# RAYCAST_PRO V1.1.3

KIYN_L

# GENERAL:

Dear customer, with a big thanks to you for your purchase my asset and supporting me, I hope that the **RaycastPro** will improving your professional workflow and accuracy of projects for make efficient ray casting. In the introduction below, I'll describe the main function of different parts and key features. Also I'll be happy if see your Review on Asset Store, Any critic, suggestion a lot help me and merciful. 💚

For any question: Kian.tianxuan@gmail.com.

## CONTENTS

## CONCEPT:

The reason why using **RaycastPro** is important that Raycasting problem exists in almost all projects, since I using unity for several years, I didn't see a clean solution to improve physics code quality and performance and it hasn't been provided so far. **RaycastPro** has simplified this process for everyone, beginners to pro, by considering a system that curbs the advanced complexities that I will explain further. Now let's see how default unity raycasting works, Here I write a code for my character to detect a wall on his forward.

```
private void Update()
{
    LayerMask wallLayer = LayerMask.GetMask("Wall");
    RaycastHit hit;
    if (Physics.Raycast(transform.position, transform.forward,
out hit, 2.4f,
            wallLayer, QueryTriggerInteraction.Collide))
    {
        // So, the character will be detecting a wall…
    }
```

Apparently, everything is simple so far. But I want my character to detect Ground, edges, fences and more… but Every situation calls, need an accurate diagnostician while:

- Unity Raycast hasn't any gizmos.
- It's hard to get accurate casting.
- Limited Option for calculating edges and heights.
- Unity Raycast Hasn't **Begin** and **End** Callback.
- Unity Raycast has limitation of efficient methods.
- It hasn't any clear code and don’t satisfy programmers.
- Managing a lot of rays is so hard when project scales.

So, in next step, we will be going to use next ray, we need a capsule cast for detect Ground, but the challenge begins when we understand using capsule isn't simple as basic raycasting:

```
var IsOnGround = false;
RaycastHit ground;
if (Physics.CapsuleCast(transform.position + transform.up,
transform.position - transform.up, 1f,
        -transform.up, out ground, 1.7f, groundLayer))
{
    IsOnGround = true;
    // Is it true?!... is 1.7f my character Height?!..
}
```

To clear this complexity, it make me a challenge to understand how I should cast a Capsule direct to ground without unexcepted bugs, I need a ray to ground just a little bit higher than my character height, so **RaycastPro** ends this conflict with a clean and easy code like below:

```
public RaySensor frontRay;
public RaySensor groundRay;
private void Update()
{
    var IsOnGround = groundRay.Performed;

    if(frontRay.Performed)
    {
        Debug.Log("Press E Key to Open door!");
    }

    if (IsOnGround)
    {
        // locomotion orders
    }
}
```

RCPRO consists of different cores, each of which works in a complementary or independent way. This system was chosen because it increases the extensibility of the tool and allows special features to reach all subsets.

For see how during the plugin, just go in top menu, in ***Tools*** > ***RaycastPro***. In the following panel, you can easily access most of the components and fast setup them to the Scene. Now some advice for:

Beginners: follow the guides bellow, you will learn how to setup your first simple ray and their usage.

Intermediates: Its enough just to get instance and using method inside class with help of Alt tooltips.

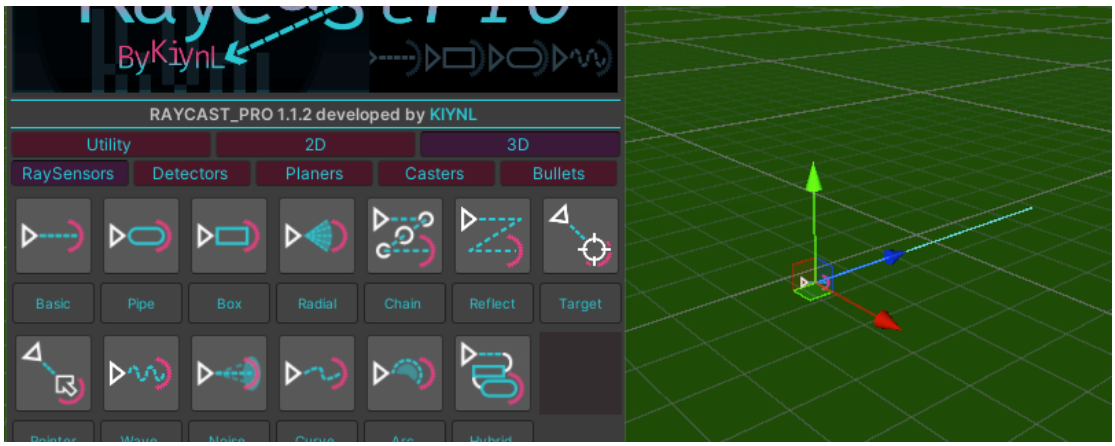
Professionals: be sure that RCPRO is optimized. I'll thank if you submit a review! ❤️‍🔥

## HOW TO SETUP

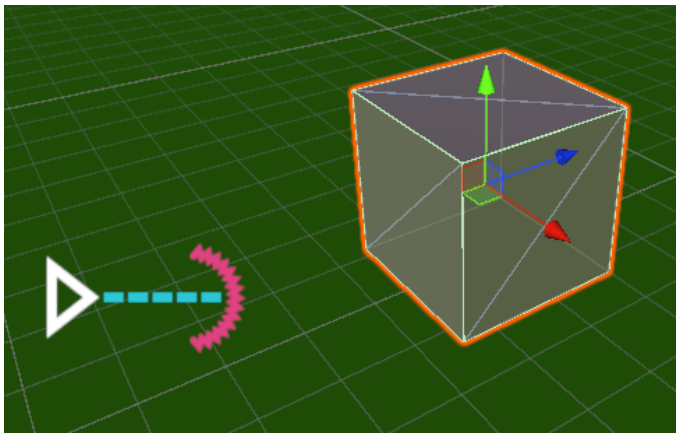
In these quick tutorials you will learn the RCPro nature.

### QUICK START

1. open the top Panel, In **3D** > **RaySensors** > just click **BasicRay**.



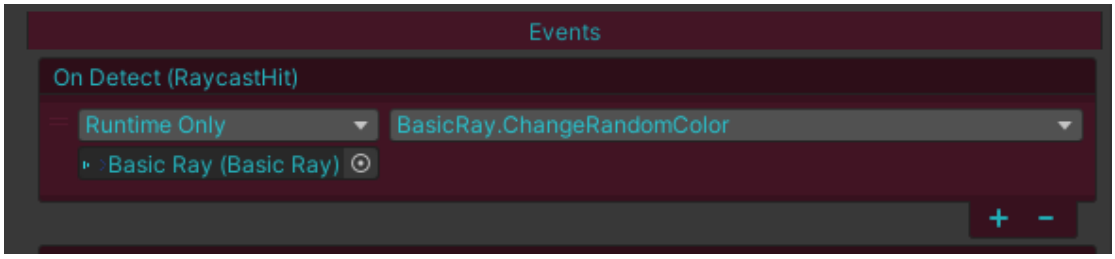

2. Add a cube in scene.



3. now click on basic ray and Open *Events* panel from inspector bottom.

Now Drag ***BasicRay*** component to ***OnDetect*** event field then select ***BasicRay*** > ***ChangeRandomColor***.

***OnDetect event will send a frequency approximately 60 times per second.***

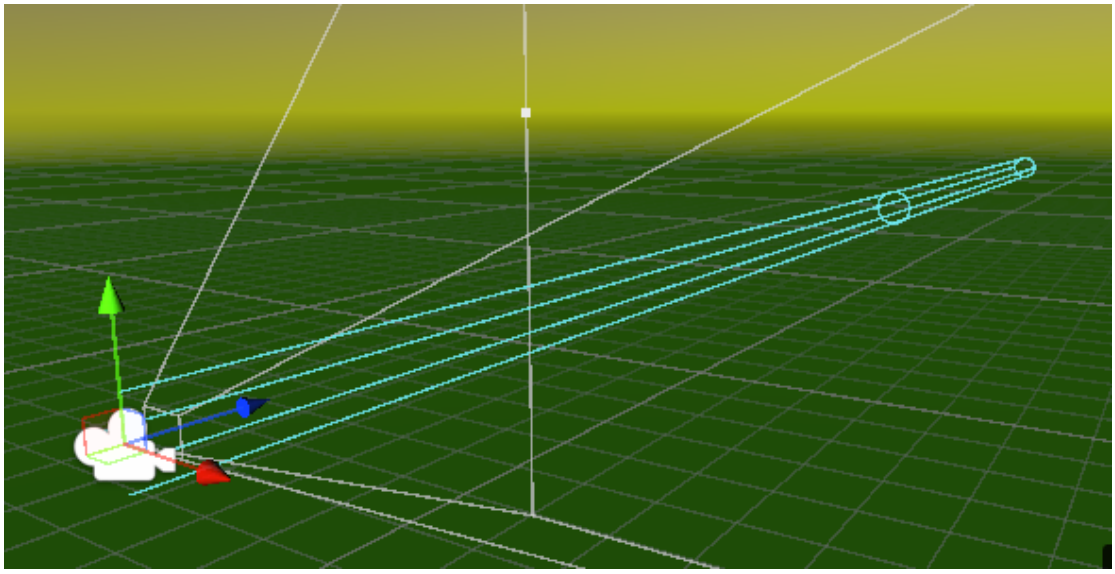



Drag the Ray front of cube in play mode, then cube start flashing colors when is detected!

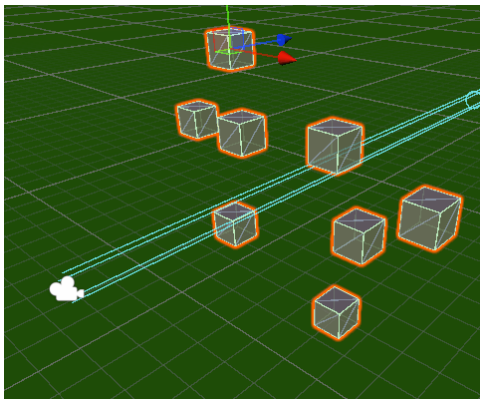
With switch events and methods, we can go further…

## SCRIPTING (DESTROY HOVERING CUBES)

**1.** start with a new scene, add a pointer ray component to camera and set the z length to 40 (it's enough)

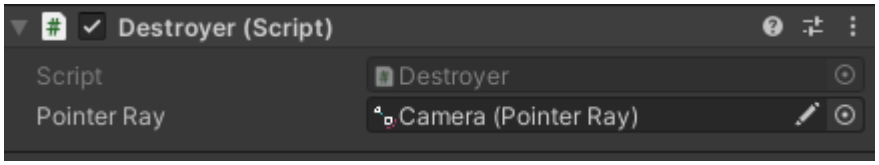


**2.** create some random cubes front of camera view.



**3.** now it's enough to add some little codes to destroy them when we click cubes. Add a script to camera and name its ***Destroyer***. Write the code below and put pointer instance in inspector.

```
public PointerRay pointerRay;
private void Update()
{
    // When left click pressed
    if (Input.GetMouseButtonDown(0))
    {
        pointerRay.DestroyHit();
    }
}
```

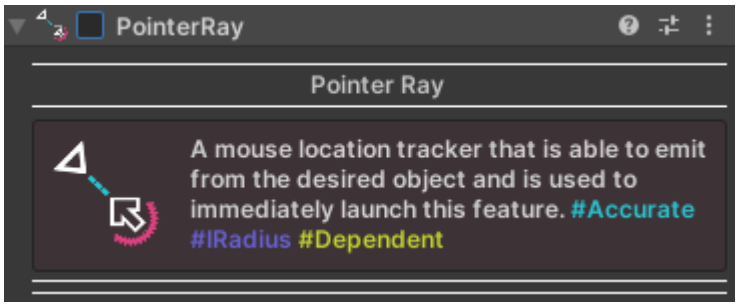



**4. Now click on cubes in play mode to destroy them! (Remember you can add particle and sound effect too, it's example to understand how rays work.)**

