Tiles of War

Version 1.0

Introduction

Tiles of War provides the following features:

- Multiple targets for rendering fog of war:
    - Image Effect (Pro-only)
    - Projector
    - Plane Overlay
- Automatic or manual visibility control of:
    - Terrain
    - Units
    - Structures (non-mobile units)
- “Ghosting” of previously visible units, terrain, and structures
    - Automatic through Unity game object configuration in editor
    - Manual through use of callbacks in code
- Customizable visibility determination (can unit X see a particular tile)
    - Manually configurable through code
- Support for 100s of units (tested with over 1000 player/non-player units)
- Support for large tile-based game worlds (tested with over 1000x1000 tile worlds)
- Suitable for both Turn-Based and Real-Time games
- Configurable texture resolution and lighting strengths

Although the TilesOfWar system is designed to be used in a tile-based system, this *doesn’t* imply either real-time or turn-based game play. It works perfectly well for either. In fact, it can be used in games that aren’t innately tile-based themselves by simply assigning an arbitrary tile size (which will affect the resolution of the resulting fog image).

Again, even though it was designed with Strategy games in mind, it’s perfectly suitable for other types of games including Rogue-Likes, World Exploration, Trading Simulation, etc.

It also doesn't limit unit movement to being strictly tile-based. It has been tested using Unity navigation meshes and still functions very well.

Note: It has *not* been *tested* on any mobile platforms. We're assuming that the plane overlay target should work but this has not been verified.

Contact Us

If you have trouble with this product for any reason, please don't hesitate to contact us. We'll do whatever we can to help resolve the issue.

The best way to reach us is by email at voxel.frog@gmail.com.

Also, check out our blog at http://voxelfrog.blogspot.com/.

C# versus Javascript

This document assumes a C# development environment. If you're coding in Javascript you'll need to convert the samples from C# to Javascript yourself. That's should be pretty straight-forward, but if you run into any major problems, please contact us and we'll do our best to help.

We implemented the *FoWCallbacks* class as a Plugin (in the Assets/Plugins folder) which means that it is directly callable from Javascript or other languages.

The other classes in this package are *not* implemented as plugins. This means that they are *not* easily accessible from Javascript code (but are still usable as scripts attached to GameObjects from within the Unity Editor). If you're using the defualt behaviors of this package this should be all that is required.

If you do need to access these classes directly from code you will either need to move them manually to the Plugins folder or create "bridge" classes to communicate between the two languages. Doing so is beyond the scope of this document.

If you need assistance with this, please try the Unity support forums (for help with the general problem of communicating between scripts written in different languages) or contact us via email (for help with the problem as it specifically relates to this package).

What is "Fog of War"

This section of the document hopes to provide a brief overview of general "fog of war" related topics and some of the options you have in how you use them in your game.

If you're already familiar with this or already know exactly what you want to do, feel free to skip this section. The terms we define here will be used in the documentation that follows (and is generally reflected in the code as well) so if you get stuck on something you might want to come back and read (or re-read) this.

This discussion focuses on strategy games (real-time or turn-based) for sake of simplicity but is applicable to many other forms of games as well.

Definition

From Wikipedia:

> "The **fog of war** is the uncertainty in situational awareness experienced by participants in military operations. The term seeks to capture the uncertainty regarding one's own capability, adversary capability, and adversary intent during an engagement, operation, or campaign."
>
> (http://en.wikipedia.org/wiki/Fog_of_war)

You've seen in it just about all major real-time and turn-based strategy games for decades now: Age of Empires, Civilazation, Warcraft, etc. being some of the most common examples.

The Basics

Terrain

When we say "terrain" concerning a game we're not necessarily talking about "hills and rivers and forests and mountains". We may well be talking about deep dark dungeon tunnels, sci-fi corridors, abstract landscapes made of floating stone pillars, whatever.

We'll still call it "terrain" for sake of discussion, and for example purposes we'll stick to "hills and rivers and forests and mountains".

In some games (maybe yours!) terrain may change over time. Rivers may dry up, rocks might get mined, the enemy or player might level a forest.

Player Units

We're going to assume that the player has some form of representation (probably multiple) in the game. In most strategy games the player creates "units" that move around the playing field doing all kinds of fun stuff.

Units can usually move under direction of the player.   Some units may not be mobile (turrets or guard towers or whatever).

Units may change over time (upgrades, damage, etc.).

These are referred to as “Player Units”.

Player Structures

We’ll also assume that the player can build “structures”. Maybe these are mines or storage buildings or houses. They’re different from “units” in that they don’t move (if they do move we’ll just call them “units” instead).

These may change over time as well (upgrades or expansions, destruction, etc.).

Non-Player Units

And what fun would it be if the player didn’t have one or more adversaries or allies? (Okay, we guess you could do a fine game without enemies, but we’re going to bet you plan to have some!)

