1 Anti-Wallhack Visibility System

1.1 Overview

Anti-Wallhack Visibility System is a Unity-based server-side solution designed to prevent wallhacks and unauthorized player detection in competitive or multiplayer games. It validates whether a player truly has line-of-sight to another using precise linecast checks and directional sampling, ensuring that visibility is based on legitimate rendering logic not client-side manipulation.

This asset is ideal for developers building secure PvP environments, tactical shooters, stealth games, or any scenario where visibility must be verified independently of the client. It runs entirely on the server or host, making it resistant to cheating and spoofed rendering.

1.1.1 Core Purpose

- Prevents wallhacks by validating visibility through geometry
- Ensures that visibility logic is server-authoritative
- Uses safe rendering logic based on face-aligned normals and line sampling

1.1.2 What It Offers

- Observer-based visibility sampling
- Dynamic line generation per face
- Field-of-view validation with aspect ratio and pitch control
- Real-time visibility events per observer/player pair
- Editor tools for debugging visibility in play mode
- UnityEvent integration for easy event handling in the Editor

1.2 Features

- Server-Side Visibility Validation
- Anti-Wallhack Detection
- Observer-Based Sampling
- Field of View Configuration
- Dynamic Line Generation
- Real-Time Visibility Events
- UnityEvent Integration for Visibility Events
- Editor Debug Tools
- Modular Architecture
- Multiplayer-Ready

1.3 Setup Guide

1.3.1 1. Add the ObserverManager

- Create an empty GameObject in your scene and name it `ObserverManager`.
- Attach the `ObserverManager` component.
- This component manages all observers and players in the visibility system.
- **Note:** The `ObserverManager` uses `[DefaultExecutionOrder(-200)]` to ensure it initializes early in the Unity update cycle, before other components like `PlayerObserver` and `PlayerVisibilityDetector`.

1.3.2 2. Add PlayerObserver to Observers

Attach the `PlayerObserver` component to a GameObject representing the player's camera (e.g., camera, AI, or player entity). This GameObject must track the position and rotation of the player's camera.

Hierarchy Recommendations:

- If the camera is fixed relative to the player, attach `PlayerObserver` to a **child GameObject** of the player (e.g., under the same parent as the `PlayerVisibilityDetector`).
- Ensure the GameObject's rotation is updated to match the camera's orientation. If the camera moves independently, the `PlayerObserver` GameObject must follow its position and rotation.

Configuration:

- `cameraFOVAngle`: Set the vertical field of view angle (e.g., 60 degrees).
- `viewDistance`: Define the maximum detection distance (e.g., 30 meters).
- `aspectRatio`: Set the aspect ratio for horizontal FOV calculation (e.g., 16:9).
- `showGizmos`: Enable to visualize the FOV cone in the Scene view (green when a player is visible, red when not).

```
[SerializeField] private float cameraFOVAngle = 60f;
[SerializeField] private float viewDistance = 30f;
[SerializeField] private Vector2 aspectRatio = new(16f, 9f);
[SerializeField] private bool showGizmos = true;
```

1.3.3 3. Add PlayerVisibilityDetector to Players

Attach `PlayerVisibilityDetector` to each player or target to be detected, preferably at the **root** of the player GameObject.

Important: A `BoxCollider` (`playerBox`) is required to calculate face-aligned normals and generate sampling lines. Ensure the `BoxCollider` is attached to a **child GameObject** of the GameObject containing the `PlayerVisibilityDetector`.

```
[SerializeField] private BoxCollider playerBox;
```

Collider Setup Recommendations:

- **Hierarchy:** Place the `BoxCollider` on a child GameObject of the player to ensure proper transform calculations.
- **Size Considerations:**
  - Larger colliders may cause early detection around corners.
  - Smaller colliders may delay visibility and risk wallhack exposure.
  - Tune size based on level design and desired detection timing.

Layer Configuration for Obstacles:

- Assign obstacles (e.g., walls, objects) to a specific layer, such as `Obstacles`.
- Configure the `obstaclesMask` field in `PlayerVisibilityDetector` to include these layers to ensure accurate visibility checks.

```
[SerializeField] private LayerMask obstaclesMask;
```

1.3.4 4. Add VisibilityUnityEventRelay (Optional)

To handle visibility events directly in the Unity Editor without scripting, attach the `VisibilityUnityEventRelay` component to the same GameObject as the `PlayerVisibilityDetector`.