## MAKE IT MANEUL

1. Just turn off the component to make it Manuel using.

***Turning off component blocks all mono events so its highly optimized and better than off option in Enum.***



**2.** Go back to the code and make a little change like bellow. A ***Cast()*** script will update the ray and it Hit Object information so the ray will kick out cubes only after clicking.

```
public PointerRay pointerRay;
private void Update()
{
    // When left click pressed
    if (Input.GetMouseButtonDown(0) && pointerRay.Cast())
    {
        pointerRay.DestroyHit();
    }
}
```

## USING EVENTS INSIDE CODE

While using events is a little technically. In following steps, it clarifies how to use it well. First you need to update ray frequently in Manuel casting.

```
private void Update()
{
    pointerRay.Cast(); // Ray update is necessary every frame


    if (Input.GetMouseButtonDown(0))
    {
        pointerRay.DestroyHit();
    }
}
```

2. Inject your method callback in event Listener.

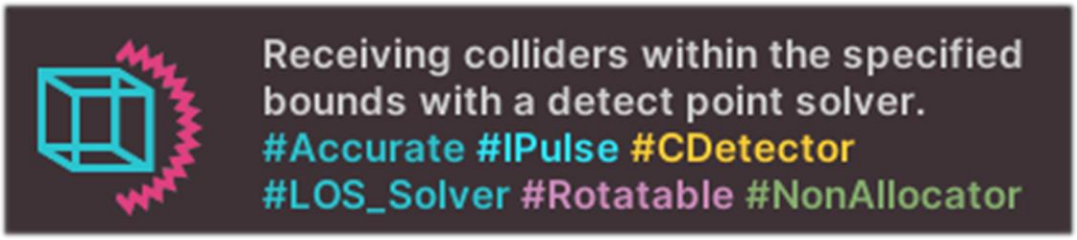
```
private void Start()
{
    pointerRay.onChange.AddListener(ChangeColor);
}

private void ChangeColor (RaycastHit hit)
{
    pointerRay.ChangeRandomColor();
}
```

## DESCRIPTION

After completing the tutorial above, you may be familiar with Ray's functionality. Now, the following explanations are for getting to know the more advanced and extensive features of this plugin.

## TAGS:

Currently, there are about 80 components in this package, whose brief description is provided in the component's header. In addition, the following tags will quickly describe some important component features.




- **#Accurate**: This tag means full accuracy of ray in matching between gizmo and physics calculation.
- **#Directional**: This tag means ray support in directional mode. Some rays without this tag don't need it logically.
- **#PathRay**: These types of rays can reference or calculate a chain of vectors and control casting along the path.
- **#CDetector**: in short of **ColliderDetector**. focus on detecting colliders, save them in a **HashSet** whole of all needed events.
- **#RDetector**: in short of **RaycastHit** Detector, with Save all raycast hits.
- **#LOS_Solver**: almost all detectors support **Line of Sight** features in raycast pro. It will calculate in between colliders to quickly understand, is the view blocked by some obstacles or not?
- **#Recursive**: This tag is for rays that have used recursive functions to determine the path. This means that in certain cases, such as mirror multiplicity, the processing may be infinite.
- **#Virtual**: This tag shows that the component is virtual and cannot be adjusted in the editor.
- **#Dependent**: This tag shows that the component needs other component as reference or suply.

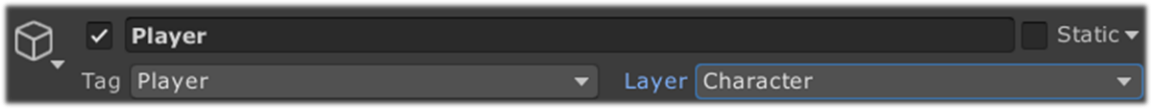
- **#IRadius:** This tag indicates that ray uses IRadius interface, which usually supports radius.
- **#IPulse:** Covers a variable that called **pulse**, which creates a gap between the processing to greatly help optimization.
- **#INonAllocator:** Indicates that the desired Detector supports NonAllocator detection. This reduces the possibility of generating garbage, but because the detectors also perform other filtering, it does not necessarily mean that they become zero.
- **#Scalable:** For components whose detection dimensions change by changing the transform's scale.
- **#Rotatable:** For components whose direction is also changed by changing the transform's rotation.
- **#Preview:** Probably this tool has been recently added and is in the testing phase. The name of the parameters or its mechanism may be changed for better performance and efficiently later.
- **#Experimental:** Experimental tools have more limited functionality and may cause crashes that alert you to be careful while using them.

## GENERAL PARAMETERS:

The following variables exist in most components and are effective for use in all tools.

### DETECT LAYER:

The detection layer is the most important parameter of all the components with which you can identify and separate the detectable colliders. Specifying layers is accessible in the header of each Game Object.




In **Detectors, also Tag filter** has been added for more control.


## INFLUENCE:

It's a handy parameter for ease of coding. You can quickly set a get this parameter by referencing any Core in script, also some

***For example: HP -= influence * damage***

## TRIGGER ITERACTION:

Specifies how ray collides with ***IsTrigger*** colliders. ***Use Global*** means to use project settings and ***Collide*** means to accept ***IsTrigger*** for hitting.

## AUTO UPDATE:

Automatic processing mode between **Fixed**, **Normal** and **Late**. In this case, the gizmos may not be updated quickly in **PlayTime**, because the **casting** isn't always performed, but it works completely reliably in the **Editor**, and you can test them perfectly real-time.

***By Turning Off the Component it will going to Manuel Option. It's most Optimized option by blocking all mono behavior updates in code.***

## GIZMOS:

This option specifies how the gizmo should be updated. The **Select** option works only when object is selected, and **Fix** will permanently active. It is recommended to set it to **Off** when you are sure It works to increase scene performance.

*Use Auto mode for show gizmos when detecting and selecting automatically. It's quick and clean in play mode by disappear casts.*

## EVENTS:

Events help you to have detection and non-detection events and to be able to control scripts through it. Keep in mind that some functions are placed in the body of the component itself for ease of coding.


## RAYSENSORS:

In general, RaySensors return an output of type Raycast Hit. They can cover the ray path by a ***LineRenderer*** and leave a ***stamp*** at the hit point.

**Performed**: It's a property and simply using for check auto updates rays' detection.

**Cast()**: It's a method for update the rays (Or any Core) info and also return true when detected.

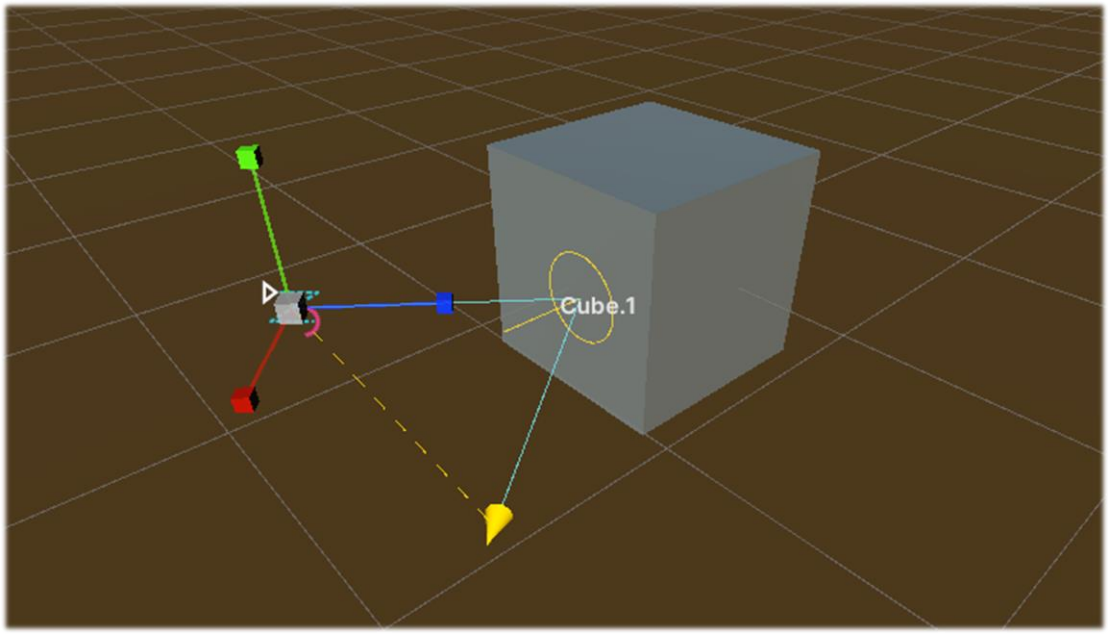
## LINER:

Liner is an abbreviation of **LineRenderer** in unity, It's a common built-in component for showing a 2D line based on position vector array. All Rays of **RCPRO** can control or more LineRenderer on their in great quality and

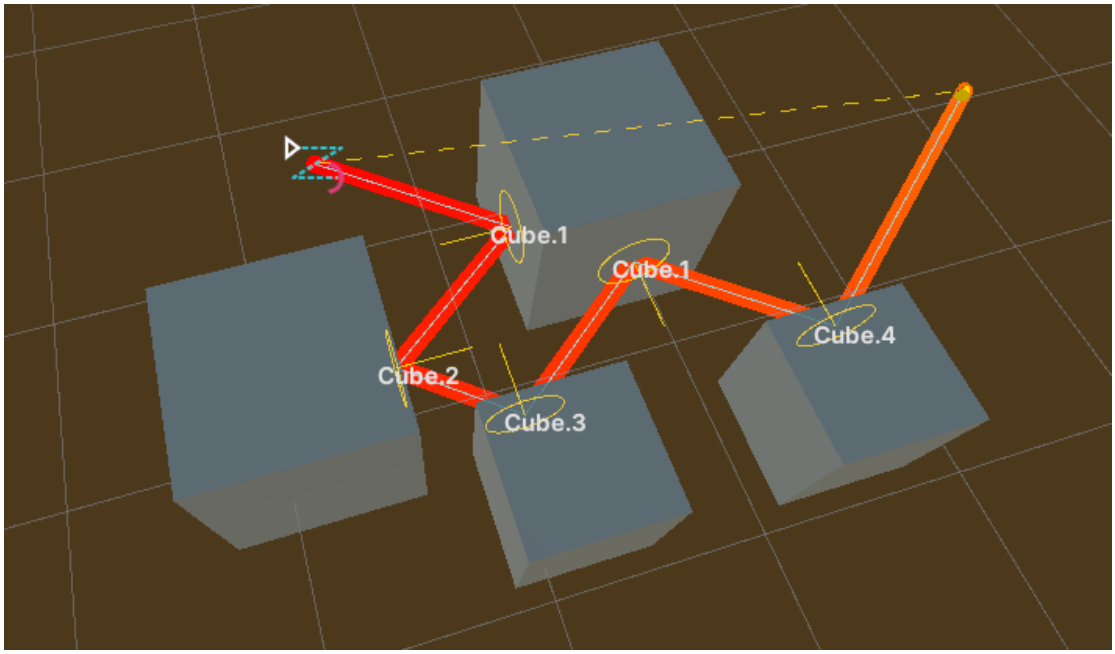
accuracy. Now you can just simply using ***Add*** button from liner to test it or following tutorial bellow.

