Laser System Documentation

March 2023

version 1.0.2

Contents

1 Greetings

Thank you for downloading my first asset, the Laser System which was inspired by the hit game Portal 2. The package comes with meshes, textures, particle systems and prefabs ready for use and automated tests that validate them. It also includes the *.blend*[1] files of the exported assets as well as the high resolution meshes that were used to bake out normal maps.
The asset does not expect the developer to know any coding besides the very basics so, in that sense, it is "artist friendly". Altough, to fully appreciate all of its features and to build upon it quite a bit of coding experience is needed.
The guide is written in an educational manner aiming to contain as much useful information as possible, or at least link to the official documentations for ease of use. If you are a novice at game/software development then this guide should prove very useful to you.

If the asset was useful to you please consider dropping a good review on the Asset Store so others can find it.

If you found something missing or not quite right, do not hesitate to write an email with your feedback.
I hope you enjoy working with the asset as much as I enjoyed creating it and if you wonder how such a system is implemented, feel free to check out the technical details in this documentation.

2 About the asset

The laser system provides a set of *Laser Actor*s that can interact with each other and notify any subscribers about the state of the system. It can also be queried at the developer's will to see its current state. In the code as well as in this documentation, *Laser Actor*s are the GameObjects[2] that are part of the laser system. Anything else will be treated as a *non actor* or sometimes just referred to as a *wall*.

While drawing out lasers is nice, the system can also be queried for its state and its actors publish events[3] that will notify the subscribers when specific interactions between the actors take place. With these queries and events at the developer's fingertips common game events such as a level complete for a puzzle game can be quickly and easily implemented.

[1] The meshes for the project were modeled using the Blender 3D software.
[2] GameObjects in Unity are fundamental elements that represent objects in the scene.
[3] Events enable a class or object to notify other classes or objects when something of interest occurs.

3 The *TL;DR* guide - long only due to pictures!

To get up to speed as fast as possible, follow the guide below.

3.1 Laser Actor hierarchy - what is what

The actors are grouped based on their similar properties and those groups have specific names. Understanding that makes navigating the project easier.

- **Laser Actor**s are the *GameObject*s part of the laser system.
- **Laser Forwarder**s can propagate a laser beam - either their own or someone else's.
- **Laser Emitter**s can shoot their **own** laser beam.
- **Laser Receiver**s can be shot by a laser beam and process it.
- **Laser Relay**s **cannot** shoot their own beam but can forward an emitter's beam.
- **Laser Target**s and distinguished Receivers - they are end targets for an emitter. Their existence makes querying the system easier.

```
Laser Actors
|
 — Laser Forwarders
   |
    — Laser Emitters
   |
    — Laser Relays
|
 — Laser Receivers
   |
    — Laser Relays
   |
    — Laser Targets
```

These are just **groups** of actors. The specific entities are detailed below. Note that **Laser Relay** is listed twice. That is because *Relays* are both *Receivers* and *Forwarders*.

3.2 Laser Actors in a nutshell

3.2.1 Laser Emitter

Laser Emitters are the actors that can shoot out lasers to seek targets. This is the only entity that is part of the group *Laser Emitter*s.

- They are the only actors that run a loop. More active emitters = more CPU used.
- You can change the laser maximum distance as well as the laser's the maximum update frequency in FPS[4]. Higher FPS = more CPU used.
- They disregard being shot by another emitter.
- Two prefabs are present - one that is part of a cube and one that is standalone - it can be added to walls you modeled for example.

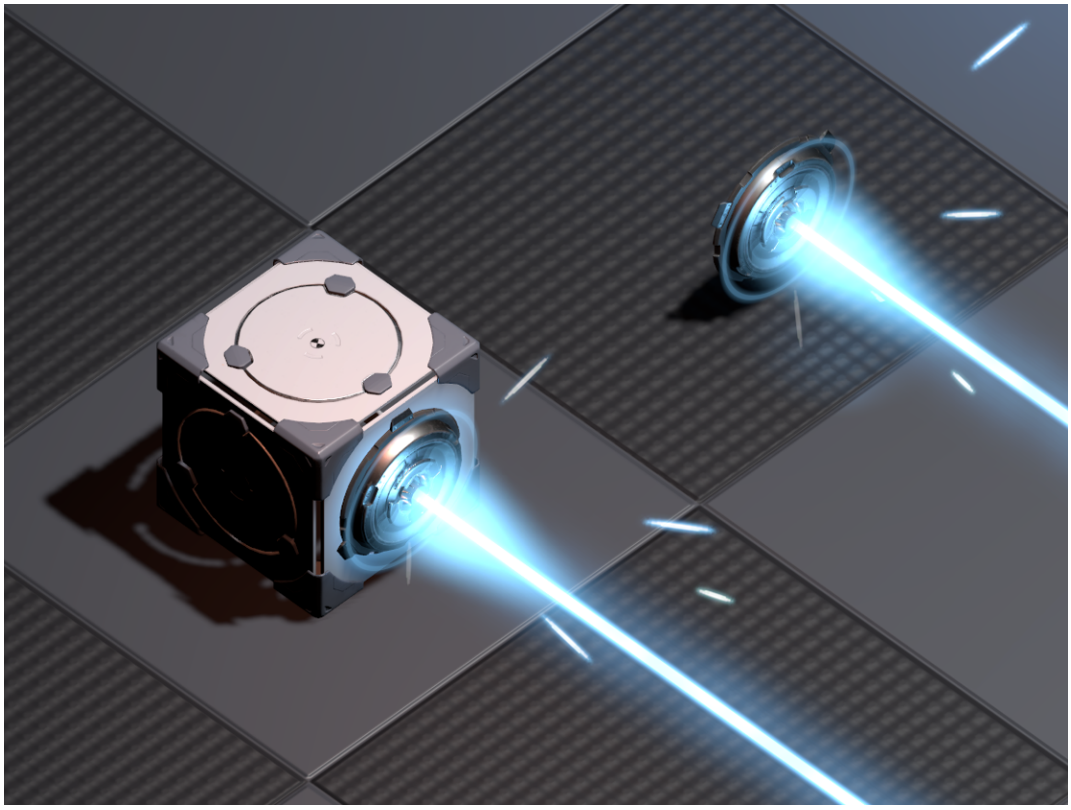


Figure 1: Laser emitter standalone or in a cube

[4]Frames per second is the frequency of image update.

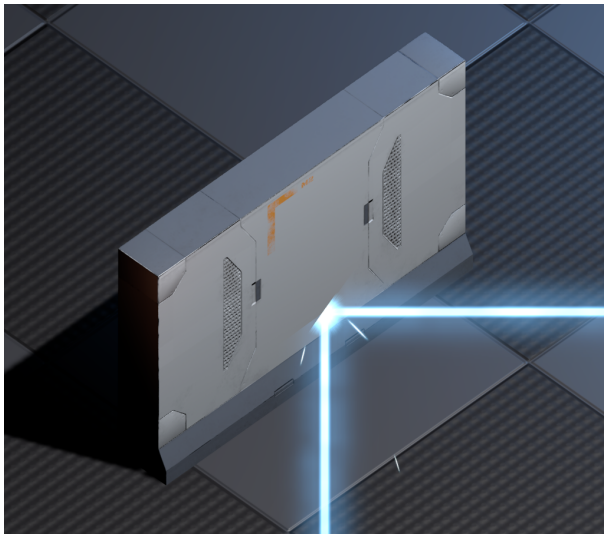
3.2.2 Laser Mirror

Laser Mirrors are *Laser Relay*s that forward the laser beam by reflecting it.

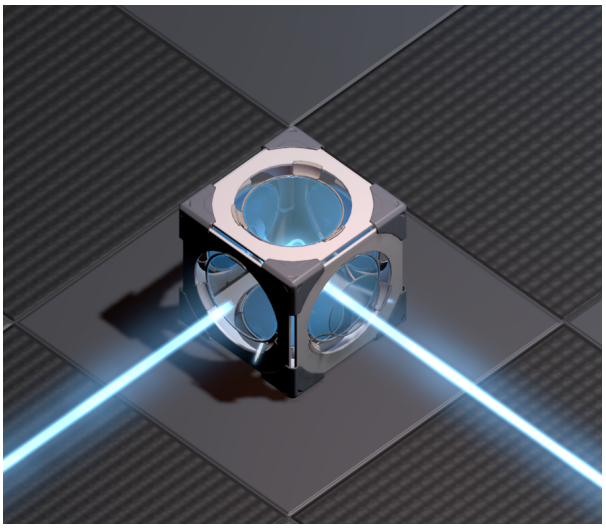
- As with all reflections, the direction of the forwarded beam depends on the angle between the emitter and the mirror.

3.2.3 Laser Repeater

Laser Repeaters are *Laser Relay*s that always forward the incoming laser beam in their own *forward* direction, that is, where they are facing.



