Runtime Monitoring

Runtime Monitoring is an easy way to monitor the value or state of C# members during runtime. Just add the `[Monitor]` attribute to a field, property, event or method and get its value or state displayed automatically in a customizable and extendable UI.

⚠ **Attention!**

> Runtime Monitoring is developed by me in my spare time, which means I can only do limited QA. If you have any questions, feature requests or encounter any issues, feel free to contact me directly and/or open an Issue on GitHub. Please keep in mind that I am developing this tool by myself and that any additional information will help me a lot, especially when reporting a bug.

Table of Contents

- Support Me ♡

Installation and Updates

⚠ Important for version updates!

> Don't forget to remove the old version from the project before importing the new one when updating Runtime Monitoring! This is especially important when updating to version 3.0.0.

Option 1. **Install via Open UPM (recommended)** openupm v3.3.0

- open `Edit/Project Settings/Package Manager`
- add a new Scoped Registry:

```
- Name: OpenUPM
- URL:  https://package.openupm.com
- Scope(s): com.baracuda
```

- click `Save`
- open `Window/Package Manager`
- click `+`
- click `Add package by name...`
- paste and `Add com.baracuda.runtime-monitoring`
- this will automatically install `com.baracuda.thread-dispatcher` as a dependecy
- take a look at Setup to see what comes next

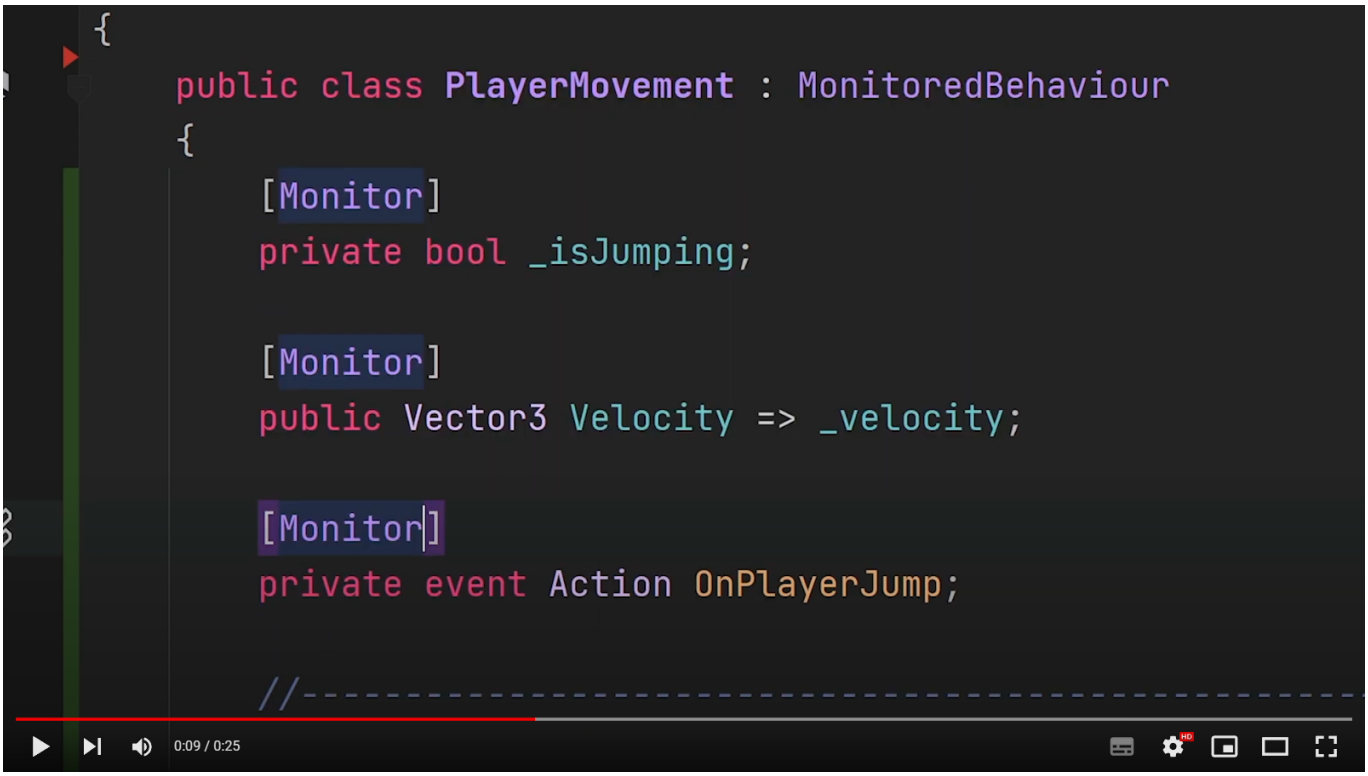
Option 2. Install via Git URL

- open `Window/Package Manager`
- click `+`
- click `Add package from git URL`
- paste and `Add https://github.com/JohnBaracuda/com.baracuda.thread-dispatcher.git` (dependency)
- paste and `Add https://github.com/JohnBaracuda/com.baracuda.runtime-monitoring.git`
- take a look at Setup to see what comes next

Option 3. Get Runtime Monitoring from the Asset Store

Option 4. Download a `.unitypackage` from Releases

> If you like runtime monitoring, consider leaving a good review on the Asset Store regardless of which installation method you chose.



Getting Started

```
using Baracuda.Monitoring;

// Monitor any field, property, event or method during runtime!

[Monitor]
private int healthPoints;

[Monitor]
public int HealthPoints { get; private set; }

[Monitor]
public int GetHealthPoints() => healthPoints;

[Monitor]
public event Action OnHealthChanged;
```

> ⚠ When monitoring instance (non static) member, objects of these classes must be registered when they are created and unregistered when they are destroyed. Learn More

```
// Register & unregister objects with members you want to monitor.
```

```
// This process can be simplified / automated (Take a look at Monitoring Objects)

public class Player : MonoBehaviour
{
    [Monitor]
    private int healthPoints;

    private void Awake()
    {
        Monitor.StartMonitoring(this);
        // Or use this extension method:
        this.StartMonitoring();
    }

    private void OnDestroy()
    {

        Monitor.StopMonitoring(this);
        // Or use this extension method:
        this.StopMonitoring();
    }
}
```

```
// Monitor static member as well as instance member

[Monitor]
public static string playerName;

[Monitor]
protected static bool IsPlayerAlive { get; set; }

[Monitor]
internal static event Action<int> OnScoreChanged;


// Use conditions to determine if a member is displayed or not.

[Monitor]
[MShowIf(Condition.CollectionNotEmpty)]
private Stack<string> errorMessages { get; }


// Reduce update overhead by providing an update event.

[Monitor]
[MUpdateEvent(nameof(OnPlayerSpawn))]
public Vector3 LastSpawnPosition { get; set; }

public static event Action<Vector3> OnPlayerSpawn;
```

