Physics Wrapper
User Guide

Matthew Langford

Updated: 2017 - 03 - 13

Contents

1 Introduction

This Physics Wrapper library aims to provide an easy-to-use way of integrating a realistic physics simulation into your application. The wrapper is built on the Bullet Physics library[1], so it can deliver accurate behaviour despite the simple interface. You may find this useful if you just want a quick way of setting up a simple simulation, or if you wish to get to grips with the fundamentals of physics engines before moving onto a more fully-featured library, with a correspondingly more complicated interface.

2 Setup

In order to make use of the wrapper, you will need a C++ development environment with access to the Bullet Physics library.

Assuming you haven't already done so, download the Physics Wrapper from Github: `https://github.com/Mattadon/ML-Bullet-Wrapper`. The repository contains the static library compiled for Windows in `lib/`. Alternatively, you can generate project files and compile the library with Premake using the provided `premake4.lua`.

When using this library in a project, you will have to add the provided `include/` directory to your include path, and add the `lib/` directory to your linker search path. You will also have to link some libraries from your Bullet install in a certain order:

- `libPhysicsWrapper` (Our library)
- `libBulletDynamics`
- `libBulletCommon`
- `libLinearMath`

On a non-microsoft compiler, you may also have to link the following library for the CImg component to work:

- `gdi32`

The wrapper has only been tested on Windows.

[1] `http://bulletphysics.org/wordpress/`

```
//Include the parts of the wrapper that we need.
//In this case, we want the physics world and the basic rigid objects
#include <SimpleBulletWrapper/include/PhysicsObjectTypes.h>
#include <SimpleBulletWrapper/include/PhysicsWorld.h>

main()
{
    /* Set up visual code, eg: an OpenGL environment */

    //Set up the physics engine itself
    //Let's use Earth gravity, and work in degrees...
    PhysicsWorld* world = new PhysicsWorld(1.0f, false);

    //Now let's add some objects
    PhysicsBox box(world);
    PhysicsPlane plane(world);

    //Main application loop
    while(running)
    {
        //Step the world at the rate of 60 steps/s
        world->stepWorld(1 / 60.0f);

        /* Code to draw/output the scene information */
    }

    //Clean up
    delete world;
}
```

Figure 1: Wrapper-specific code to set up the most basic possible simulation. All visualisation code has been omitted.

3 Basic Demo

Figure 1 demonstrates the most basic physics simulation that you can set up using this wrapper; it drops a unit cube on to an infinite plane. Though this example is very simple, all programs using the wrapper will be built around this structure:

- Create the world
- Add some physics objects
- In a loop, step the world

Of course, without a way of extracting the location of the physics objects, you cannot represent the state of the simulation. You can get transform data from the physics objects like so:

```
glm::vec3 objectPosition = this->physicsObject->getPosition();
glm::quat objectRotation = this->physicsObject->getRotationQuaternion();
```

4 The Physics World

The `PhysicsWorld` class keeps track of the state of the simulation. You must include a `PhysicsWorld` object in your program in order for a simulation to exist.

4.1 Constructor

```
PhysicsWorld(glm::vec3 gravity, bool useRadians);
```

`gravity`: Define a gravity vector in $m.s^{-2}$. For example, Earth's gravity acting in the negative Y direction would be `glm::vec3(0.0f, -9.81f, 0.0f)`.

`useRadians`: Set to true if angles are going to be provided in radians. False if in degrees.

```
PhysicsWorld(float gravityStrength, bool useRadians)
    ;
```

`gravityStrength`: Provide a multiple of G ($=-9.81m.s^{-2}$) to act in the -Y direction. In this case, Earth's gravity will be used if `1.0f` is provided.

`useRadians`: See alternative constructor, above.

4.2 Public Member Functions

```
void StepWorld(float deltaTime);
```

Advances the state of the simulation by one step. Put this in a loop to run the simulation over a period of time.

`deltaTime`: The amount of time that the simulation step will represent. Smaller values will give a finer resolution simulation.

```
bool getUsingRadians() const;
```

Returns true if the world is configured to use radian angles. False if using degrees.

```
glm::vec3 getGravity() const;
```

Find the global gravity of the physics world.

4.2.1 Collision Detection

See section 6 for more information about how collisions are tracked by this wrapper.

```
bool areColliding(int ID1, int ID2);
```

Returns true if objects with the collision IDs provided are in contact this step.

```
void setCollisionFunction(PWCollisionCallback cf);
```

Register a collision callback function with the Physics World. The callback will be called once for each collision that has happened this time step, during the call to `StepWorld(float deltaTime)`.

```
void setCollisionStartFunction(PWCollisionCallback
   csf);
```

Register a collision-start callback function function with the Physics World. The callback will be called once for each collision that has *begun* this time step, during the call to `StepWorld(float deltaTime)`.

```
void setCollisionEndFunction(PWCollisionCallback cef
   );
```

Register a collision-end callback function with the Physics World. The callback will be called once for each collision that has *ended* this time step, during the call to `StepWorld(float deltaTime)`.

4.2.2 Internally Used Functions

The following functions are used internally by the library, and you should have no need to call them.

```
void addPhysicsObject(PhysicsObject* object);
void addGravityObject(GravityObject* object);
void removePhysicsObject(PhysicsObject* object);
```

5 Physics Objects

All of the rigid objects that you add to your simulation will be some form of `PhysicsObject`. You cannot construct a base `PhysicsObject`, but all of the specific bodies will extend its functionality.

5.1 Public Member Functions

```
glm::vec3 getPosition() const;
```

Find the location of the object's origin in XYZ world coordinates.

```
glm::quat getRotationQuaternion() const;
```

Find the orientation of the object in a quaternion.

