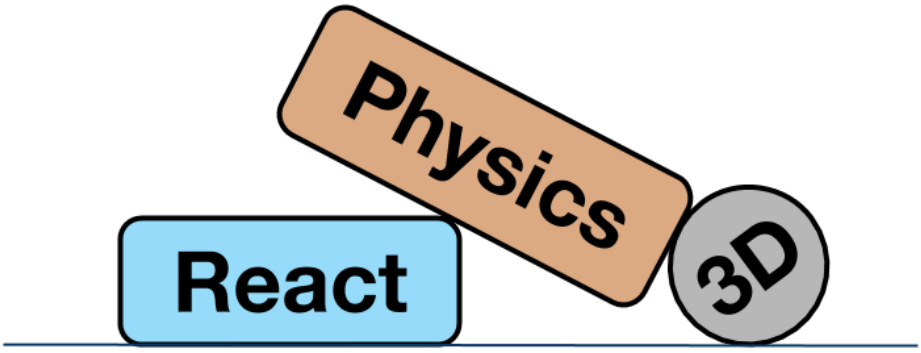
ReactPhysics3D
User Manual

Version: 0.10.0

Daniel Chappuis



http://www.reactphysics3d.com

March 5, 2024

Contents

1 Introduction **5**

2 Features **5**

3 License **5**

4 Building and installing the library **6**
- 4.1 Configure and generate the native tool files . . . 6
  - 4.1.1 Configure and generate with the command line (Linux and Mac OS X) . . . 6
  - 4.1.2 Configure and generate using the CMake graphical interface (Linux, Mac OS X and Windows) . . . 7
- 4.2 Building the library . . . 8
  - 4.2.1 Building the library using `make` on the command line (Linux, Mac OS X) . . . 8
  - 4.2.2 Building the library with Visual Studio (Windows) . . . 8
- 4.3 Installing the library . . . 9
  - 4.3.1 Installing the library using the `make` on the command line (Linux, Mac OS X) . . . 9
  - 4.3.2 Installing the library on Windows with Visual Studio . . . 9
- 4.4 CMake Options . . . 10

5 Using ReactPhysics3D in your application **11**

6 The PhysicsCommon object **13**

7 Memory Management **14**

8 Physics World **15**
- 8.1 Creating the Physics World . . . 16
  - 8.1.1 World settings . . . 16
- 8.2 Customizing the Physics World . . . 16
  - 8.2.1 Solver parameters . . . 16
  - 8.2.2 Sleeping . . . 17
- 8.3 Updating the Physics World . . . 17
- 8.4 Retrieving contacts . . . 19
- 8.5 Destroying the Physics World . . . 19

9 Rigid Body **20**
- 9.1 Creating a Rigid Body . . . 20
- 9.2 Type of a Rigid Body (static, kinematic or dynamic) . . . 20
- 9.3 Gravity . . . 21
- 9.4 Velocity Damping . . . 21
- 9.5 Sleeping . . . 21
- 9.6 Applying Force or Torque to a Rigid Body . . . 22
- 9.7 Updating a Rigid Body . . . 23

9.8 Mass, Center of Mass, Inertia Tensor . . . 25
9.8.1 Mass . . . 25
9.8.2 Center of mass . . . 25
9.8.3 Inertia Tensor . . . 25
9.9 Restricting linear/angular motion of a Rigid Body . . . 26
9.10 Destroying a Rigid Body . . . 26

10 Collider **27**
10.1 Collision Shapes . . . 27
10.1.1 Box Shape . . . 27
10.1.2 Sphere Shape . . . 28
10.1.3 Capsule Shape . . . 29
10.1.4 Convex Mesh Shape . . . 29
10.1.5 Concave Mesh Shape . . . 33
10.1.6 Heightfield Shape . . . 36
10.2 Collision filtering . . . 38
10.3 Material . . . 39
10.4 Simulation Collider . . . 40
10.5 World Query Collider . . . 40
10.6 Trigger . . . 41

11 Joints **42**
11.1 Ball and Socket Joint . . . 42
11.1.1 Cone limit . . . 43
11.2 Hinge Joint . . . 43
11.2.1 Limits . . . 44
11.2.2 Motor . . . 45
11.3 Slider Joint . . . 46
11.3.1 Limits . . . 46
11.3.2 Motor . . . 47
11.4 Fixed Joint . . . 48
11.5 Collision between the bodies of a Joint . . . 49
11.6 Destroying a Joint . . . 49

12 Ray casting **50**
12.1 Ray casting against multiple bodies . . . 50
12.1.1 The RaycastCallback class . . . 51
12.1.2 Raycast query in the world . . . 52
12.2 Ray casting against a single body . . . 52
12.3 Ray casting against the collider of a body . . . 53

13 Testbed application **53**

14 Receiving Feedback **53**
14.1 Contacts . . . 54
14.2 Triggers . . . 56

15 Profiler 57

16 Logger 57

17 Debug Renderer 58

18 Determinism 60

19 API Reference Documentation 60

20 Issues 60

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
- Debug 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

In order to build the library on your system, you first need to clone the code repository 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:

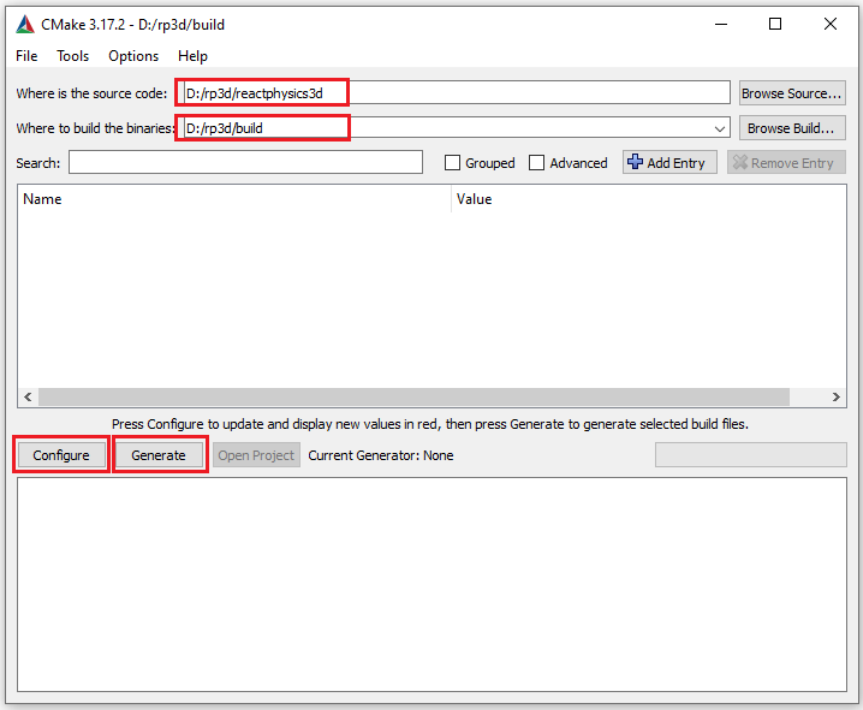
```
ccmake <path_to_library_source>
```

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

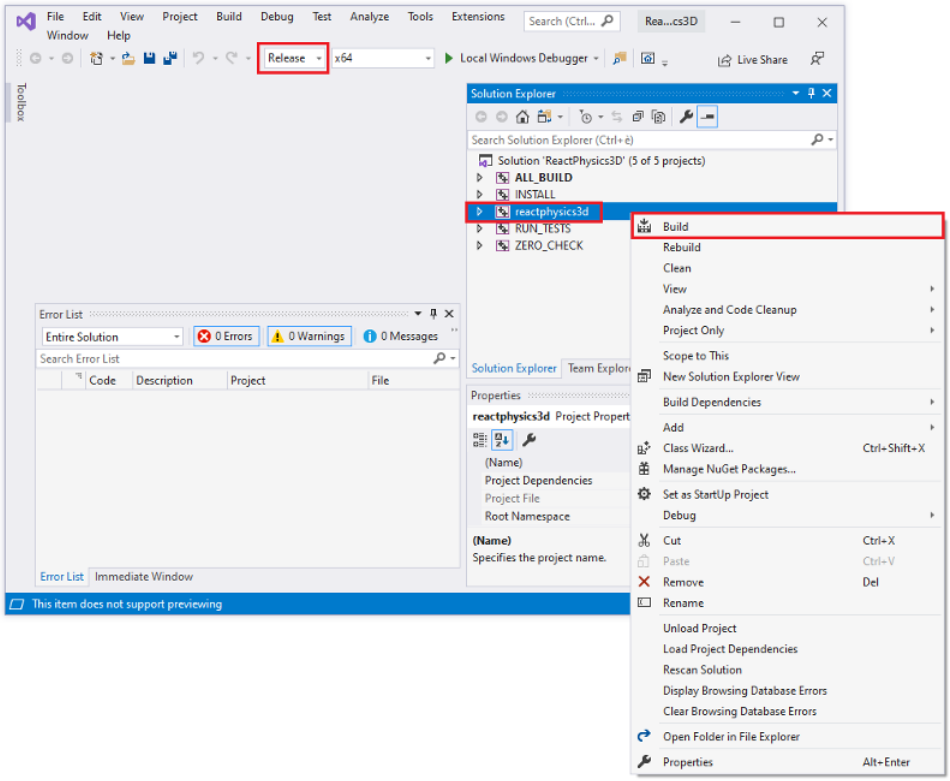
On Linux or Mac OS X, you can compile the library on the command line using the `make` command. Go into the directory where you have generated the native build tool files and run the following command:

```
make
```

The library will start compiling.

4.2.2 Building 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 header 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:

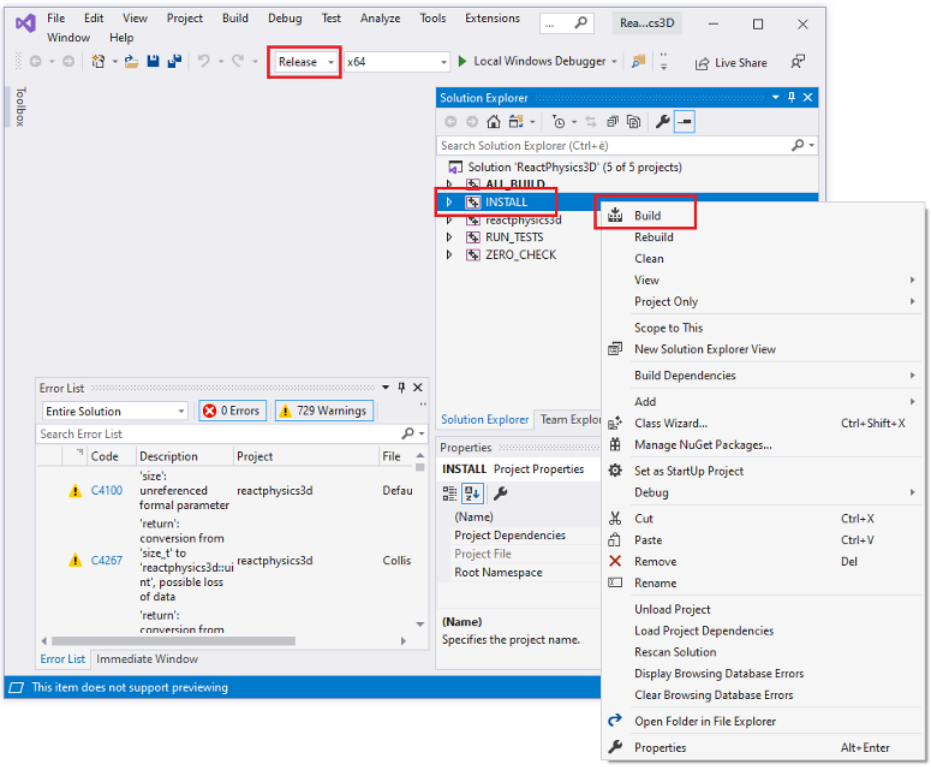
```
sudo make install
```

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 header 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 Files (x86)\ReactPhysics3D\` for instance.



4.4 CMake Options

You can find below 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 generated 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 with demos will be compiled. The testbed application uses OpenGL for rendering. Take a look at the section 13 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 15.

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 or Mac OS X, 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 or Mac OS X.

```
if(WIN32)
  list(APPEND CMAKE_PREFIX_PATH "C:\\Program Files (x86)\\
     ReactPhysics3D")
elseif(APPLE)
  list(APPEND CMAKE_PREFIX_PATH "/usr/local/lib/cmake/
     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 or Mac
    OS X
if(WIN32)
  list(APPEND CMAKE_PREFIX_PATH "C:\\Program Files (x86)\\
     ReactPhysics3D")
elseif(APPLE)
  list(APPEND CMAKE_PREFIX_PATH "/usr/local/lib/cmake/
     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 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 instantiate the `PhysicsCommon` class. This main object will then be used as a factory to instantiate 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 instantiate a physics world for instance:

```
// Create a physics world
PhysicsWorld* world = 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 by default 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 the allocated memory returned by the `allocate()` method must be 16 bytes aligned.

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 you 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.

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 `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 creation will not have any effect. 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 9.5 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. The `PhysicsWorld::update()` 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 exact 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 Retrieving contacts

Sometimes, you might need to get the contacts information (contact point, normal, penetration depth, ...) that occurs in your physics world.

If you are using a physics world to only test for collisions (you never call the `PhysicsWorld::update()` method), you can retrieve contacts information directly when you call the `PhysicsWorld::testCollision()` group of methods. Those methods take a pointer to a `CollisionCallback` class. You simply need to create a custom class that inherits from this class and override the `CollisionCallback::onContact()` method. When you call one of the `PhysicsWorld::testCollision()` methods, the `onContact()` method of your class will be called with all the information about the contacts in parameters.

However, if you are using ReactPhysics3D for a real-time simulation by calling the `PhysicsWorld::update()` method each frame, you should use the `EventListener` class to retrieve contacts as described in section 14.

8.5 Destroying the Physics World

When you don't need the physics world anymore, you can destroy it to release some memory. When the physics world 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 Rigid Body

Once the physics world has been created, you can add rigid bodies into it. A rigid body is an object that can be simulated using the laws of physics. It has a mass, a position, an orientation and one or several colliders. A rigid body can be used in different situations. It can be used to simulate a real world object simulated by the laws of physics (it will react to forces and collisions) or moved manually (like a character). It can also be used to represent a static objects that is not moving in the world (like a building for instance). We can also use a rigid body that is not simulated but there to launch a given action when colliding or overlapping with another rigid body of the world. In ReactPhysics3D, the `RigidBody` class is used to describe a rigid body.