(a) Laser mirror



(b) Laser repeater

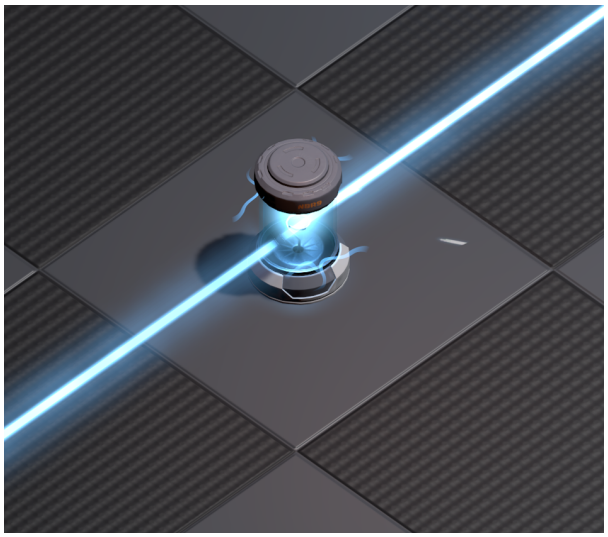
Figure 2: Mirror and repeater

3.2.4 Nonblocking Laser Receiver

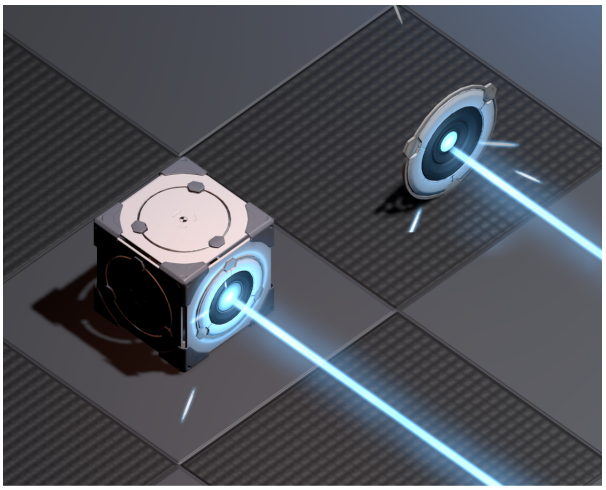
A nonblocking receiver is a *Laser Relay* that is also a *Laser Target*. It lets the laser pass through it so the laser can hit other objects as well.

3.2.5 Blocking Laser Receiver

A blocking receiver is a *Laser Target* but it is **not** a *Laser Relay*. It can be shot with a beam but it is an end destination - it does not forward that laser beam in any way.



(a) Nonblocking laser receiver



(b) Blocking laser receiver - standalone or in a cube

Figure 3: Nonblocking- and blocking laser receivers

3.3 How to get these actors to the scene

The actors are set up as *prefabs* - By default they are at the

Assets > Resources > Prefabs > LaserActors

location. To use one just drag and drop them to the scene in Unity Editor.

3.4 How to get data out of these actors (example)

To spawn or hide the beams or particle systems there is nothing that needs to be done, the system handles that for you. But to get notified of events that take place you'll need a little bit of scripting - let me show that with an example.

```
1  public class ScriptingExample : LaserActorAware {
2      private IQueryableLaserTarget _mySelectedTarget;
3
4      private void Start() {
5          SubscribeToSomeRandomEvents();
6          FindMyTargetAndSubscribeToIt();
7      }
8      private void FindMyTargetAndSubscribeToIt() {
9          // use LaserActorAware's query methods to find the actor
10
11         _mySelectedTarget =
12             FindLaserActorByRootName<IQueryableLaserTarget>("nonblocking_receiver");
13
14         // subscribe to the new emitter received event
15         _mySelectedTarget.OnNewEmitterReceived +=
16             Method_To_Run_When_New_Emitter_Hits_My_Target;
17     }
18     private void Method_To_Run_When_New_Emitter_Hits_My_Target(
19         IQueryableLaserReceiver sender, LaserHit laserHit) {
20
21         // this will run when a new emitter hits my target
22         Debug.Log($"My favourite target, {_mySelectedTarget.name} " +
23         "was hit by an emitter: {laserHit.Emitter.name}");
24     }
25     private void SubscribeToSomeRandomEvents() {
26
27         // event subscriptions again
28
29         OnAllTargetsHit += LevelComplete;
30         OnAnyReceiverHit += ReceiverHit;
31     }
32     protected override void OnDestroy() {
33         base.OnDestroy();
34         // remove the subscription
35
36         OnAllTargetsHit -= LevelComplete;
37         OnAnyReceiverHit -= ReceiverHit;
38     }
39     private void LevelComplete() {
40         Debug.Log("yay level complete");
41     }
42     private void ReceiverHit(IQueryableLaserReceiver receiver) {
43         // this runs when any receiver in
44         // the scene is hit by a laser beam
45
46         int emittersHittingThisReceiver = GetNumberOfAffectingEmitters(receiver);
47         if (emittersHittingThisReceiver > 1) {
48             Debug.Log("that receiver will burn");
49         }
50     }
51 }
```

*The script above is part of the package and you may find it at the **Assets >Scripts >Laser >Scripting** folder. It is highly recommended to open up **Assets >Scenes >Example1** scene and see this in action.*

That is a mouthful, so let's break down what is happening in the snippet above.

Right on the first line you see that the script is not a subclass of `MonoBehaviour` but of `LaserActorAware`. This `LaserActorAware` is an abstract class that serves as a base with a handful of query methods that can be used to get info on the state of the laser system. Its name, *Aware* implies that it knows about the laser system so it is the base place to start if you wonder what's happening to the laser actors.

`IQueryableLaserTarget` is an interface[5] that denotes *Target*s. The naming seems a bit overly complex but in a nutshell it means *a laser target that you cannot modify but can ask about its state and request notifications when its state changes.*

On line 11 the selected target is initialized using one of `LaserActorAware`'s methods, `FindLaserActorByRootName<T>` that is a generic method[6] for finding `IQueryableLaserActor`s[7].

Then on line 15 we have our first event subscription[8] where we want to get notified when our selected target is hit by an emitter that wasn't shooting it before. See the subscribed method below on line 18 where we implement what to do when the event fires - in this case we just log what happened.

The method on line 25 is called from `Start` as well and it contains two subscriptions - `OnAllTargetsHit` and `OnAnyReceiverHit`. These two do what their name implies - the first one is fired when all *Target*s are hit, while the other fires when any of the *Receiver*s is hit by an emitter.

In the `OnDestroy` method we unsubscribe from the two events above while below that you see what we do when they fire.

◇◇◇

This concludes the very basics of the laser system. For a more in-depth look see the rest of the guide and for the specifics of the scripts see their reference documentation.

[5]Think of interfaces as contracts - they vow to provide a certain functionality but as opposed to classes, we do not know how they implement that.

[6]Generics provide a typesafe way of code reuse by deferring the concrete types until the class is instantiated or the method is called.

[7]It expects the actor's parent object's name and finds the actor component in its children. More on that on the technical docs as well as the reference manual

[8]To simplify it, when the event is fired, invoke the method subscribed. In our case those events are fired in the laser actor scripts.

4 Building your own actors

In some cases it's possible you don't want to use the prefabs as-is but, instead, you only need certain parts of them while you'd replace the rest with your own. As an example, you might need the functionality, the beam and the particle effects but you have your own models and textures ready to use. Let's see what component goes with what for a laser actor to work.

4.1 Laser actor GameObjects and its components

See a hierarchy of the prefab *LaserEmitter* to get a better idea of how the components should look like. We'll discuss the parts after that.

4.1.1 Example: Emitter cube

```
EmitterCube_GameObject
|   + Rigidbody, Collider, LaserActorRoot
 -- SidePanel_GameObject
|   + Mesh
 -- SidePanel_GameObject
|   + Mesh
    ...
|
 -- Emitter_GameObject
    |   + Mesh, LaserEmitter, EmitterLaserDrawer
     -- LaserOrigin_GameObject
```

The hierarchy represents the GameObject hierarchy and the parts denoted with the plus sign represent the components on the objects. So let's see what are these components.

The outermost, or *root* object has a Rigidbody, a Collider and a `LaserActorRoot` script on it. Rigidbody enables physics to affect the object while a Collider is needed so it can interact with the physics system built inside Unity. The custom components is *LaserActorRoot* that is an empty *MonoBehaviour* script. It is used to mark the root parent[9] because it isn't necessary to have the *LaserActor* on the root. **All laser actors need to have a *LaserActorRoot* somewhere on their hierarchy.**

Next up there are a bunch of Meshes, they contribute to the appearance of the object only. Then there is an *Emitter* GameObject with a Mesh, a *LaserEmitter* and an *EmitterLaserDrawer*. The Mesh is self explanatory while the latter two are the heart of the whole object - *LaserEmitter* contains the logic that seeks receivers while *EmitterLaserDrawer* is a *drawer* that hooks onto the emitter's events to display laser beams or particle effects. ***LaserEmitter* and *EmitterLaserDrawer* has to be on the same GameObject - the latter is dependent on the former.** The innermost object is a mere Transform: *LaserOrigin* is the starting point of the laser beam. It can be omitted, in that case the beam will start

[9] Transform.root might not work because parenting similar GameObjects under another one to clear up the scene is a common practice.

out of the center of the GameObject the script is on.