This, combined with the above `getPosition()`, are very useful for OpenGL representation. You can draw the object in your scene by constructing the model matrix like so:

```
glm::mat4 modelTransform;

glm::vec3 objectPosition = this->physicsObject->getPosition();
glm::quat objectRotation = this->physicsObject->getRotationQuaternion();

modelTransform = glm::translate(modelTransform, objectPosition);
modelTransform = glm::rotate(modelTransform, glm::angle(objectRotation), glm::axis(objectRotation));
```

```
void resetTransform();
```

Return the object to its initial position and orientation. Cancels all forces acting on the body, and resets its velocity.

```
void applyForce(glm::vec3 force);
```

Apply a force to the object in world coordinates. This is used for forces that should be applied continuously over a period of time, like a rocket engine.

```
void applyImpulse(glm::vec3 impulse);
```

Apply an impulse to the object in world coordinates. This is used for instantaneous forces that should be applied once, such as an explosion effect.

5.1.1 Collision Detection

```
void setCollisionID(int ID);
```

Set this object's collision ID. The ID must be non-zero in order for collisions to be tracked for this object. (Physics collision-response will occur regardless.)

```
int getCollisionID();
```

Returns the value of this object's collision ID. If it has not been manually set, the function will return 0.

5.1.2 Internally Used Functions

The following functions are used internally by the library, and you should have no need to call them.

```
void Init(btCollisionShape* cs, btDefaultMotionState
   * ms, btRigidBody* rb, PhysicsWorld* world)
btRigidBody* getRigidBody() const;
```

5.2 Physics Box

`PhysicsBoxes` can be used to represent cuboids of all dimensions, such as crates or buses.

```
//Lesson 1 constructors: absolute minimum to set up a test scene
PhysicsBox(PhysicsWorld* world);
//Lesson 2: Trying out different initial conditions
PhysicsBox(glm::vec3 initialPosition, glm::vec3 initialOrientationXYZ, PhysicsWorld* world);
PhysicsBox(glm::vec3 initialPosition, glm::vec4 initialOrientationQuat, PhysicsWorld* world);
//Full-feature constructors
PhysicsBox( bool dynamic, float depth, float height, float width, float mass,
            glm::vec3 initialPosition,
            glm::vec3 initialOrientationXYZ,
            PhysicsWorld* world);
PhysicsBox( bool dynamic, float depth, float height, float width, float mass,
            glm::vec3 initialPosition,
            glm::vec4 initialOrientationQuat,
            PhysicsWorld* world);
```

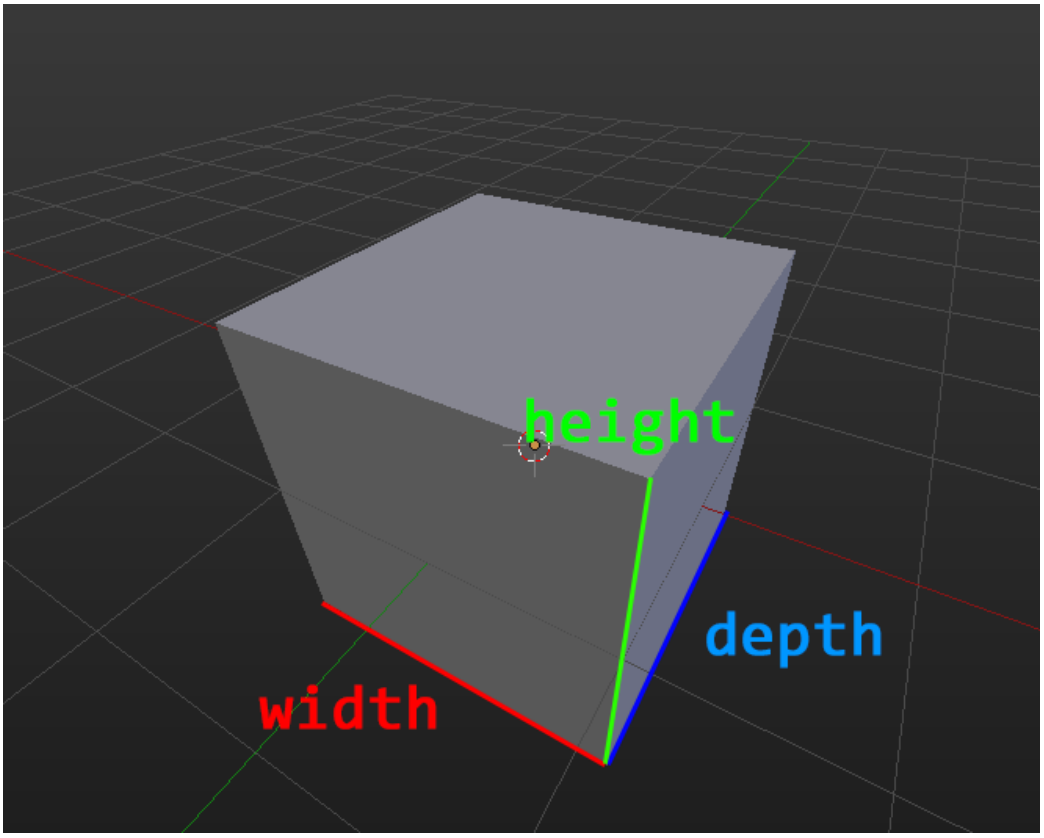
`dynamic`: If set to true, the object will move under forces. If set to false, the object will hang in space, unmoving. The latter is useful for defining a static environment to bounce other objects around in.

`initialPosition`: Set the initial position of the object's origin in world coordinates.

