p

Physics Claims Document

TNAH Engine v2

ICT398 Milestone 2 Group TNAH

Table of Contents

[Physics Data Structures 4](#_Toc87101245)

[External Libraries 5](#_Toc87101246)

[1. ReactPhysics3D 5](#_Toc87101247)

[Utilized Types 5](#_Toc87101248)

[1. CollisionBody 5](#_Toc87101249)

[setUserData Function 5](#_Toc87101250)

[getUserData Function 5](#_Toc87101251)

[2. CollisionShape 5](#_Toc87101252)

[BoxShape 5](#_Toc87101253)

[SphereShape 5](#_Toc87101254)

[CapsuleShape 5](#_Toc87101255)

[3. PhysicsWorld 5](#_Toc87101256)

[External Data Loading & Processing 6](#_Toc87101257)

[Serializer 6](#_Toc87101258)

[Code 6](#_Toc87101259)

[Deserializer 7](#_Toc87101260)

[Code 7](#_Toc87101261)

[The Rigidbody Class & The Rigidbody Component 10](#_Toc87101262)

[Rigidbody Component structure 10](#_Toc87101263)

[Rigidbody class 10](#_Toc87101264)

[Creating a Rigidbody 12](#_Toc87101265)

[1. Application 12](#_Toc87101266)

[2. Rigidbody Component 12](#_Toc87101267)

[3. Physics Engine 12](#_Toc87101268)

[4. Rigidbody 13](#_Toc87101269)

[Adding A Collider 13](#_Toc87101270)

[1. Application 13](#_Toc87101271)

[2. Rigidbody Component 14](#_Toc87101272)

[1. Box Shape 14](#_Toc87101273)

[2. Sphere Shape 14](#_Toc87101274)

[3. Capsule Shape 14](#_Toc87101275)

[3. Physics Engine 14](#_Toc87101276)

[1. Box Collider 15](#_Toc87101277)

[2. Sphere Collider 15](#_Toc87101278)

[3. Capsule Collider 15](#_Toc87101279)

[4. Collider 16](#_Toc87101280)

[Colliders 17](#_Toc87101281)

[The Collider Class 17](#_Toc87101282)

[Initialization 18](#_Toc87101283)

[Mass 18](#_Toc87101284)

[1. Box 18](#_Toc87101285)

[Equations 18](#_Toc87101286)

[Function 18](#_Toc87101287)

[2. Sphere 19](#_Toc87101288)

[Equations 19](#_Toc87101289)

[Function 19](#_Toc87101290)

[3. Capsule 19](#_Toc87101291)

[Equations 19](#_Toc87101292)

[Function 19](#_Toc87101293)

[Physics Engine and Physics Manager 21](#_Toc87101294)

[Physics Engine 21](#_Toc87101295)

[Class 21](#_Toc87101296)

[Fixed Update & Its Steps 21](#_Toc87101297)

[Step 0: Core 22](#_Toc87101298)

[Step 1: Updating Inertia Tensors 22](#_Toc87101299)

[Step 2: Processing Collisions 23](#_Toc87101300)

[Step 3: Processing Velocities 24](#_Toc87101301)

[Step 4: Processing Positions 25](#_Toc87101302)

[Step 5: Updating the Rigidbodies 25](#_Toc87101303)

[Step 6: Updating the sleep states 26](#_Toc87101304)

[Step 7: Resetting External Forces & Torque 26](#_Toc87101305)

[Debug Rendering 27](#_Toc87101306)

[Updating the Renderer 27](#_Toc87101307)

[Drawing Debug Lines 28](#_Toc87101308)

[Physics Manager 28](#_Toc87101309)

[Class 28](#_Toc87101310)

[Initialization 29](#_Toc87101311)

[Updating 30](#_Toc87101312)

[Collision Detection Engine 31](#_Toc87101313)

[Class 31](#_Toc87101314)

[Physics Events 31](#_Toc87101315)

[Class 31](#_Toc87101316)

[Physics Collision 31](#_Toc87101317)

[Class 31](#_Toc87101318)

[CollisionData Class 32](#_Toc87101319)

[Collision Resolution / Response 33](#_Toc87101320)

[Equations 33](#_Toc87101321)

[Code 33](#_Toc87101322)

[Group Declaration Sheet 35](#_Toc87101323)

`

# Physics Data Structures

At the core of the TNAH Physics Engine are base physics data structures designed to restrict and protect other physics types from interfering with one another. These structures define different C++ operators and functions restricting what other physics structures can be applied to or calculated from a given structure. Some structures also define simplified operators to return select data directly without accessing the variable within the structure.

|  |
| --- |
| struct BodyMass {  BodyMass()  { SetMass(); }   */\*\*  \* @fn void BodyMass::SetMass(float mass = 1.0f);  \*  \* @brief Sets the mass and updates the inverse mass.  \*/* void SetMass(float mass = 1.0f)  {  Mass = mass;  InverseMass = 1.0f / (Mass > 0.0f ? Mass : 1.0f); *// cant invert mass if its 0* }    glm::vec3 LocalCentreOfMass = {0,0,0};  glm::vec3 WorldCentreOfMass = {0,0,0};  float Mass = 1.0f;  float InverseMass = 1.0f; }; |

# External Libraries

## ReactPhysics3D

The structure and design of the Physics Engine is tightly integrated with the ReactPhysics3D 3rd party library for C++. Although the React library does provide robust friction impulse collision resolution for its own Rigidbody simulation, the TNAH Physics Engine, only leverages the libraries Collision Detection system and its event system to manage the collisions within a loaded scene.

### Utilized Types

Many classes and types defined within the React library are leveraged to process and manipulate physics within the scene.

#### CollisionBody

The CollisionBody is the base body class within the React Library. This type of body doesn’t respond to any type of physics within the library itself and must be positioned and orientated manually by the user, in the case of the TNAH Engine, this is done within our Physics Engine. The library reports any collisions on the body to the attached Event system, allowing the user to process the collision data their self.

##### setUserData Function

This function sets a void pointer to CollisionBody.

##### getUserData Function

This function returns the data set by the user as a void pointer or nullptr if it was never set.

#### CollisionShape

Collisions are detected based on shapes within the library. The CollisionShape is the base shape class within the library and can be attached to both CollisionBody and Rigidbody objects defined within the library.

##### BoxShape

The box shape is a basic cuboid shape using three extents for length, height and width. Although the library provides functions to retrieve half extents and documents them as such, the full extents of the shape are required to generate the shape object.

##### SphereShape

The sphere shape is a basic spherical shape using a provided radius.

##### CapsuleShape

The capsule shape is a cylindrical shape using a radius and height.

#### PhysicsWorld

A physics world within React defines the core simulation environment for the library. To simulate physics or detect collisions, a physics world object needs to be created and remain valid for the lifetime of the simulation.

# External Data Loading & Processing

### Serializer

The Serialization of the engine allows data to be loaded and saved to and from the disk allowing rigidbody objects and its colliders to be reused and reset without the original values being lost.

##### Code

std::string Serializer::*GenerateRigidBody*(*const* RigidBodyComponent& *rb*, *const* uint32\_t& *totalTabs*)  
{  
 std::stringstream ss;  
 ss << *GenerateTagOpen*("rigidbody", *totalTabs*);  
   
 *if*(*rb*.Body->GetType() == tnah::Physics::BodyType::Dynamic)  
 ss << *GenerateValueEntry*("type", "dynamic", *totalTabs*+1);  
 *else if*(*rb*.Body->GetType() == Physics::BodyType::Static)   
 ss << *GenerateValueEntry*("type", "static", *totalTabs*+1);  
 *else* ss << *GenerateValueEntry*("type", "kinematic", *totalTabs*+1);  
  
 ss << *GenerateValueEntry*("linearLock", *rb*.Body->m\_LinearRotationLock, *totalTabs*+1);  
 ss << *GenerateValueEntry*("angularLock", *rb*.Body->m\_AngularRotationLock, *totalTabs*+1);  
 ss << *GenerateValueEntry*("linearDamp", *rb*.Body->m\_LinearDampening, *totalTabs*+1);  
 ss << *GenerateValueEntry*("angularDamp", *rb*.Body->m\_AngularDampening, *totalTabs*+1);  
  
 *//No need to save the inverse mass, when loading, it can be found from the mass.* ss << *GenerateValueEntry*("mass", *rb*.Body->m\_BodyMass.Mass, *totalTabs* + 1);  
 ss << *GenerateValueEntry*("localCOM", *rb*.Body->m\_BodyMass.LocalCentreOfMass, *totalTabs* + 1);  
   
 ss << *GenerateValueEntry*("ignoreGravity", *rb*.Body->m\_IgnoreGravity, *totalTabs* + 1);  
 ss << *GenerateValueEntry*("sleepVelocity", *rb*.Body->m\_SleepVelocityThreshold, *totalTabs* + 1);  
 ss << *GenerateValueEntry*("sleepTime", *rb*.Body->m\_SleepTimeThreshold, *totalTabs* + 1);  
  
