Koreographer Custom Payload Demo Overview

for v1.6.1

Table of Contents

Overview

Koreographer enables you to build your own payload types for Koreography Events. This is a handy feature to employ if the built-in types do not cover your use case, or would otherwise require you to perform extra work to support your desired outcome.

To create your own custom payload, you need to create at least two classes:

1. A payload class
2. A custom Koreography Track class

The payload class will define the "shape" of the payload (its contents), as well as how to interact with its contents. The custom Koreography Track class will add support for the payload class to Koreographer, enabling Unity to properly handle payload serialization (save/load).

Implementation details for these two classes will be covered in depth in the following sections.

Custom Payload Class

Every payload is backed by a simple class that describes what kind of information to store and how to present it to the user. With custom payloads, you can decide for yourself what information a `KoreographyEvent` instance may carry to registered listeners.

A custom payload class must:

1. Implement the `IPayload` interface.
2. Adhere to Unity's Script Serialization requirements.
3. **Not** derive from `UnityEngine.Object`.

Implementing the IPayload Interface

A class is considered a payload if it implements the `IPayload` interface. This interface defines the following three methods:

1. `bool DoGUI(Rect displayRect, KoreographyTrackBase track, bool isSelected);`
2. `float GetDisplayWidth();`
3. `IPayload GetCopy();`

The **`DoGUI`** method is called by the Koreography Editor to draw the UI for the payload. Most built-in payloads are presented as a single field but this is not required. It is entirely possible to present the user with a button that opens a payload configuration popup. As the `DoGUI` method is called from within the Koreography Editor's `OnGUI` method, you should use Unity's IMGUI functions to generate the UI. The **`DoGUI`** method should return `true` if the payload was modified and `false` otherwise.

The **`GetDisplayWidth`** method is called by the Koreography Editor to determine an appropriate width for rendering the Payload UI in certain circumstances (e.g. the Peek UI, which is shown when a OneOff event is selected/hovered). In such circumstances, the returned value will be passed to the **`DoGUI`** method. Returning a value of **`0f`** (zero) from this method is an indication that a standard default value should be used.

It is ***very important*** to note that the `DoGUI` and `GetDisplayWidth` methods are **only defined in the Editor context**. Any implementation should therefore wrap the implementation of these methods in `UNITY_EDITOR` preprocessor directives (note that this also applies to `using` directives that reference Editor namespaces and types).

The **`GetCopy`** method is called by the Koreography Editor to create copies of payloads during certain operations (including copy/paste). How "deep" to copy your custom payload object is up to you.

Please see the examples included with this demo for more information.

Optional Functions

These functions are not entirely necessary, but are helpful and therefore recommended. All built-in payload types provide implementations for the best possible experience.

Koreography Editor Helpers

The Koreography Editor will look for the following method to assist in presentation.

- `public static string GetFriendlyName()`

The **`GetFriendlyName`** static function returns a string that describes what the Koreography Editor should display to the user for this payload type. This is how the Koreography Editor knows to call the `MaterialPayload` payload type simply "Material". If you do not provide this function, the raw type `Name` will be used.

KoreographyEvent Extension Methods

By convention, all built-in payload classes define at least two Extension Methods for the `KoreographyEvent` class. These are of the form:

1. **`bool Has[Type]Payload`:** This method allows the user to check if the `KoreographyEvent` instance contains a specific type of payload. An example would be the `HasMaterialPayload` method provided with the `MaterialPayload` type.
2. **`[TypeOfContent] Get[Content]Value`:** This is a convenience method that allows quick access to the contents of the payload directly from the `KoreographyEvent` instance. An example would be the `GetMaterialValue` method provided with the `MaterialPayload` type.

As with any extension methods, these must be implemented as part of a separate static class. You are also encouraged to implement your own extension methods for the `KoreographyEvent` class as you see fit!

Limitations

Payload classes cannot extend the `UnityEngine.Object` class or a subclass thereof. This includes the `MonoBehaviour` and `ScriptableObject` classes. This limitation exists due to serialization issues and therefore only affects the Editor context. It ***is*** possible to attach objects that inherit from these types to Koreography Events at runtime (this might occur, for example, during an initialization phase when a level is loaded in-game). Such a payload type would be an excellent candidate for the `[NoEditorCreate]` attribute.

