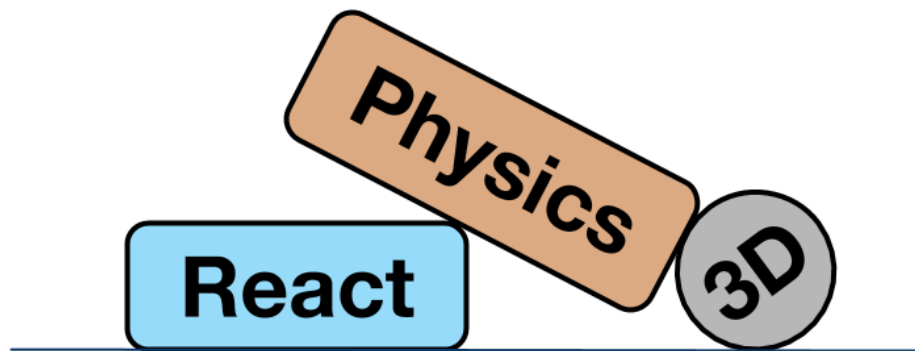# ReactPhysics3D library User Manual

Version: 0.7.1

Daniel Chappuis

May 19, 2020

Contents

1 Introduction

ReactPhysics3D is an open source C++ physics engine library that can be used in 3D simulations and games. The library is released under the ZLib license.

2 Features

The ReactPhysics3D library has the following features:

- Rigid body dynamics

- Discrete collision detection

- Collision shapes (Sphere, Box, Capsule, Convex Mesh, Static Concave Mesh, Height Field)

- Multiple collision shapes per body

- Broadphase collision detection (Dynamic AABB tree)

- Narrowphase collision detection (SAT/GJK)

- Collision response and friction (Sequential Impulses Solver)

- Joints (Ball and Socket, Hinge, Slider, Fixed)

- Collision filtering with categories

- Ray casting

- Sleeping technique for inactive bodies

- Multi-platform (Windows, Linux, Mac OS X)

- No external libraries (do not use STL containers)

- Documentation (user manual and Doxygen API)

- Testbed application with demos

- Integrated profiler

- Debugging renderer

- Logs

- Unit tests

3 License

The ReactPhysics3D library is released under the open-source ZLib license. For more information, read the "LICENSE" file.

4 Building and installing the library

First, you need to clone the code repository on your system with the following command:

```
git clone https://github.com/DanielChappuis/reactphysics3d.git
```

Note that the *git* versioning software needs to be installed on your system.

Then, you will need to build (compile) the library and install it on your system in order to use it in your project. The best way is to use CMake for that. CMake will generate the necessary files on your platform (Windows, OS X or Linux) to build the library.

CMake can be downloaded at http://www.cmake.org or using your package-management program (apt, yum, . . .) on Linux. If you have never used CMake before, you should read the page http://www.cmake.org/cmake/help/runningcmake.html as it contains a lot of useful information.

The remaining of this section will describe how to build and install the library with CMake.

4.1 Configure and generate the native tool files

Now we need to configure CMake to tell it what you want to build. Maybe you simply want to build the library in *debug* or *release* mode or maybe you also want to build the unit tests or the testbed application with demos. At the end of this step, CMake will generate the native build tool files on your platform that you will use to build the library. For instance, it can generate a Visual Studio solution on Windows, a XCode project on OS X or files for the `make` command on OS X or Linux.

4.1.1 Configure and generate with the command line (Linux and Mac OS X)

First, we will see how to configure CMake and generate the native build tool files using the CMake tool with the command line. First, you need to create a folder where you want to build the library. Then go into that folder and run the following `ccmake` command:

```
ccmake <path_to_library_source>
```

where `<path_to_library_source>` must be replaced by the path to the `reactphysics3d/` folder of the repository you have cloned. It is the folder that contains the `CMakeLists.txt` file of ReactPhysics3D. Running this command will launch the CMake command line interface. Hit the 'c' key to configure the project. There, you can also change some predefined options (see section 4.4 for more

details) and then, hit the 'c' key again to configure the build. Once you have set all the values as you like, you can hit the 'g' key to generate the native build tool files in the build directory that you have created before. Finally, you can exit the CMake interface.

4.1.2 Configure and generate using the CMake graphical interface (Linux, Mac OS X and Windows)

If your prefer, you can use the graphical user interface of CMake instead. To do this, run the `cmake-gui` program. First, the program will ask you for the source folder. You need to select the `reactphysics3d/` folder of the repository you have cloned. You will also have to select a folder where you want to build the library. Select any empty folder that is on your system. Then, you can click on *Configure*. CMake will ask you to choose an IDE that is on your system that will be used to compile the library. For instance, you can select Visual Studio, Qt Creator, XCode, ... Then, click on the *Finish* button. Then, you can change the compilation options. See section 4.4 to see what are the possible options. Once this is done, click on *Configure* again and finally on *Generate* as you can see in the following picture.
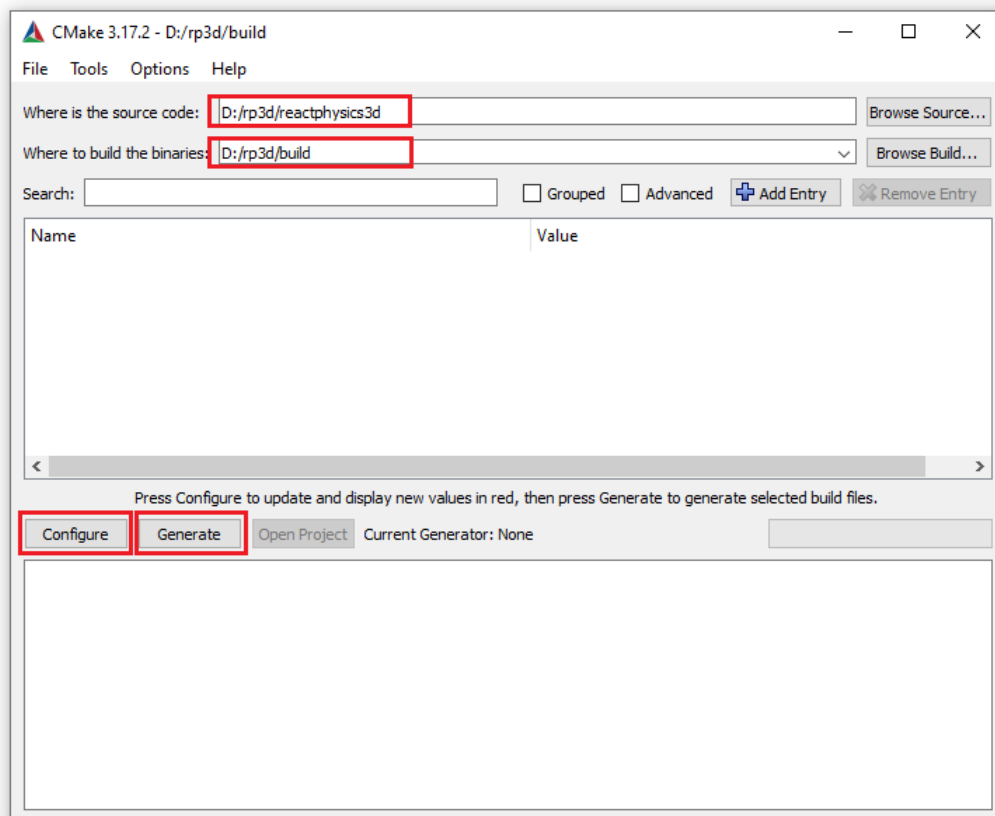
Now, if you go into the folder you have chosen to build the library, you should find the native build tool files that you will use to build the library on your platform.

4.2 Bulding the library

Now, that you have generated the native build tool files on your system, you will need to build (compile) the library.

4.2.1 Bulding the library using `make` on the command line (Linux, Mac OS X)

On Linux or Mac OS X, you can compile the library on the command line using the `make` command. Go into the directory where you have generated the native build tool files and run the following command:

```
make
```

The library will start compiling.

4.2.2 Bulding the library with Visual Studio (Windows)

If you have generated the native build tool files in the previous step on Windows, you should have obtained a Visual Studio solution of ReactPhysics3D. Now, you can open the Visual Studio solution (.sln file). Once Visual Studio is open, you first need to change the compilation mode to *Release* at the top instead of *Debug*. Then, right click on the *reactphysics* project in the Solution Explorer and click on *Build* in order to compile the library (see the following picture).
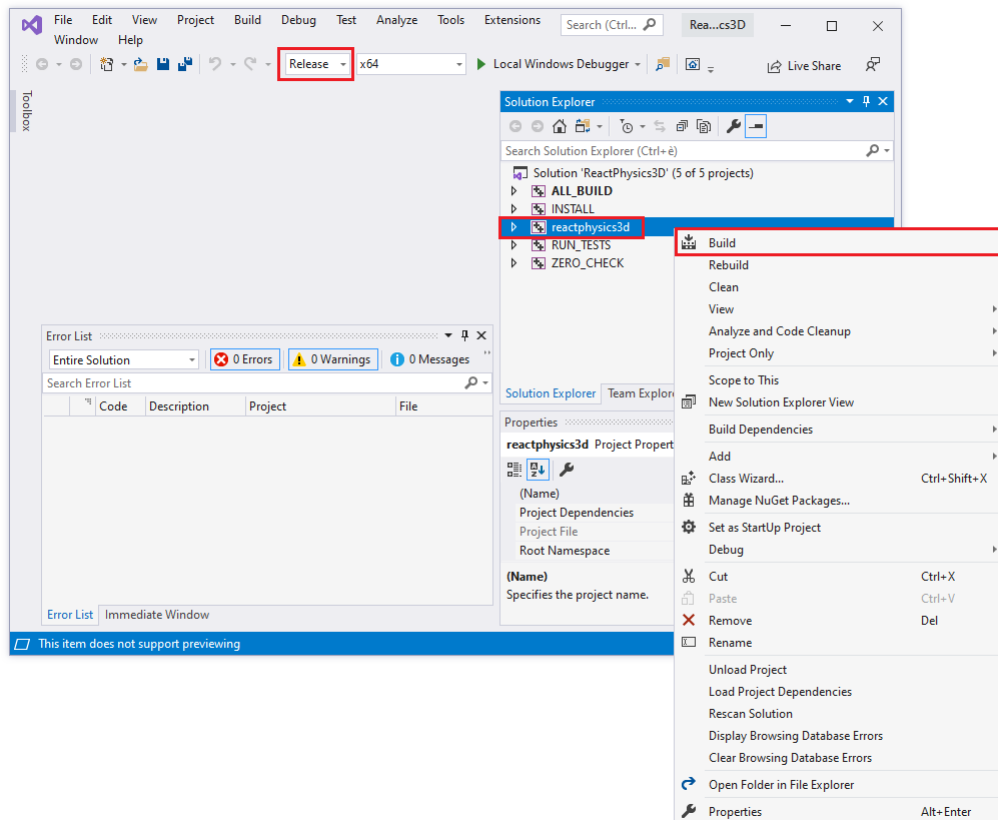
The library will start compiling.