 *if***(***rb*.Body->HasColliders()**)** {  
 ss << *GenerateTagOpen*("colliders", *totalTabs* + 1);  
 *for*(*auto* col : *rb*.Body->m\_Colliders)  
 {  
 ss << *GenerateCollider*(col.second, *totalTabs* + 2);  
 }  
 ss << *GenerateTagClose*("colliders", *totalTabs* + 1);  
 }  
 ss << *GenerateTagClose*("rigidbody", *totalTabs*);  
 *return* ss.str();  
}

std::string Serializer::*GenerateCollider*(*const* Ref<Physics::Collider>& *col*, *const* uint32\_t& *totalTabs*)  
**{** std::stringstream ss;  
 *auto* t = *col*->GetType();  
 *switch*(t)  
 {  
 *case* Physics::Collider::Type::Box:  
 {  
 ss << *GenerateTagOpen*("boxCollider", *totalTabs*);  
   
 *auto* s = *static\_cast*<rp3d::BoxShape\*>(*col*->m\_Collider);  
 glm::vec3 ext = Math::FromRp3dVec3(s->getHalfExtents()) \* 2.0f;  
 ss << *GenerateValueEntry*("extents", ext, *totalTabs* + 1);  
 ss << *GenerateValueEntry*("localPosition", *col*->m\_LocalPosition, *totalTabs* + 1);  
 ss << *GenerateValueEntry*("localOrientation", *col*->m\_LocalOrientation, *totalTabs* + 1);  
 ss << *GenerateValueEntry*("density", *col*->m\_Density, *totalTabs* + 1);  
 ss << *GenerateValueEntry*("volume", *col*->m\_Volume, *totalTabs* + 1);  
   
 ss << *GenerateTagClose*("boxCollider", *totalTabs*);  
 *break*;  
 }  
 *case* Physics::Collider::Type::Sphere:  
 {  
 ss << *GenerateTagOpen*("sphereCollider", *totalTabs*);  
 *auto* s = *static\_cast*<rp3d::SphereShape\*>(*col*->m\_Collider);  
 ss << *GenerateValueEntry*("radius", s->getRadius(), *totalTabs* + 1);  
 ss << *GenerateValueEntry*("localPosition", *col*->m\_LocalPosition, *totalTabs* + 1);  
 ss << *GenerateValueEntry*("localOrientation", *col*->m\_LocalOrientation, *totalTabs* + 1);  
 ss << *GenerateValueEntry*("density", *col*->m\_Density, *totalTabs* + 1);  
 ss << *GenerateValueEntry*("volume", *col*->m\_Volume, *totalTabs* + 1);  
 ss << *GenerateTagClose*("sphereCollider", *totalTabs*);  
 *break*;  
 }  
 *case* Physics::Collider::Type::Capsule:  
 {  
 ss << *GenerateTagOpen*("capsuleCollider", *totalTabs*);  
 *auto* s = *static\_cast*<rp3d::CapsuleShape\*>(*col*->m\_Collider);  
 ss << *GenerateValueEntry*("radius", s->getRadius(), *totalTabs* + 1);  
 ss << *GenerateValueEntry*("Height", s->getHeight(), *totalTabs* + 1);  
 ss << *GenerateValueEntry*("localPosition", *col*->m\_LocalPosition, *totalTabs* + 1);  
 ss << *GenerateValueEntry*("localOrientation", *col*->m\_LocalOrientation, *totalTabs* + 1);  
 ss << *GenerateValueEntry*("density", *col*->m\_Density, *totalTabs* + 1);  
 ss << *GenerateValueEntry*("volume", *col*->m\_Volume, *totalTabs* + 1);  
 ss << *GenerateTagClose*("capsuleCollider", *totalTabs*);  
 *break*;  
 }  
 *default*: *break*;  
 }  
  
 *return* ss.str();  
   
}

### Deserializer

The deserializer works in a similar way to the serializer and loads the required data from the scene file from the disk and creates the required Rigidbody and collider objects from the provided data.

##### Code

*void* Serializer::*GetRigidBodyFromFile*(*const* std::string& *fileContents*,  
 std::pair<size\_t, size\_t> *componentTagPositions*, GameObject& *gameObject*)  
**{** *auto* type = *GetStringValueFromFile*("type", *fileContents*, *componentTagPositions*);  
 *auto* rb = *gameObject*.AddComponent<RigidBodyComponent>(*gameObject*);  
 *if*(type == "static")  
 {  
 rb.Body->SetType(Physics::BodyType::Static);  
 }  
 *else if*(type == "kinematic")  
 {  
 rb.Body->SetType(Physics::BodyType::Kinematic);  
 }  
 rb.Body->m\_LinearRotationLock = *GetVec3FromFile*("linearLock", *fileContents*, *componentTagPositions*);  
 rb.Body->m\_AngularRotationLock = *GetVec3FromFile*("angularLock", *fileContents*, *componentTagPositions*);  
 rb.Body->m\_LinearDampening = *GetFloatValueFromFile*("linearDamp", *fileContents*, *componentTagPositions*);  
 rb.Body->m\_AngularDampening = *GetFloatValueFromFile*("angularDamp", *fileContents*, *componentTagPositions*);  
   
 rb.Body->m\_BodyMass.SetMass(*GetFloatValueFromFile*("mass", *fileContents*, *componentTagPositions*));  
 rb.Body->m\_BodyMass.LocalCentreOfMass = *GetVec3FromFile*("localCOM", *fileContents*, *componentTagPositions*);  
   
 rb.Body->m\_IgnoreGravity = *GetBoolValueFromFile*("ignoreGravity", *fileContents*, *componentTagPositions*);  
 rb.Body->m\_SleepVelocityThreshold = *GetFloatValueFromFile*("sleepVelocity", *fileContents*, *componentTagPositions*);  
 rb.Body->m\_SleepTimeThreshold = *GetFloatValueFromFile*("sleepTime", *fileContents*, *componentTagPositions*);  
  
 *auto* t = *FindTags*("colliders", *fileContents*, *componentTagPositions*.first, *componentTagPositions*.second);  
 *if*(*CheckTags*(t))  
 {  
 *//This rb has colliders  
 GetColliderFromFile*(*fileContents*, t, rb);  
 }  
}

*bool* Serializer::*GetColliderFromFile*(*const* std::string& *fileContents*, std::pair<size\_t, size\_t> *componentTagPositions*, RigidBodyComponent& *rb*)  
**{** *auto* t = *FindTags*("boxCollider", *fileContents*, *componentTagPositions*.first, *componentTagPositions*.second);  
 *while*(*CheckTags*(t))  
 {  
 *auto* ext = *GetVec3FromFile*("extents", *fileContents*, t);  
 *auto* pos = *GetVec3FromFile*("localPosition", *fileContents*, t);  
 *auto* ori = *GetVec3FromFile*("localOrientation", *fileContents*, t);  
 *auto* dense = *GetFloatValueFromFile*("density", *fileContents*, t);  
 *auto* vol = *GetFloatValueFromFile*("volume", *fileContents*, t);  
 *rb*.AddCollider(ext, pos, ori, dense, vol);  
 t = *FindTags*("boxCollider", *fileContents*, *componentTagPositions*.first, *componentTagPositions*.second);  
 }  
  
 t = *FindTags*("sphereCollider", *fileContents*, *componentTagPositions*.first, *componentTagPositions*.second);  
 *while*(*CheckTags*(t))  
 {  
 *auto* rad = *GetFloatValueFromFile*("radius", *fileContents*, t);  
 *auto* pos = *GetVec3FromFile*("localPosition", *fileContents*, t);  
 *auto* ori = *GetVec3FromFile*("localOrientation", *fileContents*, t);  
 *auto* dense = *GetFloatValueFromFile*("density", *fileContents*, t);  
 *auto* vol = *GetFloatValueFromFile*("volume", *fileContents*, t);  
 *rb*.AddCollider(rad, pos, ori, dense, vol);  
 t = *FindTags*("sphereCollider", *fileContents*, *componentTagPositions*.first, *componentTagPositions*.second);  
 }  
  
 t = *FindTags*("capsuleCollider", *fileContents*, *componentTagPositions*.first, *componentTagPositions*.second);  
 *while*(*CheckTags*(t))  
 {  
 *auto* rad = *GetFloatValueFromFile*("radius", *fileContents*, t);  
 *auto* hei = *GetFloatValueFromFile*("height", *fileContents*, t);  
 *auto* pos = *GetVec3FromFile*("localPosition", *fileContents*, t);  
 *auto* ori = *GetVec3FromFile*("localOrientation", *fileContents*, t);  
 *auto* dense = *GetFloatValueFromFile*("density", *fileContents*, t);  
 *auto* vol = *GetFloatValueFromFile*("volume", *fileContents*, t);  
 *rb*.AddCollider(rad, hei, pos, ori, dense, vol);  
 t = *FindTags*("capsuleCollider", *fileContents*, *componentTagPositions*.first, *componentTagPositions*.second);  
 }  
  
 *if*(*rb*.Body->HasColliders())  
 *return true*;  
   
 *return false*;  
}

# The Rigidbody Class & The Rigidbody Component

Rigidbody objects within the TNAH Engine are broken up into two classes, the Rigidbody class and the Rigidbody Component structure. The Rigidbody component is a structure directly used within the engine’s entity component system and allows for quick access to functions to manipulate the Rigidbody without accessing the Rigidbody directly.