“Non-player units” are usually controlled by the computer (AI) or another player. (Even if they’re controlled a player… it isn’t *our* player… hmmm… that’s confusing, but I think you get the point.)

Non-player units might **not** be hostile; they may be NPCs or computer-controlled allies.

These non-player units can be created and destroyed as the game progresses.

They may also change over time (upgrades, damage indicators, health, whatever).

Non-Player Structures

And of course, all those nifty non-player units can also probably create their own structures. Let’s assume that these structures can be created and destroyed over the course of the game and that, like player structures, don’t generally move and may change over time as well.

Terrain or Structures?

Before we continue, let’s consider something like our “trees” example. They can be destroyed (chopped down for firewood, burned, whatever). Are they part of the terrain or are they a “structure”?

If the trees mentioned above are part of the tile, then it follows that when they are destroyed the TILE itself changes. If those same trees were a structure ON the tile, then the tree “structure” would change (or be removed) when they were cut down but the tile itself would not change.

So which is it? The answer, like most things in development, is “it depends”. We really could do either depending on what *other* aspects of trees versus terrains are going to influence our decision. For our purposes, we’re just not going to decide (we can work well with either decision). But it’s probably something you need to consider and make sure you think through.

More

So there we have the basic building blocks that we're going to be dealing with. There are *tons* of other components in a good strategy game, but these are the basics that we need to deal with for our discussion of fog of war, so we'll stick with these for now.

Options for a Fog of War Implementation

The first thing we should probably decide on is how the player views our game world "terrain" (tiles, in our case).

Can they see it all, always, instantly updated if it changes? Sure, we *could* do that… but it's boring (and we really wouldn't need "fog of war").

Let's assume that, instead, we want the world to be a big mystery to the player until they've gone out and explored it.

So, we'll start by defining some states for our terrain (tiles in our case) to be in:

- **Hidden** – never been viewed by the player. The entire world probably starts out this way. The player can't see the terrain type itself, nor can they see any units or structures inside the tile.

- **Visible** – currently in direct line-of-sight to a Player Unit. Most often this means that the player can see everything going on in the tile… unit movements, structures, etc.

- **Explored** – previously viewed by a player unit but not currently directly visible. Maybe the original viewing unit moved away or was destroyed, but the player no longer has direct line-of-site to this part of the world (tile).

Hidden

Most frequently in strategy games, the player probably doesn't know anything about a tile that is "Hidden". It's not even shown on the screen. Any units or structures in the tile are hidden as well.

Visible

But what should we do when a player unit moves so that it can see a particular tile?

Let's set its state to "Visible" and show the player the REAL contents of the tile (the terrain and any units/structures).

They can see units and structures as they really are (the "real" version). If units, terrain, or structures change while they're still visible we'll show the player that as well.

Explored

But what happens when the player moves away (or the player unit is destroyed) and the player can no longer see a tile?

This is where we start to have lots of options for how we deal with what the player sees.

We *could* set it back to Hidden. That's certainly an option. Then we just have to hide everything in the tile. It's probably the simplest option. If this is what we're doing, though, we really wouldn't have an "Explored" state, so let's assume we're not doing this.

We want something a little more sophisticated, so let's mark it as "Explored". This is probably the most common way that many games handle fog of war.

What does that mean to the player? The simplest (and probably most common) thing to do is to continue to display the terrain and unit/structure contents that the player last knows about… "Ghosts", if you will.

Maybe after the player moves away the enemy units move, the structure burns down, and the forest disappears as well (we'll blame that one on wizards or something)… BUT the player DOES NOT know any of that! They continue to see the units and structures standing in a lovely forest. It's only when they move back to view that tile that we want them to see the changes.

They're not seeing the "real" units/structures/terrain… they're seeing "Ghosts"! Another way of looking at it is that we're acting as the player's memory of what they think is in the tile.

We have a few options here as well, of course: we can "ghost" the terrain, units, and structures independently. For example, we may want them to always know when the terrain updates, we may want to hide non-player units entirely unless they're directly visible, or we may want to show the structures being updated.

A Little More Complication

Given what we've just said, it seems pretty simple (and it is).

A few other "complications" you might want to consider are whether "all non-player units are treated equally". What if, for example, we want to have true "hostile" units NOT appear in Explored areas, but have "friendly" non-player units APPEAR in Explored areas.

What if we want to treat "friendly" non-player units as "so friendly that we can see everything they see"?

And we could probably come up with quite a few other examples of "complications" to our nice simple system.

Luckily all of this is possible using the terms we've already discussed.

If we want to see "friendly" non-player unit territory… treat them as Player Units as far as setting tiles to Visible/Explored/Hidden instead!

If we want to see "friendly" units in explored territory, but not see "enemy" units in explored territory we just need to make sure the code handles both options (Tiles of War handles these options independently for each entity, so it's easily done).

