Modular Integration of an Existing Middleware Physics Engine into the Rythe Game Engine

Raphael Priatama 460756

Independent Project

External Assessor: Danylo Ierkaiev

Saxion University of Applied Sciences

Graduation Coach: Yvens Rebouças Serpa

Abstract

Rythe Interactive is looking to attract users for its game engine. This is difficult since a number of features within the engine are missing. One of these features is the engine's ability to support more advanced physics simulations. Rythe Interactive is also interested in supporting these features in a modular way since modularity is one of the selling points of the engine. By going through the third-party physics engines that are available within the market, PhysX was chosen based on the needs of the company and general performance capabilities. Using PhysX, the Rythe Game Engine is able to support rigidbody dynamics, character controllers, and scene queries. Modularity was achieved by ensuring that the generic physics interface did not depend on any PhysX objects. The end result is a game engine that supports the aforementioned physics features in a way that allows for the physics engine to be swapped if necessary.

Table of Contents

1. Introduction

A game engine is defined as a framework which supports game development in areas including importing assets from other software, creating scenes using assets, and applying visual effects and game logic into the application (Sim, 2019). A physics engine on the other hand can be defined as a software library used in the film and computer games industries to realistically animate a physical system (Pytlos, 2015). In the context of video games, a physics engine can be used to simulate a number of real-world phenomena such as the interaction between 2 solid objects, the springs of a vehicle, or the way a piece of cloth moves. The physics engines can also be used to detect the collision of key objects of interest which have various uses within gameplay. Middleware on the other hand is the software that assists an application to interact or communicate with other applications, networks, hardware, and/or operating systems (Bishop,2003).

The Rythe Engine is a C++ Game Engine developed with a focus on modularity and performance. Modularity in the context of the Rythe Engine means that specific modules of the engine can be swapped or removed without requiring major modification. This can be important for some developers since they may only want or need certain modules within the Rythe Engine. With the modularity provided by Rythe, they can acquire these modules and exclude the modules that are not desired.

The physics engine in the Rythe Game Engine, which will be further referred to as **Diviner**, supports the calculation of collision and collision response between solid convex objects. This includes shapes such as cubes, spheres, and capsules. However, most other commercially available game engines in the market have physics modules that can simulate other more complex features such as joints, vehicles, character controllers, etc. Extending the currently built physics engine may require extensive effort while there are existing third party physics engines that are readily available.

1.1 Company Outline

Rythe Interactive is a startup company built for the purpose of maintaining and extending the Rythe Engine (Rythe Interactive, 2022). Rythe Interactive is a relatively new company, officially registered in the Netherlands on 22nd of January 2022. Currently many of the company's foundations are still under construction. There are currently four programmers working under this organization, each working on extending specific modules or systems in the engine. Some of the features being worked on includes: extending core engine systems, a graphics engine abstraction layer, serialization, and extending the physics engine.

Working in a shared codebase requires some form of version control system in order to support modification or extension of the codebase. At Rythe, Git is used as a version control system. Another system used to support working in this shared codebase is peer reviews. The idea is that any time a programmer wants to modify/extend the codebase, this change must be reviewed by another programmer. The main goal of this is to ensure that the coding conventions are followed and to maintain the quality of the codebase.

1.2 Problem Analysis

Rythe Interactive is in the process of finding users for its product (the Rythe Game Engine). This endeavor is much more difficult when the product itself is missing a number of important features. One of these features, as previously established, is the ability to support simulation of more complex physics phenomena within the game engine. Extending the Diviner physics engine requires a significant amount of development time and therefore will cause a significant financial burden. Because of this, a third-party physics engine is much more preferable. This however, is not the only problem Rythe Interactive is concerned with. Rythe is also concerned with creating a modular integration of this third-party physics engine since this is one of the game engine's main selling points. Although the final result of a modular physics module that can support more advanced physics simulation is already clear and established, the way in which this can be implemented is not. Therefore, research is required.

1.3 Preliminary Research and Main Question

1.3.1 Researching Physics Features

In order to get an idea of which physics features that should be prioritized, the physics modules of widely used and commercially available engines will be used as examples. One method of finding this information is by going through a popular online video game store and listing which games use what game engine. One popular online video game store that can be examined is Steam. According to Presser (2022), Steam is a digital distribution platform for video games.

Fortunately, this endeavor of going through the Steam Store and identifying their game engines has been done by Toftedahl (2019). According to Toftedahl (2019), Unreal and Unity are the most used game engines within the Steam Store. The physics features that these engines provide are described with Table 1.

Unity supports 3 physics engines: A PhysX Integration, a custom physics engine called the "Unity Physics Engine", and a Havok Integration (Unity, n.d-b). Due to the special licensing requirement for the Havok Integration causing it to be much less commercially available than the other physics engines, Havok has been removed from Table 1 displayed below.

Unreal supports 2 physics engines: a PhysX Integration, and a custom physics engine called "Chaos" (Epic Games, n.d).

Table 1

Physics Features that are supported within the physics engines of Unity and Unreal.

Physics Feature	Unity		Unreal		Total
	PhysX(Integration)	Unity Physics	PhysX(Integration)	Chaos Physics	
Rigidbody Dynamics	✓	✓	✓	✓	4
Joints	✓	✗	✓	✓	3
Scene Queries	✓	✓	✓	✗	3
Character Controller	✓	✗	✓	✗	2
Vehicle	✗*	✗	✓	✓	2
Physics Articulations	✓	✗	✗	✗	1
Cloth	✗	✗	✗	✓	1
Dynamic Destruction	✗	✗	✗	✓	1

*Unity's Vehicle Module is incomplete

Based on this information an informed decision on what physics features to implement can be made based on how often a certain physics feature appears on the physics engines of the aforementioned game engines.

The most commonly supported physics feature out of these physics engines is rigidbody dynamics, joints, scene queries, character controllers, and vehicle dynamics.

1.3.2 Modularity

As previously established, Rythe Interactive is concerned with integrating a third-party physics engine in a modular way. In order to do this, a proper definition of modularity must be established. According to Erikstad (2019), the definition of modularity can have some significant variations depending on the field in which it is used. However, it is generally a concept that is describing some sort of system. More specifically, to what degree the system can be separated and recombined.

With this definition, a more specific goal can be established. The interface that is used by a potential user in order to communicate with the physics engine should not be specific to one physics engine. In other words, it should be possible to swap the physics engine that the engine is using without having to change the interface that is used to communicate with the physics engine.

With the information established, a main question can be constructed.

1.3.3 Main Question and Sub-Questions

How can the features of a third-party physics engine be integrated into the Rythe Engine in a modular way so that the Rythe Engine can provide physics simulation capabilities that are comparable to other commercially available game engines?

There are a number of sub-questions that emerge from this question. They are listed with the following:

1. How can rigidbody dynamics be integrated into the Rythe Engine in a modular way?
2. How can character controllers be integrated into the Rythe Engine in a modular way?
3. How can scene queries be integrated into the Rythe Engine in a modular way?

1.4 Scope

There are a number of subjects not covered in this paper. The first one is the integration of the Diviner Physics Engine into the abstract module that will be created for the integration of the Middleware Physics Engine. This is due to the fact that the implementation of this will likely be similar to the integration of the third-party physics engine. The modular integration of a number of physics phenomena such as dynamic destruction, cloth, vehicle dynamics, and physics articulations will not be covered as well since it has been established that these are features that are less commonly implemented in the physics engines of commercially available game engines.

2. Theory and Ideation

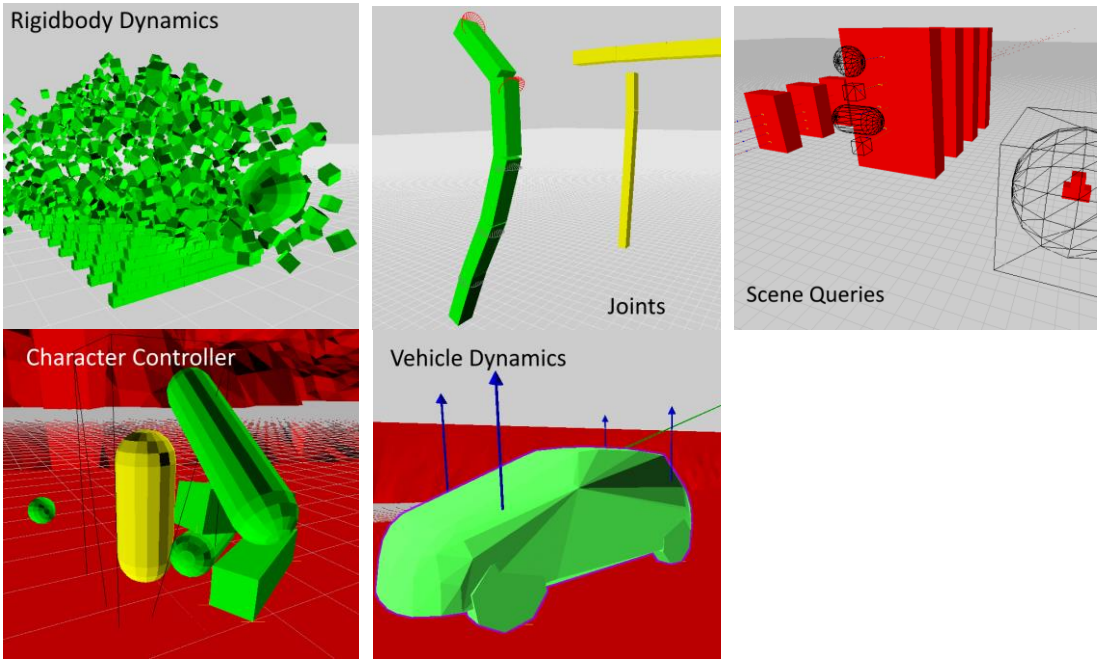


Figure 1. A set of images representing a number of physics features that can be found within a physics engine

Rigidbodies are physics objects that do not deform due to collisions (Dickinson, 2013). Rigidbodies are typically used to simulate the collision of solid objects.

Joints in the context of physics simulation is a physics feature that constrains the way objects move together (NVIDIA,2021a). Joints can be used in games as a way to simulate door hinges or ball-and-socket joints located on the shoulders of a game character.

Scene Queries can be defined as a set of utilities that can be used to detect collisions in a section of a given physics scene (NVIDIA,2021b). The section to be queried can be represented by a ray, a shape, or a sweeped shape. This can be used for a number of gameplay purposes such as detecting if a player has reached the finish line, detecting if the weapon a player is shooting hits an enemy, etc.

The definition above is the established definition of scene queries. In this paper however, when discussing scene queries, the paper also refers to another closely related feature known as trigger shapes. Trigger Shapes are objects within a physics simulation that are used to report overlaps within a scene and do not play a role in the simulation of the scene (NVIDIA,2021c).

A character controller is a capsule collider that can be instructed to move into an arbitrary direction while still being constrained by a collision environment (Unity, n.d-a). Character controllers are widely used in first person and third person games.

In the context of games, vehicle dynamics is concerned with simulating the movement of vehicles, specifically braking, acceleration, and cornering. Vehicle dynamics are primarily needed in the development of racing games (Zuvich,2000).

2.1 Physics Engine Middleware

There are a number of physics engine middleware available in the market. Including the ones previously mentioned, they are listed with the following: PhysX, Bullet, Havok, Jolt, Open Dynamics Engine, and Newton Dynamics.

In order to aid the process of selecting a third-party physics engine to integrate, a set of requirements must be established based on the needs and resources of Rythe Interactive. The requirements and the reasoning behind these requirements are described with the following:

1. **The physics engine license allows for modification and or examination of the source code and can be done without any financial cost.** If the physics middleware in question requires special licensing agreements, this will unfortunately be passed on to the user. This would be detrimental to Rythe's goal of getting more users for the game engine since it would transfer a financial burden on a user that would like to use Rythe's physics capabilities. From a development perspective, examination and modification of the source code will be important later on in the future when there is a need to modify the behavior of the simulation provided by the middleware.
2. **The physics engine supports the aforementioned physics features that have been selected for integration.** As established previously, the paper is focused on improving the engine in order to have physics simulation capabilities comparable to other commercially available game engines. It was also established that Rigid Body Dynamics, Scene Queries, Character Controllers, and Joints are the most important features to integrate in order to achieve this.
3. **The physics engine has been used in at least one commercially released game.** It is important that the physics engine has been utilized in a real game environment and can run alongside other systems in real-time. This means that common issues with the physics engine have been identified and resolved.