#### Rigidbody Component structure

|  |
| --- |
| struct RigidBodyComponent  {  Ref<Physics::RigidBody> Body = nullptr;RigidBodyComponent(GameObject& gameObject,  const BodyType& type = BodyType::Dynamic); void OnUpdate(TransformComponent& transform);void ApplyForce(const glm::vec3& direction, const glm::vec3& force);void ApplyTorque(const glm::vec3& direction,  const glm::vec3& torque); Ref<Physics::Collider> AddCollider(const glm::vec3& boxSize);Ref<Physics::Collider> AddCollider(const float& sphereRadius); Ref<Physics::Collider> AddCollider(const float& capsuleRadius,  const float& capsuleHeight);   }; |

#### Rigidbody class

Defining a range of functions and a variety of data required to process and manage physics calculations on a body itself. The structure of the class is designed to only allow public access to functions relevant to the user or application and privatizes the remaining functions and data internally, only allowing other internal physics classes to access privatized information via friending the classes.

Rigidbody functions:

\*Note, some helper/irrelevant functions and properties of this class have been removed from this snippet.

RigidBody();  
 RigidBody(TransformComponent& transform, BodyMass mass,

BodyType type = BodyType::Dynamic);  
  
 static Ref<RigidBody> Create(TransformComponent& transform, BodyMass mass,

BodyType type = BodyType::Dynamic);  
 void OnUpdate(TransformComponent& transform);  
 void AddForce(const glm::vec3& force);  
 void AddTorque(const glm::vec3& torque);  
 void AddCollider(Ref<Collider> collider);  
 std::unordered\_map<uint32\_t, Ref<Collider>> GetColliders();  
 bool HasColliders() const;void UpdateBodyProperties();  
 void SetType(const BodyType& type);

BodyType GetType() const;  
 InertiaTensor GetInertiaTensor() const;

void ResetValues();  
 bool& IgnoreGravity();  
 bool IsSleeping() const;  
 glm::vec3 CalculateLocalInertiaTensor();  
  
private:  
  
 glm::vec3 CalculateCentreOfMass();  
 void SetID(const uint32\_t id);

Rigidbody Properties:

private:BodyType m\_BodyType;  
BodyMass m\_BodyMass;  
LinearVelocity m\_LinearVelocity;  
AngularVelocity m\_AngularVelocity;  
LinearVelocity m\_ConstrainedLinearVelocity;  
AngularVelocity m\_ConstrainedAngularVelocity;  
Force m\_Force;  
Torque m\_Torque;  
LinearDampening m\_LinearDampening;  
AngularDampening m\_AngularDampening;  
InertiaTensor m\_InertiaTensor;  
 std::unordered\_map<uint32\_t, Ref<Collider>> m\_Colliders;  
  
 uint32\_t m\_TotalColliders = 0;  
uint32\_t m\_ID = 0;  
glm::vec3 m\_LinearRotationLock = {1,1,1};  
glm::vec3 m\_AngularRotationLock = {1,1,1};  
  
 glm::vec3 m\_Position = {0,0,0};

glm::quat m\_Orientation = {0,0,0,0};  
bool m\_IsSleeping = false;  
  
 float m\_SleepVelocityThreshold = 0.2f;  
  
 float m\_SleepTimeThreshold = 1.0f;  
  
 float m\_SleepTimeAccumulator = 0.0f;

bool m\_IgnoreGravity = false;rp3d::CollisionBody\* m\_CollisionBody = nullptr;

## Creating a Rigidbody

Creating a [Rigidbody](#_Rigidbody_class) within the engine is made simple for the developer due to the broken-down nature of the system and design of the Rigidbody class as well as its component class. There are four stages the engine and application take to create a Rigidbody object and intern a [Rigidbody component](#_Rigidbody_Component_structure).

### Application

Firstly, a request is made by the developer within their application. this is an example of how the external application would create the Rigidbody.

m\_Ball = m\_ActiveScene->CreateGameObject("Ball");  
m\_Ball.AddComponent<tnah::MeshComponent>("assets/meshes/sphere.fbx");  
m\_Ball.Transform().Position = {3.5f, 8.0f, -10.0f};  
m\_PhysicsSimStartPosition = m\_Ball.Transform().Position;  
auto& rb3 = m\_Ball.AddComponent<tnah::RigidBodyComponent>(m\_Ball);  
auto col = rb3.AddCollider({1.0f});

### Rigidbody Component

Secondly the Rigidbody component is created using provided parameters to link back to the attached game object. The component constructor does some sanity checks to ensure the physics world has been created before preceding to process creating the Rigidbody through the PhysicsEngine class. Then setting the type of Rigidbody and its user data. User data is a void pointer to any data the user would want to connect with the Rigidbody. For collision resolution, the gameobject is used to be able to retrieve it when processing the collision. The [setUserData()](#_setUserData_Function) function is defined by [ReactPhysics3D](#_ReactPhysics3D) and isn’t exposed to the developer.

RigidBodyComponent::RigidBodyComponent(GameObject& gameObject, const Physics::BodyType& type)  
{  
 if(Physics::PhysicsEngine::IsActive())  
 {  
 Body = Physics::PhysicsEngine::CreateRigidbody(gameObject);  
 Body->SetType(type);  
 Body->GetCollisionBody()->setUserData(&gameObject);  
 }  
}

### Physics Engine

Thirdly, the Physics Engine constructs the Rigidbody by providing the attached gameobjects transform component and default mass for the body. Once the engine Rigidbody has been created, the React [CollisionBody](#_CollisionBody) needs to be updated from the engines transform component to ensure the React physics world and the engines scene are equal regarding position and orientation of each physics object. Finally, the Physics Engine adds the Rigidbody reference to a unordered map with a identifier and increments its counter. The unordered map is used to access a given Rigidbody with a provided ID.

Ref<RigidBody> PhysicsEngine::CreateRigidbody(GameObject& gameObject)  
{  
 if(m\_PhysicsManager->m\_Active)  
 {  
 auto& transform = gameObject.Transform();  
 auto rb = RigidBody::Create(gameObject.Transform(), {});  
 transform.GetQuaternion() = glm::quat(transform.Rotation);  
 rp3d::Transform reactTransform;  
 reactTransform.setPosition(Math::ToRp3dVec3(transform.Position));  
 reactTransform.setOrientation(Math::ToRp3dQuat(transform.GetQuaternion()));  
 rb->SetCollisionBody(

m\_PhysicsManager->m\_PhysicsWorld->createCollisionBody(reactTransform)

);  
 rb->SetID(m\_PhysicsManager->m\_TotalRigidbodies);

m\_PhysicsManager->m\_Rigidbodies[rb->GetID()] = rb;   
 m\_PhysicsManager->m\_TotalRigidbodies++;  
   
 return rb;  
 }  
 return nullptr;  
}

### Rigidbody

Finally, within the Rigidbody constructor, basic setup is performed by setting the body mass, type, internal position and orientation. The core structure of a Rigidbody with no colliders is quite basic in comparison to when a collider is attached and allows the core Rigidbody to be lightweight.

RigidBody::RigidBody(TransformComponent& transform, BodyMass mass, BodyType type)  
 :m\_BodyType(type), m\_BodyMass(mass)  
{  
 m\_Position = transform.Position;  
 m\_Orientation = transform.GetQuaternion();  
}

## Adding A Collider

Adding a collider to a Rigidbody is controlled through its component and uses simple float and vec3 values to construct a given box, sphere or capsule collider shape for the body. When a collider is added to a Rigidbody, the mass and inertia tensor data for the body is recalculated allowing the developer to add as many colliders as they like to the Rigidbody with different positions and orientations. Collider positions and orientations are always relative to the parent body and are used to calculate required data, for instance, the inertia tensor of the body, among other uses within the physics processing for the body. More about [Colliders!](#_Colliders)

### Application

Adding a collider within a application is designed to be as simple as possible by only utilizing a single AddCollider() function on the Rigidbody component for a given gameobject. Examples of adding different collider shapes:

rb.AddCollider({4.0f, 4.0f, 0.5f}); *// Box*

rb.AddCollider(1.0f); *// Sphere*

rb.AddCollider(0.01f, 2.0f); *// Capsule*

The values provided to the AddCollider() function determine what type of collider the Rigidbody component will add to the body itself. Providing a glm vec3 with the full extents for a box will generate a box collider. Providing a single value as a radius will generate a sphere collider and finally, providing two values as a height and radius of a capsule will generate a capsule collider.

### Rigidbody Component

As mentioned previously, the Rigidbody component uses three different overloaded versions of the AddCollider() function to calculate and add a collider to its contained Rigidbody object. These overloaded functions act as a link between the component and the Physics engine also providing a reference back as a return value if the developer requires to further manipulate the colliders properties after being generated. Colliders can also be retrieved as a unordered map from the Rigidbody directly by calling the GetColliders() function from the Body property of the component.