If you would like to deliver these objects as part of a payload and configure them in the Koreography Editor, simply define a payload class that contains a field to do so. The `AssetPayload` type already does this for `ScriptableObject` instances, for example.

Custom Koreography Track Class

Unity's serialization system does not support polymorphism of non-`UnityEngine.Object` types. Payloads, unfortunately, fall into this category. To work around this issue, Koreographer makes use of Unity's Custom Serialization to support payload serialization within Koreography Tracks. The custom implementation allows Koreography Tracks to contain `KoreographyEvents` with varying types of payloads at runtime.

All Koreography Tracks contain a list of `KoreographyEvent` instances, each with a (possibly `null`) `IPayload` reference. When Unity serializes a Koreography Track, special logic kicks in that "unpacks" the payloads from within the list of event instances and stores them in separate, type-specific lists that are also defined on the Koreography Track. When Unity deserializes a Koreography Track, another set of special logic "repacks" the payloads from the separate, type-specific lists into the track's "mixed" list. This mixed list is the list that Koreographer uses at runtime.

In order to support serialization of any given payload type, a Koreography Track must provide the lists necessary to contain the unpacked payload instances. The built-in `KoreographyTrack` class contains the lists necessary to support the built-in payload types (those provided by Koreographer by default). To support a custom payload type, you must define a custom Koreography Track type that:

1. Extends the `KoreographyTrackBase` class or a subclass of it, and
2. Provides the required lists to support any type of payload you wish the Koreography Track to contain within its `KoreographyEvent` instances.

Defining A Koreography Track Class

All Koreography Track classes must at some point inherit from the `KoreographyTrackBase` class, an abstract class that contains all the functions and hooks necessary to work within the Koreographer system, including the custom serialization implementation. The one thing this class does *not* provide is automatic support for any given payload type. This must be added manually.

Extending KoreographyTrackBase

If your custom Koreography Track class inherits directly from the `KoreographyTrackBase` class you must define the necessary fields for any type of payload you expect the track to support. This allows you to create streamlined Koreography Tracks, saving a few bytes of memory (per Koreography Track instance) by leaving out unused "serialization fields" for payload types that will never be used. [The memory savings here are negligible unless you have hundreds of Koreography Tracks.]

Extending an Existing Koreography Track Class

If your custom Koreography Track class inherits from another Koreography Track class implementation, it will automatically gain support for the `public` or `protected` payload types supported by that class. You can then add support for additional payload types in your custom class. As such, extending the built-in KoreographyTrack class will provide automatic support for all built-in payload types (all payload fields are marked `protected` and are therefore visible to subclasses).

Required Fields

For each payload type that you wish to add to a given Koreography Track type, you must add two lists of the following format:

1. `List<[PayloadType]> _[PayloadType]s`
2. `List<int> _[PayloadType]Idxs`

where "[PayloadType]" is the literal name of the custom payload's class name. This matches how payloads in the built-in `KoreographyTrack` class are defined. See:

```
List<IntPayload> _IntPayloads;
List<int> _IntPayloadIdxs;
```

Recommended Field Attributes

While not strictly necessary, it is **highly recommended** that the required list fields also be marked with the following two attributes:

1. `[HideInInspector]` - Stops the field from appearing in the Inspector.
2. `[SerializeField]` - This is **required** unless you set the field to `public`, in which case this attribute is unnecessary.

With the above attributes applied, the previous example becomes:

```
[HideInInspector][SerializeField]
protected List<IntPayload> _IntPayloads;
[HideInInspector][SerializeField]
protected List<int> _IntPayloadIdxs;
```

Note that the examples listed are marked as **`protected`**. This keeps access restricted to the class and its subclasses.

Custom MIDI Conversion Support

Custom MIDI conversion support is added to a Koreography Track by implementing the `IMIDIConvertible` interface. If a Koreography Track specified in the MIDI Converter implements this interface, a special **Use Custom Conversion** button will appear. Pressing that button will call the `ConvertMIDIEvents` method which should handle converting the parsed MIDI data into Koreography Event information.

No special payload type is required to utilize this feature.