4.3   Installing the library

Now that you have compiled the library, you can install it on your system in order to put the compiled library file, the include files and the exported CMake targets in a standard location on your system so that it can be easily imported into your project.

4.3.1   Installing the library using the `make` on the command line (Linux, Mac OS X)

On Linux or Mac OS X, you can use the `make` command to install the library. You simply need to run the following command:

```
sudo make install
```

The library is now installed on your system. For instance, On Linux Ubuntu, the library may have been installed in the `/usr/local/lib/` folder and the include files in the `/usr/local/include/` folder.

4.3.2   Installing the library on Windows with Visual Studio

In order to install the library on your system using Visual Studio, you need to open Visual Studio with administrator rights. This is needed in order to have the correct rights to write the files in the `C:\Program Files (x86)\` folder on your computer for instance. To do that, type *Visual Studio* in the Start Menu, when Visual Studio has been found, right click on it and click on *Run as administrator*. This will open Visual Studio with administrator rights.

Then, you need to open the Visual Studio solution (.sln file) of ReactPhysics3D that has been generated previously with CMake. To do that, click on *File* in the top menu of Visual Studio, then on *Open* and *Project/Solution....* Then, you need to select the ReactPhysics3D Visual Studio solution (.sln file) on your system. Once the solution is open, you first need to change the mode at the top to *Release* instead of *Debug*. Then, right click on the *INSTALL* project in the Solution Explorer menu and click on *Build* (see the following picture). This will install the ReactPhysics3D library in a standard location on your system like `C:\Program FilesÂă(x86)\ReactPhysics3D\` for instance.
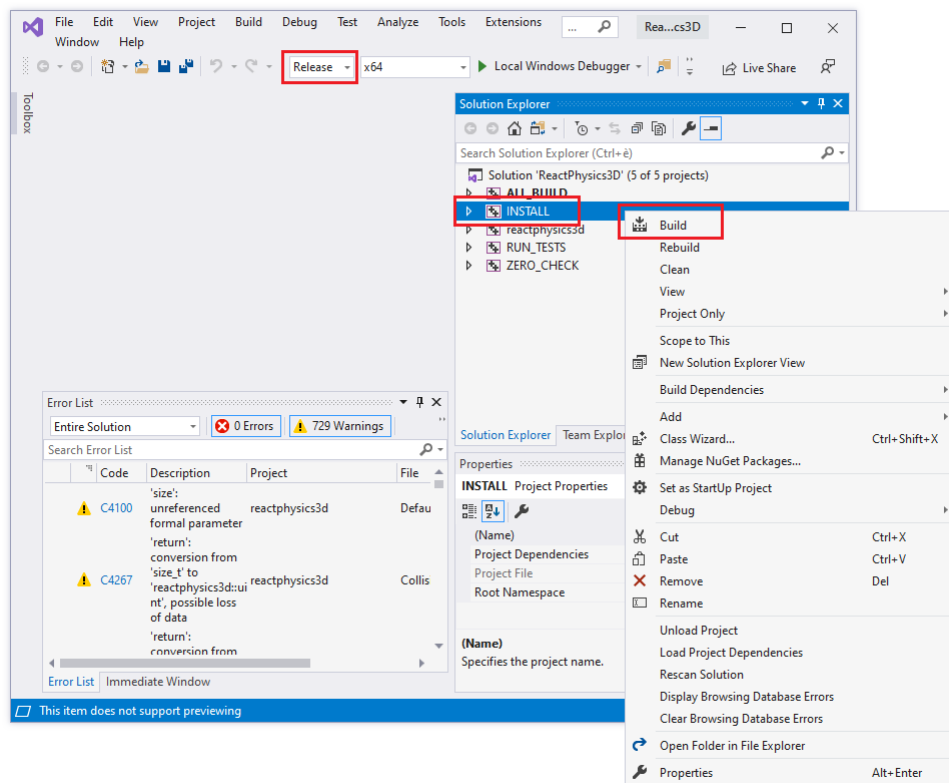
4.4 CMake Options

You can find bellow the different CMake options that you can set before building the library:

CMAKE_BUILD_TYPE If this variable is set to `Debug`, the library will be compiled in debugging mode. This mode should be used during development stage to know where things might crash. In debugging mode, the library might run a bit slow due to all the debugging information and asserts. However, if this variable is set to `Release`, no debugging information is stored and therefore, it will run much faster. This mode must be used when you compile the final release of your application.

RP3D_COMPILE_TESTBED If this variable is `ON`, the tesbed application of the library will be compiled. The testbed application uses OpenGL for rendering. Take a look at the section 14 for more information about the testbed application.

RP3D_COMPILE_TESTS If this variable is `ON`, the unit tests of the library will be compiled. You will then be able to launch the tests to make sure that they are running fine on your system.

RP3D_PROFILING_ENABLED If this variable is `ON`, the integrated profiler will collect data during the execution of the application. This might be useful to see which part of the ReactPhysics3D library takes time during its execution. This variable must be set to `OFF` when you compile the final release of your application. You can find more information about the profiler in section 17.

RP3D_DOUBLE_PRECISION_ENABLED If this variable is `ON`, the library will be compiled with double floating point precision. Otherwise, the library will be compiled with single precision.

5 Using ReactPhysics3D in your application

If you have built and installed the ReactPhysics3D on your system with CMake as explained in the section 4, it is easy to import the library in your project. You probably already have a `CMakeLists.txt` file for your project. Therefore, to import the ReactPhysics3D library, you simply need to add the following line in the `CMakeLists.txt` file of your project.

```
find_package(ReactPhysics3D REQUIRED)
```

This will tell CMake to find the installed ReactPhysics3D library on your system and import the library file and headers so that you can link it to your project. Note that if you are working on Windows, you might need to use the following code in your `CMakeLists.txt` file before calling the previous function. This will help CMake to find the installed ReactPhysics3D library on Windows.

```
if(WIN32)
  list(APPEND CMAKE_PREFIX_PATH "C:\\Program Files (x86)\\
      ReactPhysics3D")
endif()
```

Then, you need to tell CMake that your project (executable) depends on ReactPhysics3D with the following line in your `CMakeLists.txt` file:

```
target_link_libraries(helloworld ReactPhysics3D::
    ReactPhysics3D)
```

The ReactPhyscis3D repository contains a folder with an *Hello World* project here. In this folder, you can find a `CMakeLists.txt` and a `Main.cpp` file that show how to import and use the ReactPhysics3D library in a simple project.

Here is the example `CMakeLists.txt` file of the *Hello World* project:

```
# Minimum cmake version required
cmake_minimum_required(VERSION 3.8)

# Help CMake to find the installed library on Windows
```

```
if(WIN32)
  list(APPEND CMAKE_PREFIX_PATH "C:\\Program Files (x86)\\
      ReactPhysics3D")
endif()

# Import the ReactPhysics3D library
find_package(ReactPhysics3D REQUIRED)

# Project
project(HelloWorld)

# Create the executable
add_executable(helloworld Main.cpp)

# Link with the ReactPhysics3D library
target_link_libraries(helloworld ReactPhysics3D::
   ReactPhysics3D)
```

Then in your C++ source file, you need to include the main ReactPhysics3D header file with the following line:

```
// Include the main ReactPhysics3D header file
#include <reactphysics3d/reactphysics3d.h>
```

Also note that all the classes of the library are available in the `reactphysics3d` namespace or its shorter alias `rp3d`. Therefore, you can use this namespace in your code with the following declaration:

```
// Use the ReactPhysics3D namespace
using namespace reactphysics3d;
```

Here is the `Main.cpp` file of the *Hello World* project:

```
// Libraries
#include <reactphysics3d/reactphysics3d.h>
#include <iostream>

// ReactPhysics3D namespace
using namespace reactphysics3d;

// Main function
int main(int argc, char** argv) {

    // First you need to create the PhysicsCommon object.
      This is a factory module
```

```cpp
        // that you can use to create physics world and other
            objects. It is also responsible
        // for logging and memory management
        PhysicsCommon physicsCommon;

        // Create a physics world
        PhysicsWorld* world = physicsCommon.createPhysicsWorld()
            ;

        // Create a rigid body in the world
        Vector3 position(0, 20, 0);
        Quaternion orientation = Quaternion::identity();
        Transform transform(position, orientation);
        RigidBody* body = world->createRigidBody(transform);

        const decimal timeStep = 1.0f / 60.0f;

        // Step the simulation a few steps
        for (int i=0; i < 20; i++) {

            world->update(timeStep);

            // Get the updated position of the body
            const Transform& transform = body->getTransform();
            const Vector3& position = transform.getPosition();

            // Display the position of the body
            std::cout << "Body Position: (" << position.x << ",
                " << position.y << ", " << position.z << ")" <<
                std::endl;
        }

        return 0;
}
```

6   The PhysicsCommon object

The first thing you need to do when you want to use ReactPhysics3D is to instanciate
the `PhysicsCommon` class. This main object will then be used as a factory to instan-
ciate one or multiple physics worlds and other objects. This class is also responsible
for the memory management of the library. All the memory allocations are centralized
into this `PhysicsCommon` object. This class also contains the logger for the different
events that can occur.

In order to use ReactPhysics3D, you have to create an instance of the `PhysicsCommon`
class:

```
// First you need to create the PhysicsCommon object.
PhysicsCommon physicsCommon;
```

Then, you can use this object to instanciate a physics world for instance:

```
// Create a physics world
PhysicsWorld* w = physicsCommon.createPhysicsWorld();
```

When you will need to add a body into your world, you will probably need to create a collider with a given type of collision shape. Again, you will need to use the `PhysicsCommon` object to instantiate a collision shape as in the following example:

```
// Instanciate a sphere collision shape
SphereShape* sphereShape = physicsCommon.createSphereShape(
    radius);
```

As you can see, the `PhysicsCommon` object is the first thing you will need to instantiate in order to use ReactPhycsi3D in your code.

7   Memory Management

The `PhysicsCommon` class is responsible for all the memory allocations that occur in ReactPhysics3D. The base memory allocations in ReactPhysics3D are done using the `std::malloc()` and `std::free()` methods. If you want to use your own behavior to allocate and free memory, you can pass a custom memory allocator to the constructor of the `PhysicsCommon` object. You simply need to create a class that inherits from the `MemoryAllocator` class of ReactPhysics3D and overrides the `allocate()` and `release()` methods.

Note that several methods of ReactPhysics3D will create an instance of an object and return a pointer so that you can use that object. This the case for the creation of a `PhysicsWorld` or a `RigidBody` as you can see in the following code:

```
// Create a physics world
PhysicsWorld* world = physicsCommon.createPhysicsWorld();

...

// Create a rigid body
RigidBody* body = world->createRigidBody(transform);
```

Note that because those objects have been instantiated by ReactPhysics3D and not by you, the library is responsible to delete those objects. Therefore, you must not call the C++ `delete` operator on those objects. There are methods that you can call to destroy those objects when you do not need them anymore to release memory but if you don't do it, the library will do it for yourself when the `PhysicsCommon` object is deleted. The following example shows how to destroy previously created `RigidBody` and `PhysicsWorld`:

```
// Destroy a rigid body
world->destroyRigidBody(body);

...

// Destroy a physics world
physicsCommon.destroyPhysicsWorld(world);
```

8   Physics World

Once you have created a `PhysicsCommon` object (see section 6), you will have to create a physics world. A physics world is a place where you can add the bodies that you want to simulate. It is possible to create multiple physics worlds but you will probably never need more than one.

