PROFESSIONAL EDITION

Anti-Wallhack Visibility System

Server-authoritative visibility validation for competitive Unity multiplayer games.

VERSION	UNITY	PUBLISHER	UPDATED
1.1	6000.0 or newer	Extreme World Studio	August 2026

Contents

1 Introduction

- Overview
- Key Features
- Professional Edition Features

2 Quick Start

- Installation
- Step 1 – Create an ObserverManager
- Step 2 – Configure a Player Observer
- Step 3 – Configure a Target Player
- Step 4 – Add Network Adapter Components
- Step 5 – Run the Scene
- Recommended First Test

3 ObserverManager

- Overview
- Responsibilities
- Automatic Registration
- Visibility State Management
- Update Rate Control
- Recommended Update Rates
- Performance Considerations
- Debugging
- Best Practices

4 PlayerObserver

- Overview
- Responsibilities
- Observer Angle Modes
- Camera Settings
- Broad-Phase Optimization
- Detection Modes
- Observer Prediction
- Visibility Events
- Performance Recommendations
- Best Practices

5 PlayerVisibilityDetector

- Overview
- Responsibilities
- PlayerBox Integration
- Visibility Sampling System
- Fixed Lines
- Dynamic Lines
- Grid Subdivision
- Visibility Stabilization
- Velocity Compensation
- Observer Tracking
- Layer Mask Configuration
- Debug Visualization
- Performance Recommendations
- Common Mistakes
- Best Practices

6 PlayerBox

- Overview
- Purpose
- Configuration
- Velocity Compensation Integration
- Visibility Accuracy
- Recommended Sizing
- Character Variations
- Debug Visualization
- Performance Considerations
- Best Practices

7 Network Adapter Components

- Overview
- Why Network Adapters Exist
- NetworkPlayerObserver
- NetworkPlayerVisibilityDetector
- Adapting to Networking Frameworks
- Server-Authoritative Execution
- Important Update Notice
- Recommended Workflow
- Best Practices

8 Visibility Pipeline

- Overview
- Visibility Pipeline
- Stage 1 – Broad Phase
- Stage 2 – Distance Validation
- Stage 3 – FOV Validation
- Stage 4 – Visibility Sampling
- Stage 5 – Final Visibility Result
- Why The Pipeline Exists
- Performance Benefits
- Best Practices

9 Visibility Sampling System

- Overview
- Why Sampling Is Necessary
- Fixed Lines
- Dynamic Line Speed
- Dynamic Line Spacing
- Sampling Accuracy
- Debug Visualization
- Best Practices

10 Visibility Filtering

- Overview
- Why Visibility Filtering Is Important
- Filtering System
- Example Use Cases
- API Design
- Benefits
- Performance Impact
- Recommended Use Cases
- Future Development
- Best Practices

11 Networking Integration

- Overview
- Supported Frameworks
- Integration Philosophy
- Server-Authoritative Execution
- Synchronization Workflow
- Framework Integration Examples
- Photon Fusion
- Mirror
- Fish-Networking
- Netcode for GameObjects
- Photon PUN 2
- Online Integration Examples
- Common Integration Mistakes
- Best Practices

12 Velocity Compensation

- Overview
- Why Velocity Compensation Exists
- Corner Peeking
- How It Works
- Expanded Size
- Benefits
- Performance Impact
- Configuration Recommendations
- Common Mistakes
- Testing Velocity Compensation
- Using Velocity Compensation As Latency Mitigation
- Best Practices
- Summary

13 Performance Optimization

- Overview
- Understanding Visibility Cost
- Observer Update Rate
- View Distance
- Detection Modes
- Visibility Sampling Density
- Velocity Compensation
- Debug Visualization
- Benchmark Configuration
- Benchmark Results
- Scaling Recommendations
- General Optimization Strategy
- Best Practices

14 PlayerCylinder

- Overview
- Why Cylinder Geometry
- Collider Configuration
- Expansion And Compensation
- Advanced Wall Clamping
- Performance Considerations
- Debug Visualization

15 Advanced Observer Modes

- Overview
- 270 Degree Side Expansion
- Choosing An Observer Mode

16 Network Compensation

- Overview
- FOV Compensation
- Observer Prediction
- Grace Prediction
- Inactive Optimization
- Tuning Recommendations

17 Animation Compensation

- Overview
- Requirements
- How The Pose Factor Is Computed
- Box And Cylinder Behaviour
- Performance Considerations

18 Environment Detection

- Overview
- Shadow Detection
- Reflection Detection
- Environment Detection Performance

19 Benchmarking Tools

- Overview
- Benchmark Scene
- Metrics
- Reading The Results

20 Troubleshooting

- Overview
- Players Are Never Detected
- Players Become Visible Too Early
- Players Become Visible Too Late
- Visibility Flickering
- Visibility Validation Never Runs
- Debug Gizmos Are Not Visible
- Performance Is Lower Than Expected
- Networking Synchronization Issues
- Before Requesting Support
- Additional Help

21 Frequently Asked Questions (FAQ)

- General Questions
- Networking Questions
- Visibility Questions
- Performance Questions
- Integration Questions
- Gameplay Questions
- Support Questions

22 Online Documentation

- Overview
- Official Documentation
- Why Use The Online Documentation?
- Framework Integration Examples
- Performance Benchmarks
- Future Features
- Documentation Update Policy
- Recommended Workflow
- Final Notes

01

Introduction

Overview

Anti-Wallhack Visibility System is a server-authoritative visibility validation solution designed for Unity multiplayer games.

Unlike traditional client-side visibility systems, visibility decisions are performed independently of the rendering state, preventing players from gaining information through wallhack-style cheats or unauthorized visibility modifications.

The system validates whether a target should be considered visible by combining distance checks, field-of-view validation, visibility sampling, and physics-based line-of-sight tests.

This approach allows developers to synchronize only the information that players are legitimately allowed to see while maintaining a fair multiplayer environment.

The system is suitable for:

- Competitive PvP games
- Tactical shooters
- Extraction shooters
- Survival multiplayer games
- Stealth-based games
- AI perception systems
- Server-authoritative multiplayer architectures

Key Features

Server-Authoritative Visibility Validation

Visibility decisions are calculated independently of the client, reducing the possibility of wallhack exploits and unauthorized information access.

Observer-Based Detection

Each observer evaluates visibility from its own perspective using configurable field-of-view and distance settings.

Visibility Sampling System

The system combines fixed and dynamic sampling lines to determine whether a target is genuinely visible.

Broad-Phase Optimization

Early rejection checks reduce the number of expensive visibility calculations, improving scalability in multiplayer environments.

Velocity Compensation

Target visibility can be expanded based on movement speed, helping reduce inconsistencies during fast movement and corner peeking situations.

Multiplayer Ready

The architecture is designed to integrate with popular Unity networking frameworks and server-authoritative multiplayer solutions.

Debug Visualization

Built-in gizmos allow developers to visualize visibility calculations directly inside the Unity Editor.

Professional Edition Features

The Professional Edition includes everything in the Free Edition, plus the systems described in this document:

- `ObserverManager`
- `PlayerObserver`
- `PlayerVisibilityDetector`
- `PlayerBox`
- Fixed Visibility Sampling
- Dynamic Visibility Sampling
- Velocity Compensation
- 180° Observer Mode
- 360° Observer Mode
- Observer Update Rate Control
- Networking Adapter Components
- Debug Visualization Tools
- `PlayerCylinder`
- 270 Degree Observer Mode
- Animation Compensation
- Observer Prediction
- Grace Prediction
- Latency Compensation
- FOV Compensation
- Rotation Compensation
- Shadow Detection
- Reflection Detection
- Runtime Light Registration
- Advanced Wall Clamping

- Extended Benchmark Scene And Analysis Tools

Every system documented here is part of the Professional Edition.

For the most up-to-date information, examples, and framework integrations, refer to the online documentation.

Online Documentation:

> https://gustavobianco277.github.io/Anti-Wallhack-Visibility-System/

02

Quick Start

Installation

Import the Anti-Wallhack Visibility System package into your Unity project.

After importing the package, create a test scene and configure the required components.

Step 1 – Create an `ObserverManager`

Create an empty GameObject in your scene and add the: `ObserverManager` component.

The `ObserverManager` is responsible for:

- Registering observers
- Registering players
- Managing visibility states
- Dispatching visibility events
- Executing visibility updates

Only one `ObserverManager` should exist in a scene.

Step 2 – Configure a Player Observer

Create or select the object that represents the player's view.

In most projects, this is the camera or a transform that follows the camera orientation. Add:

> `PlayerObserver`

Configure:

- Observer Angle
- Camera FOV
- View Distance
- Aspect Ratio
- Detection Mode

The observer determines which targets are evaluated for visibility.

Step 3 – Configure a Target Player

Select the player character that should be detected by other observers. Add:
`PlayerVisibilityDetector` and configure a:

component.

> `PlayerBox`

The `PlayerBox` defines the visibility volume used by the detection system.

Step 4 – Add Network Adapter Components

For multiplayer projects, use the provided adapter components:

These components initialize the visibility system and forward update calls to the core visibility components.

> `NetworkPlayerObserver` `NetworkPlayerVisibilityDetector`

By default, the adapters use Unity's Update loop and are intended as examples that can be adapted to your networking framework.

Step 5 – Run the Scene

When the scene starts:

1. `ObserverManager` initializes.
2. Observers register automatically.
3. Players register automatically.
4. Visibility validation begins.
5. Visibility events are generated as players become visible or hidden.

Recommended First Test

Create two cubes separated by a wall. Assign: **`PlayerObserver`** to one object and:

to the other.

> `PlayerVisibilityDetector` `PlayerBox`

Move the observer around the wall and observe visibility changes through the debug gizmos.

This is the fastest way to verify that the system is functioning correctly before integrating it into a multiplayer project.

03

ObserverManager

Overview

`ObserverManager` is the central coordinator of the visibility system.

Its primary responsibility is to manage observers, players, visibility states, and visibility update execution.

All visibility calculations pass through `ObserverManager`, making it the core component responsible for controlling how often visibility validation occurs and how visibility information is distributed throughout the system.

Only one `ObserverManager` should exist per scene.

Responsibilities

`ObserverManager` is responsible for:

- Registering observers
- Registering players
- Managing visibility states
- Dispatching visibility events
- Controlling update frequency
- Coordinating visibility validation

The component automatically tracks all registered observers and visibility targets during runtime.

Automatic Registration

When initialized, the following components automatically register themselves with `ObserverManager`:

Manual registration is not required under normal circumstances.

> `PlayerObserver` `PlayerVisibilityDetector`

Visibility State Management

`ObserverManager` stores visibility states for every observer-target pair.

Whenever a visibility change occurs, the manager automatically dispatches a visibility event.

Examples:

Visibility events are only triggered when the state actually changes, preventing unnecessary event spam.

