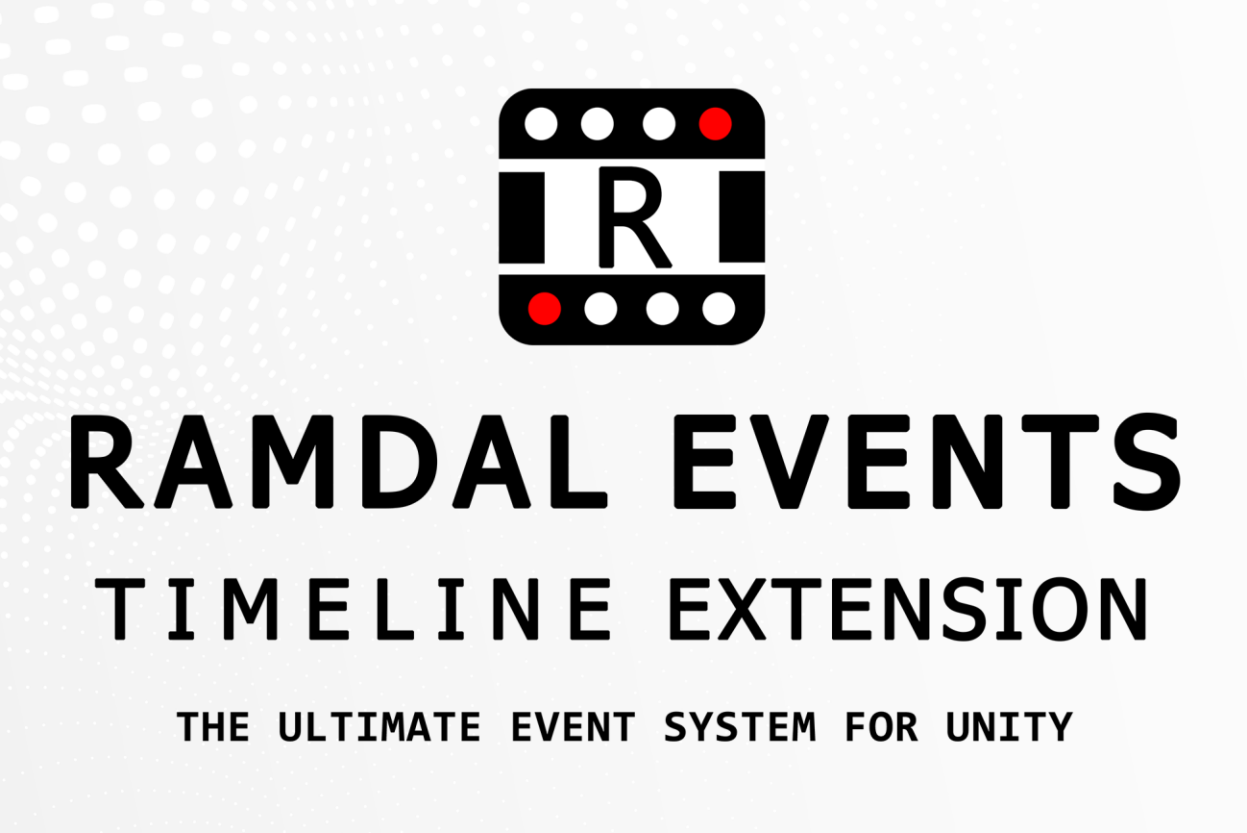
Ramdal Events Timeline Extension Documentation