There are two main ways to use ReactPhysics3D. The first one is to create bodies that you have to manually move in the physics world and test for collision between them whenever you want. To do this, you need to add some collision bodies (class `CollisionBody`) in it as described in section 9. In this case, the physics engine will not be responsible to animate your bodies. You will have to move the bodies manually by yourself because you are only interested in the collision between your bodies. A use case for this could be an application where the user has to place furniture in a room and you need to give the user some feedback when the 3D shapes of the furniture are intersecting and therefore in an invalid configuration. Here are some methods that use can use in this case:

testOverlap() This group of methods can be used to test whether the colliders of two bodies overlap or not. You can use this if you just want to know if bodies are colliding but your are not interested in the contact information.

testCollision() This group of methods will give you the collision information (contact points, normals, ...) for colliding bodies.

testPointInside() This method will tell you if a 3D point is inside a given `CollisionBody`, `RigidBody` or `Collider`.

The second way to use the library is to create bodies and let ReactPhysics3D animate their motions automatically using the laws of physics. This is done by creating rigid bodies (class `RigidBody`) in your physics world and by updating the simulation by calling the `PhysicsWorld::update()` method each frame. The rigid bodies will move according to the forces, collision between bodies and joint constraints of the physics world. A typical use case is a 3D real-time game for instance.

8.1 Creating the Physics World

In order to create a physics world, you need to call the `createPhysicsWorld()` method of the main `PhysicsCommon` object:

```
// Create the physics world
PhysicsWorld* world = physicsCommon.createPhysicsWorld();
```

This method will return a pointer to the physics world that has been created.

8.1.1 World settings

When you create a physics world as in the previous example, it will have some default settings. If you want to customize some settings, you need to create a `PhysicsWorld::WorldSettings` object and give it in parameter when you create your physics world as in the following example:

```
// Create the world settings
PhysicsWorld::WorldSettings settings;
settings.defaultVelocitySolverNbIterations = 20;
settings.isSleepingEnabled = false;
settings.gravity = Vector3(0, -9.81, 0);

// Create the physics world with your settings
PhysicsWorld* world = physicsCommon.createPhysicsWorld(
    settings);
```

The settings are copied into the world at its creation. Therefore, changing the values of your `PhysicsWorld::WorldSettings` instance after the world constructor call will not have any effects. However, some methods are available to change settings after the world creation. You can take a look at the API documentation to see what world settings can be changed in the `PhysicsWorld` class.

8.2 Customizing the Physics World

8.2.1 Solver parameters

ReactPhysics3D uses an iterative solver to simulate the contacts and joints. For contacts, there is a unique velocity solver and for joints there is a velocity and a position solver. By default, the number of iterations of the velocity solver is 10 and the number of iterations for the position solver is 5. It is possible to change the number of iterations for both solvers.

To do this, you need to use the following two methods:

```
// Change the number of iterations of the velocity solver
world->setNbIterationsVelocitySolver(15);

// Change the number of iterations of the position solver
world->setNbIterationsPositionSolver(8);
```

Increasing the number of iterations of the solvers will make the simulation more precise but also more expensive to compute. Therefore, you should change those values only if necessary.

8.2.2 Sleeping

The purpose of the sleeping technique is to deactivate resting bodies so that they are not simulated anymore. This is used to save computation time because simulating many bodies is costly. A sleeping body (or group of sleeping bodies) is awaken as soon as another body collides with it or a joint in which it is involed is enabled. The sleeping technique is enabled by default. You can disable it using the following method:

```
// Disable the sleeping technique
world->enableSleeping(false);
```

Note that it is not recommended to disable the sleeping technique because the simulation might become slower. It is also possible to deactivate the sleeping technique on a per body basis. See section 10.5 for more information.

A body is put to sleep when its linear and angular velocity stay under a given velocity threshold for a certain amount of time (one second by default). It is possible to change the linear and angular velocity thresholds using the two methods `PhysicsWorld::setSleepLinearVelocity()` and `PhysicsWorld::setSleepAngularVelocity()`. Note that the velocities must be specified in meters per second. You can also change the amount of time (in seconds)

the velocity of a body needs to stay under the threshold to be considered sleeping. To do this, use the `PhysicsWorld::setTimeBeforeSleep()` method.

8.3   Updating the Physics World

When the `PhysicsWorld` is used to animate the bodies through time according to the laws of physics, the world has to be updated each time you want to simulate a step forward in time (for instance each frame in a real-time simulation).

To update the physics world, you need to use the `PhysicsWorld::update()` method. This method will perform collision detection and update the position and orientation of the bodies according to the forces, joints constraints and collision contacts. Once you have updated the world, you will be able to retrieve the new position and orientation of your bodies in order to render the next frame. This method requires a *timeStep* parameter. This is the amount of time you want to advance the physics simulation (in seconds).

The smaller the time step you pick, the more precise the simulation will be. For a real-time application, you probably want to use a time step of at most $\frac{1}{60}$ seconds to have at least a 60 Hz framerate. Most of the time, physics engines prefer to work with a constant time step. It means that you should always call the `PhysicsWorld::update()` method with the same time step parameter. You do not want to use the time between two frames as your time step because it will not be constant.

You can use the following technique. First, you need to choose a constant time step. Let say the time step is $\frac{1}{60}$ seconds. Then, at each frame, you compute the time difference between the current frame and the previous one and you accumulate this difference in a variable called *accumulator*. The accumulator is initialized to zero at the beginning of your application and is updated at each frame. The idea is to divide the time in the accumulator in several constant time steps. For instance, if your accumulator contains 0.145 seconds, it means that we can take 8 physics steps of $\frac{1}{60}$ seconds during the current frame. Note that 0.012 seconds will remain in the accumulator and will probably be used in the next frame. As you can see, with this technique, multiple physics steps can be taken at each frame. It is important to understand that each call to the `PhysicsWorld::update()` method is done using a constant time step that is not varying with the framerate of the application.

Here is what the code looks like at each frame:

```
// Constant physics time step
const float timeStep = 1.0f / 60.0f;

// Get the current system time
long double currentFrameTime = getCurrentSystemTime();
```

```
// Compute the time difference between the two frames
long double deltaTime  = currentFrameTime -
   previousFrameTime;

// Update the previous time
previousFrameTime = currentFrameTime;

// Add the time difference in the accumulator
accumulator += mDeltaTime;

// While there is enough accumulated time to take
// one or several physics steps
while (accumulator >= timeStep) {

   // Update the Dynamics world with a constant time step
   world->update(timeStep);

   // Decrease the accumulated time
   accumulator -= timeStep;
}
```

If you want to know more about physics simulation time interpolation, you can read the nice article from Glenn Fiedler at https://gafferongames.com/post/fix_your_timestep/.

8.4   Destroying the Physics World

When you don't need the physics world anymore, you can destroy it to release some memory. When the `CollisionWorld` is destroyed, all the bodies that have been added into it and that have not been destroyed already will be destroyed.

```
// Destroy the physics world
physicsCommon.destroyPhysicsWorld(world);
```

Note that the pointer to the physics world and all the objects that have been created inside it (bodies, colliders, ...) will become invalid after this call.

9   Collision Body

Once you have created a physics world, it is time to add bodies into it. At this point, you will have to choose between the two types of bodies of ReactPhysics3D. You can create either a `CollisionBody` or a `RigidBody`. If you only want to add bodies into your world, move them manually and test if they collide without simulating their motion using the laws of physics, you can create bodies of type `CollisionBody`. However, if

you want your bodies to be physically animated, you will need to use bodies of type `RigidBody`. The `RigidBody` class is described in section 10. The current section explains how to use a `CollisionBody`.

9.1 Creating a Collision Body

When you create a collision body, you need to specify its initial transform. This transform describes the initial position and orientation of the body in the world. You need to create an instance of the `Transform` class with a vector describing the initial position and a quaternion for the initial orientation of the body.

You need to call the `PhysicsWorld::createCollisionBody()` method to create a collision body in the physics world you have previously created. This method will return a pointer to the instance of the `CollisionBody` object that has been created internally. You will then be able to use that pointer to get or set values of the body.

The following code describes how to create a collision body in the physics world.

```
// Initial position and orientation of the collision body
Vector3 position(0.0, 3.0, 0.0);
Quaternion orientation = Quaternion::identity();
Transform transform(position, orientation);

// Create a collision body in the world
CollisionBody* body;
body = world->createCollisionBody(transform);
```

In order to test collision between your body and other bodies in the world, you probably want to add some colliders to your body. Take a look at section 11 to learn what is a collider and how to use it.

9.2 Moving a Collision Body

A collision body has to be moved manually in the world because it is not simulated by the physics engine. In order to move it, you need to use the `CollisionBody::setTransform()` method to set a new position and new orientation to the body.

```
// New position and orientation of the collision body
Vector3 position(10.0, 3.0, 0.0);
Quaternion orientation = Quaternion::identity();
Transform newTransform(position, orientation);

// Move the collision body
```

```
body->setTransform(newTransform);
```

9.3 Destroying a Collision Body

In order to destroy a collision body, you need to use the `PhysicsWorld::destroyCollisionBody()` method. You need to use the pointer to the body you want to destroy in argument. Note that after calling this method, the pointer will not be valid anymore and therefore, you should not use it.

The following code shows how to destroy a collision body:

```
// Here, world is an instance of the PhysicsWorld class
// and body is a CollisionBody* pointer

// Destroy the collision body and remove it from the world
world->destroyCollisionBody(body);
```

10 Rigid Body

Once the physics world has been created, you can add rigid bodies into it. A rigid body is an object that will be simulated using the laws of physics. It has a mass, a position, an orientation and one or several colliders. It will react to forces and collisions. The physics world will compute contacts between the bodies and will update their positions and orientations accordingly at each time step. You can also create joints between bodies to link them in different configurations. In ReactPhysics3D, the `RigidBody` class (which inherits from the `CollisionBody` class) is used to describe a rigid body.

10.1 Creating a Rigid Body

In order to create a rigid body, you need to specify its transform. The transform describes the initial position and orientation of the body in the world. You need to create an instance of the `Transform` class with a vector describing the initial position and a quaternion for the initial orientation of the body.

You have to call the `PhysicsWorld::createRigidBody()` method to create a rigid body in the world. This method will return a pointer to the instance of the `RigidBody` object that has been created internally. You will then be able to use that pointer to get or set values to the body.

You can see in the following code how to create a rigid body in your world:

```
// Initial position and orientation of the rigid body
Vector3 position(0.0, 3.0, 0.0);
```

```
Quaternion orientation = rp3d::Quaternion::identity();
Transform transform(position, orientation);

// Create a rigid body in the world
RigidBody* body;
body = world->createRigidBody(transform);
```

Once your rigid body has been created in the world, you will probably want to add one or colliders to it as described in section 11.