`initialOrientationXYZ`: Set the initial rotation of the object in roll-yaw-pitch angles. The units of the values you enter depends on the value of `useRadians` that you set when initialising the world. The order that the rotations are applied is:

- YAW (about Y, up)
- PITCH (about Z)
- ROLL (about X, front)

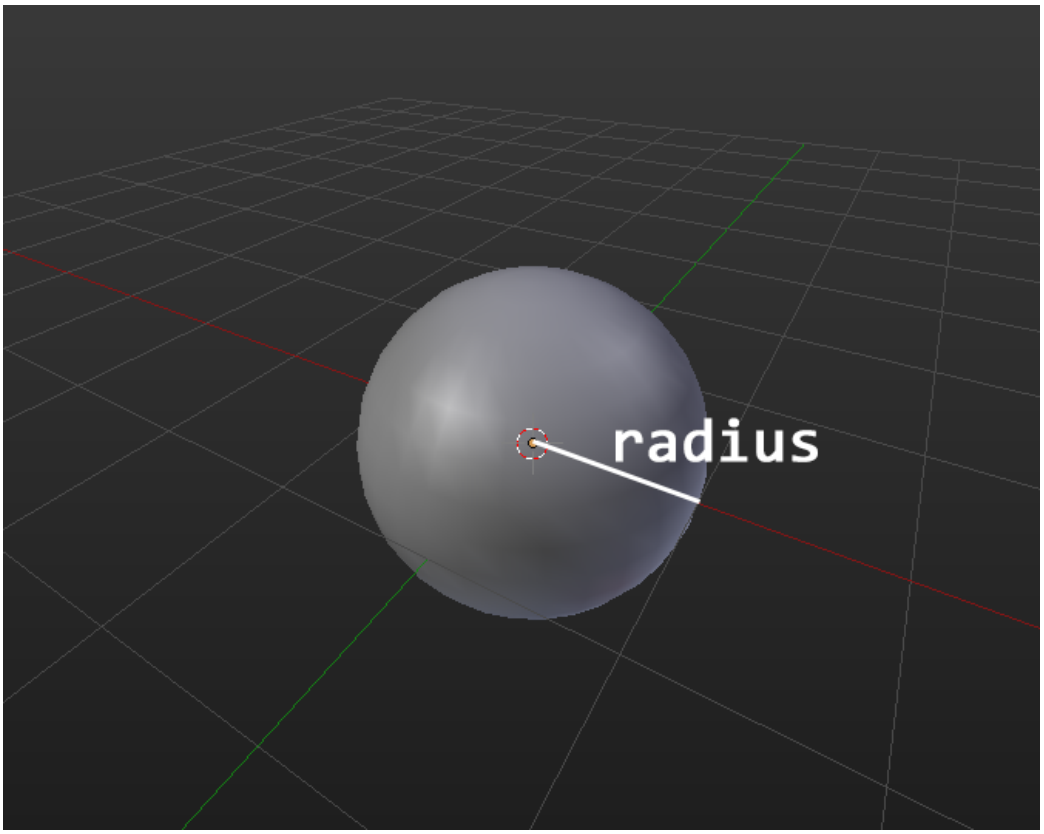
`initialOrientationQuat`: Set the initial rotation of the object by supplying an axis and an angle. The `vec4` type takes the arguments `x, y, z, w`. Use `x, y, z` to define the axis. `w` defines the angle in whatever units you set the world to use.


height
depth
width

5.3 Physics Ball

`PhysicsBalls` are used for spherical objects.

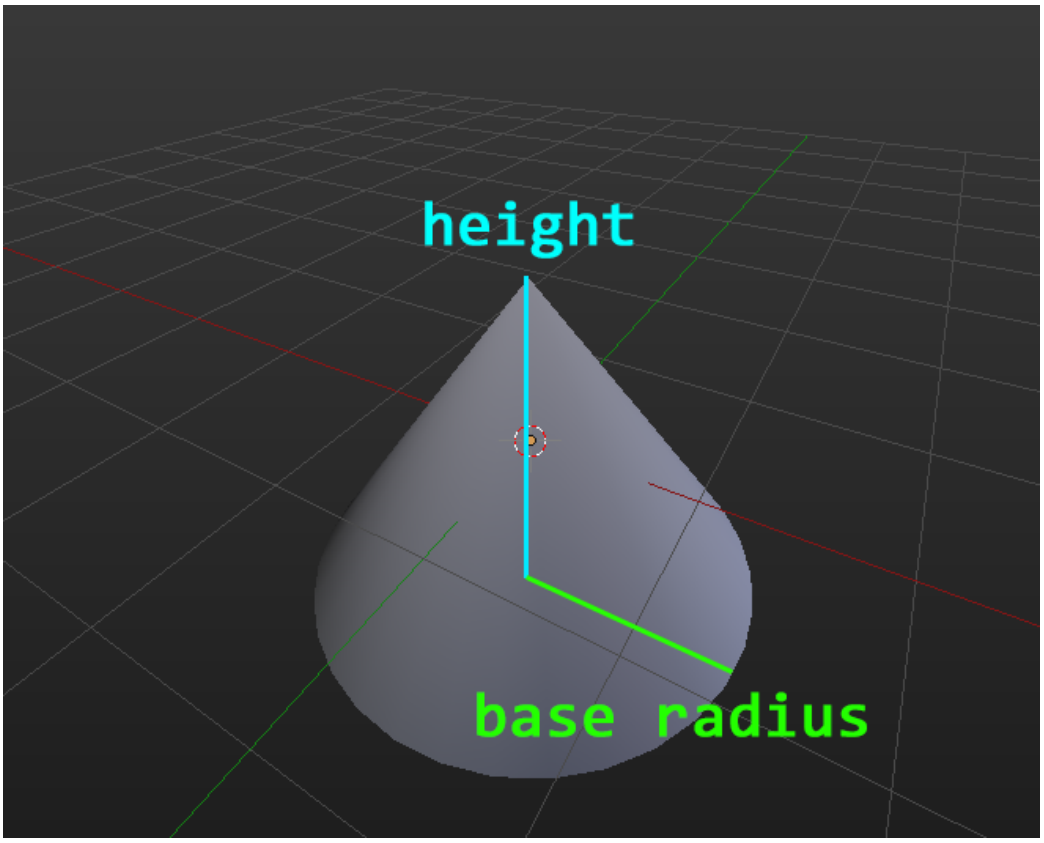
```
//Lesson 2: Trying out different initial conditions
PhysicsBall(glm::vec3 initialPosition, PhysicsWorld* world);
//Full-feature constructor
PhysicsBall(bool dynamic, float radius, float mass,
            glm::vec3 initialPosition,
            PhysicsWorld* world);
```



5.4 Physics Cone

PhysicsCones are defined with the point in the +Y direction.