Contents

Introduction

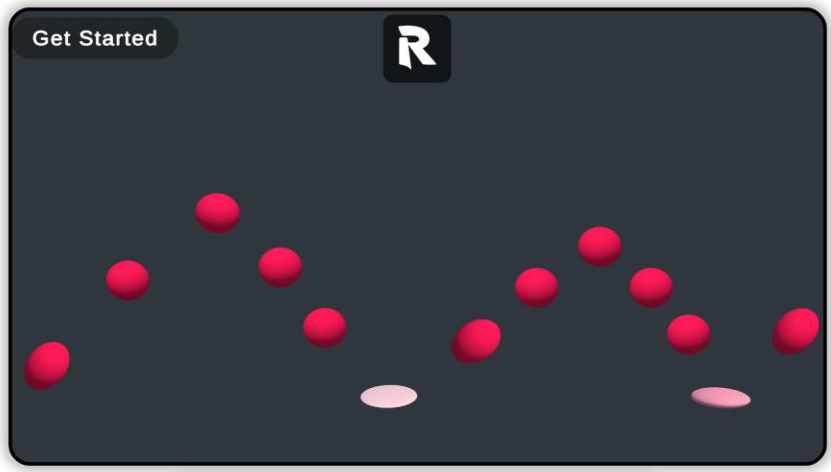
Ramdal Events Timeline Extension brings the full power of **Ramdal Events** directly into Unity Timeline, allowing you to call virtually any method and trigger existing gameplay logic directly from Timeline clips.

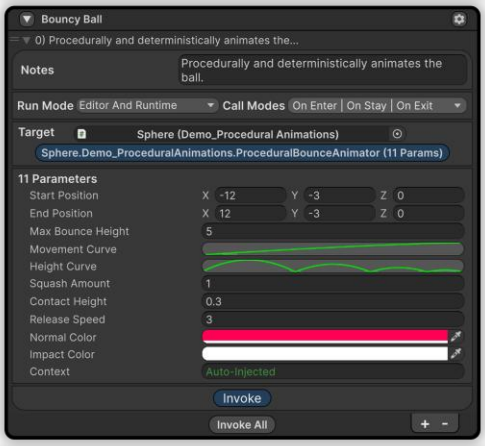
Unlike Unity Signals, you don't need to create Signal assets, receivers, additional listener components, or Timeline-specific scripts. Your existing Ramdal Events methods can be connected directly to Timeline, giving you a virtually unlimited range of actions and workflows without adding another layer of setup.

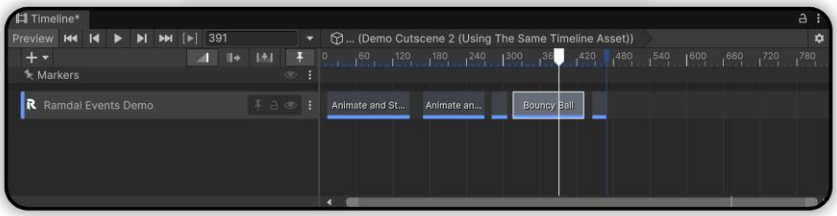
This means your Timeline isn't limited to predefined actions. Use the full Ramdal Events system to trigger virtually any gameplay logic your methods can perform, from controlling characters and animations to driving procedural systems, VFX, audio, UI, gameplay events, and complex cinematic interactions.

Build powerful Timeline-driven workflows with:

- **Advanced Timeline Events**: Trigger logic on clip enter, stay, exit, creation, and destruction.
- **No Custom Playables Required**: Build complex Timeline interactions directly from clips without writing Timeline-specific behaviours.
- **Automatic Timeline Context Injection**: Access the current Playable, FrameData, clip progress, and other Timeline data directly inside your methods.
- **Editor & Scrubbing Support**: Invoke and preview events in Play Mode, Editor Mode, or while scrubbing through the Timeline.
- **Refactor-Safe Method Connections:** Keep event connections intact when methods are renamed or refactored.
- **Project-Wide Event Tracking**: Search, locate, and manage Timeline event connections across your entire project.
- **High-Performance Execution**: Uses optimized delegate-based invocation with caching, GC-free execution, and IL2CPP/AOT support.







Create clean, flexible Timeline workflows while keeping your gameplay logic in your existing Ramdal Events methods instead of building separate Timeline-specific systems.

Why Ramdal Events Instead of Unity Signals?