10.2   Type of a Rigid Body (static, kinematic or dynamic)

There are three types of bodies: *static*, *kinematic* and *dynamic*. A *static* body has infinite mass, zero velocity but its position can be changed manually. Moreover, a static body does not collide with other static or kinematic bodies. On the other side, a *kinematic* body has infinite mass, its velocity can be changed manually and its position is computed by the physics engine. A kinematic body does not collide with other static or kinematic bodies. Finally, A *dynamic* body has non-zero mass, non-zero velocity determined by forces and its position is determined by the physics engine. Moreover, a dynamic body can collide with other dynamic, static or kinematic bodies.

For instance, you can use a *static* body for the floor, a *kinematic* body for a moving platform and a *dynamic* body for a rock that could fall on the floor.

When you create a new body in the world, it is of dynamic type by default. You can change the type of the body using the `RigidBody::setType()` method as follows:

```
// Change the type of the body to kinematic
body->setType(BodyType::KINEMATIC);
```

10.3   Gravity

By default, all the rigid bodies with react to the gravity force of the world. If you do not want the gravity to be applied to a given body, you can disable it using the `RigidBody::enableGravity()` method as in the following example:

```
// Disable gravity for this body
rigidBody->enableGravity(false);
```

10.4   Velocity Damping

Damping is the effect of reducing the velocity of the rigid body during the simulation to simulate effects like air friction for instance. By default, no damping

is applied. However, you can choose to damp the linear or/and the angular velocity of a rigid body. For instance, without angular damping a pendulum will never come to rest. You need to use the `RigidBody::setLinearDamping()` and `RigidBody::setAngularDamping()` methods to change the damping values. The damping value has to be positive and a value of zero means no damping at all.

10.5 Sleeping

As described in section 8.2.2, the sleeping technique is used to disable the simulation of resting bodies. By default, the bodies are allowed to sleep when they come to rest. However, if you do not want a given body to be put to sleep, you can use the `Body::setIsAllowedToSleep()` method as in the next example:

```
// This rigid body cannot sleep
rigidBody->setIsAllowedToSleep(false);
```

10.6 Applying Force or Torque to a Rigid Body

During the simulation, you can apply a force or a torque to a given rigid body. This force can be applied to the center of mass of the rigid body by using the `RigidBody::applyForceToCenterOfMass()` method. You need to specify the force vector (in Newton) as a parameter. If the force is applied to the center of mass, no torque will be created and only the linear motion of the body will be affected.

```
// Force vector (in Newton)
Vector3 force(2.0, 0.0, 0.0);

// Apply a force to the center of the body
rigidBody->applyForceToCenterOfMass(force);
```

You can also apply a force to any given point in world-space using the `RigidBody::applyForceAtWorldPosition()` method or in local-space with the `RigidBody::applyForceAtLocalPosition()` method. You need to specify the force vector (in Newton) and the point where to apply the given force. Note that if the point is not the center of mass of the body, applying a force will generate some torque and therefore, the angular motion of the body will be affected as well.

```
// Force vector (in Newton)
Vector3 force(2.0, 0.0, 0.0);

// Point where the force is applied
Vector3 point(4.0, 5.0, 6.0);

// Apply a force to the body
```

```
rigidBody ->applyForceAtLocalPosition(force, point);
```

It is also possible to apply a torque to a given body using the `RigidBody::applyTorque()` method. You simply need to specify the torque vector (in Newton · meter) as in the following example:

```
// Torque vector
Vector3 torque(0.0, 3.0, 0.0);

// Apply a torque to the body
rigidBody ->applyTorque(torque);
```

Note that when you call the previous methods, the specified force/torque will be added to the total force/torque applied to the rigid body and that at the end of each call to the `PhysicsWorld::update()`, the total force/torque of all the rigid bodies will be reset to zero. Therefore, you need to call the previous methods during several frames if you want the force/torque to be applied during a certain amount of time.

10.7   Updating a Rigid Body

When you call the `PhysicsWorld::update()` method, the bodies positions and orientations are updated to satisfy the contacts and joints constraint between the bodies. After calling this method, you can retrieve the updated position and orientation of each body to render it. To do that, you simply need to use the `RigidBody::getTransform()` method to get the updated transform. This transform represents the current local-to-world-space transform of the body.

As described in section 8.3, at the end of a frame, there might still be some remaining time in the time accumulator. Therefore, you should not use the updated transform directly for rendering but you need to perform some interpolation between the updated transform and the one from the previous frame to get a smooth real-time simulation. First, you need to compute the interpolation factor as folows:

```
// Compute the time interpolation factor
decimal factor = accumulator / timeStep;
```

Then, you can use the `Transform::interpolateTransforms()` method to compute the linearly interpolated transform:

```
// Compute the interpolated transform of the rigid body
Transform interpolatedTransform = Transform::
    interpolateTransforms(prevTransform, currTransform,
    factor);
```

The following code is the one from section 8.3 for the physics simulation loop but with the update of a given rigid body.

```cpp
// Constant physics time step
const float timeStep = 1.0 / 60.0;

// Get the current system time
long double currentFrameTime = getCurrentSystemTime();

// Compute the time difference between the two frames
long double deltaTime  = currentFrameTime -
    previousFrameTime;

// Update the previous time
previousFrameTime = currentFrameTime;

// Add the time difference in the accumulator
accumulator += mDeltaTime;

// While there is enough accumulated time to take
// one or several physics steps
while (accumulator >= timeStep) {

    // Update the physics world with a constant time step
    physicsWorld->update(timeStep);

    // Decrease the accumulated time
    accumulator -= timeStep;
}

// Compute the time interpolation factor
decimal factor = accumulator / timeStep;

// Get the updated transform of the body
Transform currTransform = body->getTransform();

// Compute the interpolated transform of the rigid body
Transform interpolatedTransform = Transform::
    interpolateTransforms(prevTransform, currTransform,
    factor);

// Now you can render your body using the interpolated
    transform here

// Update the previous transform
prevTransform = currTranform;
```

If you need the array with the corresponding 4×4 OpenGL transformation matrix for rendering, you can use the `Transform::getOpenGLMatrix()` method as in the following code:

```
// Get the OpenGL matrix array of the transform
float matrix[16];
transform.getOpenGLMatrix(matrix);
```

A nice article to read about this time interpolation is the one from Glenn Fiedler at https://gafferongames.com/post/fix_your_timestep/.

10.8   Destroying a Rigid Body

It is really simple to destroy a rigid body when you don't need it. You simply need to use the `PhysicsWorld::destroyRigidBody()` method. You need to use the pointer to the body you want to destroy as a parameter. Note that after calling that method, the pointer will not be valid anymore and therefore, you should not use it. When you destroy a rigid body that was part of a joint, that joint will be automatically destroyed as well.

Here is how to destroy a rigid body:

```
// Here, world is an instance of the PhysicsWorld class
// and body is a RigidBody* pointer

// Destroy the rigid body
world->destroyRigidBody(body);
```

11   Collider

A body can only collide against an other body if it has at least one collider. A collider (class `Collider`) describes the collision shape of the body. A body can have multiple colliders attached to it. When adding a collider to a body, you need to specify its collision shape (box, sphere, capsule, . . .) and its transform relative to the origin of the body.

Before adding a collider to a body, you need to create a collision shape. A collision shape can be instantiated by calling a method of the main `PhysicsCommon` object. The following example shows how instantiate a collision shape (a sphere shape) from the `PhysicsCommon` object and use it to create a new collider to a rigid body.

```
// Instantiate a sphere collision shape
float radius = 3.0f;
```

```
SphereShape* sphereShape = physicsCommon.
    createSphereCollisionShape(radius);

// Relative transform of the collider relative to the body
    origin
Transform transform = Transform::identity();

// Add the collider to the rigid body
Collider* collider;
collider = body->addCollider(&shape, transform);
```

Note that a given collision shape instance can be shared between multiple colliders. The next section the different types of collision shapes that are available in ReactPhysics3D.

11.1 Collision Shapes

As we have just seen, the collision shape is used to describe the shape of a collider. They are many types of collision shapes that you can use. They all inherit from the CollisionShape class.

11.1.1 Box Shape
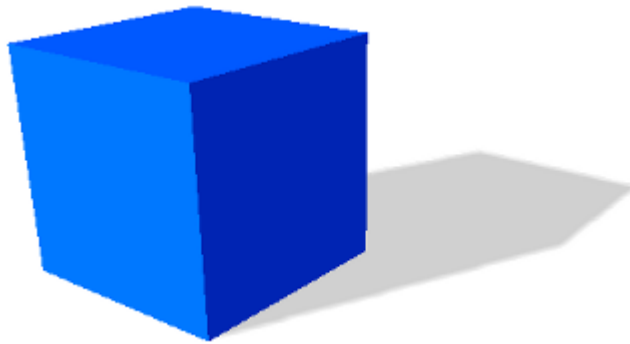
The BoxShape class describes a box collision shape. The box is aligned with the shape local X, Y and Z axis. In order to create a box shape, you only need to specify the three half extents dimensions of the box in the three X, Y and Z directions.

For instance, if you want to create a box shape with dimensions of 4 meters, 6 meters and 10 meters along the X, Y and Z axis respectively, you need to use the following code:

```
// Half extents of the box in the x, y and z directions
const rp3d::Vector3 halfExtents(2.0, 3.0, 5.0);

// Create the box shape
```

```
const rp3d::BoxShape boxShape(halfExtents);
```

11.1.2   Sphere Shape

The `SphereShape` class describes a sphere collision shape centered at the origin of the shape local space. You only need to specify the radius of the sphere to create it.

For instance, if you want to create a sphere shape with a radius of 2 meters, you need to use the following code:

```
// Create the sphere shape with a radius of 2m
const rp3d::SphereShape sphereShape(2.0);
```

11.1.3   Capsule Shape

The `CapsuleShape` class describes a capsule collision shape around the local Y axis and centered at the origin of the shape local-space. It is the convex hull of two spheres. It can also be seen as an elongated sphere. In order to create it, you only need to specify the radius of the two spheres and the height of the capsule (distance

between the centers of the two spheres).

For instance, if you want to create a capsule shape with a radius of 1 meter and the height of 2 meters, you need to use the following code:

```
// Create the capsule shape
const rp3d::CapsuleShape capsuleShape(1.0, 2.0);
```

11.1.4 Convex Mesh Shape
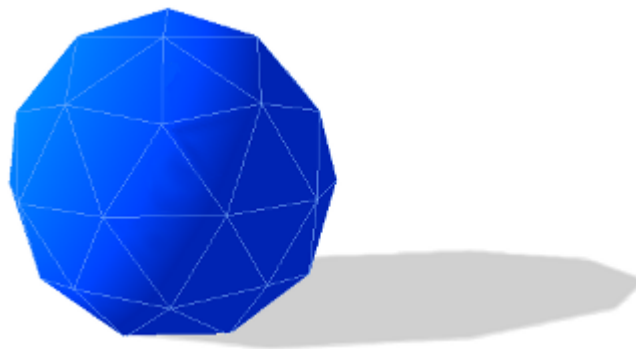
The `ConvexMeshShape` class can be used to describe the shape of a convex mesh. In order to create a convex mesh shape, you first need to create an array of `PolygonFace` to describe each face of your mesh. You also need to have an array with the vertices coordinates and an array with the vertex indices of each face of you mesh. Then, you have to create a `PolygonVertexArray` with your vertices coordinates and indices array. You also need to specify your array of `PolygonFace`. Then, you have to create a `PolyhedronMesh` with your `PolygonVertexArray`. Once this is done, you can create the `ConvexMeshShape` by passing your `PolyhedronMesh` in paramater.