1. Add a **ReflectRay** from panel.

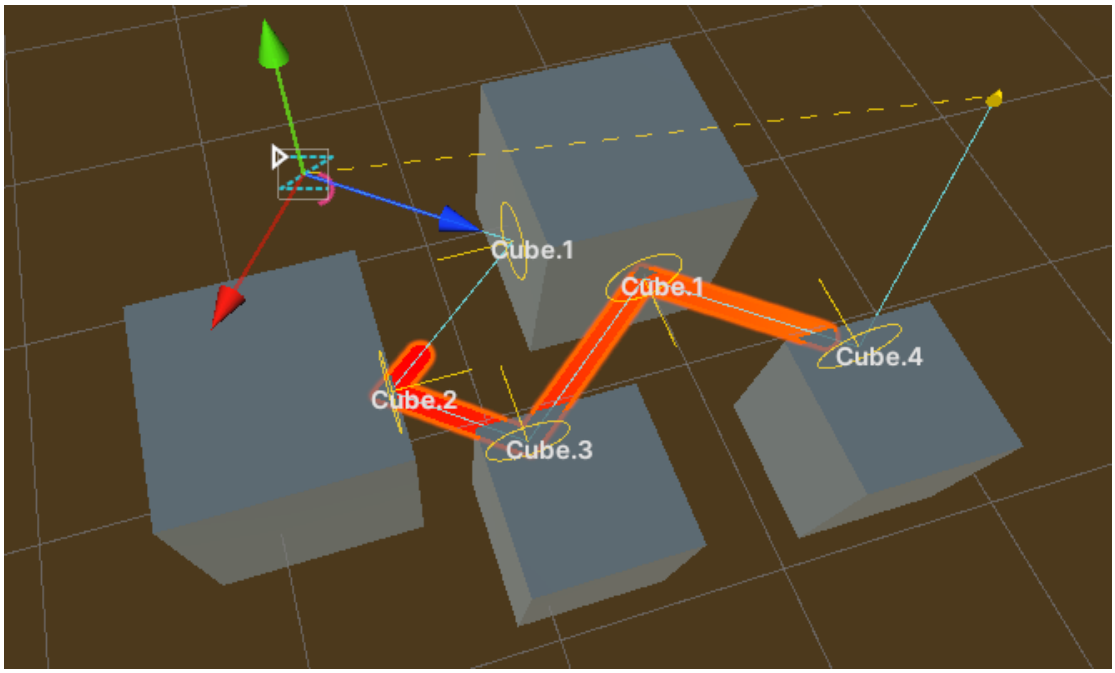
2. Bend the angle a bit, increase length by adjust Z, so that it touches a Cube, then change the **Reflect layer** to Default. **Detect Layer** to Nothing so that the Ray only reflects itself and you will see something like below.




3. By placing the Gizmo in **Fix** mode, it don't disappear more when deselected. Now going on cube, and duplicate some more.




5. Add a line and test it's Option.



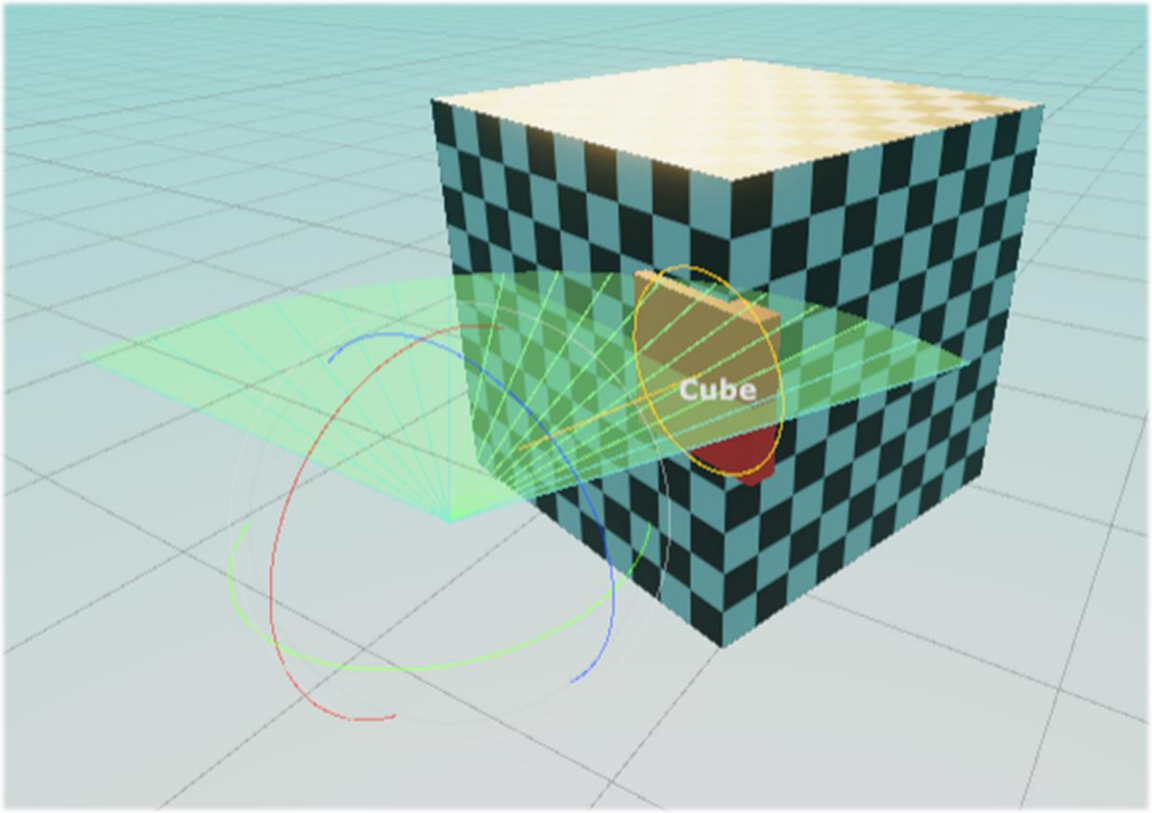


*Cut on Hit: It will be cut off when hitting the collider.*

*Also If you need to have more liners. Just select RayLiner from the Utility panel and add ray to it.*


## STAMP:

In the stamp settings, ***Stamp on Hit*** moves its location to the Hit Point when it hits, and it is placed on the Tip otherwise. At the bottom of the ***Auto Hide*** option, it will display the stamp only when encountered. Other options are related to specifying offset and direction.



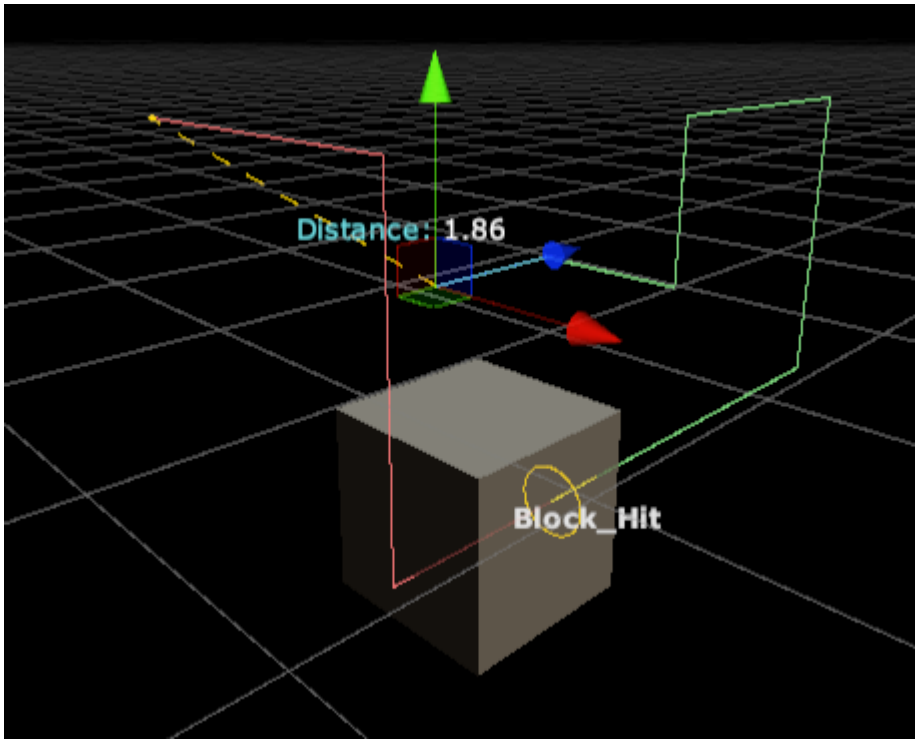

*If your stamp layer is the same as the ray Detect Layer, you may see a crazy behavior in scene, this is because the Ray tries to target the stamp itself. Make*

*sure that the stamp layer is not the same as the Detect Layer or has not a collider.*

## PATH RAY:

Basically, this category of Rays includes a list of points called **Path Points** and **Detect Index**. These points are calculated in different Path Rays and their system is like this, they send direct rays from one point to the next point in a chain, and the first detection point interrupts the process.

In the example below, you can see a **ChainRay:**



## PATH RAYS LIST:

**1.** **ChainRay:** This Ray normally receives the points and releases the Ray in its path. In Transform mode you will be able to easily animate points.

**2.** **ReflectRay:** This Ray, like other Rays, can process collisions in the Reflection Editor itself, such as the impact of a billiard ball, and the collision points will change dynamically. The important part is to place the Reflect layer.

**3.** **WaveRay:** This Ray is an algorithmic, mathematical processor that can help you in randomization or accurate processing of sinusoidal functions, although it is natural that it consumes more performance than other Rays.

**4.** **ArcRay:** This Ray is a Velocity-based processor that includes local and global states both in the initial and general angle, so that you can use it in Trajectory calculation.

**5.** **CurveRay:** This Ray consists of several Animation Curves and segments that help to create the sensor in a curve-shaped manner or creative gun shots.

**6.** **HybricRay:** And finally, if you need to combine Rays to create a nested expression, Hybrid Ray solves this problem by stacking other Rays together and processing them in one shot.

**PathCast:** This option processes all Rays one after one as PathCast. This option will show a yellow dotted gizmo around the ray and the ray will be cast seamlessly; your hybrid hit point will be displayed with an offset to its source ray.

*You can disable PathCast when you only need their Liner or vector path to avoid unnecessary processing.*

## PROPERTIES

**Direction:** Automatically selects the exact direction in either World or Local mode.

**LocalDirection:** Returns pure Local direction without the effect of Normalize.

**TipDirection:** Returns the subtraction of Tip from Base as the direction.

**HitDirection:** Returns the direction of the last ray break on the surface as the direction.

**Tip:** Returns the position of the Ray's tip.

**TipTarget:** Automatically returns the location of the last hit point of the Ray and replaces the Tip point if there is no detection.

**HitDistance:** Returns the exact distance of the ray to the hit point.

**ContinuesDistance:** Considers and returns the remaining distance of Ray from the hit point.

**DirectionLength:** Returns the size of Direction itself in pure form.

**RayLength:** This option calculates and returns the total length of the Ray.

# DETECTORS:

Detectors have various applications, whose working method is based on sending multiple Raycasts and receiving colliders. In the **RCPRO**, the detector tools use their high potential to accelerate and accurately perform detection.

## COLLIDER DETECTORS:

The most common and probably the first type you will come across are **Collider Detectors**, they simply detect Colliders in their area and pass through their filters. In the meantime, after setting the parameters that you explain, you only need to access the anthologies by getting the Detected Colliders member.

Radius: The Radius parameter in the Range Detector along with the Height are the dimension setting indicators, and you can visually and live determine the volume of the area whose colliders need to be detected.

TagFilter: As the name suggests, in addition to the Detect Layer, the Tag filter has also been added to the Collider Detectors.