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

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

Once your rigid body has been created in the world, you will probably want to add it one or more colliders as described in section 10.

9.2 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);
```

9.3 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);
```

9.4 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.

9.5 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

`RigidBody::setIsAllowedToSleep()` method as in the next example:

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

9.6 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.

9.7 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
float 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.

```

// 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/.

9.8 Mass, Center of Mass, Inertia Tensor

The mass, center of mass and inertia tensor of a rigid body are important parameters for the physical simulation of a rigid body.

9.8.1 Mass

The `RigidBody` has a mass value (in kilograms) which is 1 kilogram by default. There are two ways to set the mass of a rigid body. First, you can set it directly using the `RigidBody::setMass()` method. Secondly, it is also possible to compute this mass automatically using the mass of the colliders of the rigid body. As described in section 10.3, the material of each collider has a mass density value. This value is 1 by default. You change change the mass density value of the colliders of a rigid body and then use the `RigidBody::updateMassFromColliders()` method to automatically compute the mass of the rigid body using the mass density and shape of its colliders. Note that you will need to call this method again if you add another collider to the rigid body.

9.8.2 Center of mass

The center of mass of a `RigidBody` is the mean location of the distribution of mass of the body in space. By default the center of mass of the rigid body is located at its origin. There are two ways to set the center of mass of a rigid body. First, you can set it directly using the `RigidBody::setLocalCenterOfMass()` method. Secondly, as for the mass, the center of mass can also be computed automatically using the mass, shape and transform of all the colliders of the rigid body. As described in section 10.3, the material of each collider has a mass density value. This value is 1 by default. You can set the mass density value of the colliders and then use the `RigidBody::updateLocalCenterOfMassFromColliders()` method to automatically compute the center of mass of the rigid body. Note that you will need to call this method again if you add another collider to the rigid body.

9.8.3 Inertia Tensor

The inertia tensor of a `RigidBody` is a 3×3 matrix describing how the mass is distributed inside the rigid body which is used to calculate its rotation. The inertia tensor depends on the mass and the shape of the body. By default the local inertia tensor of a rigid body is the identity matrix. There are two ways to set the inertia tensor of a rigid body. First, you can set it directly using the `RigidBody::setLocalInertiaTensor()` method. Note that this will set the inertia tensor of the body in local-space coordinates which is usually a diagonal matrix. This method takes a `Vector3` with the three diagonal entries of the matrix. Secondly, the local inertia tensor can be computed automatically using the mass density, shape and transform of all the colliders of the body. As described in section 10.3, the material of each collider has a mass density value which is 1 by default. You can set the mass density value of the colliders and then use the `RigidBody::updateLocalInertiaTensorFromColliders()` method to

automatically compute the local inertia tensor of the body. Note that you will need to call this method again if you add another collider to the rigid body.

Note that it is also possible to automatically compute the mass, center of mass and inertia tensor of a rigid body at the same time using the `RigidBody::updateMassPropertiesFromColliders()`.

9.9 Restricting linear/angular motion of a Rigid Body

It is possible to use the `RigidBody::setLinearLockAxisFactor()` method to restrict the linear motion of a rigid body along the world-space X,Y and Z axes. For instance, the following code shows how to disable the linear motion of a rigid body along the world-space Y axis by setting the lock factor to zero for this axis.

```
// Disable motion along the Y axis
rigidBody->setLinearAxisFactor(Vector3(1, 0, 1));
```

In the same way, you can use the `RigidBody::setAngularLockAxisFactor()` method to restrict the angular motion of a rigid body around the world-space X,Y and Z axes. For instance, the following code shows how to disable the angular motion of a rigid body around the world-space Y axis by setting the lock factor to zero for this axis.

```
// Disable rotation around the Y axis
rigidBody->setAngularAxisFactor(Vector3(1, 0, 1));
```

9.10 Destroying a Rigid Body

It is really simple to destroy a rigid body when you don't need it anymore. 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);
```

10 Collider

To allow bodies to collide against each others, we need `colliders`. A collider (class `Collider`) describes the collision shape of a 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 add a new collider to a 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 presents the different types of collision shapes that are available in ReactPhysics3D.

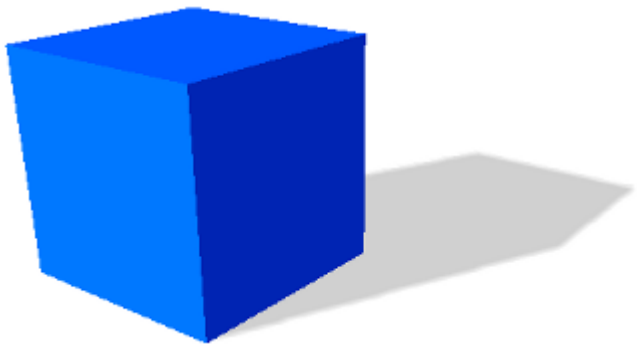
10.1 Collision Shapes

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

Note that ReactPhysics3D does not support collision between two concave shapes (`ConcaveMeshShape` and `HeightFieldShape`).

10.1.1 Box Shape

The `BoxShape` class describes a box collision shape centered at the origin of the collider. 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 Vector3 halfExtents(2.0, 3.0, 5.0);

// Create the box shape
BoxShape* boxShape = phycsicsCommon.createBoxShape(
   halfExtents);
```

10.1.2 Sphere Shape

The `SphereShape` class describes a sphere collision shape centered at the origin of the collider. 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
SphereShape* sphereShape = physicsCommon.createSphereShape
   (2.0);
```

10.1.3 Capsule Shape

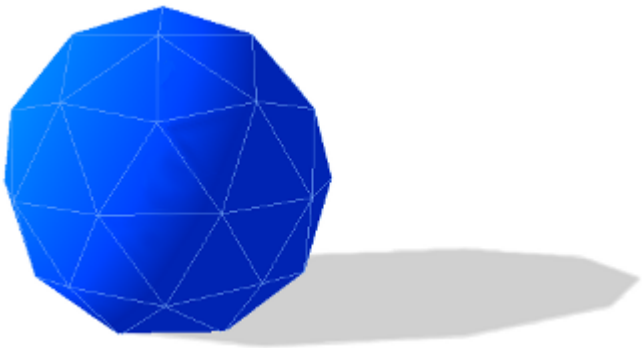
The `CapsuleShape` class describes a capsule collision shape around the local Y axis and centered at the origin of the collider. It is the convex hull of two spheres. It can also be seen as an elongated sphere. In order to create it, you only need to specify the radius of the two spheres and the height 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
CapsuleShape* capsuleShape = physicsCommon.
   createCapsuleShape(1.0, 2.0);
```

10.1.4 Convex Mesh Shape

The `ConvexMeshShape` class can be used to describe the shape of a convex mesh centered at the origin of the collider.



In order to create a `ConvexMeshShape`, we first need to create a `ConvexMesh`. A `ConvexMesh` can be instanciated once and used to create multiple `ConvexMeshShapes` with different scaling if necessary. There are two ways to create a `ConvexMesh`.

If you know the vertices and faces of your mesh, you can use the following method to create a `ConvexMesh`:

```
PhysicsCommon::createConvexMesh(const PolygonVertexArray&
    polygonVertexArray, std::vector<Message>& messages)
```

To use this method, you 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 can call the previous method to instantiate a `ConvexMesh` using your `PolygonVertexArray`.

The following example shows how to create a `ConvexMeshShape` when you know the vertices and faces of your mesh. 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] = {-3, -3, 3,
                      3, -3, 3,
                      3, -3, -3,
                      3, -3, -3,
                      -3, 3, 3,
                      3, 3, 3,
                      3, 3, -3,
                      -3, 3, -3};

// Vertex array with all vertices of the mesh
rp3d::VertexArray vertexArray(vertices, 3 * sizeof(float),
        24, rp3d::VertexArray::DataType::VERTEX_FLOAT_TYPE);

// Compute the convex mesh from the array of vertices
std::vector<rp3d::Message> messages;
ConvexMesh* convexMesh = physicsCommon.createConvexMesh(
    vertexArray, messages);

// Display the messages (info, warning and errors)
if (messages.size() > 0) {

  for (const rp3d::Message& message: messages) {

    std::string messageType;
    switch(message.type) {
      case rp3d::Message::Type::Information:
        messageType = "info";
        break;
```

```
      case rp3d::Message::Type::Warning:
        messageType = "warning";
        break;
      case rp3d::Message::Type::Error:
        messageType = "error";
        break;
    }

    std::cout << "Message (" << messageType << "): " <<
       message.text << std::endl;
  }
}

// Make sure there was no errors during mesh creation
assert(convexMesh != nullptr);
```

Note that the vertex coordinates and indices array are copied into the `ConvexMesh`. Therefore, you can release the memory of the `PolygonVertexArray` after the `ConvexMesh` creation. However, the mesh data is stored inside the `ConvexMesh`. Therefore, you should not release it until all the `ConvexMeshShapes` that have been instantiated from it are destroyed.

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.

Also note that meshes with duplicated vertices are not supported. The number of vertices you pass to create the `PolygonVertexArray` must be exactly the number of vertices in your convex mesh. The `PhysicsCommon::createConvexMesh()` will also report errors if some faces of your mesh have almost zero area (degenerated face). You will need to fix those errors in order to create the `ConvexMesh`.

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.

As discussed before, there is a second way to create a `ConvexMesh`. In case you only know the vertices of your mesh (you don't know the faces), you can use the following method:

```
PhysicsCommon::createConvexMesh(const VertexArray&
   vertexArray, std::vector<Message>& messages)
```

With this method, you only need to specify the array of vertices (`VertexArray`) of your mesh (not the faces). The library will use the QuickHull algorithm to compute a convex mesh that includes all the vertices.

```
// Array with the vertices coordinates of the convex mesh
float vertices[24] = {-3, -3, 3,
                      3, -3, 3,
                      3, -3, -3,
                      3, -3, -3,
                      -3, 3, 3,
                      3, 3, 3,
                      3, 3, -3,
                      -3, 3, -3};

// Vertex array with all vertices of the mesh
rp3d::VertexArray vertexArray(vertices, 3 * sizeof(float),
        24, rp3d::VertexArray::DataType::VERTEX_FLOAT_TYPE);

// Compute the convex mesh using only the array of vertices
std::vector<rp3d::Message> messages;
ConvexMesh* convexMesh = physicsCommon.createConvexMesh(
   vertexArray, messages);

// Display the messages (info, warning and errors)
if (messages.size() > 0) {

  for (const rp3d::Message& message: messages) {

  std::string messageType;
  switch(message.type) {
    case rp3d::Message::Type::Information:
      messageType = "info";
      break;
    case rp3d::Message::Type::Warning:
      messageType = "warning";
      break;
    case rp3d::Message::Type::Error:
      messageType = "error";
      break;
  }

  std::cout << "Message (" << messageType << "): " <<
     message.text << std::endl;
  }
}
```

```
// Make sure there was no errors during mesh creation
assert(convexMesh != nullptr);
```

Note that the `PhysicsCommon::createConvexMesh()` method takes a reference to an array of messages as parameter. If there are errors during the convex mesh creation, the method will return `null` and the array with contain some error messages describing the issues.

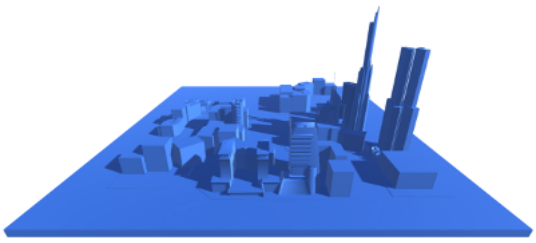
Now that we have a `ConvexMesh`, we can create a `ConvexMeshShape` as in the following example:

```
// Scaling factor
Vector3 scaling(1, 1, 1);

// Create the ConvexMeshShape
ConvexMeshShape* convexMeshShape = physicsCommon.
   createConvexMeshShape(convexMesh, scaling);
```

As you can see, you can specify a scaling factor in the `PhysicsCommon::createConvexMeshShape()` method when you create a `Convex MeshShape`. All the vertices of your mesh will be scaled from the local origin of the mesh by this factor. It means that you can use the same `ConvexMesh` for multiple `ConvexMeshShape` with a different scaling factor each time.

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

10.1.5 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. Note that collsions between two concave mesh shape are not supported.

In order to create a ConcaveMeshShape, we first need to create a TriangleMesh. A TriangleMesh describes your mesh made of triangles and can be reused multiple times to instantiate ConcaveMeshShapes with different scaling factors for instance. In order to create the TriangleMesh, we first need to instantiate a TriangleVertexArray describing the vertices and triangle faces of the mesh. We then pass the TriangleVertexArray to the PhysicsCommon::createTriangleMesh() method to create the TriangleMesh.

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.

The following code shows how to create a TriangleVertexArray and use it to create a TriangleMesh:

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

// Create the TriangleMesh
std::vector<rp3d::Message> messages;
TriangleMesh* triangleMesh = physicsCommon.
   createTriangleMesh(vertexArray, messages);

// Display the messages (info, warning and errors)
if (messages.size() > 0) {

  for (const rp3d::Message& message: messages) {

    std::string messageType;
    switch(message.type) {
      case rp3d::Message::Type::Information:
        messageType = "info";
        break;
      case rp3d::Message::Type::Warning:
        messageType = "warning";
        break;
      case rp3d::Message::Type::Error:
```

```
            messageType = "error";
            break;
        }

        std::cout << "Message (" << messageType << "): " <<
            message.text << std::endl;
    }
}