#### Box Shape

Ref<Physics::Collider> RigidBodyComponent::AddCollider(const glm::vec3& boxSize)  
{  
 if(Physics::PhysicsEngine::IsActive() && Body)  
 {  
 return Physics::PhysicsEngine::CreateBoxCollider(Body, boxSize);  
 }  
 return nullptr;  
}

#### Sphere Shape

Ref<Physics::Collider> RigidBodyComponent::AddCollider(const float& sphereRadius)  
{  
 if(Physics::PhysicsEngine::IsActive() && Body)  
 {  
 return Physics::PhysicsEngine::CreateSphereCollider(Body, sphereRadius);  
 }  
 return nullptr;  
}

#### Capsule Shape

Ref<Physics::Collider> RigidBodyComponent::AddCollider(const float& capsuleRadius, const float& capsuleHeight)  
{  
 if(Physics::PhysicsEngine::IsActive() && Body)  
 {  
 return Physics::PhysicsEngine::CreateCapsuleCollider(Body, capsuleRadius, capsuleHeight);  
 }  
 return nullptr;  
}

### Physics Engine

The physics engine, like the creation of Rigidbody objects, also handles creating and adding colliders to a Rigidbody internally including tracking all colliders within the active engine instance. Common between the different CreateXXXCollider() functions within the physics engine is the creation of the [Collider](#_Colliders) object for the body using the provided values passed from the Rigidbody component and defaulting the collider to the identity location and orientation of the attached Rigidbody. Each function creates its own React [CollisionShape](#_CollisionShape) object that is then attached to the React [CollisionBody](#_CollisionBody) contained in the Rigidbody object.

#### Box Collider

Ref<Collider> PhysicsEngine::CreateBoxCollider(Ref<RigidBody> rigidbody, const glm::vec3& boxExtents)  
{  
 if(m\_PhysicsManager->m\_Active)  
 {  
 auto shape = m\_PhysicsManager->m\_PhysicsCommon.createBoxShape(Math::ToRp3dVec3(boxExtents));  
 auto col = Collider::Create(shape, Collider::Type::Box);  
 col->m\_ID = m\_PhysicsManager->m\_TotalColliders;  
 m\_PhysicsManager->m\_Colliders[col->m\_ID] = col;  
 m\_PhysicsManager->m\_TotalColliders++;  
 rigidbody->AddCollider(col);  
 rigidbody->m\_CollisionBody->addCollider(shape, rp3d::Transform::identity());  
 return col;  
 }  
 return nullptr;  
}

#### Sphere Collider

Ref<Collider> PhysicsEngine::CreateSphereCollider(Ref<RigidBody> rigidbody, const float& radius)  
{  
 if(m\_PhysicsManager->m\_Active)  
 {  
 auto shape = m\_PhysicsManager->m\_PhysicsCommon.createSphereShape(radius);  
  
 auto col = Collider::Create(shape, Collider::Type::Sphere);  
 rigidbody->AddCollider(col);  
 col->m\_ID = m\_PhysicsManager->m\_TotalColliders;  
 m\_PhysicsManager->m\_Colliders[col->m\_ID] = col;  
 m\_PhysicsManager->m\_TotalColliders++;  
 rigidbody->m\_CollisionBody->addCollider(shape, rp3d::Transform::identity());  
 return col;  
 }  
 return nullptr;  
}

#### Capsule Collider

Ref<Collider> PhysicsEngine::CreateCapsuleCollider(Ref<RigidBody> rigidbody, const float& radius, const float& height)  
{  
 if(m\_PhysicsManager->m\_Active)  
 {  
 auto shape = m\_PhysicsManager->m\_PhysicsCommon.createCapsuleShape(radius, height);  
  
 auto col = Collider::Create(shape, Collider::Type::Capsule);  
 rigidbody->AddCollider(col);  
 col->m\_ID = m\_PhysicsManager->m\_TotalColliders;  
 m\_PhysicsManager->m\_Colliders[col->m\_ID] = col;  
 m\_PhysicsManager->m\_TotalColliders++;  
 rigidbody->m\_CollisionBody->addCollider(shape, rp3d::Transform::identity());  
 return col;  
 }  
 return nullptr;  
}

### Collider

The constructor creates the collider object to then be attached to the Rigidbody by the Physics Engine using the provided React [CollisionShape](#_CollisionShape) and Collider Type.

Collider::Collider(rp3d::CollisionShape\* collider, Type type)  
 :m\_Collider(collider), m\_Type(type)  
{  
   
 switch (m\_Type)  
 {  
 case Type::Box:  
 InitializeBox();  
 break;  
 case Type::Sphere:  
 InitializeSphere();  
 break;  
 case Type::Capsule:  
 InitializeCapsule();  
 break;  
 default:  
 break;  
 }  
}

# Colliders

The Collider class is a wrapper class used in conjunction with Rigidbody objects and the Physics Engine to contain the required information for detecting and responding accurately to collisions.

### The Collider Class

\*Note, some helper/irrelevant functions and properties of this class have been removed from this snippet.

class Collider : public RefCounted  
{  
public:  
 enum class Type { Box, Sphere, Capsule };  
   
 Collider();  
 Collider(rp3d::CollisionShape\* collider, Type type);  
  
 static Ref<Collider> Create(rp3d::CollisionShape\* collider, Type type);  
   
 void SetPosition(const glm::vec3& position);  
 void SetOrientation(const glm::quat& orientation);  
 glm::vec3 GetColliderPosition() const;  
 glm::quat GetColliderOrientation() const;  
   
 BodyMass& GetColliderMass();  
 Type GetType() const;  
 std::pair<Type, rp3d::CollisionShape\*> GetCollisionShapePair();  
 glm::vec3 GetLocalColliderInertiaTensor();

float GetVolume()  
 void SetVolume(const float value);  
 float GetDensity();  
 void SetDensity(const float value);  
  
private:  
void InitializeBox();  
void InitializeSphere();  
void InitializeCapsule();  
  
private:  
 rp3d::CollisionShape\* m\_Collider = nullptr;  
Type m\_Type = Type::Box;  
glm::vec3 m\_LocalPosition = {};  
glm::quat m\_LocalOrientation = {};  
  
BodyMass m\_Mass;  
  
 InertiaTensor m\_InertiaTensor;  
  
 uint32\_t m\_ID = 0;  
float m\_Volume = 0.0f;  
float m\_Density = 1.0f;

### Initialization

When a collider object is created, an initialization function is processed based on the colliders type. This initialization process generates and calculates the mass, inverse mass and center of mass and its local inertia tensor for the shape. By default, the material density used is 1kg/m³

#### Mass

Mass is calculated using both a shapes volume and material density. The following is the formular used:

or

#### Box

##### Equations

##### Function

void Collider::InitializeBox()  
{  
 rp3d::BoxShape\* box = static\_cast<rp3d::BoxShape\*>(m\_Collider);  
  
 const auto fullExtents = Math::FromRp3dVec3(box->getHalfExtents() \* 2.0f);  
  
 *//Volume of a cube/box: V = length \* width \* height* m\_Volume = fullExtents.x \* fullExtents.y \* fullExtents.z;  
 m\_Mass.SetMass(m\_Density \* m\_Volume);  
  
 *//Generate a inertia tensor for the shape* float factor = 1.0f / 12.0f \* m\_Mass.Mass;  
 *//Todo Check if we really should be using half or full extents* auto x = box->getHalfExtents().x \* box->getHalfExtents().x;  
 auto y = box->getHalfExtents().y \* box->getHalfExtents().y;  
 auto z = box->getHalfExtents().z \* box->getHalfExtents().z;  
 auto tensor = glm::vec3(factor \* (y + z), factor \* (x + z), factor \* (x + y));  
 m\_InertiaTensor.LocalInertiaTensor = tensor;  
   
}

#### Sphere

##### Equations

##### Function

void Collider::InitializeSphere()  
{  
 rp3d::SphereShape\* sphere = static\_cast<rp3d::SphereShape\*>(m\_Collider);  
  
 *//Volume of a Sphere: V = (4/3)πr3* m\_Volume = (4/3) \* glm::pi<float>() \* glm::pow(sphere->getRadius(), 3);  
 m\_Mass.SetMass(m\_Density \* m\_Volume);  
  
 *//Generate a inertia tensor for the shape* float diag = (2.0f / 5.0f) \* m\_Mass.Mass \* glm::pow(sphere->getRadius(),2);  
 auto tensor = glm::vec3(diag, diag, diag);  
 m\_InertiaTensor.LocalInertiaTensor = tensor;  
   
}

#### Capsule

Capsules are calculated using a cylinder and two dome shapes.

