Rumble3D User Manual

A 3D physics library

©2018 Matthias Gensheimer
All Rights Reserved.

Inhaltsverzeichnis

1 Introduction

Rumble3D is a 3D physics library. Its core features are:

- Particle simulation
- Rigid body simulation
- Force generators
- Collision detection and resolution

Due to the engine being made up of physic modules, it is fully extensible.

This manual will guide you through your first steps setting up the physics engine. It will explain the main parts of using the particle and rigid body engine aswell as the collision detection and resolution pipeline used in the rigid body engine.

First steps 2

2.1 PhysicsEngine

Each physic simulation needs one **PhysicsEngine**. It contains all **PhysicsEngineModules**, that are being used for the current simulation. Registering a module in the physics engine is as simple as:

```
1 PhysicsEngineModule* myModule = new MyModule();
2
3 PhysicsEngine engine;
4 engine.registerModule(myModule, "myModule");
```

2.2 PhysicsEngineModule

A **PhysicsEngineModule** is an abstract class, which allows the creation of new modules. A module contains an **IComputationInterface**, which can compute updates based on the data held by the module. This way The computation is separated from the data and therefore exchangeable. The function

```
1 /**
2 * \brief Get the computation interface of this module.
3 * \return The computation interface.
4 */
5 virtual IComputationInterface* getComputationInterface() const = 0;
```

is abstract, so that the module can directly communicate with the computation unit, which is derived from **IComputationInterface**. This prevents hacky casts inside implementations of **PhysicsEngineModule**.

2.3 Basic physics engine setup

In this section I guide you through the setup of a basic physics engine with particle and rigid body support.

First we will create a particle module (ParticleWorld) with a default computation interface (DefaultParticleEngineCI), which is already provided by the library.

```
r3::ParticleWorld* particleWorld;
particleWorld = new r3::ParticleWorld();

const auto ci = new r3::DefaultParticleEngineCI(1000, 0,
    particleWorld);
particleWorld->setComputationInterface(ci);
```

Then we will create a rigid body module (RigidBodyWorld) with a default computation interface (DefaultRigidBodyEngineCI), which is also provided by the library.

```
r3::RigidBodyWorld* rigidBodyWorld;
rigidBodyWorld = new r3::RigidBodyWorld();

const auto ci = new r3::DefaultRigidBodyEngineCI();
ci->setRigidBodyWorld(rigidBodyWorld);
rigidBodyWorld->setComputationInterface(ci);
```

Now that we created our modules, we need to register them in the physics engine:

```
PhysicsEngine physicsEngine;

physicsEngine.registerModule(particleWorld, "particle");
physicsEngine.registerModule(rigidBodyWorld, "rigid_body");
```

All that is left todo now, is to update our simulation by calling the **tick**-Funktion on the **PhysicsEngine**

```
physicsEngine.tick(timeDelta);
```

Note: If timeDelta is too large, strange effects can occur, like objects travelling through walls, increasing oscillations, excessive acceleration etc.

3 Collision detection

One major part of this physics engine is the detection and resolution of collisions between rigid bodies. First off you will get to know the collision detection system. The task of this system is to find all points in 3D space, where collisions occur. Additionally it will save properties of a collision at every found point. The collision points and their properties will be used later on, to resolve these collisions.

3.1 Broad phase

The broad phase takes in a number of rigid bodies and outputs a number of **CollisionPairs**. The task of the broad phase is to eliminate collision pairs, which do not collide. The generated **CollisionPairs** are only potential contacts, which means that there can be false positives. There should never be false negatives (e.g. eliminated collision pairs, that actually collide).

3.2 Intermediate phase

An intermediate phase takes in a number of **CollisionPairs** and outputs a (probably) smaller number of **CollisionPairs**. An intermediate phase behaves just like a broad phase, with the only difference being that multiple intermediate phases can be concatenated.

3.3 Narrow phase

A narrow phase takes a number of **CollisionPairs** as input and outputs a number of contacts. These contacts are then used in the collision resolution later on.

