Creating a Physics Game Toolkit in Unreal Engine

Masters Project Thesis

Jack Beasley

s5219489

MSc Computer Animation and Visual Effects

Abstract

This project developed a physics game toolkit, created in Unreal Engine. The toolkit combined both Blueprint scripting and C++ programming by prototyping new systems in Blueprints, then re-creating them in code for added efficiency. These code systems were then exposed to Blueprint in the form of key variables and functions. This created an interface for an end user to use the toolkit while also allowing for future expansion, entirely in Blueprint if desired. Finally, the finished toolkit has been showcased in the form of a demo puzzle, utilising all systems in the toolkit and displaying how they could work together to generate puzzles in a physics-based game.

Table of Contents

1 Introduction

Unreal Engine is currently one of the most widely used games engines available to developers. From March 2nd, 2015, Epic Games removed a license fee for developers using the engine, and instead took a 5% percentage of sales from successful projects (Sweeney, 2015). One of the unique elements of Unreal is Blueprints, a high-level programming language that uses interconnected nodes as opposed to text. This allows non-programmers to implement game logic without underlying knowledge of C++. For programmers, it can also be used to prototype gameplay elements quickly and experiment before committing anything to code.

A healthy combination of Blueprints and C++ within a game should be the most efficient way to structure a project. However, this is difficult for a non-programmer as they cannot make use of the added efficiency C++ gives to game logic (Epic Games, 2018). For a programmer, trying to make use of Unreal's existing systems in code is not always straightforward, with minimal documentation and sparse resources.

One area where this is particularly apparent is in physics-based games. These would perhaps benefit most from a C++ implementation with mathematical calculations but also require extensive use of Blueprint systems, such as collision events, which are sometimes difficult to utilise in code.

This project aims to investigate this area by creating a toolkit for a physics game in Unreal Engine. It will be constructed with the use of Blueprints for prototyping purposes, but all game logic will eventually be constructed in code. The toolkit will aim to provide a backbone for a general physics game, with systems that would be required within most projects. To achieve this, all systems must be exposed to Blueprints. Key variables and functions will be accessible to the user, without the ability to alter core logic. Thus, an end user could extend or alter the existing systems entirely within Blueprints.

Another main aim of the project is to investigate the relationship between Blueprints and C++ in a games project. Most learning resources that are currently available construct projects with only one of these in mind, so it would be useful to establish how harmoniously the two work together. This includes instances where efficiencies are gained while also finding any key difficulties in this approach.

Finally, the project aims to implement a unique physics game mechanic within the toolkit that could be expanded on to create a physics game. This is a solo project, so the creation of a full game is not feasible. Instead, the aim is to construct a demo level as a proof of concept of how this mechanic would work, as well as demonstrating the rest of the toolkit.

2 Previous Work

Physics games are a relatively niche area of gaming. The most successful entries in this field are *Portal* (2007) and *Portal 2* (2011), developed and published by Valve. These games are based around a unique key mechanic: the portal gun, which allows objects to enter through one portal and exit through the other. The mechanic itself doesn’t change through either of the game’s runtimes. Instead, additional systems are introduced slowly throughout to complement the core mechanic. An example level from these games is displayed in Figure 1.

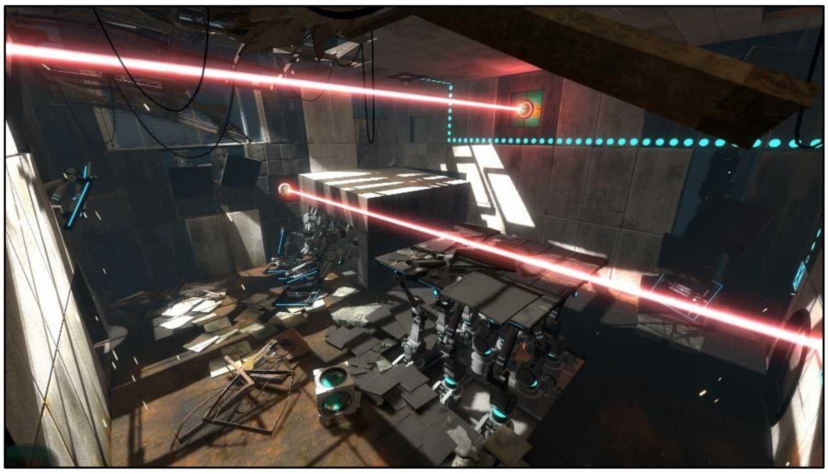


Fig. 1. A level layout from a highly successful physics game, *Portal 2* (GeForce, 2011)

When looking at toolkits as opposed to full games, Epic Games have a marketplace to showcase user-submitted projects created in the engine as free or paid content. After browsing this marketplace, there was a lack of general-purpose physics toolkits available. There was one submission that had been well reviewed, titled Ultimate FPS Puzzle Kit, which was working in a similar area. However, the toolkit was constructed entirely in Blueprints, which placed it in a different space to this project.

Engines such as Unreal and Unity often utilise external sources for the simulation of physics in their engine. For both of these, the PhysX SDK, developed by NVIDIA, is the preferred choice. PhysX uses techniques such as position-based dynamics (PBD) to calculate the result of interactions between objects, which applies a series of constraints to control the motion of objects and their interactions. These constraints can range from environmental to bending and stretching. The PBD approach is also flexible and can be used in other areas of simulation such as cloth and fluids (Bender et al., 2015).

3 Technical Background

This project utilises a number of underlying systems within the engine. This includes the PhysX engine from NVIDIA which underpins all physics simulations in Unreal as of the version used in this project, 4.25.1. The classes in this project also build on existing classes in the engine such as Actor and Character.

3.1 Blueprints

As mentioned previously, Blueprint is a visual-scripting language that uses a node system to connect logic. One feature of Blueprints used in the project is the construction script. The construction script runs when part of a Blueprint is edited within the Editor, such as scale or variable values. This allows for dynamic instancing that can be adjusted in real-time, without needing to run the game.

BeginPlay and Tick events are also a key part of working in Blueprint. BeginPlay is fairly self-explanatory in that it triggers when the game starts, while Tick triggers every frame, but can also be set to a specific time-step. The Tick event is one of the most useful de-bugging tools in the engine as it allows the developer to find the moment if and when a value changes. This is often combined with the “Print String” node, which can output a value of any type and is far more intuitive than its C++ counterpart. This functionality was an essential part of the prototyping process of new systems in this project.

3.2 Programming

The IDE used within this project was Visual Studio 2019. Visual Studio uses Intellisense as a code-completion aid. However, this did not work consistently and included some quirks such as needing to re-build the project every time a new class was created. Compile time was also a potential issue but prototyping logic in Blueprint negated the need to compile as frequently.

All classes were first created in the Unreal Editor, which automatically added them to the Visual Studio project. There are a number of parent classes available in the engine, the ones used in this project were Actor and Character. The Actor class is the base class for an object that can be placed or spawned within the world (Epic Games, 2020) and includes elements such as overlap events and spawning capabilities. The Character class is designed for a player or AI and includes a mesh, collision and built-in movement logic.

The main change from standard C++ when using Unreal are the UPROPERTY and UFUNCTION specifiers. These determine how much a property or function is exposed to the Editor; when they can be accessed and/or edited.

There are four main options when defining a UPROPERTY. The first is how editable a property is, which can range from only being visible to being editable anywhere. The second determines whether properties can be set within Blueprint by other nodes, as opposed to manually altering values. The third groups properties into readable categories and finally, there are meta specifiers with a wide-ranging array of capabilities, but these were used minimally during the project. One of these that was useful was the EditCondition specifier. This meant that the variable could only become editable in Blueprint if a condition was met. For example, a user could only set a de-spawn delay for objects if they were actually set to de-spawn. It's a small feature but did make the implementation more intuitive.

The UFUNCTION has one unique difference in that it can be partially, or entirely, defined in Blueprints. This allows a function to be called in code but defined in Blueprints by an end user. All these specifiers are vital in combining Blueprints and C++ in the same project. An example of these specifiers in code is displayed in Appendix A.

3.3 PhysX

The PhysX engine contains a number of key tools used within this project. This ranges from basic friction and damping to physics constraints, which define how two actors can interact with each other. Constraints could include rotation/translation limits and spring capabilities. Physically simulating all objects in the project allows the player to interact directly with them if required while also being able to set cheaper options for fixed objects. The PhysX engine is due to be replaced by the new Chaos engine as of Unreal Engine 5, due to release in 2021.

4 Implementation

The implementation of this project was split into two main sections. The first was the creation of a series of classes to form a physics game toolkit, that could be altered or added to as the backbone of a physics game. This included a core mechanic that could serve as the key point for any puzzles to be based around. Finally, these classes were brought together into a demo puzzle, to showcase how they would work together within a game.

The second section, carried out in tandem with the first, was an investigation into the relationship between Blueprints and C++ in a games project. This included both advantages and disadvantages of the use of both together in a project.

The classes were designed with functional logic in mind, with visuals kept deliberately basic. All visual logic in the project was exclusively kept to Blueprints, and so could easily be overwritten by an end user.

This section will explain each of the classes and their purpose within the toolkit. All classes inherit from the Actor class unless stated otherwise.

4.1 Impulse Device

This was the first class created for the project as it was intended to house the core mechanic of the game, which needed to be effective before moving on to other parts of the toolkit. The name refers to the original intention of the device, to store the momentum of one object with one laser beam and apply that momentum, as an impulse, to another object with another beam.

To achieve this, a line trace was fired from the centre of the player's screen, until it collided with an object. This would be stored as the end location for a second line trace, this time fired from the end of the device. By doing this, the player would have clear control over where a beam was fired, while still appearing to be fired from the end of the device. However, there was a key issue with this. The second line trace would often not register a hit, despite the fact it was travelling to exactly the same location as the first.

Figure 2.1 displays this effect where the beam would not update with a change in camera direction, leaving an odd-looking effect.

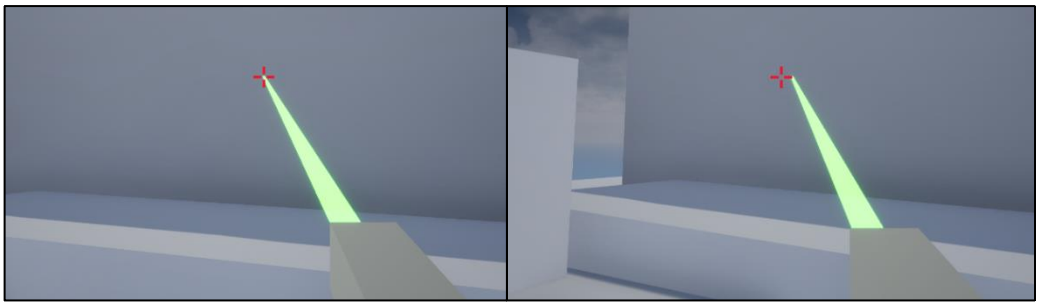


Fig. 2.1. Intended laser path (left) and laser path with the error (right)

The cause of the error was due to the second trace being fired from a different start point. Although it was traveling to the same location, the change in angle would sometimes mean the overlap with the target object would not quite trigger. Knowing this, the problem was solved by simply extending the length of second trace in the same direction.