The following example shows how to create a convex mesh shape. In this example, we create a cube as a convex mesh shape. Of course, this is only for the example. If you really need a cube collision shape, you should use the `BoxShape` instead.

```
// Array with the vertices coordinates of the convex mesh
float vertices[24];
vertices[0] = -3; vertices[1] = -3; vertices[2] = 3;
vertices[3] = 3; vertices[4] = -3; vertices[5] = 3;
vertices[6] = 3; vertices[7] = -3; vertices[8] = -3;
vertices[9] = -3; vertices[10] = -3; vertices[11] = -3;
vertices[12] = -3; vertices[13] = 3; vertices[14] = 3;
vertices[15] = 3; vertices[16] = 3; vertices[17] = 3;
vertices[18] = 3; vertices[19] = 3; vertices[20] = -3;
```

```cpp
vertices[21] = -3; vertices[22] = 3; vertices[23] = -3;

// Array with the vertices indices for each face of the mesh
int indices[24];
indices[0]=0; indices[1]=3; indices[2]=2; indices[3]=1;
indices[4]=4; indices[5]=5; indices[6]=6; indices[7]=7;
indices[8]=0; indices[9]=1; indices[10]=5; indices[11]=4;
indices[12]=1; indices[13]=2; indices[14]=6; indices[15]=5;
indices[16]=2; indices[17]=3; indices[18]=7; indices[19]=6;
indices[20]=0; indices[21]=4; indices[22]=7; indices[23]=3;

// Description of the six faces of the convex mesh
polygonFaces = new rp3d::PolygonVertexArray::PolygonFace[6];
rp3d::PolygonVertexArray::PolygonFace* face = polygonFaces;
for (int f = 0; f < 6; f++) {

    // First vertex of the face in the indices array
    face->indexBase = f * 4;

    // Number of vertices in the face
    face->nbVertices = 4;

    face++;
}

// Create the polygon vertex array
polygonVertexArray = new rp3d::PolygonVertexArray(8,
    vertices, 3 x sizeof(float),
indices, sizeof(int), 6, polygonFaces,
rp3d::PolygonVertexArray::VertexDataType::VERTEX_FLOAT_TYPE,
rp3d::PolygonVertexArray::IndexDataType::INDEX_INTEGER_TYPE)
    ;

// Create the polyhedron mesh
polyhedronMesh = new rp3d::PolyhedronMesh(polygonVertexArray
    );

// Create the convex mesh collision shape
convexMeshShape = new rp3d::ConvexMeshShape(polyhedronMesh);
```

Note that the vertex coordinates and indices array are not copied and therefore, you need to make sure that they exist until the collision shape exists. This is also true for the all the `PolygonFace`, the `PolygonVertexArray` and the `PolyhedronMesh`.

You need to make sure that the mesh you provide is indeed convex. Secondly, you should provide the simplest possible convex mesh. It means that you need to avoid coplanar faces in your convex mesh shape. Coplanar faces have to be merged together. Remember that convex meshes are not limited to triangular faces, you can

create faces with more than three vertices.

When you specify the vertices for each face of your convex mesh, be careful with their order. The vertices of a face must be specified in counter clockwise order as seen from the outside of your convex mesh.

You also need to make sure that the origin of your mesh is inside the convex mesh. A mesh with an origin outside the convex mesh is not currently supported by the library.

You can also specify a scaling factor in the constructor when you create a `Convex MeshShape`. All the vertices of your mesh will be scaled from the origin by this factor when used in the collision shape.

Note that collision detection with a `ConvexMeshShape` is more expensive than with a `SphereShape` or a `CapsuleShape`.

11.1.5 Concave Mesh Shape
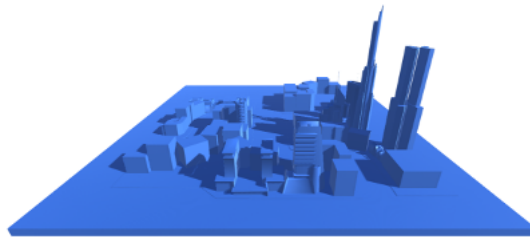
The `ConcaveMeshShape` class can be used for a static concave triangular mesh. It can be used to describe an environment for instance. Note that it cannot be used with a dynamic body that is allowed to move. Moreover, make sure to use a `ConcaveMeshShape` only when you are not able to use a convex shape and also try to limit the number of triangles of that mesh because collision detection with `ConcaveMeshShape` is quite expensive compared to convex shapes.

In order to create a concave mesh shape, you need to supply a pointer to a `TriangleMesh`. A `TriangleMesh` class describes a mesh made of triangles. It may contain several parts (submeshes). Each part is a set of triangles represented by a `TriangleVertexArray` object. A `TriangleVertexArray` represents a continuous array of vertices and indexes for a triangular mesh. When you create a `TriangleVertex Array`, no data is copied into the array. It only stores a pointer to the data. The idea is to allow the user to share vertices data between the physics engine and the rendering part. Therefore, make sure that the data pointed by a `TriangleVertexArray` remains valid during the whole `TriangleVertexArray` life.

The following code shows how to create a `TriangleVertexArray`:

```
const int nbVertices = 8;
const int nbTriangles = 12;
float vertices[3 * nbVertices] = ...;
int indices[3 * nbTriangles] = ...;
rp3d::TriangleVertexArray* triangleArray =
new rp3d::TriangleVertexArray(nbVertices, vertices, 3 *
    sizeof(float), nbTriangles,
indices, 3 * sizeof(int),
rp3d::TriangleVertexArray::VertexDataType::VERTEX_FLOAT_TYPE
    ,
rp3d::TriangleVertexArray::IndexDataType::INDEX_INTEGER_TYPE
    );
```

Now that we have a `TriangleVertexArray`, we need to create a `TriangleMesh` and add the `TriangleVertexArray` into it as a subpart. Once this is done, we can create the actual `ConcaveMeshShape`.

```
rp3d::TriangleMesh triangleMesh;

// Add the triangle vertex array to the triangle mesh
triangleMesh.addSubpart(triangleArray);

// Create the concave mesh shape
ConcaveMeshShape* concaveMesh = new rp3d::ConcaveMeshShape(&
    triangleMesh);
```

Note that the `TriangleMesh` object also needs to exist during the whole life of the collision shape because its data is not copied into the collision shape.

When you specify the vertices for each triangle face of your mesh, be careful with the order of the vertices. They must be specified in counter clockwise order as seen from the outside of your mesh.

You can also specify a scaling factor in the constructor when you create a `Concave MeshShape`. All the vertices of your mesh will be scaled from the origin by this factor when used in the collision shape.

In the previous example, the vertex normals that are needed for collision detection are automatically computed. However, if you want to specify your own vertex normals, you can do it by using another constructor for the `TriangleVertexArray`.
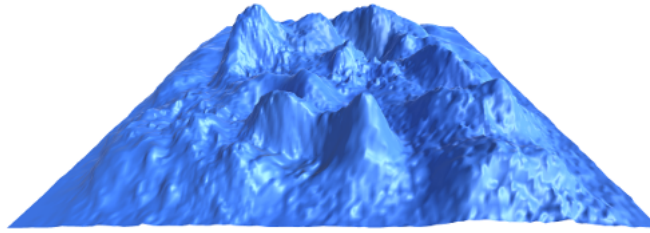
11.1.6 Heightfield Shape

The `HeightFieldShape` is a collision shape that can be used to represent a static terrain for instance. You can define a heightfield with a two dimensional grid that has a given height value at each point.

In order to create a `HeightFieldShape`, you need to have an array with all the height values of your field. You can have height values of type int, float or double. You need to give the number of rows and columns of your two dimensional grid. Note that the height values in your array must be organized such that the value at row `indexRow` and column `indexColumn` is located at the following position in the array:

```
heighFieldValues[indexRow * nbColumns + indexColumn]
```

Moreover, you need to provide the minimum and maximum height values of your height field.

Here is an example that shows how to create a `HeightFieldShape`:

```cpp
const int nbRows = 40;
const int nbColumns = 50;
float minHeight = 100;
float maxHeight = 500;

// Height values
float heightValues[nbRows * nbColumns] = ...;

// Create the heightfield collision shape
rp3d::HeightFieldShape = new rp3d::HeightFieldShape(
    nbColumns, nbRows, minHeight,
maxHeight, heightValues, rp3d::HeightFieldShape::
    HEIGHT_FLOAT_TYPE);
```

Note that the array of height values are not copied into the `HeightFieldShape`. Therefore, you need to make sure they exist during the lifetime of the `HeightField Shape` and you must not forget to release their memory when you destroy the collision shape or at the end of your application.

You can also specify a scaling factor in the constructor when you create a `Height FieldShape`. All the vertices of your mesh will be scaled from the origin by this factor when used in the collision shape.

When creating a `HeightFieldShape`, the origin of the shape will be at the center of its bounding volume. Therefore, if you create a `HeightFieldShape` with a minimum height of 100 and a maximum height of 500, the maximum coordinates of the shape on the Y axis will be 200 and the minimum coordinates will be -200.

11.2   Mass, Center of Mass, InertiaÂ Tensor

Note that the inertia tensor is a 3×3 matrix describing how the mass is distributed inside the rigid body which will be used to calculate its rotation. The inertia tensor depends on the mass and the shape of the body.

11.3   Collision filtering

By default all the collision shapes of all your bodies are able to collide with each other in the world. However, sometimes we want a body to collide only with a given group of bodies and not with other bodies. This is called collision filtering. The idea is to group the collision shapes of bodies into categories. Then we can specify for each collision shape against which categories it will be able to collide.

ReactPhysics3D uses bits mask to represent categories. The first thing to do is to assign a category to the collision shapes of your body. To do this, you need to call the `ProxyShape::setCollisionCategoryBits()` method on the corresponding proxy shape as in the following example. Here we consider that we have four bodies where each one has a single collision shape.

```
// Enumeration for categories
enum Category {
    CATEGORY1 = 0x0001,
    CATEGORY2 = 0x0002,
    CATEGORY3 = 0x0004
};


// Set the collision category for each proxy shape of
// each of the four bodies
proxyShapeBody1 ->setCollisionCategoryBits ( CATEGORY1 );
proxyShapeBody2 ->setCollisionCategoryBits ( CATEGORY2 );
proxyShapeBody3 ->setCollisionCategoryBits ( CATEGORY3 );
proxyShapeBody4 ->setCollisionCategoryBits ( CATEGORY3 );
```

As you can see, the collision shape of body 1 will be part of the category 1, the collision shape of body 2 will be part of the category 2 and the collision shapes of

bodies 3 and 4 will be part of the category 3.