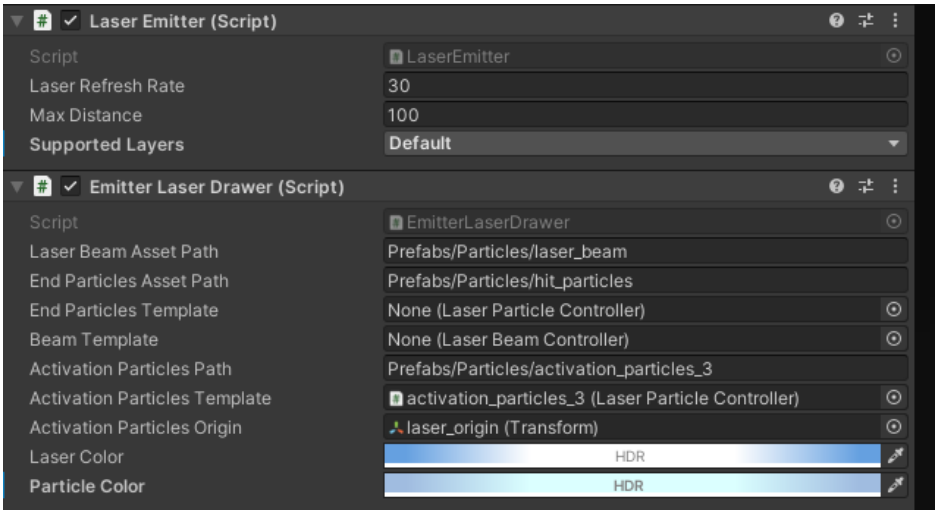


Figure 4: LaserEmitter component

Take a look at the *LaserEmitter* and *EmitterLaserDrawer* scripts. While these will be discussed in detail later on, for a basic setup of the Emitter you need to set its *refresh rate*[10] and its max distance. The *Supported layers* refer to layers that are taken into account when shooting the laser. By default it only allows the default layer. For the Drawer there are *asset path*s and *template*s. The former is a path under **Assets/Resources** for a prefab to be loaded while the latter can be dragged onto the script from the scene.

4.1.2 Example: Mirror

The *Mirror* prefab's setup is a bit less complicated than that of an *Emitter* cube's.

```
Mirror_GameObject
|   + Rigidbody, Collider, LaserActorRoot,
      LaserMirror, MultiTargetLaserDrawer, Mesh
```

In this case all the components are contained by one single GameObject that is the root and that is exactly what *LaserActorRoot* represents. *LaserMirror* contains the logic for the mirror functionality while *MultiTargetLaserDrawer* is another *drawer*, just like before. In this case it's different because multiple emitters can hit a mirror and all should be able to display a beam, that's why it is *multi-target*. Again the important point is that **LaserMirror and MultiTargetLaserDrawer must be on the same GameObject as the latter depends on the former**.

[10] How often per second will the laser update

4.2 Handling missing components from a Laser Actor build

As mentioned before, some components are dependent on others, particularly on *logic* components. When these dependencies are not present then the component in question cannot function and it is due to notify the developer of this in time.

Whenever you see a `MissingLaserAssetException` it means that one of the dependencies needed for the actual component to function was not present where it should have been. Look out for the message for more info or just take a look at the guide above or one of the prefabs to find out what could have been the problem.

5 Technical documentation - Shaders

The project by default uses Universal Render Pipeline. Some of the materials use the built-in shaders but some of them are custom made using Shader Graph. In this section we'll discuss the custom shaders that are used to the assets and the laser beam.

5.1 Laser Beam shader

The laser beam uses a LineRenderer and creates its appearance with a mixture of texture masks and procedural textures.

Find the shader at

Assets > Props > Laser > Shader

Once you have it open in the editor, let's see how it's made first. After that the exposed properties to change its appearance will be self explanatory.

5.1.1 Primary color and fog mask

To the left there is a texture 2D asset - it contains mask textures for the beam that are distributed to the *R,G,B* channels to conserve some space (a mask is a greyscale image that contains values for each pixel. In this case it does not need more space than one channel).

- The R channel contains a simple laser texture that is multiplied by the shader's color later on.
- the G channel contains the center beam only, without the fog
- the B channel contains the fog only, without the beam

In the primary color region the laser's overall color is being set. We take the laser mask, multiply that with the shader's color and then we multiply that with the *LineRenderer*'s vertex color - this way the *LineRenderer*'s material's color can contribute to the beam's color too.

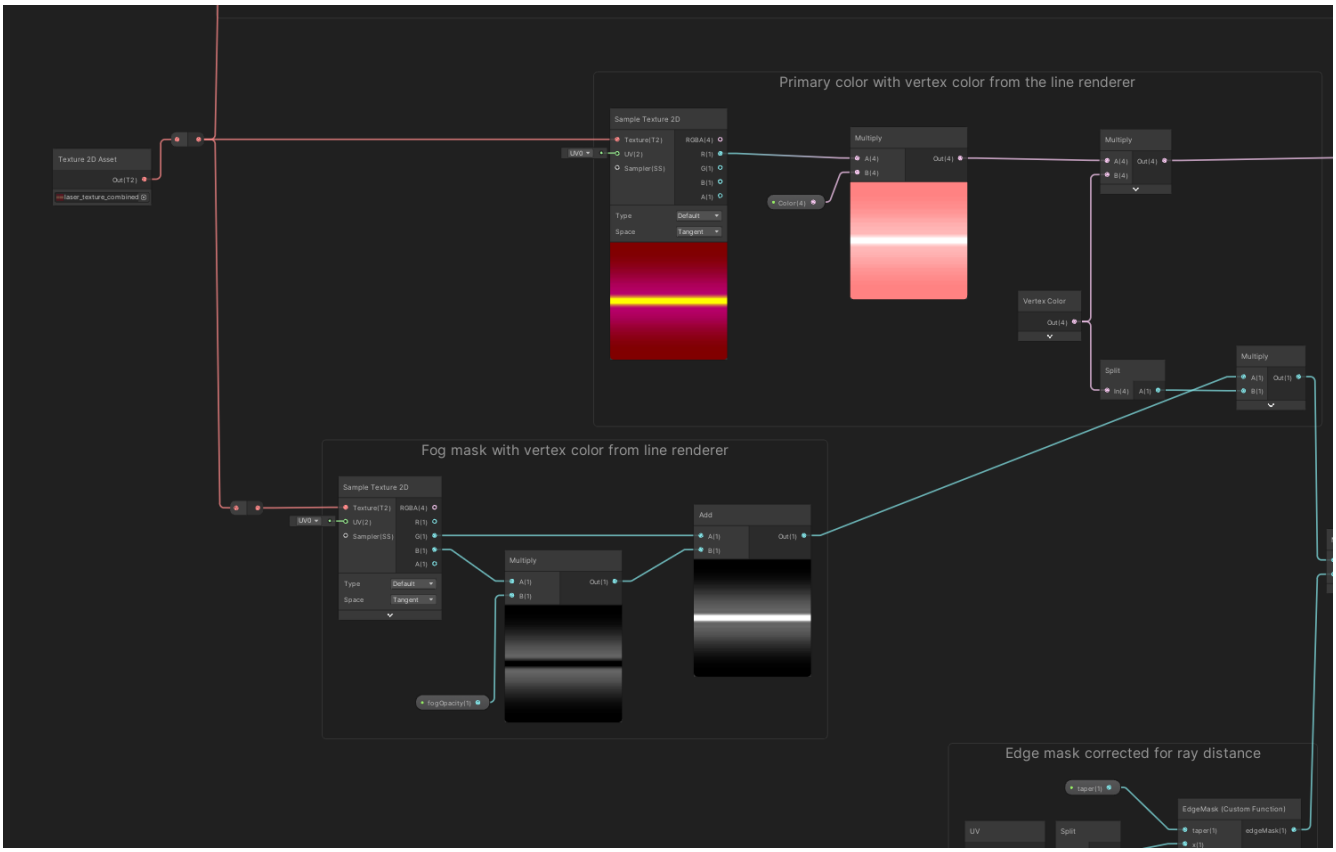


Figure 5: Color and fog settings

Then we set up the fog that mimics the hot air around the laser beam. The fog's mask is multiplied by the *fog opacity* setting to make it more or less visible. Use this to make the beam more believable to your particular scenario - in humid or dusty rooms the beams should have more visible fog around them while in a completely clean environment the fog should be almost invisible. After the multiplication the center beam (and laser itself) is added back in. Later the whole thing is multiplied by the *LineRenderer*'s alpha to adjust its visibility.

