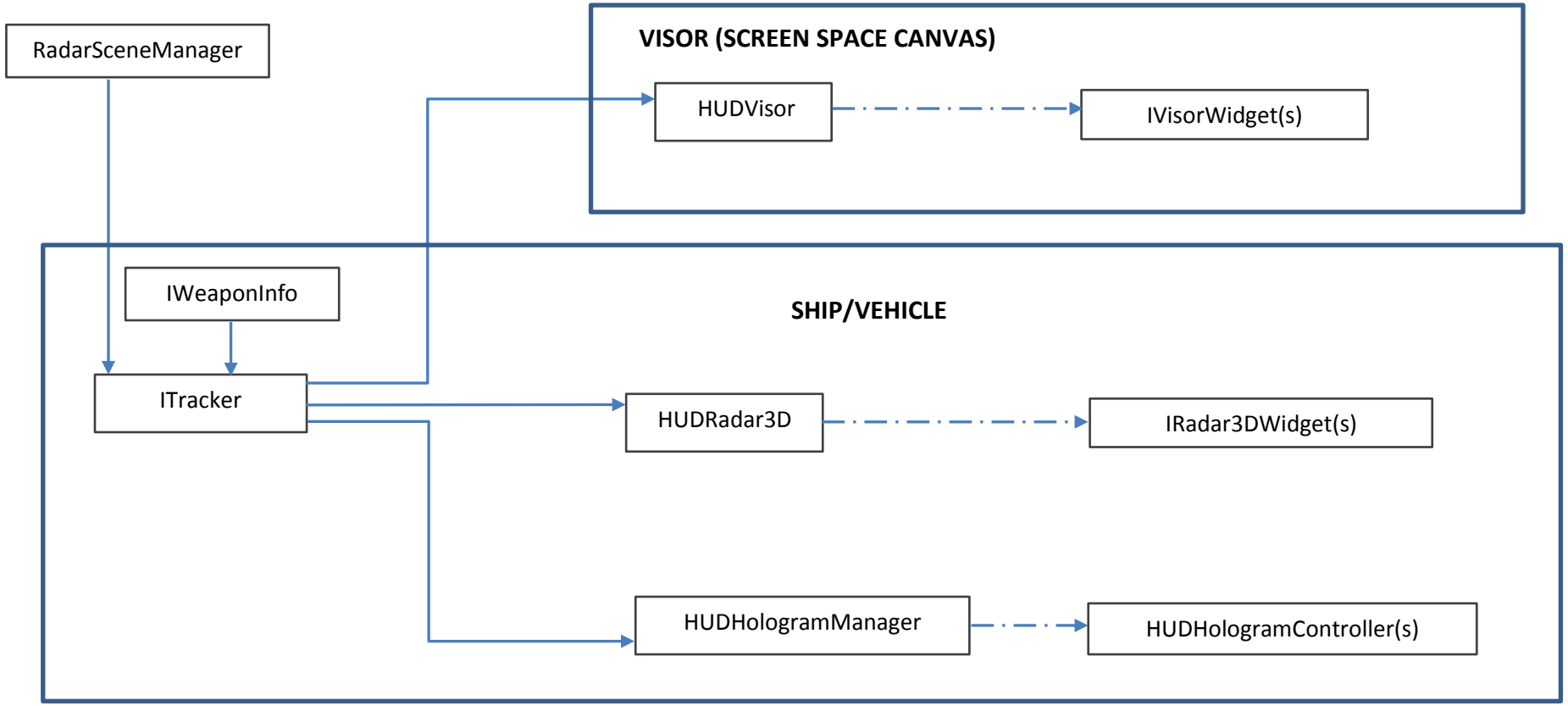
Vehicle Combat Radar

User Guide

Publisher: VSXGames

Contact: contact@vsxgames.com

Overview


RadarSceneManager
VISOR (SCREEN SPACE CANVAS)
HUDVisor
IVisorWidget(s)
IWeaponInfo
SHIP/VEHICLE
ITracker
HUDRadar3D
IRadar3DWidget(s)
HUDHologramManager
HUDHologramController(s)

1. Quick Start Guide

Enabling Target Tracking

To enable an object in your scene (for example the player ship) to track targets in the scene, do the following steps:

1. Drag the RadarSceneManager prefab into the scene;
2. Drag the DemoRadar.cs script onto the root transform of the object you wish to enable to track targets, such as the player ship (note that this will add a Rigidbody and DemoTrackable script to the game object if they are not already present);
3. Drag the DemoTrackable.cs script onto the root transform of any objects you wish to be trackable (not that this will add a rigidbody if one is not already present on the game object).