With the established requirements, a number of physics engines have been identified that meet the requirements. The following is a description of how the aforementioned physics engines meet or do not meet the established requirements. This description is summarized under Table 2.

Table 2

The requirements that need to be fulfilled by the physics engine in order to be considered a viable choice for integration and if and how the physics engine fulfills the requirements.

Requirement A	Allows for free modification and or examination of the source code
Requirement B	Supports the physics features that have been selected for integration
Requirement C	Has been used in at least one commercially released game

Physics Engine	Requirement A	Requirement B	Requirement C
PhysX	Yes, BSD	Yes	Yes
Bullet	Yes, zlib	Yes	Yes
Havok	No	Yes	Yes
Jolt	Yes, MIT	Yes	Yes
ODE	Yes, BSD	No, Missing Character Controllers	Yes
Newton Dynamics	Yes, zlib	No, Missing Scene Queries and Character Controllers	Yes

1. **PhysX**

   This physics engine is licensed with the BSD 3 license which allows developers to access and modify the source code when necessary as long as the copyright is maintained. PhysX supports all the physics features that have been previously chosen including reduced coordinate articulations and vehicle dynamics (NVIDIA,n.d.-a). PhysX is the default physics engine used in Unreal (Epic Games, n.d.-a) and Unity (Unity, n.d.-b) and has been used in a large number of games and real time applications.

1. **Bullet**

   This physics engine is licensed with the zlib license which allows for alteration of the source code as long as it is plainly stated as such. Bullet also supports the physics features previously chosen including reduced coordinate articulations and vehicle dynamics (Bullet, n.d). Bullet has been integrated into the Rockstar Advanced Game Engine which was used in "Grand Theft Auto 4" and "Red Dead Redemption" (Hardwidge,2011).

2. **Havok**

   Havok supports all of the chosen physics features including cloth and deformable objects. Havok has also been integrated into a large number of games including "Assassins Creed Odyssey" and "Beyond Two Souls". Unfortunately, Havok has special licensing requirements that must be purchased (Havok, n.d). Meaning that this does not fit the first requirement previously established.

3. **Jolt**

   This physics engine is licensed with the MIT license which allows for access and modification of the source code. Jolt supports the mentioned physics features that were chosen including vehicle dynamics. Jolt has been used in "Horizon Zero Dawn: Forbidden West" (Rouwe,2022).

4. **Open Dynamics Engine**

   The Open Dynamics Engine is licensed with the BSD license which allows for modification and redistribution of the code in binary and source code form as long as the notice is maintained. It has also been used in a number of games such as "Call of Juarez: The Cartel" and "X-Moto". Unfortunately, it does not support character controllers (Smith, n.d). Meaning that this physics engine does not meet the second established requirements.

5. **Newton Dynamics**

   The Newton Dynamics Engine is licensed with the zlib license which allows for alteration of the source code as long as it is plainly stated as such. It has also been used in a number of games such as "S.O.M.A" and "Amnesia: The Dark Descent". Unfortunately, it lacks a number of features we require such as Scene Queries and Character Controllers (Jerez, n.d). This means that it fails to meet the second requirement established.

We have narrowed down the possible selection of physics engines to three physics engines from the initial six. Bullet, PhysX, and Jolt all meet the requirements based on the three company needs. However, Rythe Interactive only needs one. Because of this, a new selection criterion must be established.

2.2 Physics Engine Performance Comparison

One aspect that has not been examined is performance. One of the intended selling points of the Rythe Engine is performance. This goal is what has driven a number of the decisions made within the company such as its use of the Entity Component System for its object model. Attempting this performance comparison within this project is beyond the scope of the paper since it requires the integration of all three physics engines and its necessary features. Instead, the performance data produced by previous works will be used.

In this regard, Jolt is unfortunately lacking any peer reviewed previous works that compare its performance with other physics engines. This is likely due to its very recent release at the time of writing. Because of this, we must immediately exclude it as a viable integration. On the other hand, there are other papers that examine the performance of Bullet and PhysX.

Gonzalez-Badillo (2014), examined the performance of Bullet and PhysX in the context of virtual assembly platforms which are tools that can be used to investigate form, fit, and function of a certain product before manufacturing real prototypes of that product. Their results show that PhysX is much more performant when simulating non convex models, while Bullet showed better performance under scenes that contain much more complex objects that require non convex meshes.

Hamono (2016), examined the performance of Bullet, PhysX, and the Open Dynamics Engine in the context of simulating the collapsing of a House under the effects of an earthquake. Their physics scene is a house-like structure consisting of a large number of rigidbodies and joints. Their results show that Bullet becomes much more performant after having more 5000 rigidbodies compared to PhysX. On the other hand, PhysX is consistently more performant when handling scenes with a large number of joints.

Erez (2015), examined the performance of Bullet, Havok, MuJoCo, ODE, and PhysX in the context of robotics. Their test scene involves singular complex entities that are chained together into one large chain of rigidbody joints. Their results for Bullet and PhysX are mixed. In their test scene where a robotic arm is grabbing a capsule PhsyX and Bullet performance is comparable. In another scene, where a pile of capsules are stacked together PhysX is about 16% faster. In a scene with a human ragdoll, Bullet is 20% faster. There is also a scene with one long joint chain where Bullet is 356% faster.

The existing research described above shows mixed results in the context of various types of physics scenes. However, Rythe Interactive is primarily interested in the performance of the physics engine in a game environment. This means that how reliable the aforementioned data from these papers are in predicting performance in a game environment depends on how close they are to a game environment.

The scene used by Erez (2015), is comparatively quite far from real game environments. Most of them involve having a singular complex physics entity that is connected together with a number of joints. This is quite far from most physics scenes of AAA games that may contain hundreds of physics objects.

Some of the scenes used by Gonzales-Badillo (2014) are a bit closer to actual game environments since some of them involve piles of non-convex objects. However, the number of entities used in their test scenes are still lacking compared to more larger games.

The best test scene that can give insight on game environments would be Harmono(2016)'s since it involves a large number of rigidbodies and joints.

As established previously Harmono's (2016) data suggest that Bullet and PhysX are quite similar in terms of rigid body dynamics performance. Bullet only becomes faster after having 5000 rigidbodies in the scene. On the performance of Joints, PhysX is consistently faster regardless of the number of joints in the scene. Because of this, we can reasonably conclude that choosing PhysX would be the preferable choice since the data we have suggests that using PhysX would allow consistently better joint performance and the benefits of Bullet would only become noticeable after a large number of rigidbodies.

3. Development

3.1 Development within the Rythe Engine

Before delving into the development of the aforementioned physics features prior to testing, it's important to establish some information regarding the Rythe Engine as this is the game engine in which the physics engine integration takes place.

The Rythe Engine uses the Entity Component System (ECS) for its object model. According to Härkönen (2019), ECS is an architectural pattern within software development that aims to manage entities in a large-scale real-time application.

The Rythe Engine currently already has a number of modules that will support the development of the physics engine. The first one is the OpenGL based renderer which was used to help visualize the physics scene. Next is the event system which can be used to subscribe to certain events that happen within the engine and to be able to broadcast those events. This will later on be important for when creating trigger events explained in section 3.5.2. Another important part of the engine is the math library. The Rythe Engine currently uses GLM as its math library. GLM is a mathematics library created based on the specifications of the OpenGL shading language and developed using C++ (g-truc,2020).

3.2 Physics Feature Prototyping Technique

Within this paper, there exists a need to find a way to be able to quickly review the viability of the integration of a certain physics feature. This is done using the sample scenes within the original PhysX repository as the foundation for the prototypes. This is helpful because we can directly interact with the API, use existing modules in the repository (basic rendering, existing examples of the use of the SDK, etc), and have direct access to the source code which allows easy debugging.

Prototyping within the sample scene is also efficient since the prototype created can then be used in the testing phase as established in section 4.1.

3.3 Modular Rigidbody Dynamics

In the context of the paper, modular rigidbody dynamics refers to the implementation of rigidbody dynamics in a way that can allow different physics engines to use the same interface. In the context of the Rythe Engine, a game engine that utilizes ECS, this means that there exists a “PhysicsComponent '' component and “Rigidbody '' component that can be used by different physics engines.

When dealing with rigidbody dynamics, there are a large number of supporting miscellaneous features that can be expected such as the ability to add impulses, setting object masses, instantiating colliders. Because of this, the author must prioritize which of these features should be worked on. The external assessor has advised that support for runtime modification of colliders should not be focused on since there is rarely any need for it within a game world.

One important requirement of this is that these components should not explicitly have middleware modules. For example, PhysX uses a PxActor in order to represent physics objects. A modular integration would require that the components are free of these kinds of objects that are specific to a certain library.

The way that this is implemented involves treating the abstract Rigibody and PhysicsComponents as “front-end” structures that are only responsible for notifying the currently active physics engine about modifications to the physics scene.

In the case of the abstract rigidbody, this is primarily done using the bitset defined within the C++ Standard Library. The idea here is when a specific parameter of the rigidbody is modified (mass, velocity, adding forces, etc) a specific bit within the bitset is set. The new value is also stored within the rigidbody. For example, when the mass of the rigidbody is modified the bitset with an index equal to the value of the enum rigidbody_flag::rb_mass is set.

Then, before the physics simulation is updated, the physics system would go through the rigidbodies within the scene and check the bitset. Each bit within the bitset can be mapped towards a certain function within the active physics engine. For example, if the physics engine notices that rigibody_flag::rb_mass has been set, the physics engine can respond with a specific function that modifies the mass. A class diagram of the RigidbodyData class is shown with Figure 2. A sequence diagram going through the process of setting a parameter within the rigidbody is shown under Appendices Section 8.1.

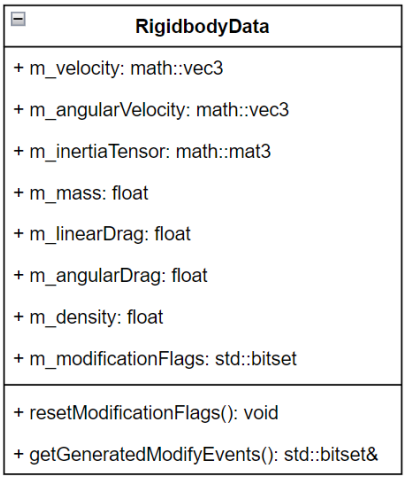


Figure 2. A UML Class Diagram depicting the variables and function of the RigidbodyData class.

The relationship between the physics component and the currently active physics engine draws some similarity with the rigidbodies relationship described above. Similar to the rigibody, the physics component also contains a bitset that is used to notify the physics engine of changes to the physics component. The physics component bitset is used when the user adds various colliders that can be added within the physics engine (cube,convex,spheres, etc). However, there are certain situations where this bitset is not enough. Especially when dealing with the modification of colliders. Before further delving into this issue, some information about how the abstract colliders are handled must be established.

The physics component has a std::vector of ColliderData. This contains the various convex shapes that the physics engine can support. These shapes need variables to support them. A sphere collider might only need one floating point value to represent its radius. A cube collider on the hand, might need three floating point values in order to represent its extents. In order to be able to store these structures within the same std::vector. ColliderData contains a union that can hold the specific values that a specific shape might need.

