Getting Started with the WrightAngle Waypoint System

1. Welcome to Waypoints!

1.1. What's This All About?

Hello and welcome! The WrightAngle Waypoint System is a handy tool for Unity game developers looking to add helpful markers and guides into their projects. Think of it as a way to point players towards goals, highlight cool stuff in the game world, or keep track of moving things. It's designed to show clear indicators whether the target is right in front of the player or hiding off-screen.

The system was built with a few key ideas in mind: keeping things running smoothly, being flexible enough for different kinds of games, and being prefly easy to set up. It can handle targets popping in and out of existence during gameplay, too. One of it's primary goals is performance; it uses clever tricks like reusing marker images instead of constantly creating new ones, and it doesn't check target positions every single frame, which saves processing power. This focus on efficiency suggests an awareness of common performance issues in game development, aiming to provide a solution that works well even when things get busy on screen.

1.2. Cool Features Overview

- **Smart Marker Reuse (Object Pooling):** Instead of making and deleting marker images all the time (which can slow things down), the system keeps a stash of ready-to-use markers. When one's needed, it grabs one, when it's done, it puts it back. This makes things much smoother, especially if lots of markers appear and disappear quickly.
- **Central Settings File (ScriptableObject):** All the main settings (like how markers look or how fast they update) are stored in a separate file (`WaypointSettings`). This means developers can create different sets of settings (like one style for the main screen, another for a minimap) and easily reuse them across different game scenes or even projects. Changing this file updates all markers using it instantly.
- **Handles Moving Targets:** Objects become trackable by adding a `Waypoint Target` script. These targets can tell the main system when they appear or disappear using simple signals (C# events). This keeps everything organized without needing complex connections between the manager and every single target.
- **On-Screen and Off-Screen Views:** When a target is visible, its marker sits right over it. When it goes off-screen, the marker can optionally stick to the edge of the screen and point towards the target's direction.
- **Dynamic Distance Scaling:** Waypoints can now automatically change their size based on their distance from the camera, making distant markers smaller and nearby ones clearer. This is fully configurable.

- **Distance Text Display (TextMeshPro):** Optionally display the distance to the target directly on the waypoint marker. Supports Metric (m/km) and Imperial (ft/mi) units, with customizable formatting, and the text always stays upright and readable.
- **Speed Controls:** Besides reusing markers, the system lets developers choose how often it recalculates marker positions (`UpdateFrequency`). This helps balance smooth visuals with game performance.
- **Works in 2D & 3D:** Whether the game is a 3D adventure or a 2D side-scroller, the system can handle it. A simple setting tells it which type of camera view is being used.

1.3. Who's It For?

This system is great for Unity developers at almost any level who need to show players where to go or what to look at. It fits right into many game types:

- Shooters (FPS/TPS) for objectives or enemy locations.
- RPGs for quests, points of interest, or teammates.
- Simulation or Strategy games for units or locations.
- Big Open-World games for navigation.
- Basically, any project needing visual pointers in 2D or 3D space.

While gefling started is straightforward (see the guide below!), there's plenty of room for customization through the seflings file and scripting for those who need more control.

1.4. What's in the Box?

The main parts are a few code files:

- WaypointUIManager: The main controller.
- WaypointSeflings: The seflings file type.
- WaypointMarkerUI: The script for the marker image prefab.
- WaypointTarget: The script for things to track.

2. Your First Waypoints: Quick Setup Guide

2.1. Goal

This quick guide shows the minimum steps to get basic waypoint markers showing up in a Unity scene.

2.2. Step 1: Create the Settings File

First up, let's make the seflings file. This holds all the rules for the waypoints.

- In the Unity Editor's Project window, find a good folder (like Assets).
- Right-click, go to Create -> WrightAngle -> Waypoint Seflings.
- Give the new file a name, maybe MyGameWaypointSeflings.

- Select it to see its options in the Inspector window. The defaults are okay for now.

2.3. Step 2: Get Your Scene Ready

The system needs a Camera to see the world and a UI Canvas to draw the markers on.

- Make sure the scene has the main Camera that the player uses.
- Make sure there's a UI Canvas (GameObject -> UI -> Canvas). If not, create one. The markers are UI elements and need a Canvas to live on.

2.4. Step 3: Add the Manager

Time to add the brains of the operation.