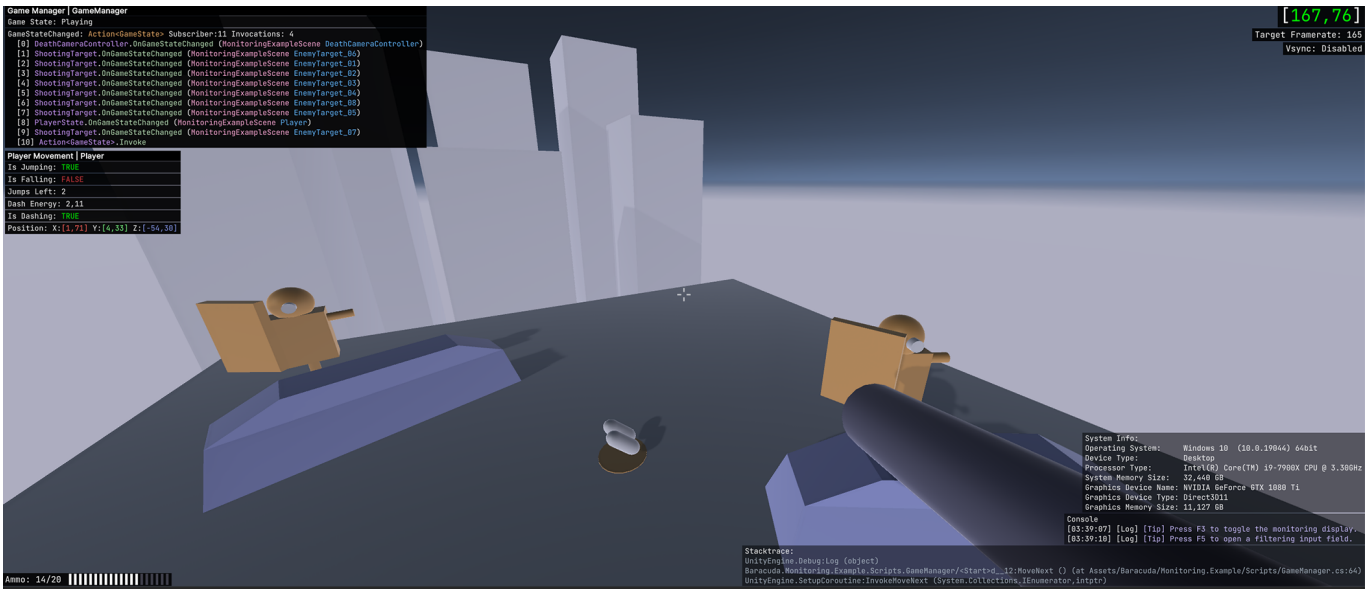
```
// Use processor methods to customize how the value is displayed.

[Monitor]
[MValueProcessor(nameof(IsAliveProcessor))]
public bool IsAlive { get; private set; }

private string IsAliveProcessor(bool value) => value? "Alive" : "Dead";


// Monitor out parameter value.

[MonitorMethod]
public bool TryGetPlayer(int playerId, out var player)
{
    // ...
}
```



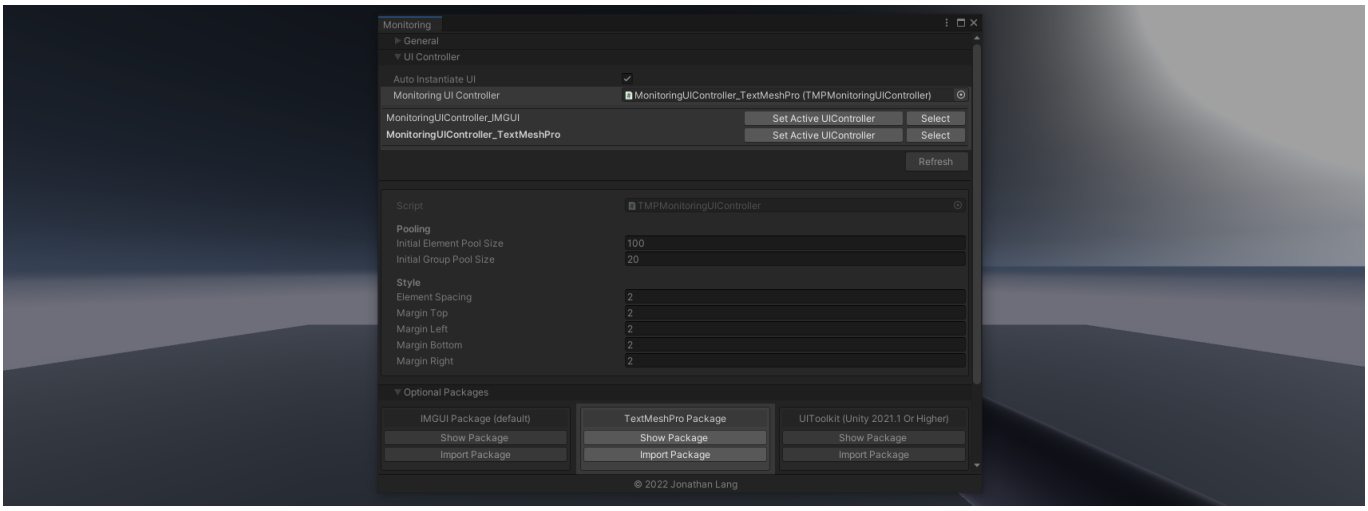
Customized Setup

> Note that since version 3.0.0, Runtime Monitoring is a UPM package and therefore immutable. Use the samples section in the package manager window to import UI resorces for IMGUI, TextMeshPro and UIToolkit. Parts of this documentation will be updated soon.

Download and import Runtime Monitoring. To setup a different UI Controller (IMGUI, TMP or UIToolkit) follow these optional steps:

- Open the settings by navigating to (menu: Tools > Runtime Monitoring > Settings).
- Depending on the Unity version and your preferences, import and optional UIController package.
- Set the prefab as the active UI Controller.
- The inspector of the set UI Controller object will be inlined and can be edited from the settings window.

> If you drag and drop a UIController asset into your scene, this controller will be used instead.



License

MIT You can use this tool for anything you want, including commercial products, as long as you're not just selling my work or using it for some other morally questionable or condemnable act.

Technical Information

- Unity Version: **2019.4** (for UIToolkit **2020.1**)
- Scripting Backend: **Mono & IL2CPP**
- API Compatibility: **.NET Standard 2.0 or .NET 4.x**
- Asset Version: openupm v3.3.0 release v3.3.0