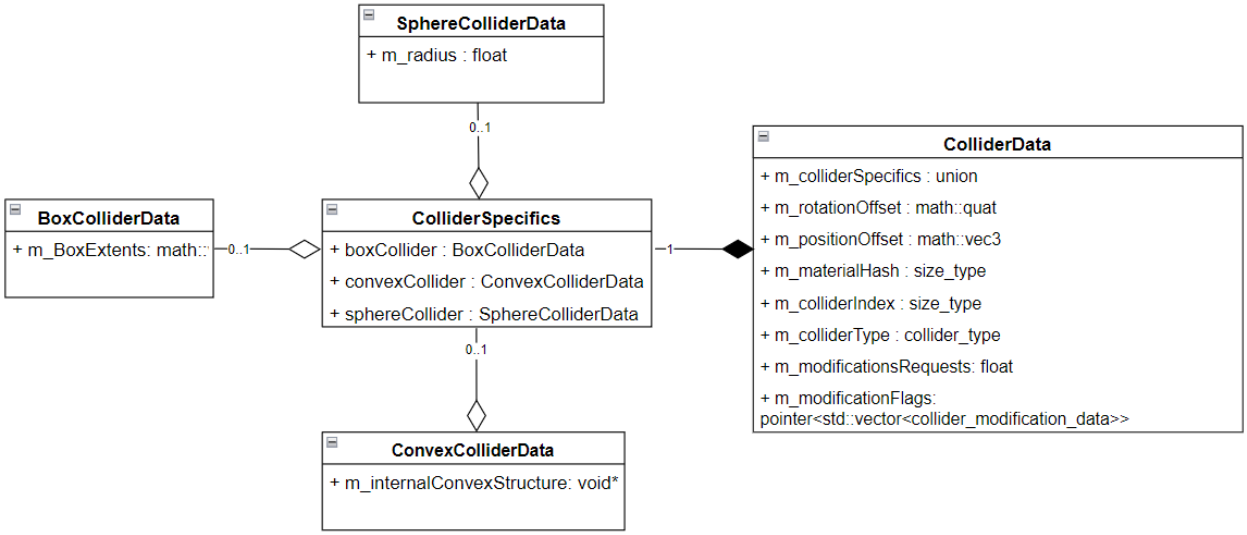


Figure 3. A UML Class Diagram depicting the relationship between the colliders and the ColliderData class.

The problem relating to bitsets can now be established. Using bitsets is not enough for the physics component since there are certain queries that need additional information. For example, if the physics material of a specific collider needs to be changed, we need to know the index of the collider within the ColliderData std::vector. An index is also needed when a specific collider needs to be removed. There are also other queries such as changing the extents of a box collider, and changing the radius of the sphere collider. In order to support these queries, the physicsComponent also has a std::vector of collider_modification_data. This structure contains the specific index of the collider that needs to be modified and the contents of the modification itself. In order to support storing different modifications in a single std::vector, a union is used.

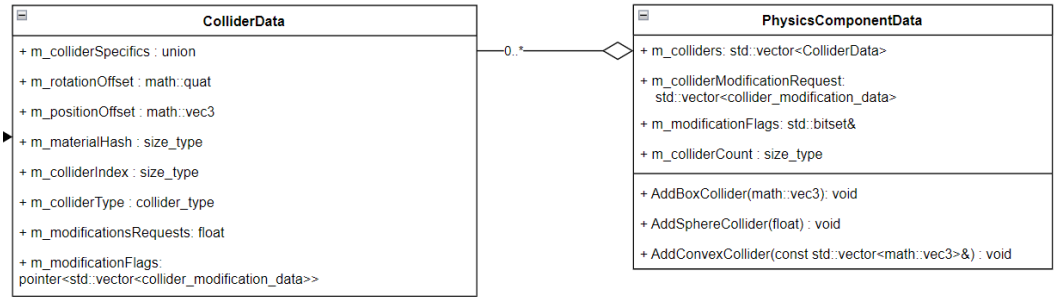


Figure 4. A UML Class Diagram depicting the relationship between the colliders and the ColliderData class.

3.4 Modular Character Controller

The goal of the modular character controller is to have some sort of “CharacterController” component that can be used by different physics engines. The way that this is achieved is relatively similar to how a modular physics component and rigidbody component is achieved; using std::bitset to notify the currently active physics engine.

There is, however, another challenge that must be overcome. The external assessor noted that character controllers are very game specific. In the sense that the necessary features of a character controller in one game can be quite different from another. Because of this, in the case of the character controller, it is

important that the user can at the very least select the desired features and modify the desired parameters.

First, the features that will be implemented must be decided. In order to do this, the sample scene within the PhysX repository can be examined. The character controller in this scene, other than the default collision resolution algorithm, implements two additional features: gravity and rigidbody feedback.

With the features decided, the next step is to give the user the ability to select these features. This can be achieved with a 'Controller Preset' module. The idea is that the 'character_controller' component will have a std::vector of controller_preset. The controller_preset contains the data necessary to implement some sort of feature for the character controller. For example, if the character controller needs to react to gravity, there could be a 'Gravity Preset' that contains the amount of gravity that is applied to the character controller and some form of accumulation. Then, if the character needs to react to rigidbodies there could be a 'Force Feedback Preset' that contains the force amount applied to bodies and a maximum mass value that can be used to limit the objects that can be pushed by the character controller. This is illustrated with the UML diagrams shown with Figure 5.

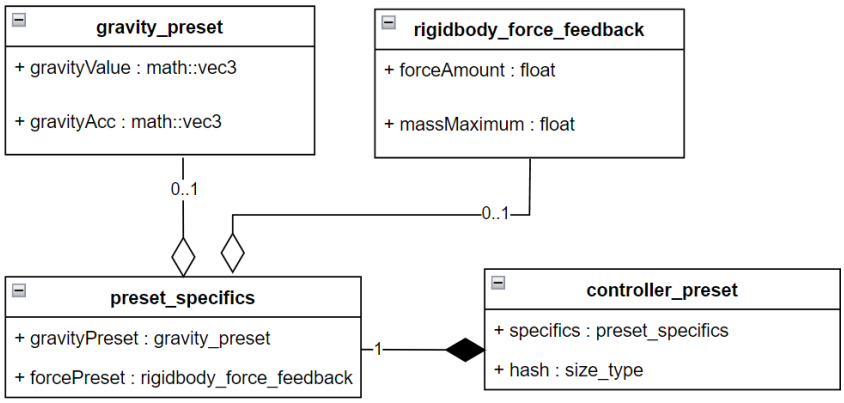


Figure 5. A UML Class Diagram depicting the relationship between the controller_preset class and the possible presets for the character controller.

As shown in Figure 5, the presets do not contain any logic. The idea is that these presets only indicate what the request from the character controller. How it is achieved is offloaded to whichever physics engine is currently active.

Within the PhysX integration, gravity is achieved by moving the character controller based on a gravity value. This value is accumulated and is only reset when the bottom part of the capsule is colliding with an object. PhysX notifies the system when this happens using the flag (PxControllerCollisionFlag::eCollisionDown) which can be raised when PxController::move is called.

Rigidbody force feedback is achieved through a hit callback from the PxController class. Everytime the character controller makes contact with a rigidbody, PxUserControllerHitReport::onShapeHit is called. When this happens, the shape is given to the system as a parameter. A force can be applied to this shape. The amount is decided by the rigidbody_force_feedback::forceAmount.

3.5 Modular Scene Queries

3.5.1 Collision Filtering

One important feature when implementing trigger shapes and scene queries is a collision filtering system. Collision filtering in the context of a physics engine refers to the act of configuring the collision detection behavior of the physics engine (NVIDIA,2021c). A collision filtering system can be used to exempt certain objects from colliding with one another or to cause them to raise an overlap event.

In this regard, the Rythe Engine's collision filtering system will take inspiration from Unreal's collision filtering system. As established by Golding (2014), Unreal's collision filtering system is object specific. Within this system each object can react in three different ways: ignore,overlap, and block. By ignoring, the objects can pass through each other, as if no collision has happened. By overlapping, the objects can pass through each other but an overlap event is triggered when this happens. By blocking, the objects cannot pass through each other.

The idea is that objects of the same type can interact differently with other objects, because each object will need to have its own filter data.

This system can be used to create some interesting phenomena where two objects of the same type can react differently depending on the needs of the user. An example of a use case of this is with implementing force fields. Assume a user of the Rythe Engine would like to create a force field that lets character controllers pass through but rigidbodies cannot. Because the collision filtering system is only specific to the object, the filter can be modified so that it ignores character controllers but blocks rigidbodies.

Achieving this collision filtering system within PhysX has two steps. The first step is to modify the default filtering callback within PhysX. This filtering callback known as PxSimulationFilterShader is a callback that can be set when a PxScene is created. This callback is called when PhysX finds a pair of objects that can potentially collide (NVIDIA,2021c). In order to achieve the desired filtering behavior, the new overridden callback executes the following steps. A snippet of this code can be found in the Appendices Section 8.2.

1. Identify how the first object reacts to the second object based on its filter data.
2. Identify how the second object reacts to the first object based on its filter data.
3. Using the two reactions above, use the lowest reaction. (ignore: 0, overlap: 1, block:2)
4. If the reaction is to block, send the pair to the collision solver
5. If the reaction is to overlap, it should trigger an overlap event
6. If the reaction is to ignore, the pair should be discarded.

The second step is to make sure that each shape within the simulation has the filter data needed. According to the documentation provided by NVIDIA (2021c), a shape within PhysX is represented with the PxShape class. Each shape contains an instance of PxFilterData. Which itself contains 4 32 bit unsigned integers (PxU32). These 4 PxU32s can be used to contain the information needed to implement the desired collision filtering system. The first PxU32 can be used to store how the object reacts to other objects depending on its type. The idea is that every 3 bits represents the three possible reactions (Ignore,Overlap,Block) for one object type. Currently 4 object types are supported: Static,Dynamic,Character Controller, and Terrain. The second PxU32 is used to store the owning shapes object type. The use of the filter data is illustrated with Figure 6.

	PxFilterData(Simulation)			
	Word0	Word1	Word2	Word3
Static(Block)	0	ObjectType	n/a	n/a
	0			
	1			
Dynamic(Overlap)	0			
	1			
	0			
Character Controller (Ignore)	**1**			
	0			
	0			
Terrain(Block)	0			
	0			
	1			

Figure 6. The contents of the PxFilterData class and how its bits are used to store the necessary information for the collision filter system.

3.5.2 Trigger Shapes and Collision Events

Trigger shapes allow a way for a user of a game engine to be notified when a collision between certain objects happens. This requires a number of collision events that the user can subscribe to in order to be notified. It was previously established how an object can become a trigger shape using the filtering system. This section describes how the event system is used in order to output collision events.

PhysX enables the possibility to create a custom callback for trigger shapes. The callback will receive the pairs of objects that are in contact with each other and the state of the contact itself (PxPairFlag::eNOTIFY_TOUCH_FOUND, PxPairFlag::eNOTIFY_TOUCH_PERSISTS, and PxPairFlag:: eNOTIFY_TOUCH_LOST). Using these enums provided by the callback each enum can then be mapped to the raise a specific event (on_trigger_enter, on_trigger_stay, and on_trigger_exit respectively).

3.5.3 Character Controllers and Collision Filtering

A notable issue that was found through the prototyping phase is how PxSimulationFilterShader does not get called when the character controller is moved. It was found that this is due to the fact that character controller collision detection is handled differently compared to the collision detection for other objects within the scene. This means that the filters that were created would be ignored for the character controller.

In order to fix this and get the desired collision filtering behavior, a sweep using NpSceneQueries::sweep is performed every time the character moves. This sweep can be overridden to ignore certain objects and collide with others. With this, the desired collision filtering is achieved

Using the PxSweepCallback the system receives all the objects that collided/overlapped with the swept character controller. With this information, the respective trigger events can be called when necessary.

3.5.4 Raycasting , Overlaps, and Sweeps

Within the collision filtering system used for trigger events, the system filters objects based on type (Dynamic, Static, Character Controller, etc). However, as established by the example given by Golding (2014), this may not be the ideal way to differentiate objects when doing scene queries (raycasting,overlaps, and sweeps). Golding (2014), examines the difference between a “visibility” raycast and a “weapon” raycast. In a scene within a game where there exists a bulletproof glass wall, a shrub, and a brick wall, a user may want the weapon raycast to ignore the shrub and get blocked by the brick wall and the bulletproof glass wall. The visibility raycast on the other hand should get blocked by the shrub and brick wall but should be able to ignore the bulletproof wall. In this situation it might be noticed that these three objects would be considered ‘static’ within the game scene. Because of this, a different method of differentiating objects can be used.