That's it! The object with the DemoRadar.cs script will now be able to access targets in the scene according to the parameters set in the DemoRadar.cs script and store it in a list in that script.

To track the targets with the visual UI, proceed to the next step.

Visualize Target Tracking

1. Go to the DemoRadar.cs script on the object that you want to display target information for, and check the 'Link to UI' field in the inspector, and uncheck it for all other DemoRadar.cs scripts in the scene.
2. Drag the *HUDVisor* prefab into the scene.

That's all! You should now see target tracking UI highlighting the targets on your screen.

To enable lead target calculation and missile locking, you must:

3. Add the DemoWeapons.cs script to the same game object as the RadarDemo.cs script.

Note that this does not actually fire weapons but simply provides dummy weapon information to enable the lead target and locking functions to display in the UI. You will have to add your own weapons code to actually enable firing weapons.

Add 3D Radar

1. Go to the DemoRadar.cs script on the player object and check 'Link to UI' if you haven't already, and uncheck it for all other DemoRadar.cs scripts in the scene.
2. Drag the *HUDRadar3D* prefab into the scene.

That's all! You should now be able to see the 3D radar displaying target information.

Add A Selected Target Hologram

1. Go to the DemoRadar.cs script on the player object and check 'Link to UI' if you haven't already, and uncheck it for all other DemoRadar.cs scripts in the scene.

2. Drag the *HUDHologram* prefab into the scene, and position it in front of the camera (in the same position as the 3D radar preferably, or around 2.5 units ahead of the camera, should be fine).

That's it! You should now be able to see a hologram of the radar's current selected target in front of the camera.

If you don't see it, make sure that 1) the radar has a selected target, and b) make sure that on each trackable target object, there is a mesh renderer somewhere in the hierarchy, or add your own custom hologram mesh to the trackable target object's DemoTrackable script.

Control Input

Drag the DemoInput prefab into the scene. You should now be able to control the radar using the following controls:

- +/- to zoom in and out of the 3D radar;
- 1 to toggle the off-screen target arrows from center to screen border in the target tracking visor;
- E - Next hostile target;
- F - Next friendly target;
- I – Target in front;
- N – Nearest hostile target;
- M – Nearest friendly target;

Before adding your own functionality to any of the interfaces or components in this package, examine them carefully to see if it is not already accessible. Careful thought has already gone into the information that is available through the interfaces such as *ITracker*, *ITrackable*, *IVisorWidget* and *IRadar3DWidget*, so most of what you need should be already available.

VR/World Space UI

To enable world space UI for the HUDVisor, do the following steps:

1. Set the Render Mode of the canvas of the HUDVisor to World Space in the Inspector;
2. IMPORTANT: Make sure the scale of the canvas is (1,1,1). When a canvas has been set to screen space before, this value often becomes changed.
3. Check the World Space UI checkbox in the inspector of the HUDVisor.

That's all. However, you may need to modify some parameters however to ensure that the visor displays properly. **Please note that most of the parameters in the HUDVisor related to size/scale refer to a visor 1 Unity unit in front of the player when in world space, or screen pixel units when in screen space**. **Tooltips have also been added to the inspector for your information.**

These settings may need adjustment:

1. The Demo_VisorWidget prefab has been designed to work with both screen space and world space, but it will likely need to be scaled down for world space. The 'World Space Scale