5.1.2 Edge mask

Last thing to contribute to the transparency of the beam is the *Edge Mask* that softens out the beam's edges so that they don't just abruptly start and end.
Edge mask is basically a function that linearly increases alpha up to a point (`taper`), then keeps it at 1, then decreases it over *taper* once again. It is best shown as a set of functions.
Mind the values, they are not technically correct - the function's X and Y axes would denote the UV X and the *alpha* multiplier of the mask. As such, both would go from *0 to 1* instead of *0 to 2* and *0 to 4*. On the plot

- **A** represents the *taper*
- **B** represents the beam in length - that is the X coordinate of its UV

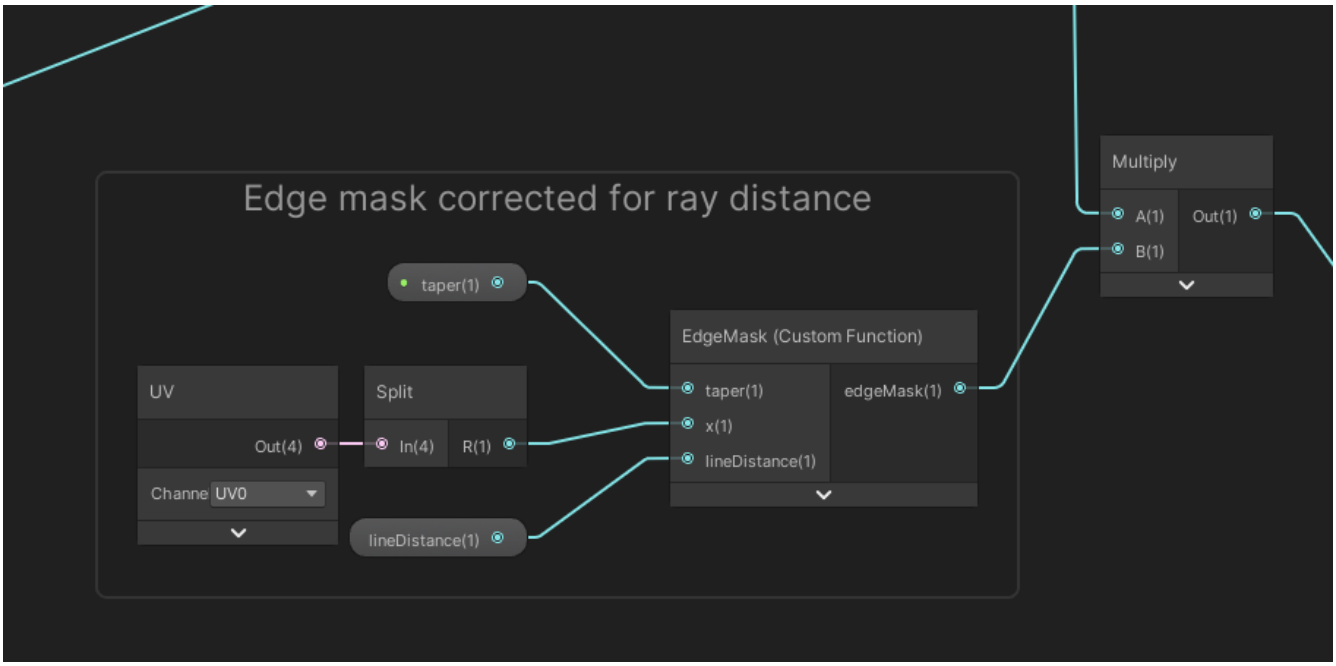


Figure 6: Edge mask

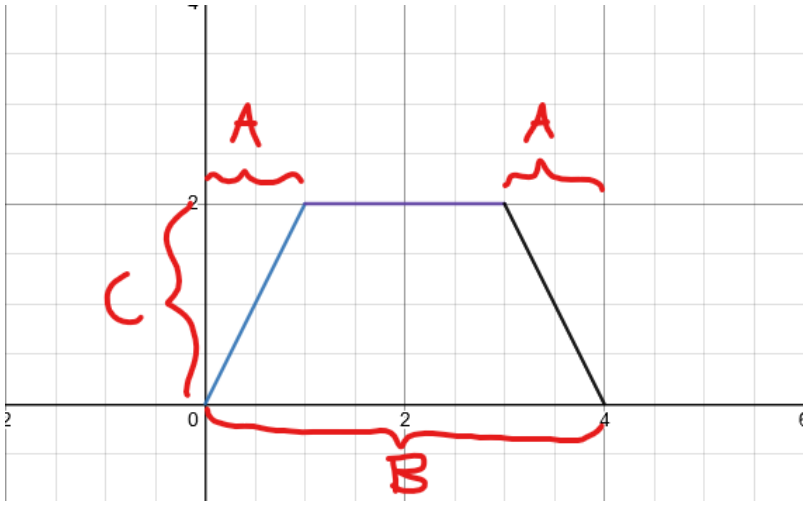


Figure 7: Edge mask visualized

- **C** represents the *alpha* of the mask.

The takeaway from using a procedural mask instead of an image texture is that these laser beams can be really short or really long. Left unadjusted, the beam would slowly fade it over time for a long laser beam, making the ends of it completely invisible while for short beams the mask would be nonexistent.

5.1.3 Fog texture

Next up is the fog texture to make the fog feel more alive.

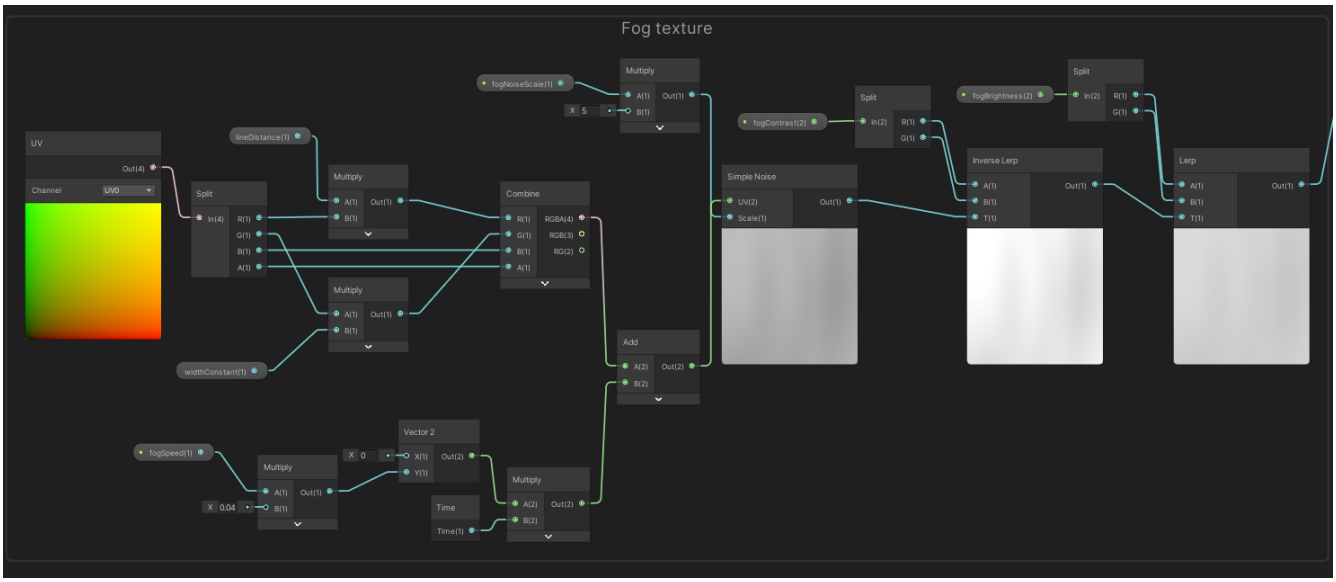


Figure 8: Fog texture

This texture is generated prodecurally using Unity's built-in simple noise. Right at the beginning you see setup of UVs beginning with the UV split, multiplied by some values then put back together. This is done to ensure that the laser beam's fog is not stretched regarless of how long the beam is or how wide the *LineRenderer* is.

`LineDistance` is calculated via code when the laser beam's start and endpoints are uncovered[11]. Using these points we get the beam's distance and we can use that to shrink the stretched UVs so that the texture looks correct on them.

`WidthConstant` is set by *LineRenderer*'s width once and the start and **cannot be modified throughout running game**. This is to avoid unnecessarily setting the width over and over again. As *LineDistance* prevented texture distortion along the X axis, *WidthConstant* prevents distortion along the Y axis.

Below this there's the fog speed settings. `FogSpeed` is another multiplier that you can use to change the speed of the noise texture moving. It is basically Time multiplied by a vector that only takes the Y axis into account so that the fog does not move along the length of the beam.

When the UV setup is done it is used as an input for the noise texture. The fog's `FogNoiseScale` sets the scale of the noise texture which is then ran through a makeshift Levels adjustment - Inverse Lerp sets the shadows and highlights while Lerp sets the midtones. If you're unsure how exactly this works lucily you don't need to know that here - just play around with the values until you get something that suits you.

[11] More on that later when we discuss the logic of the system.