What We Don't Handle

There are probably hundreds of options concerning fog of war that we didn't discuss here. Further, we probably don't address or handle them in Tiles of War "out of the box".

One notable example is "fading Explored areas". If we did this, a player would eventually "lose his/her memory" of an explored area… fading slowly from Explored to Hidden. We could have implemented that for this system but we chose not to. If it's something you're interested in, don't hesitate to let us know and maybe we can at least point you in the right direction.

Summary

So, with all that said here's what we have as far as the options we're going to consider (there certainly might be others, of course):

	Hidden	**Visible**	**Explored**
Player Unit	*(not applicable)*	Show "Real"	*(not applicable)*
Player Structure	*(not applicable)*	Show "Real"	*(not applicable)*
Terrain	Don't Show Anything	Show "Real"	**Option 1: Real** **Option 2: Ghost**
Non-Player Unit	Don't Show Anything	Show "Real"	**Option 1: Real** **Option 2: Ghost** **Option 3: Nothing**
Non-Player Structure	Don't Show Anything	Show "Real"	**Option 1: Real** **Option 2: Ghost** **Option 3: Nothing**

Player Units are easy: we always show the player the real version and they're never in "Hidden" or "Explored" territory (it's always "Visible" around the player units).

Likewise, everything that is in a "Visible" tile is easy: always show the "real" version of the terrain and whatever units or structures are in that tile since the player can actively see it.

And everything that is "Hidden"… well, it's hidden so we just don't really want to show it.

As you can see the only real options we have to consider are in the "Explored" state.

The remainder of this document will cover how to implement the above options in Unity using Tiles of War.

Internal Coordinate System

Before we start integration the fog of war system into your project, it’s important that you understand the coordinate system that we use.   We’re assuming here that you already have a good grasp of the Unity native coordinate system (which way X, Y, and Z increases).

For TilesOfWar we MATCH Unity’s coordinate system (for the most part).   That means that as Unity’s X coordinate increases, our tile X coordinates increase, and as Unity’s Z coordinate increases, our tile Z coordinates increase.

We use “Z” instead of “Y” since it matches Unity’s system most naturally.

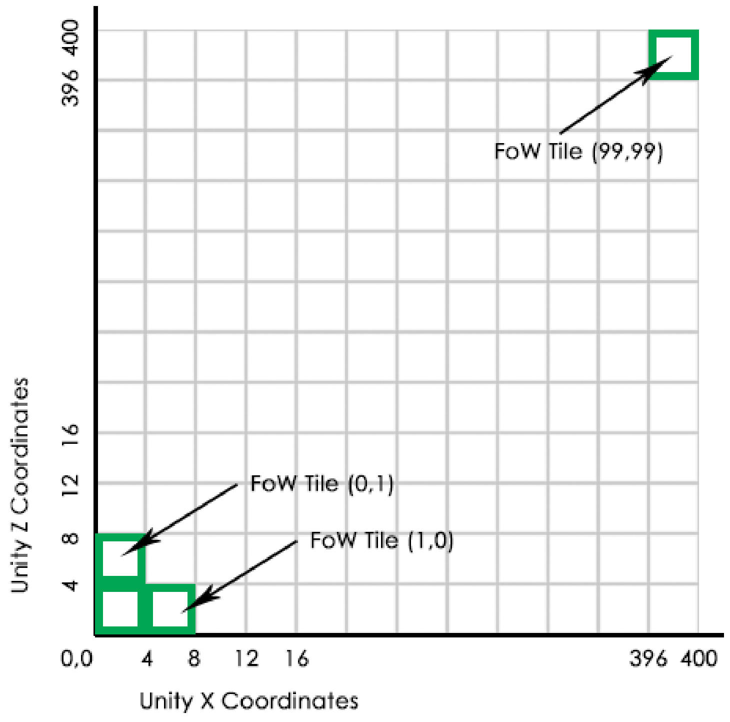
For example, let’s say you set up the FoWManager with:

WorldUnitsPerTileSize = 4 (Unity units per tile side)

Bounded in Unity world space from WorldMin=(0,0,0) to WorldMax=(400,10,400)

What that tells us is that you have a 100 x 100 tile grid with a maximum height of 10 Unity world units. Note that the only thing we use the “height” for is setting up the fog target (either the FoWPro shader, projector, or plane overlay) to make sure that it covers the entire world including height.

So, given that, internal to FoWManager tile (0,0) covers the world area (0,0,0) to (4,10,4).   Tile (0,1) covers world area (0,0,4) to (4,10,8).   Tile (1,0) covers world area (4,0,0) to (8,10,8).   Tile (99,99) covers world area (396,0,396) to (400,0,400).



If the world area of your game is offset (doesn't start at (0,0,0)), you can simply offset the WorldMin and WorldMax values appropriately. If, for example, the same world above was centered at the Unity origin (0,0,0) we'd set WorldMin = (-200,0,-200) and WorldMax = (200,10,200).