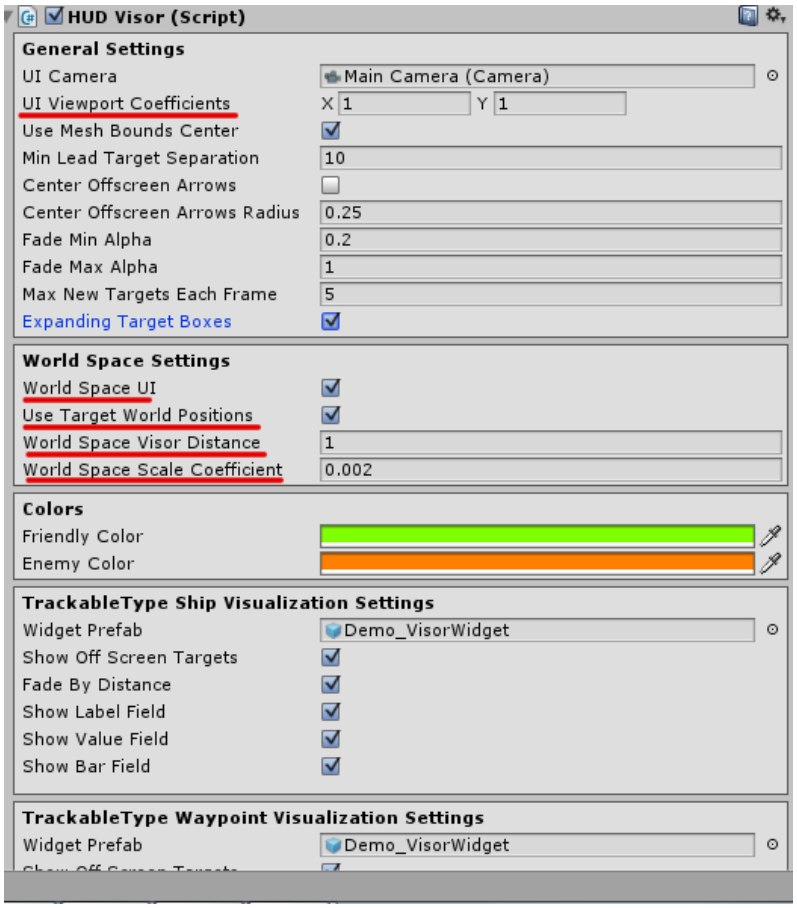
Coefficient' in the Inspector of the HUDVisor allows you to set this, and it will likely need to be a very small number (in the demo the value is 0.002).

2. The 'Center Offscreen Arrows Radius' field in the Inspector of the HUDVisor component will likely need to be changed as well, and will probably need to be around 0.3 (Unity units) for world space, and around 60 (pixels) for screen space;

Several additions have been made to enable the package to easily work with any VR platform:

- The 'Use Target World Positions' checkbox in the Inspector of the *HUDVisor* component allows you to set the target UI at the same world coordinates as the target, ensuring perfect accuracy in VR.
- The 'World Space Visor Distance' field in the Inspector of the *HUDVisor* component allows you set the distance that the UI will be displayed from the player, which is used for the offscreen arrows, as well as the rest of the UI when 'Use Target World Positions' is unchecked. This enables you to find a UI distance that is comfortable in VR and does not lead to focal isses.
- The 'UI Viewport Coefficients' field allows you to set the fraction of the viewport in which the UI takes place. This is useful as in some VR applications, the actual viewport that the user can see is smaller than the viewport of the camera.

Here's an image showing the added fields for the HUDVisor for creating world space UI:



Before adding your own functionality to any of the interfaces or components in this package, examine them carefully to see if it is not already accessible. Careful thought has already gone into the information that is available through the interfaces such as *ITracker*, *ITrackable*, *IVisorWidget* and *IRadar3DWidget*, so most of what you need should be already available.

2. Code Overview

The basic operation of this asset is as follows:

1. The **RadarSceneManager** component tracks all of the trackable objects in the scene and stores the information;
2. The **ITracker** (DemoRadar.cs) component on each ship/vehicle queries the **RadarSceneManager** component for target information and receives a list of **ITrackable** objects in return based on its scanning range;
3. The **ITracker** (DemoRadar.cs) component then uses this list when cycling targets, calculating the lead position for aiming, or locking onto the selected target.

Now for visualisation:

4. The **HUDRadar3D** controls the 3D radar, and the **HUDVisor** component controls the screen position target tracking. These components begin by querying the **ITracker** (DemoRadar.cs) component for the list of targets, to visualise the target information.
5. Both the **HUDRadar3D** and the **HUDVisor** components expose settings in the inspector so that you can completely customise the widget and its behaviour for different trackable types (based on the *RadarFunctions.TrackableType* enum in the **RadarFunctions** script – simply modify this enum with your trackable types and the inspectors will update!). This information is stored per trackable type in a list of **Visor_WidgetSettings** and a list of **Radar3D_WidgetSettings** classes for the **HUDVisor** and **HUDRadar3D** respectively.
6. For each target, the **HUDVisor** component enables a widget (specifically a component implementing the **IVisorWidget** interface) and passes information to it (based on the **Radar3D_WidgetSettings** settings information described in the previous step) using an instance of the **Visor_WidgetParameters** class.
7. For each target, the **HUDRadar3D** component enables a widget (specifically a component implementing the **IRadar3DWidget** interface) and passes information to it (based on the settings information described in the previous step) using an instance of the **Radar3D_WidgetParameters** class.