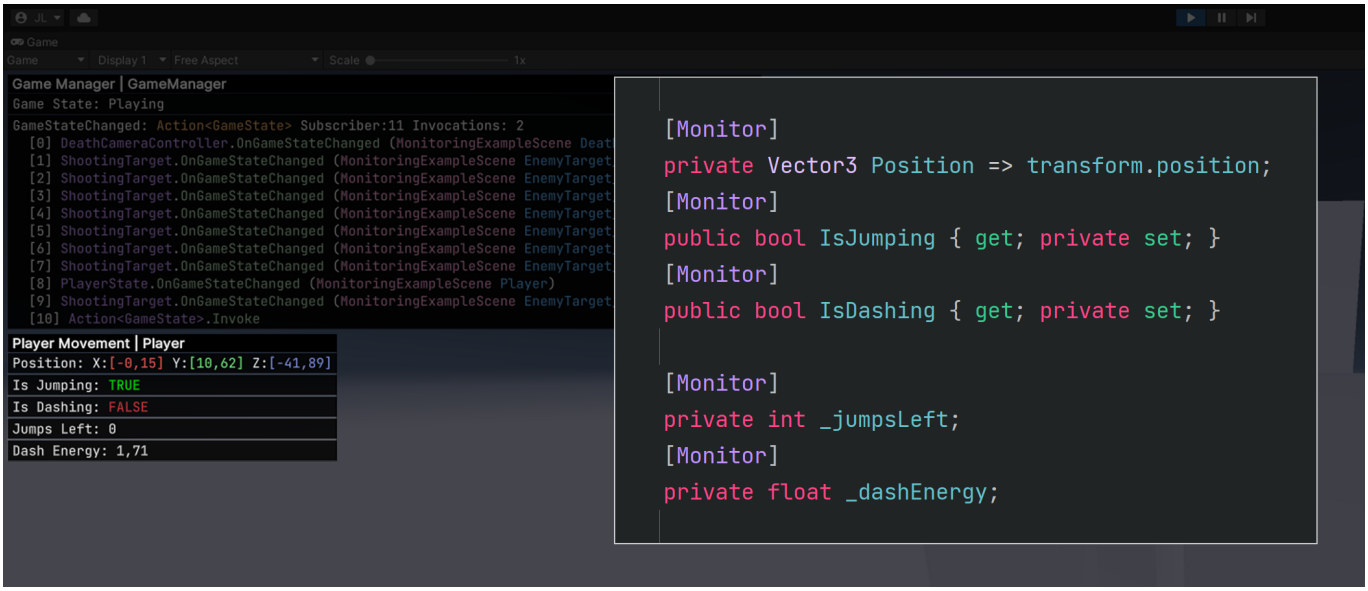
Features

- Monitor the value of a Field.
- Monitor the return value of a Property.
- Monitor the state of an Event.
- Monitor the return value & out parameter of a Method.
- Monitor static and instance member.
- Display Collections in a readable way.
- Chose one of three available UI solution presets from samples.
- IMGUI support (default).
- TextMeshPro based uGUI support.
- UIToolkit support.
- Detached UI Interface for custom UI solutions.
- Apply filter to displayed units.
- Custom control of how monitored members are displayed.
- Works both asynchronous and synchronous (WebGL supported).

- Mono & IL2CPP support.
- Drag & drop example modules. (FPSMonitor, ConsoleMonitor etc.)
- Draw conditions. (only show value if true, not null etc.)

Monitoring Member

Place the `[Monitor]` attribute an a field, property, event or method and get its value or state monitored in a customizable UI during runtime. When monitoring non static (instances), you have to register the monitored target, as shown in the next point. Other than that there is not much to it. Just try it out yourself.



> Note that the PlayerMovement class in this example is calling `this.RegisterMonitor()` in its Awake and `this.UnregisterMonitor` in its OnDestory method.

Instanced and Static Member

When monitoring non static member of a class, instances of these classes must be registered when they are created and unregistered when they are destroyed. This process can be automated or simplified, either by creating a custom Factory system that will create/instantiate objects and register them automatically, or by inheriting from a base type that will automatically register and unregister instances. You can use the following predefined base types for this purpose.

- `MonitoredBehaviour : MonoBehaviour`
- `MonitoredSingleton<T> : MonoBehaviour where T : MonoBehaviour`
- `MonitoredScriptableObject : ScriptableObject`
- `MonitoredObject : object, IDisposable`

```
using Baracuda.Monitoring;
```

```
// Monitored instance must be registered / unregistered.
public class Player : MonoBehaviour
{
    [Monitor] private int health;

    private void Awake()
    {
        Monitor.StartMonitoring(target);
        // Or use this extension method:
        this.StartMonitoring();
    }

    private void OnDestroy()
    {
        Monitor.StopMonitoring(target);
        // Or use this extension method:
        this.StopMonitoring();
    }
}
```

```
// Simplified by inheriting from MonitoredBehaviour.
public class Player : MonitoredBehaviour
{
    [Monitor] private int health;

    // Remember to call base.Awake and base.OnDestroy.
    protected override void Awake()
    {
        base.Awake();
    }

    protected override void OnDestroy()
    {
        base.OnDestroy();
    }
}
```

```
using Baracuda.Monitoring;

public class GameManager
{
    // Static member are always monitored!
    [Monitor]
    public static GameState GameState { get; }
}
```

Monitoring Fields and Properties

Monitoring fields and properties is almost identical, differing only in their technical implementations. Just place the Monitor, MonitorField or MonitorProperty on either a field or a property and get its value displayed automatically . Multiple types like Booleans, Collections, Vectors etc. are also displayed in a readable way. To customize how a monitored value is displayed you can use a Value Processor and utilize a variety of additional formatting attributes.

```
using Baracuda.Monitoring;

[MonitorField]
private int value;

[MonitorProperty]
private int Value { get; }
```

Monitoring Events

Use the `[Monitor]` or `[MonitorEvent]` attributes to monitor the state of an event. The `[MonitorEvent]` attribute accepts additional arguments to customize how the event is displayed.

Property	Description
`ShowSubscriberCount`	When enabled, the subscriber count of the event is displayed.
`ShowInvokeCounter`	When enabled, the invoke count of the event is displayed. (Can be incorrect when async profiling is enabled!)
`ShowSubscriberInfo`	When enabled, every subscribed delegate is displayed.
`ShowEventSignature`	When enabled, display the signature of the event.
`ShowEventHistory`	When enabled, the arguments of the last x amount of invocations is displayed. (Planned feature)

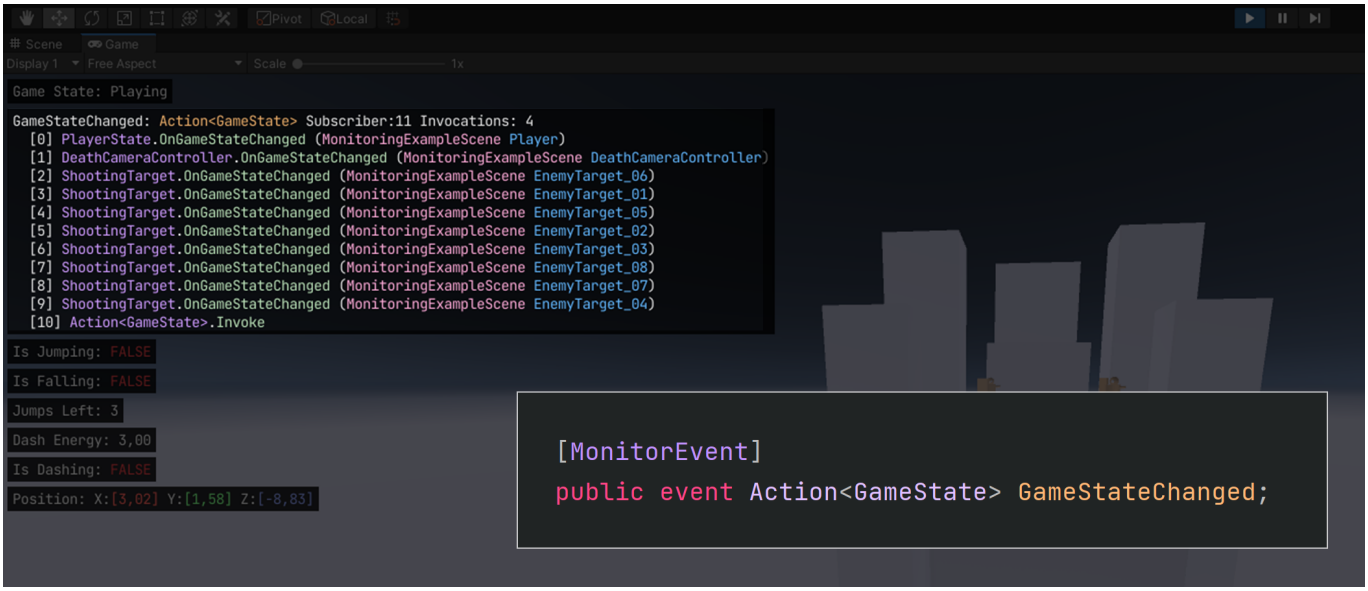
```
[Monitor]
public event Action OnGameStarted;

[Monitor]
public event Action<T> OnValueChanged;

[Monitor]
public event GameStateDelegate OnGameStateChanged;
public delegate void GameStateDelegate(GameState gameState);


// If you want to monitor how ofter an event is invoked without displaying every
subscriber.
```

```
[MonitorEvent(ShowInvokeCounter = true, ShowSubscriberInfo = false)]
public event Action<Player> OnPlayerSpawn;
```