We chose this coordinate system for simplicity (for our internal development) since it allows world positions of units, structures, tiles, etc. to be much more easily and naturally translated to tile coordinates. We also suspect that the majority of tile-based games written natively for Unity use a similar system, but of course we could be very wrong about that.

Translating Coordinates

If your tile coordinates mimic Unity's native coordinate system directly in the same way that we describe above, you can skip this section if you want.

It's also important to understand that the way WE organize tiles internally doesn't mean that YOU have to organize your tiles in the same way… you can use whatever coordinate system you want and translate between the two when calling (or receiving calls from) FoWManager.

Let's have another example…

Let's say that your world is still 100 by 100 tiles (still with 4 world units per side) and that you have your tiles running "positive-X / negative-Z" and that the entire world is centered at the origin. That would mean that YOUR tile (0,0) covers Unity world coordinates (-100,0,100) to (-96,0,96)… and that YOUR tile (99,99) is at Unity world coordinates (96,0,-96) to (100,10,-100). You can see that as YOUR tile X coordinate *increases*, the Unity world coordinate also *increases* (positive-X) by the size of your tile. Contrast that with the fact that as YOUR tile Z coordinate *increases*, the Unity world coordinate *decreases* (negative-Z).

Another way of looking at this is that if you were standing at the Unity origin and facing in the Unity positive Z direction (with the position X direction to your right): In the case above, YOUR tiles would start in the upper left and run to the lower right… while OUR internal tiles would start in the lower left and run to the upper right. Same X… different Z.

So what exactly does that mean to your game? Read on…

Receiving Tile Coordinates FROM FoWManager

Given the example above, when you receive a tile coordinate FROM FoWManager (for example, when you're being notified that a tile has become visible), you would want to:

```
public void OnTileBecomesVisible(int tileX, int tileZ)
{
   var myTileX = tileX;                        // we're both positive-X, no change
   var myTileZ = (MyNumTilesZ - 1) - tileZ;    // different tile Z directions

   // do something with myTileX and myTileZ
}
```

Notice that the fact that YOUR world isn't centered doesn't matter here… we're just translating tile coordinates to tile coordinates.

Sending Tile Coordinates TO FowManager

You'll make the opposite coordinate translation anytime YOU CALL FoWManager passing in tile coordinates.

When you CALL FoWManager (for example, to check the visiblity status of a tile for some reason) you'll just do the opposite:

```
_fowManager.GetCell(myTileX, myTileZ + (MyNumTilesZ-1));  // different Z direction
```

Sending / Receiving WORLD Coordinates

You *DO NOT* need to translate world coordinates when you pass in or receive UNITY WORLD COORDINATES.

We make that translation internally.

Fog Types Overview

This package supports three mechanisms for displaying the fog on your playing field.

When setting the Fog Type parameter in the FoWManager script the correct fog type component will automatically be added to your game.

ProShader (Unity Pro Only)

The fastest and highest quality is using the supplied FoWPro image effect.

The FoWPro image effect is usuable only with Unity Pro.

The FoWPro image effect will automatically be added to your main camera when your game runs.

If you are using multiple cameras that need to display the fog of war you will need to manually add the component to each camera.   The component is located in the Unity editor menu: **Component/Fog of War/FoWPro Shader**.

Projector (Unity and Unity Pro)

Projector-based fog is implemented using a standard Unity shadow projector.

If you select the Projector setting for FogWManager.FogType the projector will automatically be enabled when your game runs.

Note that based on how projectors work they may cause a more drastic frame-rate hit than the ProShader option.

Plane Overlay (Unity and Unity Pro)

Overlay-based fog is implemented by positioning a single plane over your playing field.

The position is based on the FoWManager WorldMin and WorldMax settings.

If you select the Overlay setting for FoWManager.FogType the overlay plane will automatically be enabled when your game runs.

Integration into Your Projects

In this section we're going to walk you through adding TilesOfWar to your project. For simplicity we're going to use the default behavior of the elements to begin with, then we'll delve into configuring different behaviors.

We use a "callback" system (C# Funcs/Actions) which we describe in the next section.

The FoWCallback class is implemented in the Plugins directory so that it can easily be called from Javascript (or whatever language you're using) as well as C#.

Integration - The Short Version

If you're an experienced developer you may not need (or want) a long-winded explanation of how to add this system to your project.