3.3.1 Collision Algorithms

	Box	**Sphere**	**Plane**
Box	Box-Box	Box-Sphere	Box-Plane
Sphere	Sphere-Box	Sphere-Sphere	Sphere-Plane
Plane	Plane-Box	Plane-Sphere	Plane-Plane

3.4 Pipeline

In figure 1 you can see the structure of the collision detection pipeline explained above. The **BroadphaseFilter** takes a set of rigid bodies as input and outputs a set of **CollisionPair**s.
It might be more apparent now, that an intermediate phase filter is needed on top of a broadphase filter, since both receive different inputs.

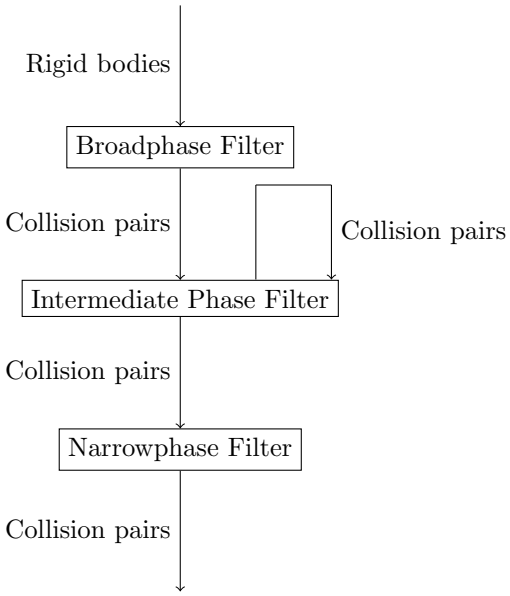

Abbildung 1: Collision detection pipeline

4 Computation Interface

This physics engine decouples data from computation. Each **PhysicsEngineModule** provides method, which returns an **IComputationInterface**. This interface consists of 4 methods:

- void onBegin()
- void step(real timeDelta)
- void integrate(real timeDelta)
- void onEnd()

In each iteration, the **PhysicsEngine** will go over every **PhysicsEngineModule**, get their computation interfaces (CI) and call

1. onBegin() on all CIs
2. then step() on all CIs
3. then integrate() on all CIs
4. and finally onEnd() on all CIs

This decoupling allows the usage of multiple computation interfaces for multiple task, such as:

- Very accurate, but computationally expensive offline calculations
- Real time simulations
- CPU and/or GPU simulations

5 Particle Engine

The particle engine simulates the movement of particles.

The main parts of the particle engine are:

- Particle
- ParticleWorld
- ParticleForceRegistry
- ParticleContactGeneratorRegistry
- ParticleEngineCI

5.1 Particle

Particles are dimensionless points in space and can therefore not collide with other particles. A particle has a multitude of properties:

Property	Range
position	$\mathbb{R}^3$
velocity	$\mathbb{R}^3$
acceleration	$\mathbb{R}^3$
forceAccumulator	$\mathbb{R}^3$
damping	$[0, 1]$
inverseMass	$\mathbb{R}_0^+$
isDead	$\{false, true\}$

Note: An inverseMass of 0 defines an object with infinite mass.

5.2 ParticleWorld

The **ParticleWorld** consists of

- A container of particles
- A force registry
- A contact generator regsitry
- A particle computation interface

Particles can be registered and unregister. Only particles, which are registered, will receive updates.

5.3 ParticleForceRegistry

The force registry contains Particle-ParticleForceGenerator entry-pairs. A force generator will add forces to the particle, which it is paired up with, in every update step. A force generator can be paired up with multiple particles and a particle can be influenced by multiple force generators.

5.4 ParticleContactGeneratorRegistry

The ParticleContactGeneratorRegistry works similarly to the ParticleForceRegistry. Except instead of having Particle-ParticleForceGenerator pairs, we now have Particle-ParticleContactGenerator pairs.

A **ParticleContactGenerator** doesn't add forces to particles, but creates **ParticleContact**s if necessary. I.e. two particles, which are connected by a **ParticleRod** should always have the same predefined distance. If that is not the case → generate a contact.