- Create a new empty GameObject in the Hierarchy (GameObject -> Create Empty). Call it something like WaypointSystem.
- Select this WaypointSystem object.
- In the Inspector, click Add Component and find Waypoint UI Manager (or look under WrightAngle).
- Now, fill in the slots in the Inspector:
    - **Settings:** Drag the MyGameWaypointSeflings file created in Step 1 onto this slot.
    - **Waypoint Camera:** Drag the main gameplay Camera from the Hierarchy onto this slot.
    - **Marker Parent Canvas:** Drag a UI element's RectTransform from inside the Canvas hierarchy onto this slot. This could be the Canvas itself, but it's befler to create an empty child Panel (like "WaypointMarkerContainer") inside the Canvas and drag that Panel's RectTransform here. This tells the manager *where* to put the marker images. Make sure whatever is dragged here is actually inside a Canvas. Don't forget to drag the seflings file onto it's slot!

2.5. Step 4: Make the Marker Prefiab

The system needs a template for what the markers look like.

- Temporarily, create a basic UI Image on the Canvas (Right-click Canvas -> UI -> Image). Name it WaypointMarker_Prefab.
- Customize this Image:
    - Give it an icon using the Source Image field (make sure the image file is set to Sprite (2D and UI) in its import seflings).
    - Adjust color, size (using the RectTransform), etc. It's often best if the icon points "up".
    - **Important:** Select this WaypointMarker_Prefab object and add the

WaypointMarkerUI component (Add Component -> WrightAngle -> Waypoint Marker UI).
    - On the WaypointMarkerUI component, find the Marker Icon slot and drag the Image component (from this same object) onto it.
- Turn this configured WaypointMarker_Prefab into a reusable template (a prefab) by dragging it from the Hierarchy into the Project window (e.g., into an Assets/Prefabs folder).
- Delete the WaypointMarker_Prefab from the Hierarchy scene view.
- Finally, select the MyGameWaypointSeflings asset (from Step 1) in the Project window. Drag the new marker prefab from the Project window onto the Marker Prefab slot in the Inspector.

2.6. Step 5: Choose Your Targets

Now, tell the system *what* to point at.

- Select any GameObject in the scene that should be a target (like an enemy, treasure, or exit).
- For each one, add the WaypointTarget component (Add Component -> WrightAngle -> Waypoint Target).
- Leave Activate On Start checked for now. This makes the target automatically trackable when the game starts.

2.7. Step 6: Run It!

Hit the Play buflon in Unity.

- Markers should appear on the screen, pointing towards the target objects.
- Move the camera around. Watch how markers stay over targets when visible, and clamp to the screen edges when the target goes out of view (if Use Off Screen Indicators is on, which it is by default).

2.8. What's Next?

That's the basic setup! The system is running. To fine-tune how it looks and behaves, explore the WaypointSeflings file and the other options described below.

3. Tweaking the System: The Settings File

3.1. Why a Settings File?

The WaypointSeflings asset file acts as the central control panel for the whole system. Using a ScriptableObject like this has some nice benefits:

- **Reuse:** Use the same seflings file in multiple scenes for consistency.

- **Presets:** Create different seflings files for different looks or behaviors (e.g., main HUD markers vs. minimap icons).
- **Easy Updates:** Change seflings in one place, and it updates everywhere that file is used. This separation of configuration from scene objects encourages a more organized workflow, especially helpful if multiple people are working on the project or if experimenting with different styles is needed.

3.2. Creating and Assigning

As covered in the Quick Start: create the asset via the Assets -> Create -> WrightAngle -> Waypoint Seflings menu, and then drag it onto the Seflings field of the WaypointUIManager component in the scene.

3.3. Core Functionality Parameters

These seflings control the basic operation:

- UpdateFrequency: How often (in seconds) the system checks and updates marker positions. A lower number (like 0.02) means smoother tracking for fast action but uses more computer power. A higher number (like 0.5) saves power but might look a bit jumpy. The default (0.1) is often a good starting point, but developers should experiment to find the best balance for their game.
- GameMode: Tells the system if the main camera is Mode3D (perspective, for 3D games) or Mode2D (orthographic, for 2D games). Sefling this correctly ensures calculations work right.
- MarkerPrefab: This is where the marker prefab (created in the tutorial) is linked. It's essential, without it, no markers can be created.
- MaxVisibleDistance: Sets the maximum distance (in game units) a target can be from the camera before its marker disappears completely. This helps performance by ignoring far-off targets and keeps the screen less cluflered. Set the distance were markers disappear.
- IgnoreZAxisForDistance2D: Only used in Mode2D. If checked (default is true), the MaxVisibleDistance check ignores depth (Z-axis) and only considers the 2D plane distance (X/Y). Useful for top-down or side-scrolling games where Z might just be for visual layering.