The creation of this device highlighted a key difference between Blueprints and C++. In the class, the laser beam was updated at regular intervals, defined by the user. In Blueprints, a Tick event could be used or, alternately, a delay node, which simply waits a set time before firing the next node. A delay equivalent does not exist in the C++ version. Instead, the WorldTimerManager is used where the user has to specify a function to be called, and whether it should be repeatable. The timer manager waits the specified time, then fires that function. This was frustrating when only a delay was required without any specific logic, as the process included having to define and clear a TimerHandle, which manages the delay. This was not clearly explained through the documentation, so was definitely an area where Blueprints is not only more intuitive, but more functional as well.

The device was originally designed to only perform its main function, to store momentum and apply an impulse. This did work as intended but as time wore on, it became clear that this did not align with the rest of the project. The logic was entirely contained within C++ and could not be easily overridden or added to by an end user.

To update this, the UFUNCTION specifiers, mentioned earlier, were utilised. First the logic was shifted so that anything outside of the firing and switching off of the laser was contained within a single function. That function was then set to be a BlueprintImplementableEvent. This meant the function could be called in code but was defined entirely in Blueprints. It was set so that when the second line trace hit any of the physics objects, the function was called. By doing this, the core function of the laser could not only be directly editable by an end user, but by separating the logic of the laser itself and the rules it applied, the device could be set to have different settings with different rules applied to each one.

The final layout of this, in Blueprints, is shown in Figure 2.2.

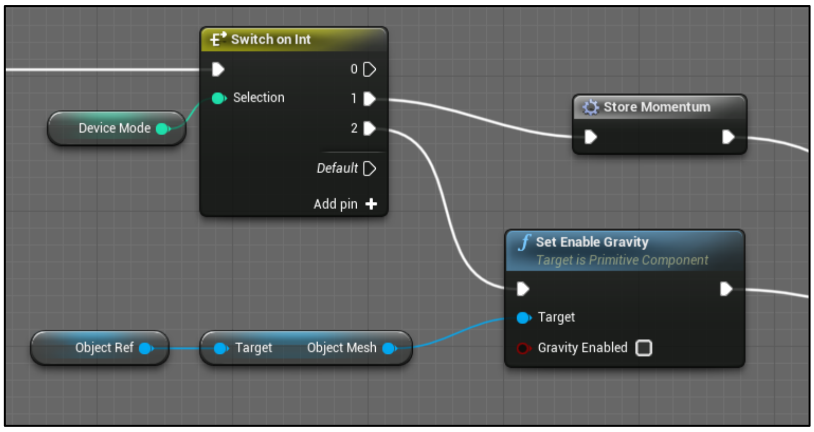


Fig. 2.2. The logic of the laser rules in Blueprint, where the Switch on Int could be extended in future for new rules

The device still had its original momentum logic, but now had a separate setting to switch the gravity of an object off or on. The device was now easily expandable for an end user to create new mechanics or edit existing ones, with the logic of firing the laser itself tucked away in code.

A class diagram for the Impulse Device is contained in Appendix B.

4.2 Impulse Object

The object class was the next to be created as they directly interacted with the device. The class itself started very simply, as all the object was required to do was simulate physics. This would mean values such as linear velocity would be stored and could be accessed by the device through a reference.

To allow for an end user to create new objects that could also be recognised by the C++ logic, inheritance was used. This meant that any children of the class would be recognised as a member of that class. This was important as the device only applied its rules when the line trace hit an object of that type. First, a blueprint class was made based on the base class. This normally would not be advisable as the class is designed to be abstract, but it allowed for Blueprint children to be made from this class. The benefit of doing this was that not only would these children inherit any changes to the C++ class, but an end user could make changes to the parent Blueprint class which the children would also inherit.

This functionality proved useful when adding a visual element to the objects. It was not always clear whether an object had been hit by a laser. The laser would turn off after a short time, but the moment of impact was less clear, particularly if the object was travelling at a high velocity.

Figure 2.3 shows the moment of impact of the laser beam on an object.

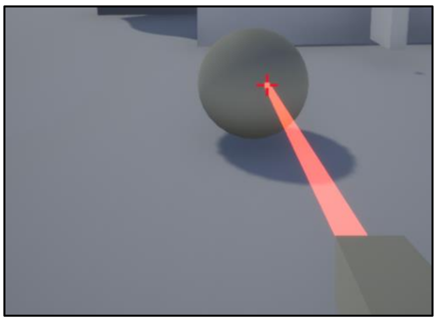


Fig. 2.3. The laser interacting with an object before any visual additions

To counteract this, a dynamic material was applied to the object. This allows for parameters to be edited in Blueprints at any time, rather than exclusively in the Material Editor. It was set to be an additive material, which is a cheaper version of translucency, with the emissive colour updating when an object was hit. This highlighted another benefit of the exposed device settings, as the colour changing could be tied exactly to the point when the object was hit and reverted after the logic was executed. Figure 2.4 displays this colour change in the object, with the new colour being the laser colour. The colour would then change back to normal when the laser switched off. As this was a visual change, it was kept to Blueprints to be updated as required by an end user.

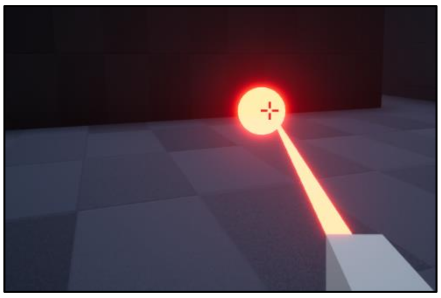


Fig. 2.4. The object taking on the colour of the laser during interaction

A class diagram for the Impulse Object is available in Appendix C.

4.3 Conveyor

The next logical step in the creation of the toolkit was to assist these objects in moving around a level. A setting that is intuitive for a high volume of objects moving from place-to-place is a factory, and a factory usually contains conveyor belts. Upon researching how a conveyor belt was achieved in other projects, the response was always to fake it, as there was no reason to go further. All the examples found transformed each object a set distance each frame, in the direction of the conveyor. This approach did not work for this project. For a physics object, transforming its position does not equate to a velocity, so its momentum could not be found. For a conveyor to work, the object would have to be physically moved by the mechanism.

To fit this criteria, a roller conveyor seemed the most appropriate as it did not require a power source, the objects could be moved by gravity. A roller conveyor has a series of cylindrical rollers, which can only rotate on one axis. The movement of the object rotates the roller, and that rotation moves the object further. An example of a roller conveyor is shown in Figure 2.5.

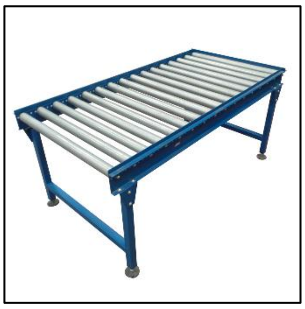


Fig. 2.5. An example of a gravity roller conveyor (Spaceguard, 2020)

4.3.1 Generation

The next question was then how to generate this conveyor. The idea of a user specifying a series of parameters, then the conveyor being generated at runtime, seemed very inefficient. Instead, the conveyor would be generated along a spline, and utilise the construction script to update this dynamically. To start, the spline length was divided by the length of a roller section to generate the correct number of sections. A for loop would then add a static mesh component to the world for each section, with tangents also determined by the spline.

This worked correctly when prototyping in Blueprints, but in code was another issue. There didn’t seem to be an obvious way to re-create the construction script directly in code, but luckily one person had found the specific steps to allow the object to behave dynamically (even, 2019). They had achieved this through trial and error, which seemed to be the only way to find the solution as most of the available advice was out-of-date for the current engine version. There were also a number of settings that needed to be set explicitly in code, such as collision, that were set by default in Blueprint, which seemed inefficient.

4.3.2 Physics Constraints

Once the generation of the conveyor was set-up, there was the issue of making the rollers work correctly in a physical sense. One key benefit of using physics constraints for this task was that they could constrain the object in local space, which was crucial as a user had to be able to have the conveyor facing in any direction and still work correctly.

The position of the constraints was set to either side of the roller and attached, with collision disabled between the roller and side section. This was to prevent potential issues in the event of any movement in the roller. This set-up allowed the roller to rotate freely on one axis but nothing else.

4.3.3 Conveyor Legs

Finally, the legs were attached with a separate for loop to allow for an independent spacing control for the user. One difference with the legs was a floor height adjustment control. This allowed an end user to tweak this value until the legs just reached the floor, then lift the conveyor as they wished, with the scale of the legs adjusting to this. The legs also did not use the spline tangent to keep them perpendicular to the floor at all times. An example of these legs is displayed in Figure 2.6.

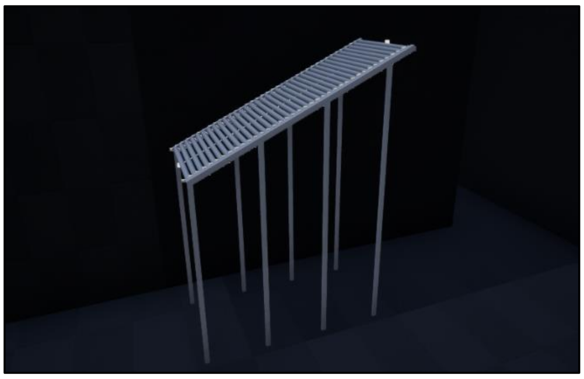


Fig. 2.6. An angled conveyor belt, displaying the dynamic scaling of the legs

4.3.4 Performance

The final conveyor worked as intended, apart from displaying poor performance as the number of conveyor sections increased. This was due to how Unreal handles static meshes. Usually when there is a series of identical meshes, Unreal uses a technique called instancing, where they take the full mesh information for the first instance, but then take the minimum required information for each identical mesh thereafter. This is unfortunately not possible when using a spline, as the spline has the potential to curve, so more information is required to allow any of the meshes to deform and follow the spline path.

The solution to this was to add a View Mesh Boolean. As the name suggests, this meant that the mesh was only displayed when the Boolean was set to true. Once the initial set-up of the parameters was completed, a user could turn off visibility for the mesh and manipulate the spline in isolation, which is obviously far less expensive and easier to manipulate. They could determine the gradient and length of the conveyor, before turning the mesh back on prior to runtime.

The conveyor does have some limitations as it stands. Adding bend to the conveyor breaks the positioning of the physics constraints, as the positional offset isn’t designed with this in mind. However, for the conveyor to work effectively, there would also need to be banking controls, so the object adjusted its motion in the direction of the bends. At this stage, adding this functionality seemed like unnecessary complexity, particularly when a second conveyor could be placed, facing a different direction, and achieve a similar effect.

The class diagram for the conveyor is available in Appendix D.

4.4 Spawner

One need that wasn’t met by the conveyor was a way to spawn objects into the level. Originally, the spawners in this section were made as individual classes. As the project wore on, it became obvious that this was not the correct approach, there would need to be a base spawner class with all the key logic, so a user could then create their own variants without worrying about the underlying system. It was also inefficient to repeat very similar logic in two classes.