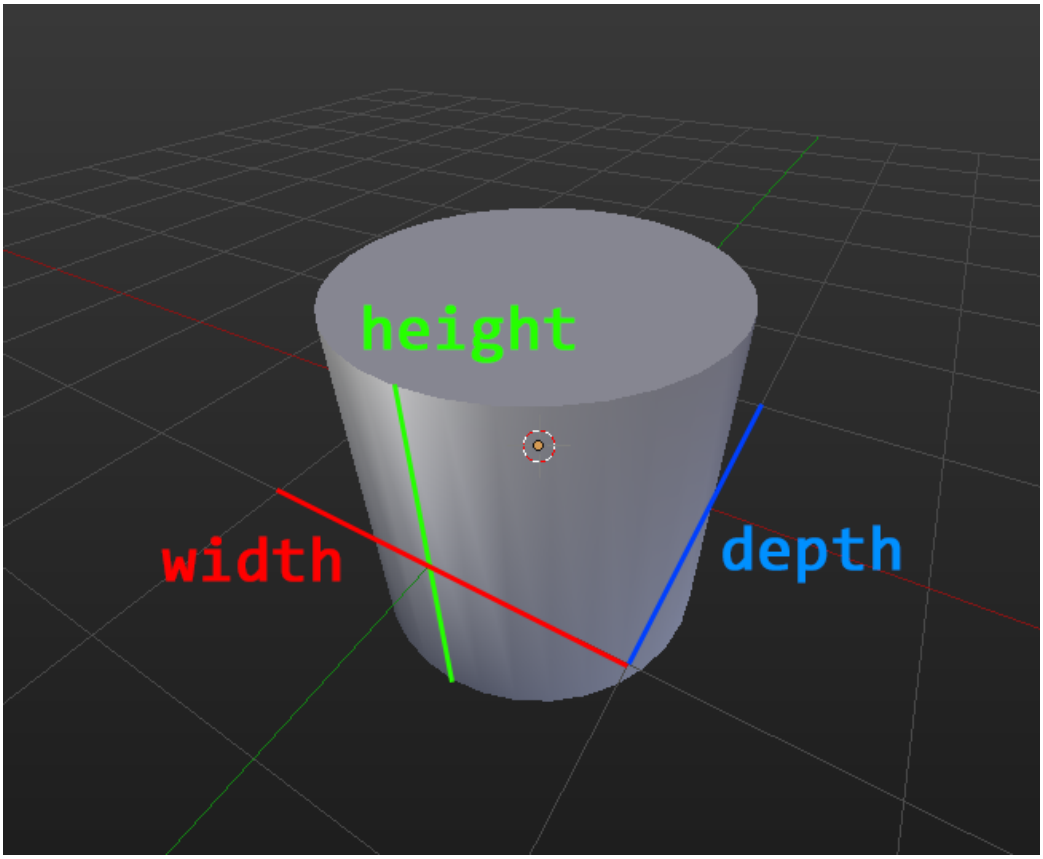
```
//Lesson 2: Trying out different initial conditions
PhysicsCone(glm::vec3 initialPosition, glm::vec3 initialOrientationXYZ, PhysicsWorld* world);
PhysicsCone(glm::vec3 initialPosition, glm::vec4 initialOrientationQuat, PhysicsWorld* world);
//Full-feature constructors
PhysicsCone(bool dynamic, float height, float baseRadius, float mass,
            glm::vec3 initialPosition,
            glm::vec3 initialOrientationXYZ,
            PhysicsWorld* world);
PhysicsCone(bool dynamic, float height, float baseRadius, float mass,
            glm::vec3 initialPosition,
            glm::vec4 initialOrientationQuat,
            PhysicsWorld* world);
```



5.5 Physics Cylinder

`PhysicsCylinders` are defined with the axis of rotation parallel to the global Y axis. They take width and depth rather than radius, because the cross section can be elliptical.

