3D Rigid Body Physics System
RB - Initial Design

Graham Rigler

November 2018

1 Introduction

This proposes the initial design for a 3D rigid body physics system, titled 'RB'. It is designed to work independently of NGL, however it is to be shown as working with NGL, within the final product, as a means of demonstration of its worth.

The proof of concept (POC) provided with this report shows initial outlining of the UML diagram provided, as well as a test executable running many unit tests through the use of gtest. TravisCI has been made use of to allow for Linux compatability to be ensured while development is carried out in a Windows environment. Coveralls has also been used to ensure that test driven development (TDD) principals are kept to and that tests are representative of the system as a whole, if TDD is adhered to throughout then code coverage should not be below 100% throughout the project.

2 Design

As is described in further detail for the 'Computer Generated Imagery Techniques' report; this implementation makes use of the commonly found physics update loop that is comprised of the following stages:

- Integration
- Broad Phase Collision Detection
- Narrow Phase Collision Detection
- Collision Solving

To facilitate benchmark testing, as well as maintainability of the system, factories have been made use of for allowing algorithms and methods to be switched out at key stages. This is used for both the integration and collision solving stages, where two algorithms/methods are planned to be implemented and able to be hot-swapped at run-time through simple factory calls. The use of a registration macro can be seen in the POC that enables this to be completed simply in a manner that allows for typo-safe registration and calling of functions based upon the class name, relying upon stringifying the name of the given class and defining it as a global `static const string` within a embedded namespace of the relevant type (i.e. `RB::Integrators` or `RB::LCP`).

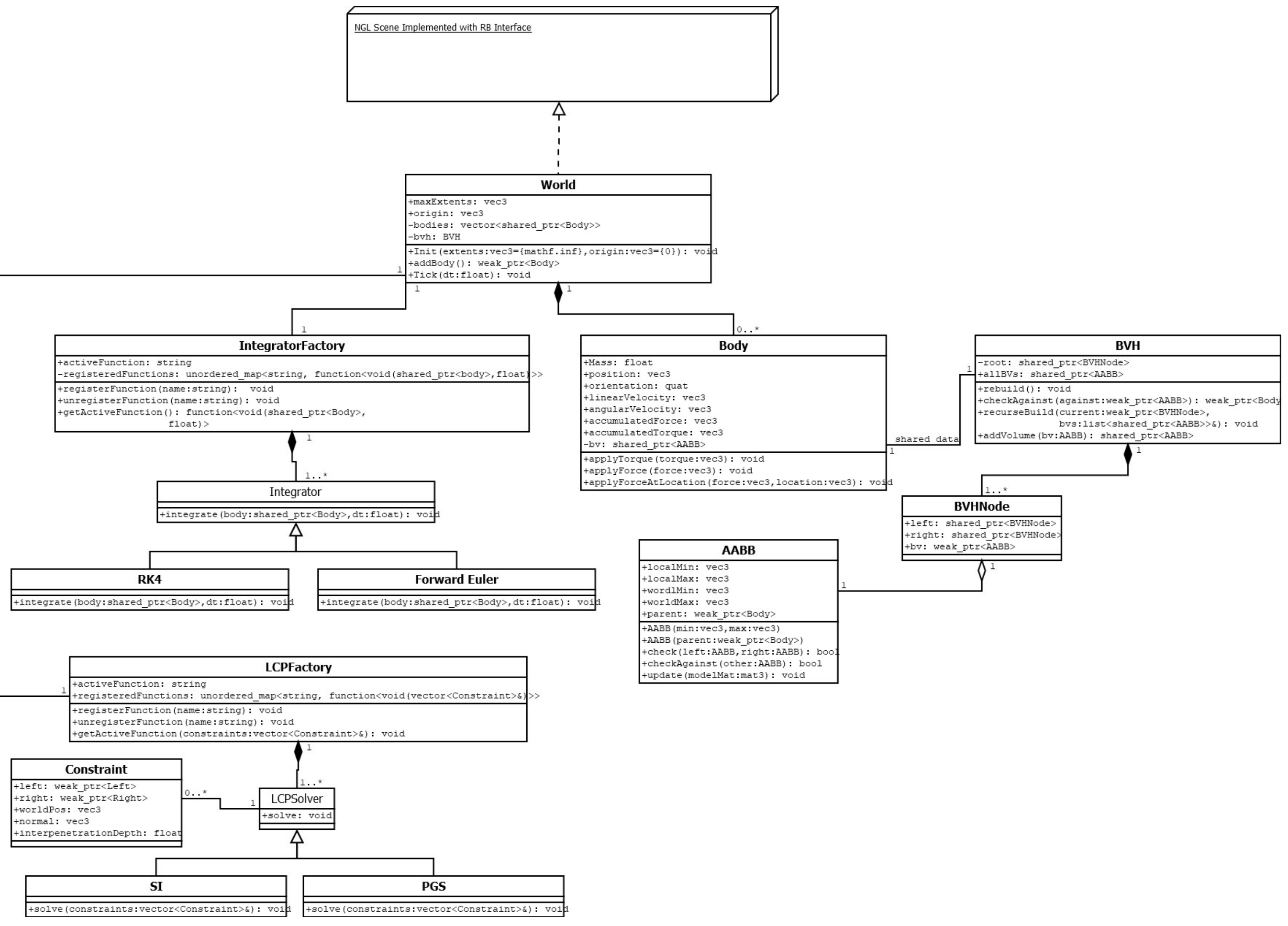
Research on existing implementations has led to axis-aligned bounding boxes (AABBs) to be used for the broad phase collision stage, alongside the use of a bounding volume hierarchy (BVH) to facilitate the need for large numbers of bodies to exist within a simulation at minimal computational impact.

