

ECS Fog of War RTS

Documentation

Version: 1.1.0

Online Documentation: https://ecs-rts-fogofwar.netlify.app

Krzysztof Bziuk

Table of contents

1. Fog of War RTS ECS 3
   - 1.1 Watch the Demo 3
   - 1.2 How It Works 3
   - 1.3 Key Features 3
   - 1.4 Main Components 3
   - 1.5 Getting Started 3
2. Quick Setup 4
   - 2.1 Setup Steps 4
3. Components 5
   - 3.1 FogOfWarManager 5
   - 3.2 Team 7
   - 3.3 VisionSource 8
   - 3.4 VisibilitySwitch 9
   - 3.5 MapFogOverlayUI 10
4. Vision Layers 11
   - 4.1 Overview 11
   - 4.2 Configuration 11
   - 4.3 Example: Multi-Level Terrain 11
5. Entity Fade Setup 13
   - 5.1 Overview 13
   - 5.2 Setup Steps 13
6. AI Vision Integration 15
   - 6.1 VisionSource on AI units 15
   - 6.2 VisibilitySwitch on player units 15
   - 6.3 How to use it in AI 15
   - 6.4 Typical setup 15
7. Examples 16
   - 7.1 Example Scenes 16
8. Performance 17
   - 8.1 Performance 17
   - 8.2 Quick Optimization Tips 17

1. Fog of War RTS ECS

Fog of War RTS ECS brings the classic fog of war mechanic from strategy games like Age of Empires and StarCraft to Unity. Built on Unity DOTS (ECS, Jobs, and Burst), the system efficiently handles up to 100,000 units while maintaining smooth performance.

Unexplored areas remain completely hidden, while previously discovered regions appear dimmed when not actively observed. The fog updates in real-time as units move across the map.

1.1 Watch the Demo

See the system in action with 100k units:

YouTube video

1.2 How It Works

The fog of war system divides your map into three visibility states:

- **Hidden** - Unexplored regions never discovered by any vision source
- **Explored** - Previously discovered areas not currently in vision range
- **Visible** - Regions currently within units' vision range

As units move, the fog updates automatically with smooth transitions between states.

1.3 Key Features

- **High Performance** - Tested with 100,000 units using Burst-compiled jobs on multiple CPU cores
- **Vision Shapes** - Circle, square, and triangle vision patterns
- **Vision Layers** - Up to 4 independent RGBA layers for multi-level terrain or team-based vision
- **Smooth Transitions** - Configurable fade-in and fade-out durations for fog and entities
- **Minimap Support** - Apply fog overlay to UI minimaps with a single component
- **Persistent Fog** - Save and load explored areas between game sessions
- **Entity Visibility** - Automatically show/hide entities based on vision with optional fade effects
- **Real-Time Updates** - Fog responds instantly to unit movement

1.4 Main Components

- `FogOfWarManager` - Core component managing fog rendering and GPU resources
- `VisionSource` - Reveals fog around units in customizable shapes
- `VisibilitySwitch` - Controls entity visibility based on fog state
- `MapFogOverlayUI` - Applies fog effect to UI minimaps

1.5 Getting Started

Ready to add fog of war to your project? Check out the Quick Setup guide to get started in minutes.

2. Quick Setup

This guide shows you how to add fog of war to your Unity project in a few simple steps.

2.1 Setup Steps

2.1.1 1. Create the Scene

1. Create a new scene
2. Create a plane: **GameObject > 3D Object > Plane**
3. Scale it to your desired map size (e.g., scale `10, 1, 10` for a 100×100 world)

2.1.2 2. Create a Sub-Scene

The system uses Unity DOTS, which requires a sub-scene:

1. Create an empty GameObject
2. Right-click it and select **New Sub Scene > Empty Scene...**
3. Name it "GameWorld"

2.1.3 3. Add FogOfWarManager

1. In the main scene, create an empty GameObject named "FogOfWarManager"
2. Add the `FogOfWarManager` component
3. Set the map bounds to match your plane:
4. For a 100×100 plane: **Left** `-50`, **Right** `50`, **Bottom** `-50`, **Top** `50`

2.1.4 4. Add a Unit with Vision

Inside the sub-scene:

1. Create a GameObject (your unit)
2. Add `VisionSourceAuthoring` component
3. Set **Scale** to `(10, 10)` for a 10-unit vision radius

2.1.5 5. Test

Enter Play Mode. You should see fog covering everything except around your unit.