4.4.1 Base Class

The base spawner class housed all of the basic functionality required to spawn and de-spawn objects. There were options for where the objects should spawn and what type of object should spawn. There were also settings for de-spawning, with a collision volume to identify overlapping objects, and a variable to set a delay before object de-spawn.

In addition to the base class, two child spawners were also created. It was thought that these would offer some useful functionality while also adding some visual interest to a level.

The class diagram for the spawners is located in Appendix E.

4.4.2 Conveyor

The conveyor spawner was designed to link directly with the existing conveyor class. In the class, there was an option to add a conveyor reference to an existing conveyor in the world. By doing this, velocity could be added in the direction of the conveyor, which minimised the threat of objects becoming stuck before they reached their destination. However, this problem could also be solved by reducing the mass of the rollers on the conveyor itself.

4.4.3 Launcher

A core element that was missing from the existing toolkit was functionality to launch objects. This feature would tie-in well with the momentum mechanic, as the potential to add upwards velocity to objects increases the options for creating puzzles.

The starting point for this spawner was adding the ability to fire an object directly at a target, without the user having to adjust any parameters if the target was moved. To achieve this, some projectile physics equations were consulted, with the known quantities being the horizontal range, the launch angle and the height difference between the origin and target locations. The target vales to find were the horizontal and vertical velocity. Equation (1) was the starting point, where Δy is the height difference, v_{oy} is the starting vertical velocity, with g and t representing acceleration due to gravity and time as standard.

$$\Delta y = v_{o_y} t + \frac{1}{2} g t^2 \qquad (1)$$

This equation was re-arranged by the substitution of t by x/ v_{ox}, x being horizontal range and assuming no air resistance. v_{oy} and v_{ox} were then substituted by $v_o\sin\theta$ and $v_o\cos\theta$ respectively, giving equation (2).

$$\Delta y = \frac{v_o \sin\theta x}{v_o \cos\theta} + \frac{1}{2} g (\frac{x}{v_o \cos\theta})^2 \qquad (2)$$

Finally, this equation was simplified and re-arranged to solve for v_o. This is displayed in equation (3), with g being multiplied by 100 to convert to Unreal units.

$$v_o = (\sqrt{\frac{gx^2}{2(x\tan\theta - \Delta y)}}) \Big/ \cos\theta \qquad (3)$$

This achieved the desired effect. With the logic hidden in code, the user could simply place the target where they desired, and the object would fire with the correct velocity to reach it.

Although this was functional, it was not flexible, as the user could not use the launcher for another purpose. With this in mind, functionality was added to fire on the launch trajectory with a user-set velocity. Another addition was the option to not de-spawn the object upon reaching its target, in case the user wanted to fire the object at something, but de-spawn the object at a later stage.

4.5 Puzzle

With these spawners now functional, the next step was to imagine them in the context of an overall level, where it may be useful to trigger all of the spawners at once. It may also be useful to have a class which manages the state of a puzzle, whether it has been completed and triggers what comes next. This task was done by the puzzle class.

4.5.1 The Class

The puzzle class contained two collision boxes, one to start the level and one to end it. The spawners within the level were stored within an array and when the puzzle started, the spawners were set to active. The advantage of having a base spawner class exposed to Blueprint here is that any future spawner variants would still be able to be added to this array and contain the same functionality to set them active. The second collision box would turn off all the spawners at a point where the user was not returning to the puzzle, to avoid unnecessary spawning of objects.

The class diagram for the Puzzle is available in Appendix F.

4.5.2 Objective Button

The objective button is a separate class that was purely designed to be attached to the puzzle class. The button uses physics constraints with a spring-like functionality so when an object falls onto it, it depresses and springs back if the object falls off. To get this to work as required was surprising difficult, balancing both the friction and the spring force. In the end it required gravity to be turned off for the button, to avoid it pressing down by itself.

These buttons could also be stored in an array, and when all were pressed, a Boolean was exposed to Blueprint as part of the CheckObjectiveButtons function. This allowed the user to set whatever functionality was required for them at the end of a level, whether that was opening a door or bridging a gap.

This class could be extended in future by taking object mass in account, so multiple objects or a single, heavier object may be required to press it. Another change could be the creation of an Objective parent class to allow for different kinds of objectives. However, the logic stored in this base class would have to be limited as the button has a fairly unique set of components.

The class diagram for the Objective Button is available in Appendix F.

4.6 Gravity Field

The final main addition to the toolkit was a gravity field, designed to attract objects that entered the beam and carry them along the length of it. The “beam” was set as a cylinder mesh that would trigger when an object overlapped with it. This mesh had a dynamic material attached to give the illusion of movement along the cylinder. To create this, a panner node was inputted into a GeneratedBand node, panning a series of bands along the length of the mesh. Dynamic parameters were added for the number of bands and the speed of the pan, for user preference and to match the speed of the objects, which were defined in code.

Figure 2.7 displays the final effect.

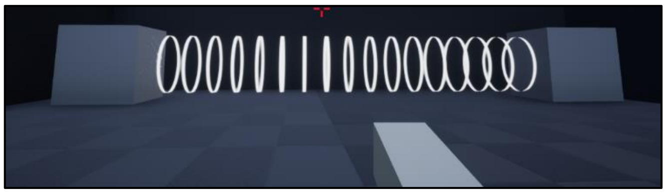


Fig. 2.7. A gravity field with the dynamic panning material attached

The issue was then how forces would be applied to the object. This was originally handled by the objects themselves, but this seemed wrong as it added a number of variables to the object that it shouldn't need. Instead, a count was kept of overlapping objects, and if there was at least one, the required force would be applied to all of them individually.

The next step was to determine what forces would be applied to each object while in the field. The first was a general deceleration of the object's velocity, then a force to send it in the direction of the beam. The final force was to attract the object towards the centreline of the field. This proved to be more difficult as there was no attraction force set-up for objects, only for particles. It was easy to attract towards a point in space, but not to move towards, then stay on, a line. The eventual solution to this was store the start and end point of the field and apply a vector towards both. This would naturally pull the object towards the minimum distance between both ends, which was on the centreline. These vectors were normalised so they could be multiplied by a user-defined variable, setting the attraction strength of the field. If the object was falling or rising too quickly before entering the field, it would end its overlap before decelerating enough and simply leave the field.

The class diagram for the Gravity Field is available in Appendix G.

4.7 Demo Puzzle

To bring all these classes together, the demo puzzle was designed to investigate how the elements of the toolkit could work together, as they would need to within a game. It was also an opportunity to find new elements to add to the toolkit, that may be discovered through the creation of the puzzle. Finally, it was a chance to iron out any bugs that could be uncovered with a heavier usage of the toolkit elements.

4.7.1 Despawner

The first addition to the toolkit was an obvious one, a general de-spawn for objects. It was important to have some way to remove objects that had been taken off their usual path by the player and could no longer be used to solve the puzzle. In this puzzle, the Despawner was applied to the lowest floor of the level.

The initial route taken was to use the same approach as the spawners, where an object overlapping with the target would de-spawn after a delay. This didn't work for a broader case as multiple objects could overlap at the same time. In that instant, the system would still be processing the first object by the time the second finished overlapping. The second object would, therefore, not de-spawn. To work around this, a count was stored of overlapping objects, if the count was more than zero, the Despawner would delete all overlapping objects at once.

The class diagram for the Despawner is located in Appendix C.

4.7.2 Conveyor Ball

The second element added was a conveyor belt for spherical objects, as the original conveyor class did not account for this. It would have been intuitive in this case to create a conveyor base class to allow for any of the children to be used in the conveyor spawner. However, the roller conveyor had too much unique logic, facilitating the physics constraints, to be compatible. The Conveyor Ball could be seen as the parent for future conveyor classes as it only contains two elements: the legs and the conveyor itself.

The class diagram for the Conveyor Ball is in Appendix H.

4.7.2 Trigger Button and Interaction

To utilise the functions within the gravity field such as toggling and reversing the field, a button was devised that would store all the fields in the level and would alter them through the press of the button. In this particular puzzle, pressing the button would turn one field on but turn the other one off, which could lift objects off their objectives if executed in the wrong order.

This implementation required the creation of an interact ability for the player to press the button. A line trace function was added to search for nearby actors and apply logic based on the result. This worked for pressing the button but could also be used in the future to allow picking up of objects or interaction with levers or switches. The class diagram for the trigger button is available in Appendix G.

4.7.3 Reset Field

Another discovery while creating the puzzle was the reset field, which resets all values stored by the device. This was created as a way to add increased complexity to a puzzle, with the player being unable to simply store, then apply, momentum to separate objects if they were caught on either side of the field. The logic of the field was also exposed to Blueprint so any future rules added to the device could be included in the reset.

Figure 2.8 shows how the field was presented within the level.

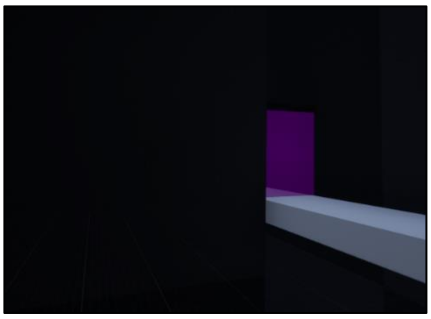


Fig. 2.8. The Reset Field, designed to wipe all stored values in the device

The collision of the field could also be useful to a puzzle as it blocks the lasers and objects but allows the player to pass through. Additional logic could be added such as the field toggling on and off to block objects or prevent the laser firing at an object from certain angles.

The class diagram for the Reset Field is located in Appendix B.

4.7.4 Bugs

Creating this puzzle did expose a few issues with parts of the toolkit. The launcher, for example, did not fire at the correct location if the meshes were scaled away from default values. This was initially confusing as it had appeared to work correctly in the past. However, it was found that the horizontal range was actually calculated using the two meshes, rather than the spawn and target locations themselves. If the mesh was scaled, the distance would not change but the position of the spawn would, causing the error.
A more serious error that occurred during this process concerned the conveyor. When the level had been packaged, it was found that the conveyor didn't appear in the level, despite working as intended in the Editor. The Blueprint prototype was updated to have the same logic and substituted in, and it worked completely fine. Upon researching this issue, it was found to be a known issue without an obvious fix, relating to the C++ creation of the spline mesh components required for the conveyor. The conveyor worked fine in other areas, so the approach to debugging this problem was unclear.

The eventual solution involved a hybrid approach, with the components created in Blueprints but all other logic dealt with in code. The final Blueprint layout is shown in Figure 2.9, with the two spawning functions being implemented in code.