Now, for each collision shape, we need to specify with which categories the shape is allowed to collide with. To do this, you need to use the `ProxyShape::setCollideWithMaskBits()` method of the proxy shape. Note that you can specify one or more categories using the bitwise OR operator. The following example shows how to specify with which categories the shapes can collide.

```
// For each shape, we specify with which categories it
// is allowed to collide
proxyShapeBody1->setCollideWithMaskBits(CATEGORY3);
proxyShapeBody2->setCollideWithMaskBits(CATEGORY1 |
    CATEGORY3);
proxyShapeBody3->setCollideWithMaskBits(CATEGORY2);
proxyShapeBody4->setCollideWithMaskBits(CATEGORY2);
```

As you can see, we specify that the body 1 will be allowed to collide with bodies from the categorie 3. We also indicate that the body 2 will be allowed to collide with bodies from the category 1 and 3 (using the bitwise OR operator). Finally, we specify that bodies 3 and 4 will be allowed to collide against bodies of the category 2.

A collision shape is able to collide with another only if you have specify that the category mask of the first shape is part of the *collide with* mask of the second shape. It is also important to understand that this condition must be satisfied in both directions. For instance in the previous example, the body 1 (of category 1) says that it wants to collide against bodies of the category 3 (for instance against body 3). However, body 1 and body 3 will not be able to collide because the body 3 does not say that it wants to collide with bodies from category 1. Therefore, in the previous example, the body 2 is allowed to collide against bodies 3 and 4 but no other collision is allowed.

In the same way, you can perform this filtering for ray casting (described in section 13). For instance, you can perform a ray cast test against a given subset of categories of collision shapes only.

11.4 Material of a Rigid Body

The material of a rigid body is used to describe the physical properties it is made of. This is represented by the `Material` class. Each body that you create will have a default material. You can get the material of the rigid body using the `RigidBody::getMaterial()` method. Then, you will be able to change some properties.

For instance, you can change the bounciness of the rigid body. The bounciness is a value between 0 and 1. The value 1 is used for a very bouncy object and the value 0 means that the body will not be bouncy at all. To change the bounciness of the material, you can use the `Material::setBounciness()` method.

You are also able to change the friction coefficient of the body. This value needs to be between 0 and 1. If the value is 0, no friction will be applied when the body is in contact with another body. However, if the value is 1, the friction force will be high. You can change the friction coefficient of the material with the `Material::setFrictionCoefficient()` method.

You can use the material to add rolling resistance to a rigid body. Rolling resistance can be used to stop a rolling object on a flat surface for instance. You should use this only with SphereShape or CapsuleShape collision shapes. By default, rolling resistance is zero but you can set a positive value using the `Material::setRollingResistance()` method to increase resistance.

Here is how to get the material of a rigid body and how to modify some of its properties:

```
// Get the current material of the body
rp3d::Material& material = rigidBody->getMaterial();

// Change the bounciness of the body
material.setBounciness(rp3d::decimal(0.4));

// Change the friction coefficient of the body
material.setFrictionCoefficient(rp3d::decimal(0.2));
```

12 Joints

Joints are used to constrain the motion of the rigid bodies between each other. A single joint represents a constraint between two rigid bodies. When the motion of the first body of the joint is known, the relative motion of the second body has at most six degrees of freedom (three for the translation and three for the rotation). The different joints can reduce the number of degrees of freedom between two rigid bodies.

Some joints have limits to control the range of motion and some joints have motors to automatically move the bodies of the joint at a given speed.

12.1 Ball and Socket Joint

The `BallAndSocketJoint` class describes a ball and socket joint between two bodies. In a ball and socket joint, the two bodies cannot translate with respect to each other. However, they can rotate freely around a common anchor point. This joint has three degrees of freedom and can be used to simulate a chain of bodies for instance.

In order to create a ball and socket joint, you first need to create an instance of the `BallAndSocketJointInfo` class with the necessary information. You need to provide the pointers to the two rigid bodies and also the coordinates of the anchor point (in world-space). At the joint creation, the world-space anchor point will be converted into the local-space of the two rigid bodies and then, the joint will make sure that the two local-space anchor points match in world-space. Therefore, the two bodies need to be in a correct position at the joint creation.

Here is the code to create the `BallAndSocketJointInfo` object:

```
// Anchor point in world-space
const rp3d::Vector3 anchorPoint(2.0, 4.0, 0.0);

// Create the joint info object
rp3d::BallAndSocketJointInfo jointInfo(body1, body2,
    anchorPoint);
```

Now, it is time to create the actual joint in the dynamics world using the `Dynamics World::createJoint()` method. Note that this method will also return a pointer to the `BallAndSocketJoint` object that has been created internally. You will then be able to use that pointer to change properties of the joint and also to destroy it at the end.

Here is how to create the joint in the world:

```
// Create the joint in the dynamics world
rp3d::BallAndSocketJoint* joint;
joint = dynamic_cast<rp3d::BallAndSocketJoint*>(world.
    createJoint(jointInfo));
```

12.2 Hinge Joint

The `HingeJoint` class describes a hinge joint (or revolute joint) between two rigid bodies. The hinge joint only allows rotation around an anchor point and around a single axis (the hinge axis). This joint can be used to simulate doors or pendulums for instance.

In order to create a hinge joint, you first need to create a `HingeJointInfo` object with the necessary information. You need to provide the pointers to the two rigid bodies, the coordinates of the anchor point (in world-space) and also the hinge rotation axis (in world-space). The two bodies need to be in a correct position when the joint is created.

Here is the code to create the `HingeJointInfo` object:

```
// Anchor point in world-space
const rp3d::Vector3 anchorPoint(2.0, 4.0, 0.0);

// Hinge rotation axis in world-space
const rp3d::Vector3 axis(0.0, 0.0, 1.0);

// Create the joint info object
rp3d::HingeJointInfo jointInfo(body1, body2, anchorPoint,
    axis);
```

Now, it is time to create the actual joint in the dynamics world using the `Dynamics World::createJoint()` method. Note that this method will also return a pointer to the `HingeJoint` object that has been created internally. You will then be able to use that pointer to change properties of the joint and also to destroy it at the end.

Here is how to create the joint in the world:

```
// Create the hinge joint in the dynamics world
rp3d::HingeJoint* joint;
joint = dynamic_cast<rp3d::HingeJoint*>(world.createJoint(
    jointInfo));
```

12.2.1 Limits

With the hinge joint, you can constrain the motion range using limits. The limits of the hinge joint are the minimum and maximum angle of rotation allowed with respect to the initial angle between the bodies when the joint is created. The limits are disabled by default. If you want to use the limits, you first need to enable them by setting the `isLimitEnabled` variable of the `HingeJointInfo` object to *true* before you create the joint. You also have to specify the minimum and maximum limit angles (in radians) using the `minAngleLimit` and `maxAngleLimit` variables of the joint info object. Note that the minimum limit angle must be in the range $[-2\pi; 0]$ and the maximum limit angle must be in the range $[0; 2\pi]$.

For instance, here is the way to use the limits for a hinge joint when the joint is created:

```
// Create the joint info object
rp3d::HingeJointInfo jointInfo(body1, body2, anchorPoint,
    axis);

// Enable the limits of the joint
jointInfo.isLimitEnabled = true;
```

```
// Minimum limit angle
jointInfo.minAngleLimit = -PI / 2.0;

// Maximum limit angle
jointInfo.maxAngleLimit = PI / 2.0;

// Create the hinge joint in the dynamics world
rp3d::HingeJoint* joint;
joint = dynamic_cast<rp3d::HingeJoint*>(world.createJoint(
    jointInfo));
```

It is also possible to use the `HingeJoint::enableLimit()`, `HingeJoint::setMinAngleLimit()` and `HingeJoint::setMaxAngleLimit()` methods to specify the limits of the joint after its creation. See the API documentation for more information.

12.2.2 Motor

A motor is also available for the hinge joint. It can be used to rotate the bodies around the hinge axis at a given angular speed and such that the torque applied to rotate the bodies does not exceed a maximum allowed torque. The motor is disabled by default. If you want to use it, you first have to activate it using the `isMotorEnabled` boolean variable of the `HingeJointInfo` object before you create the joint. Then, you need to specify the angular motor speed (in radians/seconds) using the `motorSpeed` variable and also the maximum allowed torque (in Newton · meters) with the `maxMotorTorque` variable.

For instance, here is how to enable the motor of the hinge joint when the joint is created:

```
// Create the joint info object
rp3d::HingeJointInfo jointInfo(body1, body2, anchorPoint,
    axis);

// Enable the motor of the joint
jointInfo.isMotorEnabled = true;

// Motor angular speed
jointInfo.motorSpeed = PI / 4.0;

// Maximum allowed torque
jointInfo.maxMotorTorque = 10.0;

// Create the hinge joint in the dynamics world
rp3d::HingeJoint* joint;
```

```
joint = dynamic_cast<rp3d::HingeJoint*>(world.createJoint(
    jointInfo));
```

It is also possible to use the HingeJoint::enableMotor(),
HingeJoint::setMotorSpeed() and HingeJoint::setMaxMotorTorque() meth-
ods to enable the motor of the joint after its creation. See the API documentation for
more information.

12.3   Slider Joint

The SliderJoint class describes a slider joint (or prismatic joint) that only allows
relative translation along a single direction. It has a single degree of freedom and
allows no relative rotation. In order to create a slider joint, you first need to specify
the anchor point (in world-space) and the slider axis direction (in world-space). The
constructor of the SliderJointInfo object needs two pointers to the bodies of the
joint, the anchor point and the axis direction. Note that the two bodies have to be in a
correct initial position when the joint is created.

You can see in the following code how to specify the information to create a slider
joint:

```
// Anchor point in world-space
const rp3d::Vector3 anchorPoint = rp3d::decimal(0.5) * (
    body2Position + body1Position);

// Slider axis in world-space
const rp3d::Vector3 axis = (body2Position - body1Position);

// Create the joint info object
rp3d::SliderJointInfo jointInfo(body1, body2, anchorPoint,
    axis);
```

Now, it is possible to create the actual joint in the dynamics world using the
DynamicsWorld::createJoint() method. Note that this method will also return a
pointer to the SliderJoint object that has been created internally. You will then be
able to use that pointer to change properties of the joint and also to destroy it at the
end.

Here is how to create the joint in the world:

```
// Create the slider joint in the dynamics world
rp3d::SliderJoint* joint;
joint = dynamic_cast<rp3d::SliderJoint*>(world.createJoint(
    jointInfo));
```

12.3.1 Limits

It is also possible to control the range of the slider joint motion using limits. The limits are disabled by default. In order to use the limits when the joint is created, you first need to activate them using the `isLimitEnabled` variable of the `SliderJointInfo` class. Then, you need to specify the minimum and maximum translation limits (in meters) using the `minTranslationLimit` and `maxTranslationLimit` variables. Note that the initial position of the two bodies when the joint is created corresponds to a translation of zero. Therefore, the minimum limit must be smaller or equal to zero and the maximum limit must be larger or equal to zero.