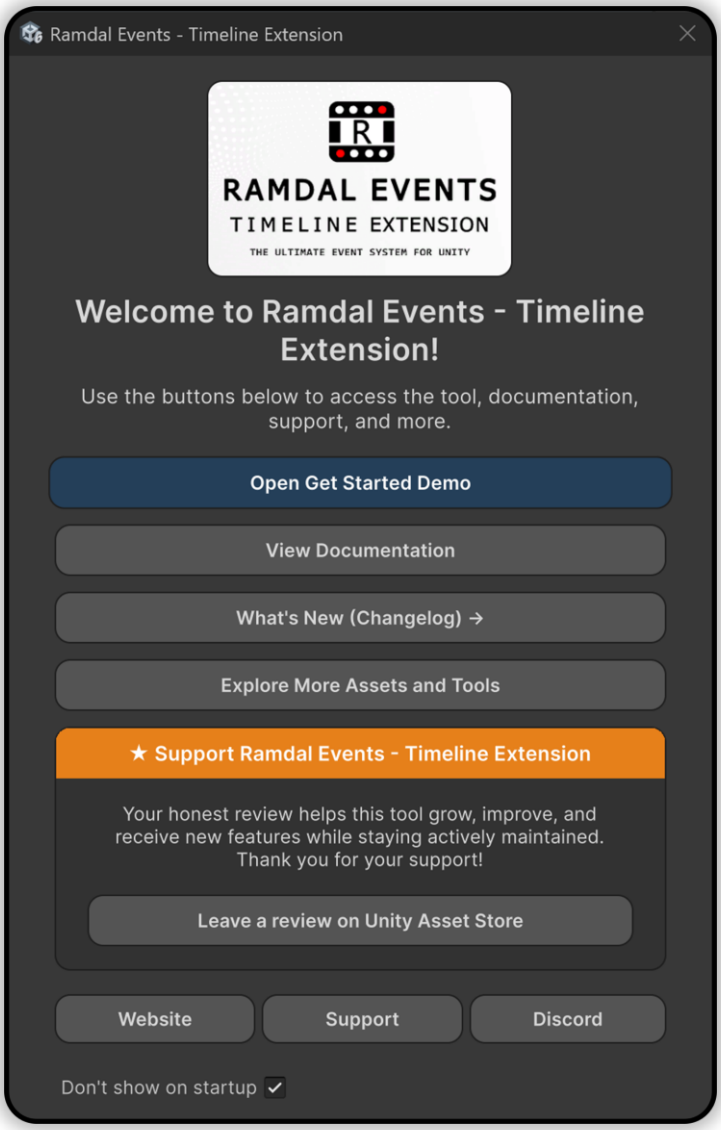
Unity Signals require separate Signal Assets and Signal Receivers, while UnityEvents limit parameter flexibility. As sequences grow, managing Signals, receivers, and connections can quickly become cumbersome.

Ramdal Events removes this extra layer by connecting Timeline directly to your methods, with flexible parameters, refactor-safe connections, diagnostics, project-wide tracking, and precise control over when events are invoked.

Getting Started

Installation

- Download and Import Ramdal Events into your Unity project.
- Download and Import the Timeline Extension into your Unity project.
- When Unity finishes compiling, the Get Started window will open automatically, providing quick access to the documentation, demos, and key features.
- If the window is closed, you can reopen it at any time from **Tools > Ramdal Games > Ramdal Events > Timeline Extension > Get Started**.



Accessing the Settings & Tools

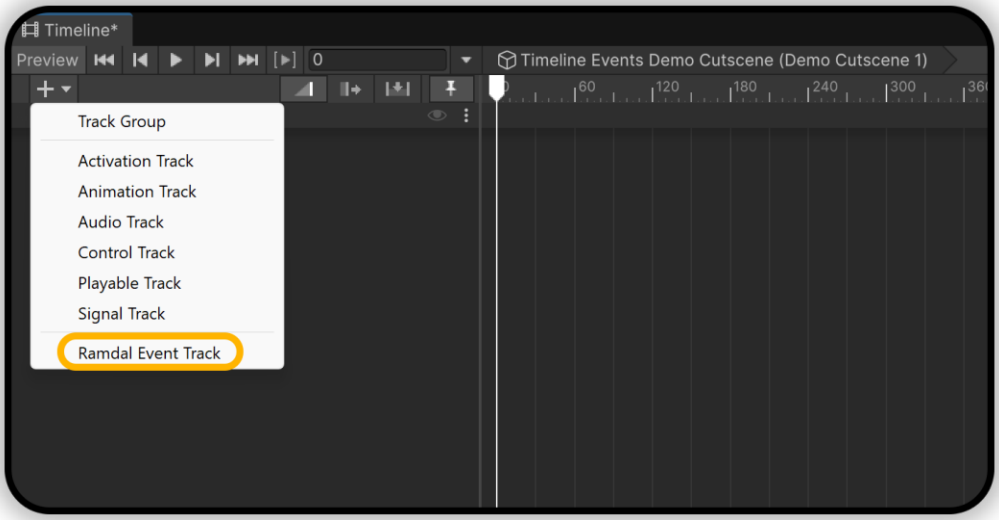
All Ramdal Events tools and settings are accessible from **Tools > Ramdal Games > Ramdal Events**, or directly from the **Settings button** available on every Ramdal Event component.