> Visible → Hidden Hidden → Visible

Update Rate Control

One of the most important responsibilities of `ObserverManager` is controlling how frequently visibility updates are executed.

This allows developers to balance accuracy and performance depending on the requirements of their game.

Unlimited Mode

Visibility updates execute as frequently as possible. Advantages:

> Call Rate Mode: Unlimited

- Maximum responsiveness
- Immediate visibility updates
- Best accuracy Disadvantages:
- Higher CPU usage
- Less scalable for large player counts Recommended for:
- Small multiplayer games
- Local testing
- Debugging

Limited Mode

Visibility updates execute at a fixed frequency.

> Call Rate Mode: Limited

Advantages:

- Reduced CPU usage
- Better scalability
- Predictable update cost Disadvantages:
- Slightly increased visibility latency Recommended for:
- Dedicated servers
- Medium to large multiplayer games
- Production environments

Recommended Update Rates

Typical values:

Update Rate	Usage
16 TPS	Large-scale games
32 TPS	Recommended default
64 TPS	Competitive environments

The included benchmark tests were executed using: **32 visibility updates per second** which provides a good balance between responsiveness and performance.

Performance Considerations

`ObserverManager` directly affects the scalability of the visibility system. The following factors influence overall performance:

- Number of observers
- Number of players
- Update rate
- View distance
- Detection mode
- Visibility sampling density

For best results:

- Use Limited mode in production environments
- Avoid unnecessarily high update rates
- Tune visibility settings according to gameplay requirements

Debugging

`ObserverManager` provides runtime information that can be used to verify:

- Registered observers
- Registered players
- Visibility state changes
- Update execution

This information is useful when diagnosing visibility issues during development.

Best Practices

- Use a single `ObserverManager` per scene
- Prefer Limited mode for production builds
- Start with 32 updates per second
- Profile performance before increasing update frequency

- Keep visibility settings proportional to the scale of the game

`ObserverManager` acts as the foundation of the visibility system and should always be configured before setting up observers and visibility targets.

04

PlayerObserver

Overview

`PlayerObserver` represents the perspective from which visibility is evaluated.

In most multiplayer games, the observer corresponds to the player's camera, but it can also be used for AI agents, security cameras, spectators, or any entity capable of observing the game world.

The observer is responsible for determining which targets should be evaluated before visibility sampling begins.

This stage acts as the first major filtering layer of the visibility pipeline and helps reduce unnecessary calculations.

Responsibilities

`PlayerObserver` is responsible for:

- Distance validation
- Field-of-view validation
- Broad-phase rejection
- Visibility target selection
- Observer prediction
- Visibility event handling

Only targets that pass the observer validation stage proceed to visibility sampling.

Observer Angle Modes

180° Mode

This mode simulates a traditional camera-based field of view.

> Max Observer Angle: 180°

Only targets located inside the observer's viewing direction are considered for visibility validation.
Advantages:

- Better performance
- More realistic player perception
- Recommended for most games

Recommended for:

- FPS games

- TPS games
- Tactical shooters
- Survival games

360° Mode

This mode removes directional restrictions and allows visibility checks in every direction around the observer.

> Max Observer Angle: 360°

Advantages:

- Useful for AI systems
- Useful for top-down games
- Useful for omnidirectional detection Disadvantages:
- More visibility candidates
- Higher processing cost Recommended for:
- AI perception systems
- Strategy games
- Top-down multiplayer games

Camera Settings

Camera FOV Angle

Defines the base field of view used during visibility calculations. Larger values:

> cameraFOVAngle

- Increase visible area
- Increase visibility candidates

Smaller values:

- Reduce visible area
- Improve performance Typical values:

> 60° - Competitive shooters 75° - General gameplay 90° - Wide field of view

View Distance

Defines the maximum distance at which targets can be considered. Targets outside this range are immediately rejected.

> viewDistance

Reducing view distance can significantly improve performance in large multiplayer environments.

Aspect Format

The selected aspect ratio is used to calculate the horizontal visibility limits of the observer.

4:3

16:9

16:10

21:9

32:9

Using the correct aspect ratio ensures that visibility calculations match the player's actual camera configuration.

Broad-Phase Optimization

Broad Phase FOV Multiplier

Before performing precise visibility validation, the system executes a broad-phase test.

> broadPhaseFOVMultiplier

This expanded FOV allows the observer to quickly reject targets that are clearly outside the viewing area.

Benefits:

- Reduces expensive calculations
- Improves scalability
- Reduces visibility sampling workload Typical values:

> 1.4 - 1.8

Recommended default:

> 1.7

Detection Modes

`PlayerObserver` supports multiple methods for selecting visibility candidates.

Closest Point

The system evaluates visibility using the closest point on the target's visibility volume. Advantages:

> `DetectionMode` : `ClosestPoint`

- Fastest method
- Lowest CPU cost Recommended for:
- Large player counts
- Performance-sensitive environments

Bounds Corners

The system evaluates multiple points around the target bounds.

> `DetectionMode` : `BoundsCorners`

Advantages:

- Higher precision
- Better edge visibility handling Disadvantages:
- More calculations Recommended for:
- Competitive games
- High precision environments

Observer Prediction

Movement-based observer prediction is available in both editions. The Professional Edition extends it with latency-aware prediction and grace prediction, described in the Network Compensation chapter.

This system helps compensate for rapid player movement and reduces inconsistencies during corner peeking situations.

Velocity Prediction

The observer can temporarily offset its virtual position based on movement velocity. Benefits:

- Smoother visibility transitions
- Better handling of fast movement
- Reduced corner-peeking inconsistencies

Prediction Axis Modes

Supported modes:

> Lateral Frontal
> `LateralAndFrontal`

These modes determine which movement directions contribute to prediction.

Prediction Weights

The following settings control prediction influence:

Increasing these values improves movement compensation but may slightly increase visibility permissiveness.

> lateralWeight frontalWeight velocityPredictionScale maxPredictionDistance

Visibility Events

`PlayerObserver` automatically receives visibility updates from `ObserverManager`. Whenever visibility changes: **Visible → Hidden Hidden → Visible** the observer updates its current visibility state. This information can be used for:

- Rendering control
- Network synchronization
- Gameplay systems
- Debugging

Performance Recommendations

For most multiplayer games:

> 180° Observer Mode `BoundsCorners`
>
> 32 Updates Per Second 16:9 Aspect Ratio

provides an excellent balance between accuracy and performance. Large-scale games should consider:

> 180° Observer Mode `ClosestPoint`
>
> Reduced View Distance Limited Update Rate

to maximize scalability.

Best Practices

- Match the observer to the player's camera orientation
- Use 180° mode whenever possible
- Tune view distance according to map size
- Use `ClosestPoint` when prioritizing performance
- Use `BoundsCorners` when prioritizing accuracy
- Profile performance before increasing visibility ranges

`PlayerObserver` serves as the first filtering stage of the visibility system and plays a critical role in balancing visibility accuracy and server performance.

05

PlayerVisibilityDetector

Overview

`PlayerVisibilityDetector` represents a visibility target within the Anti-Wallhack Visibility System.

While `PlayerObserver` determines which targets should be evaluated, `PlayerVisibilityDetector` is responsible for performing the actual visibility validation through physics-based sampling and line-of-sight checks.

This component is attached to every player or entity that can be detected by observers.

The detector works together with `PlayerBox` to generate visibility samples and determine whether a target should be considered visible.

Responsibilities

`PlayerVisibilityDetector` is responsible for:

- Visibility sampling
- Line-of-sight validation
- Fixed line generation
- Dynamic line generation
- Visibility stabilization
- Velocity compensation
- Visibility state tracking
- Observer management

This component performs the most detailed stage of the visibility pipeline.

`PlayerBox` Integration

`PlayerVisibilityDetector` requires a `PlayerBox` component.

`PlayerBox` defines the visibility volume used during visibility sampling. The detector uses the `PlayerBox` dimensions to:

- Generate visibility samples
- Calculate visibility points
- Perform line-of-sight validation
- Apply velocity compensation

For best results, the `PlayerBox` should closely match the visible shape of the character.

Visibility Sampling System

The detector uses two sampling methods:

These systems work together to improve visibility accuracy while maintaining performance.

> Fixed Lines Dynamic Lines

Fixed Lines

Fixed lines are static visibility samples distributed across the visibility volume.

They provide consistent visibility validation and ensure that important areas of the target are always checked.

Advantages:

- Stable results
- Consistent visibility validation
- Low computational cost Disadvantages:
- May miss small visibility openings
- Less adaptive than dynamic sampling

Fixed Line Configuration

Increasing these values:

> `verticalFixedLineCount` `horizontalFixedLineCount`

- Improves accuracy
- Increases CPU usage Reducing these values:
- Improves performance
- Reduces visibility precision

Dynamic Lines

Dynamic lines move across the visibility volume over time.

This allows the system to gradually sample areas that fixed lines may not cover. Advantages:

- Improved visibility coverage
- Better corner detection
- Improved detection through narrow openings Disadvantages:
- Slightly higher computational cost

Dynamic Line Configuration

> `verticalDynamicLineCount` `horizontalDynamicLineCount` `dynamicLineSpacing` `dynamicLineSpeed`

Dynamic Line Speed

Controls how quickly dynamic sampling moves across the visibility volume. Higher values:

> `dynamicLineSpeed`

- Faster coverage
- Improved responsiveness Lower values:
- Reduced CPU usage
- Slower visibility updates Recommended starting values:

> 5 - 15

depending on game requirements.

Grid Subdivision

The visibility volume is subdivided into rows and columns.

This subdivision determines how visibility samples are distributed. Configuration:

> horizontalColumnCount verticalRowCount

Higher values:

- More visibility samples
- Better accuracy Lower values:
- Better performance
- Fewer visibility checks

Visibility Stabilization

State Stabilize Time

Visibility changes near walls or corners can sometimes fluctuate rapidly.

> stateStabilizeTime

State stabilization introduces a small delay before visibility changes are confirmed. Benefits:

- Reduces flickering
- Improves visual consistency

- Creates smoother visibility transitions Typical values:

> 0.05 - 0.25 seconds

Velocity Compensation

Velocity-based visibility compensation is available in both editions. The Professional Edition adds latency, rotation and animation compensation on top of it.

Fast-moving players may partially expose themselves before their visibility volume fully enters the observer's field of view.

Velocity compensation expands the visibility volume based on movement speed. Benefits:

- Fairer visibility validation
- Improved corner peeking behavior
- Better synchronization between movement and visibility

Expanded Visibility Volume

As movement speed increases:

This allows fast-moving targets to be detected more consistently.

> Movement Speed ↑
>
> ↓
>
> Visibility Volume Expansion ↑

The expansion amount can be configured through `PlayerBox` settings and movement compensation parameters.

Observer Tracking

`PlayerVisibilityDetector` maintains a list of active observers.