NonAllocator: This tick helps to avoid updating the dimensions of List, With the active option, You will need to reserve enough memory in the detection array and the selection of colliders will be limited to the incoming array, this will prevent the creation of Garbage, Although the Raycast Pro plugin has extensive settings and filtering, and the overall goal is to make the tools more efficient than just optimization.

## HOW TO USE COLLIDER DETECTOR

Detectors are very easy to use, and you have access to Detected Colliders as direct or in events. They store all selected Colliders in a **DetectedCollider** as **HashList**.

However, you can call them through events or getting the **DetectedCollider** members.

```
private RangeDetector _rangeDetector;

private void ExplodeEnemies()
{
    _rangeDetector.Cast();

    foreach (var member in _rangeDetector.DetectedColliders)
    {
        if (member.TryGetComponent(out Cube cube))
        {

cube.TakeDamage(_rangeDetector.GetInfluncedMemberDistance(member) * 20f);
        }
    }
}
```

## SMART LOS (LINE OF SIGHT) SOLVER:

**RCPRO** support an easy, handy and very quick LOS system inside the collider detectors. You can switch between different taps to discover how it word or I'll describe their usage, and reason in bellow.


**Block Layer**: A layer intended for blocking objects like the ***Wall***. The object behind this layer will be detected

first but the detector will be filter and ignore them into **DetectedColliders** HashSet.

**Bounds Solver:** This option is considered for optimization and limits the detection point in the bounds of a cube.

**Collect LOS:** Be sure to enable this option if you need to get the Hit Info of the LOS block hit point. They are stored in a dictionary with the key to the main blocked Collider and you can easily call them as below.

```
_rangeDetector.Cast();

foreach (var member in _rangeDetector.DetectedColliders)
{
    var blockHit = _rangeDetector.DetectedLOSHits[member];
    Instantiate(ball, blockHit.point, Quaternion.identity);
}
```

**Check Line Of Sight:** By activating this option, the blocking line will be checked.

---

### IGNORE:

It a naturally physics overlaps, so no determining **detectionPoint** or **LOS**.

***Use it for giving best performance but less option.***

---

### PIVOT:

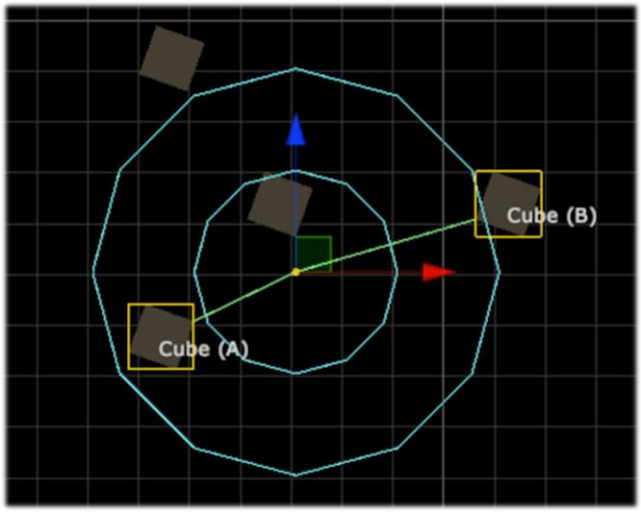
The LOS right at the pivot of the object.

***If your objects are not very complicated, this option is suitable.***

---

### NEAREST (SENSITIVE TO EDGES):

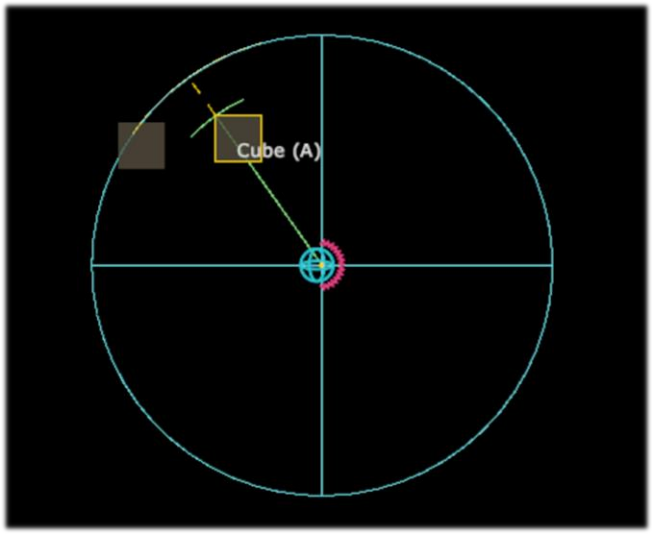
The detection point is the closest to the center of the detector.




*USAGE: Use this option when you want high detection sensitivity, for example, when the corner of the object is within the range it will be detected fast. Note that the IGNORE option works very similarly, so forget about it if you don't need the other LOS options.*

## FURTHEST (ALL INSIDE):

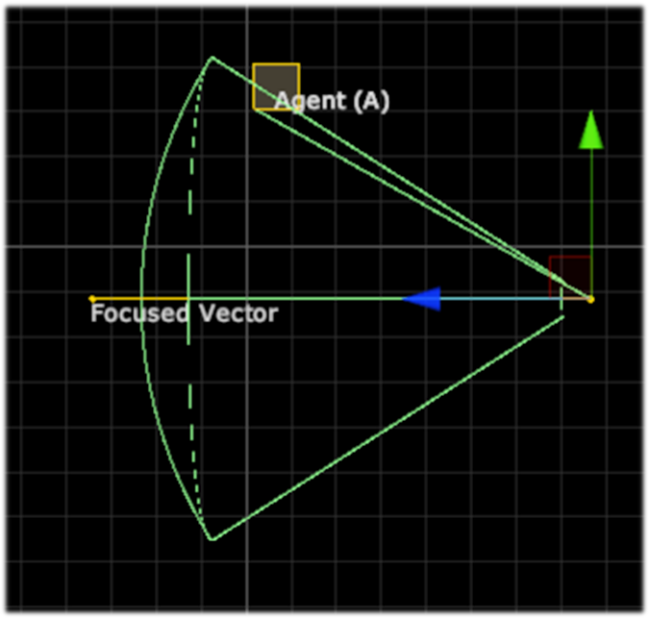
Places the detection point at the farthest place from the center of the detector.



*Use This option when you need the colliders totally inside the room.*

## FOCUSED:

As seen in the image below, the detector tries to find the closest point to the focus vector.




*This Option is highest suggested for SightDetector. because your Agent Eye can easily detect collider edges based on it's pupil nature.*

## DODGE:

The most complex and smart one is this, the algorithm tries to find a way to avoid blocking by check 6 sides of object, it gives more intelligent for the agent but also more performance needed.

*It is recommended to activate the bounds solver when using it.*

## SYNC DETECTION (CUSTOM COMMPONENT FINDING)

For get a specific component in the update function, **GetComponent** is very heavy method, so the solution is The **SyncDetection**. It only trigger when a new collider joins the **HashSet** and sync the result via dedicated list. Follow the guide below.

1. Suppose Agent date is my agent stats here. I want to find to AgentData component carriers in most optimized way here.

```
public class AgentData : MonoBehaviour
{
    public float hp = 100f;
}
```

2. Inside my detector script I'll sync my AgentData list in start first. Two **OnNew**, **OnLost** methods like bellow help me to coding on events!

```
private RangeDetector _rangeDetector;
public List<AgentData> agentData;

private void Start()
{
    _rangeDetector.SyncDetection(agentData, OnNewAgent, OnLostAgent);
}
public void OnNewAgent(AgentData data)
{
    Debug.Log("Found Agent!");
}
public void OnLostAgent(AgentData data)
{
    Debug.Log("Lost Agent!");
}
```

# PLANERS:

planar is a tool to guide the direction of the RaySensor and in this version it is offered as an **experimental version.**

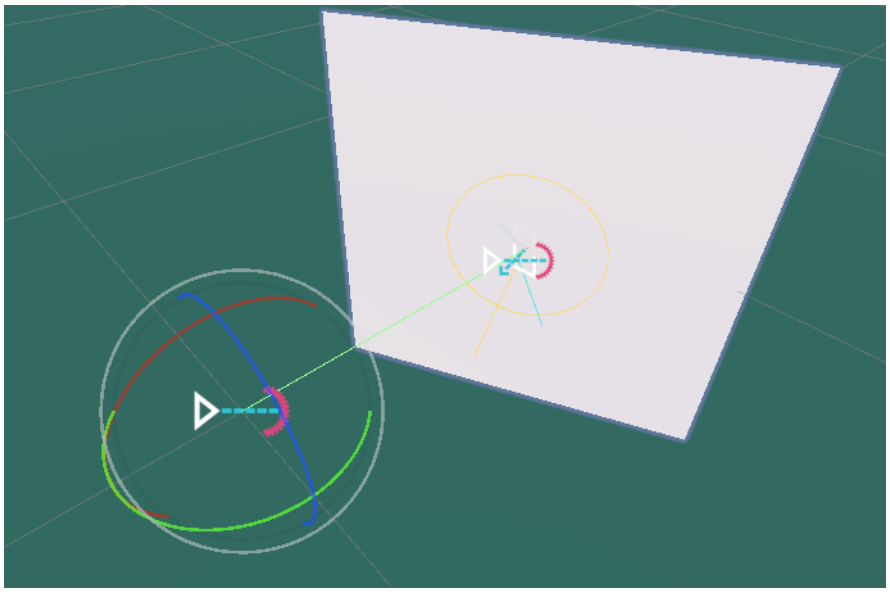
## HOW TO SETUP PLANAR:

1. Create a new scene and a Basic Ray from the Raycast Pro panel.

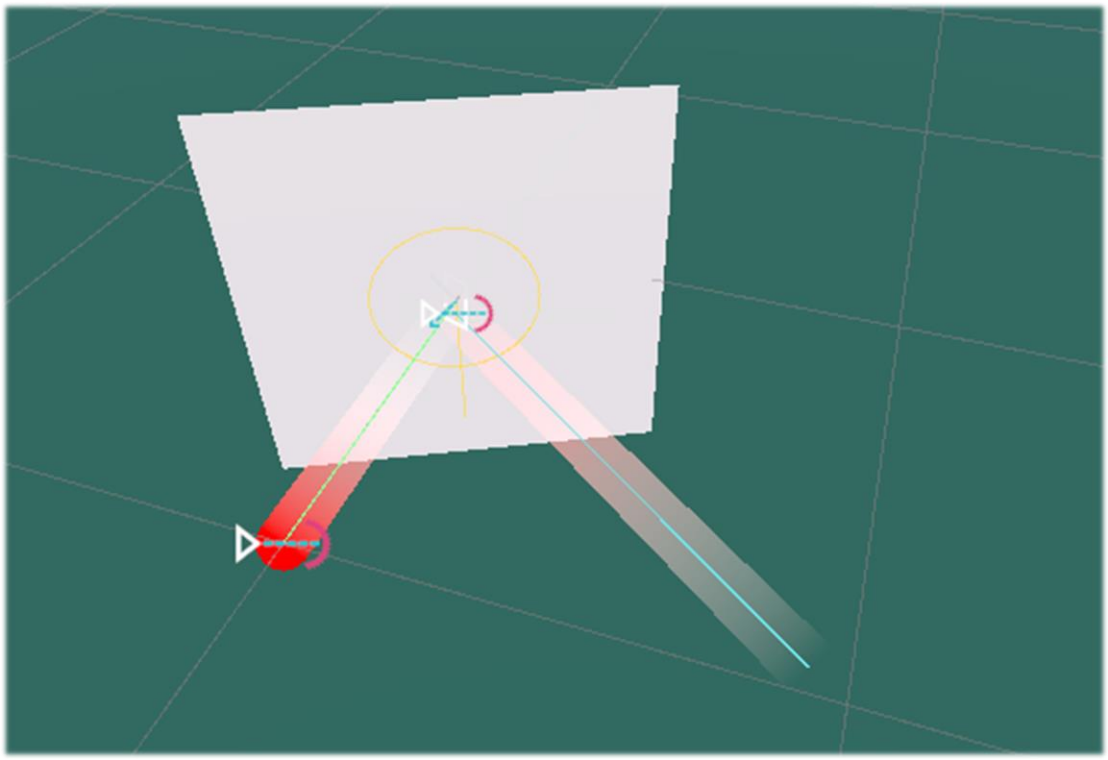

Make sure you tick Planar Sensitive, the Any option means accepting all planers. Otherwise, you have to select planar manually, which of course has better performance.