The IMIDIConvertible Interface

The `IMIDIConvertible` interface defines the following method:

- `void ConvertMIDIEvents(List<KoreoMIDIEvent> events);`

The **ConvertMIDIEvents** method takes a List of KoreoMIDIEvent structs as its only parameter. Each KoreoMIDIEvent has the following fields:

- int **note:** The MIDI note value; an integer in the range [0, 127].
- int **velocity:** The MIDI velocity value; an integer in the range [0, 127].
- int **startSample:** The start sample position of this event.
- int **endSample:** The end sample position of this event.

The **ConvertMIDIEvents** method should use the above information to generate KoreographyEvent instances and, optionally, payloads to attach to them.

Please see the example implementation included with this demo for more information.

Editor Support

The Koreography Editor is aware of custom Koreography Track types. When a custom Koreography Track type is added to a project, the Koreography Editor will provide a dropdown list with available options when the **New** button is clicked in the **Track Settings** area. When selected, a new Koreography Track of the specified type will be generated and added to the active Koreography.

Feature Restrictions

Some Editor features provided by Koreographer require that Koreography Tracks support a specific subset of payload types to function properly. These are outlined here:

- **MIDI Converter:** Koreography Tracks must support one or both of the following:
  - FloatPayload, IntPayload, and TextPayload types.
  - Implement the IMIDIConvertible interface.
- **Analysis (RMS):** Koreography Tracks must support the CurvePayload and FloatPayload types.
- **Analysis (FFT):** Koreography Tracks must support the SpectrumPayload type.

If the required payload types are not supported by the active Koreography Track type, the features will be disabled and a descriptive error or warning will appear.

Restricting Custom Koreography Track and Payload Types

If you have a custom payload type or custom koreography track that you would like to keep out of the Koreography Editor, you may add the **[NoEditorCreate]** attribute to your class. The Koreography Editor will ignore Koreography Track and payload classes marked with this attribute for data generation purposes. If instances of classes with this attribute exist in Koreography data that is *opened* in the Koreography Editor, however, then the Koreography Editor will display the contents as normal.

Please note that the **[NoEditorCreate]** attribute can be found in the **SonicBloom.Koreo** namespace.

Included Examples

The Custom Payload Demo contains two working examples: the MaterialPayload example and the MIDIPayload example.

The scripts included with these examples are heavily documented and should provide plenty of helpful insight. Duplicating them as a foundation for your own custom implementations is a very good way to get started! Not only will you maintain the working example, but your modifications won't be overwritten with future updates to Koreographer!

The MaterialPayload Example

The `MaterialPayload` example focuses on basic custom payload type implementation.

The functionality is split up across two files:

1. **MaterialPayload.cs:** Contains the `MaterialPayload` class implementation. The `MaterialPayload` enables `KoreographyEvent` instances to contain direct references to *Material* assets.
2. **CustomKoreographyTrack.cs:** Contains the `CustomKoreographyTrack` class implementation. This class extends the built-in `KoreographyTrack` class (providing access to all built-in payload types) and adds support for the `MaterialPayload` type.

The MIDIPayload Example

The `MIDIPayload` example focuses on adding support for custom MIDI conversion to a custom Koreography Track type.

The functionality is split up across two files:

1. **MIDIPayload.cs:** Contains the `MIDIPayload` class implementation. The `MIDIPayload` allows `KoreographyEvent` instances to store both MIDI note and MIDI velocity values at the same time. These payloads may be created and modified by hand in the Koreography Editor.
2. **MIDIKoreoTrack.cs:** Contains the `MIDIKoreoTrack` class implementation. This class extends the `KoreographyTrackBase` class and adds support for the `MIDIPayload` type. As a result, the `MIDIPayload` type is the only payload type supported by `KoreographyEvent` instances in a `MIDIKoreoTrack` instance. The `MIDIKoreoTrack` class implements the `IMIDIConvertible` interface, allowing it to hook into the MIDI Converter's custom conversion functionality.

Demo Content Namespace

As with all demo content, the classes contained within the example scripts outlined above are added to the **`SonicBloom.Koreo.Demos`** namespace. You are encouraged to use your own namespace (or remove the existing one altogether) if you choose to use, modify, or duplicate the demo scripts.