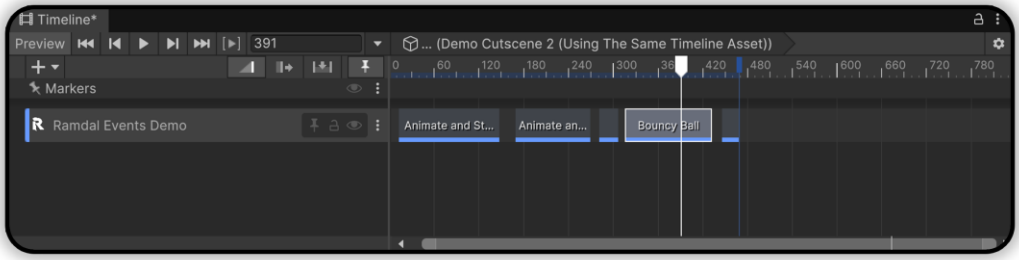
Ramdal Events Integration

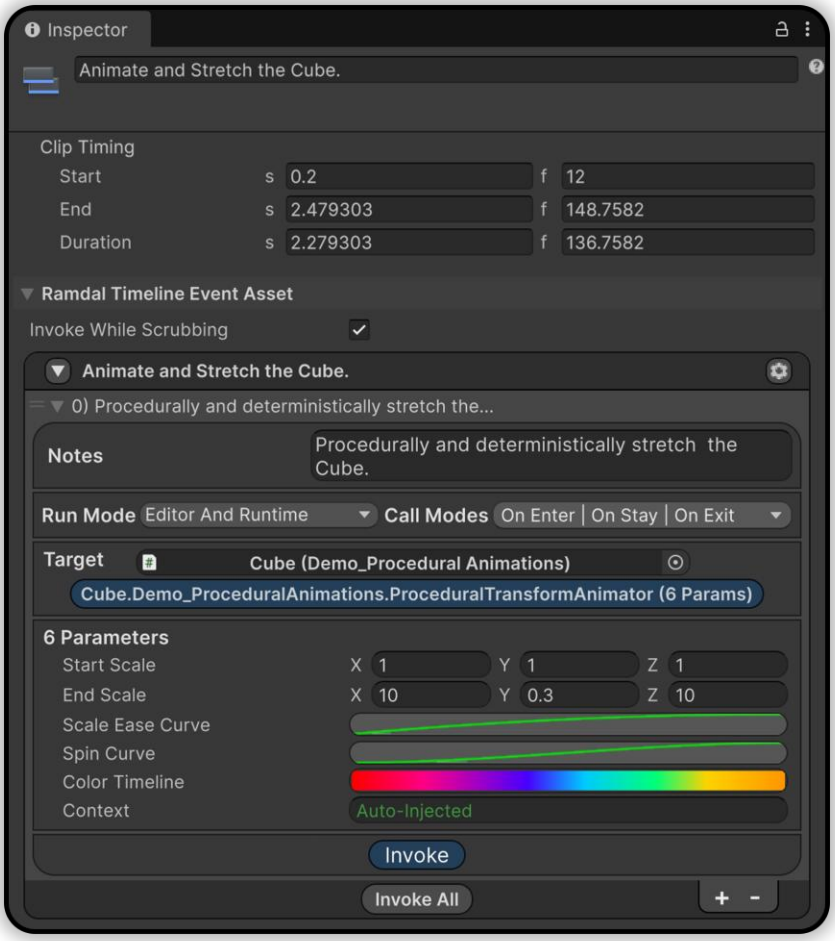
The Timeline Extension inherits the full **Ramdal Events** system. All event features, scripting options, parameter support, optimization settings, diagnostics, tracking tools, and Inspector workflows are available when working with Timeline Events.