The author instead takes inspiration from Unity’s method of differentiating objects for scene queries. According to Unity (n.d.-c), a layer mask is used. The layer mask is an integer within each game object in Unity. This layer mask is also a parameter that can be given to scene query related functions. The idea is that a user can query for objects with a specific mask by performing bit shifts to the layer mask parameter of the scene query function.

Within the Rythe Engine, the scene query functions (raycasting, sweep, and overlap) are given through a static class called “PhysicsHelpers”. Doing this allows a user to call these functions from anywhere as long as the data for the parameters are available. This is also the way that Unreal (Epic Games,n.d-b) and Unity (Unity,n.d.-c) provide access to scene query functions. Calling these functions calls a delegate that can be set by whichever physics engine is currently active. In the case of PhysX, PhysX would then call one of the following functions : NpSceneQueries::raycast, NpSceneQueries::overlap, or NpSceneQueries::sweep.

In order to allow the user to set masks for scene queries the ‘QueryMask’ class is created. The idea is that a user can set how an object reacts to a certain mask by calling the function QueryMask::setReactionToMaskIndex which takes in the mask index and how the query reacts to that mask index (ignore,overlap, or block). By default, physics components will not have a mask set, a user can choose how the query reacts to such objects using QueryMask::setReactionToDefaultMask. An instance of QueryMask is then passed as a parameter for the respective scene query function.

In order for the data in QueryMask to be used there are two things that must be done. First, the method used to filter out objects in scene queries must be overridden. This can be done by creating a class derived from PxQueryFilterCallback. This class has a function called PxQueryFilterCallback::preFilter which gets called before collision detection for a potentially colliding object happens. It also allows a user to filter out objects. In order to implement the desired filtering behavior, the preFilter function uses the layer mask of the ray that we get from QueryMask and the mask of the object which is stored within the physicsComponentData instance of the object to determine if the object should be filtered out or not. The implementation of this can be found under Appendices Section 8.3.

The second thing that must be done is to pass the mask information to the PxFilterData instances. This is necessary because it is not possible to directly pass an instance of QueryMask or the physicsComponentData since the parameters of the preFilter function only contains the PxFilterData instances of the query and the potentially colliding object. The overridden PxFilterData for the potentially colliding object would contain its mask, while the overridden PxFilterData that represents the QueryMask

would contain how it reacts to each mask and its reaction to an object without a mask. This is represented with Figure 7.

PxFilterData(Potentially Colliding Object)			
Word0	Word1	Word2	Word3
NumericLimit	NumericLimit	Mask	n/a

	PxFilterData(QueryMask)			
	Word0	Word1	Word2	Word3
Mask 0 (Ignore)	0 0	ReactionToDefault	n/a	n/a
Mask 1 (Overlap)	**1** 0			
Mask 2 (Block)	0 **1**			
Mask 3 (Ignore)	0 0			
Mask 4 (Block)	0 **1**			
Mask 5 (Ignore)	0 0			

Figure 7. The PxFilterData of a shape that is considered a potentially colliding object (left) and the PxFilterData passed as a parameter of PxQueryFilterCallback::preFilter representing the QueryMask of the scene query.

As shown in Figure 7, the filter data of the potentially colliding object has the mask of the object but also has its first 2 unsigned integers (word0 and word1) set to the numeric limit. This is done to make sure that the internal AND operator that happens within the SDK between the 2 filter data passes.

Figure 7 also shows how to represent the way that the query reacts to certain masks (ignore,overlap,and block). The figure shows that 2 bits are used to represent the three possible reactions. This is done so that more masks can be supported while only using one of the integers.

3.6 Miscellaneous Features

In order to support debugging while implementing the integration of PhysX the PhysX Visual Debugger (PVD) was implemented by the author. PVD is an application that can be used to debug, visualize, and interact with a PhysX scene (NVIDIA, 2021d). This application can be used for a number of tasks such as visualizing collisions with a physics scene, profiling memory and performance, and examining the properties of a physics object.

When the PhysX SDK is used within the Rythe Engine. The Rythe Engine can be configured to tell the SDK to output a pxd2 file when the engine stops running. This pxd2 file can then be opened using the PVD application. Within the application, a user would be able to view the objects within the physics simulation at every step of the simulation. The user can also examine the properties of each object. Figure 8 shows the user interface of the PVD application.

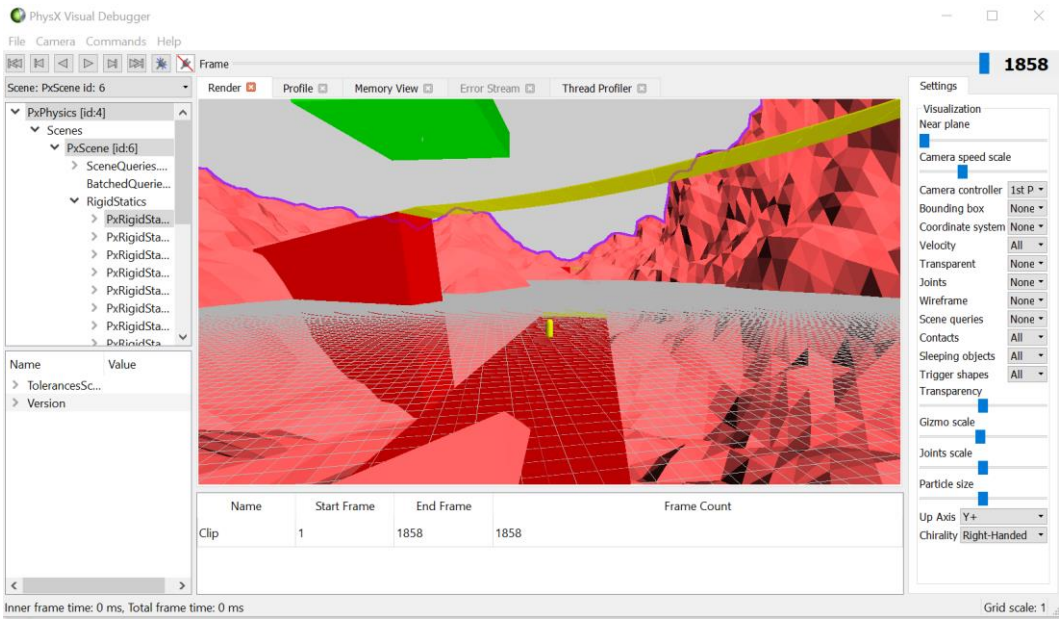


Figure 8. A snapshot of the PhysX Visual Debugger, showing a recording of one of the sample scenes within the PhysX repository.

4. Testing And Discussion

The modularity of the implemented physics features have been established within the Development section. The goal of this testing section is to ensure that the integration of the specific features are done properly.

4.1 Test Method

Each physics feature will undergo a similar test: A comparison between the specific sample scene from the original PhysX repository that shows a demo of the physics feature and the same scene reproduced within the Rythe Engine using the created PhysX integration.

This test is aided with the PhysX Visual Debugger (PVD). As established previously, the debugger will allow the examination of a physics scene over the course of a certain period of time. It also allows a user to examine the properties of a physics object such as its pose, velocity, mass, etc.

PVD will be run on both the specific sample scene and the replicated version created within the Rythe Engine. The resulting pxd2 files from running these two programs will then be examined. Each scene will have a set of frames where 'crucial' events are taking place. This can be the collision of 2 specific objects, an object entering a specific location of the test scene, etc.

The test is considered to have a positive outcome when the scene produced within Rythe is similar to the scene produced within the sample scene from the PhysX repository. In order to empirically decide this the methods presented by Toll (2012) are used. Toll (2012), presents four strategies in which a certain simulation within computer games can be unit tested. However, only two out of these four strategies will be used: The **Intuitive Imprecise Test** and the **Exact Tests Calculated by a Second Implementation**.

The **Intuitive Imprecise Test** refers to the idea of testing based on behavior instead using precise numerical values. The paper uses the example of a ball colliding with a plane. In the Intuitive Imprecise Test, the test is considered successful if the resulting position of the ball is above the plane. On the other hand, the **Exact Tests Calculated by a Second Implementation** is executed by using another implementation of the simulation to test the first one. In the ball and plane example, there would be 2 implementations of the simulation, where one is used to test the other. Each physics feature will be tested using one of these strategies. Which strategy is used and the justification for using it is further elaborated below.

4.2 Rigidbody Dynamics

The sample scene used to test Rigidbody Dynamics is called "SnippetHelloGRB'' within the PhysX repository. This test scene contains a large sphere collider, and 40 sets of cube collider stacks, and a flat plane. The cube colliders are stacked into a triangle stack with 20 cubes at its base. An image of this scene is displayed with Figure 9.

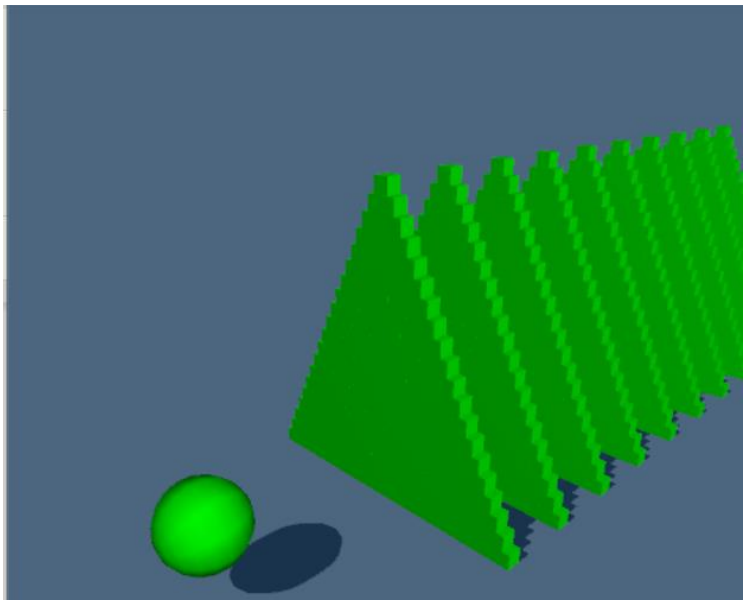


Figure 9. A snapshot of the “SnippetHelloGRB'' scene within the PhysX repository.

In this test scene, we examine the position and orientation of the ball as it makes contact with the other objects within the scene such as the cube stacks and the ground plane. This examination is only up to the 10th stack in order to shorten the pvd recording. Because of this the original scene was modified so that it contained only 10 cube stacks instead of the initial 40.

4.2.1 Precision Threshold and Testing Method

‘SnippetHelloGRB’ does not depend on any user/timed input. Everything is set before the simulation. PhysX, as established previously, is also deterministic. Because of this, it's reasonable to assume that the scenes produced would be near identical.

The **Exact Tests Calculated by a Second Implementation** strategy will be used for this scene. In this specific scenario, the second implementation would be the PhysX repository scene.

The precision threshold will be set to 2 decimal points to be considered identical. Although the choice of this relatively low threshold is mostly arbitrary, this threshold was chosen due to the idea that the goal of game physics is believability instead of accuracy (Bergen,2010).

4.2.2 Rigidbody Dynamics Test Results

Table 3 is the position of the ball in the original sample scene known as “SnippetHelloGRB”. The original pxd2 file can be found under the Appendices Section 9.4.

Table 3

The position and orientation of the ball in the "SnippetHelloGRB" scene at certain points of the scene.