Observers are automatically added and removed by the visibility system. This allows the detector to:

- Track visibility state per observer
- Perform targeted visibility validation
- Avoid unnecessary calculations

Layer Mask Configuration

Obstacle Layers

Obstacle layers determine which objects block visibility. Examples:

> Walls Buildings Terrain
>
> Static Geometry

Only layers included in the obstacle mask participate in line-of-sight validation. Incorrect layer configuration is one of the most common causes of visibility issues.

Debug Visualization

`PlayerVisibilityDetector` provides extensive runtime visualization tools. Available debug views include:

> Fixed Lines Dynamic Lines Detected Lines Blocked Lines Visibility Volume Face Normals

These visualizations help diagnose:

- Visibility issues
- Incorrect collider sizing
- Sampling density problems
- Layer mask configuration errors

Performance Recommendations

For most projects:

provides an excellent balance between responsiveness and performance. For large multiplayer environments:

> Moderate Fixed Lines Moderate Dynamic Lines
>
> 32 Updates Per Second Velocity Compensation Enabled

- Reduce sampling density
- Lower update frequency
- Reduce view distance
- Profile before increasing line counts

Common Mistakes

Excessive Sampling Density

Increasing every sampling setting simultaneously can significantly increase CPU usage without noticeable gameplay improvements.

Incorrect `PlayerBox` Size

Boxes that are too large:

Boxes that are too small:

> Early Detection

> Delayed Detection

Incorrect Layer Masks

If obstacle layers are not configured correctly:

may occur.

> False Positives False Negatives
>
> Visibility Inconsistencies

Best Practices

- Start with moderate sampling values
- Use velocity compensation for multiplayer games
- Keep obstacle masks clean and organized
- Profile performance before increasing line density
- Use debug visualization during integration

`PlayerVisibilityDetector` is the component responsible for performing the final visibility validation and is one of the most important parts of the Anti-Wallhack Visibility System.

06

PlayerBox

Overview

`PlayerBox` defines the visibility volume used by the Anti-Wallhack Visibility System.

Unlike traditional gameplay colliders, `PlayerBox` is not intended to represent physical collision. Instead, it defines the area used by the visibility system when determining whether a player should be considered visible.

All visibility sampling performed by `PlayerVisibilityDetector` is based on the dimensions provided by `PlayerBox`.

Proper `PlayerBox` configuration is one of the most important factors affecting visibility accuracy.

Purpose

`PlayerBox` serves as the geometric representation of a visibility target. The component is responsible for:

- Defining the visibility volume
- Providing sampling boundaries
- Supporting velocity compensation
- Generating visibility points
- Improving visibility consistency

The visibility system uses this volume to determine where visibility samples should be placed.

Configuration

Center

Defines the local offset of the visibility volume.

> Center

This allows the visibility volume to be aligned with the visual representation of the character. Examples:

- Aligning the box with the torso
- Adjusting for custom character pivots
- Correcting imported model offsets

Size

Defines the base dimensions of the visibility volume.

> Size

This represents the standard visibility area when no compensation is applied. Typical examples:

FPS Character

TPS Character

> Width: 0.6
>
> Height: 1.8
>
> Depth: 0.6

Actual values should be adjusted according to the visual size of the character.

> Width: 0.8
>
> Height: 1.8
>
> Depth: 0.8

Expanded Size

Defines the maximum expansion applied to the visibility volume. This value is used by the velocity compensation system.

> Expanded Size

As movement speed increases, the visibility volume can temporarily expand toward the configured limits.

Benefits:

- Improved fairness
- Better corner peeking behavior
- Reduced visibility inconsistencies

Velocity Compensation Integration

Velocity-based expansion is available in both editions.

When enabled, the system adjusts the visibility volume according to player movement speed. Example:

Standing Still

↓

Base Size

Running

↓

Expanded Size

This allows fast-moving targets to become visible more consistently during rapid movement.

Visibility Accuracy

`PlayerBox` directly affects visibility behavior.

Oversized Boxes

If the visibility volume is significantly larger than the visual character:

This may cause players to become visible sooner than expected.

> Earlier Detection Increased Visibility Area Reduced Precision

Undersized Boxes

If the visibility volume is significantly smaller than the visual character:

This may allow players to expose parts of their character before visibility is properly detected.

> Delayed Detection Reduced Visibility Area Potential Exposure Issues

Recommended Sizing

A good `PlayerBox` should:

- Cover the main body of the character
- Match the visual silhouette
- Exclude unnecessary empty space
- Remain consistent across all player characters Avoid using extremely aggressive dimensions.

Moderate sizing typically produces the most natural results.

Character Variations

Different character types may require different `PlayerBox` configurations. Examples:

Standard Human Character

Heavy Character

> Medium Width Standard Height

Small Character

> Increased Width Standard Height

The visibility volume should always reflect the intended gameplay balance.

> Reduced Width Reduced Height

Debug Visualization

`PlayerBox` integrates with the visibility debugging system. When gizmos are enabled, developers can visualize:

This allows rapid adjustment and validation during development.

> Visibility Volume Expanded Volume Sampling Area Detection Boundaries

Performance Considerations

`PlayerBox` itself has a negligible performance cost. However, larger visibility volumes may:

- Increase visibility candidates
- Increase successful visibility checks
- Increase network synchronization frequency For best results:
- Use realistic dimensions
- Avoid excessive expansion values
- Profile changes in multiplayer environments

Best Practices

- Match the box to the visible body shape
- Use moderate expansion values
- Validate sizing with debug gizmos
- Test corner peeking scenarios
- Keep dimensions consistent across player characters

`PlayerBox` serves as the foundation of the visibility system and plays a major role in determining the accuracy and fairness of visibility validation. The Professional Edition adds `PlayerCylinder` as a more precise alternative.

07

Network Adapter Components

Overview

Both editions include default networking adapter components designed to simplify integration between the Anti-Wallhack Visibility System and multiplayer frameworks.

These components act as a bridge between the visibility system and the networking solution used by the project.

Their purpose is to initialize the visibility system and forward update execution to the core visibility components.

The included adapters are intended as reference implementations and can be customized to fit the requirements of a specific networking framework.

Why Network Adapters Exist

The core visibility system is framework-agnostic. Components such as: `ObserverManager` `PlayerObserver` `PlayerVisibilityDetector` `PlayerBox` do not depend on any specific networking solution.

Instead, the network adapters are responsible for connecting those components to the update loop used by the multiplayer framework.

This design allows the visibility system to remain flexible and compatible with multiple networking solutions.

`NetworkPlayerObserver`

Purpose

`NetworkPlayerObserver` acts as the integration layer for `PlayerObserver`.

It initializes the observer and forwards execution calls to the visibility system.

The default implementation is intentionally simple and can be adapted to any networking framework.

Responsibilities

`NetworkPlayerObserver` is responsible for:

- Initializing `PlayerObserver`
- Registering visibility event callbacks
- Forwarding observer updates
- Maintaining observer visibility state
- Providing a starting point for networking integration

Initialization

During startup:

The adapter automatically initializes the observer and prepares it for visibility validation.

Awake()

↓

InitializeSystem()

↓

Observer Registration

Server Tick Execution

The adapter forwards update execution to:

Example:

> `PlayerObserver`.ServerTick(time)

The included implementation uses Unity's Update method for demonstration purposes.

private void Update()

{

var time = Time.time; playerObserver.ServerTick(time);

}

Visibility Events

`NetworkPlayerObserver` automatically subscribes to:

and updates: `ObserverManager.OnVisibilityChanged` whenever the visibility state changes.

> playerObserver.IsVisible

This allows visibility information to be consumed by rendering systems, gameplay systems, or networking logic.

`NetworkPlayerVisibilityDetector`

Purpose

`NetworkPlayerVisibilityDetector` acts as the integration layer for `PlayerVisibilityDetector`. It initializes the detector and forwards visibility execution calls.

Responsibilities

`NetworkPlayerVisibilityDetector` is responsible for:

- Initializing `PlayerVisibilityDetector`
- Forwarding visibility updates
- Providing a networking integration example
- Connecting the detector to the game update loop

Initialization

During startup:

The detector is automatically prepared for visibility validation.

Awake()

↓

InitializeSystem()

↓

Player Registration

Server Tick Execution

The adapter forwards execution to:

Example:

> `PlayerVisibilityDetector`.ServerTick(time)

The included implementation is intended as a starting point and should be adapted for multiplayer environments.

private void Update()

{

var time = Time.time; playerVisibilityDetector.ServerTick(time);

}

Adapting to Networking Frameworks

The default adapters use Unity's Update method because they must remain framework-independent.

However, multiplayer projects should execute visibility validation using the networking framework's server-authoritative update loop.

Examples:

The exact implementation depends on the framework being used. Additional integration examples are available in the online documentation.

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

Server-Authoritative Execution

For multiplayer games, visibility validation should always execute on the server or host authority. Recommended flow:

This ensures that visibility decisions remain authoritative and resistant to client-side manipulation.

Network Tick

↓

ServerTick()

↓

Visibility Validation

↓

Visibility Result

↓

Network Synchronization

Important Update Notice

The included adapter components are intended as reference implementations.

Most projects will eventually customize them to match their networking architecture.

After adapting the components to your project, it is recommended to maintain your own customized versions.

Reason:

If the adapters have been heavily modified, importing a newer version of the asset may overwrite those customizations.

> Future asset updates may replace the default adapter scripts.

Recommended Workflow

This approach minimizes maintenance effort and prevents accidental loss of integration code.

1. Import the asset
2. Configure the default adapters
3. Adapt them to your networking framework
4. Move customized versions into your project architecture

5. Keep your project-specific implementation under source control

Best Practices

- Execute visibility validation on the server or host
- Use the networking framework's fixed simulation tick whenever possible
- Keep adapters lightweight
- Avoid placing gameplay logic inside the adapters
- Treat the adapters as integration layers rather than gameplay systems
- Maintain customized versions after integration

The Network Adapter Components provide a simple and flexible foundation for connecting the Anti-Wallhack Visibility System to virtually any Unity multiplayer framework.

08

Visibility Pipeline

Overview

The Anti-Wallhack Visibility System uses a multi-stage visibility pipeline designed to maximize performance while maintaining accurate visibility validation.

Instead of immediately performing expensive line-of-sight calculations against every target, the system progressively filters candidates through multiple validation stages.

This approach allows large numbers of players to be processed efficiently while maintaining reliable visibility results.

The visibility pipeline is one of the most important concepts in the system and understanding it helps developers tune performance and accuracy more effectively.

Visibility Pipeline

Each stage progressively reduces the number of candidates that require further processing.

Broad Phase

↓

Distance Validation

↓

FOV Validation

↓

Visibility Sampling

↓

Final Visibility Result

Stage 1 – Broad Phase

Purpose

The Broad Phase is the first optimization layer.