- Configure the `onVisibilityChanged` UnityEvent in the Inspector to trigger actions when visibility changes (e.g., enabling/disabling UI, playing animations).
- Ensure the `PlayerVisibilityDetector` component is present, as it is required by `VisibilityUnityEventRelay`.

```
[SerializeField] private UnityEvent<bool> onVisibilityChanged;
```

1.3.5 5. Connect Components

- `PlayerObserver`, `PlayerVisibilityDetector`, and `VisibilityUnityEventRelay` auto-register with `ObserverManager` at runtime.

1.3.6 6. Enable Gizmos for Debugging

- In Scene view, enable Gizmos to visualize:
  - Observer FOV cones (green when a player is visible, red when not)
  - Face normals
  - Linecast paths (fixed and dynamic lines)

1.3.7 7. Listen to Visibility Events (Optional)

You can handle visibility changes in two ways:

Option 1: Using VisibilityUnityEventRelay (Recommended for Editor) Configure the `onVisibilityChanged` UnityEvent in the Inspector to trigger actions when a player becomes visible or invisible.

Option 2: Manual Subscription (Advanced) Subscribe to visibility changes programmatically:

```
observerManager.OnVisibilityChanged += (observerId, player,
   isVisible) =>
{
    if (isVisible)
        Debug.Log($"Player {player.name} is now visible to
           Observer {observerId}");
    else
        Debug.Log($"Player {player.name} is no longer visible to
           Observer {observerId}");
};
```

1.4 Component Breakdown

1.4.1 ObserverManager

- Manages registration and unregistration of observers and players.
- Tracks visibility states and dispatches `OnVisibilityChanged` events.
- Reuses observer IDs for efficiency.
- Uses `[DefaultExecutionOrder(-200)]` to ensure early initialization in the Unity update cycle, allowing dependent components to rely on it.

Key Fields:

- `Instance`: Singleton instance for global access.
- `OnVisibilityChanged`: Event triggered when a player's visibility changes for an observer.

Key Methods:

- `RegisterObserver(PlayerObserver observer)`: Assigns a unique ID to an observer.
- `RegisterPlayer(PlayerVisibilityDetector player)`: Adds a player to the visibility tracking system.
- `UnregisterPlayer(PlayerVisibilityDetector player)`: Removes a player from tracking.
- `UnregisterObserver(int observerId)`: Removes an observer and recycles its ID.
- `ChangeVisibility(int observerId, PlayerVisibilityDetector player, bool state)`: Updates visibility state and triggers events.
- `IsPlayerVisibleToObserver(int observerId, PlayerVisibilityDetector player)`: Checks if a player is visible to an observer.
- `GetAllPlayersExcept(PlayerVisibilityDetector exclude)`: Returns all registered players except the specified one.
- `GetAllObserverIds()`: Returns all active observer IDs.

- `GetObserver(int id)`: Retrieves an observer by ID.
- `IsValidObserver(int id)`: Verifies if an observer ID is valid.

1.4.2 PlayerObserver

- Represents the player's camera, tracking its position and rotation for visibility checks.
- Defines FOV parameters and performs visibility checks based on distance, horizontal, and vertical angles.
- Subscribes to events and provides optional Gizmo visualization (FOV cone).

Key Fields:

- `myPlayer`: Reference to the associated `PlayerVisibilityDetector` (automatically fetched from parent if not set).
- `cameraFOVAngle`: Vertical field of view angle.
- `viewDistance`: Maximum detection distance.
- `aspectRatio`: Aspect ratio for horizontal FOV calculation.
- `showGizmos`: Enables visualization of the FOV cone (green for visible, red for not visible).

Key Methods:

- `DetectVisiblePlayersByFOV()`: Checks which players are within the FOV and distance.
- `HandleVisibility(...)`: Handles visibility events for debugging.

Hierarchy Note: Place the `PlayerObserver` on a GameObject that matches the player's camera position and rotation. If the camera is fixed, it can be a child of the player GameObject, with only the rotation updated to match the camera.

1.4.3 PlayerVisibilityDetector

- Calculates face normals
- Performs linecasts (fixed and dynamic lines)
- Tracks visibility per observer
- Notifies `ObserverManager`

Key Fields:

- `playerBox`: Reference to the `BoxCollider` (must be on a child GameObject).
- `obstaclesMask`: Layer mask for obstacles that block visibility (e.g., walls, objects).
- `dynamicLineSpeed`: Controls the speed of dynamic line movement for visibility sampling. Higher speeds improve detection accuracy but may impact performance.