Ground Truth			
Event	**Frame**	**Ball Position**	**Ball Orientation**
Ball Bounces on Ground	29	[0 , 5.38 , 44.49]	[-0.12 , 0 , 0 , 0.99]
Ball collision with 1st stack	56	[0 , 13.78 , 3.67]	[-0.15 , 0 , 0 , -0.99]
Ball collision with 3rd stack	69	[0 , 16.8 , -15.88]	[0.83 , 0 , 0 , -0.56]
Ball collision with 5th stack	84	[0, 19.5 , -38.28]	[0.87, 0 , 0 , 0.5]
Ball collision with 7th stack	96	[0, 20.96 , -56.09]	[0.27, 0 , 0 , 0.96]
Ball collision with 9th stack	108	[0.01, 21.9 , -73.8]	[-0.4 , 0 , 0 , 0.91]
Final Position after 10th stack	125	[0.04, 22.3 , -98.7]	[-0.95, 0 , 0 , 0.32]

Table 4 shows the results from the pvd file created from the replicated scene within the Rythe Engine. The original pxd2 file can be found under the Appendices Section 9.4.

Table 4

The second test result shows the position and orientation of the ball in the "SnippetHelloGRB" scene recreated within the Rythe Engine.

Rythe Engine Test					
Event	**Frame**	**Ball Position**	**Ball Orientation**	**Position Deviation**	**Orientation Deviation**
Ball Bounces on Ground	29	[0, 5.38 , 44.49]	[-0.12, 0 , 0 , 0.99]	0	0
Ball collision with 1st stack	56	[0, 13.78 , 3.67]	[-0.15, 0 , 0 , -0.99]	0	0
Ball collision with 3rd stack	69	[0, 16.8 ,-15.88]	[0.83, 0 , 0 ,-0.56]	0	0
Ball collision with 5th stack	84	[0, 19.46, -38.28]	[0.87, 0 , 0 , 0.5]	0.04	0
Ball collision with 7th stack	96	[0.01, 20.95 , -56.08]	[0.27, 0 , 0 , 0.96]	0.01732	0
Ball collision with 9th stack	108	[0.02, 21.89, -73.77]	[-0.4, 0 , 0 , 0.91]	0.03317	0
Final Position after 10th stack	125	[0.04 , 22.27, -98.68]	[-0.95, 0 , 0 , 0.32]	0.03605	0

4.2.3 Rigidbody Dynamics Test Discussion

The result of the test shows some deviation can be found after the ball collides with the 5th stack and onwards where the position deviates up to 0.04. Investigation of the Rythe scene reveals that there are some minor differences with the mass of the ball within the Rythe Integration and the PhysX scene. The ball in the Rythe Scene has a mass of 525398.812 while the sample scene ball has a mass of 525398.719.

This error is likely due to the floating-point math used when converting from mass to density. This issue can be solved by having a way to directly set density within the Rythe Integration. However, it's important to note that the difference is arguably minimal. It is more desirable to allocate time into the other physics features.

4.3 Character Controller

There are 3 features that need to be tested: slide and collide, gravity, and interaction between rigidbodies.

For the testing of character controllers, it was found that it would be preferable to create a custom scene instead of replicating an already built scene from the PhysX repository. This is because the scene given by PhysX contains physics features that are not implemented yet such as joints and height fields.

4.3.1 Creating the Capsule Controller Test Scene

Each object within the created scene is placed in order to test one of the three aforementioned features. Figure 10 shows the scene taken from the recording of the PhysX PVD Debugger, followed by a description containing the reasoning behind placing those objects and the expected behavior based on running the scene within the PhysX repository.

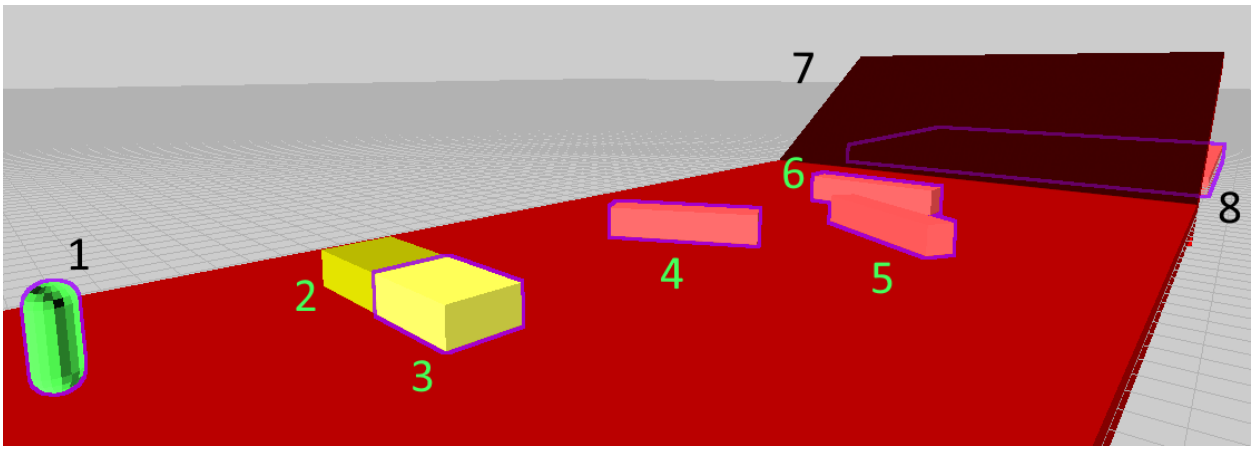


Figure 10. A snapshot of the PVD recording of the scene used to test the Capsule Controller

The objects marked as 2 and 3 are rigidbodies. The expected result is that the character controller marked as 1 will push the rigidbodies away. The objects marked with 4 and 5 are static colliders that are rotated on the y axis. The character controller will collide and slide through them. The object marked as 6 is also a static collider. It is placed in a direction perpendicular towards the direction the movement of the character controller. The character would stop completely when hitting this object. A hard coded jump command will then be executed so that the character can proceed. It will then go through the inclined platform marked as 7. The character would climb the platform. After reaching the top edge of the platform, the character would then fall down to the flat platform marked as 8.

4.3.2 Precision Threshold and Testing Method

The precision that was expected in the rigidbody dynamics scene should not be expected for this scene. The reason for this is due to the fact that the character controller within the Rythe Engine is moved using non fixed updates, since the move commands are called from a separate system which simulates how a

user would interact with the physics interface. While the test scene created within the PhysX repository is called with a time step consistent with the advancement of the simulation.

For this scene, the **Intuitive Imprecise Test** strategy will be used. In the context of this scene, success is achieved if the expected behavior established at section 4.3.1 is achieved within the Rythe Engine test scene.

4.3.3 Capsule Controller Test Results

Videos showing the comparison of the PVD recordings produced by the PhysX repository scene and the Rythe Engine scene is shown in the Appendices Section 9.6. The original pxd2 file can be in the Appendices Section 9.4. A summary results of the character controller test is shown with table 5.

Table 5

A summary of the expected behaviors of the capsule controller scene and whether the behavior was achieved within the Rythe Engine scene.

Capsule Controller Test	
Expected Behavior	**Result**
Rigidbodies moved when in contact with the capsule	Behaviour Achieved
The capsule slides through static collider 4 and 5	Behaviour Achieved
The capsule is stopped by static collider 6	Behaviour Achieved
The capsule jumps over static collider 6	Behaviour Achieved
The capsule climbs inclined platform number 7	Behaviour Achieved
The capsule falls on to flat platform 8	Behaviour Achieved

4.3.4 Capsule Controller Test Discussion

Comparing the scene from the PhysX repository and the scene recreated within the Rythe Engine, it can be seen that the expected behavior has been achieved. The rigid bodies are moved out of the way when the character controller makes contact with them. The character controller goes through the static collider 2 and 3 by sliding through the rotated colliders and stopping when making contact with collider 4. After jumping above collider 4 the character is then able to climb the platform marked as 5. It then lands on the final flat platform marked as 6. In summary, out of the 6 expected behaviors, all 6 of them were achieved within the Rythe Engine scene.

It is important to note that the final position of the rigid bodies after being pushed by the character can be visually seen to be different. Further examination shows that final position and rotation of the

rigidbodies in the PhysX repository is [5.79, 1 , -8.851] [0.63, 1 , -7.91] and [0, 0.23 , 0 , 0.97] [0, -0.56 , 0 , 0.83] respectively while the final position and rotation of the rigidbodies in the Rythe Engine scene is [5.34 , 1 , -9.05] [0.92 , 1 , -8.03] and [0, 0.36 , 0 , 0.93] [0, -0.5, 0 , 0.87]. This is likely due to the fact that the movement commands of the character controller are less consistent causing the amount of feedback force applied to the bodies to defer. This shouldn't be regarded as a significant issue since the final behavior of pushing rigidbodies has been achieved.

4.4 Scene Queries

There are 2 tests that need to be done for scene queries. The first test is focused on collision filtering and trigger events. The second on the other hand is focused on testing scene queries (raycast,overlap,and sweep).

Custom scenes will be made for both tests. This is due to the fact that some of the features being tested (the custom collision filtering system and the mask system) are custom features not found within the PhysX repository.

4.4.1 Creating the Scene Query Test scenes

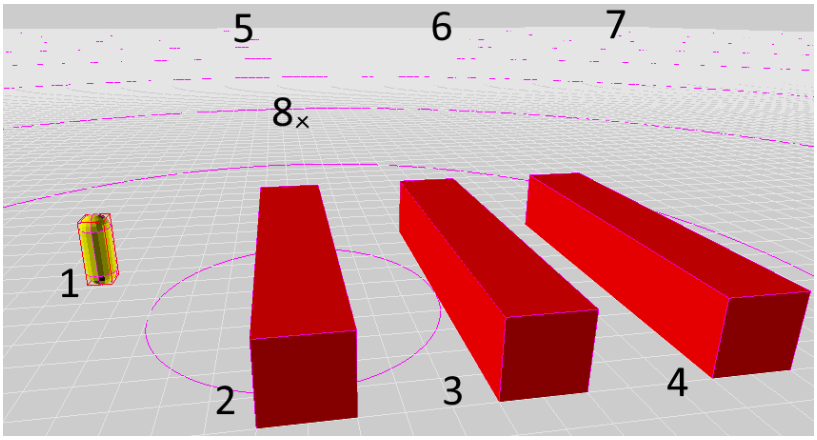


Figure 12. A snapshot of the PVD recording created by running the collision filtering test scene within the Rythe Engine.

The first test scene shown with Figure 9 shows the test scene recorded using PVD. There are 4 objects within this scene. Object 1 is a capsule character controller. It will move towards objects 2,3 and 4. Object 4 is a collider that reacts to dynamic objects and character controllers by blocking them. Object 3 is a collider that reacts to dynamic objects and character controllers by overlapping them. The first time a dynamic object overlaps with object 3, 2 pairs of dynamic sphere objects will spawn above object 2 and 4 (These locations are marked with 5 and 7). For every step that a dynamic object has entered object 3 but has not exited it, a line of dynamic sphere colliders are created starting at the location marked as 8. Finally, Object 2 reacts to character controllers by overlapping them but reacts to dynamic objects by blocking them. When a character controller enters object 2 for the first time a sphere collider is spawned above object 3 (this location is marked with 6).

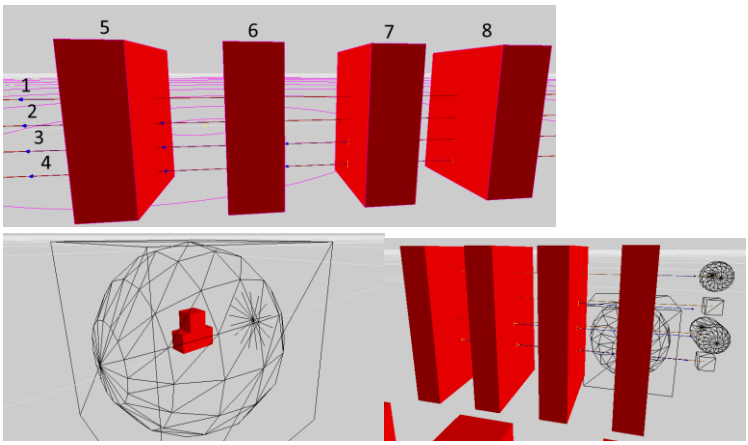


Figure 13. Snapshot of the PVD recording created by running the scene query test scene within the Rythe Engine.