2. Now select a Reflect Planar from the panel and place it in front of the Ray as shown below. Then turn it slightly until a reflected ray has been seen.

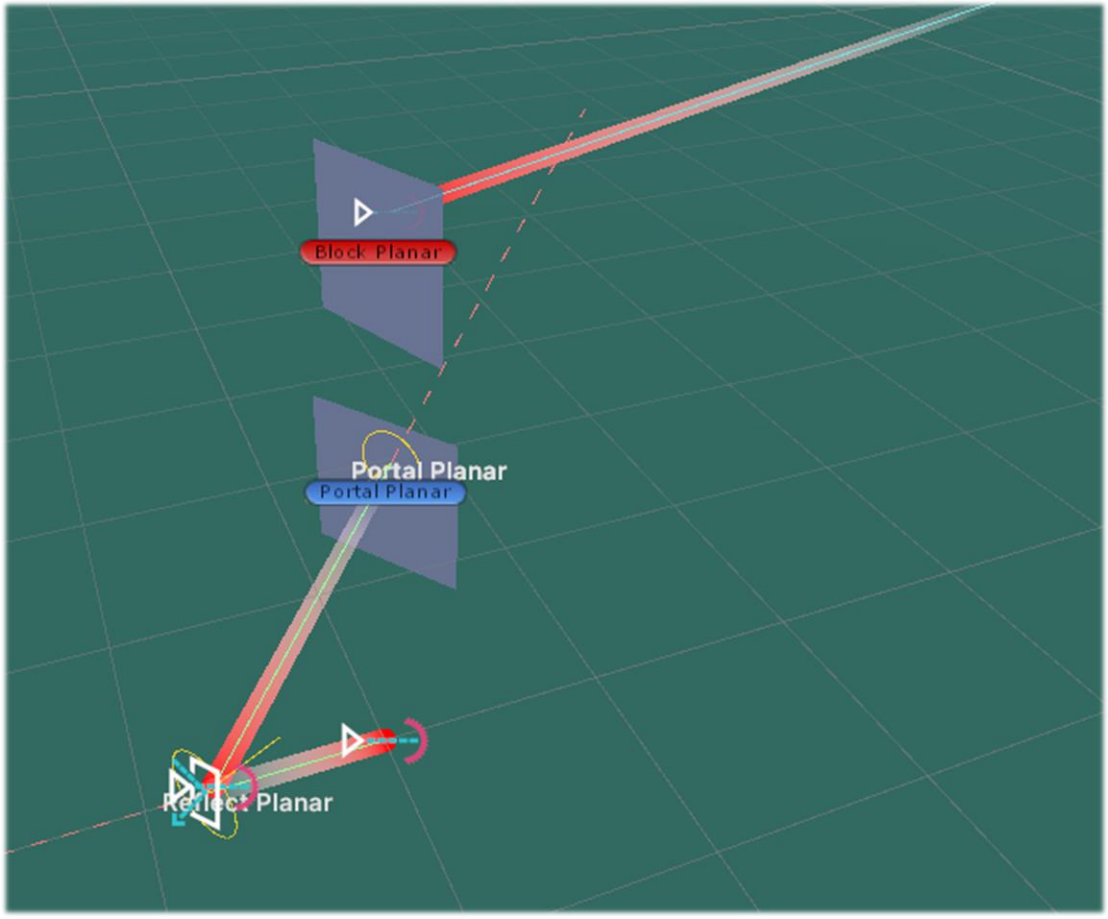
Hint: planers cannot work in Real-time in the Editor due to the creation of clones, but by pressing the play button, the process of creating clones starts.



3. Stop again and play after setting a **Liner** to Cut On Hit mode. The clone automatically tries to copy the parameters of the Ray behind it.

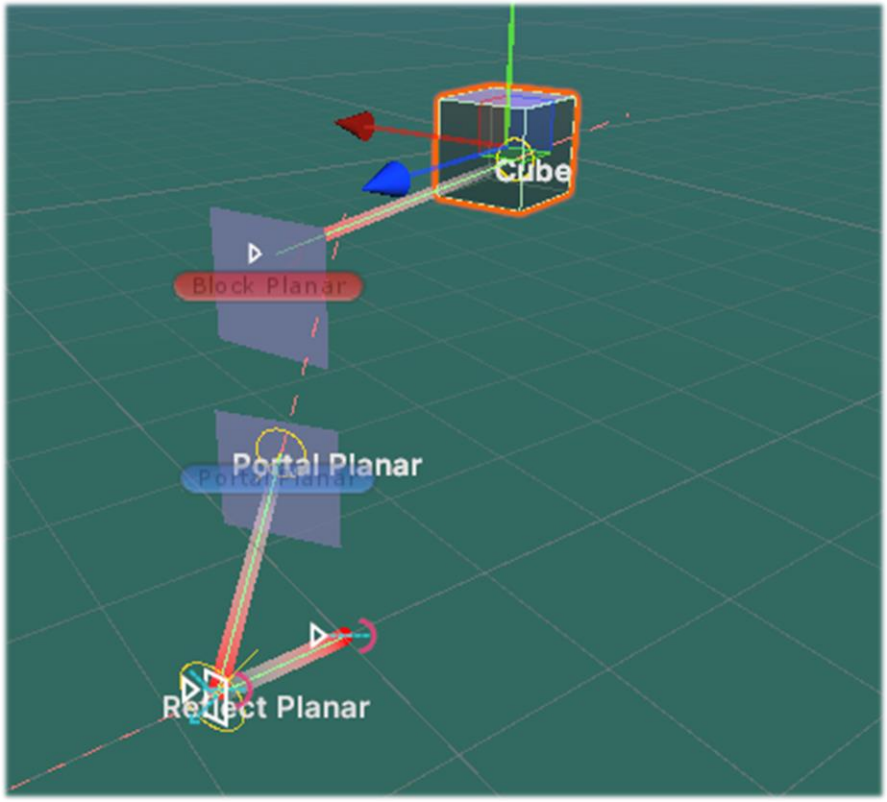


4. Then expand the process and extend the Ray's path by placing a Planar Portal further along. In its first parameter, Portal Planar asks you for the outer location, it is important that the dimensions of the exit object are also flat, otherwise you may experience strange behavior. Use **Block Planar** for output, which is used as an auxiliary Planar.




5. Next you need to access the Clone collision point, there are two ways. An easy way for non-coders is to use Stamp, which you can use as collision coordinates. The second way is Script, which we are currently testing second way.

First create a Box like below in the Wall layer and add this layer to the Ray as well, by adding the following code it will push it back when **CloneHit** hits.




```
No asset usages
public class RayTest : MonoBehaviour
{
    public RaySensor raySensor;  Changed in 0+ assets
    Event function
    private void Update()
    {
        if (raySensor.ClonePerformed)
        {
            // Box will be force out
            raySensor.CloneHit.transform.Translate( translation: -raySensor.CloneHit.normal * Time.deltaTime);
        }

    }
}
```

**ClonePerformed: Use this Property when you need the Clone detection condition. Also, in if, it causes Error missing Reference to be solved.**

**CloneHit: This Property returns the clone RaycastHit, use it for access clone hit data.**

**LastClone: This Property returns the last CloneRay itself.**

## BASE DIRECTION:

**It is to produce Clone Ray, which can be based on the following formulas. Please be careful when using this option because it seems a bit complicated.**


## LENGTH CONTROLL:

**Continues:** This option is the most common possible mode you can see, the Ray takes its remaining length out of the planar, which is normal.

**Constant:** As it is known, certain length of Ray is out of planar.

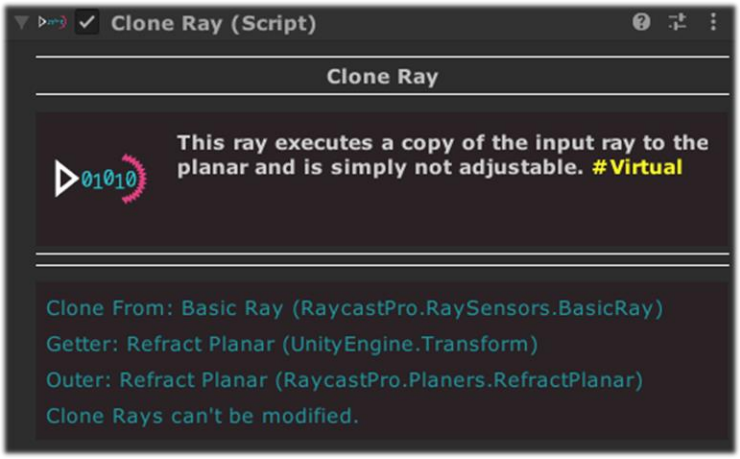
**Sync:** The same length of input Ray comes out of the Planar.

## OUTER TYPE:

This option specifies the type of Ray output, in the case of single-model Rays such as Box and Pipe, the output can be a Reference like themselves, but **PathRays** must be cloned due to their different structure. It is better to keep this option on Auto for now.

## CLONE RAY:

CloneRay is the sequence of the ray points that entered into a **Planar**. You can access its information from **CloneHit** & **LastClone** properties.




*The CloneHit property returns the resulting Hit from the planar.*

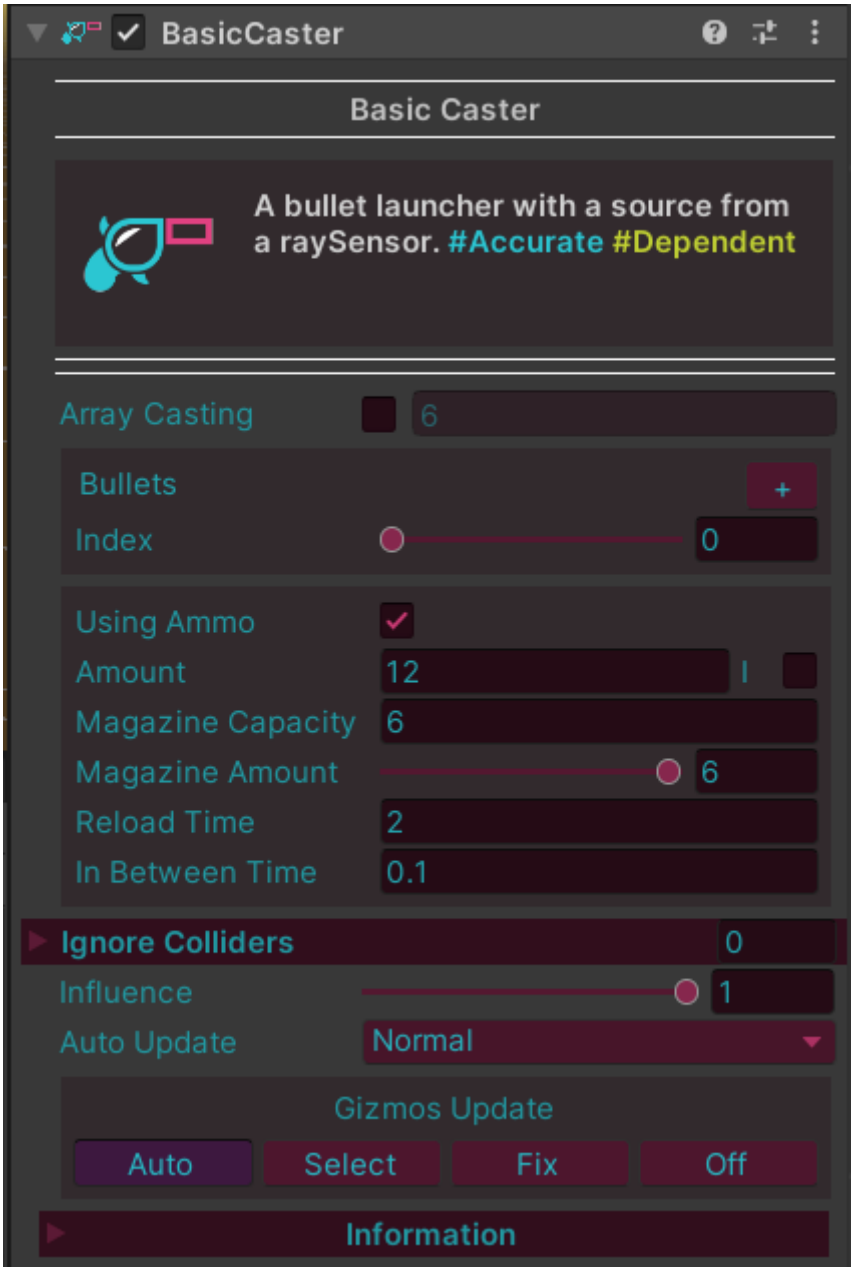
*LastClone for Get final clone information.*