3. Components

3.1 FogOfWarManager

The `FogOfWarManager` component is the central coordinator that owns all GPU resources and acts as the bridge between the ECS world and the render pipeline.

3.1.1 Overview

`FogOfWarManager` is a singleton `MonoBehaviour` that manages the fog of war system. It creates render textures, uploads vision data to the GPU, and applies the fog effect to cameras with matching tags.

The manager automatically creates a `FogConfig` ECS entity that ECS systems use to coordinate processing. Disabling the manager GameObject also pauses all ECS fog processing.

3.1.2 FogOfWarManager (Inspector)

Map Bounds

Property	Type	Default	Description
Left Bound	`float`	`-5`	Left edge of the fogable area in world space
Right Bound	`float`	`5`	Right edge of the fogable area in world space
Bottom Bound	`float`	`-5`	Bottom edge of the fogable area in world space
Top Bound	`float`	`5`	Top edge of the fogable area in world space

Visuals

Property	Type	Default	Description
Camera Tags	`string[]`	`["MainCamera"]`	Tags of cameras that receive the fog screen effect
Injection Point	`InjectionPoint`	`AfterRenderingPostProcessing`	URP render pass event at which the fog effect is applied: `BeforeRe` `AfterRenderingTransparents`, `BeforeRenderingPostProcessing`
Unexplored Fog Texture	`Texture`	`null`	Tiling texture overlaid on unexplored areas (leave null for solid colo
Unexplored Fog Tint	`Color`	`(1, 1, 1)`	Color tint for unexplored areas
Explored Fog Texture	`Texture`	`null`	Tiling texture for areas that have been explored but are not current
Explored Fog Tint	`Color`	`(0.5, 0.5, 0.5, 0.5)`	Color tint for explored areas
Fade In Duration (seconds)	`float`	`0.2`	Time for fog to clear when an area enters vision
Fade Out Duration (seconds)	`float`	`0.2`	Time for fog to return when an area leaves vision
Entity Fade Duration (seconds)	`float`	`0.2`	Time for entity fade transitions (used by `UpdateVisibilitySystem`
Filter Mode	`FilterMode`	`Bilinear`	Filtering applied to fog render textures: `Point`, `Bilinear`, or `Tril`
Blur Strength	`float`	`3`	Gaussian blur intensity (0 = no blur, 3 = maximum)

Vision Layers

Property	Type	Default	Description
Layers	`Texture`	`null`	Optional RGBA texture that defines areas where vision can reveal fog. Each channel (R, G, B, A) creates a separate layer that can be matched with vision sources. White areas allow vision, black areas block it. This enables features like team-specific vision or terrain-based visibility restrictions (see Vision Layers)

Performance

Property	Type	Default	Description
Texture Size	`Vector2Int`	`(1024, 1024)`	Resolution of internal fog render textures. Higher values produce more accurate fog boundaries and less pixelated edges, but consume more GPU memory
Max Vision Sources	`int`	`100`	Maximum number of vision sources that will be processed per frame. If more vision sources exist in the scene, the excess will be ignored

3.1.3 Public API

`GetDataToSave()` → `Vector2Int[]`

Returns a run-length encoded representation of the explored area. Each `Vector2Int` stores the start (inclusive) and end (exclusive) pixel index of a run of explored pixels.

`Load(Vector2Int[] data)`

Restores a previously saved explored state by writing data back into the internal render texture.

3.2 Team

The `Team` setup assigns an entity to a single faction.

3.2.1 Overview

In the ECS version there are multiple related types:

Type	Kind	Purpose
`TeamAuthoring`	`MonoBehaviour`	Placed on a sub-scene GameObject; baked into ECS at build/enter play mode
`TeamMember`	`IComponentData`	Stores the entity's runtime team assignment
`PlayerTag`	`IComponentData`	Automatically added when the team is set to `Player`

3.2.2 TeamAuthoring (Inspector)

Add `TeamAuthoring` to any GameObject inside a Unity sub-scene that should belong to a team.

Property	Type	Default	Description
Team	`Team`	`Player`	The team assigned to the baked entity

3.3 VisionSource

Entity with a `VisionSource` component reveals fog when it belongs to the `Player` team and is also used to detect whether any `VisibilitySwitch` entities are inside its vision range.

> **Info**
>
> `VisionSourceAuthoring` depends on a team assignment. `TeamAuthoring` is added automatically, but you still need to choose which team owns this vision source. See Team.

> **Warning**
>
> This component reveals fog only when assigned to `Player`. Do not place it on enemy units unless you want to use their vision for AI logic. See AI Vision Integration.