For complete information about Ramdal Events and its features, see the **Ramdal Events documentation**.

Create Your First Ramdal Timeline Event







- To add a **Ramdal Event Track** to your Timeline, click the **+** button and select **Ramdal Event Track**.
- You can add as many **Event Assets** as needed to the track. Each event can have its own logic and persistent listeners.
- Double-click the **Event Title** to rename it. The custom name will be displayed on the Timeline clip.
- For information on **exposing methods**, scripting settings, and other Ramdal Events features, refer to the [Ramdal Events documentation].

Timeline Event System

The **Timeline Events Extension** introduces several Timeline-specific features designed to give you precise control over when and how your persistent listeners are invoked.

Persistent Listeners

Each Timeline clip uses a **Ramdal Event Asset** containing its own persistent listeners. These listeners define which methods are invoked and how they respond to the Timeline.

Each listener can be configured independently, allowing a single clip to trigger multiple methods and combine different behaviors without additional Timeline components or scripts.

Call Modes

Each **Persistent Listener** can be configured with one or multiple **Call Modes**, allowing the same listener to respond to different stages of a Timeline clip's lifecycle.

On Enter

Invoked when Timeline enters the clip.

This is ideal for triggering one-time actions at the beginning of a clip, such as starting an animation, enabling an object, playing an audio effect, or triggering gameplay logic.

On Stay

Invoked while the Timeline is inside the clip.

This is useful for continuously driving logic throughout the clip's duration, such as procedural animation, movement, blending values, or other continuously evaluated systems.

On Exit

Invoked when Timeline leaves the clip.

Use this when you need to perform an action at the end of a clip, such as stopping an effect, disabling an object, resetting a state, or triggering the next stage of gameplay logic.

On Playable Create

A special lifecycle call mode that is **independent of the clip's position on the Timeline**.

The listener is invoked when its Playable is created, meaning it can execute before Timeline begins evaluating the clips, even if the clip is positioned at the very end of the Timeline.

This makes it ideal for:

- Initializing systems.
- Preparing data or references.
- Setting initial states.
- Preparing objects before Timeline playback begins.

On Playable Destroy

The counterpart to **On Playable Create**.

This mode is also **independent of the clip's position** and is invoked when the Playable is destroyed after Timeline playback has finished or the Playable is otherwise released.

It is particularly useful for:

- Cleanup logic.
- Restoring object states.
- Resetting flags or temporary values.
- Reverting changes made during Timeline playback.

This is especially important when using **Editor Mode**, where certain changes made through event invocation may persist after playback. Using **On Playable Destroy** provides a convenient place to restore those states.

Combining Call Modes

A single Persistent Listener can use **multiple Call Modes simultaneously**.

This makes Timeline Events highly flexible without requiring separate listeners or custom Playable behaviours for every stage of your sequence.

Invoke While Scrubbing

Allows Timeline Events to be invoked while manually scrubbing through the Timeline.

When enabled, the event can respond to Timeline evaluation as you move the playhead in the Editor.

This is **disabled by default** and can be enabled when you need events to react during Timeline editing and previewing.

Note: For state-changing events used in Editor Mode, always consider adding a reset method through **On Playable Destroy**. See **Editor Invocation & State Reset** for more details.

Editor Invocation & State Reset

When an event is invoked in **Editor Mode**, any changes it makes to an object's state will **persist after Timeline evaluation**. Unity does not automatically revert these changes when you stop playback or move the Timeline playhead.

For example, if an event changes an object's position, visibility, animation state, or other properties, those changes can remain applied in the scene.

You can use **On Playable Destroy** to call your own reset logic and restore the object's original state. The reset listener can be set to **Editor Only**, so your reset method runs only in the Editor and is excluded from builds.

This allows you to freely preview state-changing events in the Editor while ensuring the scene is restored when the Timeline is finished and destroyed.