You can see in the following example how to set the limits when the slider joint is created:

```cpp
// Create the joint info object
rp3d::SliderJointInfo jointInfo(body1, body2, anchorPoint,
    axis);

// Enable the limits of the joint
jointInfo.isLimitEnabled = true;

// Minimum translation limit
jointInfo.minTranslationLimit = -1.7;

// Maximum translation limit
jointInfo.maxTranslationLimit = 1.7;

// Create the hinge joint in the dynamics world
rp3d::SliderJoint* joint;
joint = dynamic_cast<rp3d::SliderJoint*>(world.createJoint(
    jointInfo));
```

You can also use the `SliderJoint::enableLimit()`, `SliderJoint::-setMinTranslationLimit()` and `SliderJoint::setMaxTranslationLimit()` methods to enable the limits of the joint after its creation. See the API documentation for more information.

12.3.2 Motor

The slider joint also has a motor. You can use it to translate the bodies along the slider axis at a given linear speed and such that the force applied to move the bodies does not exceed a maximum allowed force. The motor is disabled by default. If you want to use it when the joint is created, you first have to activate it using the `isMotorEnabled` boolean variable of the `SliderJointInfo` object before you create the joint. Then, you need to specify the linear motor speed (in meters/seconds) using the `motorSpeed` variable and also the maximum allowed force (in Newtons) with the `maxMotorForce`

variable.

For instance, here is how to enable the motor of the slider joint when the joint is created:

```
// Create the joint info object
rp3d::SliderJointInfo jointInfo(body1, body2, anchorPoint,
    axis);

// Enable the motor of the joint
jointInfo.isMotorEnabled = true;

// Motor linear speed
jointInfo.motorSpeed = 2.0;

// Maximum allowed force
jointInfo.maxMotorForce = 10.0;

// Create the slider joint in the dynamics world
rp3d::SliderJoint* joint;
joint = dynamic_cast<rp3d::SliderJoint*>(world.createJoint(
    jointInfo));
```

It is also possible to use the `SliderJoint::enableMotor()`, `SliderJoint::setMotorSpeed()` and `SliderJoint::setMaxMotorForce()` methods to enable the motor of the joint after its creation. See the API documentation for more information.

12.4 Fixed Joint

The `FixedJoint` class describes a fixed joint between two bodies. In a fixed joint, there is no degree of freedom, the bodies are not allowed to translate or rotate with respect to each other. In order to create a fixed joint, you simply need to specify an anchor point (in world-space) to create the `FixedJointInfo` object.

For instance, here is how to create the joint info object for a fixed joint:

```
// Anchor point in world-space
rp3d::Vector3 anchorPoint(2.0, 3.0, 4.0);

// Create the joint info object
rp3d::FixedJointInfo jointInfo1(body1, body2, anchorPoint);
```

Now, it is possible to create the actual joint in the dynamics world using the `DynamicsWorld::createJoint()` method. Note that this method will also return a

pointer to the `FixedJoint` object that has been created internally. You will then be able to use that pointer to change properties of the joint and also to destroy it at the end.

Here is how to create the joint in the world:

```cpp
// Create the fixed joint in the dynamics world
rp3d::FixedJoint* joint;
joint = dynamic_cast<rp3d::FixedJoint*>(world.createJoint(
    jointInfo));
```

12.5 Collision between the bodies of a Joint

By default, the two bodies involved in a joint are able to collide with each other. However, it is possible to disable the collision between the two bodies that are part of the joint. To do it, you simply need to set the variable `isCollisionEnabled` of the joint info object to *false* when you create the joint.

For instance, when you create a `HingeJointInfo` object in order to construct a hinge joint, you can disable the collision between the two bodies of the joint as in the following example:

```cpp
// Create the joint info object
rp3d::HingeJointInfo jointInfo(body1, body2, anchorPoint,
    axis);

// Disable the collision between the bodies
jointInfo.isCollisionEnabled = false;

// Create the joint in the dynamics world
rp3d::HingeJoint* joint;
joint = dynamic_cast<rp3d::HingeJoint*>(world.createJoint(
    jointInfo));
```

12.6 Destroying a Joint

In order to destroy a joint, you simply need to call the `DynamicsWorld::destroyJoint()` method using the pointer to a previously created joint object as argument as shown in the following code:

```cpp
// rp3d::BallAndSocketJoint* joint is a previously created
    joint

// Destroy the joint
world.destroyJoint(joint);
```

It is important that you destroy all the joints that you have created at the end of the simulation. Also note that destroying a rigid body involved in a joint will automatically destroy that joint.

13 Ray casting

You can use ReactPhysics3D to test intersection between a ray and the bodies of the world you have created. Ray casting can be performed against multiple bodies, a single body or any proxy shape of a given body. Note that ray casting only works from the outside of the bodies. If the origin of a ray is inside a collision shape, no hit will be reported.

The first thing you need to do is to create a ray using the `Ray` class of React-Physics3D. As you can see in the following example, this is very easy. You simply need to specify the point where the ray starts and the point where the ray ends (in world-space coordinates).

```
// Start and end points of the ray
rp3d::Vector3 startPoint(0.0, 5.0, 1.0);
rp3d::Vector3 endPoint(0.0, 5.0, 30);

// Create the ray
rp3d::Ray ray(startPoint, endPoint);
```

Any ray casting test that will be described in the following sections returns a `RaycastInfo` object in case of intersection with the ray. This structure contains the following attributes:

worldPoint Hit point in world-space coordinates

worldNormal Surface normal of the proxy shape at the hit point in world-space coordinates

hitFraction Fraction distance of the hit point between *startPoint* and *endPoint* of the ray. The hit point p is such that $p = startPoint + hitFraction \cdot (endPoint - startPoint)$

body Pointer to the collision body or rigid body that has been hit by the ray

proxyShape Pointer to the proxy shape that has been hit by the ray

Note that you can also use collision filtering with ray casting in order to only test ray intersection with specific proxy shapes. Collision filtering is described in section 11.3.

13.1 Ray casting against multiple bodies

This section describes how to get all the proxy shapes of all bodies in the world that are intersected by a given ray.

13.1.1 The RaycastCallback class

First, you have to implement your own class that inherits from the `RaycastCallback` class. Then, you need to override the `RaycastCallback::notifyRaycastHit()` method in your own class. An instance of your class have to be provided as a parameter of the raycast method and the `notifyRaycastHit()` method will be called for each proxy shape that is hit by the ray. You will receive, as a parameter of this method, a `RaycastInfo` object that will contain the information about the raycast hit (hit point, hit surface normal, hit body, hit proxy shape, . . .).

In your `notifyRaycastHit()` method, you need to return a fraction value that will specify the continuation of the ray cast after a hit. The return value is the next maxFraction value to use. If you return a fraction of 0.0, it means that the raycast should terminate. If you return a fraction of 1.0, it indicates that the ray is not clipped and the ray cast should continue as if no hit occurred. If you return the fraction in the parameter (hitFraction value in the `RaycastInfo` object), the current ray will be clipped to this fraction in the next queries. If you return -1.0, it will ignore this ProxyShape and continue the ray cast. Note that no assumption can be done about the order of the calls of the `notifyRaycastHit()` method.

Here is an example about creating your own raycast callback class that inherits from the `RaycastCallback` class and how to override the `notifyRaycastHit()` method:

```
// Class WorldRaycastCallback
class MyCallbackClass : public rp3d::RaycastCallback {

public:

    virtual decimal notifyRaycastHit(const RaycastInfo& info)
        {

        // Display the world hit point coordinates
        std::cout << "Hit point : " <<
                     info.worldPoint.x <<
                     info.worldPoint.y <<
                     info.worldPoint.z <<
                     std::endl;

        // Return a fraction of 1.0 to gather all hits
        return decimal(1.0);
    }
};
```

13.1.2 Raycast query in the world

Now that you have your own raycast callback class, you can use the `raycast()` method to perform a ray casting test on a collision world or a dynamics world.

The first parameter of this method is a reference to the `Ray` object representing the ray you need to test intersection with. The second parameter is a pointer to the object of your raycast callback object. You can specify an optional third parameter which is the bit mask for collision filtering. It can be used to raycast only against selected categories of proxy shapes as described in section 11.3.

```
// Create the ray
rp3d::Vector3 startPoint(1 , 2, 10);
rp3d::Vector3 endPoint(1, 2, -20);
Ray ray(startPoint, endPoint);

// Create an instance of your callback class
MyCallbackClass callbackObject;

// Raycast test
world->raycast(ray, &callbackObject);
```

13.2 Ray casting against a single body

You can also perform ray casting against a single specific collision body or rigid body of the world. To do this, you need to use the `CollisionBody::raycast()` method. This method takes two parameters. The first one is a reference to the `Ray` object and the second one is a reference to the `RaycastInfo` object that will contain hit information if the ray hits the body. This method returns true if the ray hits the body. The `RaycastInfo` object will only be valid if the returned value is *true* (a hit occured).

The following example shows how test ray intersection with a body:

```
// Create the ray
rp3d::Vector3 startPoint(1 , 2, 10);
rp3d::Vector3 endPoint(1, 2, -20);
Ray ray(startPoint, endPoint);

// Create the raycast info object for the
// raycast result
RaycastInfo raycastInfo;
```

```
// Raycast test
bool isHit = body->raycast(ray, raycastInfo);
```

13.3   Ray casting against the proxy shape of a body

You can also perform ray casting against a single specific proxy shape of a collision body or rigid body of the world. To do this, you need to use the `ProxyShape::raycast()` method of the given proxy shape. This method takes two parameters. The first one is a reference to the `Ray` object and the second one is a reference to the `RaycastInfo` object that will contain hit information if the ray hits the body. This method returns true if the ray hits the body. The `RaycastInfo` object will only be valid if the returned value is *true* (a hit occured).

The following example shows how to test ray intersection with a given proxy shape:

```
// Create the ray
rp3d::Vector3 startPoint(1 , 2, 10);
rp3d::Vector3 endPoint(1, 2, -20);
Ray ray(startPoint, endPoint);

// Create the raycast info object for the
// raycast result
RaycastInfo raycastInfo;

// Test raycasting against a proxy shape
bool isHit = proxyShape->raycast(ray, raycastInfo);
```

14   Testbed application

The testbed application is a graphical interface where you can select and see some demo scenes using the ReactPhysics3D library.

Follow the instructions described in section 4 to compile the testbed application. Note that OpenGL is required to compile it.

The testbed application can be found in the `testbed/` folder of the ReactPhysics3D library. Do not hesitate to take a look at the code of the demo scenes to better understand how to use the library in your application.