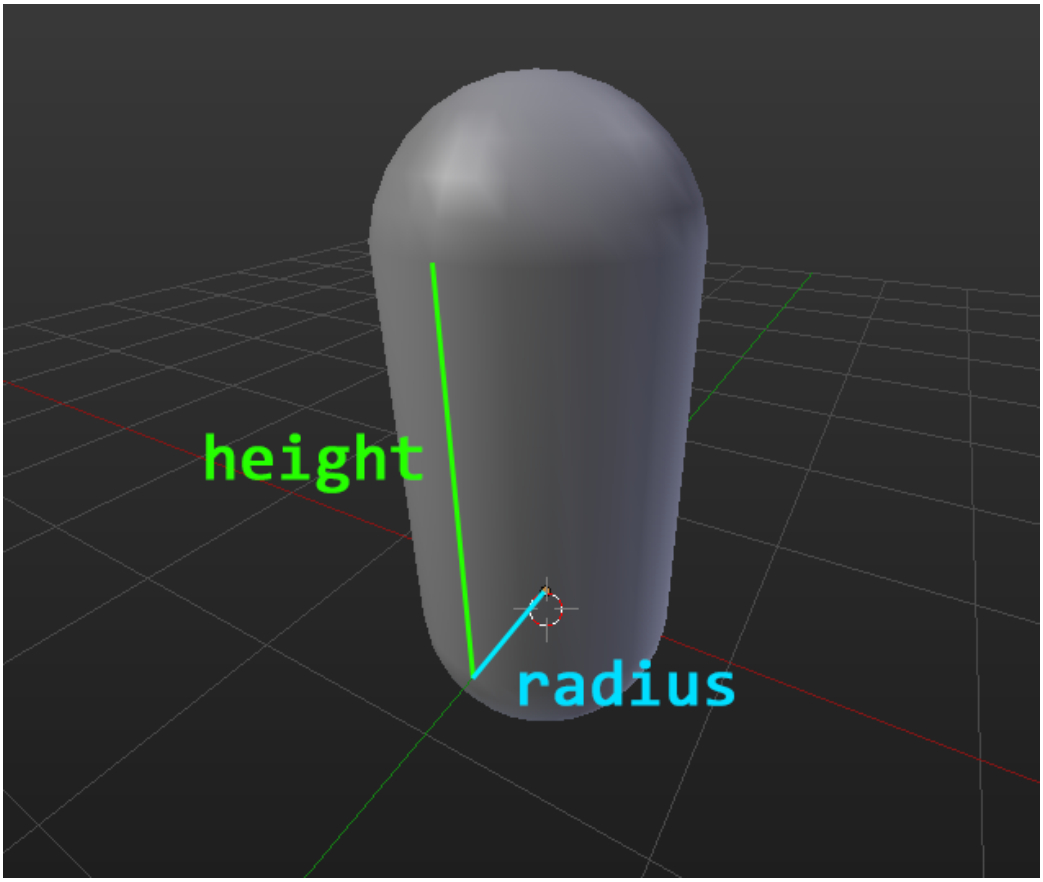
```
//Lesson 2: Trying out different initial conditions
PhysicsCylinder(glm::vec3 initialPosition, glm::vec3 initialOrientationXYZ, PhysicsWorld* world);
PhysicsCylinder(glm::vec3 initialPosition, glm::vec4 initialOrientationQuat, PhysicsWorld* world);
//Full-feature constructor
PhysicsCylinder(bool dynamic, float depth, float height, float width, float mass,
                glm::vec3 initialPosition,
                glm::vec3 initialOrientationXYZ,
                PhysicsWorld* world);
PhysicsCylinder(bool dynamic, float depth, float height, float width, float mass,
                glm::vec3 initialPosition,
                glm::vec4 initialOrientationQuat,
                PhysicsWorld* world);
```



5.6 Physics Capsule

`PhysicsCapsules` are defined with the axis of rotation parallel to the global Y axis. The height argument only accounts for the cylindrical section, so the total height of the shape will be $h + 2r$, where h = height and r = radius.

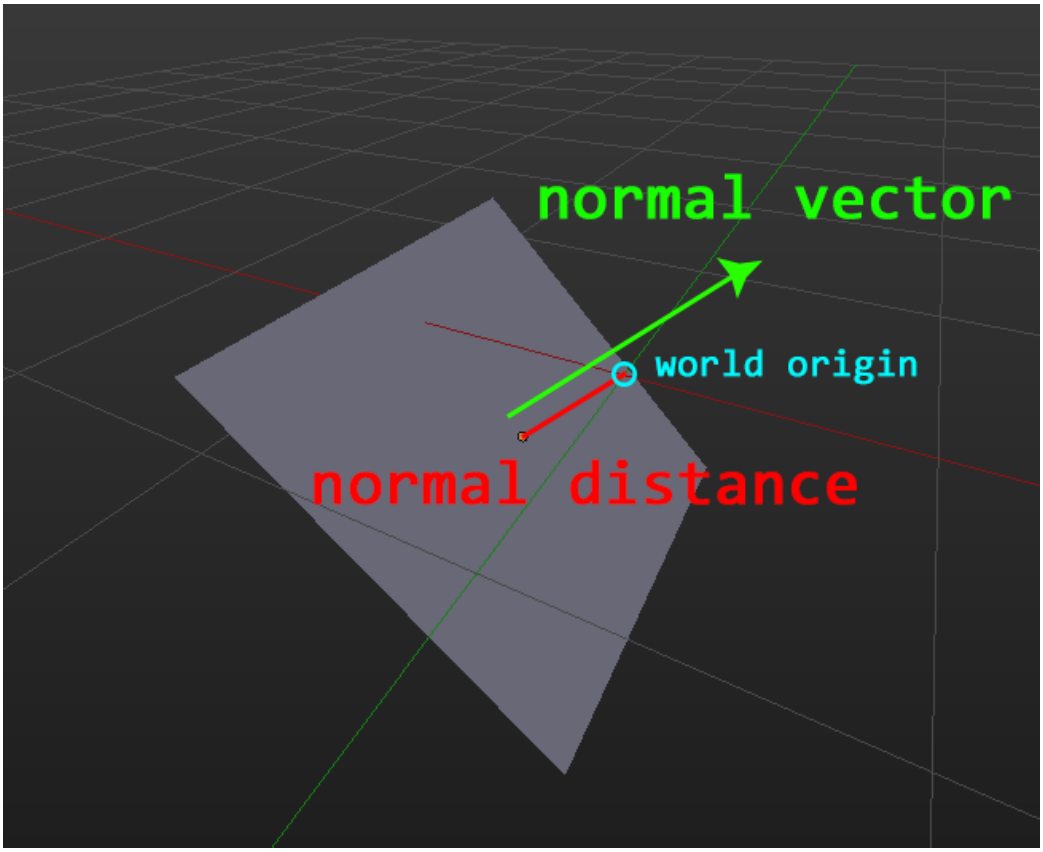
```
//Lesson 2: Trying out different initial conditions
PhysicsCapsule(glm::vec3 initialPosition, glm::vec3 initialOrientationXYZ, PhysicsWorld* world);
PhysicsCapsule(glm::vec3 initialPosition, glm::vec4 initialOrientationQuat, PhysicsWorld* world);
//Full-feature constructor
PhysicsCapsule( bool dynamic, float radius, float height, float mass,
                glm::vec3 initialPosition,
                glm::vec3 initialOrientationXYZ,
                PhysicsWorld* world);
PhysicsCapsule( bool dynamic, float radius, float height, float mass,
                glm::vec3 initialPosition,
                glm::vec4 initialOrientationQuat,
                PhysicsWorld* world);
```