Monitoring Methods

A method can be monitored like a field or property with the additional feature that their out parameters are monitored too. Default parameter values can be set by passing them to the constructor of the `[MonitorMethod]` attribute. Even methods that return void can be monitored if they have at least one out parameter. Default and Global Value Processor are applied to monitored out parameters, meaning that collections, vectors etc. assigned via out parameter are displayed in a readable way and not just formatted with ToString().

```
using Baracuda.Monitoring;

[Monitor]
public string GetName()
{
    return "Hello World";
}

[MonitorMethod(3, 5)]
public int Multiply(int lhs, int rhs)
{
    // Displayed value will be 15;
    return lhs * rhs;
}

[MonitorMethod]
public bool TryGetPlayer(int playerIndex, out Player player)
{
    // Method will be called and both the return value and the out parameter
player are monitored.
```

```
        //...
    }

    [MonitorMethod]
    public void Populate(out Vector3[] verts)
    {
        // verts will be displayed in a readable way.
        //...
    }
```

> This example additionally shows that out parameters are formatted too.

Attributes

Use Attributes to customize the monitoring process & display of your member. The attributes provided are divided into three broad categories, first the "Monitoring Attributes" to determine which C# member to monitor, second the "Meta Attributes" to customize how a member is monitored and third other attributes used for various purposes.

Monitoring Attributes

Attribute	Base Type	Description
[Monitor]	Attribute	Monitor a field, property, event or method
[MonitorValue]	MonitorAttribute	Monitor a field or property
[MonitorProperty]	MonitorValueAttribute	Monitor a property
[MonitorField]	MonitorValueAttribute	Monitor a field
[MonitorEvent]	MonitorAttribute	Monitor an event

Attribute	Base Type	Description
[MonitorMethod]	MonitorAttribute	Monitor a method

Meta Attributes

Attribute	Description
[MUpdateEvent]	Set an event that will trigger an refresh/update (more)
[MValueProcessor]	Set a method that will process the value before it is displayed as a string (more)
[MShowIf]	Set custom validation logic (more)
[MEnabled]	Set the default enabled state of the monitored member.
[MTag]	Set optional tags used for filtering

Meta Attributes (Formatting)

Attribute	Description
[MOptions]	Contains (almost) all of the options below.
[MFormat]	Custom format string used to display the members value if possible.
[MLabel]	Custom label for the member (otherwise humanized name).
[MFontSize]	Set the font size for the monitored member.
[MFontName]	Pass the name of a custom font style that will be used for the monitored member.
[MGroupName]	Set the group for the monitored member.
[MGroupElement]	Whether or not the monitored member should be wrapped in a group.
[MShowIndex]	If the monitored member is a collection, determine if the index of individual elements should be displayed or not.
[MElementIndent]	The indent of individual elements of a displayed collection.
[MPosition]	The preferred position of an individual monitored member on the canvas.
[MTextAlign]	Horizontal text align.
[MOrder]	Relative vertical order of the monitored member.
[MGroupOrder]	Relative vertical order of the group of the monitored member.
[MRichText]	Override local RichText settings.
[MTextColor]	Set the text color for the element.
[MBackgroundColor]	Set the background color for the monitored member.
[MGroupColor]	Set the background color for the group of the monitored member.

Attribute	Description
[MStyle]	UIToolkit only. Provide optional style names.

Other Attributes

Attribute	Description
[GlobalValueProcessor]	Declare a method as a global value processor for a specific type (more)
[DisableMonitoring]	Disable monitoring for the target class or assembly

Value Processor

You can add the MValueProcessorAttribute to a monitored field, porperty or method to gain more control of its string representation. Use the attribute to pass the name of a method that will be used to parse the current value to a string. The value processor method must accept a value of the monitored members type, can be both static and non static (when monitoring a non non static member) and must return a string.

```
[MValueProcessor(nameof(IsAliveProcessor))]
[Monitor]
private bool isAlive;

private string IsAliveProcessor(bool isAliveValue)
{
    return isAliveValue ? "Player is Alive" : "Player is Dead!";
}
```

```
[MValueProcessor(nameof(IListProcessor))]
[Monitor] private IList<string> names = new string[] {"Gordon", "Alyx", "Barney"};

private string IListProcessor(IList<string> elements)
{
    var str = string.Empty;
    foreach (var name in elements)
    {
        str += name;
        str += "\n";
    }
    return str;
}
```

> This example demonstrates how to create a simple progress bar with the help of a value processor. Note that this code is already relatively optimized.

Static Value Processor

Static processor methods can have certain overloads for objects that implement generic collection interfaces, which allow you to process the value of individual elements of the collection instead of the whole collection all at once.

```
//IList<T> ValueProcessor
[MValueProcessor(nameof(IListProcessor))]
[Monitor] private IList<string> names = new string[] {"Gordon", "Alyx", "Barney"};

private static string IListProcessor(string element)
{
    return $"The name is {element}";
}

[MValueProcessor(nameof(IListProcessorWithIndex))]
[Monitor] private IList<string> Names => names;

private static string IListProcessorWithIndex(string element, int index)
{
    return $"The name at index {index} is {element}";
}
```

```
//IDictionary<TKey, TValue> ValueProcessor
[MValueProcessor(nameof(IDictionaryProcessor))]
[Monitor] private IDictionary<string, bool> isAliveDictionary = new
Dictionary<string, bool>
{
    {"Bondrewd", true},
    {"Lyza", false}
};
```

```
private static string IDictionaryProcessor(string name, bool isAlive)
{
    return $"{name} is {(isAlive ? "alive" : "dead")}";
}
```

```
//IEnumerable<T> ValueProcessor
[MValueProcessor(nameof(IEnumerableValueProcessor))]
[Monitor]
private IEnumerable<int> randomNumbers = new List<int>
{
    1, 43, 14, 65, 23, 174, 16, 2, 786, 4, 89
};

private static string IEnumerableValueProcessor(int number)
{
    return $"{number} is {((number & 1) == 0? "Even" : "Odd")}";
}
```

Global Value Processor

You can declare a static method as a global value processor that is then used to process the value for every monitored member of the given type (instanced value processors will still be preferred). The value processor method must have a valid signature, meaning that is has to accept the monitored type as a fist or second argument, can optionally accept an IFormatData object as a first argument and must return a string.

```
// Custom type
public struct Version
{
    public readonly short Major;
    public readonly short Minor;
}

[GlobalValueProcessor]
private static string GlobalValueProcessorVersion(Version version)
{
    return $"Version: {version.Major}.{version.Minor}";
}

[GlobalValueProcessor]
private static string GlobalValueProcessorVersion(IFormatData ctx, Version
version)
{
    return $"{ctx.Label}: {version.Major}.{version.Minor}";
}
```

Conditional Display

You can use the MShowIfAttribute to define conditions that control if a monitored value is displayed or not. Note that the value will still be monitored but not drawn, meaning that fields, properties, events & methods will still be accessed. There are three different ways to validate if the targeted member is displayed or not.