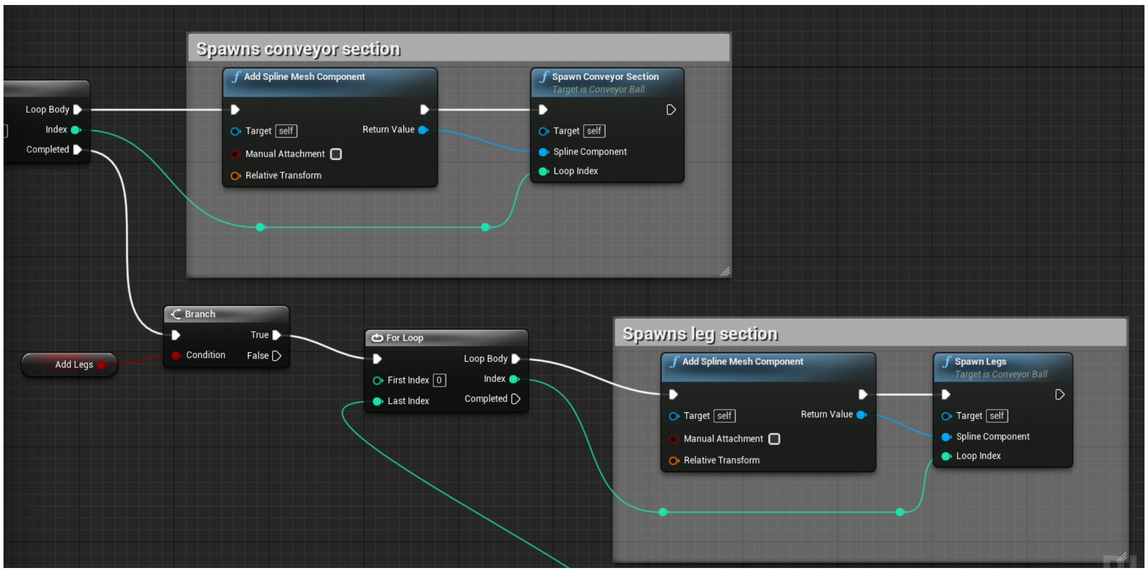


Fig. 2.9. Part of the Blueprint implementation of the conveyor belt

4.7.5 Final puzzle

The final puzzle contained all elements of the toolkit, including the new sections created while constructing the puzzle. There were also a few features included to round off the puzzle. These included animations to open and close a door on completion of the puzzle and some UI to indicate what mode the device was currently in.

A screenshot of the final puzzle is displayed in Figure 2.10.

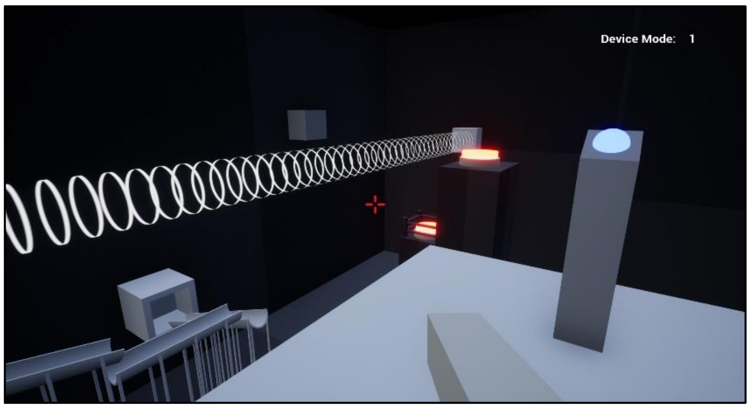


Fig. 2.10. A still from the finished demo puzzle

5 Conclusion

In summary, the project has achieved its aim of creating a physics game toolkit in Unreal Engine. The toolkit as it stands provides a good backbone for a physics game project. There are ways to spawn and transport objects, and systems to add complexity to a level, such as the gravity and reset fields. Each system is open to expansion, either in code or in Blueprint, and all placeholder visuals are contained in Blueprint to be easily overwritten. A unique mechanic was also added: a device that could store the momentum of an object and apply it to another, with room for additional mechanics to be added. This is one of the areas in which the project could be developed the most, by introducing more ways to interact with objects and create more complex levels.

The other main aim in the project was to investigate the relationship between Blueprints and C++ within a project in Unreal Engine. For the most part this relationship was intuitive. The function and property specifiers could clearly define how logic would be exposed to Blueprint, despite an initial steep learning curve. However, there were also some major problems, the most serious one being the disappearing conveyors upon packaging. The lack of any useful error message or accessible documentation made it very difficult to work towards a solution; had Blueprints been used in isolation, this wouldn’t even have been an issue. In summary, for the most part it would definitely be recommended to use the two systems together; utilising the faster logic of C++ and the quick prototyping of new systems in Blueprints. However, if there is a bump in the road, it can be very time-consuming to get over it.

One key element that was missing within the project was tools to develop the environment of a level. The focus of the project was firmly on functionality, but when designing the demo puzzle, it would have been useful to have procedural tools for generating a walkway or physically wiring elements together. In addition to this, the project could be developed further by extending it into a short game. New mechanics could be slowly introduced, and through this process, more ideas for expanding the toolkit could be found.

In terms of improving the existing toolkit, one element that comes to mind is the conveyor. Currently, it is expensive to create long sections, and although there is a partial solution in hiding the mesh and modifying the spline itself, it would be useful to make the system itself more efficient. The first step to doing this would be finding an alternative to the spline for instancing meshes, as it is designed for full flexibility, not efficiency.

The nature of the project did often present challenges, as all systems needed to interact with objects entirely through physics, where the obvious implementation would be to fake these interactions. Working through these problems has added a deeper understanding of the physics systems in the Engine, with the project as a whole giving a good insight into the Unreal-specific additions to standard C++, as well as how to utilise the real-time de-bugging features of the Engine.

6 References

Bender, J., Muller, M. and Macklin, M., 2015. Position-Based Simulation Methods in Computer Graphics. *EUROGRAPHICS 2015* [online].Available from: http://mmacklin.com/EG2015PBD.pdf [Accessed 15 August 2020].

Epic Games, 2018. *Balancing Blueprint and C++* [online]. Available from: https://docs.unrealengine.com/en-US/Resources/SampleGames/ARPG/BalancingBlueprintAndCPP/index.html [Accessed 18 July 2020].

Epic Games, 2020. *AActor* [online]. Available from: https://docs.unrealengine.com/en-US/API/Runtime/Engine/GameFramework/AActor/index.html [Accessed 12 August 2020].

even, 2019. *Unreal Engine 4.21 Convert spline actor BP to c++ code* [online]. Available from: https://freelancerlife.info/en/blog/unreal-engine-421-convert-spline-actor-bp-c-code/ [Accessed 29 July 2020].

GeForce, 2011. *Screenshots- Portal 2* [online]. Available from: https://www.geforce.co.uk/games-applications/pc-games/portal-2/screenshots [Accessed 15 August 2020].

Portal, 2007. [game]. Designed by Kim Swift. US: Valve.

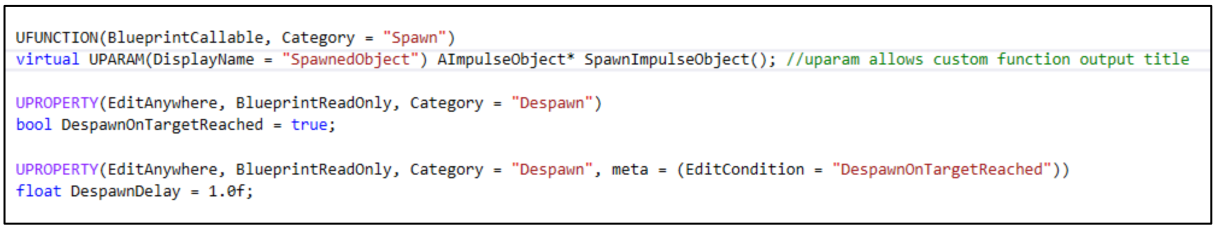
Portal 2, 2011. [game]. Directed by Joshua Weier. US: Valve.

SpaceGuard, 2020. *630mm wide Gravity roller conveyor- 75mm pitch* [online]. Available from: https://www.packingtables.co.uk/product/gravity-roller-conveyor-75mm-pitch/ [Accessed 24 July 2020].

Sweeney, T., 2015. *If You Love Something, Set It Free* [online]. Available from: https://www.unrealengine.com/en-US/blog/ue4-is-free [Accessed 11 August 2020].

7 Appendices

Appendix A:



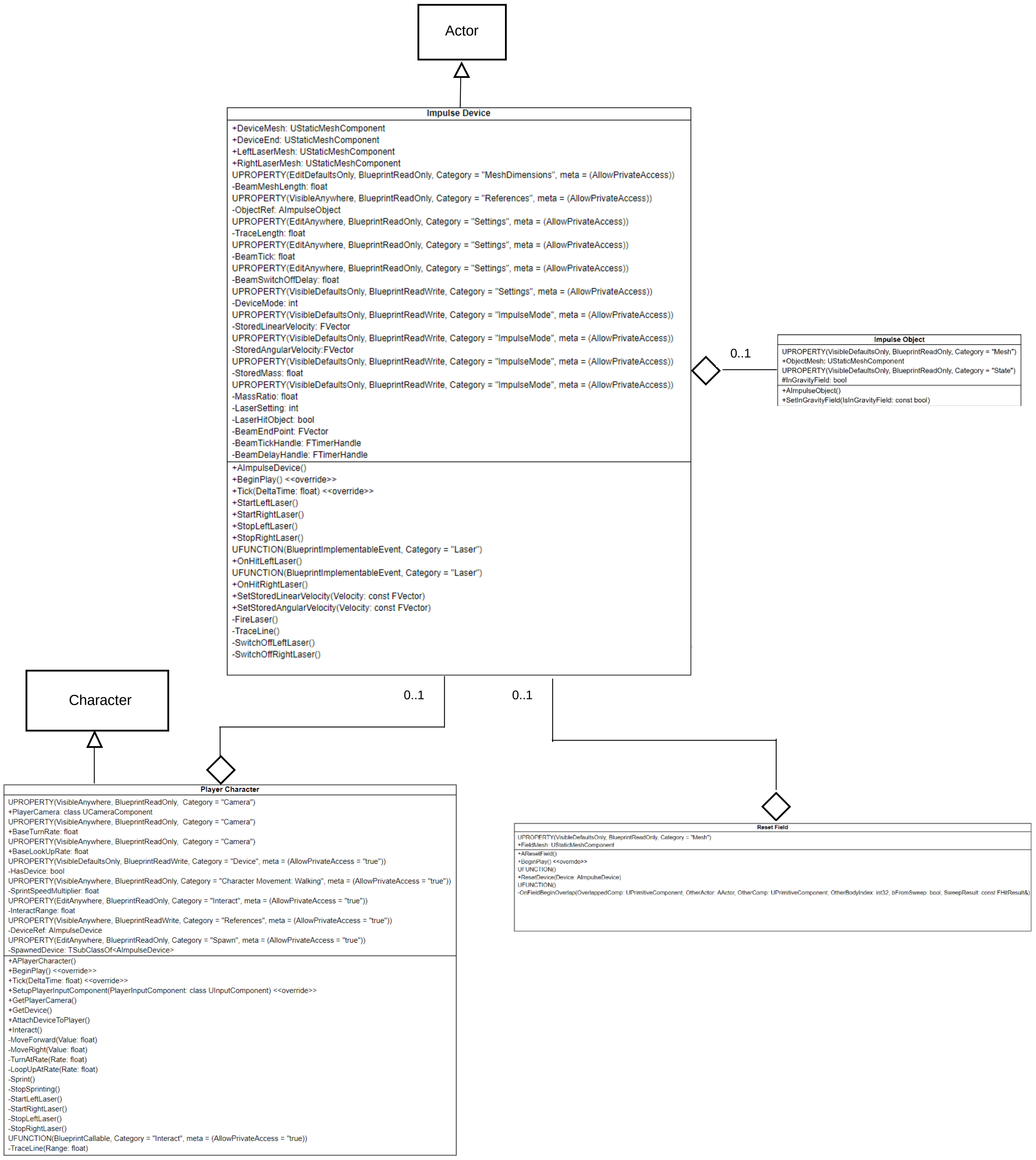
```
UFUNCTION(BlueprintCallable, Category = "Spawn")
virtual UPARAM(DisplayName = "SpawnedObject") AImpulseObject* SpawnImpulseObject(); //uparam allows custom function output title

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Despawn")
bool DespawnOnTargetReached = true;

UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Despawn", meta = (EditCondition = "DespawnOnTargetReached"))
float DespawnDelay = 1.0f;
```