3.3.1 Overview

Type	Kind	Purpose
`VisionSourceAuthoring`	`MonoBehaviour`	Placed on a sub-scene GameObject; baked into ECS at build/enter play mode
`VisionSource`	`IComponentData`	The runtime ECS component used by `CalculateVisionSystem`

3.3.2 VisionSourceAuthoring (Inspector)

Add `VisionSourceAuthoring` to any GameObject inside a Unity sub-scene that should act as a vision source.

Property	Type	Default	Description
Shape	`VisionShape`	`Circle`	The geometry of the vision area: `Circle`, `Square`, or `Triangle`
Layer	`VisionLayer`	`R`	Determines which fog layer this source reveals (see Vision Layers)
Position Offset	`Vector2`	`(0, 0)`	Local XZ offset from the entity's pivot
Rotation Offset	`float`	`0`	Angle offset in degrees applied on top of the entity's world rotation
Scale	`Vector2`	`(1, 1)`	Width (X) and height (Z) of the vision area

3.4 VisibilitySwitch

Entities with a `VisibilitySwitch` component are shown or hidden based on whether they are seen by the player. They also store information about which team is currently detecting them.

> **Info**
>
> `VisibilitySwitchAuthoring` requires a team assignment. `TeamAuthoring` is added automatically, but you still need to choose which team owns this entity. See Team.

> **Warning**
>
> Do not place this component on player units unless you want to use the same visibility data for AI logic. See AI Vision Integration.

3.4.1 Overview

In the ECS version there are multiple related types:

Type	Kind	Purpose
`VisibilitySwitchAuthoring`	`MonoBehaviour`	Placed on a sub-scene GameObject; baked into ECS at build/enter play mode
`DetectionLayer`	`IComponentData`	Which fog layer(s) can reveal this entity
`VisibilityState`	`IComponentData`	Stores which teams currently detect this entity
`DynamicBuffer<CheckPoint>`	`IBufferElementData`	Local-space sampling positions
`FadePercent`	`IComponentData`	Optional smooth fade value for fade transitions (see Entity Fade Setup)

3.4.2 VisibilitySwitchAuthoring (Inspector)

Add `VisibilitySwitchAuthoring` to any GameObject inside a Unity sub-scene that should be revealed or hidden by the fog.

Property	Type	Default	Description
Layer	`VisionLayer`	`R`	Which vision layer(s) can reveal this entity (see Vision Layers)
Use Center As Checkpoint	`bool`	`true`	When enabled, uses the entity's center as a single check point
Check Points	`Vector2[]`	`[(0, 0)]`	Array of local XZ offsets for visibility sampling (only visible when "Use Center As Checkpoint" is disabled)
Use Fade Transition	`bool`	`false`	When enabled, the entity fades in/out instead of instantly appearing/disappearing. Requires a material with a specially configured shader (see Entity Fade Setup)

Check Points

The **Check Points** array allows you to sample multiple positions on large entities. An entity is treated as visible as soon as *any* check point falls inside *any* matching vision source.

When **Use Center As Checkpoint** is enabled, the system uses a single point at `(0, 0)` relative to the entity's transform.

3.5 MapFogOverlayUI

The `MapFogOverlayUI` component renders the fog state onto a UI element, making it easy to display fog of war on a minimap.

3.5.1 Overview

`MapFogOverlayUI` is a standard `MonoBehaviour` that works with Unity's UI system. It reads the global fog shader properties set by `FogOfWarManager` and applies them to an `Image` or `RawImage` component.

The component requires either an `Image` or `RawImage` on the same GameObject.

3.5.2 Setup

1. Add an `Image` or `RawImage` component to a UI GameObject
2. Assign a sprite to the `Image` / `RawImage` to display your minimap background
3. Add the `MapFogOverlayUI` component to the **same** GameObject
4. Configure the Inspector properties below

3.5.3 MapFogOverlayUI (Inspector)

Property	Type	Default	Description
Unexplored Fog Texture	`Texture2D`	`null`	Tiling texture for unexplored areas on the minimap
Unexplored Fog Tint	`Color`	`(0, 0, 0, 1)`	Color tint for the unexplored fog overlay
Explored Fog Texture	`Texture2D`	`null`	Tiling texture for explored-but-not-visible areas
Explored Fog Tint	`Color`	`(1, 1, 1, 0.5)`	Color tint for the explored area overlay

4. Vision Layers

Vision layers allow you to create independent fog channels on the same map. This enables multi-level environments, team-based vision, or any scenario where different groups of units should have separate lines of sight.