Validated by Condition

Set a condition for the monitored value that when met will display the monitored value. This is useful to display important debugging information only when it is necessary.

```
// Queue will only be displayed if there are errors. (Queue is not empty)
[Monitor]
[MShowIf(Condition.CollectionNotEmpty)]
private Queue<string> errorCache;

// Property will only be displayed if Network is not available. (false)
[Monitor]
[MShowIf(Condition.False)]
private bool NetworkAvailable { get; }

// Property will only be displayed if MainCamera is not available or set. (null)
[Monitor]
[MShowIf(Condition.Null)]
private Camera MainCamera { get; }
```

Validated by Comparison

Very similar to Validated by Condition but more dynamic. Pass in another object value and determine a comparison type that is then used on the current value of the monitored member and the value passed as an argument. If the comparison evaluates to be true, the member will be displayed.

```
// Will only be displayed if the errorCode is 404
[Monitor]
[MShowIf(Comparison.Equals, 404)]
private int errorCode;

// Will only be displayed more than one player is active.
[Monitor]
[MShowIf(Comparison.Greater, 1)]
private int ActivePlayerCount { get; }
```

Validated by Member

Very dynamic way of determining if a member is displayed or not. Pass in the name of a field, property, method or event.

- Passed fields, properties and methods must return a Boolean that determines if the member is displayed or not.
- Passed methods can also accept the current value of the monitored member and use its for a more dynamic evaluation.
- Passed events must be `Action<bool>` and can be used to toggle the display of the member

```
// Will only be displayed if the field 'monitor' is true.
[Monitor]
[MShowIf(monitor)]
private bool IsAlive {get;}

[SerializeField] private bool monitor = true;


// Will only be displayed if the property 'IsDebug' is true.
[Monitor]
[MShowIf(monitor)]
private bool IsAlive {get;}

pubic static bool IsDebug { get; }


// Will only be displayed if the method 'Validate' returns true.
[Monitor]
[MShowIf(Validate)]
private Entity ActiveTarget {get;}

private bool Validate(Entity target)
{
    return target != null && this.IsActive;
}
```

Update Event

When monitoring a field or a property (Value units) you can provide an 'OnValueChanged' event that will tell the monitored unit that the state of the member has changed.

This event can either be an `Action` or an `Action<T>`, with T being the type of the monitored field or property. Note that once a valid update event was provided the unit will not be evaluated during an update cycle

anymore, unless `UpdateOptions` are explicitly set to `UpdateOptions.Auto` or `UpdateOptions.FrameUpdate`.

Passing an event will slightly reduce performance overhead for values or member that you know will update rarely. It is however not required.

```
private int healthPoints;
public event Action<int> OnHealthChanged;

[Monitor]
[MUpdateEvent(nameof(OnHealthChanged))]
public int HealthPoints
{
    get => healthPoints;
    private set
    {
        healthPoints = value;
        OnHealthChanged?.Invoke(healthPoints);
    }
}
```

```
[Monitor]
[MUpdateEvent(nameof(OnGameStateChanged))]
private bool isGamePaused;

public event Action OnGameStateChanged;

public void PauseGame()
{
    isGamePaused = true;
    OnGameStateChanged?.Invoke();
}

public void ContinueGame()
{
    isGamePaused = false;
    OnGameStateChanged?.Invoke();
}
```

Monitoring UI

Use `Baracuda.Monitoring.Monitor.UI`to access `IMonitoringUI` API.

Member	Description

Member	Description
`bool Visible`	Get or set the visibility of the current monitoring UI
`T GetCurrent<T>()`	Get the current monitoring UI instance
`void SetActiveMonitoringUI(MonitoringUI monitoringUI)`	Set the active MonitoringUI
`void ApplyFilter(string filter)`	Filter displayed units by their name, tags etc. more
`void ResetFilter()`	Reset active filter. more
`event Action<bool> VisibleStateChanged()`	Event invoked when the monitoring UI became visible or invisible.

UI Formatting

Formatting attributes can be used to apply custom styling options on how a monitored member is displayed. There are multiple ways to reduce boiler plate code.

Attribute	Description
`[MOptions]`	Contains (almost) all of the options below.
`[MFormat]`	Custom format string used to display the members value if possible.
`[MLabel]`	Custom label for the member (otherwise humanized name).
`[MFontSize]`	Set the font size for the monitored member.
`[MFontName]`	Pass the name of a custom font style that will be used for the monitored member.
`[MGroupName]`	Set the group for the monitored member.
`[MGroupElement]`	Whether or not the monitored member should be wrapped in a group.
`[MShowIndex]`	If the monitored member is a collection, determine if the index of individual elements should be displayed or not.
`[MElementIndent]`	The indent of individual elements of a displayed collection.
`[MPosition]`	The preferred position of an individual monitored member on the canvas.
`[MTextAlign]`	Horizontal text align.
`[MOrder]`	Relative vertical order of the monitored member.
`[MGroupOrder]`	Relative vertical order of the group of the monitored member.
`[MRichText]`	Override local RichText settings.
`[MTextColor]`	Set the text color for the element.

Attribute	Description
`[MBackgroundColor]`	Set the background color for the monitored member.
`[MGroupColor]`	Set the background color for the group of the monitored member.
`[MStyle]`	UIToolkit only. Provide optional style names.
`[MAttributeCollection]`	Can be used to create custom preset attributes to reduce boiler plate code.

The `[MOptions]` attribute contains almost all of the other options. Individual attributes will override settings passed with the `[MOptions]` attribute but depending on you preferences you can either use multiple attributes or just the `[MOptions]` attribute.

```
// Will be displayed as "Value: 3.141"
[Monitor]
[MOptions(Format = "0.000", FontSize = 16)]
private float pi = 3.14159265359;

// Effectively the same as
[Monitor]
[MFormat("0.000")]
[MFontSize(16)]
private float pi = 3.14159265359;
```

Reducing Boiler Plate Code

- Creating a custom attribute and inheriting from `MOptionsAttribute` will let you use the constructor of the custom attribute to set multiple values. The attribute can then be used on multiple monitored members to apply its settings.
- Any formatting or meta attribute applied to a class is also applied to every monitored member declared within it. Attributes directly applied to a member within such a class will always override the class scoped values meaning that you can still apply individual options to members within such a class. The `[MTag]` attribute is the only attribute that will is not overridden but added. This means that tags applied to a class scope will apply to every monitored member of that class even if these members have a custom tag attribute themselves.
- Any formatting or meta attribute applied to a custom attribute that inherits from `[MAttributeCollection]` will be added to monitored member with that custom attribute. This is a similar way like the fist option of this list but does not require overriding a constructor. It acts more like a bridge or proxy.