Appendix A: Some examples of UPROPERTY and UFUNCTION specifiers in code

Appendix B:



Impulse Device

```
+DeviceMesh: UStaticMeshComponent
+DeviceEnd: UStaticMeshComponent
+LeftLaserMesh: UStaticMeshComponent
+RightLaserMesh: UStaticMeshComponent
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess))
-BeamMeshLength: float
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "References", meta = (AllowPrivateAccess))
-ObjectRef: AImpulseObject
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Settings", meta = (AllowPrivateAccess))
-TraceLength: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Settings", meta = (AllowPrivateAccess))
-BeamTick: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Settings", meta = (AllowPrivateAccess))
-BeamSwitchOffDelay: float
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "Settings", meta = (AllowPrivateAccess))
-DeviceMode: int
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "ImpulseMode", meta = (AllowPrivateAccess))
-StoredLinearVelocity: FVector
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "ImpulseMode", meta = (AllowPrivateAccess))
-StoredAngularVelocity:FVector
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "ImpulseMode", meta = (AllowPrivateAccess))
-StoredMass: float
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "ImpulseMode", meta = (AllowPrivateAccess))
-MassRatio: float
-LaserSetting: int
-LaserHitObject: bool
-BeamEndPoint: FVector
-BeamTickHandle: FTimerHandle
-BeamDelayHandle: FTimerHandle
---
+AImpulseDevice()
+BeginPlay() <<override>>
+Tick(DeltaTime: float) <<override>>
+StartLeftLaser()
+StartRightLaser()
+StopLeftLaser()
+StopRightLaser()
UFUNCTION(BlueprintImplementableEvent, Category = "Laser")
+OnHitLeftLaser()
UFUNCTION(BlueprintImplementableEvent, Category = "Laser")
+OnHitRightLaser()
+SetStoredLinearVelocity(Velocity: const FVector)
+SetStoredAngularVelocity(Velocity: const FVector)
-FireLaser()
-TraceLine()
-SwitchOffLeftLaser()
-SwitchOffRightLaser()
```

Impulse Object

```
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+ObjectMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "State")
#InGravityField: bool
---
+AImpulseObject()
+SetInGravityField(IsInGravityField: const bool)
```

Player Character

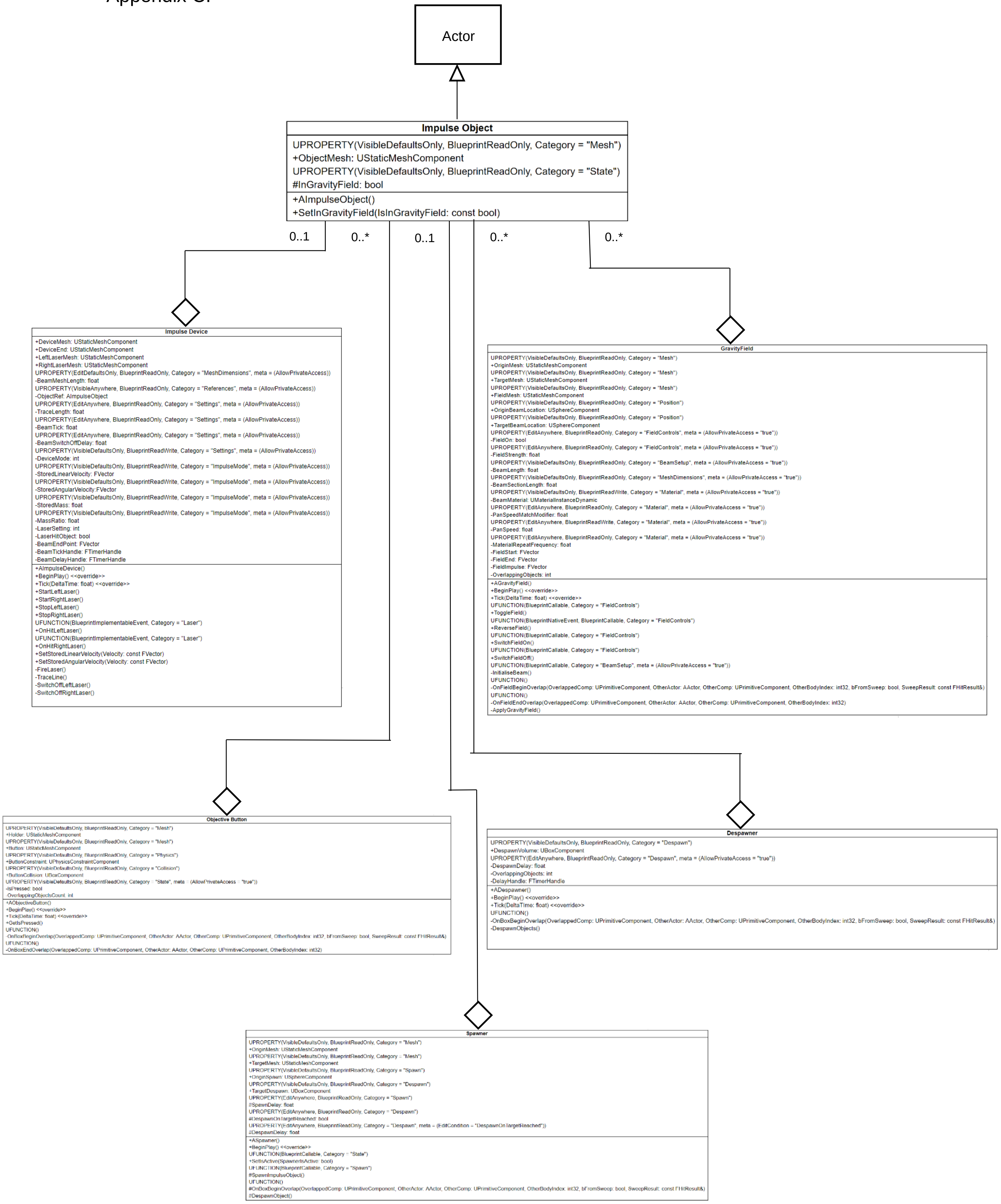
```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly,  Category = "Camera")
+PlayerCamera: class UCameraComponent
UPROPERTY(VisibleAnywhere, BlueprintReadOnly,  Category = "Camera")
+BaseTurnRate: float
UPROPERTY(VisibleAnywhere, BlueprintReadOnly,  Category = "Camera")
+BaseLookUpRate: float
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "Device", meta = (AllowPrivateAccess = "true"))
-HasDevice: bool
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Character Movement: Walking", meta = (AllowPrivateAccess = "true"))
-SprintSpeedMultiplier: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Interact", meta = (AllowPrivateAccess = "true"))
-InteractRange: float
UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = "References", meta = (AllowPrivateAccess = "true"))
-DeviceRef: AImpulseDevice
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawn", meta = (AllowPrivateAccess = "true"))
-SpawnedDevice: TSubClassOf<AImpulseDevice>
---
+APlayerCharacter()
+BeginPlay() <<override>>
+Tick(DeltaTime: float) <<override>>
+SetupPlayerInputComponent(PlayerInputComponent: class UInputComponent) <<override>>
+GetPlayerCamera()
+GetDevice()
+AttachDeviceToPlayer()
+Interact()
-MoveForward(Value: float)
-MoveRight(Value: float)
-TurnAtRate(Rate: float)
-LoopUpAtRate(Rate: float)
-Sprint()
-StopSprinting()
-StartLeftLaser()
-StartRightLaser()
-StopLeftLaser()
-StopRightLaser()
UFUNCTION(BlueprintCallable, Category = "Interact", meta = (AllowPrivateAccess = "true))
-TraceLine(Range: float)
```

Reset Field

```
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+FieldMesh: UStaticMeshComponent
---
+AResetField()
+BeginPlay() <<override>>
UFUNCTION()
+ResetDevice(Device: AImpulseDevice)
UFUNCTION()
-OnFieldBeginOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32, bFromSweep: bool, SweepResult: const FHitResult&)
```

Appendix B: Class diagram for the Impulse Device class

Appendix C:



Impulse Object
```
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+ObjectMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "State")
#InGravityField: bool
+AImpulseObject()
+SetInGravityField(IsInGravityField: const bool)
```

Impulse Device
```
+DeviceMesh: UStaticMeshComponent
+DeviceEnd: UStaticMeshComponent
+LeftLaserMesh: UStaticMeshComponent
+RightLaserMesh: UStaticMeshComponent
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess))
-BeamMeshLength: float
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "References", meta = (AllowPrivateAccess))
-ObjectRef: AImpulseObject
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Settings", meta = (AllowPrivateAccess))
-TraceLength: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Settings", meta = (AllowPrivateAccess))
-BeamTick: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Settings", meta = (AllowPrivateAccess))
-BeamSwitchOffDelay: float
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "Settings", meta = (AllowPrivateAccess))
-DeviceMode: int
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "ImpulseMode", meta = (AllowPrivateAccess))
-StoredLinearVelocity: FVector
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "ImpulseMode", meta = (AllowPrivateAccess))
-StoredAngularVelocity:FVector
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "ImpulseMode", meta = (AllowPrivateAccess))
-StoredMass: float
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "ImpulseMode", meta = (AllowPrivateAccess))
-MassRatio: float
-LaserSetting: int
-LaserHitObject: bool
-BeamEndPoint: FVector
-BeamTickHandle: FTimerHandle
-BeamDelayHandle: FTimerHandle
+AImpulseDevice()
+BeginPlay() <<override>>
+Tick(DeltaTime: float) <<override>>
+StartLeftLaser()
+StartRightLaser()
+StopLeftLaser()
+StopRightLaser()
UFUNCTION(BlueprintImplementableEvent, Category = "Laser")
+OnHitLeftLaser()
UFUNCTION(BlueprintImplementableEvent, Category = "Laser")
+OnHitRightLaser()
+SetStoredLinearVelocity(Velocity: const FVector)
+SetStoredAngularVelocity(Velocity: const FVector)
-FireLaser()
-TraceLine()
-SwitchOffLeftLaser()
-SwitchOffRightLaser()
```