# CASTERS:

Casters are very fast bullet shooting pool managers, they are dependent on some **RaySensors** source and **Bullets**, before making them, make sure you have prepared the bullets in prefab and RaySensor that you need.

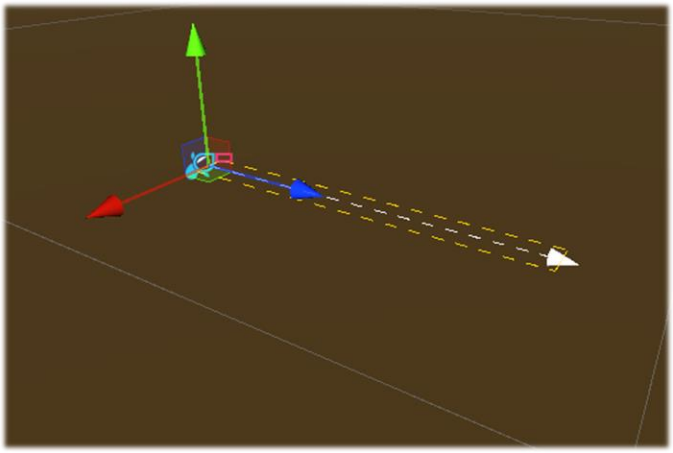
## BASIC CASTER:

This Caster model is only used for simple tasks and quick Bullet Casting, in any case, it can only mount Basic Bullet and fortunately, it does not need RaySensor, and it shoots in its **forward** direction in 3D and to the **right** in 2D.




## HOW TO SETUP BASIC CASTER:

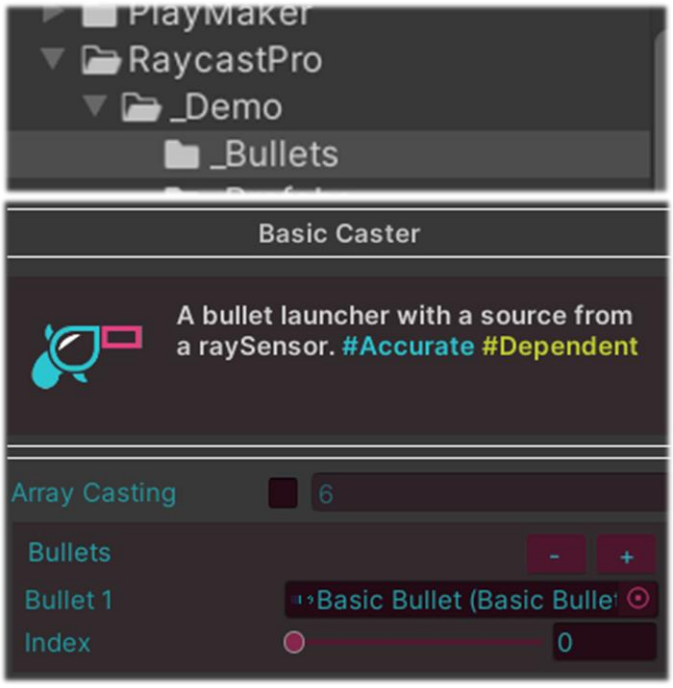
1. First, make a simple Caster from the panel.



2. Now that's enough, put a bullet on it. You can use Panel to make bullets.

***Be sure the bullet source is a prefab.***

***Fast start: for simplify tutorial I put some ready to use in the _Bullets folder, Check it!***




3. The gun will shooting automatically when start play mode.

4. For make it **Manuel**, just turn off **AutoShoot**. then Create a simple script and get the BasicCaster and call it whenever needed with the Cast method. Pay attention that the **_index** parameter refers to the bullet number [zero: first one]

```
public BasicCaster myGun;
private void Update()
{
    if (Input.GetButtonDown("Fire1"))
    {
        myGun.Cast(0);
    }
}
```

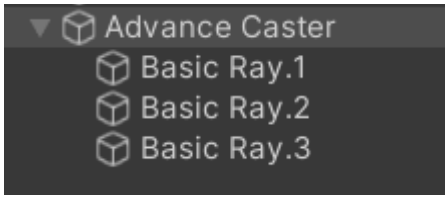
5. now Press the space key to Fire now!

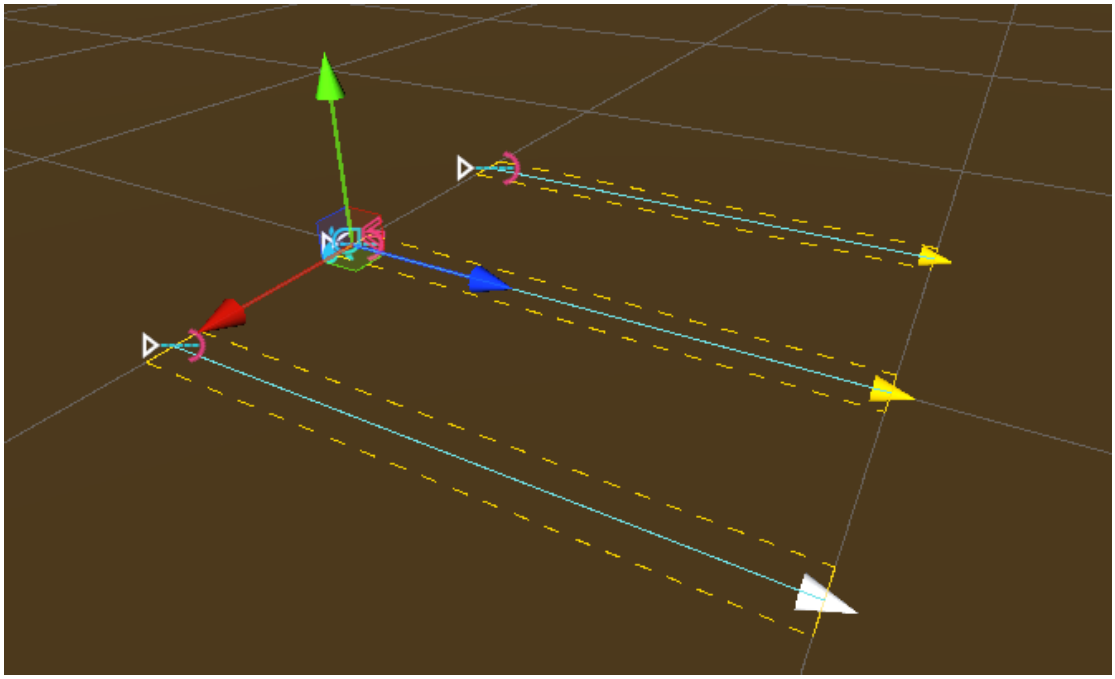
## ADVANCE CASTER:

If you have understood the above tutorial well, now you can use **AdvanceCaster** to make complex weapons, which will make your work very fast and accurate and still with the **ArrayCasting** standard. I will explain further.

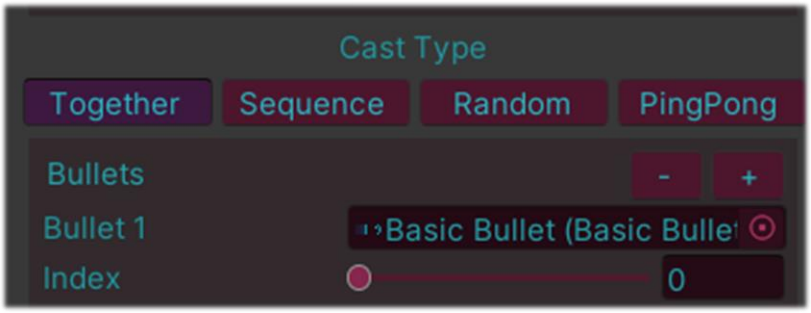
### SETUP ADVANCE CASTER:

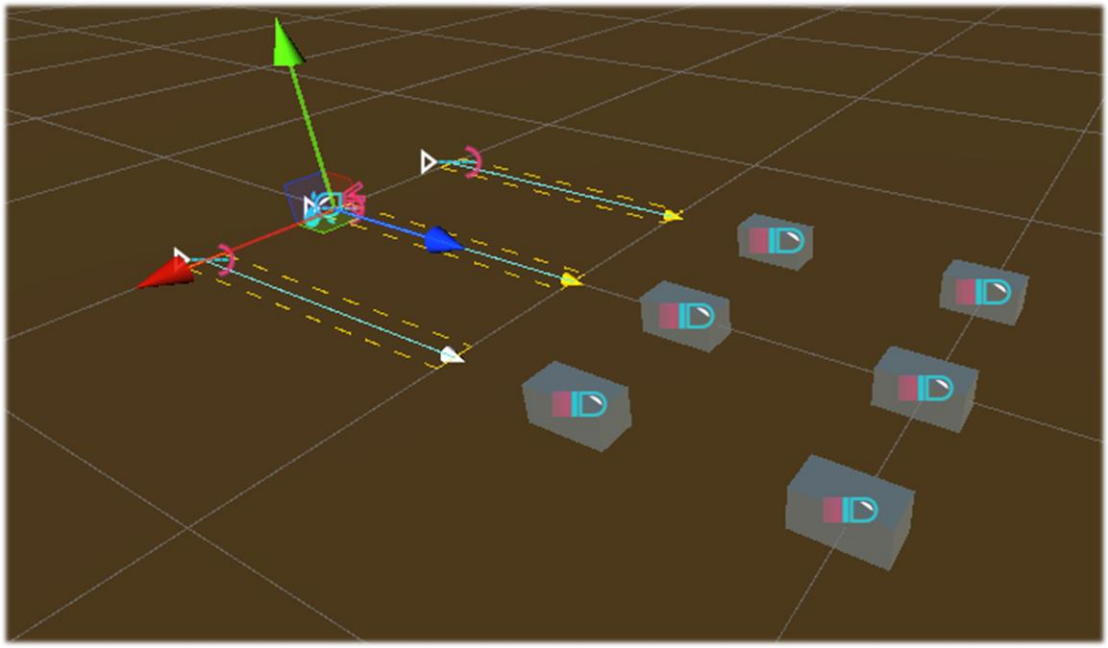
1. To start, make an Advance Caster like the one below with 3 simples BasicRays that each one in its own place. Also insert RaySensors into Caster to see a gizmo like below.