5.1.4 Pulse

Last but not least, there's a slow pulse in the beam that indicates the laser's direction.

$$P_{pixel} = [\sin((UV_{pixel_x} - Time \cdot v_{pulse}) \cdot \mathrm{f}_{pulse} + \frac{\pi}{2}) + off_{pulStr}] \cdot f_{fog} \cdot str_{pulse} \tag{1}$$

It is just a fancy, buffed up Sine wave with a bunch of multipliers to change its appearance. Here's the snippet from shader graph and the explanation below.

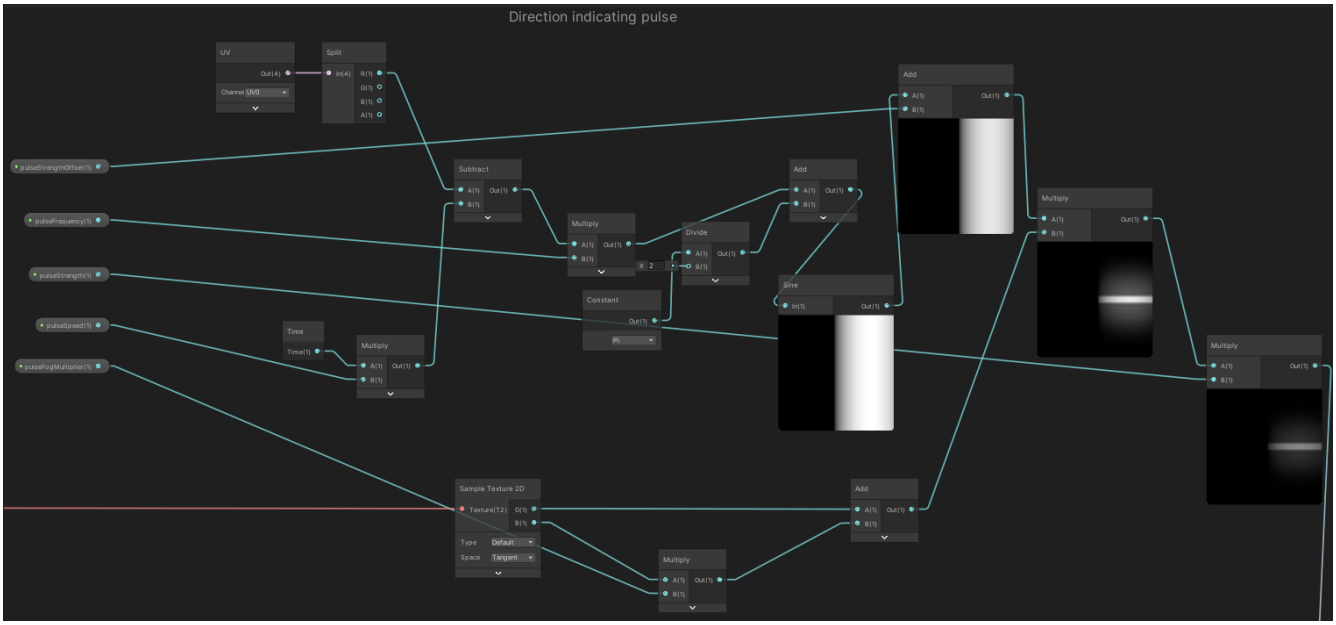


Figure 9: Direction indicating pulse

The pulse is effectively a Sine wave that changes the intensity of the primary color from the beam's start to its end. As before, we use *Time* to make the texture appear moving. This time, to make it go forwards we subtract the time multiplied by a `PulseSpeed` value from the X coordinate of the UV. That is then further multiplied by `PulseFrequency` to get change the wave's frequency and we add $\frac{\pi}{2}$ to make it start at the top which is 1. This is the X input for the Sine function. We then add a `PulseStrengthOffset` to this to lift the curve. We multiply that with a fog and beam textures again (taking fog opacity into account) and there's a final multiplier `PulseStrength` at the end, to control how prominent the effect is.

At the very end this is added on top of the calculated color and there is the final color of the shader.

5.2 Laser Prop shader

The other bit more complex shader apart from the laser beam is the shader for the laser props. This shader uses masks to mix between and primary and a secondary color, a paint, a dirt color and roughness[12] highlights. Using masks instead of simple textures make it easier for you to change the props' color scheme to conform to your game's overall look and feel. The shader uses PBR textures with metalness so if you're unfamiliar with the concept, you may check out the link provided.

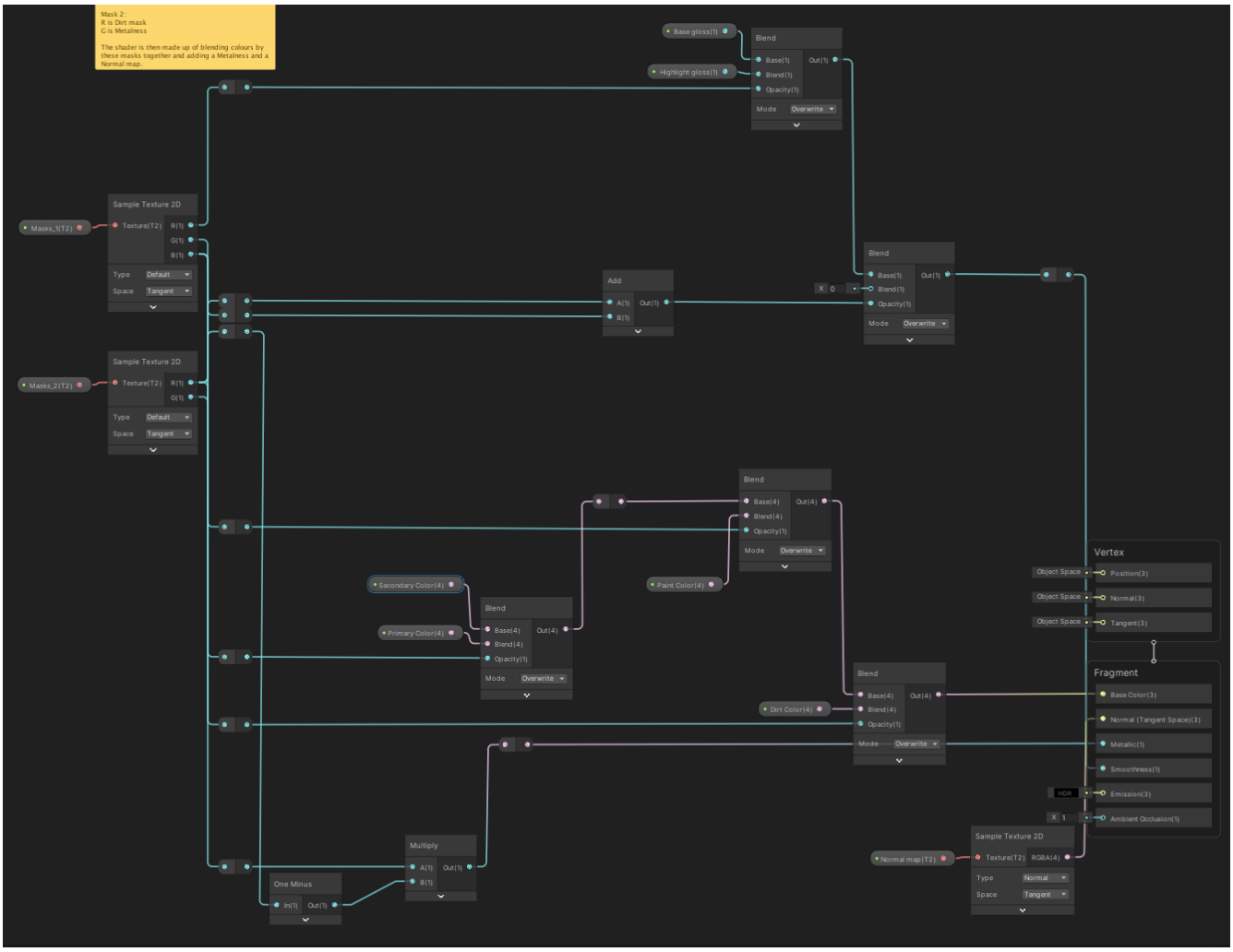


Figure 10: Laser prop shader

There are two image textures that contain masks - each of them have a mask texture in their *R*, *G*, *B* channels. For *Masks 1* texture

- **R** has the roughness mask
- **G** has the color mask (mixing between primary and secondary colors)

[12]You may hear roughness, gloss or smoothness - roughness is the inverse of the other two.

- **B** has the paint mask

and for *Masks 2*

- **R** has the dirt mask
- **G** has the metalness mask
- **B** is empty.

At the top right we blend between a `Base gloss` and a `Highlight gloss` using the *roughness mask* and make sure that the dirty or painted parts of the mesh are completely rough. Use the *Base gloss* to define how reflective the material overall and use the *Highlight gloss* to define how reflective the highlights are.

Next up is the base color of the prop. We blend between the *Primary* and *Secondary* colors then use the result to blend between that and the *Paint* color, then on top of that all we blend the *Dirt* color.