Its goal is to quickly eliminate targets that are obviously outside the observer's area of interest.

Before any detailed calculations occur, the system evaluates whether a target falls within an expanded detection region.

This stage uses:

> broadPhaseFOVMultiplier

Benefits:

- Very low computational cost
- Quickly rejects invalid targets
- Reduces workload for later stages

Without Broad Phase filtering, every target would proceed to expensive visibility calculations.

Stage 2 – Distance Validation

Purpose

The second stage validates distance.

Targets beyond the observer's configured visibility range are immediately discarded. The system compares: **Distance To Target** against:

Benefits:

> viewDistance

- Extremely fast calculation
- Large reduction in visibility candidates
- Important for large multiplayer maps

Distance validation is one of the most effective performance optimizations available.

Stage 3 – FOV Validation

Purpose

Targets that pass Broad Phase and Distance validation proceed to Field of View validation. At this stage the system evaluates: **Observer Direction** against:

using:

> Target Position

Only targets located inside the observer's viewing region continue through the pipeline.

> cameraFOVAngle aspectFormat maxObserverAngle

180° Observer

In 180° mode: **Targets Behind The Observer** are rejected immediately.

This significantly reduces visibility workload and is recommended for most multiplayer games.

360° Observer

In 360° mode: **All Directions** are considered valid.

This increases flexibility but may increase visibility processing costs.

Stage 4 – Visibility Sampling

Purpose

After passing all previous stages, the target proceeds to visibility sampling. This is where the actual visibility validation occurs.

The system generates visibility samples using: **Fixed Lines Dynamic Lines** generated from:

The detector then performs physics-based visibility checks.

> `PlayerBox`

Fixed Lines

Provide stable visibility coverage. Benefits:

- Consistent results
- Predictable behavior
- Low overhead

Dynamic Lines

Provide moving visibility samples. Benefits:

- Better visibility coverage
- Improved corner detection
- Improved detection through small openings

Velocity Compensation

If enabled, the visibility volume may expand according to target movement speed. Benefits:

- Fairer visibility behavior
- Better handling of fast-moving targets
- Reduced corner-peeking inconsistencies

Stage 5 – Final Visibility Result

Purpose

The final stage determines whether the target should be considered visible. If visibility sampling succeeds: **Visible** is returned. Otherwise:

> Hidden

is returned.

The visibility state is then sent to: `ObserverManager` which updates visibility information and dispatches visibility events when necessary.

Why The Pipeline Exists

A naive implementation would perform line-of-sight checks against every target. Example:

> 100 Players
> ↓
> 100 Visibility Calculations

The Anti-Wallhack Visibility System instead performs progressive filtering:

100 Players

↓

Broad Phase

↓

35 Players

↓

Distance Validation

↓

18 Players

↓

FOV Validation

↓

7 Players

↓

Visibility Sampling

This dramatically reduces the number of expensive physics checks.

Performance Benefits

The visibility pipeline provides:

- Better scalability

- Lower CPU usage
- More predictable performance
- Improved multiplayer support

Most visibility candidates are eliminated before visibility sampling is ever performed.

This is one of the primary reasons the system remains efficient even in multiplayer environments with large numbers of players.

Best Practices

- Use 180° observer mode whenever possible
- Tune view distance according to map size
- Keep Broad Phase enabled
- Avoid excessive visibility ranges
- Use appropriate sampling density
- Profile performance before increasing accuracy settings

The Visibility Pipeline forms the foundation of the Anti-Wallhack Visibility System and is responsible for balancing accuracy, fairness, and performance.

09

Visibility Sampling System

Overview

The Visibility Sampling System is the core mechanism responsible for determining whether a target should be considered visible.

After a target passes the observer validation stages, the system performs a series of visibility samples distributed across the target's visibility volume.

Unlike simple raycast-based approaches that rely on a single line-of-sight test, the Anti-Wallhack Visibility System uses multiple samples to produce more reliable visibility results.

This approach reduces false negatives while maintaining scalability.

Why Sampling Is Necessary

A single raycast can produce unreliable visibility results. Examples:

Small Openings Corners Railings Windows Partial Exposure

A single visibility point may incorrectly classify a target as hidden even when part of the target is actually visible.

The Visibility Sampling System solves this problem by evaluating multiple points across the visibility volume.

Sampling Sources

The visibility system uses two sampling systems:

Fixed Lines Dynamic Lines

These systems complement each other and work together to improve visibility accuracy.

Fixed Lines

Purpose

Fixed Lines provide permanent visibility samples distributed across the visibility volume. They represent the baseline visibility coverage of the system.

Fixed lines never move and remain stable throughout gameplay.

Advantages

Fixed Lines provide:

- Consistent visibility checks
- Predictable behavior
- Stable visibility validation

- Low computational overhead

Because they remain static, they are highly efficient and easy to debug.

Configuration

The number of fixed samples is controlled by:

`verticalFixedLineCount` `horizontalFixedLineCount`

Increasing these values:

More Samples

↓

Higher Accuracy

↓

Higher CPU Cost

Reducing these values:

Fewer Samples ↓ Lower CPU Cost ↓ Lower Accuracy

> Dynamic Lines Purpose

Dynamic Lines continuously move across the visibility volume.

Their objective is to sample regions that Fixed Lines may not permanently cover.

This allows the system to detect visibility opportunities that occur between fixed samples.

Advantages Dynamic Lines provide:

- Better coverage over time
- Improved corner visibility detection
- Improved detection through narrow openings
- Increased sampling flexibility

Dynamic Movement Unlike Fixed Lines:

Static Position

Dynamic Lines:

Move Across The Visibility Volume

This movement gradually scans additional visibility regions without permanently increasing sample density.

Configuration

Dynamic behavior is controlled by:

`verticalDynamicLineCount` `horizontalDynamicLineCount` `dynamicLineSpacing` `dynamicLineSpeed`

Dynamic Line Speed

`dynamicLineSpeed`

`dynamicLineSpeed`

Defines how quickly Dynamic Lines move across the visibility volume. Higher values:

- Faster visibility coverage
- Improved responsiveness
- More frequent sampling changes Lower values:
- Slower coverage
- Reduced processing variation
- More stable sampling patterns

Recommended Values

Most projects perform well using:

5 - 15

Developers should profile and adjust according to gameplay requirements.

Dynamic Line Spacing

`dynamicLineSpacing`

`dynamicLineSpacing`

> Controls the distance between Dynamic Lines. Smaller spacing:
>
> More Coverage Higher Accuracy Higher Cost

Larger spacing:

Less Coverage Lower Cost

> Grid Subdivision
>
> The visibility volume is divided into a configurable grid. This grid determines where visibility samples are generated. The subdivision is controlled by:
>
> horizontalColumnCount verticalRowCount

Higher Density

Higher subdivision values produce:

- More visibility samples
- Better visibility precision
- Increased CPU usage Recommended for:

Competitive Multiplayer Games

Lower Density

Lower subdivision values produce:

- Fewer visibility samples
- Reduced CPU usage
- Faster processing Recommended for:

Large-Scale Multiplayer Games

Sampling Accuracy

The overall accuracy of the visibility system depends on the combination of:

Fixed Lines Dynamic Lines Grid Density `PlayerBox` Size

Increasing every setting simultaneously is rarely necessary. A balanced configuration usually provides the best results.

Interaction With Velocity Compensation

Velocity Compensation works together with visibility sampling. When movement speed increases:

Player Speed ↑

↓

Visibility Volume Expansion ↑

↓

Sampling Coverage ↑

This allows visibility checks to adapt naturally to movement.

Debug Visualization

The Visibility Sampling System includes extensive debugging support. Available debug views include:

Fixed Lines Dynamic Lines Detected Lines Blocked Lines Visibility Volume

Debug visualization is strongly recommended during initial integration and tuning.

Performance vs Accuracy

One of the most important tuning decisions is balancing accuracy against processing cost.

Accuracy-Oriented Configuration

Example:

High Fixed Line Count High Dynamic Line Count Dense Grid

Results:

- Better visibility precision

- Better corner detection
- Higher CPU cost

Performance-Oriented Configuration

Example:

Lower Fixed Line Count Lower Dynamic Line Count Reduced Grid Density

Results:

- Lower CPU usage
- Better scalability
- Slightly reduced precision

Recommended Starting Configuration For most multiplayer games:

Moderate Fixed Lines Moderate Dynamic Lines

32 Updates Per Second Velocity Compensation Enabled

provides an excellent balance between responsiveness and performance.

Developers should start with conservative values and increase density only when necessary.

Best Practices

- Begin with moderate sampling density
- Profile before increasing line counts
- Use Dynamic Lines to improve coverage instead of excessive Fixed Lines
- Validate changes using debug gizmos
- Test visibility around corners and small openings
- Balance performance and accuracy according to game scale

The Visibility Sampling System is the final stage responsible for determining visibility and plays a central role in the accuracy and fairness of the Anti-Wallhack Visibility System.

10

Visibility Filtering

Overview

Many multiplayer games require certain entities to remain synchronized regardless of visibility. Examples include:

> Teammates Friendly NPCs Squad Members Pets
>
> Minimap Targets Revive Targets

In these situations, visibility validation should not determine whether those entities are synchronized to the client.

Visibility Filtering allows developers to selectively exclude targets from visibility validation while still benefiting from the Anti-Wallhack Visibility System for all other entities.

Why Visibility Filtering Is Important

Consider a team-based game. Example:

Player A

↓

Blue Team

Player B

↓

Blue Team

If Player B moves behind a wall:

> Wall
>
> ↓
>
> Player B

the current visibility system may determine: **Player B = Hidden** and synchronization may stop.

However, many games still need to know the location of teammates for:

- Squad indicators
- Team markers
- Nameplates
- Revive systems
- Minimap displays

- Tactical communication

In these cases, teammates should bypass visibility validation entirely.

Filtering System

Visibility Filtering lets developers decide whether a target should participate in visibility validation.

Conceptually:

If validation is skipped, the target remains synchronized normally.

Target

↓

Visibility Filter

↓

Validate Visibility?

↓

Yes / No

Example Use Cases

Team-Based Games

Allow teammates to remain synchronized at all times. Example:

> Blue Team
>
> Always Visible To Blue Team

while enemy players continue using visibility validation.

Friendly NPCs

Friendly AI characters may always remain visible to the player. Example:

> Friendly NPC
>
> ↓
>
> Ignore Visibility Validation

Pets And Companions

Companion entities may remain synchronized regardless of visibility state. Example:

> Pet
>
> ↓
>
> Always Visible

Spectator Systems

Spectator cameras may require unrestricted visibility. Example:

> Spectator
>
> ↓
>
> Ignore Visibility Restrictions

API Design

The exact implementation may evolve over time.

One possible approach is a visibility filter callback:

The callback would determine whether visibility validation should proceed.