The following subsections describe the demo scenes that can be found in the testbed application.
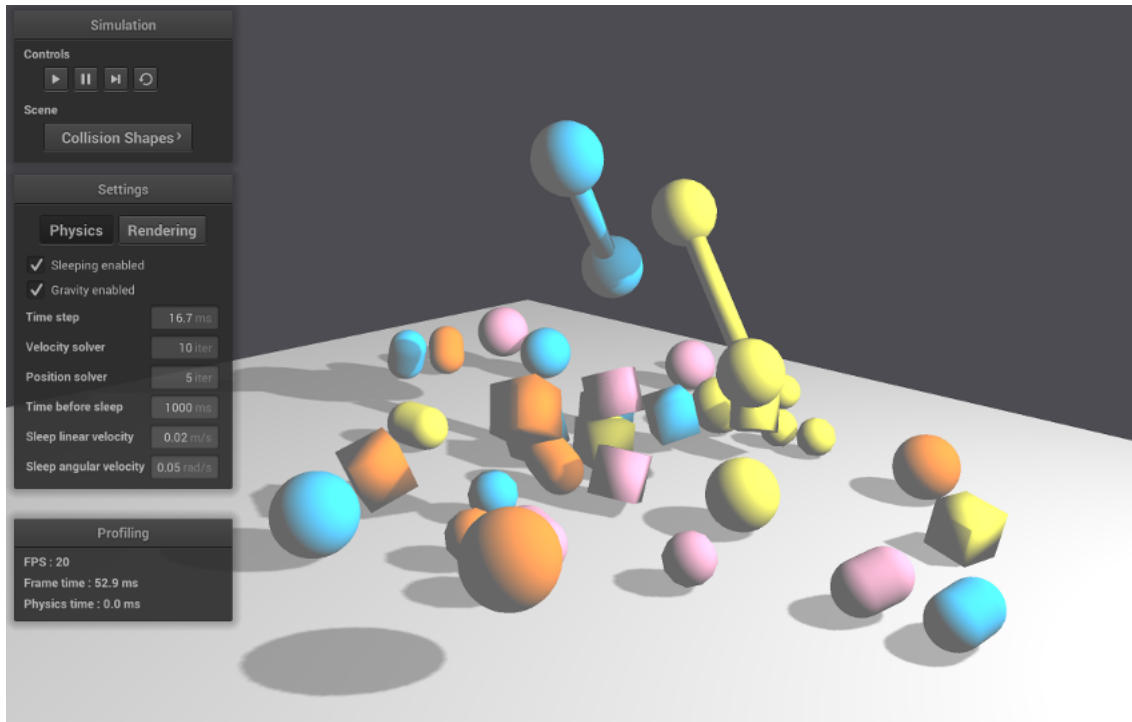
14.1 Cubes Scene

In this scene, you will see how to create a floor and some cubes using the Box Shape for collision detection. Because of gravity, the cubes will fall down on the floor. After falling down, the cubes will come to rest and start sleeping (become inactive). In this scene, the cubes will become red as they get inactive (sleeping).

14.2 Cubes Stack Scene

This scene has a dynamics world and a pyramid of cubes.

14.3 Joints Scene

In this scene, you will learn how to create different joints (ball and socket, hinge, slider, fixed) into the dynamics world. You can also see how to set the motor or limits of the joints.

14.4 Collision Shapes Scene

In this scene, you will see how to create a floor (using the Box Shape) and some other bodies using the different collision shapes available in the ReactPhysics3D library like capsules, spheres, boxes and convex meshes. Those bodies will fall down to the floor.

14.5 Heightfield Scene

In this scene, you will see how to use the Height field collision shape of the library. Several bodies will fall down to the height field.

14.6 Raycast Scene

In this scene, you will see how to use the ray casting methods of the library. Several rays are thrown against the different collision shapes. It is possible to switch from a collision shape to another using the spacebar key.

14.7 Collision Detection Scene

This scene has a collision world and several collision bodies that can be move around with keyboard keys. This scene shows how to manually compute collision detection in a collision world.

14.8 Concave Mesh Scene

In this scene, you will see how to use the static concave mesh collision shape of the library.

15 Retrieving contacts

There are several ways to get the contacts information (contact point, normal, penetration depth, . . .) from the `DynamicsWorld`.

15.1 Contacts of a given rigid body

If you are interested to retrieve all the contacts of a single rigid body, you can use the `RigidBody::getContactManifoldsList()` method. This method will return a linked list with all the current contact manifolds of the body. A contact manifold can contains several contact points.

Here is an example showing how to get the contact points of a given rigid body:

```cpp
// Get the head of the linked list of contact manifolds of
   the body
const ContactManifoldListElement* listElem;
listElem = rigidbody->getContactManifoldsList();

// For each contact manifold of the body
for (; listElem != nullptr; listElem = listElem->next) {
```

```
        ContactManifold* manifold = listElem->contactManifold;

        // For each contact point of the manifold
        for (int i=0; i<manifold->getNbContactPoints(); i++) {

            // Get the contact point
            ContactPoint* point = manifold->getContactPoint(i);

            // Get the world-space contact point on body 1
            Vector3 pos = point->getWorldPointOnBody1();

            // Get the world-space contact normal
            Vector3 normal = point->getNormal();
        }
}
```

Note that this technique to retrieve the contacts, if you use it between the `Dynamics World::update()` calls, will only give you the contacts at the end of each frame. You will probably miss several contacts that have occured in the physics internal sub-steps. In section 16, you will see how to get all the contact occuring in the physis sub-steps of the engine.

15.2   All the contacts of the world

If you want to retrieve all the contacts of any rigid body in the world, you can use the `DynamicsWorld::getContactsList()` method. This method will a `List` with the all the current contact manifolds of the world. A contact manifold may contain several contact points.

The following example shows how to get all the contacts of the world using this method:

```
rp3d::List<const rp3d::ContactManifold*> manifolds;

// Get all the contacts of the world
manifolds = dynamicsWorld->getContactsList();
rp3d::List<const rp3d::ContactManifold*>::iterator it;

// For each contact manifold
for (it = manifolds.begin(); it != manifolds.end(); ++it) {
    const rp3d::ContactManifold* manifold = *it;

        // For each contact point of the manifold
        rp3d::ContactPoint* contactPoint = manifold->
            getContactPoints();
        while (contactPoint != nullptr) {
```

```
        // Retrieve the world contact point and normal
            rp3d::Vector3 worldPoint = manifold->getShape1()
                ->getLocalToWorldTransform() * contactPoint->
                getLocalPointOnShape1();
            rp3d::Vector3 worldNormal = contactPoint->
                getNormal();

        // Move to the next contact point
            contactPoint = contactPoint->getNext();
        }
}
```

Note that this technique to retrieve the contacts, if you use it between the `Dynamics World::update()` calls, will only give you the contacts are the end of each frame. You will probably miss several contacts that have occured in the physics internal sub-steps. In section 16, you will see how to get all the contact occuring in the physis sub-steps of the engine.

16   Receiving Feedback

Sometimes, you want to receive notifications from the physics engine when a given event occurs. The `EventListener` class can be used for that purpose. In order to use it, you need to create a new class that inherits from the `EventListener` class and overrides some methods that will be called by the ReactPhysics3D library when some events occur. You also need to register your class in the physics world using the `DynamicsWorld::setEventListener()` as in the following code:

```
// Here, YourEventListener is a class that inherits
// from the EventListener class of reactphysics3d
YourEventListener listener;

// Register your event listener class
world.setEventListener(&listener);
```

16.1   Contacts

If you want to be notified when two bodies that were separated before become in contact, you need to override the `EventListener::beginContact()` method in your event listener class. Then, this method will be called when the two separated bodies becomes in contact.

If you receive a notification when a new contact between two bodies is found, you need to override the `EventListener::newContact()` method in your event listener class. Then, this method will be called when a new contact is found.

17 Profiler

If you build the library with the `RP3D_PROFILING_ENABLED` variable enabled (see section 4.4), a real-time profiler will collect information while the application is running. Then, at the end of your application, when the destructor of the `DynamicsWorld` class is called, information about the running time of the library will be displayed in the standard output. This can be useful to know where time is spent in the different parts of the ReactPhysics3D library in case your application is too slow.

Each collision or dynamics world has its own profiler. By default, the profiling report wil be written in a text file next to the executable. If you have multiple worlds in your application, there will be one profile file for each world. The profile files will be named after the name of the worlds. By defaults worlds will have names: world, world1, world2, world3, . . . You can change the name of the world by setting it into the `WorldSettings` object when you create the world (see section 8).

It is also possible to output the profiling report to another destination. To do this, you have to create your own profiler object before the creation of the physics world. You will then be able to add one or more profile destinations to the profiler. A destination can be either a file or an output stream (`std::ostream`) of your choice. For each destination, you also have to select the output format of the profiling report. When this is done, you have to give the pointer to your profiler object in paramater when you create the world.

The following example shows how to create your own profiler object and add a file destination (custom_profile.txt) and a stream destination (std::cout):

```cpp
// Create the profiler
rp3d::Profiler* profiler = new rp3d::Profiler();

// Add a log destination file
profiler->addFileDestination("custom_profile.txt", Profiler
    ::Format::Text);

// Add an output stream destination
profiler->addStreamDestination(std::cout, Profiler::Format::
    Text);

// Create the physics world with your profiler
rp3d::CollisionWorld world(rp3d::WorldSettings(), nullptr,
    profiler);
```

18 Logger

ReactPhysics3D has an internal logger that can be used to output logs while running the application. This can be useful for debugging for instance. To enable the logger, you need to build the library with the `RP3D_LOGS_ENABLED` variable enabled (see section 4.4).

Each collision or dynamics world has its own logger. By default, logs wil be written in an HTML file next to the executable. If you have multiple worlds in your application, there will be one log file for each world. The logs files will be named after the name of the worlds. By defaults worlds will have names: world, world1, world2, world3, ... You can change the name of the world by setting it into the `WorldSettings` object when you create the world (see section 8).

It is also possible to output the logs to another destination. To do this, you have to create your own logger object before the creation of the physics world. You will then be able to add one or more logs destinations to the logger. A destination can be either a file or an output stream (`std::ostream`) of your choice. For each destination, you also have to select the output format of the logs (text or HTML). When this is done, you have to give the pointer to your logger object in paramater when you create the world.

The following example shows how to create your own logger object and add a file destination (custom_log.html) and a stream destination (std::cout):

```cpp
// Create the logger
rp3d::Logger* logger = new rp3d::Logger();

// Log level (infor, warning and error
uint logLevel = static_cast<uint>(Logger::Level::Info) |
    static_cast<uint>(Logger::Level::Warning) |
static_cast<uint>(Logger::Level::Error);

// Add a log destination file
logger->addFileDestination("custom_log.html", logLevel,
    Logger::Format::HTML);

// Add an output stream destination
logger->addStreamDestination(std::cout, logLevel, Logger::
    Format::Text);

// Create the physics world with your logger
rp3d::CollisionWorld world(rp3d::WorldSettings(), logger);
```

19   Debug Renderer

...

20   API Documentation

Some documentation about the API of the code has been generated using Doxygen. You will be able to find this documentation in the library archive in the folder `/documentation/API/html/`. You just need to open the `index.html` file with your favorite web browser.

The API documentation is also available online here: `http://www.reactphysics3d.com/documentation/api/html/`

21   Issues

If you find some bugs, do not hesitate to report them on our issue tracker here:

`https://github.com/DanielChappuis/reactphysics3d/issues`

Thanks a lot for reporting the issues that you find. It will help us to correct and improve the library.