##### Equations

##### Function

void Collider::InitializeCapsule()  
{  
 rp3d::CapsuleShape\* capsule = static\_cast<rp3d::CapsuleShape\*>(m\_Collider);  
   
 *// Volume of a Capsule: V = πr2((4/3)r + a)* m\_Volume = glm::pi<float>() \* glm::pow(capsule->getRadius(), 2) \* ((4.0f/3.0f) \* glm::pi<float>() + capsule->getHeight());  
 m\_Mass.SetMass(m\_Density \* m\_Volume);  
  
 *//Generate a inertia tensor for the shape* float mass = m\_Mass.Mass;  
 float height = capsule->getHeight();  
 float radius = capsule->getRadius();  
 float radiusSqr = glm::pow(capsule->getRadius(), 2);  
 float heightSqr = glm::pow(height, 2);  
 float radiusDoubleSqr = radiusSqr + radiusSqr;  
 float fac1 = 2.0f \* radius / (4.0f \* radius + 3.0f \* height);  
 float fac2 = 3.0f \* height / (4.0f \* radius + 3.0f \* height);  
 float sum1 = 0.4f \* radiusDoubleSqr;  
 float sum2 = 0.75f \* height \* radius + 0.5f \* heightSqr;  
 float sum3 = 0.25 \* radiusSqr + (1.0f / 12.0f) \* heightSqr;  
 float Ixxzz = fac1 \* mass \* (sum1 + sum2) + fac2 \* mass \* sum3;  
 float Iyy = fac1 \* mass \* sum1 + fac2 \* mass \* 0.25f \* radiusDoubleSqr;  
 auto tensor = glm::vec3(Ixxzz, Iyy, Ixxzz);  
 m\_InertiaTensor.LocalInertiaTensor = tensor;  
}

# Physics Engine and Physics Manager

As mentioned earlier, the Physics Engine is the driving force behind the physics system within the engine and handles everything from the creating of Rigidbody objects and Colliders to attach to those bodies but the Physics Engine also controls the main physics loop of the engine and the Physics Manager.

### Physics Engine

#### Class

\*Note, some helper/irrelevant functions and properties of this class have been removed from this snippet.

|  |
| --- |
| *class* PhysicsEngine { *public*: *static* Ref<PhysicsManager> *GetManager*(); *static bool IsActive*(); *static bool Initialise*(rp3d::EventListener \* *collisionEventListener*); *static void OnFixedUpdate*(Timestep *deltaTime*, PhysicsTimestep *timestep*, entt::registry& *componentRegistry*);  *static void OnUpdate*(Timestep *timestep*);  *static void Destroy*(); *static void DestroyRigidbody*(Ref<RigidBody> *rigidbody*);  *static* Ref<RigidBody> *CreateRigidbody*(GameObject& *gameObject*); *static* Ref<Collider> *CreateBoxCollider*(Ref<RigidBody> *rigidbody*, *const* glm::vec3& *halfExtents*); *static* Ref<Collider> *CreateSphereCollider*(Ref<RigidBody> *rigidbody*, *const float*& *radius*); *static* Ref<Collider> *CreateCapsuleCollider*(Ref<RigidBody> *rigidbody*, *const float*& *radius*, *const float*& *height*);  *private*: *static void EnableLogging*(); *static void PhysicsLoggerInit*();  *static void ProcessCollisions*();  *static void ProcessRigidbodyVelocities*(*const* Timestep& *deltaTime*, entt::registry& *componentRegistry*);  *static void ProcessRigidbodyPositions*(*const* Timestep& *deltaTime*, entt::registry& *componentRegistry*);  *static void ResetRigidbodyForcesAndTorques*(entt::registry& *componentRegistry*);  *static void UpdateInertiaTensor*(entt::registry& *componentRegistry*);  *static void UpdateBodies*(entt::registry& *componentRegistry*);  *static void UpdateSleepState*(entt::registry& *componentRegistry*, Timestep *deltaTime*);  *private*:  *static* Ref<PhysicsManager> *m\_PhysicsManager*; *static* TransformComponent *m\_ColliderTransform*;   *friend class* tnah::Scene;  *friend class* tnah::Renderer; }; |

#### Fixed Update & Its Steps

The OnFixedUpdate() function within the Physics Engine is processed every application cycle and accepts both the applications Timestep (time between frames) , a PhysicsTimestep (time between physics steps) and a reference to the current entity component registry. The PhysicsTimestep property is managed by the application and provides a fixed time interval for physics calculations. The main goal of fixed updates to ensure the physics is calculated and updated separately to the rendering loop of the application as physics calculations are costly on performance, causing stutters and performance issues if tied to the applications framerate directly. All steps of the physics update processes all Rigidbody objects active in the scene by using the component registry to access and get all required Rigidbody and transform components required.

##### Step 0: Core

The core of the function does two basic tasks before preceding to process any physics calculations, firstly a simple check to ensure the [PhysicsManager](#_Physics_Manager) is initialized and active within the application and secondly, updating the PhysicsManager. More on the [PhysicsManager updating](#_Updating)!

*void* PhysicsEngine::*OnFixedUpdate*(Timestep *deltaTime*, PhysicsTimestep *timestep*, entt::registry& *componentRegistry*)  
{  
 *if*(*IsActive*())  
 {  
 *m\_PhysicsManager*->OnFixedUpdate(*deltaTime*, *timestep*);

##### Step 1: Updating Inertia Tensors

The first job to update is each Rigidbody and its inertia tensor from the previous update. Doing this ensures the body and its inertia tensor are accurate to the current position and orientation of the body within the world.

*void* PhysicsEngine::*UpdateInertiaTensor*(entt::registry& *componentRegistry*)  
**{** *auto* view = *componentRegistry*.view<TransformComponent, RigidBodyComponent>();  
 *for*(*auto* e : view)  
 {  
 *auto*& rb = view.get<RigidBodyComponent>(e).Body;  
 *auto*& trans = view.get<TransformComponent>(e);  
 glm::mat3 rot = glm::mat3\_cast(rb->m\_Orientation);  
 *auto*& world = rb->m\_InertiaTensor.WorldInverseInertiaTensor;  
 *auto*& local = rb->m\_InertiaTensor.LocalInverseInertiaTensor;  
 world[0][0] = rot[0][0] \* local.x;  
 world[0][1] = rot[1][0] \* local.x;  
 world[0][2] = rot[2][0] \* local.x;  
  
 world[1][0] = rot[0][1] \* local.y;  
 world[1][1] = rot[1][1] \* local.y;  
 world[1][2] = rot[2][1] \* local.y;  
  
 world[2][0] = rot[0][2] \* local.z;  
 world[2][1] = rot[1][2] \* local.z;  
 world[2][2] = rot[2][2] \* local.z;  
  
 world = rot \* world;  
 }  
**}**

##### Step 2: Processing Collisions

Now all bodies in the scene have been updated, its now time to process any collisions that occurred. Collisions are managed by the React library but their information is managed and stored by our [CollisionDetectionEngine](#_Collision_Detection_Engine) wrapper class. Once React detects a collision, our [PhysicsEvent](#_Physics_Events) system is notified with all the relevant collision information. More on how we get and manage collision events is covered in the [CollisionDetectionEngine](#_Collision_Detection_Engine) section! More on the equations and code behind collisions is covered in the [Collision Resolution](#_Collision_Resolution_/) section!

This function sets up and processes the [PhysicsCollision](#_Physics_Collision) data from a queue each update. Once a collision has been processed, its popped or removed from the queue allowing the next collision to be processed. The collision queue contains all relevant information about a single collision and all its collision points. Each collision point is looped for a single collision event and a collision impulse is calculated. A collision point contains information regarding both Rigidbodies involved in the collision, the contact point in world space on each body, the contact normal in world space, the penetration depth of both bodies intersecting and relevant Rigidbody and game object references. Prior to calculating a collision impulse, the function gets and checks the type property of both bodies to detect if both bodies are static or sleeping, if so, we skip calculating a impulse as both bodies cant be moved thus avoiding costly calculations.

*void* PhysicsEngine::*ProcessCollisions*()  
{  
 *while*(!*m\_PhysicsManager*->*m\_CollisionDetectionEngine*->GetCurrentCollisions().empty())  
 {  
 *//loop over the collisions data and process a response for the collision  
 auto*& collision = *m\_PhysicsManager*->*m\_CollisionDetectionEngine*->GetCurrentCollisions().front();  
 *while*(!collision.GetCollisionData().empty())  
 {  
 *auto*& item = collision.GetCollisionData().front();  
  
 *auto* rb1 = item.GetRigidBodies().first;  
 *auto* rb2 = item.GetRigidBodies().second;  
  
 *if*((rb1->IsSleeping() || rb1->m\_BodyType == BodyType::Static) &&

(rb2->IsSleeping() || rb2->m\_BodyType == BodyType::Static))  
 **{** collision.GetCollisionData().pop();  
 *continue*;  
 }

//

// Collision Calculations from here are in the [Collision Resolution Section](#_Collision_Resolution_/)

//

collision.GetCollisionData().pop();  
 }  
 *m\_PhysicsManager*->*m\_CollisionDetectionEngine*->GetCurrentCollisions().pop();  
 }  
**}**

##### Step 3: Processing Velocities

Once an impulse has been processed for any bodies that have collided, their velocities are processed and updated based on several factors. Firstly, we need to ensure the body is dynamic and it’s not asleep, we also do a check to ensure the body has colliders attached as without a collider, the body would never be able to collide with any other body in the world, rendering velocity useless due to unset mass properties.