> bool ShouldValidate(`PlayerObserver` observer,
>
> `PlayerVisibilityDetector` target);

Example

ShouldValidate()

↓

True

↓

Run Visibility Validation

Skipped Validation

ShouldValidate()

↓

False

↓

Keep Target Synchronized

Benefits

Visibility Filtering provides:

- Better support for team-based games

- Easier integration with existing gameplay systems
- Reduced custom networking code
- Improved flexibility
- Better compatibility with minimap and squad systems

Performance Impact

Visibility Filtering may slightly improve performance because filtered targets can bypass expensive visibility calculations.

Example:

The impact depends on the number of filtered entities.

Teammates

↓

Skip Validation

↓

Reduced Visibility Workload

Recommended Use Cases

Visibility Filtering is particularly useful for:

> Competitive Team Shooters Battle Royale Games Extraction Shooters
>
> Co-op Games Tactical Games Squad-Based Games

Future Development

Visibility Filtering is available in both editions.

The goal is to provide a lightweight and framework-agnostic solution that can be integrated into any multiplayer architecture.

The final implementation may include:

- Team filtering
- Friendly NPC filtering
- Custom entity filtering
- User-defined visibility rules

Additional details and implementation examples will be published in the online documentation as the feature evolves.

Best Practices

When available:

- Filter teammates whenever gameplay requires permanent team awareness
- Continue validating enemy players through the visibility system
- Keep filtering rules simple and predictable
- Avoid excessive custom filtering logic

Visibility Filtering is intended to improve flexibility while maintaining the security benefits of server-authoritative visibility validation.

11

Networking Integration

Overview

The Anti-Wallhack Visibility System is designed to be networking-framework agnostic.

The visibility system itself does not depend on any specific multiplayer solution and can be integrated into a wide range of server-authoritative architectures.

This approach allows developers to use the visibility system with their preferred networking framework while maintaining a consistent visibility validation workflow.

Supported Frameworks

The system has been designed to integrate with:

Additional frameworks can also be supported through custom integration adapters.

> Photon Fusion Photon PUN 2 Mirror
>
> Fish-Networking
>
> Netcode for GameObjects (NGO)

Integration Philosophy

The visibility system should not control the networking framework.

Instead, the networking framework should control when visibility updates are executed. Recommended architecture:

Networking Framework

↓

Network Adapter Components

↓

ServerTick()

↓

Visibility Validation

↓

Visibility Result

↓

Network Synchronization

This keeps the visibility system independent from networking-specific logic.

Server-Authoritative Execution

For maximum security, visibility validation should always execute on the authoritative side of the game. Depending on the architecture, this may be:

> Dedicated Server Host
>
> Server Authority Instance

Visibility calculations should never rely on client-side authority.

Synchronization Workflow

A typical visibility workflow looks like:

Only entities considered visible should be synchronized to the observing player.

Observer

↓

Visibility Validation

↓

Visible?

↓

Yes / No

↓

Network Synchronization

This reduces unnecessary network traffic and helps prevent information leaks that could be exploited by wallhack tools.

Framework Integration Examples

Both editions include generic adapter components:

These components provide a simple starting point and demonstrate the expected integration pattern. The included implementations use Unity's Update loop for simplicity.

> `NetworkPlayerObserver` `NetworkPlayerVisibilityDetector`

Production multiplayer projects should move visibility execution to the framework's server tick.

Photon Fusion

Photon Fusion is one of the recommended networking solutions for competitive multiplayer games. Typical integration:

```
FixedUpdateNetwork()
↓
ServerTick()
```

Recommended authority:

Visibility validation should only execute on the authoritative simulation.

```
State Authority
```

Mirror

Mirror integrations typically execute visibility updates inside the server update cycle. Recommended flow:

Server

↓

ServerTick()

↓

Visibility Validation

Clients should receive only the final visibility result.

Fish-Networking

Fish-Networking follows a similar server-authoritative model.

Visibility validation should execute inside the framework's network tick. Recommended flow:

Server Tick

↓

ServerTick()

↓

Visibility Result

Netcode for GameObjects

Netcode for GameObjects supports server-authoritative architectures and can integrate directly with the visibility system.

Recommended flow:

The visibility system should execute before replication decisions are made.

Server

↓

Visibility Validation

↓

RPC / Synchronization

Photon PUN 2

Photon PUN projects often use a host-authoritative architecture.

In these cases, visibility validation should execute only on the authoritative host. Recommended flow:

Master Client

↓

Visibility Validation

↓

Synchronization

Online Integration Examples

Because networking frameworks evolve frequently, complete integration examples are maintained in the online documentation.

The online documentation contains:

- Updated integration examples
- Framework-specific setup instructions
- Best practices
- Troubleshooting information
- Future framework support

The online documentation should always be considered the primary source for networking integration guidance.

Common Integration Mistakes

Executing Visibility Validation On Every Client

Incorrect:

> Client A Client B Client C
> ↓
> Visibility Validation

This creates inconsistent visibility states and weakens security.

Correct:

Server

↓

Visibility Validation

↓

Clients

Running Visibility Validation Before Initialization

Incorrect:

Correct:

> ServerTick()
> ↓
> InitializeSystem()

All visibility components must be initialized before updates begin.

> InitializeSystem()
> ↓
> ServerTick()

Mixing Gameplay Logic With Adapter Components

The adapter components should remain lightweight. Avoid placing:

- Gameplay rules
- Combat systems
- Inventory systems
- Team management systems inside the network adapters.

Their primary responsibility is visibility integration.

Best Practices

- Execute visibility validation only on the authoritative side
- Use the framework's fixed network tick whenever possible

- Keep adapters lightweight
- Separate visibility logic from gameplay logic
- Follow framework-specific synchronization recommendations
- Refer to the online documentation for updated integration examples

The networking integration layer allows the Anti-Wallhack Visibility System to remain flexible, scalable, and compatible with a wide range of multiplayer architectures.

12

Velocity Compensation

Overview

Velocity Compensation improves visibility validation for moving targets and is available in both editions.

In traditional visibility systems, visibility calculations are often performed using a fixed visibility volume regardless of movement speed.

This can create situations where a fast-moving player becomes partially exposed before the visibility system properly detects them.

Velocity Compensation addresses this issue by dynamically expanding the visibility volume based on movement speed.

The result is a more consistent and fair visibility experience, especially in fast-paced multiplayer games.

Why Velocity Compensation Exists

Consider the following situation:

Without compensation, a fast-moving player may visually expose themselves before their visibility volume is sampled effectively.

Player Running

↓

Approaches Corner

↓

Partially Exposed

↓

Visibility Check

This can create situations where:

- The moving player can see first
- Detection occurs later than expected
- Visibility feels inconsistent

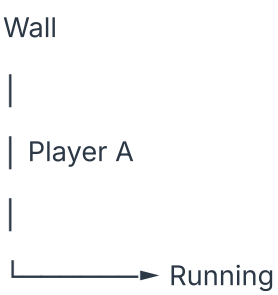
Velocity Compensation reduces these issues by anticipating movement exposure.

Corner Peeking

One of the most common visibility problems in multiplayer games occurs during corner peeking.

Example:

As Player A moves around the corner, part of the character becomes visible before the center of the visibility volume fully exits cover.



Without compensation:

With Velocity Compensation:

> Visual Exposure
> ↓
> Visibility Detection

This creates more natural visibility behavior.

Movement Prediction

↓

Expanded Visibility Volume

↓

Earlier Detection

How It Works

Velocity Compensation expands the visibility volume according to movement speed. Conceptually:

> Movement Speed ↑
> ↓
> Visibility Volume Expansion ↑

When a player is standing still: **Base Size** is used.

When movement speed increases: **Expanded Size** gradually becomes active.

This expansion allows visibility samples to cover the player's expected exposure area more effectively.

Expanded Size

Velocity Compensation relies on the `PlayerBox` configuration. Two dimensions are used:

> Size Expanded Size

Size

Represents the default visibility volume. Example:

> Standing Player

Expanded Size

Represents the maximum visibility volume that may be used during movement. Example:

> Running Player

The system automatically interpolates between both volumes according to movement speed.

Benefits

Velocity Compensation provides several gameplay advantages:

Fairer Visibility

Players become visible closer to the moment they visually expose themselves.

Improved Corner Behavior

Visibility transitions become smoother when moving around obstacles.

Better Multiplayer Experience

High-speed movement becomes more predictable for both players.

Reduced Visibility Inconsistencies

The visibility system reacts more naturally to rapid movement.

Performance Impact

Velocity Compensation has a very low performance cost.

The system does not perform additional raycasts or visibility passes. Instead, it adjusts the visibility volume already used by the sampling system. Benefits:

> Low CPU Cost Low Memory Cost
>
> High Gameplay Benefit

Because of this, Velocity Compensation is recommended for most multiplayer games.

Configuration Recommendations

Conservative Configuration

Recommended for:

> Tactical Shooters Competitive Games

Characteristics:

- Small expansion values
- Higher precision
- Lower visibility permissiveness

Balanced Configuration

Recommended for:

Characteristics:

> General Multiplayer Games

- Moderate expansion values
- Good balance between fairness and precision

Aggressive Configuration

Recommended for:

Characteristics:

> Fast-Paced Arcade Games

- Larger expansion values
- Earlier visibility detection
- More forgiving movement behavior

Common Mistakes

Excessive Expansion

Very large expansion values may cause players to become visible too early. Example:

> Player Still Behind Cover
>
> ↓
>
> Visibility Triggered

This can make visibility feel overly aggressive.

No Expansion

Disabling or minimizing expansion may reintroduce corner-peeking inconsistencies. Example:

> Player Exposed
>
> ↓
>
> Visibility Triggered Too Late

Oversized `PlayerBox`

Combining large `PlayerBox` dimensions with aggressive expansion values can lead to excessive visibility. Always validate sizing using debug visualization.

Testing Velocity Compensation

The recommended testing procedure is:

Debug visualization should be enabled during tuning.

1. Create a wall
2. Create two players
3. Move around corners at different speeds
4. Observe visibility transitions
5. Adjust Expanded Size if necessary

Using Velocity Compensation As Latency Mitigation

The Professional Edition includes dedicated latency-aware visibility compensation, described in the Network Compensation chapter.

As a result, players with higher latency may occasionally experience visibility transitions that occur slightly later than expected during fast movement or corner peeking situations.

To partially mitigate this behavior, developers can increase the velocity compensation settings. Conceptually:

Higher Ping

↓

Higher Movement Uncertainty

↓

Increased Velocity Compensation

↓

Earlier Visibility Detection

This approach does not replace true latency compensation, but it can help reduce visibility inconsistencies in multiplayer environments where moderate network latency is expected.

Recommended Usage

Scenario	Recommendation
LAN / Local Network	Conservative Velocity Compensation
Typical Online Multiplayer	Balanced Velocity Compensation
High-Ping Environments	More Aggressive Velocity Compensation

Important Considerations

Increasing velocity compensation may cause players to become visible slightly earlier than their actual visual exposure.