Resolving those contacts is done by the **ParticleContactResolver** and is part of the **ParticleEngineCI**.

5.5 ParticleEngineCI

The **ParticleEngineCI** (CI: computation interface) is responsible for transforming the data in a **ParticleWorld**. This means:

- Calculating particle forces by using the force registry
- Integrating these forces
- Detect contacts and resolve them by using the ParticleContactResolver

Since ParticleEngineCI is only an abstract class, there needs to be an implementation for it. There is already a default implementation of this computation interface called **DefaultParticleEngineCI**.

6 Rigid Body Engine

The rigid body engine simulates the movement of rigid bodies.

The main parts of the rigid body engine are:

- RigidBody
- RigidBodyWorld

6.1 RigidBody

A rigid body is defined by following properties:

Property	Range
position	$\mathbb{R}^3$
rotation	$\mathbb{H}$
velocity	$\mathbb{R}^3$
acceleration	$\mathbb{R}^3$
forceAccumulator	$\mathbb{R}^3$
torqueAccumulator	$\mathbb{R}^3$
linearDamping	$[0, 1]$
angularDamping	$[0, 1]$
inverseMass	$\mathbb{R}_0^+$
isDead	$\{false, true\}$
inverseInertiaTensor	$\mathbb{R}^{3\times3}$

6.2 RigidBodyWorld

The **RigidBodyWorld** consists of:

- A container of rigid bodies
- A **ForceRegistry**
- A **RigidBodyCI**

Rigid bodies can be registered and unregistered.

6.3 ForceRegistry

It holds a set of RigidBody-ForceGenerator pairs. The force generators are used to act forces onto the rigid bodies they are coupled with.

6.4 RigidBodyCI

The **RigidBodyCI** uses the data in a **RigidBodyWorld** to transform it. This means:

- Calculating forces by using the **ForceRegistry**
- Detecting contacts by using the **ContactDetector**
- Resolving those contacts by using the **CollisionResolver**
- Integrating the accumulated forces

The library already provides an implementation of such a RigidBodyCI, which is called **Default-RigidBodyCI**.

7 Collision resolution

The collision resolution stage follows the collision detection stage. In this stage contacts are being resolved (eliminated) by moving the affected rigid bodies accordingly.

7.1 Collision Resolver

Every **RigidBodyEngineCI** (CI = Computation Interface) contains one **CollisionResolver**. It holds a set of **ICollisionResolverFilter**. When there are collisions to resolve, it passes them through all previously registered filters. This pipeline can be seen in figure 2.

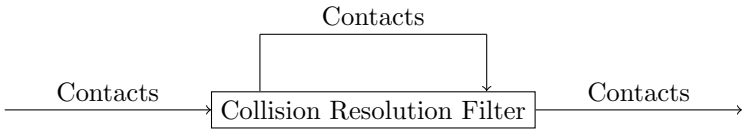

Abbildung 2: Collision resolution pipeline

7.2 Filter

A **ICollisionResolutionFilter** (as seen in listing 1) computes and applies changes to rigid bodies by using the contact data generated in the collision detection stage.

```
1  class R3D_DECLSPEC ICollisionResolutionFilter
2  {
3  public:
4          virtual ~ICollisionResolutionFilter();
5
6          /**
7          * \brief Resolve given contacts.
8          * \param collisionData The contacts to resolve.
9          * \param timeDelta The time step of the current physics
10         * update.
11         */
12         virtual void resolve(CollisionData& collisionData,
13         real timeDelta) = 0;
14
15 protected:
16         explicit ICollisionResolutionFilter();
17 };
```

Listing 1: Collision resolution filter

There already are two implementations of such a filter:

- VelocityResolver
- InterpenetrationResolver

The **InterpenetationResolver** changes the position and rotation of rigid bodies, so that they are no longer overlapping.

The **VelocityResolver** changes the velocity of colliding rigid bodies, according to their contact normal, physics material and fricition before the impact.