// Make sure there was no errors during mesh creation
assert(triangleMesh != nullptr);
```

The vertices and faces data are copied into the `TriangleMesh`. Therefore, after the `TriangleMesh` creation, you can release your `TriangleVertexArray` but you need to keep the `TriangleMesh` until all the `ConcaveMeshShapes` that have been instantiated from it are destroyed.

In the previous example, the vertices normals that are needed for collision detection are automatically computed. However, you can specify your own vertices normals by using another constructor for the `TriangleVertexArray`. Note that each vertex normal is computed as weighted average of the face normals of all the neighboring triangle faces. Therefore, if you specify your mesh with duplicated vertices when you create the *TriangleVertexArray*, the automatic vertices normals computation will not give correct normals because each vertex of the mesh will only be part of a single triangle face. In this case, you must provide your own vertices normals when you create the *TriangleVertexArray*.

The `PhysicsCommon::createTriangleMesh()` will also report errors if some faces of your mesh have almost zero area (degenerated face). You will need to fix those errors in order to create the `TriangleMesh`. It is also a good idea to avoid coplanar faces for better collision detection.

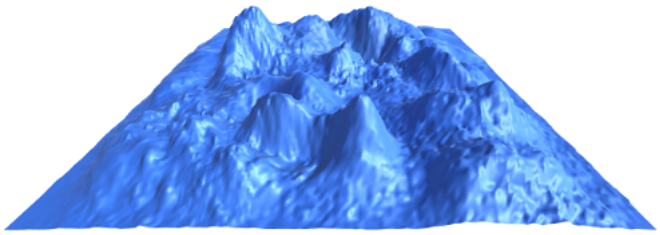
Now that we have a `TriangleMesh`, we can create the actual `ConcaveMeshShape`.

```
const Vector3 scaling(1, 1, 1);

// Create the ConcaveMeshShape using the TriangleMesh
ConcaveMeshShape* concaveMeshShape = physicsCommon.
    createConcaveMeshShape(triangleMesh, scaling);
```

As you can see, you can specify a scaling factor in the `PhysicsCommon::createConcaveMeshShape()` method when you create a `ConcaveMeshShape`. All the vertices of your mesh will be scaled from the local origin of the mesh by this factor. Note that you can use the same `TriangleMesh` instance for

creating multiple `ConcaveMeshShape` with a different scaling factor each time.

10.1.6 Heightfield Shape



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

First, we need to create an `HeightField`. To do that, we need to have an array with all the height values of our field. We can have height values of type integer, float or double. We need to specify the number of rows and columns of our two dimensional grid. Note that the height values in our array must be organized such that the value at row `indexRow` and column `indexColumn` is located at the following position in the array:

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

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

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

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

std::vector<rp3d::Message> messages;
HeightField* heightField = physicsCommon.createHeightField(
   NB_POINTS_WIDTH, NB_POINTS_LENGTH,
               heightValues, rp3d::HeightField::
                  HeightDataType::HEIGHT_FLOAT_TYPE,
               messages);

// Display the messages (info, warning and errors)
if (messages.size() > 0) {
```

```
  for (const rp3d::Message& message: messages) {

    std::string messageType;
    switch(message.type) {
      case rp3d::Message::Type::Information:
        messageType = "info";
        break;
      case rp3d::Message::Type::Warning:
        messageType = "warning";
        break;
      case rp3d::Message::Type::Error:
        messageType = "error";
        break;
    }

    std::cout << "Message (" << messageType << "): " <<
       message.text << std::endl;
  }
}

// Make sure there was no errors during the height field
   creation
assert(heightField != nullptr);
```

The height values are copied into the `HeightField` instance. Therefore, after the `HeightField` creation, you can release the memory of your array of height values but you need to keep the `TriangleMesh` until all the `HeightFieldShapes` that have been instantiated from it are destroyed.

Once we have our `HeightField` instance, we can use it to create an actual `HeightFieldShape` as in the following example:

```
  HeightFieldShape* heightFieldShape = physicsCommon.
     createHeightFieldShape(heightField);
```

You can also specify a scaling factor in the `PhysicsCommon::createHeightShape()` method when you create a `HeightFieldShape`. All the vertices of your mesh will be scaled from the local origin of the shape by this factor.

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.

10.2 Collision filtering

By default all the colliders of the 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 colliders of bodies into categories. Then we can specify for each collider against which categories of colliders it will be able to collide.

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

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


// Set the collision category for each collider of
// each of the four bodies
colliderBody1->setCollisionCategoryBits(CATEGORY1);
colliderBody2->setCollisionCategoryBits(CATEGORY2);
colliderBody3->setCollisionCategoryBits(CATEGORY3);
colliderBody4->setCollisionCategoryBits(CATEGORY3);
```

As you can see, the collider of body 1 will be part of the category 1, the collider of body 2 will be part of the category 2 and the colliders of bodies 3 and 4 will be part of the category 3.

Now, for each collider, we need to specify with which categories it is allowed to collide with. To do this, you need to use the `Collider::setCollideWithMaskBits()` method. 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 collider can collide.

```
// For each collider, we specify with which categories it
// is allowed to collide
colliderBody1->setCollideWithMaskBits(CATEGORY3);
colliderBody2->setCollideWithMaskBits(CATEGORY1 | CATEGORY3)
   ;
colliderBody3->setCollideWithMaskBits(CATEGORY2);
colliderBody4->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 collider is able to collide with another only if you have specify that the category mask of the first collider is part of the *collide with* mask of the second collider. 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 12). For instance, you can perform a ray cast test against a given subset of categories of colliders only.

10.3 Material

The material of a collider is used to describe the physical properties it is made of. This is represented by the `Material` class. Each collider that you create will have a default material. You can get the material of the collider using the `Collider::getMaterial()` method.

You can use the material to set those physical properties.

For instance, you can change the bounciness of the collider. 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 collider will not be bouncy at all. To change the bounciness of the material, you can use the `Material::setBounciness()` method.

It is also possible to set the mass density of the collider which has a default value of 1. As described in section 9.8, the mass density of a collider can be used to automatically compute the mass, center of mass and inertia tensor of a rigid body. In order to change the mass density of a collider, you need to use the `Material::setMassDensity()` method.

You are also able to change the friction coefficient of the collider. This value needs to be between 0 and 1. If the value is 0, no friction will be applied when the collider is in contact with another collider. 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.

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

```
// Get the current material of the collider
Material& material = collider->getMaterial();

// Change the bounciness of the collider
material.setBounciness(0.4);

// Change the friction coefficient of the collider
material.setFrictionCoefficient(0.2);
```

A collider can have different behaviors. It can be a *simulation collider*, a *world query collider* or a *trigger*. It can have also have two behaviors (except *simulation collider* and *trigger* at the same time). By default, when you create a collider, it will be a *simulation collider* and a *world query collider* but not a *trigger*. We will now take a look at those behaviors.

10.4 Simulation Collider

A simulation collider is a collider that will be able to collide against other simulation colliders in the world. It means that when two simulation colliders collide, contacts will be created and used to compute the forces/torques needed to make their bodies to bump into each other. For instance, if you want to simulate a body that is falling on the floor and bumping against it, you will need to have a least of simulation collider in both the falling body and the floor body.

To set a collider as being a simulation collider, you need to use the `Collider::setIsSimulationCollider()` method as in the following example:

```
bombCollider->setIsSimulationCollider(true);
```

When you create a collider, it is a simulation collider by default.

10.5 World Query Collider

A world query collider is a collider used to perform world queries on it. World queries are manual queries perform on the physics world like raycasting or testing if two collider collide or overlap using the following method for instance:

testOverlap() Those methods can be used to test whether the colliders of two bodies overlap or not.

testCollision() Those methods will give you the collision information (contact points, normals, ...) between two colliding bodies.