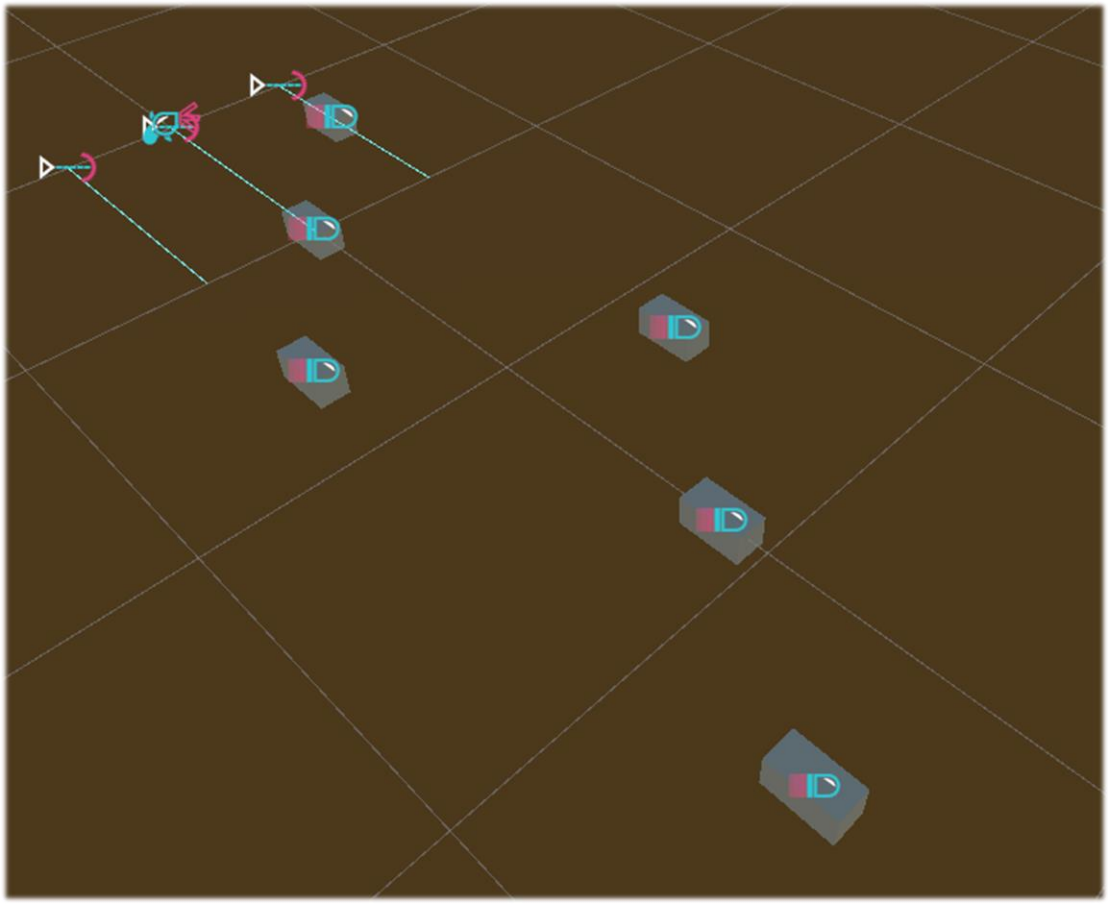



2. Now put a **BasicBullet** in the magazine as in the previous example and set the **CastType** to **Together**. By pressing the play key, the triple shot starts, and six bullets are fired from the caster each period of time.

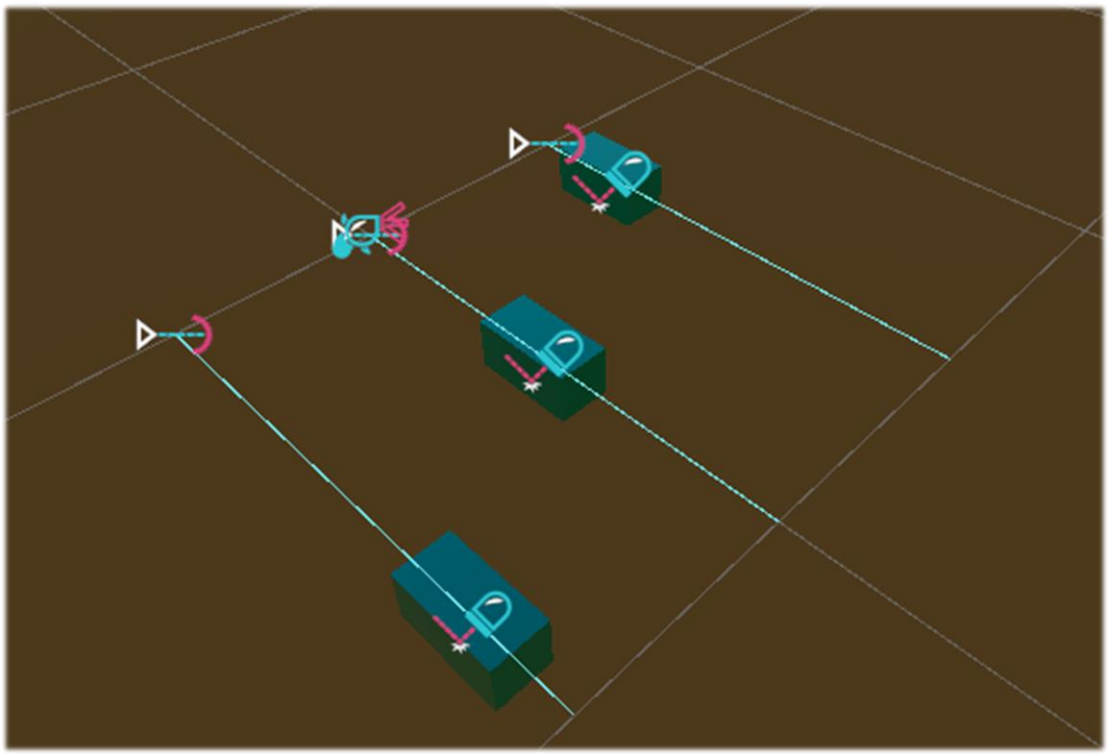


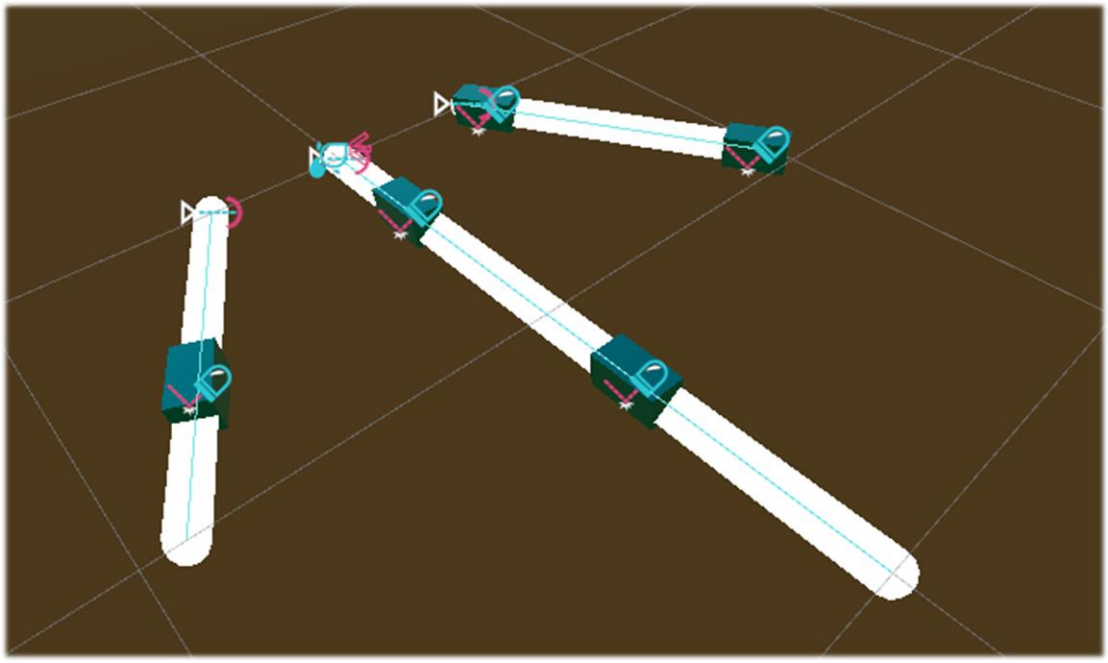


3. With the success of the first test, change the **CastType** mode and you can see that the Sequence works as follows. It is clear how the other modes work, but we will go further and try with different bullet models and RaySensor.

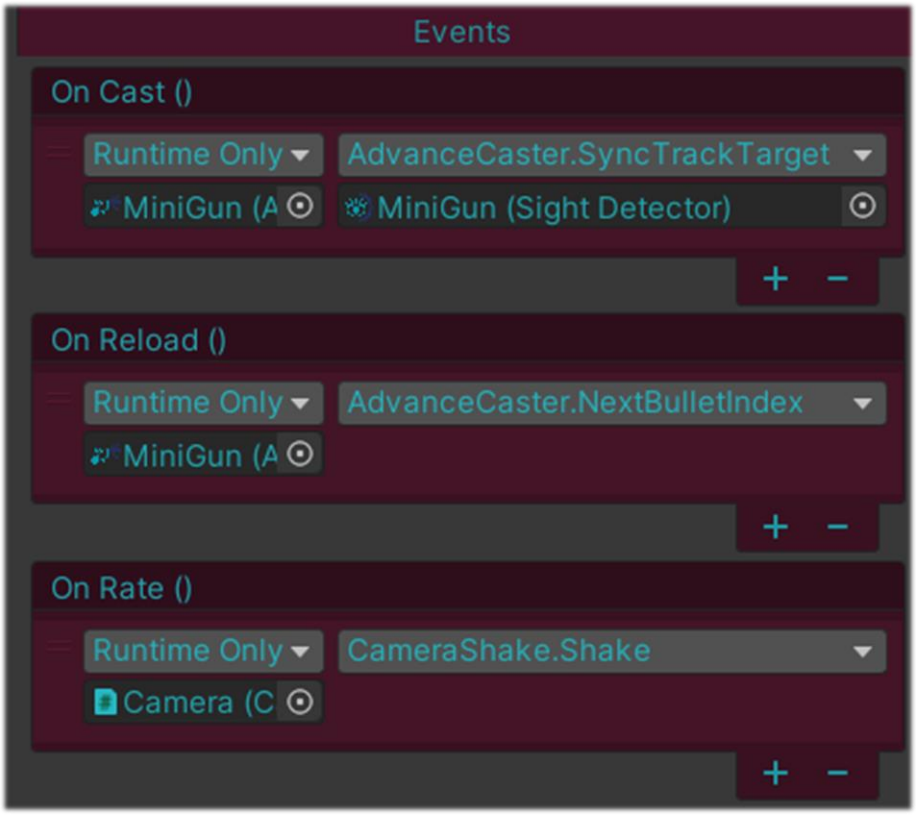


5. Now add the **PathBullet** to the magazine and change the index to 1 to fire this bullet model. If the **Render Pipeline** is standard, the color of the shooting bullet should be Cyan, but if you pay attention, these bullets do not go out of the path of the Ray. The reason is that the **PathBullet** always moves along the path of Ray no matter what type of that. Now change the size and direction of Ray, randomly and see the result.