The second test scene shown with Figure 12 shows a scene recorded using PVD. The image on the top left shows 4 cuboid shaped objects and 4 raycasts. Cuboid objects 5,6,7,and 8 use the masks 0,1,2, and default respectively. The ray marked as 1 is a default ray that is blocked by any object. The ray marked as 2 is set to overlap objects with mask 0 but become blocked by objects with mask 1. The ray marked as 3 is set to overlap objects with mask 0 and 1 but become blocked by objects with mask 2. The ray marked as 4 is not blocked by anything and overlaps all objects.

The image on the bottom right shows another 4 cuboids and 4 sweeps. Similar to the behavior of the raycast test, the four cuboids are given different masks and the sweeps are modified to filter different masks.

The image on the bottom left shows 3 blocks. Within this image there are 4 overlap queries happening, 2 using a cube as the overlap shape and the other 2 using a sphere. The overlaps are assigned to search for specific masks. Each of the three blocks are then assigned with different masks (0,1, and 2).

4.4.2 Precision Threshold and Testing Method

Similar to the character controller test, the **Intuitive Imprecise Test** strategy will be used to test the scenes. This is due to the fact that the main concern in these scenes are behavior related (do the collision events get triggered, does the collision filter work, etc). In the context of the scenes, this means that success is achieved if the behavior described in section 4.4.1 is achieved within the Rythe Engine.

For the second scene related to scene queries, the logs showing the results of the ray will also be used since it summarizes the results of the ray without any need of visual examination of the PVD scenes.

4.4.3 Scene Query Test Results

Videos showing the PVD recordings produced by the PhysX repository scene and the Rythe Engine scene are shown in the Appendices Section 9.7. The resulting logging information showing the results of the scene query test in the PhysX repository and the Rythe Engine scene is the Appendices Section 9.8. The original pxd2 file can be found under the Appendices Section 9.4. A summary of the results of the collision filtering test and the scene query test is shown with Table 6 and 7 respectively.

Table 6

A summary of the expected behaviors of the collision filtering test and whether the behavior was achieved within the Rythe Engine scene.

Collision Filtering Test	
Expected Behavior	**Result**
Controller overlaps object 2	Behaviour Achieved
When the controller overlaps object 2, a sphere is spawned on location 6	Behaviour Achieved
Controller overlaps object 3	Behaviour Achieved
When a sphere exits object 3, spheres are spawned on location 5 and 7	Behaviour Achieved
While the sphere spawned on location 6 collides with object 2 with a non zero velocity, a line of spheres is spawned	Behaviour Achieved
The spheres spawned above object 2 and 4 are blocked	Behaviour Achieved
Controller is blocked by object 4	Behaviour Achieved

Table 7

A summary of the expected behaviors of the scene query test and whether the behavior was achieved within the Rythe Engine scene.

Raycast Test	
Expected Behavior	**Result**
The default raycast is blocked the first object it hits	Behaviour Achieved
Raycast can be set to overlap objects with mask 0 but become blocked by objects with mask 1 and 2	Behaviour Achieved
Raycast can be set to overlap objects with mask 0 and 1 but become blocked by objects with mask 2	Behaviour Achieved
Raycast can be set to overlap all objects	Behaviour Achieved

Sweep Test	
Expected Behavior	**Result**
The default sweep is blocked by first object it interacts with	Behaviour Achieved
Sweep can be set to overlap objects with mask 0 but become blocked by objects with mask 1 and 2	Behaviour Achieved
Sweep can be set to overlap objects with mask 0 and 1 but become blocked by objects with mask 2	Behaviour Achieved
Sweep can be set to overlap all objects	Behaviour Achieved

Overlap Test	
Expected Behavior	**Result**
Overlap can be set to only find objects with mask 0	Behaviour Achieved
Overlap can be set to only find objects with mask 1	Behaviour Achieved
Overlap can be set to only find objects with mask 2	Behaviour Achieved
By default, overlap finds all objects regardless of its mask	Behaviour Achieved

4.4.4 Scene Query Test Discussion

The collision filtering scene shows the expected behavior explained in section 4.4.1. All the events that need to be triggered (i.e., the trigger enter event on the character controller, the trigger exit event causing the spawning of spheres above object 2 and 4, and the trigger stay event that causes a line of sphere colliders to spawn).

One important difference to note is that the number of colliders spawned on trigger stay is different from the original scene. The original scene spawns 159 colliders which means that the trigger stay event was triggered 159 times. The Rythe scene spawns 149 colliders, meaning that the trigger stay event was triggered 149 times. This is likely due to the fact that the Rythe scene capsule controller is moved at a non-fixed update causing it to exit the trigger at a different time.

The scene query also shows the expected behavior explained in section 4.4.1. The raycast executed are blocked by the correct objects and register the overlaps correctly. The same can be said for the sweeps and overlaps. It should be noted that the pvd recording of the Rythe Engine does show that the queries flicker at each time step, this is likely due to the fact these queries are called from a non-fixed update and should not be regarded as an issue.

In summary, the collision filtering scene achieved all 7 expected behaviors and the scene query test achieved all 12 of the expected behaviors.

5. Post-Testing Development

5.1 Implementation of the Terrain Component

Through attempting to replicate the test scene for the rigidbody dynamics test, it was found that support for adding flat planes was not yet implemented. Within the Rythe Engine, this was done by introducing a physics_enviroment component. The idea is that this component could later on be used to support the addition of other terrain objects such as height fields. The component works in a similar way to the physics_component by storing updates to its state in a std::bitset.

5.2 Setting Masses

Initial Testing for Rigidbody Dynamics testing revealed that the inertia tensors of the objects within the Rythe Engine scene were not set properly causing the application to not be able to correctly reproduce the PhysX repository scene. It was found that this is due to the fact that PxRigidbody::setMass was used instead of PxRigidBodyExt::updateMassAndInertia which causes the inertia tensor to not update correctly. PxRigidBodyExt::updateMassAndInertia takes in the new density of an object in order to set mass and inertia. In order to get this new density value, we use the old density of the object and multiply it with the ratio of the new mass and the old mass.

6. Conclusion and Future Work

The goal of this paper is to achieve a modular integration of rigidbody dynamics, character controllers, and scene queries within the Rythe Engine.

Modular Rigidbody dynamics was achieved using bitsets and stored commands in order to pass queries to the physics scene. The rigidbody integration was then tested using the **Exact Test Calculated by a Second Implementation** strategy. The final integration is able to produce a scene with the biggest position deviation of 0.04 and no rotational deviation from the original PhysX scene.

Modular character controllers use the same concept but also add the idea of “controller presets'' that can be added or removed in order to add features to the character controller. The character controller integration was tested using the **Intuitive Imprecise Test** strategy. Out of the 6 established behaviors of the test, all of them were achieved within the Rythe Engine scene.

Modular scene queries were achieved by using delegates that can be set by the active physics engine and using the event system within the core engine that can be used to raise physics events. The integration also adds a collision filtering system that can filter objects by type. The scene query integration was tested using the **Intuitive Imprecise test** strategy. Out of the 12 established behaviors of the test, all of them were achieved within the Rythe Engine scene.

The work done within this paper has resulted in the Rythe Engine becoming closer to having comparable physics capabilities of the popular game engines such as Unity and Unreal. It was also accomplished in a modular way which is in line with the selling point of the engine which is its modularity. This modular integration will also benefit the engine in the future if a switch in the physics engine is necessary later on.

Future work can go into a number of different directions. It would be important to also test this abstraction layer with other physics engines as a way to improve and test it. Another aspect that was not thoroughly examined is the memory and performance impact of the abstraction layer towards the runtime of the engine. It would also be important to integrate the other physics features such as joints, vehicles, cloth, etc.

7. Sources

Bishop, T. A., & Karne, R. K. (2003, March). A Survey of Middleware. In *Computers and Their Applications* (pp. 254-258).

Coumans, E. (n.d.). *Bulletphysics/bullet3:Bullet physics SDK: Real-time collision detection and multi-physics simulation for VR, games, visual effects, robotics, machine learning etc..* GitHub. Retrieved April 3, 2022, from https://github.com/bulletphysics/bullet3

Dickinson, C. (2013). *Learning game physics with Bullet Physics and OpenGL*. Packt Publishing Ltd.
Epic Games. (n.d.-a). *Physics*. Unreal Engine Documentation. Retrieved February 25, 2022, from https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Physics/

Epic Games. (n.d.-b). *UWorld::LineTraceSingleByChannel*. Unreal Engine Documentation. Retrieved May 2,2022, From
https://docs.unrealengine.com/4.26/en-US/API/Runtime/Engine/Engine/UWorld/LineTraceSingleByChannel/

Erez, T., Tassa, Y., & Todorov, E. (2015, May). Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *2015 IEEE international conference on robotics and automation (ICRA)* (pp. 4397-4404). IEEE.

Erikstad, S. O. (2019). Design for modularity. In *A holistic approach to ship design* (pp. 329-356). Springer, Cham.

Golding, J. (2014, April 17). *Collision filtering in Unreal engine 4*. Unreal Engine. Retrieved May 9, 2022, from https://www.unrealengine.com/en-US/blog/collision-filtering

Gonzalez-Badillo, G., Medellin-Castillo, H. I., Lim, T., Ritchie, J. M., Sung, R. C., & Garbaya, S. (2014). A new methodology to evaluate the performance of physics simulation engines in haptic virtual assembly. *Assembly Automation*.

g-truc. (n.d.). *g-TRUC/GLM: OpenGL Mathematics (GLM)*. GitHub. Retrieved May 25, 2022, from https://github.com/g-truc/glm

Hamano, T., Onosato, M., & Tanaka, F. (2016, October). Performance comparison of physics engines to accelerate house-collapsing simulations. In *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)* (pp. 358-363). IEEE.

Hardwidge, B. (2011, February 17). *AMD talks GPU gaming physics*. bit. Retrieved March 5, 2022, from https://www.bit-tech.net/reviews/tech/graphics/amd-manju-hegde-gaming-physics/3/

Härkönen, T. (2019). Advantages and Implementation of Entity-Component-Systems.

Havok. (n.d.). *Havok-Powered*. Havok. Retrieved March 11, 2022, from https://www.havok.com/havok-powered/

Jerez, J., & Suero, A. (n.d.). *About Newton Dynamics*. Newton Dynamics: Games using Newton. Retrieved April 27, 2022, from http://newtondynamics.com/forum/games.php

NVIDIA. (2021a). *Joints*. Joints - NVIDIA PhysX SDK 4.1 Documentation. Retrieved April 13, 2022, from https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Manual/Joints.html

NVIDIA. (2021b). *Scene queries*. Scene Queries - NVIDIA PhysX SDK 4.1 Documentation. Retrieved April 13, 2022, from https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Manual/SceneQueries.html

NVIDIA. (2021c). *Rigid body collision*. Rigid Body Collision - NVIDIA PhysX SDK 4.1 Documentation. Retrieved April 13, 2022, from https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Manual/RigidBodyCollision.html

NVIDIA. (2021d). *PhysX visual debugger*. NVIDIA Developer. Retrieved June 3, 2022, from https://developer.nvidia.com/physx-visual-debugger

NVIDIA. (n.d.). *NVIDIA PhysX 4.5 and 5.0 SDK*. NVIDIA Developer. Retrieved April 13, 2022, from https://developer.nvidia.com/physx-sdk

Presser, R. (2022, February 11). *Why is gaming platform steam at the heart of a growing Web3 Rift?* The Business of Business. Retrieved April 27, 2022, from https://www.businessofbusiness.com/articles/what-is-steam-a-gaming-platform-at-the-heart-of-a-growing-web3-rift/

Pytlos, M., Gilbert, M., & Smith, C. C. (2015). Modeling granular soil behavior using a physics engine. *Géotechnique Letters*, *5*(4), 243-249.