5.7 Physics Plane

`PhysicsPlane` defines an infinite static plane that no dynamic object may pass. It takes a vector for the normal of the plane and a distance, which will define the shortest distance between a point on the plane and the world origin.

```
//Lesson 1: absolute minimum to set up a test scene
PhysicsPlane(PhysicsWorld* world);
//Full-feature constructor
PhysicsPlane(glm::vec3 normal, float normalDistance, PhysicsWorld* world);
```



5.8 Physics Heightmap

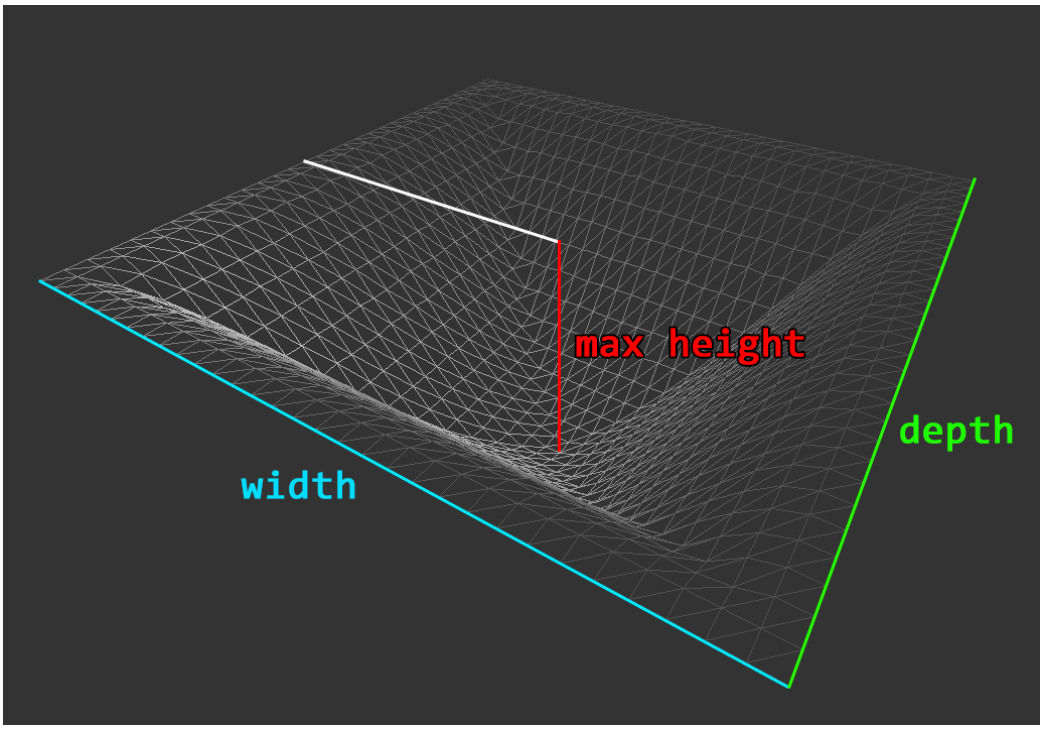
`PhysicsHeightmap` loads in the data from a **.bmp** file and converts it to a heightmap. This is a terrain with different elevations at each point, but no overhangs. Heightmaps cannot move by design; use them to define a world for other objects to interact with.

A white pixel in the .bmp corresponds to a point of maximum elevation, while a black pixel represents minimum elevation. Make sure that you give the constructor the relative path to the .bmp file. If the program cannot find the file at the provided location, it will instead load a flat heightmap (essentially a non-infinite plane).

```
PhysicsHeightmap(   std::string filename,
                    float depth, float width, float maxHeight,
                    glm::vec3 position,
                    PhysicsWorld* world);
```

The heightmap:

produces the terrain:



When choosing the resolution of your heightmap, be aware that if your physics objects span multiple points on the heightmap, performance will take a hit. Eg: for $1m^3$ objects, aim for around 1 point per metre in the heightmap.

```
HeightfieldData getHeightMapData() const;
```

Return the data for the heightmap in a special wrapper. You may wish to make use of this if you want to draw the heightmap in OpenGL or similar, as you can avoid converting a bitmap to a vector of floats a second time.

5.8.1 Heightfield Data

`HeightfieldData` wraps up the relevant information for an imported heightmap.

```
std::vector<float>* getData() const;
```

Get a pointer to the vector of floats that underpins the heightmap.

```
unsigned getWidth() const;
unsigned getDepth() const;
```