- `verticalFixedLineCount`, `horizontalFixedLineCount`: Number of fixed lines for visibility sampling.
- `verticalDynamicLineCount`, `horizontalDynamicLineCount`: Number of dynamic lines for visibility sampling.
- `dynamicLineSpacing`: Spacing between dynamic lines.
- `horizontalColumnCount`, `verticalRowCount`: Number of columns and rows for line sampling.
- `alignmentThreshold`, `targetAlignmentThreshold`: Thresholds for face normal alignment.
- `showGizmos`, `dynamicLineColor`, `fixedLineColor`, `detectedLineColor`, `faceLineColor`, `boxColor`: Gizmo visualization settings.

Key Methods:

- `UpdateVisibilityFromAlignedNormals()`
- `AddObserver(...)`
- `RemoveObserver(...)`

Important Adaptation Note for Networking (e.g., Photon Fusion): The default `Update()` method calls `UpdateVisibilityFromAlignedNormals()` every frame, which works for single-player or non-networked scenarios. For multiplayer frameworks like Photon Fusion, replace this with a call in `FixedUpdateNetwork()` (not `Render()`) to ensure deterministic simulation, client-side prediction, and resimulation support. This is crucial for linecast-based checks, as they depend on fixed-tick physics and networked state.

Example for Photon Fusion (Host Mode):

```
public class PlayerVisibilityDetector : NetworkBehaviour
{
    public override void FixedUpdateNetwork()
    {
        if (Object.HasStateAuthority)
        {
            UpdateVisibilityFromAlignedNormals();
        }
    }
}
```

- **Why FixedUpdateNetwork()?** It runs at fixed ticks (e.g., 60Hz), integrates with Fusion's simulation loop, and supports rollbacks for accurate multiplayer sync. Avoid `Render()` for logicuse it only for visual interpolation.
- **Benefits:** Reduces desyncs, improves performance, and maintains server authority.

1.4.4 VisibilityUnityEventRelay

- Relays visibility events from `ObserverManager` to a `UnityEvent` for easy configuration in the Unity Editor.

- Requires a `PlayerVisibilityDetector` on the same GameObject.

Key Fields:

- `onVisibilityChanged`: A `UnityEvent<bool>` that triggers when the player's visibility changes, passing `true` (visible) or `false` (invisible).

Key Methods:

- `HandleVisibilityChanged(int observerId, PlayerVisibilityDetector detector, bool isVisible)`: Filters and forwards visibility events for the attached `PlayerVisibilityDetec`

1.4.5 ObserverManagerEditor

- Custom inspector for debugging
- Shows observer/player lists
- Displays visibility status
- Editor-only, no runtime impact

1.5 How Visibility Works

1.5.1 Core Concepts

- **Field of View (FOV):** Defines detection cone, calculated using `cameraFOVAngle` and `aspectRatio`.
- **Face-Aligned Normals:** Sample directions from bounding box.
- **Line Sampling:** Grid of vertical/horizontal lines per face, including fixed and dynamic lines.
- **Linecast Validation:** Physics checks for unobstructed paths.
- **Dynamic Line Speed:** Controls how fast dynamic lines move, impacting detection accuracy and performance.
- **Visibility Events:** Triggered on visibility state changes, accessible via `ObserverManager` or `VisibilityUnityEventRelay`.

1.5.2 Example Flow

1. `PlayerObserver` detects players in its FOV based on `cameraFOVAngle`, `viewDistance`, and `aspectRatio`.
2. System calculates normals and sampling lines (fixed and dynamic).
3. Linecasts are performed for both fixed and dynamic lines.
4. If any line reaches the target, visibility is confirmed.
5. `ObserverManager` updates state and triggers events via `OnVisibilityChanged` or `VisibilityUnityEventRelay`.

1.6 Customization

1.6.1 Field of View Settings

- `cameraFOVAngle`, `viewDistance`, `aspectRatio`: Adjust to match the player's camera settings.
- Ensure `aspectRatio` reflects the camera's aspect ratio for accurate horizontal FOV calculation.

1.6.2 Line Sampling Resolution

- `fixedLineCount`, `dynamicLineCount`, `samplingMode`
- **Dynamic Line Speed (`dynamicLineSpeed`):** Controls the speed of dynamic line movement. Higher values improve detection accuracy by scanning faster but may increase performance cost. In some cases, high speeds may reduce the need for fixed lines, allowing users to disable them by setting `verticalFixedLineCount` and `horizontalFixedLineCount` to 0.

1.6.3 Obstacle Layer Configuration

- Set `obstaclesMask` to include layers for objects that should block visibility (e.g., `Obstacles` layer). Ensure all relevant scene objects are assigned to these layers to prevent false positives in visibility checks.

1.6.4 Event Handling

- **Using VisibilityUnityEventRelay (Recommended):** Configure the `onVisibilityChanged` UnityEvent in the Inspector to trigger actions like UI updates, animations, or sound effects without writing code.
- **Manual Subscription:** Use `ObserverManager.OnVisibilityChanged` for custom logic in scripts.