GravityField
```
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+OriginMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+TargetMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+FieldMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Position")
+OriginBeamLocation: USphereComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Position")
+TargetBeamLocation: USphereComponent
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "FieldControls", meta = (AllowPrivateAccess = "true"))
-FieldOn: bool
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "FieldControls", meta = (AllowPrivateAccess = "true"))
-FieldStrength: float
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "BeamSetup", meta = (AllowPrivateAccess = "true"))
-BeamLength: float
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess = "true"))
-BeamSectionLength: float
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "Material", meta = (AllowPrivateAccess = "true"))
-BeamMaterial: UMaterialInstanceDynamic
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Material", meta = (AllowPrivateAccess = "true"))
-PanSpeedMatchModifier: float
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Material", meta = (AllowPrivateAccess = "true"))
-PanSpeed: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Material", meta = (AllowPrivateAccess = "true"))
-MaterialRepeatFrequency: float
-FieldStart: FVector
-FieldEnd: FVector
-FieldImpulse: FVector
-OverlappingObjects: int
+AGravityField()
+BeginPlay() <<override>>
+Tick(DeltaTime: float) <<override>>
UFUNCTION(BlueprintCallable, Category = "FieldControls")
+ToggleField()
UFUNCTION(BlueprintNativeEvent, BlueprintCallable, Category = "FieldControls")
+ReverseField()
UFUNCTION(BlueprintCallable, Category = "FieldControls")
+SwitchFieldOn()
UFUNCTION(BlueprintCallable, Category = "FieldControls")
+SwitchFieldOff()
UFUNCTION(BlueprintCallable, Category = "BeamSetup", meta = (AllowPrivateAccess = "true"))
-InitialiseBeam()
UFUNCTION()
-OnFieldBeginOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32, bFromSweep: bool, SweepResult: const FHitResult&)
UFUNCTION()
-OnFieldEndOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32)
-ApplyGravityField()
```

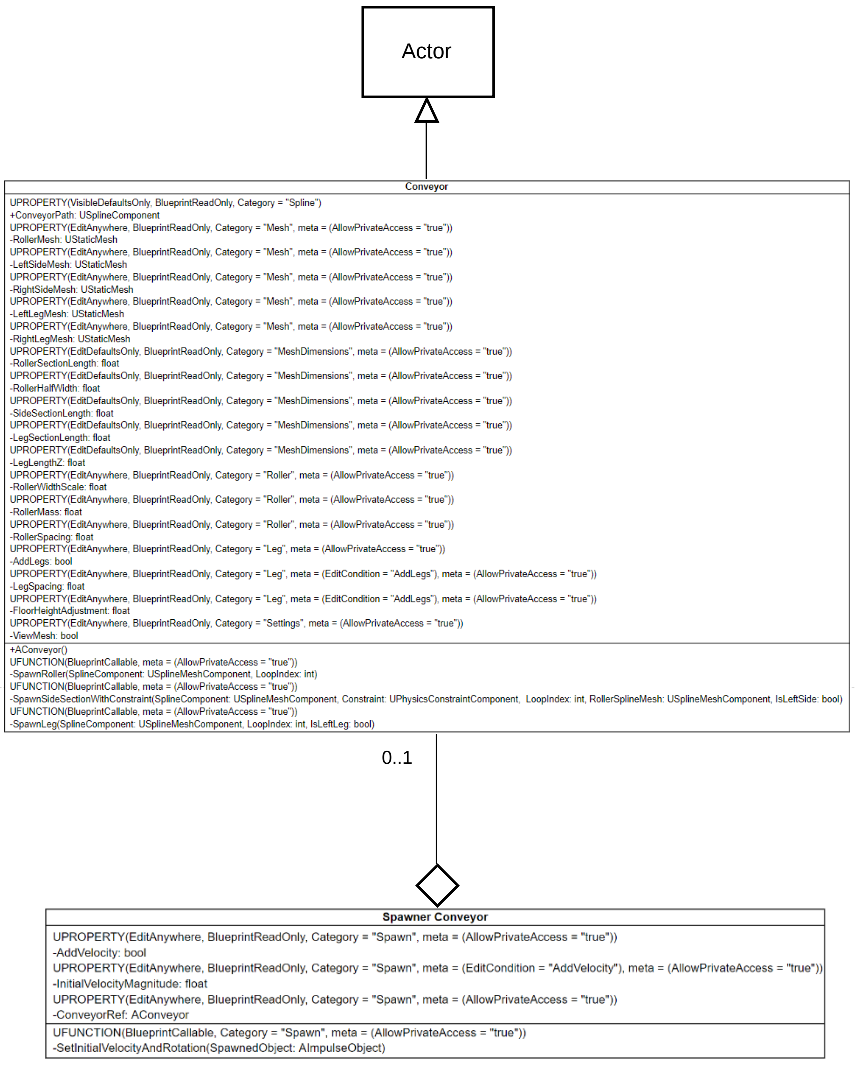
Objective Button
```
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+Holder: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+Button: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Physics")
+ButtonConstraint: UPhysicsConstraintComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Collision")
+ButtonCollision: UBoxComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "State", meta = (AllowPrivateAccess = "true"))
-IsPressed: bool
-OverlappingObjectsCount: int
+AObjectiveButton()
+BeginPlay() <<override>>
+Tick(DeltaTime: float) <<override>>
+GetIsPressed()
UFUNCTION()
-OnBoxBeginOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32, bFromSweep: bool, SweepResult: const FHitResult&)
UFUNCTION()
-OnBoxEndOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32)
```

Despawner
```
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Despawn")
+DespawnVolume: UBoxComponent
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Despawn", meta = (AllowPrivateAccess = "true"))
-DespawnDelay: float
-OverlappingObjects: int
-DelayHandle: FTimerHandle
+ADespawner()
+BeginPlay() <<override>>
+Tick(DeltaTime: float) <<override>>
UFUNCTION()
-OnBoxBeginOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32, bFromSweep: bool, SweepResult: const FHitResult&)
-DespawnObjects()
```

Spawner
```
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+OriginMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+TargetMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Spawn")
+OriginSpawn: USphereComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Despawn")
+TargetDespawn: UBoxComponent
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawn")
#SpawnDelay: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Despawn")
#DespawnOnTargetReached: bool
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Despawn", meta = (EditCondition = "DespawnOnTargetReached"))
#DespawnDelay: float
+ASpawner()
+BeginPlay() <<override>>
UFUNCTION(BlueprintCallable, Category = "State")
+SetIsActive(SpawnerIsActive: bool)
UFUNCTION(BlueprintCallable, Category = "Spawn")
#SpawnImpulseObject()
UFUNCTION()
#OnBoxBeginOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32, bFromSweep: bool, SweepResult: const FHitResult&)
#DespawnObject()
```

Appendix C: Class diagram for the Impulse Object class

Appendix D:

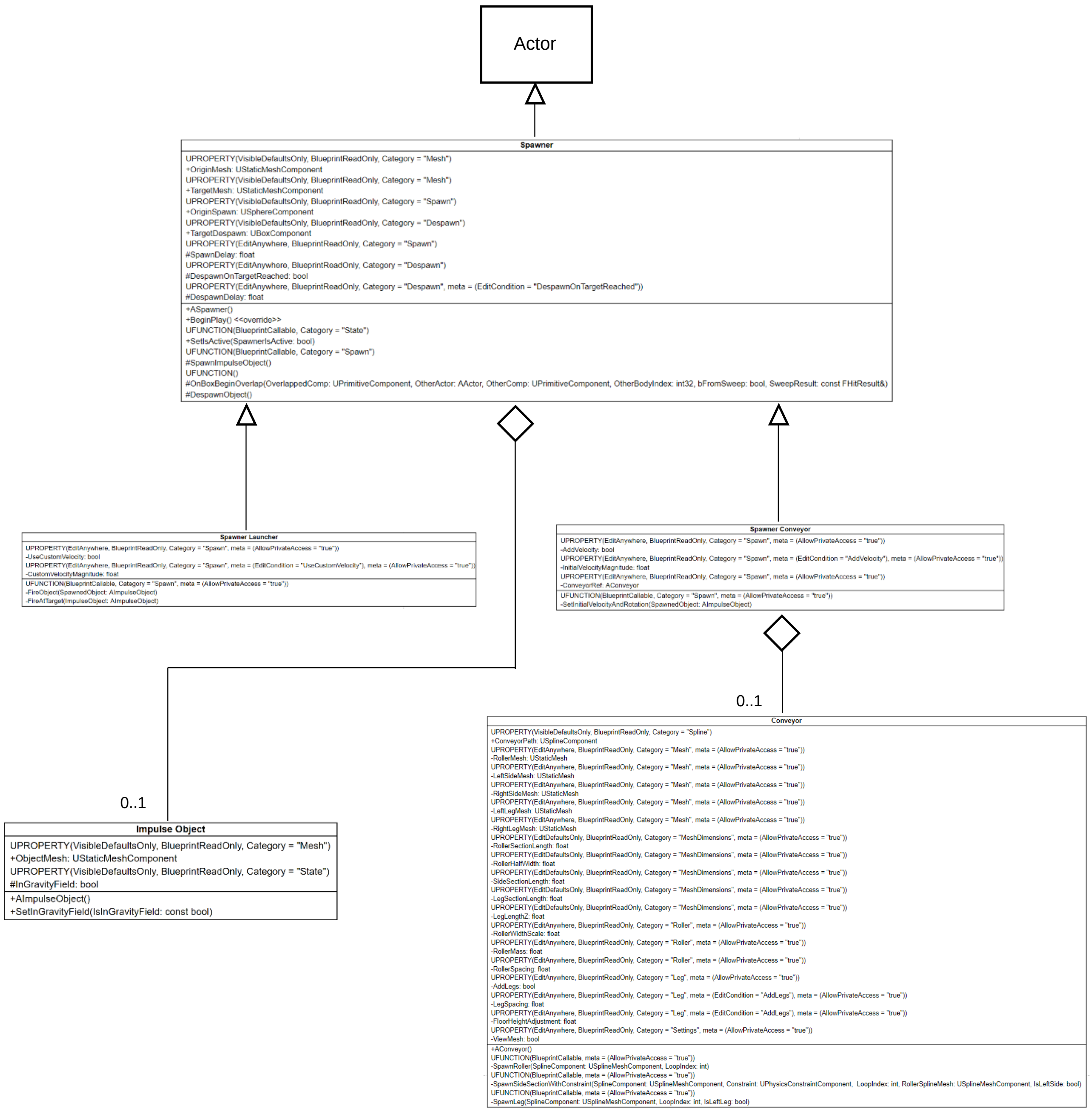


```
Conveyor
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Spline")
+ConveyorPath: USplineComponent
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Mesh", meta = (AllowPrivateAccess = "true"))
-RollerMesh: UStaticMesh
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Mesh", meta = (AllowPrivateAccess = "true"))
-LeftSideMesh: UStaticMesh
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Mesh", meta = (AllowPrivateAccess = "true"))
-RightSideMesh: UStaticMesh
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Mesh", meta = (AllowPrivateAccess = "true"))
-LeftLegMesh: UStaticMesh
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Mesh", meta = (AllowPrivateAccess = "true"))
-RightLegMesh: UStaticMesh
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess = "true"))
-RollerSectionLength: float
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess = "true"))
-RollerHalfWidth: float
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess = "true"))
-SideSectionLength: float
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess = "true"))
-LegSectionLength: float
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess = "true"))
-LegLengthZ: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Roller", meta = (AllowPrivateAccess = "true"))
-RollerWidthScale: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Roller", meta = (AllowPrivateAccess = "true"))
-RollerMass: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Roller", meta = (AllowPrivateAccess = "true"))
-RollerSpacing: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Leg", meta = (AllowPrivateAccess = "true"))
-AddLegs: bool
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Leg", meta = (EditCondition = "AddLegs"), meta = (AllowPrivateAccess = "true"))
-LegSpacing: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Leg", meta = (EditCondition = "AddLegs"), meta = (AllowPrivateAccess = "true"))
-FloorHeightAdjustment: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Settings", meta = (AllowPrivateAccess = "true"))
-ViewMesh: bool
+AConveyor()
UFUNCTION(BlueprintCallable, meta = (AllowPrivateAccess = "true"))
-SpawnRoller(SplineComponent: USplineMeshComponent, LoopIndex: int)
UFUNCTION(BlueprintCallable, meta = (AllowPrivateAccess = "true"))
-SpawnSideSectionWithConstraint(SplineComponent: USplineMeshComponent, Constraint: UPhysicsConstraintComponent,  LoopIndex: int, RollerSplineMesh: USplineMeshComponent, IsLeftSide: bool)
UFUNCTION(BlueprintCallable, meta = (AllowPrivateAccess = "true"))
-SpawnLeg(SplineComponent: USplineMeshComponent, LoopIndex: int, IsLeftLeg: bool)
```

```
Spawner Conveyor
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawn", meta = (AllowPrivateAccess = "true"))
-AddVelocity: bool
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawn", meta = (EditCondition = "AddVelocity"), meta = (AllowPrivateAccess = "true"))
-InitialVelocityMagnitude: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawn", meta = (AllowPrivateAccess = "true"))
-ConveyorRef: AConveyor
UFUNCTION(BlueprintCallable, Category = "Spawn", meta = (AllowPrivateAccess = "true"))
-SetInitialVelocityAndRotation(SpawnedObject: AImpulseObject)
```