4.1 Overview

The fog system uses RGBA channels to provide up to four independent fog layers (R, G, B, A). Each vision source reveals specific layers, and entities are only visible when revealed by matching layers.

4.2 Configuration

4.2.1 1. FogOfWarManager - Layers Texture

Open the `FogOfWarManager` component and locate the **Vision Layers** section.

The **Layers** texture is an optional RGBA texture that defines where each layer can reveal fog:

- **Leave empty** - All layers work everywhere on the map
- **Assign RGBA texture** - Control which areas each layer can reveal
- **White** (1.0) in a channel = that layer can reveal fog in this area
- **Black** (0.0) in a channel = that layer cannot reveal fog in this area

For example, to create a two-floor building: - Paint the **ground floor** area with **Red channel only** (R=1, G=0, B=0, A=0) - Paint the **upper floor** area with **Green channel only** (R=0, G=1, B=0, A=0)

You can create this texture in any image editor (Photoshop, GIMP, etc.) and import it as a regular texture.

4.2.2 2. VisionSource - Layer Property

Select any GameObject with `VisionSourceAuthoring` and set the **Layer** property to determine which fog channels this source reveals:

You can select multiple layers by clicking the dropdown and checking multiple options.

4.2.3 3. VisibilitySwitch - Layer Property

Select any GameObject with `VisibilitySwitchAuthoring` and set the **Layer** property to determine which vision layers can reveal this entity:

An entity becomes visible when **any** vision source on a matching layer covers one of its check points.

4.3 Example: Multi-Level Terrain

Let's create a map with normal terrain, a ramp leading up, and a raised hill where ground units can't see the upper level from below:

4.3.1 Step 1: Create the Layers Texture

1. Create a new texture in your image editor matching your map size
2. Paint the **normal terrain area** with pure **red** (R=255, G=0, B=0)
3. Paint the **raised hill area** with pure **green** (R=0, G=255, B=0)
4. Paint the **ramp/transition area** with **yellow** (R=255, G=255, B=0) so units can see both levels there
5. Import the texture into Unity and assign it to **FogOfWarManager → Vision Layers → Layers**

4.3.2 Step 2: Configure Ground Units

For units that walk on normal terrain:

1. Add `VisionSourceAuthoring` component
2. Set **Layer** to `R` (red channel only)
3. Add `VisibilitySwitchAuthoring` component
4. Set **Layer** to `R` (red channel only)

These units can only reveal and be revealed on the normal terrain level.

4.3.3 Step 3: Configure Flying Units

For flying units that can see both terrain levels:

1. Add `VisionSourceAuthoring` component
2. Set **Layer** to `R, G` (both red and green channels)
3. Add `VisibilitySwitchAuthoring` component
4. Set **Layer** to `R, G` (both red and green channels)

These units can reveal and be revealed on both terrain levels.

4.3.4 Step 4: Handle Units Moving Between Levels

When a ground unit walks up the ramp to the hill, change its layers at runtime:

1. Get the entity's `VisionSource` component
2. Change `Layer` from `R` to `R, G`
3. Get the entity's `DetectionLayer` component
4. Change `VisionLayer` from `R` to `R, G`

The unit can now see both levels. When it walks back down, change the layers back to `R` only.

4.3.5 Result

- Ground units on normal terrain can see each other but not up to the hill
- Flying units can see everything on both terrain levels
- Ground units that walk up the ramp gain vision of the upper hill
- The system automatically handles fog revealing based on the layers texture and unit settings

5. Entity Fade Setup

This page explains how to set up entities with smooth fade transitions when they become visible or hidden by the fog of war.

5.1 Overview

Entities with the `VisibilitySwitch` component can either appear/disappear instantly or fade smoothly. To enable fade transitions, you need to configure the global fade duration in `FogOfWarManager`, enable fade on the entity's `VisibilitySwitchAuthoring` component, and create a material with a shader that supports the fade effect.