The new velocity is generated and processed based on the current deltaTime step of the application rather than the physics timestep as velocity and positions need to be updated matching the applications framerate to ensure smooth motion. Using the provided formulars:

Applying gravity with:

Applying Linear and Angular Dampening:

*void* PhysicsEngine::*ProcessRigidbodyVelocities*(*const* Timestep& *deltaTime*, entt::registry& *componentRegistry*)  
**{** *auto* view = *componentRegistry*.view<RigidBodyComponent>();  
 *for*(*auto* e : view)  
 {  
 *auto*& rb = view.get<RigidBodyComponent>(e).Body;  
   
 *if*(rb->GetType() == BodyType::Dynamic && !rb->IsSleeping() && rb->HasColliders())  
 {  
 *const auto* linear = rb->m\_LinearVelocity.Velocity;  
 *const auto* angular = rb->m\_AngularVelocity.Velocity;  
   
 rb->m\_ConstrainedLinearVelocity.Velocity = linear + *deltaTime*.GetSeconds() \* rb->GetBodyMass().InverseMass \*  
 rb->m\_LinearRotationLock \* rb->m\_Force.Forces;  
   
 rb->m\_ConstrainedAngularVelocity.Velocity = angular + *deltaTime*.GetSeconds() \* rb->m\_AngularRotationLock \*  
 rb->GetInertiaTensor().WorldInverseInertiaTensor \* rb->m\_Torque.Torques;  
 }  
  
 *if*(*m\_PhysicsManager*->GetGravityState() && rb->GetType() != BodyType::Static && !rb->IsSleeping() && !rb->IgnoreGravity() && rb->HasColliders())  
 {  
 rb->m\_ConstrainedLinearVelocity.Velocity += *deltaTime*.GetSeconds() \* rb->GetBodyMass().InverseMass \*  
 rb->m\_LinearRotationLock \* rb->GetBodyMass().Mass \* *m\_PhysicsManager*->GetGravity();  
 }  
  
 *if*(rb->GetType() != BodyType::Static && !rb->IsSleeping() && rb->HasColliders())  
 {  
 *auto* lDamp = glm::pow(1.0f - rb->m\_LinearDampening.Dampening, *deltaTime*.GetSeconds());  
 *auto* aDamp = glm::pow(1.0f - rb->m\_AngularDampening.Dampening, *deltaTime*.GetSeconds());  
 rb->m\_ConstrainedLinearVelocity.Velocity \*= lDamp;  
 rb->m\_ConstrainedAngularVelocity.Velocity \*= aDamp;  
 }  
 }  
**}**

##### Step 4: Processing Positions

Once all the velocity values have been finalized, the PhysicsEngine process the positions of each body. Using the velocity values, the orientation and position is calculated and stored.

*void* PhysicsEngine::*ProcessRigidbodyPositions*(*const* Timestep& *deltaTime*, entt::registry& *componentRegistry*)  
**{** *auto* view = *componentRegistry*.view<TransformComponent, RigidBodyComponent>();  
 *for*(*auto* e : view)  
 {  
 *auto*& rb = view.get<RigidBodyComponent>(e).Body;  
 *auto*& trans = view.get<TransformComponent>(e);  
  
 *if*(rb->GetType() == BodyType::Dynamic && !rb->IsSleeping() && rb->HasColliders())  
 {  
   
 trans.Position += rb->m\_ConstrainedLinearVelocity.Velocity \* *deltaTime*.GetSeconds();  
 rb->m\_Position = trans.Position;  
   
 rb->m\_Orientation += glm::quat(0.0, rb->m\_AngularVelocity) \* *deltaTime*.GetSeconds();  
 rb->m\_Orientation = glm::normalize(rb->m\_Orientation);  
   
 }  
 *else if*(rb->GetType() == BodyType::Kinematic && !rb->IsSleeping())  
 {trans.Position += rb->m\_ConstrainedLinearVelocity.Velocity \* *deltaTime*.GetSeconds();  
 rb->m\_Position = trans.Position;  
 }  
 }  
**}**

##### Step 5: Updating the Rigidbodies

Now that the new positions and orientation of the body has been calculated, the engine now updates each body with its new position and orientation, based off its body type, this function will only set its position or both position and orientation.

*void* PhysicsEngine::*UpdateBodies*(entt::registry& *componentRegistry*)  
**{** *auto* view = *componentRegistry*.view<TransformComponent, RigidBodyComponent>();  
 *for*(*auto* e : view)  
 {  
 *auto*& rb = view.get<RigidBodyComponent>(e).Body;  
 *auto*& transform = view.get<TransformComponent>(e);  
   
 rb->OnUpdate(transform);  
   
 rb->m\_LinearVelocity.Velocity = rb->m\_ConstrainedLinearVelocity.Velocity;  
 rb->m\_AngularVelocity.Velocity = rb->m\_ConstrainedAngularVelocity.Velocity;  
  
 *auto* t = rb->m\_CollisionBody->getTransform();  
 t.setPosition(Math::ToRp3dVec3(rb->m\_Position));  
 *if*(rb->GetType() == BodyType::Kinematic)  
 {  
 rb->m\_CollisionBody->setTransform(t);  
 }  
 *else* {  
 t.setOrientation(Math::ToRp3dQuat(rb->m\_Orientation));  
 rb->m\_CollisionBody->setTransform(t);  
 transform.Rotation = glm::eulerAngles(rb->m\_Orientation);  
 transform.QuatRotation = rb->m\_Orientation;  
 }  
  
 *for*(*auto*& c : rb->m\_Colliders)  
 {  
 *auto* col = c.second;  
 col->SetPosition(transform.Position + col->GetColliderPosition());  
 col->SetOrientation(rb->m\_Orientation + col->GetColliderOrientation());  
 }  
 }  
}

##### Step 6: Updating the sleep states

\*Note Sleep functionality is only operational partially and isn’t fully debugged or tested.

Once the bodies have been updated, the engine checks to see if the body is allowed to sleep. This is don’t by checking if the velocity is below a threshold and a set amount of time has passed, then puts the body to sleep to avoid processing it again when its not required.

*void* PhysicsEngine::*UpdateSleepState*(entt::registry& *componentRegistry*, Timestep *deltaTime*)  
**{** *if*(*IsActive*())  
 {  
 *auto* view = *componentRegistry*.view<RigidBodyComponent, TransformComponent>();  
 *for*(*auto* e : view)  
 {  
 *auto*& rigidbody = view.get<RigidBodyComponent>(e).Body;  
 *auto*& transform = view.get<TransformComponent>(e);  
  
 *auto* sleepLinear = length2(rigidbody->GetLinearVelocity().Velocity \* rigidbody->GetLinearVelocity().Velocity);  
 *auto* sleepAngular = length2(rigidbody->GetAngularVelocity().Velocity \* rigidbody->GetAngularVelocity().Velocity);  
 *auto* sleepCap = glm::pow(rigidbody->m\_SleepVelocityThreshold, 2);  
 *auto* sleepAngularCap = 3.0f \* (glm::pi<*float*>() / 180.0f);  
  
 *if*(rigidbody->GetType() == BodyType::Static)  
 *continue*;  
  
 *if*(sleepLinear > sleepCap || sleepAngular > sleepAngularCap || *m\_PhysicsManager*->GetGravityState() || !*m\_PhysicsManager*->m\_SleepAllowed)  
 {  
 rigidbody->m\_SleepTimeAccumulator = 0.0f;  
 }  
 *else* {  
 rigidbody->m\_SleepTimeAccumulator += *deltaTime*.GetSeconds();  
 }  
  
  
 *if*(rigidbody->m\_SleepTimeAccumulator >= rigidbody->m\_SleepTimeThreshold)  
 {  
 rigidbody->Sleep();  
 }  
   
 }  
 }  
**}**

##### Step 7: Resetting External Forces & Torque

Finally the external force and torque values are reset back to zero to ensure the bodies don’t have either applied to the body more than once.

*void* PhysicsEngine::*ResetRigidbodyForcesAndTorques*(entt::registry& *componentRegistry*)  
**{** *auto* view = *componentRegistry*.view<RigidBodyComponent>();  
 *for*(*auto* e : view)  
 {  
 *auto*& rigidbody = view.get<RigidBodyComponent>(e).Body;  
 rigidbody->m\_Force.Forces = {0.0f, 0.0f, 0.0f};  
 rigidbody->m\_Torque.Torques = {0.0f, 0.0f, 0.0f};  
 }  
**}**

#### Debug Rendering

##### Updating the Renderer

Every application loop, the collider renderer is updated with new line and triangle information to render to the application window. This information is provided by React and is simply collected within the engine, processed into the required data structures and then passed to OpenGL for rendering. Rendering the collider/Rigidbody shapes is a debug future and can be disabled by setting the debug flag on the Application to false.