Appendix D: Class diagram for the Conveyor class

Appendix E:



```
Spawner
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+OriginMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+TargetMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Spawn")
+OriginSpawn: USphereComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Despawn")
+TargetDespawn: UBoxComponent
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawn")
#SpawnDelay: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Despawn")
#DespawnOnTargetReached: bool
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Despawn", meta = (EditCondition = "DespawnOnTargetReached"))
#DespawnDelay: float
+ASpawner()
+BeginPlay() <<override>>
UFUNCTION(BlueprintCallable, Category = "State")
+SetIsActive(SpawnerIsActive: bool)
UFUNCTION(BlueprintCallable, Category = "Spawn")
#SpawnImpulseObject()
UFUNCTION()
#OnBoxBeginOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32, bFromSweep: bool, SweepResult: const FHitResult&)
#DespawnObject()
```

```
Spawner Launcher
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawn", meta = (AllowPrivateAccess = "true"))
-UseCustomVelocity: bool
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawn", meta = (EditCondition = "UseCustomVelocity"), meta = (AllowPrivateAccess = "true"))
-CustomVelocityMagnitude: float
UFUNCTION(BlueprintCallable, Category = "Spawn", meta = (AllowPrivateAccess = "true"))
-FireObject(SpawnedObject: AImpulseObject)
-FireAtTarget(ImpulseObject: AImpulseObject)
```

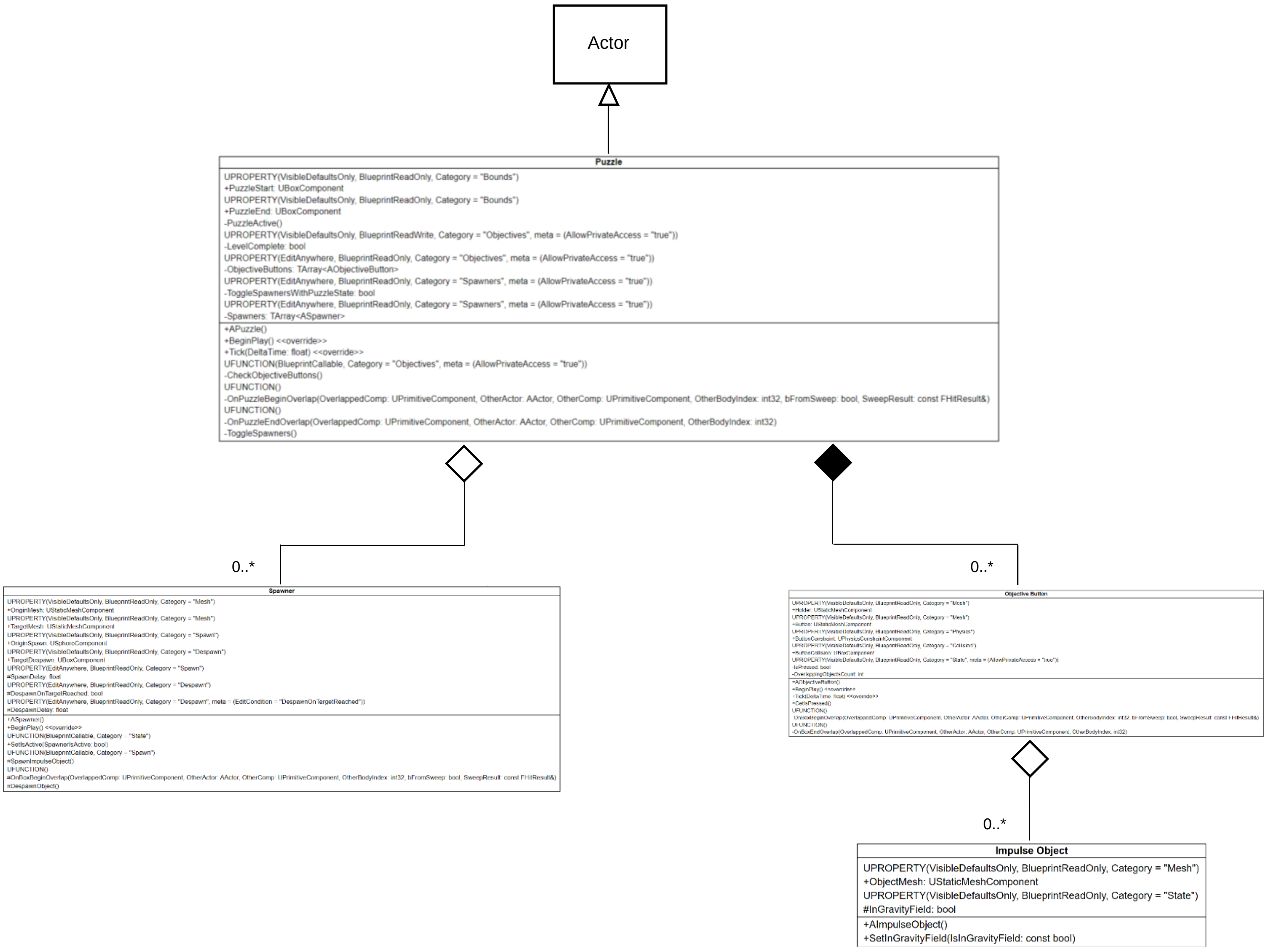
```
Spawner Conveyor
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawn", meta = (AllowPrivateAccess = "true"))
-AddVelocity: bool
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawn", meta = (EditCondition = "AddVelocity"), meta = (AllowPrivateAccess = "true"))
-InitialVelocityMagnitude: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawn", meta = (AllowPrivateAccess = "true"))
-ConveyorRef: AConveyor
UFUNCTION(BlueprintCallable, Category = "Spawn", meta = (AllowPrivateAccess = "true"))
-SetInitialVelocityAndRotation(SpawnedObject: AImpulseObject)
```

```
Impulse Object
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+ObjectMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "State")
#InGravityField: bool
+AImpulseObject()
+SetInGravityField(IsInGravityField: const bool)
```

```
Conveyor
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Spline")
+ConveyorPath: USplineComponent
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Mesh", meta = (AllowPrivateAccess = "true"))
-RollerMesh: UStaticMesh
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Mesh", meta = (AllowPrivateAccess = "true"))
-LeftSideMesh: UStaticMesh
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Mesh", meta = (AllowPrivateAccess = "true"))
-RightSideMesh: UStaticMesh
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Mesh", meta = (AllowPrivateAccess = "true"))
-LeftLegMesh: UStaticMesh
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Mesh", meta = (AllowPrivateAccess = "true"))
-RightLegMesh: UStaticMesh
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess = "true"))
-RollerSectionLength: float
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess = "true"))
-RollerHalfWidth: float
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess = "true"))
-SideSectionLength: float
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess = "true"))
-LegSectionLength: float
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess = "true"))
-LegLengthZ: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Roller", meta = (AllowPrivateAccess = "true"))
-RollerWidthScale: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Roller", meta = (AllowPrivateAccess = "true"))
-RollerMass: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Roller", meta = (AllowPrivateAccess = "true"))
-RollerSpacing: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Leg", meta = (AllowPrivateAccess = "true"))
-AddLegs: bool
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Leg", meta = (EditCondition = "AddLegs"), meta = (AllowPrivateAccess = "true"))
-LegSpacing: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Leg", meta = (EditCondition = "AddLegs"), meta = (AllowPrivateAccess = "true"))
-FloorHeightAdjustment: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Settings", meta = (AllowPrivateAccess = "true"))
-ViewMesh: bool
+AConveyor()
UFUNCTION(BlueprintCallable, meta = (AllowPrivateAccess = "true"))
-SpawnRoller(SplineComponent: USplineMeshComponent, LoopIndex: int)
UFUNCTION(BlueprintCallable, meta = (AllowPrivateAccess = "true"))
-SpawnSideSectionWithConstraint(SplineComponent: USplineMeshComponent, Constraint: UPhysicsConstraintComponent,  LoopIndex: int, RollerSplineMesh: USplineMeshComponent, IsLeftSide: bool)
UFUNCTION(BlueprintCallable, meta = (AllowPrivateAccess = "true"))
-SpawnLeg(SplineComponent: USplineMeshComponent, LoopIndex: int, IsLeftLeg: bool)
```