The remaining textures are the metalness (that is inverted, the software I used for texturing exported it the other way Unity exptected) and normal map which is baked from the high resolution mesh.

5.3 Activation particles shader

The last more interesting shader is the activation particle waves that is seen with the *Laser Repeater* and *Nonblocking Receiver* when they are hit.
The idea is to move two noise textures towards each other using Tiling and offset and add them together to create a more interesting pattern. `Pattern Speed` is used as a multiplier for the offset value.

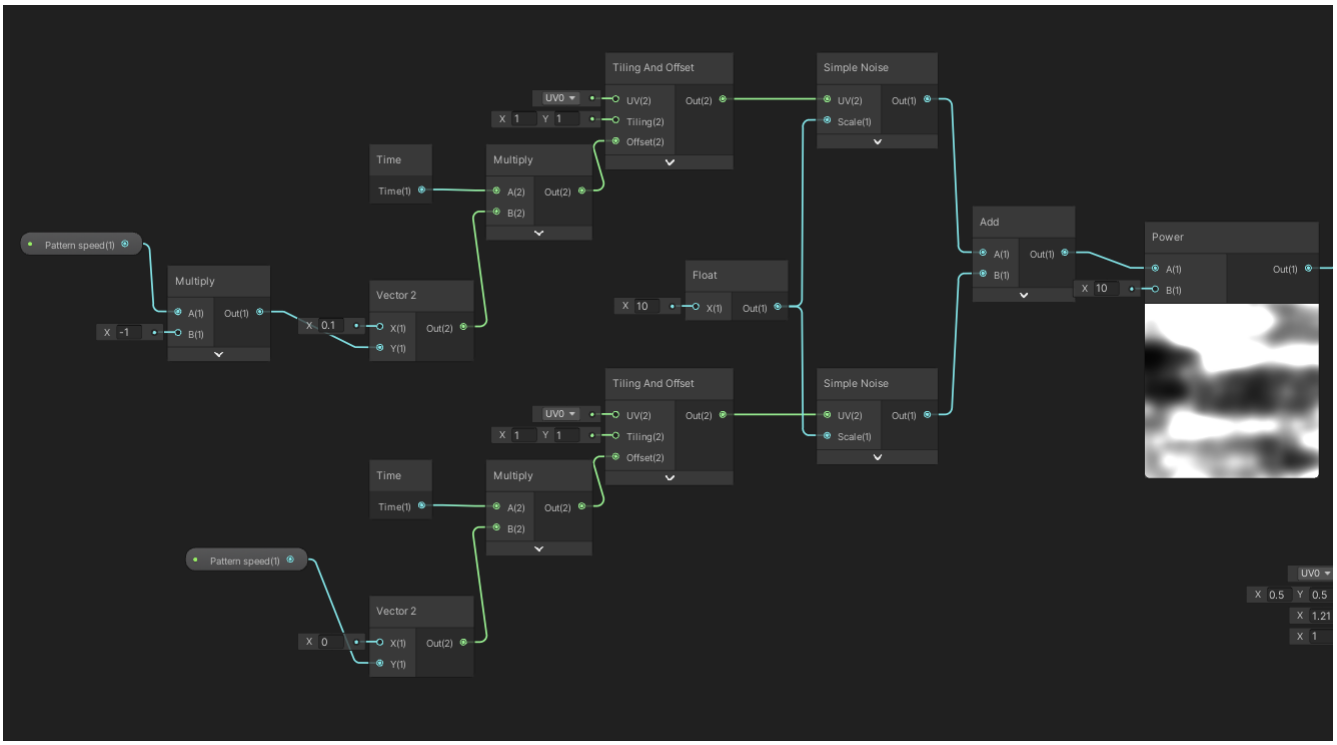


Figure 11: Activation particles - first half

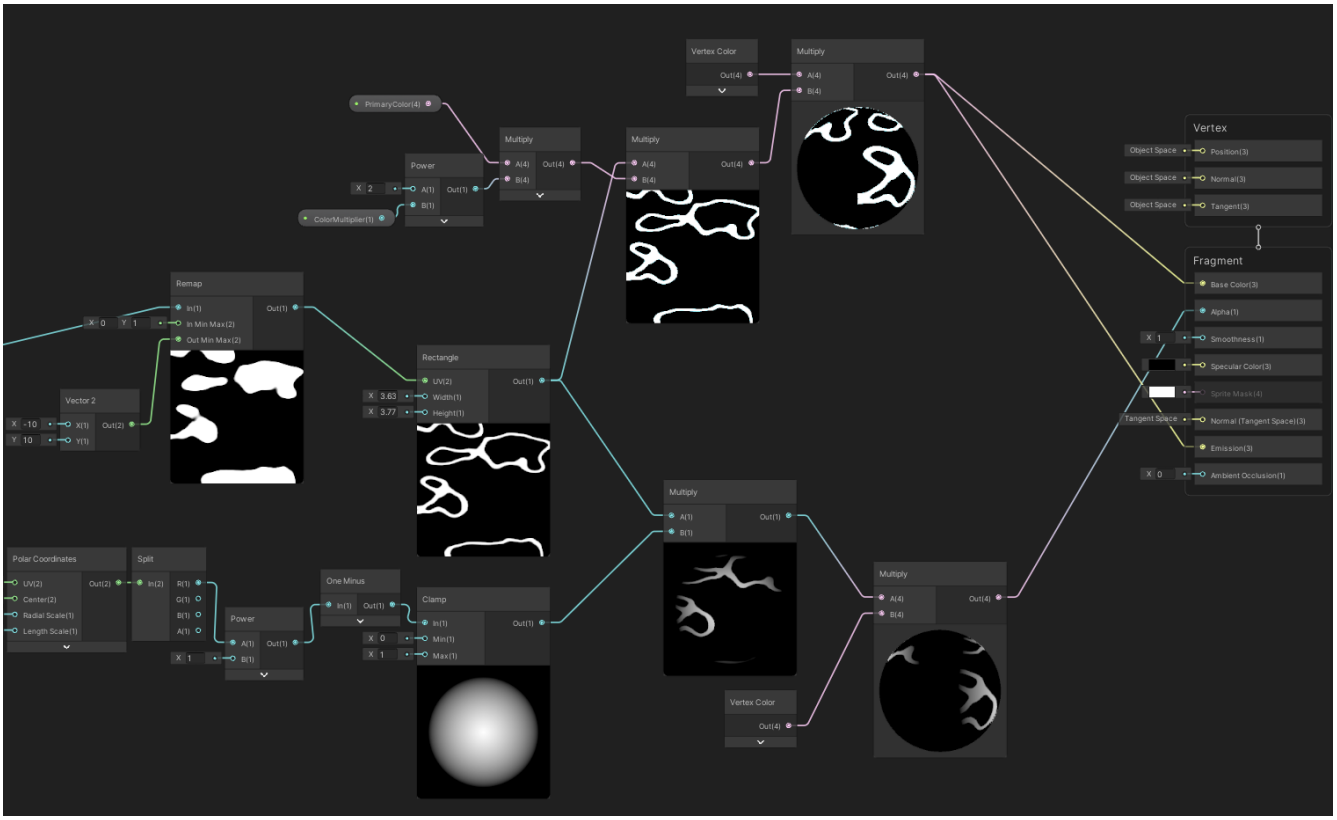


Figure 12: Activation particles - second half

The wavy texture is then remapped to be crisp and only contain black or white and we use that as an input for the Rectangle node to get only a store for the edges. That is the pattern you see in the activation particles. Below that we set up a rounded edge mask using Polar Coordinates so that the particles don't just get cut off abruptly. We multiply with the particle *vertex color* here too so that Color over lifetime can be used. At the top there's just the usual multiplication with the color and vertex color to tint the particles.

5.4 Reference Shader Properties with default settings

Find the short description of the shader properties and their default values below.

5.4.1 Laser Beam shader

Property name	Type	Default value	Description
Color	Color (HDR)	RGBA(32, 90, 191, 255), intensity 3.237	The color of the laser beam
Taper	float	10	Changes the edge fade out falloff distance
Fog opacity	float slider	0.742	Changes the opacity of the fog around the laser beam
Fog contrast	Vector2	(0, 0.5)	Shadows and highlights for the fog levels
Fog brightness	Vector2	(0.29, 0.7)	Midtones for the fog levels
Pulse frequency	float	5	Changes the frequency of the pulse Sine wave - how many waves you see on the beam
Pulse speed	float	0.5	Multiplier for the pulse speed
Pulse strength	float	0.3	Multiplier for the pulse overall strength
Pulse fog multiplier	float	0.76	Controls how visible the pulse is on the fog
Pulse strength offset	float	-0.2	Controls the translation of the pulse Sine wave - pushes the values (highs and lows) in a direction

5.4.2 Laser prop shader

Property name	Type	Default value	Description

Primary color	Color	RGBA(214, 7, 77, 255)	Model base color
Secondary color	Color	RGBA(218, 19, 45, 255)	Model accent color
Dirt color	Color	RGBA(23, 13, 24, 255)	Dirt color on the model
Base gloss	float	0.49	The model's overall smoothness
Highlight gloss	float	0.72	The smoothness of the highlights
Normal map	Texture2D	-	Tangent space normal map for the model
Masks 1	Texture2D	-	Roughness - Color - Paint masks combined
Masks 2	Texture2D	-	Dirt mask - metalness combined

5.4.3 Activation particles shader

Property name	**Type**	**Default value**	**Description**
Pattern speed	float slider	0.8	Movement speed of the wave pattern
Primary color	Color HDR	RGBA(0, 62, 191, 255), intensity 0.805	Pattern color
Color multiplier	float	3	Intensity multiplier for the color

6 Technical documentation - Laser logic

In this section we discuss the implementation of the very core of the asset, the part that drives everything in it: the laser logic. This section covers the high level overview of the system but will also contain the technicalities of small optimisations as well.