Rouwe, J. (2022). *Jrouwe/joltphysics*. GitHub. Retrieved April 7, 2022, from https://github.com/jrouwe/JoltPhysics

Rythe Interactive. (2020). *Rythe Interactive*. github. https://github.com/Rythe-Interactive

Sim, Z. H., Chook, Y., Hakim, M. A., Lim, W. N., & Yap, K. M. (2019, October). Design of virtual reality simulation-based safety training workshop. In *2019 IEEE International Symposium on Haptic, Audio and Visual Environments and Games (HAVE)* (pp. 1-6). IEEE.

Smith, R. (n.d.). *Products that use ode*. ODE. Retrieved April 27, 2022, from http://ode.org/wiki/index.php/Products_that_use_ODE

Toftedahl, M. (2019, September 30). *Which are the most commonly used game engines?* Game Developer. Retrieved April 5, 2022, from https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines-

Toll, D., & Olsson, T. (2012, March). Why is unit-testing in computer games difficult?. In *2012 16th European Conference on Software Maintenance and Reengineering* (pp. 373-378). IEEE.

Unity. (n.d.-a). *Character controller component reference*. Unity. Retrieved April 27, 2022, from https://docs.unity3d.com/Manual/class-CharacterController.html

Unity. (n.d.-b). *Physics*. Unity. Retrieved February 25, 2022, from https://docs.unity3d.com/Manual/PhysicsSection.html

Unity. (n.d.-c). *Physics*. Unity. Retrieved May 16, 2022, from https://docs.unity3d.com/ScriptReference/Physics.html

van den Bergen, G., & Gregorius, D. (Eds.). (2010). *Game physics pearls*. Natick: AK Peters.

Zuvich, T. (2000). Vehicle dynamics for racing games. In *Game Developers Conference Proc*.

8. Reflection

The reflection is based on how I think I scored based on the competencies of the graduation. They are listed and described below.

1. Technical Research and Analysis

Good. In order to implement the PhysX Integration I used my understanding of how a physics engine is used in a game, I researched and used the features within the standard library to implement the features. I also at times needed to look through the PhysX codebase in order to debug issues related to scene queries. I then used this knowledge to successfully integrate the necessary features within the codebase of the Rythe Engine.

I think one way this area could have been improved is if the code review process was done for all of the code implemented within this project. Since this could have better revealed flaws within my technical understanding.

2. Designing and prototyping

Excellent. Each physics feature was tested first within the PhysX repository, so each physics feature had 2 iterations: implementation within the PhysX repository and then implementing it within the Rythe Engine. The direction of development was guided by external sources or advice from my external assessor. The prototyping method was effective because it allowed me to familiarize myself with the PhysX API while having to deal with fewer abstractions. The PhysX repository prototype was then later on used for testing purposes, meaning that no work was wasted.

3. Testing and rolling out

Good. The testing done within the project was based on the research by a previous work related to unit testing in games. The Rigidbody Dynamics testing resulted in finding an issue related to setting masses within the Rythe Engine integration. Using the data from the test (PVD recordings), the deviances within the tests were also analyzed (position deviances in the rigidbody test, rigidbody positions in the character controller test).

One way for the testing could have been improved is if the physics features were tested in a real game scene. This could have potentially revealed more issues in the API or implementation and would have better proved that the integration is usable in the context of a game.

4. Investigating and analyzing

Excellent. I used the needs of the company to filter out the possible physics engines. Later on, I used existing papers in order to get a performance benchmark on PhysX and Bullet (compared to doing a benchmark myself since this would be much more time consuming and may not be as valid). Most of the

sources I used are either official documentation or somewhat recently released research papers (2018-2022), with some articles when appropriate.

5. Conceptualizing

Good. I explored a large number of physics engines. I then filtered them based on the needs of the company previously established (free modification and access to the code, has the physics features needed, has been used in a released game, performance capabilities).

One part of the conceptualizing phase that was not analyzed is the API of the third-party physics engines. This is an important aspect since it has an influence on the development since how complicated the API is will influence the speed of development.

6. Designing

Good. The final product is a usable physics engine with Rigidbody Dynamics, Charcter Controllers, and Scene Queries. This would likely be enough to make a number of game genres (First/Third Person Shooters, walking simulators, etc).

At the time of writing, character controllers, scene queries, and some parts of rigidbody dynamics have not gone through the code review which I would argue prevents me from getting an excellent. Since this would be the one of the final steps to considering it finished.

7. Enterprising competences

Good. The goal of improving the physics capabilities of the Rythe Engine was done by using an existing physics engine (PhysX). Which is more financially and technically feasible since the work of implementing the features is already done, thus saving the time and resources needed to develop them from scratch.

8. Working in a project-based way

Sufficient. I scheduled the task that needed to be accomplished within the week. Which was further divided down to the things that needed to be done within the day. This allows me to see if I am falling behind. I also scheduled bi-weekly meetings with my external assessor in order to discuss the project and get advice. This methodology was enough to get the project done but lacked long term planning and retrospective.

My ability to balance between working on the project and working on the paper was also somewhat lacking since there were situations where one or the other was lagging behind. I think a better way of handling this balance is by having set goals for both the paper and the project (i.e having the goal to complete a certain section of the paper before going back to the project, the goal of completing feature x before going back to the paper). By doing this, I can avoid constantly switching between them and can better focus on one.

I think the team aspect of the project could have been improved. Other than my meetings with my external assessor and some of the code reviews from colleagues at Rythe Interactive, this project was somewhat of an individual endeavor.

Perhaps one way of improving it would be to do this free written assignment together if possible. For example, in the Rythe Engine, while I am working on physics, someone else could be doing the free written graduation working on rendering, core engine systems, etc. Which could then lead to stand ups, more peer reviews, and retrospective with other people.

9. Communication

Good. The writing follows the APA citation guide, the writing is formal and well sourced. Effort was spent on collecting images to visualize physics features, creating UML diagrams in the development section, creating images to visualize the use of bit flags in the scene query section, using colored tables to display data, etc.

10. Learning ability and reflectivity

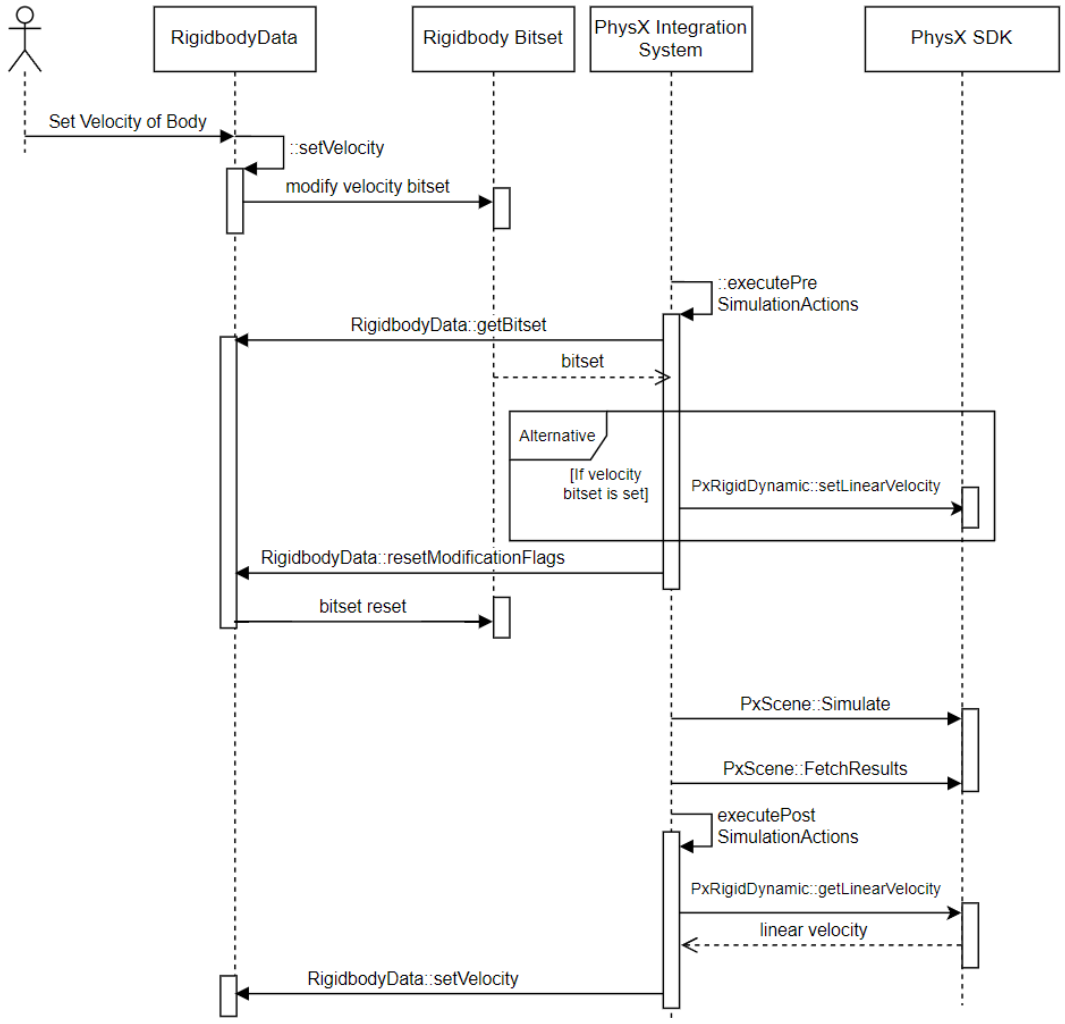
Good. I was able to spot the weak and strong parts of the graduation and suggest some possible ways of improvement. I used the advice from my external assessor in order to get feedback regarding my work.

11. Responsibility

Good. The development of the project followed the guidelines set by Rythe Interactive, The paper followed the guidelines for citations and sourcing, and the goals of the project were achieved. I worked under a fixed and consistent schedule that I independently created myself.

9. Appendices

9.1 Sequence Diagram for setting the velocity of a rigidbody



9.2 Code snippet of the filter callback used to recreate Unreal's collision filtering system.

```
physics_object_reaction getReaction(PxU32 reactionList, PxU32 otherObjectTye)
{
    physics_object_reaction result = physics_object_reaction::reaction_ignore;

    for (size_t currentBitPosition = physics_object_reaction::reaction_max * otherObjectTye, objectReactionIndex = 0;
        objectReactionIndex < physics_object_reaction::reaction_max; currentBitPosition++, objectReactionIndex++)
    {
        if ((reactionList >> currentBitPosition) & 1)
        {
            return static_cast<physics_object_reaction>(objectReactionIndex);
        }
    }

    return result;
}

PxFilterFlags filterShader(
    PxFilterObjectAttributes attributes0, PxFilterData filterData0,
    PxFilterObjectAttributes attributes1, PxFilterData filterData1,
    PxPairFlags& pairFlags, const void* constantBlock, PxU32 constantBlockSize)
{
    PxU32 firstObjType = filterData0.word1;
    PxU32 secondObjType = filterData1.word1;

    //get reaction of objects to each other
    physics_object_reaction firstToSecond = getReaction(filterData0.word0, secondObjType);
    physics_object_reaction secondToFirst = getReaction(filterData1.word0, firstObjType);

    physics_object_reaction lowestReaction = firstToSecond < secondToFirst ? firstToSecond : secondToFirst;

    if (lowestReaction == physics_object_reaction::reaction_block)
    {
        pairFlags = PxPairFlag::eCONTACT_DEFAULT;
    }
    else if (lowestReaction == physics_object_reaction::reaction_overlap)
    {
        pairFlags = PxPairFlag::eTRIGGER_DEFAULT;
        pairFlags |= PxPairFlag::eNOTIFY_TOUCH_PERSISTS;
    }
    else
    {
        return PxFilterFlag::eSUPPRESS;
    }

    return PxFilterFlag::eDEFAULT;
}
```