1.6.5 Gizmo Visualization

- Toggle `showGizmos` per observer
- Visualize normals, cones, fixed/dynamic linecasts, and detected lines with customizable colors (`dynamicLineColor`, `fixedLineColor`, `detectedLineColor`, `faceLineColor`, `boxColor`).
- `PlayerObserver` shows the FOV cone (green for visible, red for not visible).

1.6.6 Performance Optimization

- Adjust `dynamicLineSpeed` for a balance between accuracy and performance
- Lower line counts (`verticalFixedLineCount`, `horizontalFixedLineCount`, `verticalDynamicLi` `horizontalDynamicLineCount`) for distant targets
- Use update intervals
- Disable gizmos in builds

- Apply layer masks (`obstaclesMask`) to limit linecast checks
- **For Networking (e.g., Photon Fusion):** Move calls to `FixedUpdateNetwork()` for fixed-tick execution, reducing overhead and ensuring sync. Avoid `Update()` in networked builds to prevent desyncs.
- Pool data and profile performance

1.7 Editor Tools

1.7.1 Gizmo Visualization

- Face normals
- Fixed and dynamic sampling lines
- Detection cone (green for visible, red for not visible)
- Visibility status (with distinct colors for fixed, dynamic, and detected lines)

1.7.2 ObserverManager Editor

- Observer/player lists
- Visibility status
- Debug controls

1.8 Performance Tips

1. Tune `dynamicLineSpeed` to balance detection accuracy and performance
2. Adjust line resolution (`verticalFixedLineCount`, `horizontalFixedLineCount`, `verticalDynamicLineCount`, `horizontalDynamicLineCount`)
3. Limit detection frequency
4. Disable gizmos in production
5. Use `obstaclesMask` to filter relevant layers
6. Pool and reuse data
7. Profile and scale

1.9 Extensibility

1.9.1 AI Integration

- Trigger AI behaviors via `VisibilityUnityEventRelay` or `OnVisibilityChanged`
- Combine with pathfinding
- Filter by team/threat/distance

1.9.2 Multiplayer Compatibility

Works with:

- Photon Fusion
- Photon PUN
- Mirror
- Netcode for GameObjects
- Fish-Networking

Server-side logic ensures secure detection. Sync visibility states across clients. **Photon Fusion Specific:** For best results, inherit components from `NetworkBehaviour` and call visibility methods (e.g., `UpdateVisibilityFromAlignedNormals()`) in `FixedUpdateNetwork()` on the host. This aligns with Fusion's tick-based simulation for deterministic linecasts and avoids desyncs. See the adaptation note in `PlayerVisibilityDetector` for an example.

1.9.3 UI and Gameplay Integration

- Use `VisibilityUnityEventRelay` to show/hide indicators, trigger animations, or play sounds
- Log visibility changes for analytics

1.10 Demo Scene Setup

The asset includes a demo scene to help users test the visibility system. The following scripts are included to facilitate interaction and testing but are **not** part of the core functionality:

1.10.1 TimeController

- **Purpose**: Adjusts the game's time scale (`Time.timeScale`) to test the visibility system at different speeds, useful for evaluating `dynamicLineSpeed` behavior.
- **Usage**: Press **Up Arrow** to increase or **Down Arrow** to decrease the time scale. The current value is displayed via a UI `Text` component.
- **Configuration**: Assign a `Text` component to the `timeScale` field in the Inspector. Adjust `incrementSpeed` to control how quickly the time scale changes.

```
[SerializeField] private Text timeScale;
[SerializeField] private float incrementSpeed = 2;
```

1.10.2 PlayerSwitcher

- **Purpose**: Allows switching between multiple players in the demo scene to test visibility from different perspectives.
- **Usage**: Press **Right Arrow** to switch to the next player or **Left Arrow** to switch to the previous player. The camera follows the active player's `PlayerObserver`.

- **Configuration**: Assign an array of `SimplePlayerController` components to the `players` field and the camera's `Transform` to the `cam` field. Each player must have a `PlayerObserver` assigned to its `observer` field.

```
[SerializeField] private SimplePlayerController[] players;
[SerializeField] private Transform cam;
```

1.10.3 SimplePlayerController

- **Purpose**: Provides basic movement and rotation controls for players in the demo scene.
- **Usage**: Use **WASD** keys for movement (forward, backward, right, left) and **Q/E** keys for rotation. The `observer` field links to the GameObject with the `PlayerObserver`.
- **Configuration**: Assign the `PlayerObserver` GameObject to the `observer` field. Adjust `moveSpeed` and `rotationSpeed` for desired control responsiveness.

```
[SerializeField] private float moveSpeed = 5f;
[SerializeField] private float rotationSpeed = 100f;
[SerializeField] public Transform observer;
```

Note: These scripts are included in the demo scene to help users explore the visibility system. They are not required for the asset's core functionality and can be replaced with custom player controllers or camera systems in your project.