The short version:

1. Add the FogOfWar prefab to your scene and set the FoWManager script values appropriately through the Unity Inspector
2. Add the FoWPlayerUnit to each of your player unit or structure GameObjects
    a. Set the view radius of each instance (or prefab)
3. Add the FoWNonPlayerUnit to each of your non-player unit or structure GameObjects
    a. Set the "Auto Manage Renderers" value of each instance (or prefab)
    b. Set the "Show in Explored" and "Show In Hidden" values however you'd like
4. Add the FoWGhost to your non-player unit ghosts (if you're using ghosts)
    a. **Important:** you'll need to use the **OnAddGhost** and **OnRemoveGhost** callbacks to create and destroy ghosts
5. From your own code, set the optional FoWCallbacks callbacks for VisibilityTest, OnTileFirstVisible, OnTileBecomesXXX, OnNonPlayerUnitBecomesXXX, OnAddGhost, OnRemoveGhost, etc. You may or may not need any or all of those… it all depends on what behavior you want.
    a. You can find the FoWCallbacks instance easily from your code by calling the "FoWCallbacks.FindInstance()" method

If any of that didn't make sense, keep reading…

Basic Setup

We're assuming that you already have a working game (or at least prototype) where you have coded the terrain and units. We strongly recommend that you only integrate this package into your project once you have the basics of your game already working.

To get started:

- If it isn't already in your project, import the TilesOfWar package into your Unity project:
    a. From the Unity menu: Assets -> Import Package -> Custom Package

Adding FogOfWar - FoWManager

- Add the FogOfWar prefab to your scene. It doesn't matter where you place it since it moves itself based on how you configure WorldMin and WorldMax values.
    a. In the project window: Assets/TilesOfWar/Prefabs/FogOfWar

If you take a look at the prefab, it's pretty simple. It's a parent GameObject with the FoWManager and FoWCallbacks scripts attached and a children named "FoWProjector" (used if using projector-based fog) and "FoWOverlay" (used if using overlay-based fog).

- Configure the FoWManager script:
    a. **Fog Type**: set to the mechanism you wish to use for displaying the fog.
        i. **ProShader** – use this if you have Unity Pro. It will result in the best performance and visual quality.
        ii. **Projector** – use this if you do *not* have Unity Pro. This will provide good visual quality but may affect performance if you have many renderers under the fog.
        iii. **Overlay** – use this if performance is not adequate using Projector or ProShader.
    b. **World Min**: set to the minimum world coordinates that represent your game world
    c. **World Max**: set to the maximum world coordinates that represent your game world
        i. Note: make sure to set the Y vector component to something *above* the highest point in any of your tiles
    d. **World Units Per Tile Side**: set to the number of Unity units per tile
- Once you have everything working in your game, you can come back and play with the FoWManager settings to get the "look" that you want (light strengths, etc.)
    a. **Texture Resolution** – range from 1 to 4. Sets the number of pixels (per tile side) for the generated fog texture. Larger values will increase texture memory usage.

b. **Texture Point Filter** – if set, the generated fog texture will be point filtered. If you choose this option (mainly for "stylistic" purposes) you'll definitely want to set Texture Resolution to 1 to save some memory.

c. **Visible Light Strength Max** – the maximum strength of light (0.0-1.0) for a Visible tile. Unless you have a particular need to "darken" all Visible tiles you should probably leave this set to 1.0

d. **Visible Light Strength Min** – range 0.0 – 1.0. Only used if "Fade Visibility" is set (see below)

e. **Explored Light Strength** – light strength (0.0-1.0) for "Explored" tiles.

f. **Hidden Light Strength** – light strength (0.0-1.0) for "Hidden" tiles. Usually you'll leave this at 0.0

g. **Fade Visibility** – if set, visible areas fade away slightly as they get further from an active Player Unit. The light will fade per tile between VisibleLightStrengthMax and VisibleLightStrenthMin

h. **Show Explored** – if set, the system will use the "Explored" tile state. If not set, tiles will either be "Hidden" or "Visible".

i. **Initial Max Viewers** – the number of initial "slots" for player units. If you know you'll commonly reach a certain value, set to this. You can certainly leave it at it's default value (it grows as needed).

Run your application. If everything is set up correctly you should see everything "black" (since we haven't added any player units (viewers) to the game yet).

Player Units (Viewers) - FoWPlayerUnit

Each of your player units, whether Structures or Units (or whatever you want to call them), needs to have a **FoWPlayerUnit** script attached.

The FoWPlayerUnit script is what allows us to track where all of your units are so that we can update the fog as they move around.

The FoWPlayerUnit script only has one setting:

- **View Radius In Tiles** – number of tiles from the viewer that should be marked as "Visible"

If you have one or more prefabs for your player units (we're betting you do) simply add the FoWPlayerUnit script to each of your unit prefabs.

If you are creating your units dynamically (not from a prefab), just add the FoWPlayerUnit script when each unit is created.

The FoWPlayerUnit script communicates with the FoWManager script to let it know where your player's are and what they can see.

Integration - Advanced

Non-Player Units – FoWNonPlayerUnit

Each of your non-player units, whether Structures or Units (or whatever you want to call them), needs to have a **FoWNonPlayerUnit** script attached.

This lets the system know where each unit is and coordinate creating of Ghosts (see below), hiding units, etc. when the tile that the unit is in changes visibility state (or the unit moves to a different state tile).