3. Modifying And Customizing Widgets

This package has been designed to be easy to implement, but to also provide a high level of customisation.

Add More Trackable Types

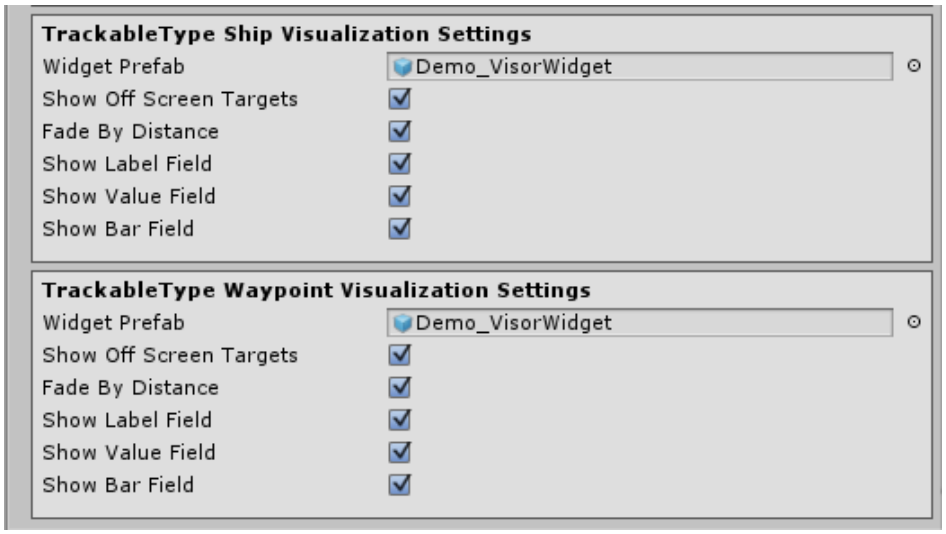
To add more trackable types – simply modify the *TrackableType* enum in the *RadarFunctions* script and all of the inspectors will update automatically!

Add More Teams

To add more teams – simply modify the *Team* enum in the *RadarFunctions* script and all of the inspectors will update automatically!

Customizing Visor UI Widgets

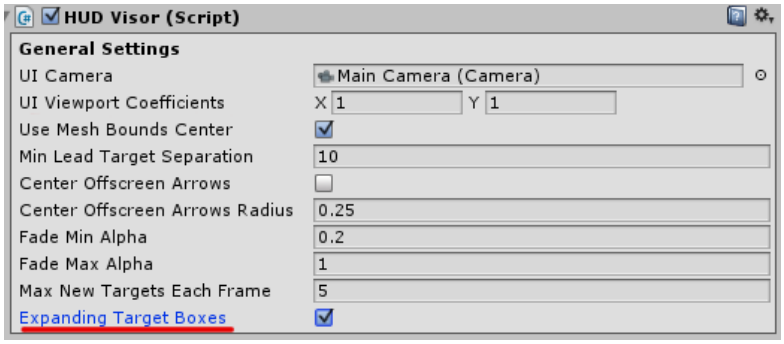
The *HUDVisor* component, located on the child object named *Visor* on the *HUDVisor* prefab, provides the ability to customize widgets on a per-Trackable-Type basis through a custom inspector. Here is an example of the types Ship and Waypoint:



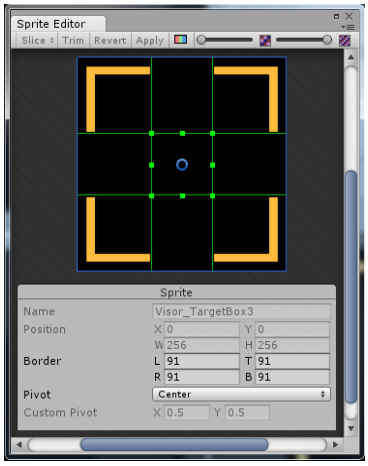
Simply set the properties for different types to your liking, and even use a totally different widget prefab of your own (it must implement the *IVisorWidget* interface however)!