2.1 Design Decisions of Note

- Lack of Factory base class
  While there is some shared code between the two factory classes, it was decided that no real benefit in development would be made from doing this due to the small number of factories and the syntactical complexity of using `std::function`. Direct implementations would be better used, albeit the slight breach of true object orientated development.
- Not making use of NGL maths and AABB capability
  Introducing dependencies to a library is not a lightly taken decision and as such it was decided that strongly coupling RB to NGL would provide no benefit in use. Instead, extra care is to be taken to ensure that RB will interface well with NGL and use it as a basis of its interface development.

3 Testing

Test driven development was experimented with during the development of the POC, as was mention in the introductory section. This is intended to be continued throughout development, unless a point is reached at which it is deemed unsuitable for this context. Unit testing has been shown extensively in what is a relatively basic POC for this type of project, with a number of tests existing for a largely non-functional code-base. As is expected with TDD, this is due to a large number of the existing test cases relating to simple movement of data between classes or such simple tasks as passing or failing known AABB test-cases.


NGL Scene Implemented with RB Interface
World
+maxExtents: vec3
+origin: vec3
-bodies: vector<shared_ptr<Body>>
-bvh: BVH
+Init(extents:vec3={mathf.inf},origin:vec3={0}): void
+addBody(): weak_ptr<Body>
+Tick(dt:float): void
1
1
1
IntegratorFactory
+activeFunction: string
-registeredFunctions: unordered_map<string, function<void(shared_ptr<body>,float)>>
+registerFunction(name:string): void
+unregisterFunction(name:string): void
+getActiveFunction(): function<void(shared_ptr<Body>,
float)>
1
1..*
Integrator
+integrate(body:shared_ptr<Body>,dt:float): void
RK4
+integrate(body:shared_ptr<Body>,dt:float): void
Forward Euler
+integrate(body:shared_ptr<Body>,dt:float): void
1
0..*
Body
+Mass: float
+position: vec3
+orientation: quat
+linearVelocity: vec3
+angularVelocity: vec3
+accumulatedForce: vec3
+accumulatedTorque: vec3
-bv: shared_ptr<AABB>
+applyTorque(torque:vec3): void
+applyForce(force:vec3): void
+applyForceAtLocation(force:vec3,location:vec3): void
shared data
1
1
BVH
-root: shared_ptr<BVHNode>
+allBVs: shared_ptr<AABB>
+rebuild(): void
+checkAgainst(against:weak_ptr<AABB>): weak_ptr<Body
+recurseBuild(current:weak_ptr<BVHNode>,
bvs:list<shared_ptr<AABB>>&): void
+addVolume(bv:AABB): shared_ptr<AABB>
1
1..*
BVHNode
+left: shared_ptr<BVHNode>
+right: shared_ptr<BVHNode>
+bv: weak_ptr<AABB>
1
1
AABB
+localMin: vec3
+localMax: vec3
+wordlMin: vec3
+worldMax: vec3
+parent: weak_ptr<Body>
+AABB(min:vec3,max:vec3)
+AABB(parent:weak_ptr<Body>)
+check(left:AABB,right:AABB): bool
+checkAgainst(other:AABB): bool
+update(modelMat:mat3): void
1
LCPFactory
+activeFunction: string
+registeredFunctions: unordered_map<string, function<void(vector<Constraint>&)>>
+registerFunction(name:string): void
+unregisterFunction(name:string): void
+getActiveFunction(constraints:vector<Constraint>&): void
1
1..*
Constraint
+left: weak_ptr<Left>
+right: weak_ptr<Right>
+worldPos: vec3
+normal: vec3
+interpenetrationDepth: float
0..*
1
LCPSolver
+solve: void
SI
+solve(constraints:vector<Constraint>&): void
PGS
+solve(constraints:vector<Constraint>&): void