Developers should balance:

to find the configuration that best matches their game's networking conditions.

> Fairness vs
>
> Visibility Precision

Professional Edition

Dedicated latency-aware visibility systems use network latency information directly during visibility calculations, so velocity compensation no longer has to be used as a workaround.

Velocity compensation therefore remains useful for movement accuracy, while latency handling is delegated to the dedicated systems.

Best Practices

- Enable Velocity Compensation for multiplayer games
- Start with moderate expansion values
- Test corner-peeking situations frequently
- Use realistic `PlayerBox` dimensions
- Avoid excessive expansion values
- Increase compensation gradually when supporting higher latency environments
- Validate changes using debug gizmos

Summary

Velocity Compensation is one of the most important features of the visibility system.

It improves fairness during movement, reduces corner-peeking inconsistencies, and can even be used as a practical latency-mitigation technique in environments where dedicated latency-aware visibility systems are not available.

When properly configured, it provides a significant gameplay benefit while maintaining a very low performance cost.

13

Performance Optimization

Overview

The Anti-Wallhack Visibility System has been designed to scale efficiently by combining multiple filtering stages, configurable update rates, and optimized visibility sampling.

While the system is highly configurable, proper tuning is essential to achieve the best balance between visibility accuracy and server performance.

This chapter explains the most important settings that affect performance and provides practical recommendations based on benchmark testing.

Understanding Visibility Cost

The total workload of the visibility system depends on several factors:

In general, performance is affected more by the number of visibility candidates than by the total number of players present in the game.

Number of Players

↓

Number of Observers

↓

Visibility Candidates

↓

Visibility Sampling

↓

CPU Usage

A player located on the opposite side of a large map often generates no meaningful visibility workload because they are rejected early by the pipeline.

Observer Update Rate

One of the most important performance settings is the `ObserverManager` update rate.

Recommended Configuration

This configuration was used during the official benchmark tests and provides an excellent balance between responsiveness and performance.

> 32 Updates Per Second

Benefits:

- Responsive visibility updates
- Stable CPU usage
- Suitable for most multiplayer games

Lower Update Rates

Examples:

Benefits:

> 16 TPS
>
> 20 TPS

- Lower CPU usage
- Better scalability
- Suitable for large player counts Trade-off:
- Slightly slower visibility updates

Higher Update Rates

Examples:

Benefits:

> 48 TPS
>
> 64 TPS

- Faster visibility updates
- Improved responsiveness Trade-off:
- Increased CPU usage

Higher update rates should only be used when gameplay requirements justify the additional cost.

180° vs 360° Observer Mode

Observer angle selection has a significant impact on performance.

180° Mode

The observer only evaluates targets located within the viewing direction. Benefits:

> Recommended

- Fewer visibility candidates
- Lower CPU usage
- Better scalability Recommended for:

- FPS games
- TPS games
- Tactical shooters
- Survival games

360° Mode

The observer evaluates targets in every direction. Benefits:

- Omnidirectional awareness
- Useful for AI systems
- Useful for top-down games Trade-offs:
- More visibility candidates
- Increased CPU usage

The official benchmark tests demonstrated that 360° mode generates a noticeably higher workload than 180° mode because more players pass the observer filtering stages.

View Distance

View distance directly affects how many targets reach visibility validation.

Large View Distance

Benefits:

- Longer detection range

Trade-offs:

- More visibility candidates
- Increased CPU usage

Small View Distance

Benefits:

- Better scalability
- Fewer visibility calculations Recommended for:
- Indoor maps
- Arena shooters
- Small multiplayer environments

Detection Modes

`PlayerObserver` provides two detection modes.

`ClosestPoint`

Benefits:

> Performance-Oriented

- Lowest CPU cost
- Fast visibility evaluation Recommended for:
- Large player counts
- Performance-sensitive environments

`BoundsCorners`

Benefits:

> Accuracy-Oriented

- Improved edge visibility handling
- Better precision

Trade-offs:

- More calculations Recommended for:
- Competitive multiplayer games
- Tactical shooters
- High-precision environments

Visibility Sampling Density

Sampling density has a direct impact on visibility workload. The main contributors are:

> Fixed Lines Dynamic Lines Grid Density

High Density

Benefits:

- Better visibility precision
- Better corner detection
- Improved small opening detection Trade-offs:
- Increased CPU usage

Moderate Density

Recommended for most games. Provides:

- Good visibility quality
- Good scalability
- Balanced performance

Low Density

Recommended for:

- Large-scale games
- Very high player counts
- Dedicated servers with strict performance budgets

Velocity Compensation

Velocity Compensation has a very small performance impact.

Because it primarily adjusts visibility volume dimensions rather than introducing additional visibility passes, it is generally recommended to keep it enabled.

Benefits:

- Better corner behavior
- Improved fairness
- Minimal CPU cost

Debug Visualization

Debug visualization should be used during development and testing. However, excessive debug rendering may affect editor performance. Recommended workflow:

Development

↓

Debug Enabled

Production

↓

Debug Disabled

This ensures that benchmark and gameplay performance measurements remain representative.

Benchmark Configuration

The official benchmark tests were performed using:

This configuration was selected because it represents a realistic production setup for most competitive multiplayer games.

> `ObserverManager` : 32 Updates Per Second Observer Mode: 180°
>
> Environment Detection: Disabled

Benchmark Results

These results represent a high-density benchmark environment designed to stress the visibility system.

Real multiplayer matches typically distribute players across larger areas, resulting in fewer simultaneous visibility candidates and lower average workload.

Scaling Recommendations

Small Games

Recommended:

> Up to 20 Players

- 32 TPS
- `BoundsCorners`
- Moderate Sampling
- Velocity Compensation Enabled

Medium Games

Recommended:

> 20 - 50 Players

- 32 TPS
- Moderate Sampling
- Optimized View Distance

Large Games

Recommended:

> 50+ Players

- 16–32 TPS
- `ClosestPoint`
- Lower Sampling Density
- Conservative View Distance

General Optimization Strategy

When optimizing performance: Adjust settings in this order:

1. View Distance
2. Observer Mode

3. Update Rate
4. Detection Mode
5. Sampling Density

This usually provides the largest performance gains with the smallest impact on visibility quality.

Best Practices

- Use 180° mode whenever possible
- Start with 32 TPS
- Keep view distance proportional to map size
- Use moderate sampling density
- Enable Velocity Compensation
- Profile before increasing accuracy settings
- Test under realistic multiplayer conditions

The Anti-Wallhack Visibility System is designed to provide scalable visibility validation while allowing developers to tune accuracy and performance according to the needs of their game.

14

PlayerCylinder

Overview

`PlayerCylinder` is the Professional Edition alternative to `PlayerBox`. Instead of representing the target with an axis-aligned box, it approximates the player using cylindrical geometry, which matches the silhouette of a humanoid character far more closely.

A box always contains empty volume at its corners. During corner peeking that empty volume is what produces the most visible inconsistencies, because a sampling line can hit a corner region where no character actually exists. `PlayerCylinder` removes most of that error.

Why Cylinder Geometry

- Reduces false positives caused by the unused corner volume of a box
- Produces more consistent results when the observer looks at the target from an angle
- Improves accuracy of visibility transitions around walls and cover
- Keeps the same event and networking flow used by `PlayerBox`

Collider Configuration

The cylinder dimensions are driven by the `PlayerVisibilityDetector`. Radius, height and center are read from the configured collider, so the visibility volume always follows the character controller or capsule already used by the game.

`PlayerCylinder` can be swapped for `PlayerBox` without changing any other component. The detector exposes the collider type, and the sampling system adapts automatically.

Expansion And Compensation

The following fields control how the cylinder is expanded during compensation:

- Expansion Factor - Multiplier applied to the cylinder when compensation is active. Higher values make visibility more permissive.
- Cylinder Expansion Axis Adjustment - Per-axis scaling of the expansion, allowing horizontal and vertical growth to be tuned independently.
- Skin Width - Small margin kept between the expanded cylinder and surrounding geometry.
- Speed Line Clamp - Upper bound applied to speed-driven expansion, preventing extreme values at very high velocities.

Advanced Wall Clamping

When the cylinder is expanded, it can intersect nearby walls. Advanced wall clamping shrinks the expanded volume back so that it never crosses solid geometry, which prevents the system from

reporting visibility through a wall.

Clamping can be disabled for advanced scenarios, but doing so is not recommended: without it, aggressive expansion values will produce false positives.

Performance Considerations

- Cylinder sampling is slightly more expensive than box sampling; budget for it when planning player counts
- Rays Clamp Per Frame limits how many clamping checks run in a single frame
- Use `PlayerBox` for distant or low-priority targets when maximum precision is not required

Debug Visualization

Editor gizmos draw the base cylinder, the expanded cylinder and the sampling points. Enabling them while tuning expansion values is the fastest way to understand how the volume reacts to movement.

15

Advanced Observer Modes

Overview

The Free Edition provides 180 degree and 360 degree observer modes. The Professional Edition adds a 270 degree mode designed for fast-paced competitive gameplay.

270 Degree Side Expansion

The 270 degree mode keeps the normal camera field of view and adds a temporary side expansion toward the direction the camera is rotating. It is a middle ground between the strict 180 degree cone and the fully distance-based 360 degree mode.

This behaviour exists because a player who turns quickly will render targets that were outside the field of view a few frames earlier. Expanding only toward the rotation direction compensates for that without permanently widening detection. Rotation drives the compensation and latency amplifies it, so the same turn reaches further on a poor connection than on a local one, while a stationary observer gains no extra range no matter how high the ping is.

- Side FOV Expansion - Additional angle applied toward the rotation direction at full rotation speed, before any latency bonus.
- Min Side FOV Angular Speed - Rotation speed at which the expansion starts.
- Max Side FOV Angular Speed - Rotation speed at which the expansion reaches its maximum.
- Side FOV Ping Bonus - Extra angle added on top of Side FOV Expansion as ping grows, reaching full strength at Max Ping For Full Compensation. It is still multiplied by the rotation factor, so it only applies while the camera is turning, and it requires the RotationAndLatency compensation mode. Set it to zero to keep the expansion driven by rotation alone. The combined range is capped at 90 degrees per side, which keeps the mode at roughly 270 degrees of total coverage.

Choosing An Observer Mode

- 180 degrees - Highest precision, lowest cost. Best for tactical games with slower camera movement.
- 270 degrees - Balanced, and the default. Recommended for competitive shooters with fast turns and mixed latency, since the side expansion adapts to each player's measured ping.
- 360 degrees - Distance-based detection around the observer, intended for projects that do not send the client camera direction to the server, and for simpler games where view direction is not part of the design. FOV and rotation compensation are not used in this mode, which makes it cheaper on bandwidth and more expensive on CPU, because every target within view distance reaches the sampling stage. Line of sight is still validated, so occluded players remain hidden.