3.4. Off-Screen Indicator Parameters

These control how markers look when the target is outside the camera's view:

- UseOffScreenIndicators: The main switch (default is true). If checked, markers for off-screen targets stick to the screen edges and point towards them. If unchecked, markers just disappear when the target leaves the view.

- ScreenEdgeMargin: Adds some padding (in pixels) between the absolute edge of the screen and the off-screen markers. This stops them from feeling squished right at the border (default is 50 pixels).
- FlipOffScreenMarkerY: A simple fix (default is false). If an off-screen marker icon points down instead of up towards the target, check this box to flip it 180 degrees automatically.

3.5.Settings Summary Table

Here's a quick reference for the WaypointSeflings parameters:

Parameter Name	What It Does (Simple Explanation)	Typical Value/Options
Core Functionality		
UpdateFrequency	How often markers update (seconds). Lower=smoother, higher=faster perf.	0.01 - 1.0
GameMode	Match camera type (3D Perspective or 2D Orthographic).	Mode3D / Mode2D
MarkerPrefab	The marker image prefab template.	GameObject (Prefab)
MaxVisibleDistance	How far away markers can be seen.	Positive number (e.g., 1000)
IgnoreZAxisForDistance2D	In 2D mode, ignore depth for distance checks?	true / false
Off-Screen Indicator		
UseOffScreenIndicators	Show markers clamped to screen edge for off-screen targets?	true / false
ScreenEdgeMargin	Pixel padding from screen edges for off-screen markers.	0 - 100 (e.g., 50)
FlipOffScreenMarkerY	Flip off-screen marker vertically if it points down?	true / false

4. Under the Hood: How the Main Parts Work (A Little)

4.1. WaypointUIManager: The Boss

Think of the `WaypointUIManager` component as the central coordinator. It's placed on an object in the scene and needs references to the Settings file, the main Camera, and the UI Canvas area where markers will live. It even checks its own setup when the game starts (`ValidateSetup`) and warns in the console if something essential is missing, like a reference or the marker prefab in the settings file, preventing runtime problems.

How it works, simply put: it wakes up periodically (based on `UpdateFrequency` in the settings). It looks at all the active `Waypoint Target` objects nearby. For each one, it figures out if it's on-screen or off-screen and calculates the distance. If a marker is needed, it grabs one from its pool of reusable marker images (this avoids creating new ones constantly). Then, it tells that specific marker (`WaypointMarkerUI`) how to display itself (position, rotation, scale, and distance text). When a target is too far away or no longer needs a marker, the manager puts the marker back into the pool for later use. It also listens for signals (static events `OnTargetEnabled`, `OnTargetDisabled`) from `WaypointTarget` components, so it knows when targets become active or inactive without needing direct links to every single one. This event-based communication makes the system flexible and scalable, as the manager doesn't need to know about every possible target beforehand. However, if markers aren't behaving as expected, tracking down whether the signal was sent and received correctly might require checking both the target and the manager. When the manager object is destroyed, it cleans up by releasing all pooled markers and stopping listening for target signals (`OnDestroy`).

4.2. WaypointMarkerUI: The Marker Itselfi

This script lives on the marker prefab. Its main job is to control the visual appearance of one single marker instance, specifically the `Marker Icon` (the Image component assigned in the prefab's Inspector) and optionally a `Distance Text Element` (a TextMeshProUGUI component). It takes instructions from the `WaypointUIManager` via its `UpdateDisplay` method.

How it works: When `UpdateDisplay` is called, the script receives screen position, on/off-screen status, distance, and other data.