Automatic Timeline Data Injection

```
[RamdalEvent("ProceduralTransformAnimator_ID")]
0 references
public void ProceduralTransformAnimator(Vector3 startScale, Vector3 endScale, AnimationCurve scaleEaseCurve,
    AnimationCurve spinCurve, Gradient colorTimeline, TimelineEventContext context) {
    // Returns deterministic clip progress (0.0 to 1.0).
    float progress = context.Progress;
    // Evaluate the animation curves.
    float scaleMultiplier = scaleEaseCurve.Evaluate(progress);
    float spinMultiplier = spinCurve.Evaluate(progress);

    objectRenderer.transform.localScale = Vector3.LerpUnclamped(startScale, endScale, scaleMultiplier);

    objectRenderer.transform.localRotation = Quaternion.Euler(0f, spinMultiplier * 360f, 0f);

    Color evaluatedColor = colorTimeline.Evaluate(progress);
    if (Application.isPlaying)
        SetColor(evaluatedColor);
    else
        SetColorEditor(evaluatedColor);
}
```

TimelineEventContext is a special context object provided by Ramdal Timeline Events that gives your methods direct access to information about the currently evaluated Timeline clip.

By adding a *TimelineEventContext* parameter to an exposed method, you can access Timeline-specific data without manually retrieving it from the *PlayableDirector*.

The context provides information such as the current Playable, frame evaluation data, and normalized clip progress.

Playable

Gets the **Playable** associated with the current Timeline clip.

See the Unity documentation for Playable for more information.

FrameData

Gets the current **FrameData** provided by Timeline during evaluation.

See the Unity documentation for FrameData for more information.

Progress

Gets the normalized playback progress of the current Timeline clip, ranging from **0 to 1**.

Scene Object References

Why Scene References Need Special Handling

Unity Assets, including Timeline Assets, cannot directly serialize references to objects that belong to a scene. This means scene objects cannot be stored directly inside Timeline clips or assets.

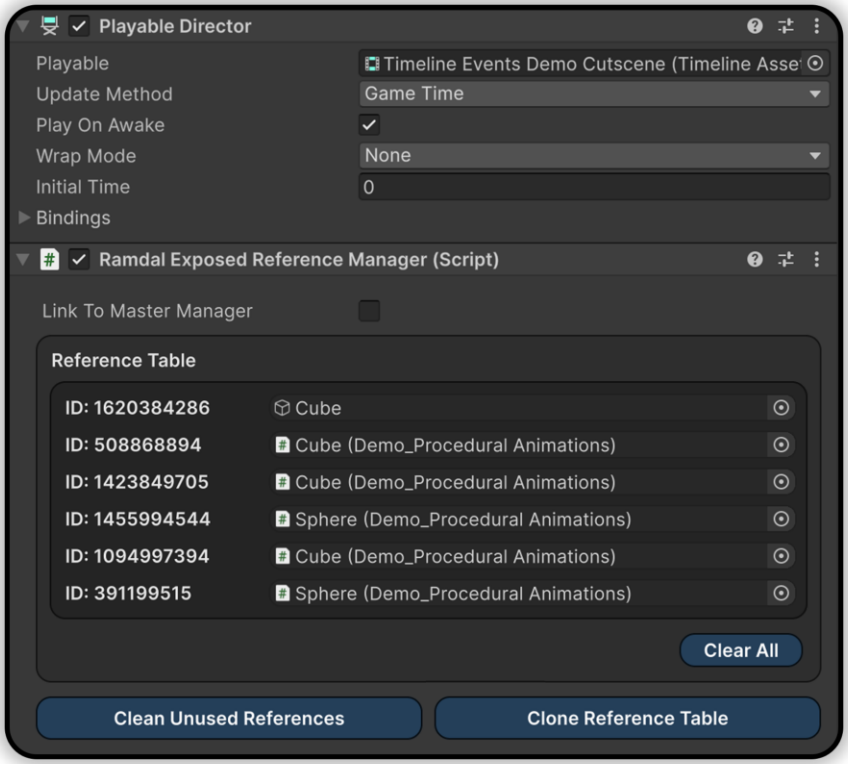
The Timeline Extension solves this through the *RamdalExposedReferenceManager*, which stores and resolves scene object references separately.

Ramdal Exposed Reference Manager

The **Ramdal Exposed Reference Manager** is automatically added alongside the **PlayableDirector** and acts as the storage and resolver for Timeline scene references.