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

raycast() Those methods will try to find intersection between a ray and colliders of the world.

For instance, if you want to test a body of your scene with raycasting, you need to set the collider of that body as being a world query collider.

To set a collider as being a world query collider, you need to use the `Collider::setIsWorldQueryCollider()` method as in the following example:

```
bombCollider->setIsWorldQueryCollider(true);
```

A collider can be a world query collider and a simulation collider at the same time. When you create a collider, it is a world query collider by default.

10.6 Trigger

A trigger, is a collider that cannot collide with any other colliders but can only report when it is overlapping with another collider. For instance, consider a game where a player moves around and has to avoid touching some bombs. The player has a rigid body with a capsule collider for instance and the bombs are rigid bodies where each one has a sphere collider. Now, you do not want the player to bump against the bombs but you only want to know when the collider of the player overlaps with the collider of a bomb. In this case, you can set bombs colliders as being triggers. In this case, no rigid bodies in the world will be able to collide against the bombs colliders but you can receive notifications when a collider (the player body collider for instance) overlaps with a bomb collider. Therefore, you can use this to know when the player touches a bomb of the world.

To set a collider as being a trigger, you need to use the `Collider::setIsTrigger()` method as in the following example:

```
bombCollider->setIsTrigger(true);
```

Note that a collider cannot be a trigger and a simulation collider at the same time. When you create a collide it is not a trigger by default.

The section 14.2 describes how to use the `EventListener` class to receive notifications when colliders overlap with some triggers.

11 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.

11.1 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. The `BallAndSocketJointInfo` class contains different constructors that you can use whether you want to specify the anchor point in world-space or local-space coordinates. The joint will make sure that the two local-space anchor points match in world-space. Make sure that the two bodies are in a valid position (with respect to the joint constraint) at the beginning of the simulation.

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

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

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

Now, it is time to create the actual joint in the physics world using the `PhysicsWorld::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 physics world
```

```
BallAndSocketJoint* joint;
joint = dynamic_cast<BallAndSocketJoint*>(world->createJoint
    (jointInfo));
```

11.1.1 Cone limit

With the ball and socket joint, it is possible to enable a cone limit. Let's call the `anchor axis` for body 1 the axis from body 1 center of mass to the joint anchor point. The cone limit can be used to constraint the angle between the anchor axis of body 1 and the anchor axis of body 2. The idea is to limit the angle of the anchor axis of body 2 inside a cone around the anchor axis of body 1. The cone is defined by its main axis which is the anchor axis of body 1 and is also defined by the cone half angle.

In the following example, we enable the cone limit for a given ball and socket joint.

```
// Set the maximum half angle (in radians)
joint->setConeLimitHalfAngle(45.0 * PI / 180.0);

// Enable the cone limit for this joint
joint->enableConeLimit(true);
```

11.2 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 and also the hinge rotation axis. The `HingeJointInfo` class contains different constructors that you can use whether you want to specify the anchor point or the rotation axis in local-space or world-space. Make sure that the two bodies are in a valid position (with respect to the joint constraint) at the beginning of the simulation.

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

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

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

// Create the joint info object
```

```
HingeJointInfo jointInfo(body1, body2, anchorPoint, axis);
```

Now, it is time to create the actual joint in the physics world using the `PhysicsWorld::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 physics world
HingeJoint* joint;
joint = dynamic_cast<HingeJoint*>(world->createJoint(
    jointInfo));
```

11.2.1 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
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 physis world
HingeJoint* joint;
joint = dynamic_cast<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.

11.2.2 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
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 physics world
HingeJoint* joint;
joint = dynamic_cast<HingeJoint*>(world->createJoint(
    jointInfo));
```

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

11.3 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 and the slider axis direction. The constructor of the `SliderJointInfo` object needs two pointers to the bodies of the joint, the anchor point and the axis direction. The `SliderJointInfo` class contains different constructors that you can use whether you want to specify the anchor point and the direction axis in local-space or world-space. Make sure that the two bodies are in a valid position (with respect to the joint constraint) at the beginning of the simulation.

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

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

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

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

Now, it is possible to create the actual joint in the physics world using the `PhysicsWorld::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 physics world
SliderJoint* joint;
joint = dynamic_cast<SliderJoint*>(world->createJoint(
   jointInfo));
```

11.3.1 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:

```
// Create the joint info object
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 physics world
SliderJoint* joint;
joint = dynamic_cast<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.

11.3.2 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
```

```
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 physics world
SliderJoint* joint;
joint = dynamic_cast<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.

11.4 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 to create the `FixedJointInfo` object. The `FixedJointInfo` class contains different constructors that you can use whether you want to specify the anchor point in local-space or world-space. Make sure that the two bodies are in a valid position (with respect to the joint constraint) at the beginning of the simulation.

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

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

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

Now, it is possible to create the actual joint in the physics world using the `PhysicsWorld::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:

```
// Create the fixed joint in the physics world
FixedJoint* joint;
joint = dynamic_cast<FixedJoint*>(world->createJoint(
    jointInfo));
```

11.5 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:

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

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

// Create the joint in the physics world
HingeJoint* joint;
joint = dynamic_cast<HingeJoint*>(world->createJoint(
    jointInfo));
```

11.6 Destroying a Joint

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

```
// 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.

12 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 collider 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 ReactPhysics3D. 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
Vector3 startPoint(0.0, 5.0, 1.0);
Vector3 endPoint(0.0, 5.0, 30);

// Create the ray
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 collider 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 rigid body that has been hit by the ray

collider Pointer to the collider 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 colliders. Collision filtering is described in section 10.2.

12.1 Ray casting against multiple bodies

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

12.1.1 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 collider 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 collider, ...).

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 collider 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 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);
    }
};
```

12.1.2 Raycast query in the world

Now that you have your own raycast callback class, you can use the `PhysicsWorld::raycast()` method to perform a ray casting test on a physics 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 colliders as described in section 10.2.

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

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

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

12.2 Ray casting against a single body

You can also perform ray casting against a single specific rigid body of the world. To do this, you need to use the `RigidBody::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
Vector3 startPoint(1 , 2, 10);
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);
```

12.3 Ray casting against the collider of a body

You can also perform ray casting against a single specific collider of a rigid body of the world. To do this, you need to use the `Collider::raycast()` method of the given collider. 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 collider:

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

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

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

13 Testbed application

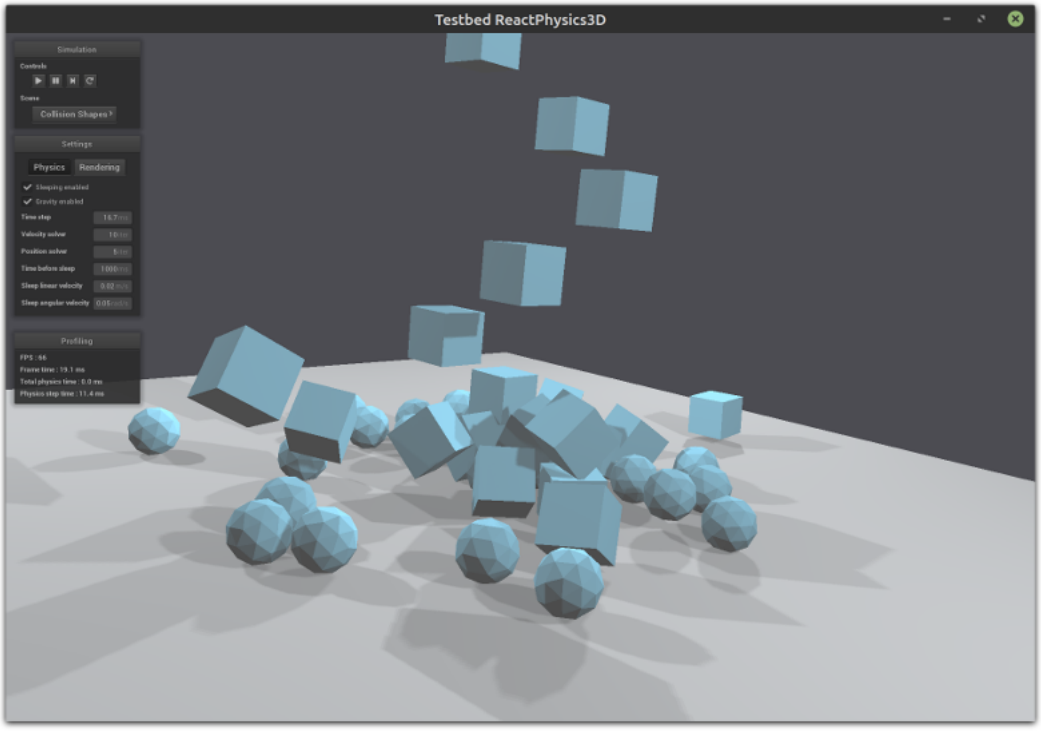
The testbed application contains a graphical interface where you can select and see several demo scenes using the ReactPhysics3D library.

As described in section 4, you can enable a `CMake` option to build the testbed application when you build the library. 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.

14 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 `PhysicsWorld::setEventListener()` as in the following code:

```
// Your event listener class
class YourEventListener : public EventListener {

  ...
};

YourEventListener listener;

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

14.1 Contacts

The `EventListener` contains a `onContact()` method. If you override this method in your custom event listener class, this method will be called when you call the `PhysicsWorld::update()` method if there are contacts to report during the current update.

The `onContact()` method will be called with a `CollisionCallback::CallbackData` object in parameter. This object has a `getNbContactPairs()` and a `getContactPair()` methods that will allow you to get each contact pair (class `CollisionCallback::ContactPair`) that occured during the physics world update. Note that a contact pair contains a list of con-

tact points between two bodies. Once you get a contact pair, you can use the `getBody1()` or `getBody2()` methods to get the bodies in contact, you can use the `getCollider1()` or `getCollider2()` methods to get the colliders in contact and the `getNbContactPoints()` and `getContactPoint()` methods to get the contact points (class `CollisionCallback::ContactPoint`) of the contact pair. Note that the `onContact()` method will report contacts after the collision detection phase and before the contact resolution.

The contact pair also has a `getEventType()` method that gives information about the type of contact event for the pair. It can be `ContactStart` if the two bodies of the contact pair where not colliding in the previous call to `PhysicsWorld::update()` and are now colliding. It can be `ContactStay`, if the two bodies where colliding in the previous frame and are still colliding or it can be `ContactExit` if the two bodies where colliding in the previous frame but are not colliding anymore. Note that in this last situation, there will be no contact points in the contact pair.

As you know, rigid bodies can sleep. During that time, no contacts will be reported using the `EventListener::onContact()` method. Therefore, If you stop receiving contact notifications for a rigid body, it does not necessarily mean that there is no contacts anymore but it could mean that the body has fallen asleep. You need to use the `getEventType()` method to know if the body is not colliding anymore (when you receive an event of type `ContactExit`).

In the following example, you can see how to override the `EventListener::onContact()` method to get the current contact points of the physics world:

```
// Your event listener class
class YourEventListener : public EventListener {

  // Override the onContact() method
  virtual void onContact(const CollisionCallback::
     CallbackData& callbackData) overrride {

    // For each contact pair
    for (uint p = 0; p < callbackData.getNbContactPairs(); p
       ++) {

      // Get the contact pair
      CollisionCallback::ContactPair contactPair =
         callbackData.getContactPair(p);

      // For each contact point of the contact pair
      for (uint c = 0; c < contactPair.getNbContactPoints();
          c++) {

          // Get the contact point
```

```
            CollisionCallback::ContactPoint contactPoint =
               contactPair.getContactPoint(c);

            // Get the contact point on the first collider and
                convert it in world-space
            Vector3 worldPoint = contactPair.getCollider1()->
               getLocalToWorldTransform() * contactPoint.
               getLocalPointOnCollider1();