Set the values of the script to get the unit behavior you want.   Note that you can set the values differently on each unit or unit type.

- **Create Ghosts** – if set, the system will callback to you when a "ghost" needs to be created
    a. **IMPORTANT:** If you set "Create Ghosts", you'll also need to set up the **OnAddGhost** and **OnRemoveGhost** callbacks (see below)
- **Auto Manage Renderers** – if set, the system will automatically show and hide units based on the Show In Explored and Show In Hidden.   This is the simplest way to use the system.
- **Show In Explored** – if set (and Auto Manager Renderers is also set) the system will automatically show the non-player unit when it is in Explored territory.   We'll discuss this more below.
- **Show in Hidden** – if set (and Auto Manager Renderers is also set) the system will automatically show the non-player unit when it is in Hidden territory.   We're doubting you want this set, but it's here just in case.

To have non-player units and structures APPEAR in explored areas you can simply set "Show in Explored" to true.   Otherwise, leave it unchecked.   Likewise for "Show in Hidden".

Important: If you want to manage *everything* yourself, don't set "Auto Manage" and use the OnNonPlayerUnitBecomesXXX callbacks instead.   This is flexible but requires the most work on your part.

Important: If you use the "Auto Manage Renderers" feature, we'll disable ALL *renderers* under the GameObject containing the FoWNonPlayerUnit script.   It will ***not*** disable animations or other scripts that *might* need to be disabled (for performance reasons or based on whatever it is you're doing on your units).   If you need to do this, you'll need to use the callbacks as mentioned above.

Non-Player Ghosts – FoWGhost

If you want ghosts to be managed automatically for you, the system will use the OnAddGhost and OnRemoveGhost callbacks (see below).

You configure whether each Non-Player Unit should create ghosts in the FoWNonPlayerUnit script that you attach to the unit (or prefab).

We'll discuss the callbacks more in detail below.

Important: Any entity that you return from OnAddGhost should have the **FoWGhost** script attached.

The FoWGhost script has no settable parameters.

If you want to manage ghosts manually, *don't* set "Create Ghosts" on any of the FoWNonPlayerUnit scripts and use the OnTileBecomesVisible/Explored/Hidden callbacks instead. Doing this manually is beyond the scope of this document.

Terrain Ghosts – Using Callbacks

If you don't want to support terrain "ghosts" or if your terrain doesn't change during gameplay, you can skip this section.

Rather than making any assumptions about how you're representing your game play area ("Terrain"), pathfinding, etc. we allow you complete control by only providing callbacks to let you know when a terrain tile should change states.

This allows you complete control over what does or doesn't happen within your game's terrain space.

Unfortunately, this also means that you're going to have to do some work to get terrain "ghosting" to work properly… but don't worry, it isn't hard at all.

We'll talk more about how to set up the callbacks in the next section, so for now just read along.

The **OnTerrainBecomesVisible** callback is called anytime a tile becomes Visible (hence its not-particularly-clever name). When this is called you can be sure that *some* player unit or structure can directly view the tile. This may be because a new unit or structure is created or because one moves into a nearby tile (or even "teleports" to this tile).

When OnTerrainBecomesVisible is called you will probably want to set the tile completely visible, including any special particles or animations. If you've displayed a "ghost" of tile, this is where you will hide or destroy the ghost.

Important: If you're using the default behavior of the FoWNonPlayerUnit scripts on each of you non-player units, you DO NOT need to set them visible or deal with unit ghosts. Any units that are in the tile will also receive a OnNonPlayerUnitBecomesVisible callback or be handled automatically (depending on how you've set them up). *This applies to the terrain/unit Explored and Hidden callbacks as well.*

The **OnTerrainBecomesExplored** callback is called when a tile becomes Explored, for whatever reason (usually a unit or structure that could previously see the tile moving away or being destroyed).

When a tile becomes explored, you will probably want to store a separate copy of what it looks like to the player… the "ghost" and hide (turn off renderers) or disable the original "real" tile completely. Then, when the tile becomes visible again, you'll do the opposite.

The **OnTerrainBecomesHidden** callback is really only provided to handle edge cases where you may be resetting terrain visibility manually. During most games (and with the default behavior of this system) this *should* never be called. When we’re using this system we usually just assign it to the same method/function as we’ve set OnTerrainBecomesExplored.

Callbacks

TilesOfWar uses Funcs and Actions as “callbacks” into your code.   This generally performs faster than SendMessage or .net Events.

Generally, Actions (which do not have return values) are used to *notify* you that something interesting has happened or to ask you to do something.   Funcs (which have return values) are used to *ask* you for some value.