It allows Timeline Events to safely reference scene objects while keeping those references outside the Timeline Asset itself.

This allows each Timeline instance to use its own scene objects, or multiple instances to share the same references when needed.



Reference Table

The manager stores all exposed scene references in a **Reference Table**.

Each binding is assigned a unique ID. Timeline stores this ID instead of the scene object itself, and the manager uses the ID to resolve the correct object when the event is evaluated.

Manager Tools

The manager provides several tools for managing the reference table.

Clear All

Removes all references from the current table.

Clean Unused References

Removes invalid or dead references that are no longer available.

Clone Reference Table

Copies the reference table from another *RamdalExposedReferenceManager*.

Linked Reference Managers

Managers can be connected using **Link To Master Manager**.

When linked:

- The manager uses the master's reference table.
- Reference changes are stored in the master manager.
- Multiple PlayableDirectors can share the same scene references.

When unlinked:

- The manager maintains its own independent reference table.
- Changes only affect that manager.

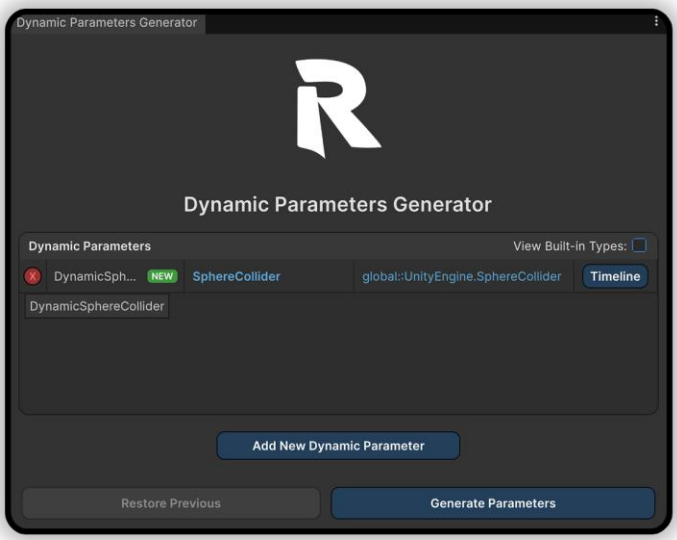
Link managers when multiple Timeline instances should use the same scene object bindings. Keep them independent when each Timeline instance needs its own scene references.

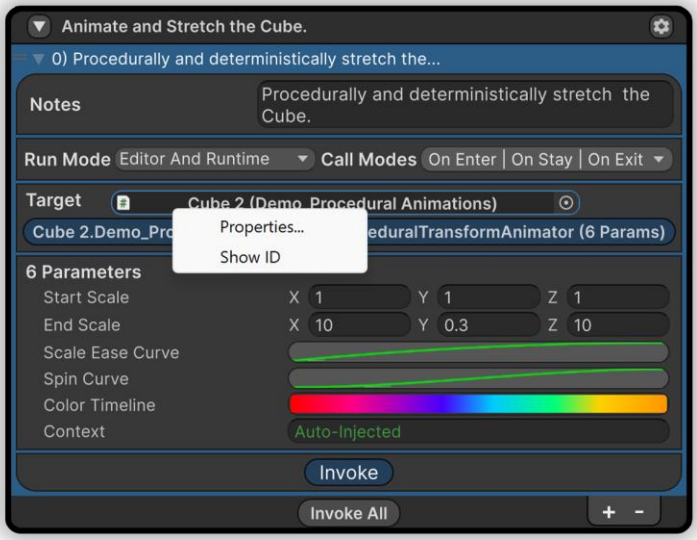
Ramdal Exposed Reference

Ramdal Exposed Reference allows Timeline Events to safely serialize and resolve scene object references.

Unity Timeline cannot directly serialize scene objects inside assets or nested data structures. By wrapping a scene object inside *RamdalExposedReference<T>,* the reference can be stored safely and resolved through the *RamdalExposedReferenceManager* when the event is invoked.

Standard Scene Object Parameters





For normal supported types such as **GameObject**, **BoxCollider**, and other **UnityEngine.Object** types, no manual conversion is required.