> The code segments below all do the same but with different approaches.

```
class MyClass
{
    [Monitor]
    [MFormat("0.000")]
    [MFontSize(16)]
    [MPosition(UIPosition.UpperRight)]
```

```
    [MGroupElement(false)]
    [MTag("Gameplay")]
    pubic int value1;

    [Monitor]
    [MFormat("0.000")]
    [MFontSize(16)]
    [MPosition(UIPosition.UpperRight)]
    [MGroupElement(false)]
    [MTag("Gameplay")]
    pubic int value2;

    [Monitor]
    [MFormat("0.000")]
    [MFontSize(16)]
    [MPosition(UIPosition.UpperRight)]
    [MGroupElement(false)]
    [MTag("Gameplay")]
    pubic int value3;
}
```

```
public class MyFromatting : MOptionsAttribute
{
    public MyFromatting()
    {
        Format = "0.00";
        FontSize = 16;
        Position = UIPosition.UpperRight;
        GroupElement = false;
        Tags = new string[] {"Gameplay"};
    }
}

class MyClass
{
    [Monitor]
    [MyFromatting]
    pubic int value1;

    [Monitor]
    [MyFromatting]
    pubic int value2;

    [Monitor]
    [MyFromatting]
    pubic int value3;
}
```

```
[MFormat("0.000")]
[MFontSize(16)]
[MPosition(UIPosition.UpperRight)]
[MGroupElement(false)]
[MTag("Gameplay")]
class MyFromatting : MAttributeCollection
{
}

class MyClass
{
    [Monitor]
    [MyFromatting]
    pubic int value1;

    [Monitor]
    [MyFromatting]
    pubic int value2;

    [Monitor]
    [MyFromatting]
    pubic int value3;
}
```

```
[MFormat("0.000")]
[MFontSize(16)]
[MPosition(UIPosition.UpperRight)]
[MGroupElement(false)]
[MTag("Gameplay")]
class MyClass
{
    [Monitor]
    pubic int value1;

    [Monitor]
    pubic int value2;

    [Monitor]
    pubic int value3;
}
```

When applying multiple formatting attributes either directly on the monitored member or via proxy, it may happen that the same attribute is applied with different values. In this case there is a clear structure which attribute will be used.

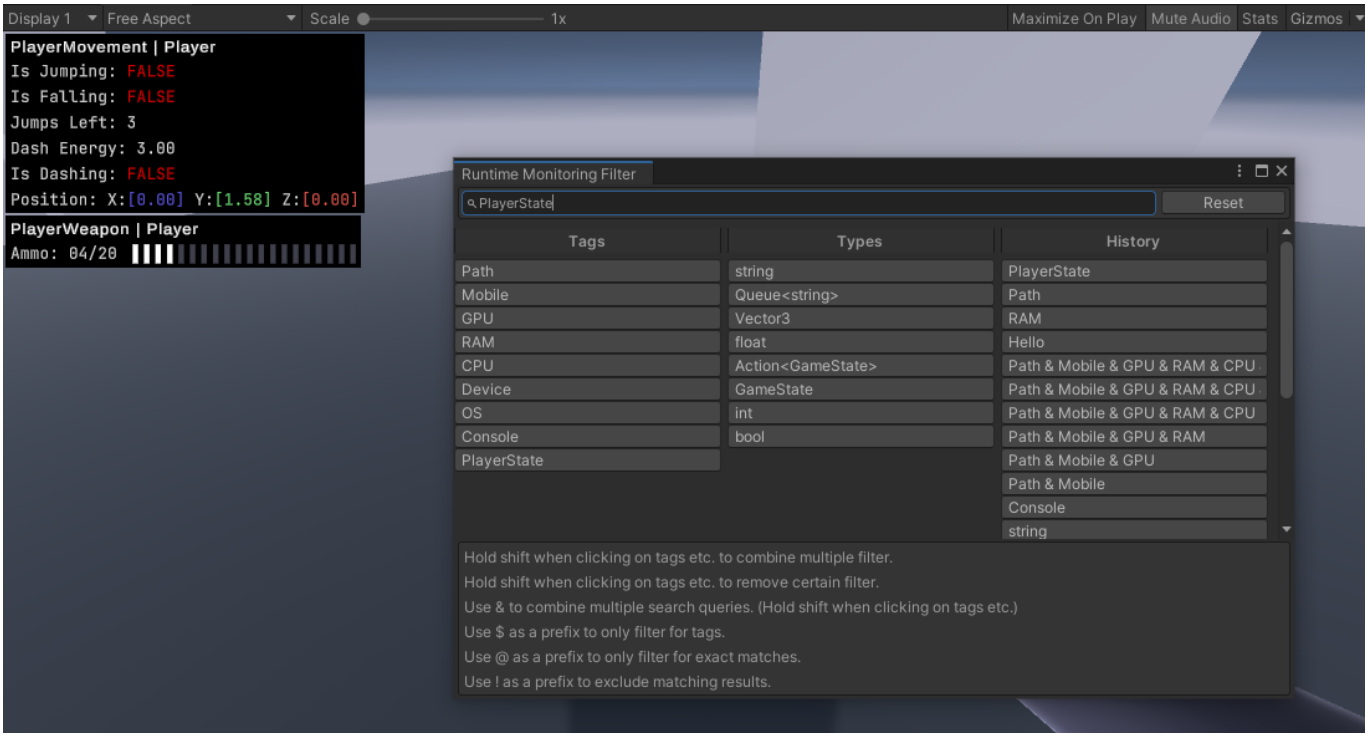
1. Directly applied to the monitored member.
2. Applied to the monitored member by `[MOptions]`
3. Applied to the monitored member by a custom `[MAttributeCollection]`

4. Directly applied to the monitored members class.
5. Applied to the monitored members class by `[MOptions]`
6. Applied to the monitored members class by a custom `[MAttributeCollection]`

> Note that this is for display purposes only. There is no need to apply that many formatting options to every monitored member. Formatting is 100% optional.

UI Filtering

You can filter the currently displayed elements using the monitoring filter API from `IMonitoringUI`. Or by using the Filter Editor window (menu: Tools > Runtime Monitoring > Filter Window) Filtering will check for a variety of matches. If you want more explicit filtering you can disable most of these checks by navigating to (menu: Tools > Runtime Monitoring > Settings: Filtering). You can also use the settings to determine if filtering should be case sensitive or case insensitive. By default filtering is case insensitive!



Optional Filter	Description
Filter Label	Enable filtering using the displayed label. (case insensitive) This option is the most intuitive.
Filter Static or Instance	Filter for static or non static member with *static* or *instance*
Filter Type	Enable filtering using the name of the type of the monitored member. (e.g. *bool*, *int*, *queue* etc.)
Filter Declaring Type	Enable filtering using the name of the members declaring type. (e.g. *MonoBehaviour*, *Player*, *GameController* etc.)
Filter Member Type	Enable filtering using the member type (*Field*, *Property*, *Event*, *Method*)
Filter Tags	Enable filtering using tags applied by the `[MTag]` attribute. more

Filter API

```
using Baracuda.Monitoring;

// Apply filter. (This filter will only show fields and properties)
Monitor.UI.ApplyFilter("Field & Property");

// Reset filter.
Monitor.UI.ResetFilter();
```

Absolute Filtering

> Filter stating with a `@` are always case sensitive and only use the actual name of the monitored member.