1.11 Troubleshooting

1.11.1 Players not detected

Possible Causes:

- FOV angle or view distance too low
- Aspect ratio incorrect
- Sampling lines blocked by incorrect `obstaclesMask` configuration
- Layer masks excluding target
- `playerBox` not properly set as a child GameObject
- Insufficient `dynamicLineSpeed` for fast-moving targets
- `PlayerObserver` GameObject not aligned with the camera's position/rotation

Solutions:

- Increase FOV (`cameraFOVAngle`) and distance (`viewDistance`)
- Set correct `aspectRatio` to match the camera
- Verify `obstaclesMask` includes relevant layers (e.g., `Obstacles`)
- Check layer masks for targets
- Ensure `playerBox` is a child of the GameObject with `PlayerVisibilityDetector`

- Increase `dynamicLineSpeed` for better detection of fast-moving targets
- Ensure the `PlayerObserver` GameObject tracks the camera's position and rotation

1.11.2 Events not firing

Possible Causes:

- `ObserverManager` missing or inactive
- Registration failed
- Visibility state not changing due to low `dynamicLineSpeed`
- `VisibilityUnityEventRelay` not properly configured or missing `PlayerVisibilityDetector`
- `PlayerObserver` not registered due to incorrect hierarchy
- Other scripts executing before `ObserverManager` due to custom execution order

Solutions:

- Ensure `ObserverManager` is active and initialized early (`[DefaultExecutionOrder(-200)]` ensures this)
- Confirm registration of `PlayerVisibilityDetector`, `PlayerObserver`, and `VisibilityUnityEve`
- Increase `dynamicLineSpeed` to improve detection responsiveness
- Verify that `VisibilityUnityEventRelay` is attached to the same GameObject as `PlayerVisibilityDetector` and that `onVisibilityChanged` is configured
- Ensure `PlayerObserver` is a child of the player (if camera is fixed) or tracks the camera's transform
- Check script execution order in Unity to ensure `ObserverManager` runs before dependent scripts
- Use debug logs to track event triggers

1.11.3 Gizmos not showing

Possible Causes:

- Gizmos disabled
- Scene view not focused
- `showGizmos` is false
- `PlayerObserver` not properly aligned with the camera

Solutions:

- Enable Gizmos in toolbar
- Set `showGizmos = true`
- Focus Scene view
- Verify `PlayerObserver` GameObject matches the camera's position and rotation

1.12 Final Notes

The system is modular, extensible, and built for real-world use. Whether you're building a tactical shooter, stealth game, or competitive PvP experience, it gives you the tools to control visibility with precision and security.

1.12.1 Best Practices

- Run checks on server/host
- Tune `dynamicLineSpeed` and line counts for performance
- Use `VisibilityUnityEventRelay` for easy event handling in the Editor
- Use events for gameplay
- Combine with networking frameworks
- Use gizmos during development
- Ensure `playerBox` is a child of the GameObject with `PlayerVisibilityDetector`
- Configure `obstaclesMask` to include relevant obstacle layers
- Place `PlayerObserver` on a GameObject that tracks the player's camera position and rotation
- Rely on `ObserverManager`'s early initialization (`[DefaultExecutionOrder(-200)]`) for stable setup

1.13 FAQ

Does this system work with Photon Fusion or PUN 2? Yes. Fully compatible with both. Visibility logic runs on server/host and syncs across clients.

Can I use this for AI agents? Yes. Observers can represent AI entities, and `VisibilityUnityEventRelay` can trigger AI behaviors.

Is it compatible with URP or HDRP? Yes. Rendering-independent. Uses physics and geometry.

Does it support mobile platforms? Yes. Optimize `dynamicLineSpeed` and line counts for performance.

Can I use this in single-player games? Definitely. Useful for AI, stealth, and scripted events via `VisibilityUnityEventRelay`.

Does it require a specific collider type? No. Works with `BoxCollider`, `CapsuleCollider`, `MeshCollider`. Ensure the collider is a child of the GameObject with `PlayerVisibilityDetector`.

Can I customize the visibility logic? Yes. Modular and extensible. Override sampling, add filters, or use `VisibilityUnityEventRelay` for easy event-driven logic.