Expanding Target Boxes (Visor Widgets)

As of version 1.2, the option for 'expanding target boxes' based on the mesh size of the target has been moved to HUDVisor as a single setting for ease of use. To enable expanding target boxes, simply check the field in the inspector of the HUDVisor component (see below):

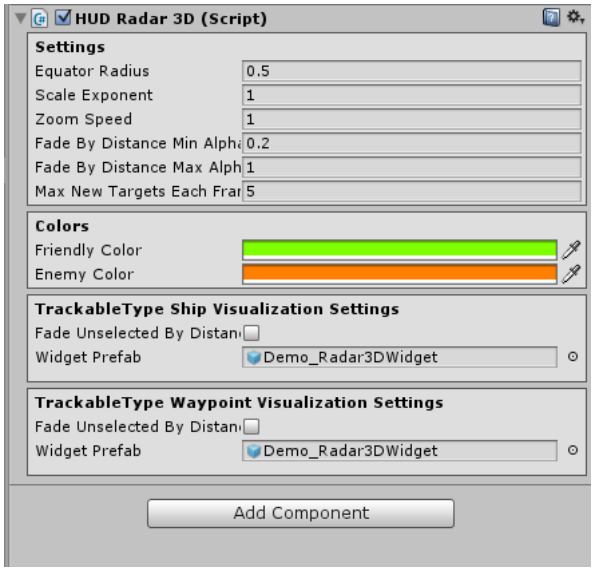


Note: If you're creating your own target box sprites (and locking sprites) and you would like them to expand and contract without changing the size of the pixels in the sprite, you must '9-slice' them in the Sprite Editor after you import them, by dragging the borders so that the sprite pixels lie outside.



Customizing Radar3D UI Widgets

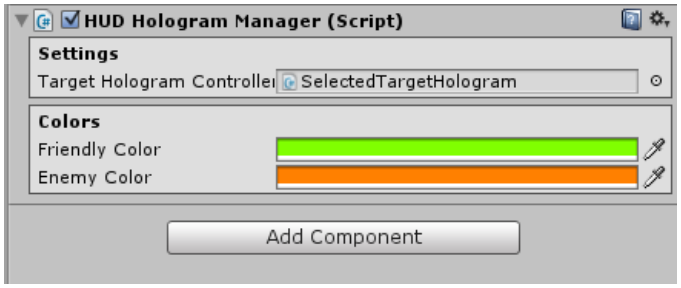
Like the *HUDVisor* component, the HUDRadar3D component (which is located on the *HUDRadar3D* prefab, provides the ability to customize widgets on a per-Trackable-Type basis through a custom inspector. Here is an example of the types Ship and Waypoint:



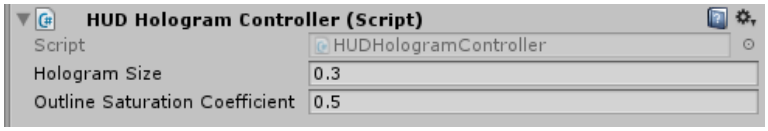
Simply set the properties for different types to your liking, and even use a totally different widget prefab of your own (it must implement the *IRadar3DWidget* interface however)!

Customizing Hologram UI Widgets

The *HologramManager* component, located on the *HUDHologram* prefab, provides the ability to customize the colors of the holograms per team:



The *HUDHologramController* component (located on a child of the *HUDHologram* prefab) provides the ability to modify basic properties of the displayed hologram:



4. Hologram Shaders

There are two hologram shaders provided – one is *VSX.Vehicles/Hologram* (for a simple hologram such as the 3D radar sphere) and the other is *VSX.Vehicles/HologramOutlined* which is for the target hologram. Both of these have a variety of properties that can be modified to your liking.

5. That's it!

Thank you for purchasing this asset!

If you have any suggestions, bugs that you've found, or any issues whatsoever with this package, don't hesitate to send an email to contact@vsxgames.com and I will get back to you as soon as possible.

Have fun and good luck with your project!