*void* PhysicsEngine::*OnUpdate*(Timestep timestep)  
**{** *UpdateColliderRenderer*();  
**}**

*void* PhysicsEngine::*UpdateColliderRenderer*()  
**{** *if*(*m\_PhysicsManager*->m\_Active)  
 {  
 *//Check and return if the collider rendering objects haven't been created yet  
 // Only check one object, if one is null then they all are  
 if*(*m\_PhysicsManager*->m\_LinesVertexArray == *nullptr*) *return*;  
 *if*(!*m\_PhysicsManager*->m\_ColliderRendererInit) *return*;  
  
 *//Check and only update if we really want to render the colliders  
 if*(*m\_PhysicsManager*->m\_ColliderRender)  
 {  
 *auto* renderer = *GetColliderRenderer*();  
 *const* rp3d::uint nbLines = renderer->getNbLines();  
 *if*(nbLines > 0)  
 {  
 *const* uint32\_t size = nbLines \* *sizeof*(rp3d::DebugRenderer::DebugLine);  
 *m\_PhysicsManager*->m\_LinesVertexArray->Bind();  
 *m\_PhysicsManager*->m\_LinesVertexBuffer->SetData(size, renderer->getLinesArray(), DrawType::STREAM, TypeMode::DRAW);  
 *m\_PhysicsManager*->m\_LinesVertexArray->SetIndexSize(nbLines \* 2);  
  
 VertexBufferLayout layout = *m\_PhysicsManager*->m\_LinesVertexBuffer->GetLayout();  
 layout.SetStride(*sizeof*(rp3d::Vector3) + *sizeof*(rp3d::uint32));  
 *auto*& elements = layout.GetElements();  
 elements.at(1).Offset = *sizeof*(rp3d::Vector3);  
 *m\_PhysicsManager*->m\_LinesVertexBuffer->SetLayout(layout);  
 *m\_PhysicsManager*->m\_LinesVertexBuffer->Unbind();  
 *m\_PhysicsManager*->m\_LinesVertexArray->Unbind();  
 }  
  
 *// Triangles  
 const* rp3d::uint nbTriangles = renderer->getNbTriangles();  
 *if*(nbTriangles > 0)  
 {  
 *const* uint32\_t size = nbTriangles \* *sizeof*(rp3d::DebugRenderer::DebugTriangle);  
 *m\_PhysicsManager*->m\_TriangleVertexArray->Bind();  
 *m\_PhysicsManager*->m\_TriangleVertexBuffer->SetData(size, renderer->getTrianglesArray(), DrawType::STREAM, TypeMode::DRAW);  
 *m\_PhysicsManager*->m\_TriangleVertexArray->SetIndexSize(nbTriangles \* 3);  
  
 VertexBufferLayout layout = *m\_PhysicsManager*->m\_TriangleVertexBuffer->GetLayout();  
 layout.SetStride(*sizeof*(rp3d::Vector3) + *sizeof*(rp3d::uint32));  
 *auto*& elements = layout.GetElements();  
 elements.at(1).Offset = *sizeof*(rp3d::Vector3);  
 *m\_PhysicsManager*->m\_TriangleVertexBuffer->SetLayout(layout);  
 *m\_PhysicsManager*->m\_TriangleVertexBuffer->Unbind();  
 *m\_PhysicsManager*->m\_TriangleVertexArray->Unbind();  
 }  
 }  
 }  
**}**

##### Drawing Debug Lines

**{** *//Collider rendering should only be used for debugging and in the editor to set sizes  
 if*((m\_IsEditorScene || Application::*Get*().GetDebugModeStatus()) && Physics::PhysicsEngine::*IsColliderRenderingEnabled*() && passes == 0)  
 {  
 *auto* pair = Physics::PhysicsEngine::*GetColliderRenderObjects*();  
 *auto* lineArr = pair.first.first;  
 *auto* lineBuf = pair.first.second;  
  
 *auto* triArr = pair.second.first;  
 *auto* triBuf = pair.second.second;  
 Renderer::*SubmitCollider*(lineArr, lineBuf,triArr,triBuf);  
 }  
**}**

### Physics Manager

The Physics Manager contains all the required low level objects for the physics system to operate, some objects include: React PhysicsWorld object, React PhysicsCommon object, React Logger object and all the required rendering objects for the physics debug renderer. The manager also holds constant world properties like the values of gravity and whether gravity is enabled.

#### Class

\*Note, some helper/irrelevant functions and properties of this class have been removed from this snippet.

*class* PhysicsManager : *public* RefCounted  
 {  
 *public*:PhysicsManager();~PhysicsManager();  
  
 glm::vec3& GetGravity() { *return* m\_Gravity; }  
 *void* SetGravity(*const* glm::vec3& *gravity*) { m\_Gravity = *gravity*; }  
 *void* SetGravityState(*const bool*& *state*) { m\_GravityEnabled = *state*; }  
 *bool*& GetGravityState() { *return* m\_GravityEnabled; }  
 *bool*& GetSleepState() { *return* m\_SleepAllowed; }  
  
 Ref<CollisionDetectionEngine>& GetCollisionDetectionEngine();  
   
 *private*:  
 *bool* Initialise(rp3d::EventListener \* *collisionEventListener*); *void* OnFixedUpdate(Timestep *deltaTime*, PhysicsTimestep *timestep*) *const*; *void* Destroy();rp3d::RigidBody\* CreateRigidBody(*const* TransformComponent& *transform*) *const*; *void* DestroyRigidBody(rp3d::CollisionBody\* *rigidBody*) *const*;rp3d::CollisionBody\* CreateCollisionBody(*const* TransformComponent& *transform*) *const*; *void* DestroyCollisionBody(rp3d::CollisionBody \* *body*) *const*; *void* CreateColliderRenderer();  
   
 *private*:  
rp3d::PhysicsCommon m\_PhysicsCommon;  
 rp3d::PhysicsWorld\* m\_PhysicsWorld = *nullptr*;  
   
rp3d::DefaultLogger\* m\_PhysicsLogger = *nullptr*;  
  *bool* m\_ColliderRender = *false*;  
 *bool* m\_ColliderRendererInit = *false*;  
   
 Ref<VertexArray> m\_LinesVertexArray;  
Ref<VertexBuffer> m\_LinesVertexBuffer;  
Ref<VertexArray> m\_TriangleVertexArray;

Ref<VertexBuffer> m\_TriangleVertexBuffer;  
Ref<Shader> m\_Shader;  
VertexBufferLayout m\_ColliderLayout;  
  
 *bool* m\_GravityEnabled = *true*;  
  
 glm::vec3 m\_Gravity = {0.0f, -9.8f, 0.0f};  
 *bool* m\_Active = *false*;  
 *bool* m\_Logging = *false*;  
  
 std::unordered\_map<uint32\_t, Ref<RigidBody>> m\_Rigidbodies;  
  
 uint32\_t m\_TotalRigidbodies = 0;  
  
 std::unordered\_map<uint32\_t, Ref<Collider>> m\_Colliders;  
  
 uint32\_t m\_TotalColliders = 0;

*static* Ref<CollisionDetectionEngine> *m\_CollisionDetectionEngine*;  
  
 *bool* m\_SleepAllowed = *false*;  
};

#### Initialization

When the manager is initialized, multiple objects are checked and a false is returned if their invalid. If the required objects pass the check, the event listener is set and the manager is set to active.

*bool* PhysicsManager::Initialise(rp3d::EventListener\* *collisionEventListener*)  
**{** m\_PhysicsWorld = m\_PhysicsCommon.createPhysicsWorld();  
 *if*(m\_PhysicsWorld == *nullptr* || *collisionEventListener* == *nullptr*)  
 *return false*;  
   
 m\_PhysicsWorld->setEventListener(*collisionEventListener*);  
 m\_Active = *true*;  
   
 *return true*;  
**}**

The first time the physics debug renderer is enabled, it checks and creates any required objects if they haven’t already. Once the required rendering objects are created, debug rendering is toggled on and off using the same function.

*void* PhysicsManager::CreateColliderRenderer()  
**{** m\_LinesVertexArray = VertexArray::*Create*();  
 m\_LinesVertexBuffer = VertexBuffer::*Create*();  
  
  
 m\_TriangleVertexArray = VertexArray::*Create*();  
 m\_TriangleVertexBuffer = VertexBuffer::*Create*();  
 m\_ColliderLayout = {  
 {ShaderDataType::Float3, "a\_Position"},  
 {ShaderDataType::UInt, "a\_VertexColor"}  
 };  
 m\_LinesVertexBuffer->SetLayout(m\_ColliderLayout);  
 m\_LinesVertexArray->AddVertexBuffer(m\_LinesVertexBuffer);  
  
 m\_TriangleVertexBuffer->SetLayout(m\_ColliderLayout);  
 m\_TriangleVertexArray->AddVertexBuffer(m\_TriangleVertexBuffer);  
   
 m\_Shader = Shader::*Create*("Resources/shaders/default/physics/physics\_vertex.glsl","Resources/shaders/default/physics/physics\_fragment.glsl");  
  
 m\_ColliderRendererInit = *true*;  
**}**

#### Updating

When the physics manager is updated, the collision detection engine handles the updating of the world. Typically updating the physics world using the update() function within React would process and update any Rigidbody objects active in React but because the engine doesn’t use React Rigidbodies, this update function only updates the collision detection and reports it to the event listener.