How the actors work internally is going to be detailed after we tooka look at the high level overview of the path of the raycast that finds the receivers. If you only need a basic introduction then the next subsection is going to be plenty for you. If you need to know about the intricacies of these actors, check our the subsections later.

6.1 Seeking receivers: the bird's eye view

Let's discuss briefly how the laser travels from the emitter through the relays up to the targets and what notifications are invoked along the way.

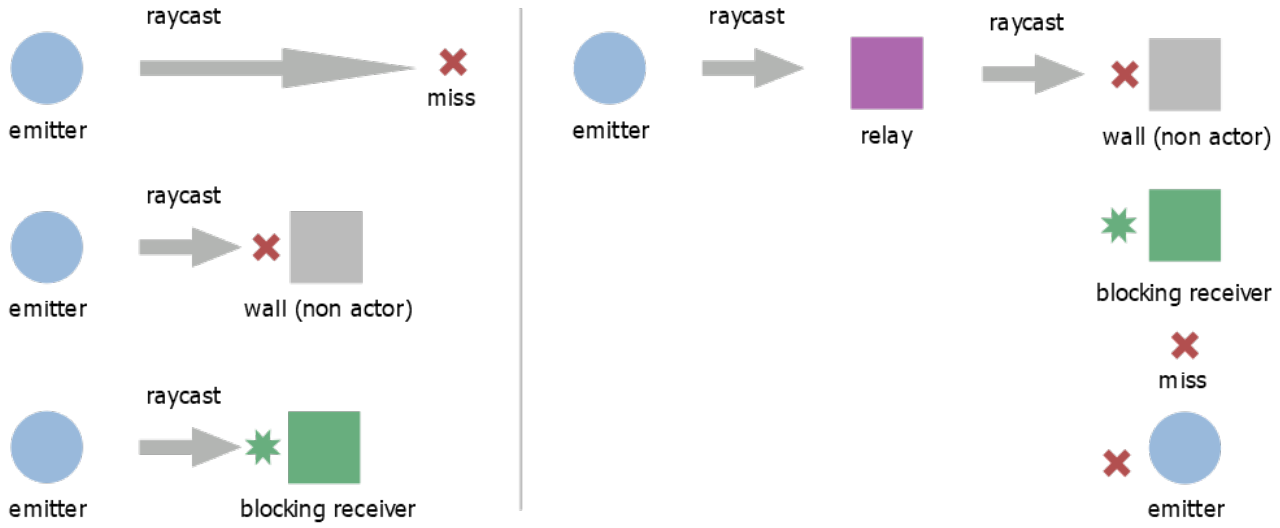


Figure 13: An emitter's raycasts

Upon waking up the emitter invokes an *EmitterActivated* event **that happens during OnEnable, between Awake and Start**. The emitter shoots a raycast to its forward direction up to a predefined distance. It is possible to not have anything in the emitter's path or not have anything in it's path close enough. In either of those cases, Unity's raycast system will not return a hit so the emitter will just invoke the *LaserMiss* event. It does so every *iteration*[13] the emitter fires, not just once.

When the emitter shoots raycasts that hit a *nont-actor* - a GameObject that isn't part of the laser system, a wall for example, it invoked a *LaserHitNonActor* event (again, every iteration) and does not do

[13] didn't write *frame* on purpuse - more on that later

anything else.

Our first scenario with the emitter finding its target is when it hits a *Blocking Receiver*. This time the emitter forwards the hit by finding a laser receiver component on the hit target and invoking its *Hit* method. Since we hit a Blocking Receiver, it fires its *OnNewEmitterReceived* event - **only** when the emitter hasn't hit it before, so this happens once after new contact. Then every iteration it invokes its *HitByLaser* event and returns a result to the emitter. The Blocking Receiver is blocking because it does not forward the laser in any way. Then, after returning from the invocation of the receiver, the emitter fires a *ChainReturned* event to publish its hit results - the naming denotes that it could hit multiple actors while finally returning and it actually publishes all the actors in its hit history. After this all it fires a *LaserHitActor* to notify of the nearest hit.

The interesting part of the system is how a chain of hits are handled - hits that start from an emitter and are forwarded by relays until they hit something or miss completely. The emitter shoots a ray that hits a relay - everything happens exactly the same as before with the Blocking Receiver except that the ray now gets *forwarded*, meaning, not unlike the emitter, the relay invoked the hit receiver's *Hit* method to pass the chain recursively but subtracts the hit distance from its raycast. If it hit a wall or missed its shot it once again acts the same as an emitter. Either way the chain is going to terminate - it could run out of raycast distance, hit a wall, hit a Blocking Receiver or just miss. After termination the result that contains info on the hit receivers is passed back to the emitter.

6.2 Laser Actor internal details

The next few sections are going to contain the internal details of the Laser Actors. Read this to understand the system better if you intend to extend it along the way or you're just interested in how the system works.

6.2.1 Laser Emitter

The emitter's heart and soul is a *Coroutine* - here referred to as *iteration* - that does the following:

- Clear the currently stored affected receivers*
- Shoot the raycast
- If we had receivers we don't affect anymore, notify them*
- Save the affected receiver state*
- *wait for next iteration*

The keen-eyed[14] might have caught that most of the points are marked with an asterisk. For the emitter to decide which receivers it has been affecting so far it has to remember them for at least one iteration and that state has to be managed by a cleanup before shooting rays and saving the collected receivers in the end so we have something to compare to in the next iteration. The emitter can be queried for its

[14] not visually impaired

affected receivers but this might be off a frame due to coroutines' execution order.

To have an idea on what the emitter needs to shoot a raycast, see the points below.

- The raycast's direction - this is the emitter's forward direction.
- The layer masks the raycast takes into account
- The distance of the raycast
- The raycast origin - the laser origin object if present or the emitter's pivot.

This iteration waits for a set amount of time that can be configured in the inspector. The input value is in frames per second. The higher the value the more accurate the laser's beam is going to be but also the more CPU resources it's going to burn.

Modifying this value runtime is not going to take effect.

Emitters also have a maximum distance they could reach - this is needed for the raycast for optimizations. Generally speaking start with a safe low-ish value that's going to reach every actor you need to.

The emitter can be activated and deactivated. Activation happens by default when starting the game. During deactivation it notifies its receivers that they are no longer affected and stops the coroutine.

Laser Emitters are responsible for notifying the receivers they were affecting that they no longer affect them, if for some reason the hit stops. This information is critical for safely deactivating emitters: deactivation logic is implemented in the emitter's OnDisable method. If for some reason this method is omitted then the emitter will not notify any of its receivers and as such the event notifications in those receivers are not going to be invoked.

6.2.2 Laser Relays

Like all actors besides the emitter, relays do not have their internal loops. A relay is a laser actor that can forward an emitter's hit but it doesn't have it own laser beam. It also needs to process being shot by an emitter.

Even though Unity's framework provides utilities for raycasts that can hit multiple targets the system relies on singular raycasts and the recursive thought of relays. Reasons to do so:

- Creating arrays for raycast results would eventually accumulate and more frequent GC stops would degrade the game's performance and as such feel (bear in mind that we could have multiple emitters with high iteration speeds),
- If we use Raycast non alloc to get around it then we'd be putting hard limits to the number of targets in a row or just use more space than needed and simultaneously increase complexity.

All actions implemented inside a relay are going to take effect only when an emitter is shooting a relay and as such activating it. There are however a handful of cases to consider when

implementing these actions.

Preventing a recursive infinite loop

When an emitter hits a relay it first stores the emitter in its collection if it isn't already present then fires off an *OnNewEmitterReceived* event. Now, depending on whether the relay is already part of the chain two things could happen:

If the relay is already part of the chain of the emitter then it's not going to fire any more events and it won't forward the laser beam. It'll just return an empty result as it the emitter has hit a wall. This is a necessity to avoid a `StackOverflow` error if a Repeater and a mirror are facing each other. It does not mean another emitter cannot use the same relay, it just means that it can use only once during an iteration.