9.3 Code Snippet of the query callback in order to implement the query mask system

```
physics_object_reaction getQueryReaction(PxU32 reactionList, int otherObjectType)
{
    physics_object_reaction result = physics_object_reaction::reaction_ignore;

    size_t queryReactionMax = physics_object_reaction::reaction_max - 1;

    for (size_t i = queryReactionMax * otherObjectType, j = 0; j < queryReactionMax; i++, j++)
    {
        if ((reactionList >> i) & 1)
        {
            return static_cast<physics_object_reaction>(j + 1);
        }
    }

    return result;
}
```

```
PxQueryHitType::Enum preFilter(
    const PxFilterData& rayFilterData, const PxShape* shape, const PxRigidActor* actor, PxHitFlags& queryFlags) override
{
    ecs::entity ent = { static_cast<ecs::entity_data*>(actor->userData) };

    //get filter data of shape
    const PxFilterData objectFilter = shape->getQueryFilterData();

    //immediately return if ray is default
    if (rayFilterData.word0 == std::numeric_limits<PxU32>::max())
    {
        return PxQueryHitType::eBLOCK;
    }
    //if object has default mask use ray default reaction instead
    if (objectFilter.word2 == std::numeric_limits<PxU32>::max())
    {
        if (rayFilterData.word1 == physics_object_reaction::reaction_block)
        {
            return PxQueryHitType::eBLOCK;
        }
        else if (rayFilterData.word1 == physics_object_reaction::reaction_overlap)
        {
            return PxQueryHitType::eTOUCH;
        }
        else
        {
            return PxQueryHitType::eNONE;
        }
    }

    PxU32 objectType = objectFilter.word2;

    //check reaction of ray towards mask
    physics_object_reaction reactToRay = getQueryReaction(rayFilterData.word0, objectType);

    if (reactToRay == physics_object_reaction::reaction_block)
    {
        return PxQueryHitType::eBLOCK;
    }
    else if (reactToRay == physics_object_reaction::reaction_overlap)
    {
        return PxQueryHitType::eTOUCH;
    }

    //return respective reaction
    return PxQueryHitType::eNONE;
}
```

9.4 The PVD files for all test scenes

https://drive.google.com/drive/folders/1CyaINnRVlhuW5hfIw5t_i2jZpqa_cVfT?usp=sharing

9.5 The PVD recordings of the Rigidbody Dynamics Scene taken from the PhysX repository Prototype and the Rythe Engine

Rythe Engine PVD Video: https://youtu.be/Z7Y0wAhynNY

Physx Repo PVD Video: https://youtu.be/zAO4EsTrOJQ

9.6 The PVD recordings of the Capsule Controller Scene taken from the PhysX repository Prototype and the Rythe Engine

Rythe Engine PVD Video: https://youtu.be/sFdj7EPIukE

Physx Repo PVD Video: https://youtu.be/hj-vmcO0UGs

9.7 The PVD recordings of the Scene Query and Collision Filtering Scenes taken from the PhysX repository Prototype and the Rythe Engine

Rythe Engine PVD Video

Collision Filtering: https://youtu.be/Z51ddcKQ1NI

Scene Query: https://youtu.be/jVzfXOEDJSQ

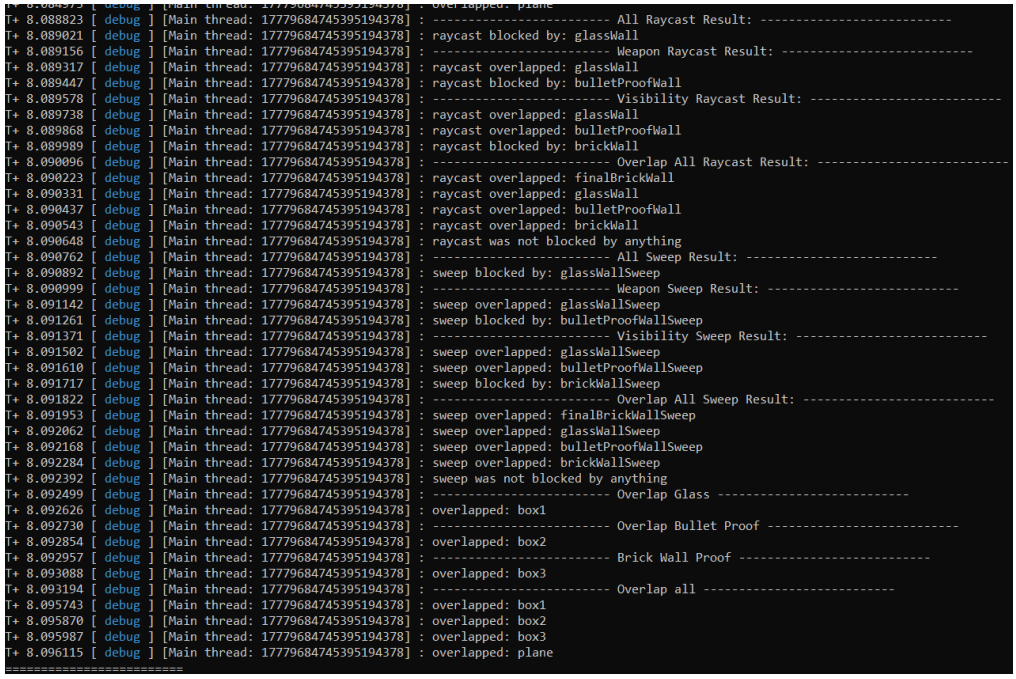
PhysX Repo PVD Video

Collision Filtering: https://youtu.be/4gn_e2W6J3Y

Scene Query: https://youtu.be/761iDUKt6UQ

9.8 Log Information relating to the scene query test taken from the PhysX repository Prototype and the Rythe Engine

Rythe Engine:


T+ 8.088823 [debug] [Main thread: 1777968474539519437 8] : ------------------------- All Raycast Result: -------------------------
T+ 8.089021 [debug] [Main thread: 1777968474539519437 8] : raycast blocked by: glassWall
T+ 8.089156 [debug] [Main thread: 1777968474539519437 8] : ------------------------- Weapon Raycast Result: -------------------------
T+ 8.089317 [debug] [Main thread: 1777968474539519437 8] : raycast overlapped: glassWall
T+ 8.089447 [debug] [Main thread: 1777968474539519437 8] : raycast blocked by: bulletProofWall
T+ 8.089578 [debug] [Main thread: 1777968474539519437 8] : ------------------------- Visibility Raycast Result: -------------------------
T+ 8.089738 [debug] [Main thread: 1777968474539519437 8] : raycast overlapped: glassWall
T+ 8.089868 [debug] [Main thread: 1777968474539519437 8] : raycast overlapped: bulletProofWall
T+ 8.089989 [debug] [Main thread: 1777968474539519437 8] : raycast blocked by: brickWall
T+ 8.090096 [debug] [Main thread: 1777968474539519437 8] : ------------------------- Overlap All Raycast Result: -------------------------
T+ 8.090223 [debug] [Main thread: 1777968474539519437 8] : raycast overlapped: finalBrickWall
T+ 8.090331 [debug] [Main thread: 1777968474539519437 8] : raycast overlapped: glassWall
T+ 8.090437 [debug] [Main thread: 1777968474539519437 8] : raycast overlapped: bulletProofWall
T+ 8.090543 [debug] [Main thread: 1777968474539519437 8] : raycast overlapped: brickWall
T+ 8.090648 [debug] [Main thread: 1777968474539519437 8] : raycast was not blocked by anything
T+ 8.090762 [debug] [Main thread: 1777968474539519437 8] : ------------------------- All Sweep Result: -------------------------
T+ 8.090892 [debug] [Main thread: 1777968474539519437 8] : sweep blocked by: glassWallSweep
T+ 8.090999 [debug] [Main thread: 1777968474539519437 8] : ------------------------- Weapon Sweep Result: -------------------------
T+ 8.091142 [debug] [Main thread: 1777968474539519437 8] : sweep overlapped: glassWallSweep
T+ 8.091261 [debug] [Main thread: 1777968474539519437 8] : sweep blocked by: bulletProofWallSweep
T+ 8.091371 [debug] [Main thread: 1777968474539519437 8] : ------------------------- Visibility Sweep Result: -------------------------
T+ 8.091502 [debug] [Main thread: 1777968474539519437 8] : sweep overlapped: glassWallSweep
T+ 8.091610 [debug] [Main thread: 1777968474539519437 8] : sweep overlapped: bulletProofWallSweep
T+ 8.091717 [debug] [Main thread: 1777968474539519437 8] : sweep blocked by: brickWallSweep
T+ 8.091822 [debug] [Main thread: 1777968474539519437 8] : ------------------------- Overlap All Sweep Result: -------------------------
T+ 8.091953 [debug] [Main thread: 1777968474539519437 8] : sweep overlapped: finalBrickWallSweep
T+ 8.092062 [debug] [Main thread: 1777968474539519437 8] : sweep overlapped: glassWallSweep
T+ 8.092168 [debug] [Main thread: 1777968474539519437 8] : sweep overlapped: bulletProofWallSweep
T+ 8.092284 [debug] [Main thread: 1777968474539519437 8] : sweep overlapped: brickWallSweep
T+ 8.092392 [debug] [Main thread: 1777968474539519437 8] : sweep was not blocked by anything
T+ 8.092499 [debug] [Main thread: 1777968474539519437 8] : ------------------------- Overlap Glass -------------------------
T+ 8.092626 [debug] [Main thread: 1777968474539519437 8] : overlapped: box1
T+ 8.092730 [debug] [Main thread: 1777968474539519437 8] : ------------------------- Overlap Bullet Proof -------------------------
T+ 8.092854 [debug] [Main thread: 1777968474539519437 8] : overlapped: box2
T+ 8.092957 [debug] [Main thread: 1777968474539519437 8] : ------------------------- Brick Wall Proof -------------------------
T+ 8.093088 [debug] [Main thread: 1777968474539519437 8] : overlapped: box3
T+ 8.093194 [debug] [Main thread: 1777968474539519437 8] : ------------------------- Overlap all -------------------------
T+ 8.095743 [debug] [Main thread: 1777968474539519437 8] : overlapped: box1
T+ 8.095870 [debug] [Main thread: 1777968474539519437 8] : overlapped: box2
T+ 8.095987 [debug] [Main thread: 1777968474539519437 8] : overlapped: box3
T+ 8.096115 [debug] [Main thread: 1777968474539519437 8] : overlapped: plane
==========

PhysX Repo:

```
---------------------- all raycast ----------------------
raycast finally hit glassWall
---------------------- weapon raycast ----------------------
ray overlapped actor glassWall
raycast finally hit bulletProofGlass
---------------------- visibility raycast ----------------------
ray overlapped actor glassWall
ray overlapped actor bulletProofGlass
raycast finally hit brickWall
---------------------- overlap all raycast ----------------------
ray overlapped actor finalBrickWall
ray overlapped actor glassWall
ray overlapped actor bulletProofGlass
ray overlapped actor brickWall
raycast not blocked by anything
---------------------- block anything sweep ----------------------
sweep finally blocked by glassWallSweep
---------------------- weapon sweep ----------------------
sweeped actor glassWallSweep
sweep finally blocked by bulletProofGlassSweep
---------------------- visibility sweep ----------------------
sweeped actor glassWallSweep
sweeped actor bulletProofGlassSweep
sweep finally blocked by brickWallSweep
---------------------- overlap everything sweep ----------------------
sweeped actor finalBrickWallSweep
sweeped actor glassWallSweep
sweeped actor bulletProofGlassSweep
sweeped actor brickWallSweep
sweep not blocked by anything
---------------------- overlap with mask 0 ----------------------
overlapped actor box1
sweep overlapped
---------------------- overlap with mask 1 ----------------------
overlapped actor box2
sweep overlapped
---------------------- overlap with mask 2 ----------------------
overlapped actor box3
sweep overlapped
---------------------- overlap all ----------------------
overlapped actor box1
overlapped actor box2
overlapped actor box3
overlapped actor groundplane
sweep overlapped
```

9.9 Branch containing all of the implemented physics features

https://github.com/Rythe-Interactive/Rythe-Engine.rythe-legacy/tree/subfeature/scene-queries

9.10 Peer Reviewed and Merged Rigidbody Dynamics branch

https://github.com/Rythe-Interactive/Rythe-Engine.rythe-legacy/pull/22

9.11 Professional Product

https://youtu.be/e0303OhvkVg