            ...
        }
      }
   }

};
```

14.2 Triggers

As described in section 10.6 it is possible to set a collider as being a trigger. You can do this if you do not want any collision with this collider but you are only interested in knowing when another collider enters or exit the volume of the collider. If you do this, you will need to override the `onTrigger()` method of the `EventListener`. This method will be called and report all the collision with your triggers when you call the `PhysicsWorld::update()` method.

The `onTrigger()` method will be called with an `OverlapCallback::CallbackData` object in parameter. This object has a `getNbOverlappingPairs()` and a `getOverlappingPair()` methods that will allow you to get each overlapping pair (class `OverlapCallback::OverlapPair`) that occured during the physics world update. Once you get an overlapping pair, you can use the `getBody1()` or `getBody2()` methods to get the bodies in contact and the `getCollider1()` or `getCollider2()` methods to get the trigger colliders in contact. Note that the `onTrigger()` method will report overlaping triggers after the collision detection phase and before the contact resolution.

The overlapping pair also has a `getEventType()` method that gives information about the type of overlapping event for the pair. It can be `OverlapStart` if the two bodies of the pair where not overlapping in the previous call to `PhysicsWorld::update()` and are now overlapping. It can be `OverlapStay`, if the two bodies where overlapping in the previous frame and are still overlapping or it can be `OverlapExit` if the two bodies where overlapping in the previous frame but are not overlapping anymore.

As you know, rigid bodies can sleep. During that time, no overlap will be reported using the `EventListener::onTrigger()` method. Therefore, If you stop receiving trigger overlap notifications for a rigid body, it does not necessarily mean that there is no overlap anymore but it could mean that the body has fallen asleep. You need

to use the `getEventType()` method to know if the body is not overlapping anymore (when you receive an event of type `OverlapExit`).

Note that the `onTrigger()` method is only called for colliders that you have set as triggers using the `Collider::setIsTrigger()` method.

15 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 `PhysicsWorld` class is called, information about the running time of the library will written to a file. 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 physics 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).

16 Logger

ReactPhysics3D has an internal logger that can be used to get logs from the library while the application is running. This can be useful for debugging for instance.

The logger is part of the `PhysicsCommon` class. By default there is no logger set. If you want to create your custom logger to receive logs from ReactPhysics3D, you can inherit the `Logger` class and override the `Logger::log()` method. Then, you have to use the `PhysicsCommon::setLogger()` method to set the logger.

Getting the logs is the way for you to see the errors reported by the library and therefore, you should not just ignore the errors from the logs. You should at least redirect the logs (warnings and errors) in the standard output while debugging your application as described in the example below.

If you don't want to create your custom logger class, you can use the default logger of ReactPhysics3D (class `DefaultLogger`). With this logger, you can output the logs into a file or a std::stream. The logs can have different format like raw text of HTML. In order to instantiate the default logger, you need to call the `PhysicsCommon::createDefaultLogger()` method. Then, you can customize this logger and finally, you need to set this logger using the

`PhysicsCommon::setLogger()` method.

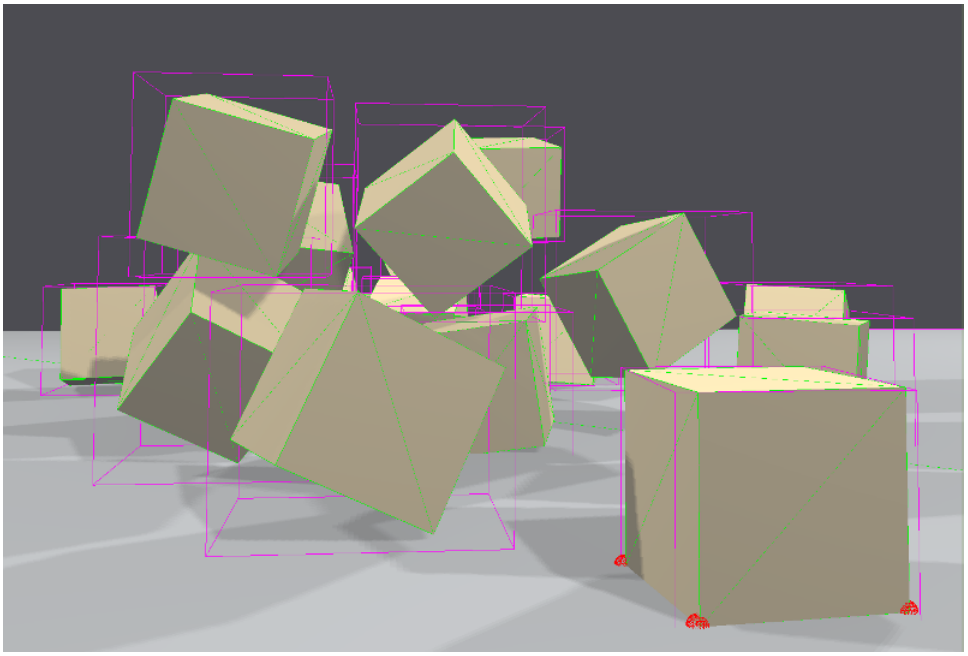
The following code shows how to create a default logger that will output logs (warning and errors) in HTML into a file and in raw text into the standard output:

```
// Create the default logger
DefaultLogger* logger = physicsCommon.createDefaultLogger();

// Log level (warnings and errors)
uint logLevel = static_cast<uint>(static_cast<uint>(Logger::
   Level::Warning) | static_cast<uint>(Logger::Level::Error)
   ;

// Output the logs into an HTML file
logger->addFileDestination("rp3d_log_" + name + ".html",
   logLevel, DefaultLogger::Format::HTML);

// Output the logs into the standard output
logger->addStreamDestination(std::cout, logLevel,
   DefaultLogger::Format::Text);

// Set the logger
physicsCommon.setLogger(logger);
```

17 Debug Renderer

For debugging purpose, it can be useful to display the physics objects of ReactPhysics3D on top of your simulation. This is possible using the `DebugRenderer` class of the library that allows you to display the colliders shapes, the AABB of the colliders or the contact points for instance. This can help you to check for instance if the physics collider of ReactPhysics3D is correctly aligned with the object you are rendering. When debug rendering is enabled, ReactPhysics3D will generate each time you call the `PhysicsWorld::update()` method (each frame), some rendering primitives (lines and triangles) that you will be able to retrieve and display on you simulation (with OpenGL or DirectX for instance). Note that ReactPhysics3D will only generate the arrays of lines and triangles but you are responsible to draw them.

By default, the debug rendering is disabled. You can activate it using the `PhysicsWorld::setIsDebugRenderingEnabled()` method. Note that you should disable it for the final release because this can be quite expensive to compute. You can get a reference to `DebugRenderer` of the physics world using the `PhysicsWorld::getDebugRenderer()` method. You also need to enable debugging for each body that you want to debug using the `Body::setIsDebugEnabled()` method. By default, debuging is disabled for all bodies. Then, you need to select the debug items (class `DebugRenderer:DebugItem`) you want to display. For instance, you might want to display the shapes of



the colliders (`DebugItem::COLLISION_SHAPE`), the broad-phase AABBs of the colliders (`DebugItem::COLLIDER_BROADPHASE_AABB`), the colliders AABBs (`DebugItem::COLLIDER_AABB`), the contact points (`DebugItem::CONTACT_POINT`) or the contact normals (`DebugItem::CONTACT_NORMAL`). You need to use the `DebugRenderer::setIsDebugItemDisplayed()` method to select if you want or not a given debug item to be displayed.

The following code shows how to enable debug rendering in order to display contact points and contact normals:

```
// Enable debug rendering
physicsWorld->setIsDebugRenderingEnabled(true);

// Enable debugging for each rigid body that we want to
   display
for (RigidBody* body: rigidBodies) {
  body->setIsDebugEnabled(true);
}

// Get a reference to the debug renderer
DebugRenderer& debugRenderer = physicsWorld->
   getDebugRenderer();

// Select the contact points and contact normals to be
   displayed
debugRenderer.setIsDebugItemDisplayed(DebugRenderer::
   DebugItem::CONTACT_POINT, true);
debugRenderer.setIsDebugItemDisplayed(DebugRenderer::
   DebugItem::CONTACT_NORMAL, true);
```

Now each time you call the `PhysicsWorld::update()` method, ReactPhysics3D will generate arrays with lines and triangles so that you can draw them in your application. You can use the `DebugRenderer::getNbLines()` and `DebugRenderer::getNbTriangles()` methods to know the number of lines and triangles to draw. Then, you need to call the `DebugRenderer::getLinesArray()` and `DebugRenderer::getTrianglesArray()` methods to get pointers to the beginning of the lines and triangles arrays. The lines array contain elements of type `DebugRenderer:DebugLine` with the two points of a line and their colors and the triangles array has elements of type `DebugRenderer:DebugTriangle` with the three points of each triangle and their colors. Note that the vertices of the lines and triangles are defined in world-space coordinates of the physics world.

18 Determinism

Sometimes, it is important that the simulation of a physics world behaves exactly the same each time we run it. Because of the differences in compilers and computer hardware it is quite difficult to achieve this between different machines. However, ReactPhysics3D should be deterministic when compiled with the same compiler and running on the same machine. In order to obtain two similar runs of the simulation of a physics world, it is adviced to completely destroy and recreate the physics world, the bodies and the joints inside it (in order to reset all the internal data cached for the simulation). You must also create the bodies and joints in the same order each time and make sure that all the calls to the methods of your physics world, bodies and joints happen in the same order.

19 API Reference Documentation

ReactPhysics3D contains Doxygen documentation for its API.

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

You can also find it in the library archive in the folder `/documentation/API/html/`. You just need to open the `index.html` file with your favorite web browser.

20 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 correct and improve the library.