*void* PhysicsManager::OnFixedUpdate(Timestep *deltaTime*, PhysicsTimestep timestep) *const***{** *//m\_PhysicsWorld->update(timestep.GetSimulationSpeed());  
 m\_CollisionDetectionEngine*->FixedUpdate(m\_PhysicsWorld, *deltaTime*.GetSeconds());  
**}**

# Collision Detection Engine

### Class

The Collision Detection Engine is a wrapper for React and manages two requirements for the engine. Firstly, is updating the physics world for collisions to be detected and secondly is to register and store all collisions in the previous update. These collisions are stored in a queue and are processed by the [PhysicsEngine](#_Physics_Engine) in the [Process Collisions](#_Step_2:_Processing) step.

*class* CollisionDetectionEngine : *public* RefCounted  
{  
*public*:  
 *static* Ref<CollisionDetectionEngine> *Create*();  
   
 CollisionDetectionEngine();  
   
 *void* FixedUpdate(rp3d::PhysicsWorld\* *physicsWorld*, PhysicsTimestep *physTime*);  
  
 std::queue<PhysicsCollision>& GetCurrentCollisions() { *return* m\_CurrentCollisions; }  
  
 *void* RegisterCollision(PhysicsCollision *collision*) { m\_CurrentCollisions.push(*collision*); }  
  
*private*:  
  
 std::queue<PhysicsCollision> m\_CurrentCollisions;  
};

### Physics Events

#### Class

The events class is a derived class from React and is alerted on every collision within the world with the collision data. This data is then processed and stored in the collision queue.

*class* PhysicsEvents : *public* rp3d::EventListener  
{  
*public*: *void* onContact(*const* CallbackData& *callbackData*) *override*; *void* onTrigger(*const* rp3d::OverlapCallback::CallbackData& *callbackData*) *override*;  
};

### Physics Collision

The physics collision class contains all the required data and functions for a individual collision event. This includes all the contact points and a queue of the contact points collision information.

#### Class

*class* PhysicsCollision  
{  
*public*:

//CollisionData class defined here

PhysicsCollision();  
   
 PhysicsCollision(rp3d::CollisionCallback::ContactPair *contactPair*);  
  
 uint32\_t GetTotalContactPoints() *const* { *return* m\_TotalContactPoints; }  
   
 std::queue<CollisionData>& GetCollisionData() { *return* m\_CollisionData; }  
  
*private*:  
  
 uint32\_t m\_TotalContactPoints = 0;  
   
 */\*\*  
 \** **@var** *m\_CollisionData  
 \*  
 \** **@brief** *A queue holding data about a collision.  
 \*/* std::queue<CollisionData> m\_CollisionData;  
   
};

#### CollisionData Class

The collision data class holds all the information regarding a contact point in a collision, this information includes the point, contact normal, penetration depth and pointer/ref’s back to the gameobjects and rigidbodies involved in the collision.

*class* CollisionData  
{  
*public*:  
 glm::vec3 GetContactPoint() *const* { *return* m\_ContactPoint; }  
 glm::vec3 GetContactNormal() *const* { *return* m\_ContactNormal; }  
 *float* GetPenetration() *const* { *return* m\_PenetrationDepth; }  
   
  
 std::pair<glm::vec3, glm::vec3> GetContactPositions() { *return* {m\_Body1ContactPosition, m\_Body2ContactPosition}; }  
  
 std::pair<GameObject\*, GameObject\*> GetGameObjects() { *return* {m\_Body1GameObject, m\_Body2GameObject}; }  
  
 std::pair<Ref<RigidBody>, Ref<RigidBody>> GetRigidBodies() { *return* {m\_Body1RigidBody, m\_Body2RigidBody}; }  
  
*private*:  
   
  
 glm::vec3 m\_ContactPoint = {0,0,0};  
 glm::vec3 m\_ContactNormal = {0,0,0};  
  
 glm::vec3 m\_Body1ContactPosition = {0,0,0};  
 glm::vec3 m\_Body2ContactPosition = {0,0,0};  
  
 *float* m\_PenetrationDepth = 0.0f;  
  
 GameObject\* m\_Body1GameObject = *nullptr*;  
 GameObject\* m\_Body2GameObject = *nullptr*;  
  
 Ref<RigidBody> m\_Body1RigidBody = *nullptr*;  
 Ref<RigidBody> m\_Body2RigidBody = *nullptr*;  
  
 *friend class* PhysicsCollision;  
};

# Collision Resolution / Response

### Equations

Frictionless Collision Impulse:

Linear Impulse: and

Angular Impulse: and

### Code

*constexpr float* restitution = 0.4f;  
  
*auto*& t1 = item.GetGameObjects().first->Transform();  
glm::vec3 lv1 = rb1->GetLinearVelocity();  
glm::vec3 av1 = rb1->GetAngularVelocity();  
  
*auto*& t2 = item.GetGameObjects().second->Transform();  
glm::vec3 lv2 = rb2->GetLinearVelocity();  
glm::vec3 av2 = rb2->GetAngularVelocity();  
  
glm::vec3 cp1 = item.GetContactPositions().first;  
glm::vec3 cp2 = item.GetContactPositions().second;  
glm::vec3 cn = item.GetContactNormal();  
  
glm::vec3 r1 = cp1 - t1.Position;  
glm::vec3 r2 = cp2 - t2.Position;  
  
*if*(rb1->GetType() != BodyType::Static && !rb1->IsSleeping())  
{  
 t1.Position += cn \* ((item.GetPenetration() / 2.0f) \* -1);  
}  
*if*(rb2->GetType() != BodyType::Static && !rb2->IsSleeping())  
{  
 t2.Position -= cn \* ((item.GetPenetration() / 2.0f) \* -1);  
}  
  
*auto* restitution\_multiplier = -(1.0f + restitution);  
  
*auto* relative\_velocity = lv1 - lv2;  
  
*auto* r1xn = glm::cross(r1, cn);  
  
*auto* r2xn = glm::cross(r2, cn);  
  
*auto* total\_inverse\_mass = rb1->GetBodyMass().InverseMass + rb2->GetBodyMass().InverseMass;  
  
*auto* numerator = restitution\_multiplier \* (glm::dot(cn, relative\_velocity) + glm::dot(av1, r1xn) - glm::dot(av2, r2xn));  
  
*auto* denominator = total\_inverse\_mass + (r1xn \* rb1->GetInertiaTensor().WorldInverseInertiaTensor \* r1xn) + (r2xn \* rb2->GetInertiaTensor().WorldInverseInertiaTensor \* r2xn);  
  
*auto* lambda = (numerator / denominator);  
  
*auto* linear\_impulse = lambda \* cn;  
  
*if*(rb1->GetType() == BodyType::Dynamic && !rb1->IsSleeping())   
{  
 lv1 += linear\_impulse \* rb1->GetBodyMass().InverseMass;  
 av1 += (lambda \* rb1->GetInertiaTensor().WorldInverseInertiaTensor \* r1xn);  
 rb1->m\_LinearVelocity.Velocity = lv1;  
 rb1->m\_AngularVelocity.Velocity = av1;  
}  
  
*if*(rb2->GetType() == BodyType::Dynamic && !rb2->IsSleeping())  
**{** lv2 -= linear\_impulse \* rb2->GetBodyMass().InverseMass;  
 av2 -= (lambda \* rb2->GetInertiaTensor().WorldInverseInertiaTensor \* r2xn);  
 rb2->m\_LinearVelocity.Velocity = lv2;  
 rb2->m\_AngularVelocity.Velocity = av2;  
}

# Group Declaration Sheet

**ICT398 Group Declaration Sheet (Please submit this completed sheet, individual accounting and self and peer assessment)**

**Assignment/Project Name: Milestone 2**

**Group Name: Team Names Are Hard (TNAH)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Member’s Names** | **Brief Description of Tasks** (Details would be in the individual accounting spreadsheet.  If more space is needed, attach extra sheets) | **Member’s contribution to the total work (%)** | **Date signed** | **Signature** |
| Dylan Blereau | Events, Mathematics, Collision Resolution | 33 | 06/11/21 | ***Dylan Blereau*** |
| Christopher Logan | Events, Collision management, Collision Resolution | 33 | 06/11/21 | ***Christopher Logan*** |
| Bryce Standley | ­­Collision Resolution, Data management, Events, Mathematics | 34 | 06/11/21 | ***Bryce Standley*** |
| ***Total percentage*** *(should total to 100% if all work is completed)* |  | 100% |  |  |

**Instructions:**

Please complete the information as specified above. It should list each group member’s percentage contribution to the submitted work. *For example if there are 4 members and they all make equal contributions to the total work, then each gets 25%*. This statement needs to be signed by all group members to indicate their agreement to percentage breakdown. The contribution percentage listed here is a summary of the individual accounting spreadsheet. The individual mark depends on this sheet. If everyone does an equal amount of contribution, everyone gets the project mark. A lower contribution will attract a pro rata mark. You would not get a higher mark than the project mark.

The purpose of this declaration is to provide documentary evidence of each group member’s contribution to the submitted work. This gets demonstrated during the lecture at the start of the semester. Please present this sheet together with the submitted work.

The **individual accounting spreadsheet** must also be completed and submitted with the assignment (detailed evidence). The **self and peer assessment form** should be completed and submitted separately (individually and not as a group submission). The final version of your **team charter** should also be included.