If the relay isn't part of the chain yet then the *HitByLaser* event is invoked then the raycast is forwarded. Relays need to provide a direction and origin to their raycasts too. This is actually what makes a custom relay implementation as well - the **raycast origin** and the **raycast direction** and this is what distinguishes the implemented relays. The rest is already implemented in the base class.

Preventing self collision

Certain relays, like the Mirror need to start its raycasts very close to the collider's surface to avoid breaking the illusion of a mirrored laser beam. However, if we'd shoot the raycast exactly from the point on incoming contact it would lead to self collision (shooting itself) due to the inaccuracy of floating point variables. Circumventing the issue, the outgoing ray is offset forward on its projected origin so that it always starts outside of the collider. The relay then checks if after forwarding the raycast the hit receiver is itself and returns an empty hit if so, again, all this to avoid the possibility of a `StackOverflow` error.

Updating the raycast history and remaining distance

Before forwarding the raycast the relay adds itself to the raycast history and subtracts the hit distance it took to hit it from the previous forwarder from the remaining distance. If it runs out of distance then the next hit will not reach anything. Once these and the origin and direction are set, the relay forwards the hit.

Maintaining information on its emitters

Receivers (and as such, relays) keep track of the emitters that are currently affecting them. This collection is updated every time a formerly unknown emitter starts to hit the receiver to include the new emitter, and every time an emitter that stopped hitting the receiver **notifies it to remove itself from the receiver's collection** as noted before alongside the emitter's discussion but repeated here for accentuation.

6.2.3 Nonblocking Laser Receiver

The Nonblocking Receiver stands out in a sense that it shoots its forwarding ray only to create an illusion of the ray passing through it. When a laser hits it, it translates the origin a bit inwards along the incom-

ing direction to prevent self collision by shooting from outside its collider and shoots the ray forwards maintaining its incoming direction. When any of the laser actors move fast or spin you may need to set the refresh rate of the emitter higher to prevent a jagged look on the laser beam.

7 Getting information from the system - the queryable interfaces

Admittedly the `IQueryableLaserActor` naming can be quite misleading, given there are already Queryable interfaces inside C#. Interpret these interfaces of mine as *laser actors who you can ask, or query for information but cannot change*. See, the laser system is intended to be a closed system for modifications - all of those are handled by the system itself - but it is an open system for information access, helping you implement various kinds of logic for your game.

See the hierarchy of the *queryable* interfaces below:

```
IQueryableLaserActor
|
 — IQueryableLaserForwarder
    |
     — IQueryableLaserEmitter
    |
     — IQueryableLaserRelay
|
 — IQueryableLaserReceiver
    |
     — IQueryableLaserRelay
    |
     — IQueryableLaserTarget
```

The hierarchy is quite the same as the actor hierarchy shown before and not without reason. These interfaces contain the events and methods / properties that you could call for information access. Their naming is a good hint as to which actors implement which interfaces.

As an example we'll look at one of them but for the complete picture consult the reference manual.

```
1  public interface IQueryableLaserActor : IEquatable<IQueryableLaserActor>
2  {
3      int GetInstanceID();
4
5      string name { get; }
6
7      string tag { get; }
8
9      Transform FindLaserRoot();
10
11     string ToString();
12 }
13
```

```
14 public interface IQueryableLaserReceiver : IQueryableLaserActor
15 {
16     delegate void LaserEmitterAttached(IQueryableLaserReceiver sender, LaserHit laserHit);
17     event LaserEmitterAttached OnNewEmitterReceived;
18
19     delegate void LaserEmitterDetached(IQueryableLaserReceiver sender, LaserEmitter laserEmitter);
20     event LaserEmitterDetached OnEmitterDetached;
21
22     delegate void NotifyHitByLaser(IQueryableLaserReceiver sender, LaserHit incomingHit);
23     event NotifyHitByLaser HitByLaser;
24
25     delegate void NotifyAllHitsCeased(IQueryableLaserReceiver sender);
26     event NotifyAllHitsCeased AllLaserHitsCeased;
27
28     ISet<LaserEmitter> AttachedEmitters { get; }
29
30     int AttachedEmitterCount { get; }
31 }
```

`IQueryableLaserActor` provides convenience methods / properties so that you do not need to fetch its *MonoBehaviour* to get this information: the instance id, name and tag members are implemented by GameObject already. The remaining is a `ToString` method for ease of debugging and a `FindLaserRoot` method - as mentioned earlier - so that you can move around a compound object without breaking it in case the laser actor component is not on the top object of the hierarchy.

`IQueryableLaserReceiver` knows all this and it provides some events it will invoke when affected by another actor:

- `LaserEmitterAttahced` is invoked once when an emitter hits the receiver for the first time
- `LaserEmitterDetached` is invoked once when an emitter no longer hits this receiver
- `HitByLaser` is invoked every iteration a laser is hitting this emitter
- `AllLaserHitsCeased` is invoked when though they used to, now no emitter is hitting the given receiver

It can also be asked which emitters are attached to it and as a convenience, how many of them are attached.

8 Technical documentation - Drawing

While shooting raycasts is fine and gets us the events invoked to write code against it is also necessary to show our player some cool laser beams and particle effects. Now that the logic of the laser with all of its raycasts is explained the following will make more sense - come natural even.
Drawing has a hard dependency on the laser logic and its events so in a sense, it is a standalone system that you would write on top of the logic and events the laser system provides. As a design choice, the laser logic does not know about the drawing at all. It just publishes its events and anyone who subscribes to them can take advantage of the laser system.

8.1 Beam and particle controllers

The laser system uses the `LaserBeamController` and `LaserParticleController` scripts to refer to and control the particle systems or the beam itself with all its properties that are exposed from the shader. These scripts provide functionality to set the colors or lights, if the particle system has any.

8.2 Drawing in a nutshell

It is crucial to maintain a consistent outlook of the beam of an emitter, no matter how many 'hops' it goes through. As such, even if all the beams are new *LineRenderer*s that go from actor to actor, they need to look like together as if it's a single redirected laser beam.

We can abuse the fact that every single laser beam is always backed by exactly one emitter, no matter how many relays it passes through. This way, we can register a color for an emitter at startup and just query that color later when the relay begins to forward the incoming beam.

Backed by the laser logic's events, drawing the necessary effects is just a matter of subscribing to the system's events and invoking the draw method. There are however a few necessary components to mention when it comes to what to draw exactly. Some of these components are unique to certain laser actors while others are used by all of them.

- The resources path of the laser beam asset
- The resources path of the "end particles" - the particle system that plays when the forwarder hits a wall
- Activation particles resources path - some actors play some particle effects when they are active (emitter) or hit by a laser (e.g. Blocking Laser Receiver)
- Mirror particles path for the laser mirror
- *Templates* for all the above, if you want to drag and drop the particles from the inspector.

8.2.1 Laser Color Registry

Every emitter has a single beam color assigned to it. The beam's shader takes care of the color variations for the same beam. This color can be set at startup or changed using the `UpdateLaserColor` method exposed on the emitter's drawer. Since relays have access to the same registry they can query the color and set it as the beam's color before drawing the beam out.

9 Caching inside the laser system

The system uses caches - storing previously found information for faster access - to prevent redundant calls of the Unity API and as such degrading performance.

9.1 Laser Receiver Cache

Somewhere along the lines it was mentioned that when a raycast hits an object it checks if it is part of the laser system or not so it can decide how to react. This is done by trying to get a component using GetComponent from the Collider that was hit. A *GetComponent* call is expensive, especially if we do that every single iteration for every emitter, and it is outright wasteful when we hit the same object over and over again - so we know exactly that it is a laser actor.

Internally the cache first tries to find the stored laser component inside its storage by the ID of the hit collider. If it finds it then it omits the *GetComponent* call, effectively reducing it to a simple indexing in a Dictionary. If it didn't find the component stored then it needs to make the *GetComponent* call so it can store it for later. Then it returns the component. If however the GameObject did not have a laser component on it then it will store a `null` so that walls won't be queried over and over again either for a laser component.

A direct consequence of this process is that **Once the cache stores an object as a laser actor or a non actor it will NOT adjust even if later on we add a laser component to the object or exactly the opposite - remove the laser component.** As such it is advised to setup the laser components in the editor and leave them as is. If you *really* need to add or remove laser components from objects then you must call the cache's `Purge` method to clear the cache right after.

9.2 Laser Drawer Cache

During the drawing process multiple laser beams need to be drawn and when those beams make contact some particle effects are due as well. We do not know though how many emitters could affect a single receiver simultaneously or one after the other. If we would just instantiate a new GameObject every time it is needed then we'd subject the runtime to very frequent GC stops that would severely impact the feel of the game. To get around it once again we could abuse the fact that only emitters are capable of shooting their own rays and relays just forward it, so, every time a relay hits something there's always an emitter behind it. This way we could assign beams and particle effects to emitters so that we instantiate a resource when the emitter hasn't hit our receiver before and just keep using that whenever the emitter affects the receiver once again. Note that, since emitters could affect multiple receivers, Laser Drawer Caches are unique to receivers as opposed to sharing a single cache.