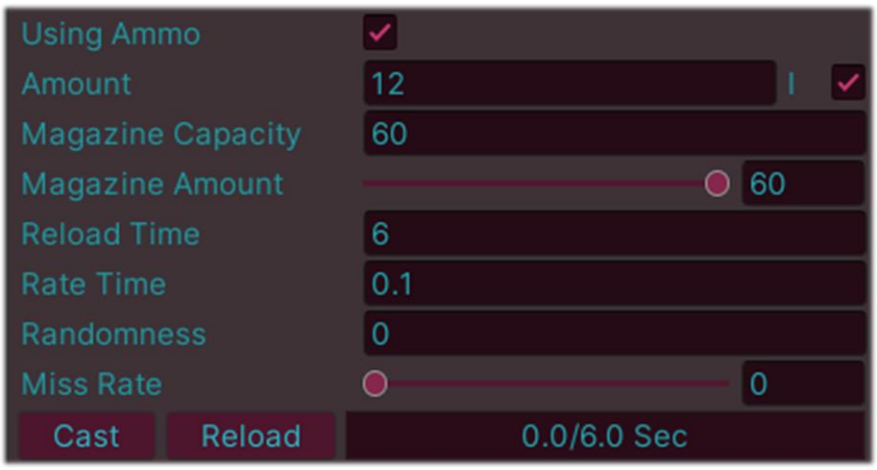


Hint: Don't forget that the best way to add modifiers like particles or camera shake. Changing bullets or... is the use of events.




## AMMO:

This method has been optimized by deactivating the instantiated bullets and reusing them to reduce garbage.



**Using Ammo:** If you tick Ammo, the projectile firing system will work in the normal way of the gun. Otherwise, it works unconditionally, and you can code manually.

**Amount:** Obviously, this value shows the number of all the beams. Option I in front of this parameter is the infinite number of bullets.

**Magazine Capacity:** The amount of magazine that is available in each firing period until Reload Time.

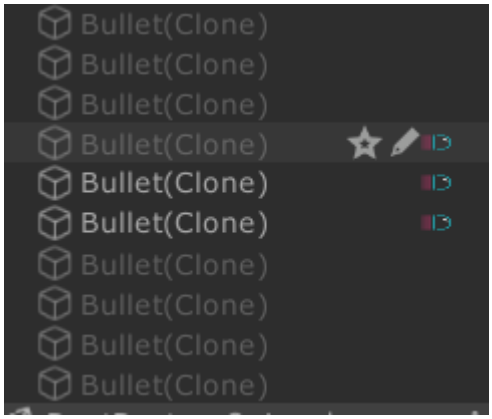
**Magazine Amount:** The current volume and the initial amount of the magazine, which cannot exceed the total volume of the magazine.

**Reload Time:** The pause time between each magazine filling cycle.

**Rate Time:** The time between firing each shot.

## ARRAY CASTING (POOL MANAGER):

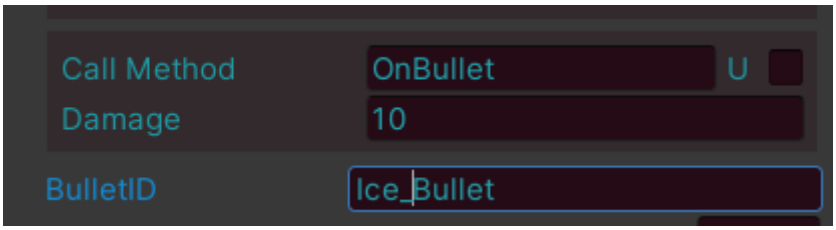
This method has been optimized by deactivating the instantiated bullets and reusing them to reduce garbage production. If your game uses a lot of bullets, use this option.

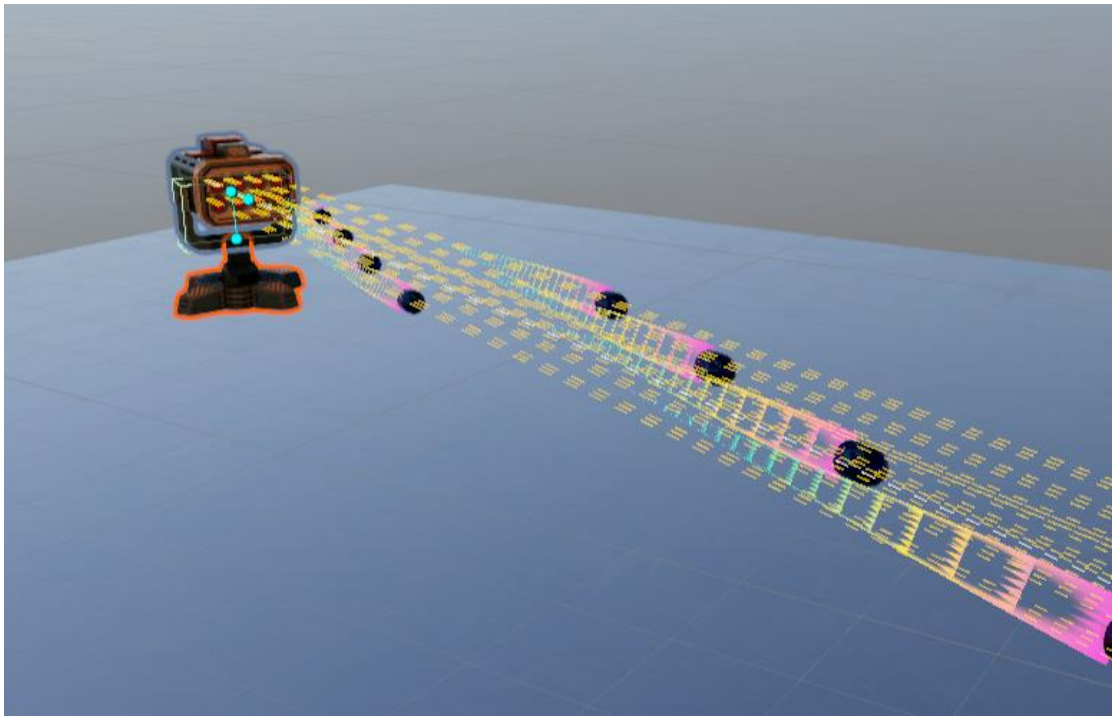



production. If your game uses a lot of bullets, use this option.

### Hints To Use:

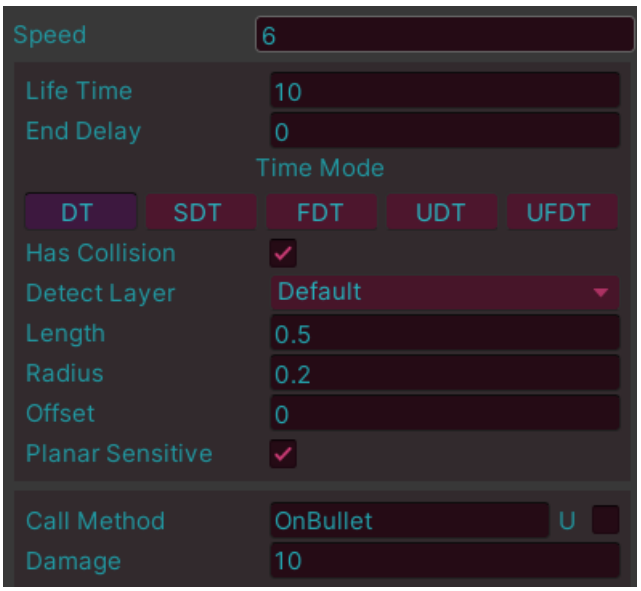
1. Array Casting is currently not modifiable in-game, so make sure the Array range is as large as you want.
2. In order to be able to change different bullets when shooting in the same way, you must use different Bullet ID.





## BULLETS:

Currently, five bullet models, both in 3D and 3D, are supported in this package, which makes almost 99% of the shooting system workable. After a brief description of the general parameters, we will examine each bullet separately.



**Speed:** This parameter is very clear, whatever value determines the speed; it travels the same unit per second.

**LifeTime:** Bullet life is in seconds, which triggers OnEnd when it runs out.

**End Delay:** This option is a pause before disabling or destroying the bullet, its use is when you want it to remain stable for a few seconds before death.

**Collision Ray:** By inserting a Ray Sensor into the body of the bullet, you can use it as a Collision. Keep in mind that this Ray should be a Child so that you can take advantage of its Planar Sensitive.

**Call Method:** This is the easiest way for you to pass an object-oriented script inside the target object with enough bullet information. Just write the desired method name on hit target object.


```
public float myCharacterHealth = 100;
void OnBullet(Bullet _bullet)
{
    myCharacterHealth -= _bullet.damage;
}
```

## INSTANT BULLET:

As its name suggests, this bullet hits the target as soon as it is fired.

## PATH BULLET:

**PathBullet** has an automatic system that always follows the path of the Ray, and it doesn't matter if your Ray is PathRay or not. In addition to the speed, there are two other ways to control the movement of the bullet.


**Local:** In the Local parameter, when the bullet is casted, it bakes the ray path at the beginning of the cast, and you will no longer see the path of the bullet being disturbed by the movement of the caster. But if this tick is not present, the bullet will always move on the Path, even if the Path itself is moving.

**RigidBody:** If you add a *RigidBody* to the bullet prefab, the bullet will use ***RigidBody.SetPosition*** instead of the **transform.Translate** algorithm, which will have a physical effect on the Colliders, but it can continue its way without affect by them.

## PHYSICAL BULLET:

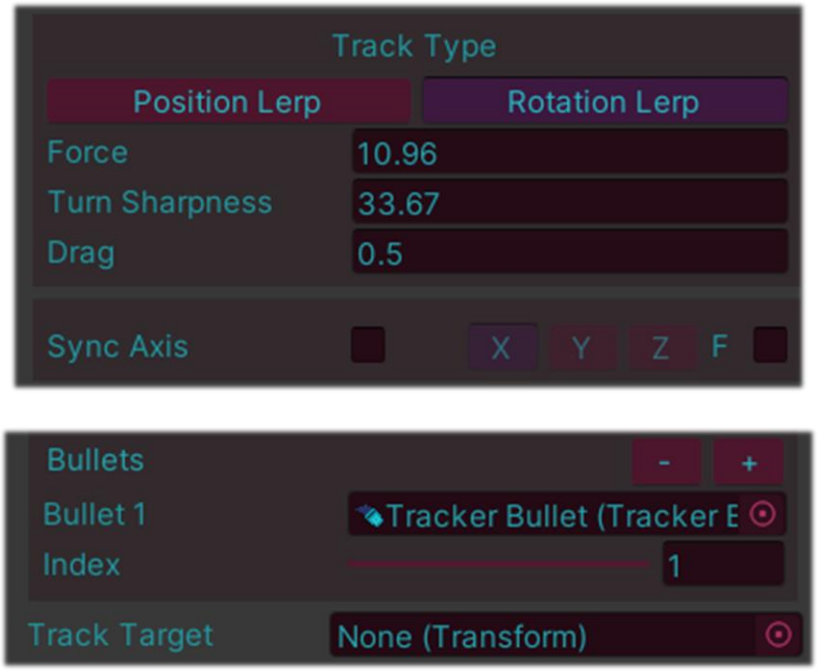
This bullet itself has a RigidBody, which is thrown with the initial force when fired.

## TRACKER BULLET:

Tracker Bullet is a bullet that can follow its Caster **Track Target** based on two different algorithms.

**Position Lerp:** the first algorithm is based on the location that will always reach the destination after a certain period, for example, I can use the Projection type of ranged heroes in Dota2 or other strategy games.

**Rotation Lerp:** The second algorithm, which is based on rotation, acts like a ballistic missile and tries to change direction towards the target, but there is no guarantee that it will always hit it.




**Force:** Initial throwing power.

**Turn Sharpness:** The power to change the curvature of the following direction.

**Drag:** The amount of friction that decrease bullet force to zero.