5.2 Setup Steps

5.2.1 1. Configure Global Fade Duration

The global fade duration for all entities is controlled in the **FogOfWarManager** component:

- Select the GameObject with the `FogOfWarManager` component in your scene
- Set the **Entity Fade Duration** property to the desired fade time in seconds (e.g., 0.2 for a quick fade, 1.0 for a slower fade)

5.2.2 2. Enable Fade in VisibilitySwitchAuthoring

On each entity that should fade:

1. Select the entity GameObject
2. Find the `VisibilitySwitchAuthoring` component
3. Enable the **Use Fade Transition** checkbox

5.2.3 3. Create Material with Shader Graph

The entity needs a material with a shader that supports fade transitions. Follow these steps to create it in Shader Graph:

Step 1: Create a new Shader Graph

- Right-click in Project window → Create → Shader Graph → URP → Lit Shader Graph (or Unlit, depending on your needs)

Step 2: Set Shader to Transparent

- Open the Shader Graph
- In the Graph Inspector, find the **Surface Type** setting
- Change it from **Opaque** to **Transparent**

Step 3: Add FadePercent Property

- In the Blackboard, click the **+** button to add a new property
- Select **Float**
- Name it **exactly** `FadePercent` (case-sensitive)
- In the property settings:
- Set **Reference** to `FadePercent` (no underscore)
- Set **Shader Declaration** to `Hybrid Per Instance` (this is critical for ECS rendering)
- Set **Mode** to **Slider**
- Set **Min** to `0`
- Set **Max** to `1`
- Set **Default** to `1`

Step 4: Wire Up the Fade Effect

- Drag the `FadePercent` property into the graph
- Connect it to the **Alpha** input of the Master Stack (for a simple alpha fade)
- Or use it in more complex ways (multiply with texture alpha, use in a dissolve effect, etc.)

Step 5: Create Material

- Save the Shader Graph
- Create a new Material using this shader
- Assign the material to your entity's Renderer

6. AI Vision Integration

6.1 VisionSource on AI units

Adding `VisionSourceAuthoring` to computer controlled units is optional.
You only need to do it if you want those teams to have their own vision and if you want to integrate that vision with AI behavior.
Without `VisionSourceAuthoring`, those units do not contribute any team vision data.

6.2 VisibilitySwitch on player units

Adding `VisibilitySwitchAuthoring` to player units is also optional.
By default, this setup is mainly used to show or hide enemies depending on whether the player can see them.
But if you want to know whether enemy teams can see the player, then player units also need `VisibilitySwitchAuthoring`.
That is what makes the system write detection data for them into `VisibilityState`.

6.3 How to use it in AI

If you want to integrate your own AI with vision, read `VisibilityState` and call `IsDetectedByTeam(Team team)`.
Example:

```
bool seesPlayer = visibilityState.IsDetectedByTeam(Team.Computer1);
```

6.4 Typical setup

If you want AI teams to use vision:

1. Add `TeamAuthoring` to the entities.
2. Add `VisionSourceAuthoring` to units that should provide vision for that team.
3. Add `VisibilitySwitchAuthoring` to units or objects that should report who sees them.
4. In your AI systems, read `VisibilityState` and call `IsDetectedByTeam(...)`.

7. Examples

The package includes two example scenes that demonstrate different aspects of the fog of war system.

7.1 Example Scenes

7.1.1 Performance Test

A stress test scene designed to showcase the system's ability to handle large numbers of units.

Features:

- **Unit Count Slider** - Adjust how many units are active on the map in real-time
- **Random Movement** - Units randomly move around the map, revealing fog as they go
- **Minimap** - Shows the fog state on a minimap overlay
- **Camera Controls** - WASD to move, Q/E to zoom in/out
- **Fog Toggle** - Button to enable/disable the fog system
- **FPS Counter** - Top-right corner displays current framerate to observe performance

This scene is perfect for testing performance limits and seeing how the system handles hundreds or thousands of units simultaneously.

7.1.2 Simple Example

A practical demonstration showing typical fog of war configuration in a game scenario.

Features:

- **Player Unit** - A unit that randomly moves around the scene and reveals the map
- **Enemy Units** - Several enemy units placed on the map with fade transitions enabled
- **Fade Effect** - Enemies smoothly fade in when the player unit sees them and fade out when vision is lost

This scene demonstrates the basic setup you would use in a real game, including entity visibility and smooth fade transitions.

8. Performance

The fog of war system is built with performance in mind using Unity's DOTS (ECS, Jobs, and Burst compiler) to achieve exceptional efficiency.

8.1 Performance

The system has been tested and optimized to handle **up to 100,000 units** with smooth performance.

Watch the performance demonstration with 100k units on YouTube:

YouTube video

8.2 Quick Optimization Tips

If you need to improve performance further:

- **Texture Size** - Lower resolution (`512×512` or `1024×1024`) reduces GPU memory and rendering cost
- **Max Vision Sources** - Set this to match your actual unit count to avoid wasting memory
- **Blur Strength** - Set to `0` to skip blur passes if soft edges aren't needed
- **Vision Shape** - `Circle` is the cheapest shape; use it when possible