Tag only Filtering

> Filter stating with a `$` only use tags applied with the custom `[MTag]` attribute. more

Combining Filters

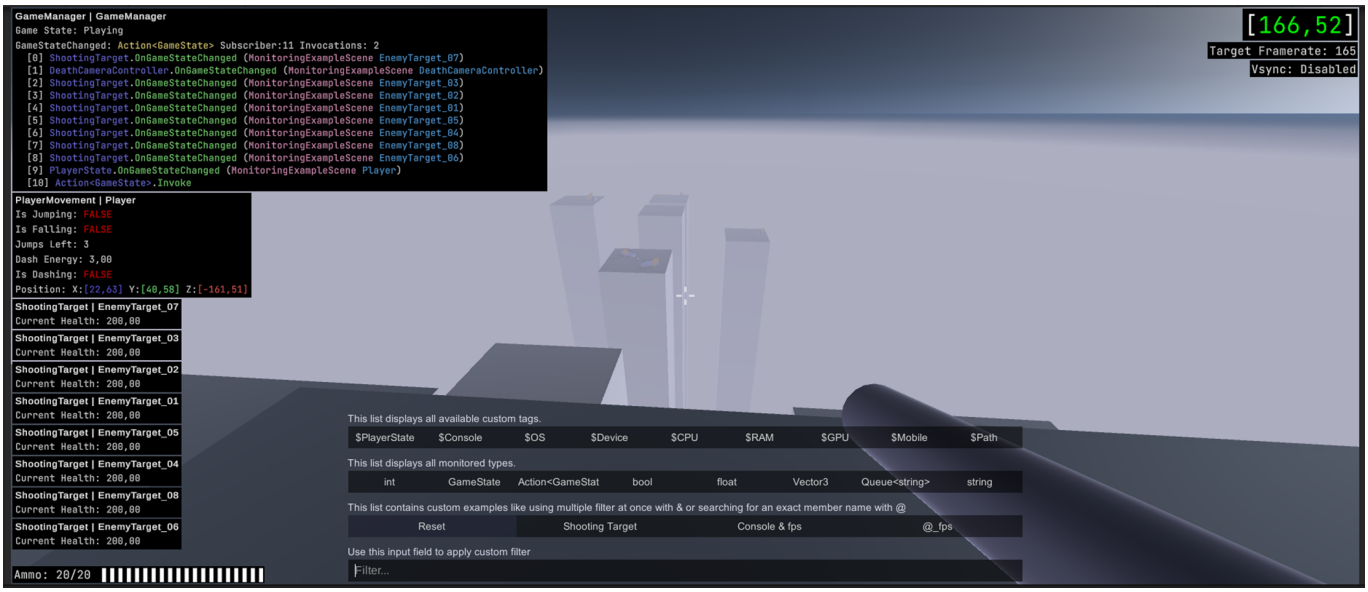
> Use a `&` symbol to combine multiple filters.

Negation Filter

> Append an `!` to the beginning of a filter to negate it.

You can change all of the symbols mentioned above in the monitoring settings by navigating to (menu: Tools > Runtime Monitoring > Settings: Filtering).

> This example shows a custom filtering setup in the example scene.



Systems and API

The `Monitor` class (`Baracuda.Monitoring.Monitor`) is the primary access point to access monitoring systems and API.

```
bool Initialized { get; }
```

> Returns true once the system has been fully initialized. This might take some time after playmode has been entered depending on whether or not threaded profiling has been enabled in the settings or not.

```
IMonitoringUI UI { get; }
```

> Access API to control the monitoring UI.

```
IMonitoringSettigns Settings { get; }
```

> Access to the monitoring settings asset. Edit settings via (menu: Tools > Runtime Monitoring > Settings)

```
IMonitoringEvents Events { get; }
```

> Access monitoring event handlers.

```
IMonitoringRegistry Registry { get; }
```

> Primary interface to access cached data.

```
void StartMonitoring<T>(T target) where T : class
```

> Register an object that is monitored.

`void StopMonitoring<T>(T target) where T : class`

> Unregister an object that is monitored.

```
//Example how to get / resolve a monitoring system interface.
IMonitoringUI monitoringUI = Monitor.UI
```

Type Interfaces

Important internally used types implement interfaces that should make it more easy to work with and extend this tool. These interfaces are especially used when creating a custom UI controller.

`IMonitorHandle`

> Internally used handle for instances of a monitored member

`IMonitorProfile`

> Internally used profile describing a monitored member

Monitoring UI API

Use `Baracuda.Monitoring.UI` to access UI API via the `IMonitoringUI` interface.

`bool Visible`

> Get or set the visibility of the UI.

`event Action<bool> VisibleStateChanged()`

> Event invoked when the monitoring UI became visible or invisible.

`T GetCurrent<T>()`

> Get the currently active MonitoringUIController casted to a concrete implementation.

`void ApplyFilter(string filter)`

> Filter displayed units by their name, tags etc. more

`void ResetFilter()`

> Reset active filter. more

Monitoring Events

Use `Baracuda.Monitoring.Events` to access monitoring event handlers via the `IMonitorEvents` interface.

`event ProfilingCompletedDelegate ProfilingCompleted`

> Event is invoked when profiling process for the current system has been completed. Subscribing to this event will instantly invoke a callback if profiling has already completed.

```
event Action<IMonitorHandle> MonitorHandleCreated
```

> Event is called when a new MonitorHandle was created.

```
event Action<IMonitorHandle> MonitorHandleDisposed
```

> Event is called when a MonitorHandle was disposed.

Monitoring Registry

Use `Baracuda.Monitoring.Registry` to access cached data via the `IMonitoringRegistry` interface.

```
IReadOnlyList<IMonitorHandle> GetMonitorHandles(HandleTypes handleTypes);
```

> Get a list of monitoring handles for all targets.

```
IMonitorHandle[] GetMonitorHandlesForTarget<T>(T target) where T : class
```

> Get a list of IMonitorHandles registered to the passed target object.

```
IReadOnlyList<string> UsedTags { get; }
```

> Get a list of all custom tags, applied by `[MTag]` attributes that can be used for filtering.

```
IReadOnlyList<string> UsedFonts { get; }
```

> Get a collection of used font names.

```
IReadOnlyList<Type> UsedTypes { get; }
```