16

Network Compensation

Overview

Server-authoritative visibility is evaluated on the server, but the player reacts to what was rendered on their own machine some milliseconds earlier. Network compensation closes that gap so that legitimate actions are not rejected because of latency.

The Professional Edition provides three complementary systems: FOV compensation, observer prediction and grace prediction. They can be combined.

FOV Compensation

FOV compensation temporarily widens the detection cone based on camera rotation speed, network latency, or both.

- FOV Compensation Mode - Selects the source of compensation: rotation, latency, or both.
- Min Angular Speed - Minimum camera rotation speed, in degrees per second, required before compensation begins.
- Max Angular Speed - Rotation speed at which maximum compensation is applied.
- Max Rotation FOV Expansion - Maximum expansion, in degrees, caused by camera rotation speed.
- Max Ping FOV Expansion - Maximum expansion, in degrees, caused by ping.
- Max Ping For Full Compensation - Ping at which the latency factor reaches its maximum.
- Smooth Speed - Speed of the transition, preventing the cone from snapping between values.

Observer Prediction

Observer prediction offsets the evaluation point based on movement and latency, anticipating where the observer will be when the result reaches the client.

- Observer Prediction Mode - Selects velocity, latency, or both as the prediction source.
- Max Prediction Distance - Upper bound applied to the predicted offset.
- Velocity Prediction Scale - Scale applied to the velocity component.
- Latency Expansion Multiplier - Scale applied to the latency component.
- Prediction Axis Mode - Restricts prediction to the lateral axis, the frontal axis, or both.
- Lateral Weight and Frontal Weight - Per-axis weighting of the predicted offset.

Grace Prediction

Grace prediction scales the strength of prediction with measured ping, so that players on good connections are affected as little as possible while players on poor connections still receive consistent results. Fish-Networking reports the owner round-trip time through the visibility driver, bounded by Max Reported Ping Ms so a client cannot inflate its own compensation.

- Use Grace Prediction - Enables the system.
- Grace Ping Start - Ping at which grace prediction starts to take effect.
- Grace Ping Max - Ping at which grace prediction reaches full strength.
- Min Grace Scale - Lower bound applied to the scale, keeping a minimum amount of prediction active.

Inactive Optimization

When a player stops moving, part of the per-frame work can be paused. This only suspends updates of reflection surfaces and does not change visibility results for a stationary target.

- Use Inactive Optimization - Enables the optimization.
- Inactive Delay - Delay, in seconds, before the optimization is applied after the player becomes idle.
- Movement Threshold - Minimum movement distance required to consider the player active.
- Rotation Threshold - Minimum rotation angle required to consider the player active.

Tuning Recommendations

- Start with compensation disabled and confirm that base visibility behaves correctly
- Enable one system at a time and observe the gizmos while moving and rotating
- Prefer moderate values: excessive compensation trades wallhack resistance for responsiveness
- Validate the final configuration under simulated latency, not only on a local host

17

Animation Compensation

Overview

A character collider is usually a static capsule or box, but the rendered character is not. During animations such as leaning, crouching or reloading, parts of the body extend beyond the collider. Animation compensation adjusts the visibility volume from the animated pose so that detection matches what is actually drawn.

Requirements

A Humanoid Animator must be present on the character, because the system reads standard bone positions from the animated pose. If no Animator is assigned, or the avatar is Generic instead of Humanoid, the feature is skipped, the base collider is used and a warning is logged once. On a dedicated server the Animator must use Always Animate, otherwise culling stops the bones from moving and no overflow is ever measured.

How The Pose Factor Is Computed

Bone positions are compared against the expanded collider, and the system measures how far the farthest bone extends beyond it, in meters, evaluating the horizontal and vertical axes separately. That overflow is added to the volume together with Animation Expansion Margin, so the limb ends up fully covered rather than merely touched. While every bone stays inside the expanded collider the overflow is zero and no correction is applied at all, which is the case for most animations. Growth is applied immediately, because an exposed limb must be covered on the same tick, while shrinking is smoothed by Smooth Speed so the volume does not pop between poses.

Velocity and latency scale the collider, while animation adds an absolute expansion in meters on top of it, so enabling animation compensation does not cancel the other compensation systems. Max Animation Expansion caps how much the animated pose alone can add, protecting the volume against broken poses or a misconfigured rig.

Box And Cylinder Behaviour

Both `PlayerBox` and `PlayerCylinder` support animation compensation. The cylinder path expands radius and height separately, which produces a closer fit for humanoid characters. The box path expands the horizontal and vertical extents, still respecting the configured expansion axes. In both cases wall clamping keeps the expanded volume from crossing solid geometry.

Performance Considerations

- Bone sampling runs only while the feature is enabled on the detector
- Cost scales with the number of tracked players, not with animation complexity

- Disable it for distant targets where pose accuracy is not perceivable

18

Environment Detection

Overview

Environment detection extends line-of-sight validation with indirect visual information. A player whose body is fully hidden may still be given away by a projected shadow or by a reflection in a mirror. The Professional Edition can validate both.

Environment Detection is a flags field. Shadow and Reflection can be enabled independently or together, and the field can be set to Disabled to use direct line-of-sight only.

Shadow Detection

`ShadowLight` Component

`ShadowLight` marks a light source capable of producing shadow visibility information. Add it to the lights that should participate in detection and register them on the `PlayerVisibilityDetector`.

Supported Light Types

- Directional - Infinite directional light, typically sunlight
- Point - Radial light source
- Spot - Cone-based light source

Shadow Modes

- Simple - Fast single-line shadow evaluation
- Accurate - Multi-line shadow sampling for improved precision

Light Settings

- Light Type - Defines the light source type
- Range - Maximum shadow influence distance
- Spot Angle - Cone angle used by spot lights
- Near Horizontal Angle - Controls how far the shadow projection spreads near the horizon
- Sync With Unity Light - Automatically synchronizes the configuration from the attached Unity Light component

Detector Settings

- Light Sources - List of `ShadowLight` components used for shadow visibility checks
- Min Shadow Vertical Angle - Minimum vertical angle required for a shadow to be considered valid
- Shadow Line Count - Number of lines used to sample the projected shadow
- Shadow Light Refresh Rate - How often the detector refreshes nearby or assigned shadow lights

- Enable Shadow Ground Projection - Projects shadow samples onto the environment for improved accuracy

Reflection Detection

`ReflectionSurface` Component

`ReflectionSurface` marks a mirror, window or any reflective plane that can reveal a player. Surfaces register themselves automatically while the component is enabled, so unlike `ShadowLight` there is no list to assign on the detector.

Surface Orientation

The reflection plane is defined by the transform position and its local positive Z axis, shown as the blue arrow in the Editor. That axis must point away from the reflective face. The yellow gizmo line indicates the current normal direction.

Reflection Modes

- Simple - Reflects a single main player point. Fastest option.
- Accurate - Reflects multiple collider bounds points for better precision near the surface edges.

Surface Settings

- Size - Width and height of the reflective area
- Edge Tolerance - Extra margin around the surface, preventing visibility from dropping abruptly near the edges

Detection Settings

- Max Distance - Maximum distance at which observers can detect reflections
- Reflection Surface FOV - Minimum viewing angle required before reflection checks run
- Reflection Mode - Simple or Accurate sampling
- Obstacle Mask - Layers that block the line between the observer and the surface itself

How Detection Works

- The observer runs a cheap surface check first: distance, then viewing angle, then a linecast to the surface center using the surface Obstacle Mask
- Only surfaces that pass this check are evaluated for that tick
- The observer position is mirrored across the reflection plane, and the point where the mirrored line crosses the plane becomes the reflection point
- The reflection point must land inside the surface area, including Edge Tolerance
- Two final linecasts validate the path: observer to reflection point, and reflection point to player, both using the detector Obstacles Mask

Enabling Show Reflection Lines on the detector draws the full path in the Scene view. Cyan is the observer to reflection point segment, green is reflection point to player, and red marks a rejected

reflection.

Environment Detection Performance

- Use Simple mode when edge precision is not critical
- Keep Max Distance and Reflection Surface FOV as tight as the map allows, since both reject surfaces before any reflection math runs
- Restrict obstacle masks to layers that can realistically block a shadow or a reflection
- Limit the number of active shadow lights and reflection surfaces in large maps
- Reduce Shadow Light Refresh Rate before reducing shadow line count

19

Benchmarking Tools

Overview

Both editions ship a benchmark scene. The Professional Edition extends it with a higher-density stress setup and with performance analysis tools that report the real cost of the visibility system.

Benchmark Scene

The scene spawns a configurable number of players inside a bounded area and runs the visibility system under a server-like update rate. Player count, spawn radius, team count, warmup time and test duration are all configurable.

Metrics

- System Time - Time consumed by the visibility system itself
- Cost Per Player - System time divided by the number of tracked players
- Raycasts Per Frame - Number of physics queries issued
- Hit Ratio - Proportion of queries that were blocked
- FPS Impact - Framerate difference against the same scene with the system removed

Reading The Results

Benchmark figures represent a deliberately dense worst case. Real multiplayer maps distribute players over much larger areas, which reduces the number of simultaneous visibility checks and lowers the measured cost considerably.

Use the benchmark to compare configurations rather than as an absolute prediction: change one setting at a time, re-run, and keep the configuration that meets the target player count.

20

Troubleshooting

Overview

This chapter covers the most common issues encountered when integrating the Anti-Wallhack Visibility System and provides recommended solutions.

Most problems are related to component setup, initialization order, layer masks, or networking integration.

Before investigating advanced issues, always verify that the system has been configured correctly.

Players Are Never Detected

Symptoms

Possible Causes

> Targets remain hidden at all times No visibility events are triggered No visible state changes occur

`ObserverManager` Missing

Verify that an `ObserverManager` exists in the scene.

The visibility system requires a single active `ObserverManager` instance.

Components Not Initialized

Verify that: **`PlayerObserver`** **`PlayerVisibilityDetector`** have been initialized correctly.

For networking integrations, ensure that: **InitializeSystem()** is executed before:

View Distance Too Small

> ServerTick()

Verify that the target is located within: **viewDistance** configured in `PlayerObserver`.

Incorrect FOV Configuration

Verify:

Incorrect values may prevent targets from entering the visibility pipeline.

> cameraFOVAngle aspectFormat maxObserverAngle

Players Become Visible Too Early

Symptoms

Possible Causes Oversized `PlayerBox` Verify:

> Targets visible before fully leaving cover Visibility feels overly aggressive

> Size Expanded Size

Excessively large visibility volumes may increase detection range.

Excessive Velocity Compensation

Large expansion values may cause targets to become visible sooner than expected. Reduce: **Expanded Size** and retest.