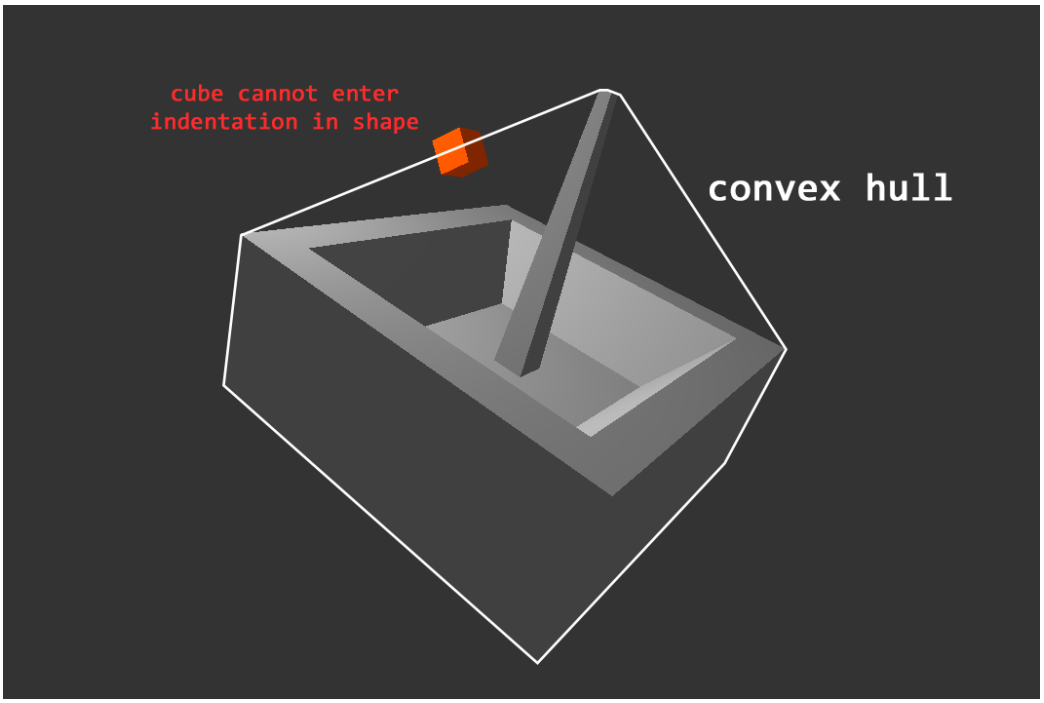
Useful for drawing the heightmap at the correct scale.

5.9 Convex Mesh

`PhysicsConvexMeshes` load an **.obj** file into the physics engine to use as a convex hull. The collision shape will be the smallest convex shape that fits around the vertices in the .obj. Convex hull objects can be dynamic.

You can generate an .obj file by exporting a mesh from most 3D modelling programs, such as Blender. Remember to check the option to triangulate the faces on export. Aim to have no more than a couple of hundred vertices in the mesh for performance reasons. If the program cannot find the .bmp file at the provided location, it will write an error to console and will exit.

```
PhysicsConvexMesh(  bool dynamic, std::string filename, float mass,
                    glm::vec3 initialPosition,
                    glm::vec3 initialOrientationXYZ,
                    PhysicsWorld* world);
PhysicsConvexMesh(  bool dynamic, std::string filename, float mass,
                    glm::vec3 initialPosition,
                    glm::vec4 initialOrientationQuat,
                    PhysicsWorld* world);
```

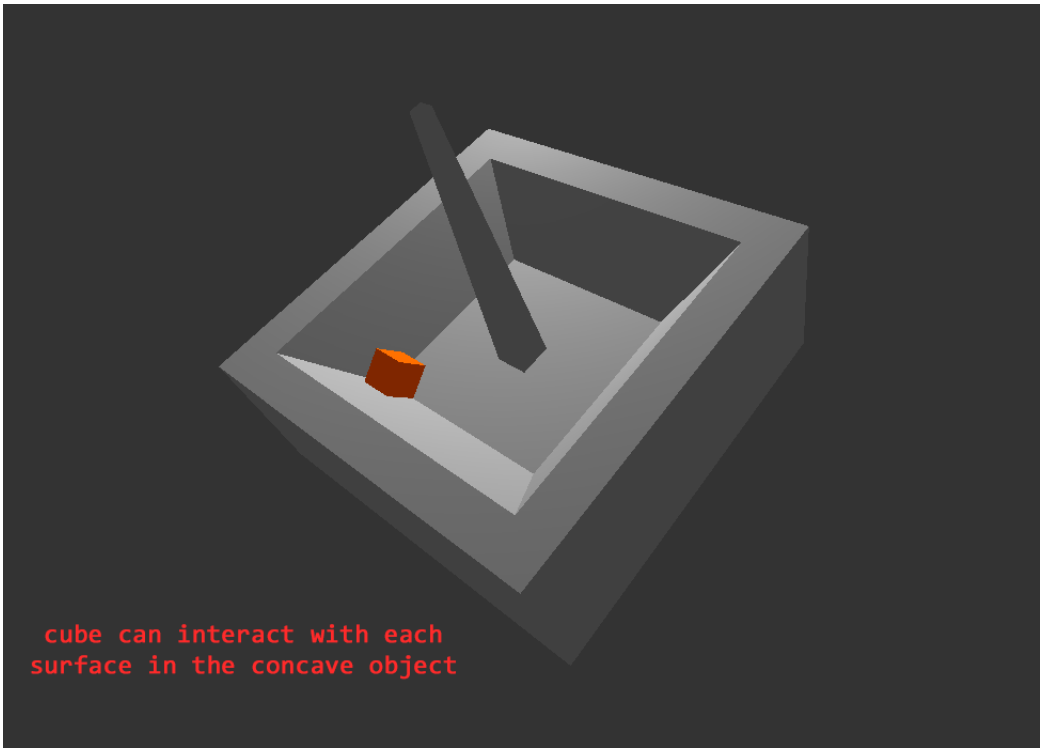


5.10 Concave Mesh

`PhysicsConcaveMeshes` also load an **.obj** file into the physics engine. Concave shapes are mush more complicated than even convex hulls, and will have an impact on performance if there are many vertices. Due to their complexity, concave meshes cannot move, so they are used to provide an environment for other objects to move around in.

If the program cannot find the .bmp file at the provided location, it will write an error to console and will exit.

```
PhysicsConcaveMesh( std::string filename,
                    glm::vec3 position,
                    glm::vec3 orientationXYZ,
                    PhysicsWorld* world);
PhysicsConcaveMesh( std::string filename,
                    glm::vec3 position,
                    glm::vec4 orientationQuat,
                    PhysicsWorld* world);
```



6 Collision Detection

The wrapper provides simple and easy-to-use collision detection functionality. When you move on to more complex engines with more powerful interfaces, the basics that you learn here should give you a head-start.