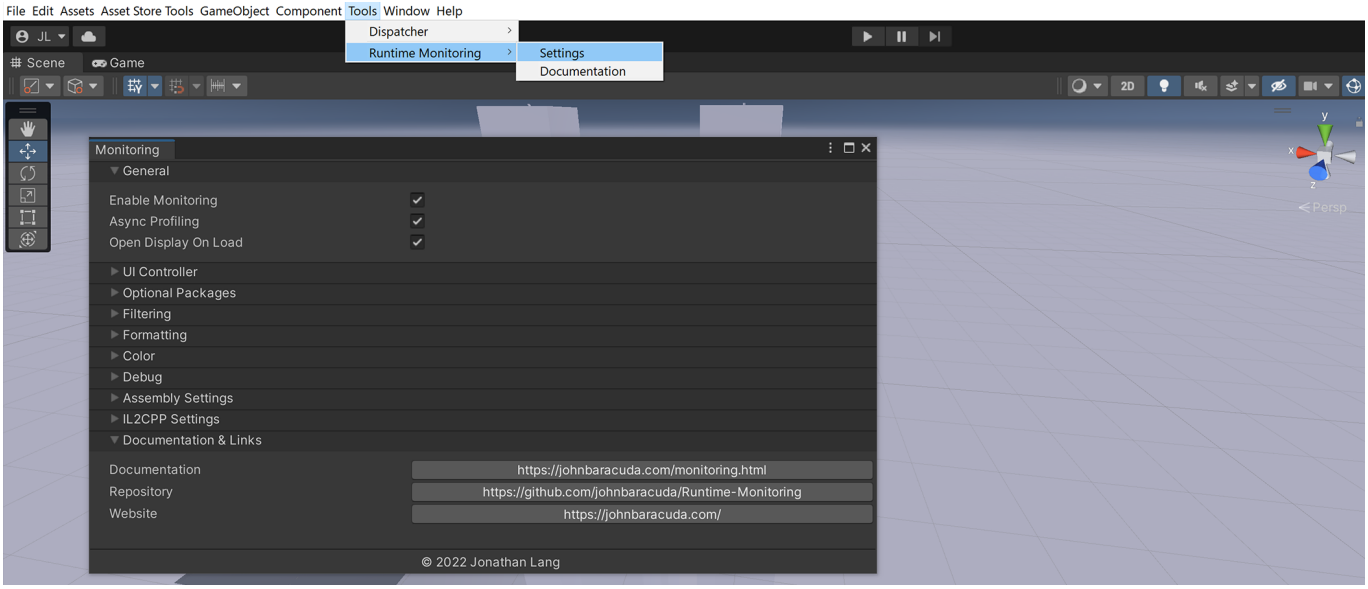
> Get a collection of monitored types.

```
IReadOnlyList<string> UsedTypeNames { get; }
```

> Get a collection of monitored type names converted to a readable string.

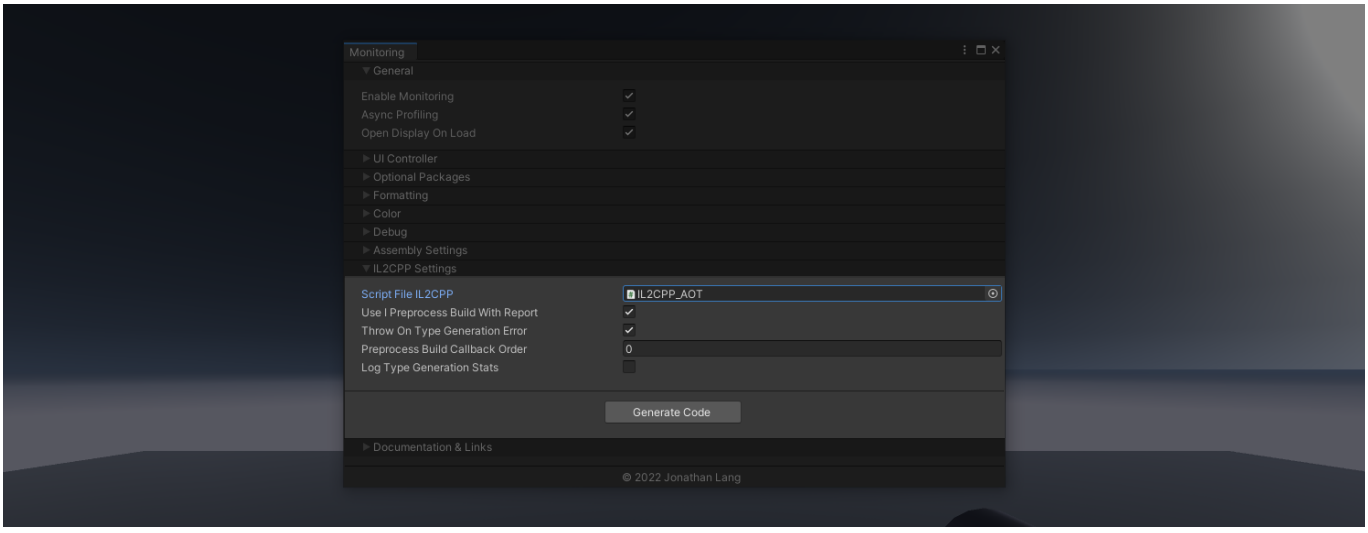
Monitoring Settings

Use `Baracuda.Monitoring.Settings` to access active settings via the `IMonitoringSettings` interface. You can configure and access the settings file via (menu: Tools > Runtime Monitoring > Settings)



Runtime Compatibility

Both Mono and IL2CPP runtimes are supported. RTM is making extensive use of dynamic type and method creation during its profiling process. In order to create these types, IL2CPP requires AOT compilation (Ahead of time compilation) When using IL2CPP runtime a list of types is generated shortly before a build to give the compiler the necessary information to generate everything it needs during runtime. You can manually create this list form the settings window.



Optimizations

In general RTM tries to be as optimized as possible by reducing allocations where ever possible and by doing a lot of the heavy work during the initial profiling. However due to the nature of some types and the creation of strings, allocations cannot be prevented. If performance is a sensitive aspect of your project, here are some tips and tricks to keep in mind.

Reference Types & Value Types

Since there is no easy way to check whether the actual value of a reference type has changed when it is evaluated, a monitored Reference Type is processed with ToString() each time it is evaluated. This, by default happens either every Update or Tick cycle and may generate a lot of garbage in a very short time, without much scope for automatic optimization. For this reason, it is recommended to pass an UpdateEvent for the monitored value whenever possible to reduce memory allocations. Of course, this shouldn't matter much if you're only debugging one value, but could be detrimental if you want to keep monitoring your member in a release or shipped build. The same is not true for monitored Value Types, as these are compared to a cached value to ensure that an update event and a string creation is only triggered when the value has actually changed.

Collections

Because collections are Reference Types the same applies here but on an even greater scale. Pass an update event when ever possible if you intend to monitor a collection over a longer period. Now because the example below requires a lot of boiler plate code I would not recommend this if you just want to quickly debug the values of a collection. I also want to mention that a better solution is planned and WIP.

```
[Monitor]
[MUpdateEvent(nameof(OnNamesChanged))]
public List<string> Names = new List<string>() {"Riebeckite", "Prisoner",
"Feldspar"};

private event Action OnNamesChanged;

public void AddName(string name)
{
    Names.Add(name);
    OnNamesChanged();
}

public void RemoveName(string name)
{
    Names.Remove(name);
    OnNamesChanged();
}
```

Frequently Asked Questions

How can I disable the tool in a release?

Set the **Enable Monitoring** field in the **Monitoring Settings** to **EditorOnly** This will install dummy systems in a build.

How can I uninstall the tool?

You can just remove the plugin by deleting the folder Assets/Baracuda.

Support Me

I spend a lot of time working on this and other free assets to make sure as many people as possible can use my tools regardless of their financial status. Any kind of support I get helps me keep doing this, so consider leaving a star ☆ making a donation or follow me on my socials to support me ♡

- Donation (PayPal.me)
- Linktree
- Twitter
- Itch