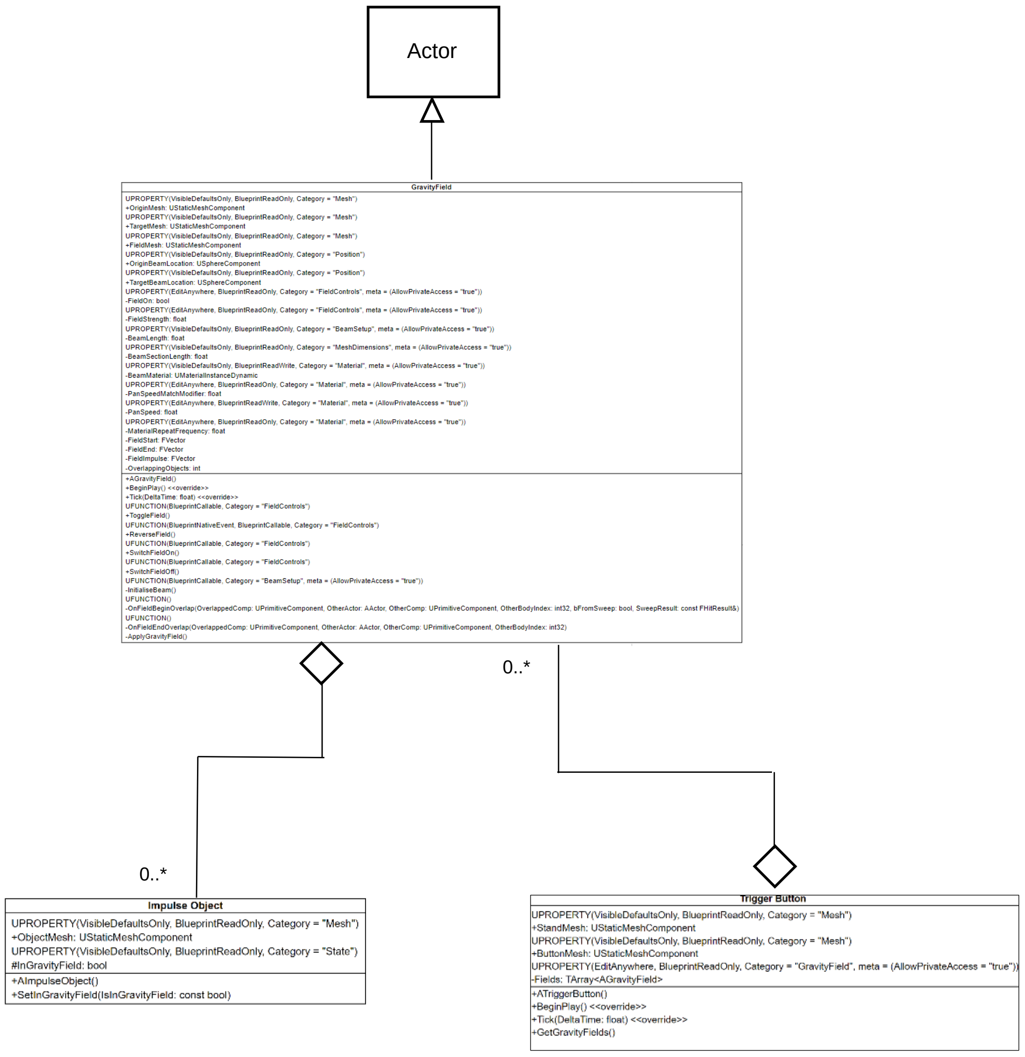
Appendix E: Class diagram for the Spawner classes

Appendix F:


Actor
Puzzle
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Bounds")
+PuzzleStart: UBoxComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Bounds")
+PuzzleEnd: UBoxComponent
-PuzzleActive()
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "Objectives", meta = (AllowPrivateAccess = "true"))
-LevelComplete: bool
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Objectives", meta = (AllowPrivateAccess = "true"))
-ObjectiveButtons: TArray<AObjectiveButton>
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawners", meta = (AllowPrivateAccess = "true"))
-ToggleSpawnersWithPuzzleState: bool
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawners", meta = (AllowPrivateAccess = "true"))
-Spawners: TArray<ASpawner>
+APuzzle()
+BeginPlay() <<override>>
+Tick(DeltaTime: float) <<override>>
UFUNCTION(BlueprintCallable, Category = "Objectives", meta = (AllowPrivateAccess = "true"))
-CheckObjectiveButtons()
UFUNCTION()
-OnPuzzleBeginOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32, bFromSweep: bool, SweepResult: const FHitResult&)
UFUNCTION()
-OnPuzzleEndOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32)
-ToggleSpawners()
0..*
Spawner
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+OriginMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+TargetMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Spawn")
+OriginSpawn: USphereComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Despawn")
+TargetDespawn: UBoxComponent
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Spawn")
#SpawnDelay: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Despawn")
#DespawnOnTargetReached: bool
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Despawn", meta = (EditCondition = "DespawnOnTargetReached"))
#DespawnDelay: float
+ASpawner()
+BeginPlay() <<override>>
UFUNCTION(BlueprintCallable, Category = "State")
+SetIsActive(SpawnerIsActive: bool)
UFUNCTION(BlueprintCallable, Category = "Spawn")
#SpawnImpulseObject()
UFUNCTION()
#OnBoxBeginOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32, bFromSweep: bool, SweepResult: const FHitResult&)
#DespawnObject()
0..*
Objective Button
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+Holder: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+Button: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Physics")
+ButtonConstraint: UPhysicsConstraintComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Collision")
+ButtonCollision: UBoxComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "State", meta = (AllowPrivateAccess = "true"))
-IsPressed: bool
-OverlappingObjectsCount: int
+AObjectiveButton()
+BeginPlay() <<override>>
+Tick(DeltaTime: float) <<override>>
+GetIsPressed()
UFUNCTION()
-OnBoxBeginOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32, bFromSweep: bool, SweepResult: const FHitResult&)
UFUNCTION()
-OnBoxEndOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32)
0..*
Impulse Object
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+ObjectMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "State")
#InGravityField: bool
+AImpulseObject()
+SetInGravityField(IsInGravityField: const bool)

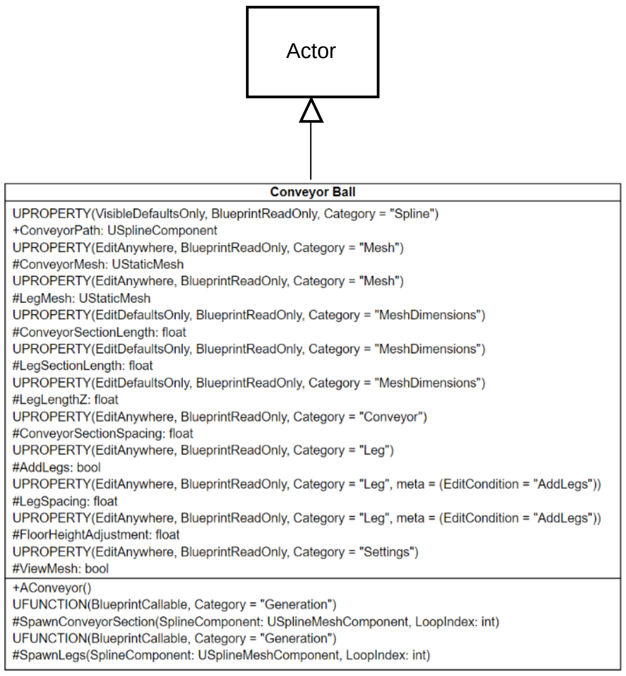
Appendix F: Class diagram for the Puzzle and Objective Button classes

Appendix G:


Actor
GravityField
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+OriginMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+TargetMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+FieldMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Position")
+OriginBeamLocation: USphereComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Position")
+TargetBeamLocation: USphereComponent
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "FieldControls", meta = (AllowPrivateAccess = "true"))
-FieldOn: bool
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "FieldControls", meta = (AllowPrivateAccess = "true"))
-FieldStrength: float
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "BeamSetup", meta = (AllowPrivateAccess = "true"))
-BeamLength: float
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions", meta = (AllowPrivateAccess = "true"))
-BeamSectionLength: float
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = "Material", meta = (AllowPrivateAccess = "true"))
-BeamMaterial: UMaterialInstanceDynamic
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Material", meta = (AllowPrivateAccess = "true"))
-PanSpeedMatchModifier: float
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Material", meta = (AllowPrivateAccess = "true"))
-PanSpeed: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Material", meta = (AllowPrivateAccess = "true"))
-MaterialRepeatFrequency: float
-FieldStart: FVector
-FieldEnd: FVector
-FieldImpulse: FVector
-OverlappingObjects: int
+AGravityField()
+BeginPlay() <<override>>
+Tick(DeltaTime: float) <<override>>
UFUNCTION(BlueprintCallable, Category = "FieldControls")
+ToggleField()
UFUNCTION(BlueprintNativeEvent, BlueprintCallable, Category = "FieldControls")
+ReverseField()
UFUNCTION(BlueprintCallable, Category = "FieldControls")
+SwitchFieldOn()
UFUNCTION(BlueprintCallable, Category = "FieldControls")
+SwitchFieldOff()
UFUNCTION(BlueprintCallable, Category = "BeamSetup", meta = (AllowPrivateAccess = "true"))
-InitialiseBeam()
UFUNCTION()
-OnFieldBeginOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32, bFromSweep: bool, SweepResult: const FHitResult&)
UFUNCTION()
-OnFieldEndOverlap(OverlappedComp: UPrimitiveComponent, OtherActor: AActor, OtherComp: UPrimitiveComponent, OtherBodyIndex: int32)
-ApplyGravityField()
0..*
0..*
Impulse Object
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+ObjectMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "State")
#InGravityField: bool
+AImpulseObject()
+SetInGravityField(IsInGravityField: const bool)
Trigger Button
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+StandMesh: UStaticMeshComponent
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Mesh")
+ButtonMesh: UStaticMeshComponent
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "GravityField", meta = (AllowPrivateAccess = "true"))
-Fields: TArray<AGravityField>
+ATriggerButton()
+BeginPlay() <<override>>
+Tick(DeltaTime: float) <<override>>
+GetGravityFields()

Appendix G: Class diagram for the Gravity Field and Trigger Button classes

Appendix H:



```
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Spline")
+ConveyorPath: USplineComponent
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Mesh")
#ConveyorMesh: UStaticMesh
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Mesh")
#LegMesh: UStaticMesh
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions")
#ConveyorSectionLength: float
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions")
#LegSectionLength: float
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "MeshDimensions")
#LegLengthZ: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Conveyor")
#ConveyorSectionSpacing: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Leg")
#AddLegs: bool
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Leg", meta = (EditCondition = "AddLegs"))
#LegSpacing: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Leg", meta = (EditCondition = "AddLegs"))
#FloorHeightAdjustment: float
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Settings")
#ViewMesh: bool
```

```
+AConveyor()
UFUNCTION(BlueprintCallable, Category = "Generation")
#SpawnConveyorSection(SplineComponent: USplineMeshComponent, LoopIndex: int)
UFUNCTION(BlueprintCallable, Category = "Generation")
#SpawnLegs(SplineComponent: USplineMeshComponent, LoopIndex: int)
```

Appendix H: Class diagram for the Conveyor Ball class