- **Scaling:** If distance scaling is enabled in settings, it calculates the appropriate scale based on distance and applies it to the marker.
- **Position & Rotation:**
    - *On-Screen:* It places the marker icon directly over the target's calculated position on the screen and resets its rotation to be upright.
    - *Off-Screen:* If off-screen indicators are enabled, it calculates the correct position clamped to the edge of the screen (using the `ScreenEdgeMargin`) and rotates the marker to point towards the hidden target's direction. The `FlipOffScreenMarkerY` setting is used here to correct the rotation if needed. Pointing the wrong way, the flip setting helps the marker.
- **Distance Text:** If distance text display is enabled and a `TextMeshProUGUI` element is assigned, it formats the distance according to the selected unit system and settings, then updates the text content. Crucially, it ensures the distance text itself always remains upright and screen-aligned, regardless of the parent marker's rotation.
- As a small optimization, it also automatically disables raycasting on the marker icon and text element, assuming markers usually don't need to be clicked.

4.3. WaypointTarget: The Script

This is the simplest component. It's just a script added to any GameObject to tell the WaypointUIManager, "Hey, track this thing!".

It has a couple of seflings:

- ActivateOnStart: If checked, the target automatically tries to register itself with the manager when the game starts. If unchecked, it needs to be activated later using code.
- DisplayName: An optional name for easier identification, not used by default visuals.

How it works: When this component gets enabled or disabled (or if its GameObject is activated/deactivated), it sends out a signal (a static event: OnTargetEnabled or OnTargetDisabled). The WaypointUIManager listens for these signals to know when to start or stop tracking the target and show/hide its marker. This signaling system is key to how targets can dynamically appear and disappear during gameplay. When it will be be disabled, it sends a signal. Developers can also trigger these signals manually from other scripts using the ActivateWaypoint() and DeactivateWaypoint() functions on the WaypointTarget component. This is useful for controlling markers based on game events like quests starting or items being picked up. In the editor, selecting an object with this component shows a helpful colored sphere (gizmo) indicating if it's currently being tracked (green) or not (yellow).

5. Going Further: Tips, Tricks & Fixes

5.1. Perfiormance Tips

While designed to be efficient, here are ways to keep things running smoothly:

- **Adjust Update Speed:** The UpdateFrequency in WaypointSeflings is the main performance knob. Higher values (e.g., 0.2 seconds instead of 0.1) mean less frequent updates, saving CPU power, especially with many targets. Experiment to find a good balance.
- **Use Max Distance:** Keep MaxVisibleDistance reasonable. Don't track targets hundreds of miles away if they aren't relevant. This culls distant targets, reducing the manager's workload.
- **Simple Markers:** Keep the MarkerPrefab visuals reasonably simple. Complex hierarchies or effects on the prefab can impact UI performance.
- **Deactivate Unused Targets:** If a target is no longer needed (e.g., quest completed, enemy defeated), make sure its WaypointTarget component is

deactivated either by disabling the component/GameObject or by calling DeactivateWaypoint() via script. This removes it from the manager's active list.

5.2.Controlling Markers with Code

Often, waypoints shouldn't be active all the time. They might only appear when a quest starts or disappear when an item is collected. This is done by unchecking ActivateOnStart on the WaypointTarget component and then calling its functions from other scripts.

Here's a basic example for activating a quest target:

```
using UnityEngine;
using WrightAngle.Waypoint; // Need this line!

public class QuestController : MonoBehaviour
{
  public GameObject questObjectiveObject; // Assign in Inspector

 public void BeginQuest()
 {
      if (questObjectiveObject!= null)
      {
          WaypointTarget target =
questObjectiveObject.GetComponent<WaypointTarget>();
          if (target!= null &&!target.IsRegistered) // Check if it has the component and isn't already
active
          {
              target.ActivateWaypoint(); // Tell the system to start tracking it
          }
      }
 }

 public void EndQuest()
 {
       if (questObjectiveObject!= null)
      {
          WaypointTarget target =
questObjectiveObject.GetComponent<WaypointTarget>();
```

```
        if (target!= null && target.IsRegistered) // Check if it's currently active
        {
            target.DeactivateWaypoint(); // Tell the system to stop tracking it
        }
    }
 }
}
```

Remember to add using WrightAngle.Waypoint; at the top of any script that needs to talk to the waypoint components. Call ActivateWaypoint() to show the marker and DeactivateWaypoint() to hide it.

5.3. Customizing Marker Looks

The visual style comes entirely from the MarkerPrefab created and assigned in the WaypointSeflings.