Players Become Visible Too Late

Symptoms

Possible Causes

> Players visually exposed before detection occurs Corner peeking feels inconsistent

Visibility Volume Too Small

Verify that `PlayerBox` dimensions match the visual size of the character.

Velocity Compensation Too Conservative

Increase velocity compensation gradually and retest movement scenarios.

Update Rate Too Low

Very low update rates may delay visibility updates. Recommended starting value:

> 32 TPS

Visibility Flickering

Symptoms

Possible Causes

> Rapid Visible/Hidden transitions Unstable visibility near walls

State Stabilization Too Low

Increase: **stateStabilizeTime** to smooth transitions.

Very Small Visibility Openings

Windows, railings, and thin geometry may naturally produce visibility fluctuations. Increase sampling density if necessary.

Visibility Validation Never Runs

Symptoms

Possible Causes ServerTick Not Executing Verify that:

> No visibility updates No visibility events

> ServerTick()

is called regularly.

For multiplayer projects, ensure it executes inside the networking framework's authoritative update loop.

Components Disabled

Verify that: `NetworkPlayerObserver` `NetworkPlayerVisibilityDetector` are enabled.

Debug Gizmos Are Not Visible

Symptoms

Possible Causes

> No lines visible in Scene View
>
> No debugging information displayed

Gizmos Disabled

Verify that Unity Gizmos are enabled in the Scene View.

Debug Options Disabled

Verify that the corresponding debug options are enabled in the component inspector.

Performance Is Lower Than Expected

Symptoms

Possible Causes Excessive Sampling Density Reduce:

> Fixed Lines Dynamic Lines Grid Density

> High CPU usage
>
> Unexpected visibility cost

Excessive View Distance

Reduce: **viewDistance** where possible.

Using 360° Mode Unnecessarily

Prefer: **180° Mode** whenever gameplay allows.

Update Rate Too High

Verify that `ObserverManager` is not running at an unnecessarily high update frequency.

Networking Synchronization Issues

Symptoms

Possible Causes

> Different visibility results across clients Inconsistent visibility states

Visibility Validation Executing On Clients

Visibility validation should execute only on the authoritative side. Incorrect:

Correct:

> Client
>
> ↓
>
> ServerTick()

> Server
>
> ↓
>
> ServerTick()

Adapter Components Not Integrated Correctly

Review: **`NetworkPlayerObserver`** **`NetworkPlayerVisibilityDetector`** and verify that they are connected to the networking framework's update loop.

Before Requesting Support

Before reporting an issue, verify:

Most integration issues can be resolved by checking the items above.

- `ObserverManager` exists

- Components initialized correctly
- ServerTick() executes
- View Distance configured
- FOV configured
- Layer masks configured
- Debug visualization enabled
- Visibility volume configured
- Networking integration validated

Additional Help

For updated examples, integration guides, and future improvements, refer to the online documentation.

The online documentation is continuously updated and may contain newer information than this PDF version.

21

Frequently Asked Questions (FAQ)

General Questions

What Is The Purpose Of The Anti-Wallhack Visibility System?

The Anti-Wallhack Visibility System is a server-authoritative visibility validation solution designed to reduce information exposure in multiplayer games.

Instead of relying on client-side rendering state, the system determines visibility using physics-based validation, field-of-view checks, and visibility sampling.

This helps prevent wallhack-style exploits and unnecessary network synchronization.

Is This A Networking Framework?

No.

The Anti-Wallhack Visibility System is a visibility validation framework. It integrates with networking solutions but does not replace them.

Examples:

Photon Fusion Mirror Fish-Networking Netcode for GameObjects Photon PUN

Does The System Replace Interest Management?

> Not necessarily.
>
> The system can be used alongside interest management solutions or as a visibility layer that helps determine what information should be synchronized.
>
> The exact architecture depends on the requirements of the project.

Networking Questions

Does It Work With Photon Fusion?

Yes.

The system was designed with server-authoritative multiplayer architectures in mind and integrates well with Photon Fusion.

The recommended approach is to execute visibility validation inside:

FixedUpdateNetwork()

on the authoritative side.

Does It Work With Mirror?

Yes.

Mirror projects can execute visibility validation on the server and synchronize only the resulting visibility information to clients.

Does It Work With Fish-Networking?

Yes.

Fish-Networking can integrate with the visibility system through the provided adapter architecture.

Does It Work With Netcode For GameObjects?

Yes.

Visibility validation can be executed on the server and used to control synchronization decisions.

Can I Use Another Networking Framework?

Yes.

The visibility system is networking-framework agnostic.

Any multiplayer framework capable of executing server-authoritative logic can integrate with the system.

Visibility Questions

Which Observer Mode Should I Use?

For most multiplayer games:

180° Mode

is recommended. Benefits:

- Better performance
- Fewer visibility candidates
- More realistic player perception

When Should I Use 360° Mode?

360° mode is useful when directional visibility restrictions are not desired. Examples:

- AI systems
- Top-down games
- RTS games
- Omnidirectional detection

Because more targets pass the observer filtering stage, 360° mode generally has a higher performance cost.

What Is The Difference Between `ClosestPoint` And `BoundsCorners`?

`ClosestPoint`

Benefits:

- Faster

Lower CPU usage

Recommended for:

Large Player Counts

`BoundsCorners`

Benefits:

- Higher accuracy
- Better edge visibility handling

Recommended for: Competitive Multiplayer Games

What Is Velocity Compensation?

Velocity Compensation expands the visibility volume based on movement speed. Benefits:

- Improved corner behavior
- Better visibility consistency
- Fairer multiplayer interactions

The feature is included in both editions.

Can Velocity Compensation Help With Latency?

Partially.

Increasing velocity compensation can help reduce visibility inconsistencies in higher-latency environments.

However, it should be considered a practical workaround rather than a replacement for dedicated latency-aware visibility systems.

Performance Questions

How Many Players Does The System Support?

The exact number depends on:

- Visibility settings
- Update rate
- Sampling density
- Map design
- Hardware

The official benchmark included tests with:

10 Players

25 Player

50 Players

100 Players

using:

32 Updates Per Second 180° Observer Mode Environment Detection Disabled

> Real multiplayer games typically distribute players across larger environments and often produce lower workloads than synthetic stress tests.

What Update Rate Should I Use?

> Recommended default:
>
> 32 TPS

This provides a good balance between responsiveness and performance.

Does Increasing Sampling Density Always Improve Results?

Not necessarily.

Beyond a certain point, increasing sampling density produces diminishing returns while increasing CPU usage.

Start with moderate values and profile before making adjustments.

Integration Questions

Can I Modify The Network Adapter Components?

Yes.

The included adapters are intended as reference implementations.

Most projects will eventually adapt them to match their networking architecture.

Will Asset Updates Overwrite My Adapter Changes?

Potentially.

If the adapter scripts are modified directly, future updates may replace them with newer default versions.

Recommended workflow:

Import Asset ↓ Adapt Components ↓ Move Customized Versions Into Project Code

> This helps protect project-specific modifications.

Can I Create My Own Adapter Components?

> Absolutely.
>
> The visibility system was designed to support custom integrations.
>
> Many projects create their own adapter layer to match existing architecture and coding standards.

Gameplay Questions

Can I Use The System For PvE Games?

Yes.

The visibility system can be used for:

- AI perception
- Stealth mechanics
- NPC awareness
- Visibility-based gameplay

It is not limited to PvP environments.

Can Teammates Always Remain Visible?

Yes. Visibility Filtering allows developers to selectively bypass visibility validation for teammates and other friendly entities.

The system is available in the Professional Edition and is configured directly on the `PlayerVisibilityDetector`.

Examples:

Team Markers

Friendly NPCs Squad Members Pets

Can I Use The System For Spectators?

Yes.

Custom visibility rules can be implemented through networking integration and future filtering systems.

Support Questions

Where Can I Find Updated Examples?

Framework-specific integration examples are maintained in the online documentation. The online documentation should always be considered the most up-to-date reference.

The PDF And Online Documentation Are Different. Which One Is Correct?

The online documentation is the primary source of information.

Because the system continues to evolve, the online documentation may contain:

- New examples
- Updated recommendations
- Additional integrations
- New features

before future PDF revisions are published.

Where Should I Start?

Recommended order:

1. Quick Start

1. Core Components
2. Network Adapter Components
3. Visibility Pipeline
4. Networking Integration

> 6. Performance Optimization

This provides the fastest path from installation to production integration.

If your question is not covered here, refer to the online documentation or future documentation updates for additional guidance.

22

Online Documentation

Overview

This PDF provides an offline reference for the Professional Edition of the Anti-Wallhack Visibility System.

However, the online documentation should always be considered the primary and most up-to-date source of information.

Because the system continues to evolve, the online documentation may receive updates before future PDF revisions are published.

Official Documentation

The latest documentation is available at:

> https://gustavobianco277.github.io/Anti-Wallhack-Visibility-System/

The online version includes:

- Updated setup guides
- Framework integration examples
- Troubleshooting information
- Frequently asked questions
- Performance recommendations
- Benchmark results
- New documentation updates

Why Use The Online Documentation?

Unlike a PDF, the online documentation can be updated immediately whenever:

- New features are released
- Existing features are improved
- Integration examples are updated
- New frameworks are supported
- Best practices evolve

This ensures that developers always have access to the most current information available.

Framework Integration Examples

The online documentation contains framework-specific integration examples that are intentionally omitted from this PDF.

Examples include:

These examples may change over time as networking frameworks receive updates.

> Photon Fusion Photon PUN 2 Mirror
>
> Fish-Networking Netcode for GameObjects

For this reason, the online documentation should be consulted whenever implementing or updating multiplayer integrations.

Performance Benchmarks

The latest benchmark results are maintained online. Benchmark documentation includes:

- Test configurations
- Update rate settings
- Observer configurations
- Performance measurements
- Future benchmark updates

As new versions of the system are released, benchmark information may be expanded and updated.

Future Features

The online documentation may also contain information about upcoming features that are not yet included in the current PDF version.

Examples:

These sections are subject to change as development continues.

> Additional Integrations New Optimization Techniques
>
> Future Professional Edition Features

Documentation Update Policy

The online documentation is considered the authoritative source for:

Future PDF revisions will periodically incorporate these updates, but the website should always be treated as the most current reference.

> Feature Documentation Integration Examples Troubleshooting Guides Optimization Recommendations Release Notes

Recommended Workflow

For the best experience:

This approach combines the convenience of an offline reference with the flexibility of continuously updated online resources.

PDF Documentation

↓

Learn Core Concepts

↓

Online Documentation

↓

Framework Integration

↓

Production Deployment

Final Notes

Thank you for using the Anti-Wallhack Visibility System.

The goal of this project is to provide a flexible, scalable, and server-authoritative visibility solution that can be integrated into a wide variety of Unity multiplayer games.

Feedback, suggestions, and future improvements will continue to shape the project as it evolves. For the latest information, examples, and updates, always refer to the official online documentation.