEFT_UPPER_LEG**,
  BODYPART_LEFT_LOWER_LEG, **BODYPART_LEFT_FOOT**, **BODYPART_RIGHT_UPPER_LEG**, **BO↩**
  DYPART_RIGHT_LOWER_LEG,
  BODYPART_RIGHT_FOOT, **BODYPART_LEFT_UPPER_ARM**, **BODYPART_LEFT_LOWER_ARM**, **B↩**
  ODYPART_LEFT_HAND,
  BODYPART_RIGHT_UPPER_ARM, **BODYPART_RIGHT_LOWER_ARM**, **BODYPART_RIGHT_HAND**,
  BODYPART_RIGHT_SHEATH,
  BODYPART_LEFT_SHEATH, **BODYPART_COUNT** }

  Enum type of the bones inside the 3D model.

Public Member Functions

- Ragdoll (Ogre::SceneNode *aNode, Ogre::Entity *anEnt, btSoftRigidDynamicsWorld *aWorld)

  Constructor of the class.
- virtual ~Ragdoll ()

  Destructor of the class.
- void initRagdoll ()

  Function that does everything to initiate the Ragdoll. Create bones, translate them, etc.
- btRigidBody * localCreateRigidBody (btScalar mass, const btTransform &startTransform, btCollisionShape *shape)

  Method of inner use to create rigid bodies.

- void copyModelStateToRagdoll ()

  This function will copy the state of the model to the RagDoll so animations can also move the rigid bodies.

Public Attributes

- Ogre::SceneNode * mNode
- Ogre::Entity * mEntity
- btSoftRigidDynamicsWorld * mWorld
- Ogre::SkeletonInstance * mSkeleton
- btCollisionShape * m_shapes [BODYPART_COUNT]
- btRigidBody * m_bodies [BODYPART_COUNT]
- Ogre::Bone * m_Bones [BODYPART_COUNT]
- Ogre::Quaternion m_BoneInitialOrientation [BODYPART_COUNT]
- Ogre::Vector3 m_BoneInitialPosition [BODYPART_COUNT]
- Ogre::Vector3 m_BodyInitialPosition [BODYPART_COUNT]
- Ogre::Quaternion m_BodyInitialOrientation [BODYPART_COUNT]
- Ogre::Vector3 m_RagdollOffset

3.11.1 Detailed Description

This class will modulate the physic body of the character accurately, creating a bunch of capsules that will wrap his body.

3.11.2 Constructor & Destructor Documentation

3.11.2.1 Ragdoll::Ragdoll (Ogre::SceneNode * *aNode*, Ogre::Entity * *anEnt*, btSoftRigidDynamicsWorld * *aWorld*)

Constructor of the class.

Parameters

aNode	the Ogre3D node of the main character.
anEnt	the entity of the main character.
aWorld	the physic world in which the character is going to be added.

3.11.3 Member Function Documentation

3.11.3.1 btRigidBody * Ragdoll::localCreateRigidBody (btScalar *mass*, const btTransform & *startTransform*, btCollisionShape * *shape*)

Method of inner use to create rigid bodies.

Parameters

mass	the mas of the rigid body.
startTransform	the initial transform of the body.
shape	the capsule shape of the body.

Returns

the created body.

3.11.4 Member Data Documentation

3.11.4.1 btRigidBody* Ragdoll::m_bodies[BODYPART_COUNT]

array of rigid bodies that will wrap the main character body

3.11.4.2 Ogre::Quaternion Ragdoll::m_BodyInitialOrientation[BODYPART_COUNT]

array containing the initial rotation of every rigid body

3.11.4.3 Ogre::Vector3 Ragdoll::m_BodyInitialPosition[BODYPART_COUNT]

array containing the initial position of every rigid body

3.11.4.4 Ogre::Quaternion Ragdoll::m_BoneInitialOrientation[BODYPART_COUNT]

array containing the initial rotation of every bone

3.11.4.5 Ogre::Vector3 Ragdoll::m_BoneInitialPosition[BODYPART_COUNT]

array containing the initial position of every bone

3.11.4.6 Ogre::Bone* Ragdoll::m_Bones[BODYPART_COUNT]

array of bones that will be used in the RagDoll creation

3.11.4.7 Ogre::Vector3 Ragdoll::m_RagdollOffset

Offset between the ragdoll and the renderized character

3.11.4.8 btCollisionShape* Ragdoll::m_shapes[BODYPART_COUNT]

array of shapes that will wrap the main character body

3.11.4.9 Ogre::Entity* Ragdoll::mEntity

the Ogre3D entity of the main character

3.11.4.10 Ogre::SceneNode* Ragdoll::mNode

the Ogre3D node of the main character

3.11.4.11 Ogre::SkeletonInstance* Ragdoll::mSkeleton

the skeleton obtained from the entity of the main character

3.11.4.12 btSoftRigidDynamicsWorld* Ragdoll::mWorld

the physics' world

The documentation for this class was generated from the following files:

- Ragdoll.h
- Ragdoll.cpp

Index