- **Design:** Build the prefab using standard Unity UI elements (Images, Text, etc.).
- **Icon:** Make sure the Marker Icon field on the WaypointMarkerUI script (on the prefab root) points to the main Image element that should be positioned and rotated.
- **Rotation:** The system assumes the marker icon points "up" (along its local Y-axis) by default for off-screen rotation. If it points down, use the FlipOffScreenMarkerY sefling. If it points sideways, the prefab's rotation might need adjusting, or the calculation code in WaypointMarkerUI could be modified.

5.4. Troubleshooting Common Issues

Having trouble? Here are some common fixes:

- **Problem: Markers aren't showing up at all.**
  - **Things to Check:**
    - Is there exactly one active `WaypointUIManager` in the scene, and is it enabled?
    - Check the Console window for error messages when starting the game. The manager logs errors if setup is wrong.
    - Are `Settings`, `Waypoint Camera`, and `Marker Parent Canvas` all assigned correctly on the `WaypointUIManager`?
    - Is the `Marker Parent Canvas` object active and definitely inside an active UI Canvas?
    - Is a `WaypointSettings` asset assigned to the manager?
    - Inside the `WaypointSettings` asset, is the `Marker Prefab` slot assigned?
    - Does the `MarkerPrefab` itself exist, have `WaypointMarkerUI` script

on its root, and is the `Marker Icon` field assigned on that script? If using distance text, is the `Distance Text Element` also assigned if `DisplayDistanceText` is enabled?
    - Are `WaypointTarget` components on the objects to track? Are those objects active?
    - Is `ActivateOnStart` checked on the `WaypointTarget`, OR has `ActivateWaypoint()` been called via script?
    - Are the targets within the `MaxVisibleDistance` from the camera?
    - If using Distance Scaling, ensure `MinScaleFactor` isn't 0 causing them to be invisible by default if very far, and that `DefaultScaleFactor` is greater than 0.
    - The frequent need to double-check these Inspector assignments suggests that ensuring all the pieces are correctly linked during setup is crucial, as missing references are a likely cause if things aren't working. Don't forget to check none of the references are missing.
- **Problem: On-screen markers show, but off-screen ones don't.**
    - **Things to Check:**
        - In WaypointSeflings, is UseOffScreenIndicators checked?
        - Are the targets still within MaxVisibleDistance even when off-screen?
- **Problem: Off-screen markers point the wrong way.**
    - **Things to Check:**
        - Does the marker icon graphic point "up"? If not, try checking FlipOffScreenMarkerY in WaypointSeflings or adjust the prefab rotation.
        - Is GameMode (2D/3D) in WaypointSeflings set correctly for the camera being used?
- **Problem: Game performance drops with many waypoints.**
    - **Things to Check:**
        - Increase UpdateFrequency in WaypointSeflings (try 0.2 or higher).
        - Lower MaxVisibleDistance in WaypointSeflings.
        - Simplify the graphics/complexity of the MarkerPrefab.
        - Make sure targets are being deactivated (DeactivateWaypoint() or disabling object/component) when no longer needed.
- **Problem: Console shows errors about missing references on startup.**
    - **Things to Check:**
        - Carefully re-check all assigned slots on the WaypointUIManager (Seflings, Camera, Canvas Parent) and within the assigned WaypointSeflings asset (Marker Prefab). The error message usually specifies which one is missing.

6. Quick Ref: What Those Words Mean

In case it helps, here's what some of the key terms mean:

- `WaypointUIManager`: The main script that controls everything.
- `WaypointSettings`: The project file holding all the settings.
- `WaypointMarkerUI`: The script on the marker prefab that handles its look, scaling, and distance text.
- `WaypointTarget`: The script added to objects to make them trackable.
- `Object Pooling`: Reusing marker images to help the game run faster.
- `Marker Prefab`: The template image/UI element used for the markers, potentially including a TextMeshPro element.
- `On-Screen Marker`: How the marker looks when the target is visible.
- `Off-Screen Indicator`: How the marker looks (stuck to screen edge) when the target is hidden.
- `UpdateFrequency`: How often the system updates marker positions.
- `MaxVisibleDistance`: How far away a target can be before its marker vanishes.
- `Distance Scaling`: Feature to change marker size based on distance.
- `Distance Text`: Optional text on markers showing distance to target.
- `TextMeshProUGUI (TMPro)`: Unity's advanced text solution, used for the distance display.
- `ScriptableObject`: A type of Unity asset file that stores data (like `WaypointSettings`).

I wish you the best, and good luck with your project :)