We’d recommend adding your own script where you can configure any callbacks that you need.   A Sample starter script is shown below.

```
public class MyFoWCallbacks : MonoBehaviour
{
    public void Startup()
    {
        var callbacks = FindObjectOfType(typeof(FoWCallbacks)) as FoWCallbacks;
        if (callbacks != null)
        {
            callbacks.VisibilityTest = CheckTileVisibility;

            callbacks.OnTileBecomesVisible = OnTileBecomesVisible;
            callbacks.OnTileBecomesExplored = OnTileBecomesExplored;
            callbacks.OnTileBecomesHidden = OnTileBecomesHidden;

            callbacks.OnNonPlayerUnitBecomesVisible = OnEnemyVisible;
            callbacks.OnNonPlayerUnitBecomesHidden = OnEnemyHidden;
            callbacks.OnNonPlayerUnitBecomesExplored = OnEnemyExplored;

            callbacks.OnAddGhost = AddGhost;
            callbacks.OnRemoveGhost = RemoveGhost;
        }
    }

    private GameObject AddGhost(GameObject theNonPlayer)
    {
        // create a "ghost" of the specified
        // non-player game object
        return null;
    }

    private void RemoveGhost(GameObject theNonPlayer, GameObject theGhost)
    {
        // destroy theGhost and clean up any
        // of your own data as neeeded
    }

    private void OnEnemyExplored(GameObject theNonPlayer)
    {
        // do whatever you need to do when a
        // non-player unit enters an "explored" part of the map
    }
    private void OnEnemyHidden(GameObject theNonPlayer)
    {
        // do whatever you need to do when a non-player
        // unit enters an "hidden" part of the map
    }

    private void OnEnemyVisible(GameObject theNonPlayer)
    {
        // do whatever you need to do when a non-player
        // unit enters an "visible" part of the map
    }

    private void OnTileBecomesHidden(int tileX, int tileZ)
    {
        // do whatever you need to do when a tile
        // becomes hidden
    }

    private void OnTileBecomesExplored(int tileX, int tileZ)
    {
        // do whatever you need to do when a tile
        // becomes explored
    }

    private void OnTileBecomesVisible(int tileX, int tileZ)
    {
        // do whatever you need to do when a tile
        // becomes visible
    }

    private bool CheckTileVisibility(int fromTileX, int fromTileZ,
                                     int toTileX, int toTileZ)
    {
        // return true if tile (toTileX,toTileZ)
        // is visible from (fromTileX,fromTileZ)
        // otherwise, return false
        return true;
    }
}
```

We’ll talk more about each of the callbacks below.

Note that you can use both the callbacks *and* the FoWNonPlayerUnit auto render management without problem (you just won’t need to manage the renderers yourself, but you can do other activities inside the callbacks).

Tile Callbacks (Actions)

The system will call you whenever a tile changes state… when it FIRST becomes visible, when it changes to either “Hidden”, “Explored”, or “Visible”.

- **OnTileFirstVisible** – called whenever a tile FIRST becomes Visible.

- **OnTileBecomesVisible** – called whenever a tile becomes Visible (from either Hidden or Explored)

- **OnTileBecomesExplored** – called whenever a tile becomes Explored (usually from Visible)

- **OnTileBecomesHidden** – called whenever a tile becomes Hidden (not usually called unless you're manually setting tiles to Hidden state)

Each of the above take the FoW internal tile coordinates (see notes above about coordinate translation) as parameters.

Note that the first time a tile becomes visible **both** "OnTileFirstVisible" and "OnTileBecomesVisible" will be called.

```
    public Action<int, int> OnTileFirstVisible = null;
    public Action<int, int> OnTileBecomesVisible = null;
    public Action<int, int> OnTileBecomesExplored = null;
    public Action<int, int> OnTileBecomesHidden = null;
```

Non-Player Callbacks (Actions)

The "OnNonPlayerUnitBecomesXXX" callbacks let you know when any of your non-player units enter a tile of a different status OR the tile they are currently in changes.

- **OnNonPlayerUnitBecomesVisible** – called whenever a GameObject containing the FoWNonPlayerUnit script enters a Visible tile (or the tile state changes to Visible). The actual GameObject will be passed as a parameter.

- **OnNonPlayerUnitBecomesExplored** – called whenever a GameObject containing the FoWNonPlayerUnit script enters an Explored tile (or the tile state changes to Explored). The actual GameObject will be passed as a parameter.

- **OnNonPlayerUnitBecomesHidden** – called whenever a GameObject containing the FoWNonPlayerUnit script enters a Hidden tile (or the tile state changes to Hidden). The actual GameObject will be passed as a parameter.

You don't have to use these, of course, especially if you're using the default configuration and Auto Manage Renderers.

```
    public Action<GameObject> OnNonPlayerUnitBecomesVisible = null;
    public Action<GameObject> OnNonPlayerUnitBecomesExplored = null;
    public Action<GameObject> OnNonPlayerUnitBecomesHidden = null;
```

Ghost Callbacks (Func/Action)

Whenever the system needs to create or remove a ghost for a non-player unit (which may include structures) it calls back to your code to provide the ghost (or remove it).