- Open the **Parameters Generator**, find the required type, enable **Timeline Support**, and generate the parameters.
- The generated parameter type will automatically use the required exposed reference workflow.
- Each exposed reference can be right-clicked to view or change its unique ID. This ID is used to locate and resolve the reference from the **Reference Table**.

Nested Scene References

```
/// <summary>
/// An example of a nested parameter type containing a Scene Object.
/// Normally, Timeline assets cannot store direct references to Scene objects like GameObjects and UnityEngine.Object inheritance.
/// By wrapping it in a RamdalExposedReference<T>, you can safely expose it and serialize it !
/// </summary>
[Serializable]
2 references
public class NestedSceneObject {
    public RamdalExposedReference<GameObject> Target;
    public float number;
    public string name;
}


/// <summary>
/// Demonstrates resolving scene objects hidden deep inside nested classes/structs.
///
/// HOW TO USE: Because the scene object is nested, Ramdal Events cannot resolve it automatically.
/// You must manually pass the 'RamdalExposedReferenceManager' (usually attached to your
/// PlayableDirector) into this method's parameter via the Inspector.
///
/// Note: Calling .Resolve() is NOT expensive! It does not use Find() or Reflection.
/// </summary>
[RamdalEvent("ResolveNestedSceneObject_ID")]
0 references
public void ResolveNestedSceneObject(NestedSceneObject sceneObject, RamdalExposedReferenceManager manager) {
    // Resolve the exposed reference to retrieve the scene GameObject.
    GameObject resolvedGo = sceneObject.Target.Resolve(manager);

    Debug.Log($"Resolved GameObject: {resolvedGo?.name ?? "Null"} | Number: {sceneObject.number} | Name: {sceneObject.name}");
}
```

When a scene object is contained inside a custom class or struct, Ramdal Events cannot automatically expose the nested reference to Timeline. Replace the scene object field with a *RamdalExposedReference<T>* using the required object type. This allows Timeline to serialize the reference safely and resolve it through the *RamdalExposedReferenceManager* when the event is invoked.

Resolving References

Nested exposed references require the *RamdalExposedReferenceManager* to resolve the stored reference.

The manager looks up the reference using its unique ID and returns the corresponding scene object.

Reference resolution is lightweight and does not use *Find()* or Reflection.

RamdalExposedReference API

ExposedID

Returns the unique ID used to identify this reference inside the Ramdal Exposed Reference Manager reference table.

Dispose()

Removes this reference from the provided *RamdalExposedReferenceManager*.

UnderlyingType()

Returns the actual referenced type T instead of the *RamdalExposedReference<T>* wrapper.

Resolve()

Resolves and returns the referenced object from the *RamdalExposedReferenceManager* as T.

ResolveObject()

Resolves and returns the referenced object as *object*.

Limitations & Important Notes

Ref/Out and Pointer Parameters

Ramdal Events supports ref, out, and pointer parameters. Changes made to these parameters modify the underlying data. Because Timeline Assets can be shared by multiple PlayableDirectors, these changes can affect other Timeline instances using the same asset.

RamdalExposedReference<T> scene references behave differently. They use a separate parameter instance for each *RamdalExposedReferenceManager*, so changes only affect that Timeline instance and do not modify the original parameter or other instances.

Scene References Modified During Play Mode

Scene references added or changed while in Play Mode are not saved when exiting Play Mode.

Make reference changes outside Play Mode if they need to persist.

Editor State Persistence

Events invoked in Editor Mode can leave changes applied to scene objects after Timeline evaluation.

See **Editor Invocation & State Reset** for information on restoring these changes safely.

Ramdal Events Limitations

All limitations and restrictions that apply to Ramdal Events also apply to Timeline Events.

See the Ramdal Events documentation for the complete list of limitations and supported scenarios.

Support and Useful Links

Ramdal Events Video Tutorials: youtube.com/playlist?list=PLPoPvIVslWnE
Timeline Extension Video Tutorials: youtube.com/playlist?list=PLXOT9dHBi5dc
Support: support@ramdalgames.com
Contact: contact@ramdalgames.com
Ramdal Games: https://ramdalgames.com
Mohamed Haftari: https://mohamedhaftari.com
Join Our Discord Community
Explore More Assets and Tools