Firstly, we should identify what we mean by collision detection. The underlying engine handles physics collision-response automatically, so you do not have to worry about it. We are concerned with finding out when collisions have happened, and executing code as a result. For example, a projectile object may be destroyed once it collides with something else.

6.1 Collision IDs

To inform the wrapper that it should track collisions for an object, you must set the collision ID on that object to a non-zero value. All collision objects have the member function:

```
void setCollisionID(int ID);
```

Which you would use like this:

```
//Create a unit cube
PhysicsBox box(world);
//Set ID to track collisions
box.setCollisionID(1);
```

Both objects involved in a collision must have a non-zero ID, so if you want to get responses for when something hits the ground (eg: a heightmap), remember to set the collision ID for the ground as well!

Multiple objects can share a collision ID if they should behave the exact same way, but note: Only one collision for each ID pair is recorded each simulation step. So if a box (ID = 1) and a ball (ID = 1) both collide with a plane (ID = 2) on the same tick, you will lose one of the collisions. Generally, only use the same ID multiple times for stuff like sets of static objects forming a compound collision shape, where it's only possible for something else to collide with one part at a time.

6.2 Collision Check

The simplest form of collision checking just requires you to call a function on the world object that takes two IDs as arguments.

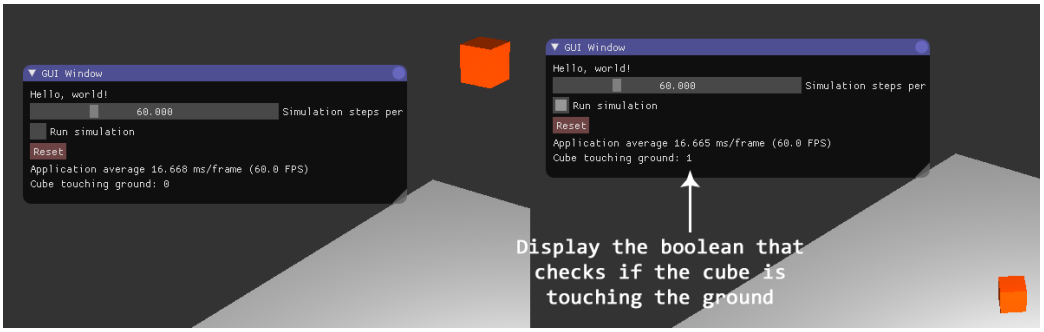
```
bool areColliding(int ID1, int ID2);
```

This function will return true if on the previous simulation step, objects with IDs *ID1* and *ID2* were colliding. You can call this during any iteration of your main loop as you would any other function:

```
PhysicsBox box(world);
box.setCollisionID(1);

PhysicsPlane defaultPlane(world);
defaultPlane.setCollisionID(2);

//...
//In the main loop somewhere
world->stepWorld(1 / 60.0f);
//Do something with this boolean
bool cubeGroundTouching = world->areColliding(1,2);
```



6.3 Collision Callbacks

The simple collision check is fine if you occasionally just want to know if a collision occurred, but what if you want to get some more information about the collision, or check for collisions starting and ending every frame anyway?

The `PhysicsWorld` member functions:

```
void setCollisionFunction(PWCollisionCallback cf);
void setCollisionStartFunction(PWCollisionCallback csf);
void setCollisionEndFunction(PWCollisionCallback cef);
```

are used to register user-defined callback functions with the Physics World. The functions that you provide will be called by the physics world whenever a collision occurs, starts, or ends (respectively). You functions will be called once for each collision that has occurred this step, so if on one frame:

- Box collides with ground
- Ball collides with ground

Your collision function will be called twice: one time for the Box-Ground collision and once for the Ball-Ground collision.

6.3.1 PWCollisionCallback

These registration functions take a rather strange argument: `PWCollisionCallback`. It's just an internal alias for a function that returns `void` and takes a `PhysicsCollision` as an argument. For example, you could write a function like this:

```
void callbackTest(PhysicsCollision collision)
{
    //Do something
}
```

And it would be fine to register it with your Physics World like this:

```
//In the main function, set up world...
world->setCollisionFunction(callbackTest);
```

And then, whenever a collision occurs, the world would call `callbackTest`, supplying information about the collision in the `PhysicsCollision` argument. But what is a `PhysicsCollision`?

6.3.2 PhysicsCollision

A `PhysicsCollision` is a wrapper class for information about the nature of a collision. Since you don't have different callback functions for each object in your world, you have to use the `PhysicsCollision` to work out what caused the callback.

```
//Check to see if a particular ID was involved in
   the collision
bool contains(int ID) const;
//Check to see if a particular pair of IDs were
   involved
bool contains(int ID1, int ID2) const;
```

These two functions are provided to help you determine the nature of the collision. If you only care if a certain ID has collided, then you can use the first function. If you are looking for a specific collision, you can supply both IDs:

```
void callbackTest(PhysicsCollision collision)
{
    //Collision between box and ground
    if (collision.contains(1, 2))
    {
        cubeGroundTouching = true;
    }
}
```

There is some additional information encapsulated in the collision wrapper:

```
glm::vec3 getPos1() const;
glm::vec3 getPos2() const;
```

These functions return the position of the origins of each of the colliding shapes in world coordinates. This may be useful to you if for example you wanted to instantiate some debris objects at the site of a collision.

```
glm::vec3 getNormal() const;
```

`getNormal()` returns the vector at normal to the collision. This is useful if you want to do something in the direction of the collision. For example: you could apply an impulse in this direction to cause your object to bounce off of surfaces at high speed:

```
void callbackStartTest(PhysicsCollision collision)
{
```

```
    //Collision between box and ground
    if (collision.contains(1, 2))
    {
        //Apply a large impulse in the direction of the collision
        box->applyImpulse(collision.getNormal() * 30.0f);
    }
}
```