- **OnAddGhost** – called whenever the FoWManager determines that a non-player unit containing the FoWNonPlayerUnit script (and with CreateGhosts set) has moved "out of sight" of a player unit. The non-player GameObject is passed as a parameter. Your callback method should return the ghost GameObject.

- **OnRemoveGhost** – called whenever the FoWManager determines that a ghost instance needs to be removed. Both the original GameObject that the ghost represents (first parameter) and the Ghost GameObject itself (second parameter) are passed to you.

Ghosts are removed whenever:

  - A Ghost has become "visible" to a player (and as such the player now knows that the ghost isn't real)

  - The original non-player unit that the ghost represents becomes "visible" (and as such the player now knows that the ghost isn't real / the unit is no longer in the old "remembered" position)

This allows you complete control over how Unity GameObjects get created in your game. For example, you may or may not be using an object pooling mechanism and would not want us to create or destroy the units directly.

```
    public Func<GameObject, GameObject> OnAddGhost = null;
    public Action<GameObject, GameObject> OnRemoveGhost = null;
```

OnAddGhost is called (as you can probably guess) when the system needs a ghost created. The same method will be called regardless of the unit type, which is why the original GameObject that we're "ghosting" is passed to you. You can use this to determine the type of ghost to create (or even whether or not to create a ghost for that unit type). How you do that is up to you.

Return "null" if you do *not* want a ghost created for that unit type.

```
    private GameObject AddGhost(GameObject theNonPlayer)
    {
        // create a "ghost" of the specified
        // non-player game object
        return null;
    }
```

OnRemoveGhost is called (again, rather obviously) when the system needs a ghost removed. That lets you return it to an object pool or whatever management activities you need to perform.

```
    private void RemoveGhost(GameObject theNonPlayer, GameObject theGhost)
    {
        // destroy theGhost and clean up any
        // of your own data as neeeded
    }
```

Visibility Testing (Func)

By default the system considers everything in a radius around a player unit to be visible. It doesn't take the type of terrain or structures into consideration. This is "good enough" for a lot of games, so it's the default behavior.

Given the sheer number of different ways you could be representing your terrain and structures and how you might want to determine visibility, we chose to provide you with an optional callback mechanism for letting the system know whether a particular tile is visible from some other tile.

```
  public Func<int, int, int, int, bool> VisibilityTest;
```

You can set this in whichever of your own scripts you like, most likely in the Unity Startup method. If you're using other callbacks we'd recommend setting them all in one place but it's entirely up to you of course.

Important: this callback will be called for *each* tile that is within view of a game object containing the FoWPlayerUnit script on *each frame*… which is *very frequently.* If you implement this you will need to take care that it performs well or your game's overall performance may suffer.

If you want the default behavior, don't set this value, just leave it "null".

Example:

```
    private bool CheckTileVisibility(int fromTileX, int fromTileZ,
                                     int toTileX, int toTileZ)
    {
        // return true if tile (toTileX,toTileZ)
        // is visible from (fromTileX,fromTileZ)
        // otherwise, return false
        return true;
    }
```

Layers (Optional)

If you are not using the Projector fog type, you can skip this section.

If you are using the Projector target and don't want the fog projected onto certain components of your game, you're going to need to configure the projector's Ignore Layers value:

- Select the FogOfWar component in your game, and then select the underlying FoWProjector. Set the “Ignore Layers” value to the layers in your game that you do NOT want the fog projected onto.

Unfortunately, the FoWPro shader and plane overlay do not have a mechanism for ignoring particular layers.

Particles

If you are not using the Projector fog type, you can skip this section.

Unity projectors do not work with particles, unfortunately.   This means that particle systems in Hidden or Explored territory won’t be lit correctly (they “shine through”).

If you have particles on your units or structures (or even your terrain) we recommend that you disable them when their tile is Hidden or Explored.

You can do this by using the **OnTileBecomesExplored** and **OnTileBecomesHidden** callbacks or **OnNonPlayerUnitBecomesExplored** and **OnNonPlayerUnitBecomesHidden** callbacks (see above).

A Brief Note about the Demo

The demo code included with Tiles of War should not be considered a “shining example of game design”.   We’re pretty sure you weren’t going to, but we just thought we’d say it just in case.

It isn’t meant to be a great game… in fact it would be hard to consider it a “game” at all… it’s just a simple demo.

Much of the code in the demo is “seat of the pants” and is neither well-designed nor particularly efficient.

A ***lot*** (well, most) of the code in the demo isn’t something you’d need to (or want to) do in a real game… it’s there to support “playing around with” the FoWManager settings.

Follow this document and use the demo as a reference when needed and you should be fine. Again, if you run into trouble don’t hesitate to contact us!

With that said, though, you’re more than welcome to use it for whatever you’d wish.

Release Notes

Version 1.0 – Initial Release