OpenTimelineIO Documentation

Release 0.10.0.dev1

Pixar Animation Studios

May 29, 2020

Contents

CHAPTER 1

Overview

OpenTimelineIO (OTIO) is an API and interchange format for editorial cut information. You can think of it as a modern Edit Decision List (EDL) that also includes an API for reading, writing, and manipulating editorial data. It also includes a plugin system for translating to/from existing editorial formats as well as a plugin system for linking to proprietary media storage schemas.

OTIO supports clips, timing, tracks, transitions, markers, metadata, etc. but not embedded video or audio. Video and audio media are referenced externally. We encourage 3rd party vendors, animation studios and visual effects studios to work together as a community to provide adaptors for each video editing tool and pipeline.

CHAPTER 2

Links

OpenTimelineIO Home Page

OpenTimelineIO Discussion Group

CHAPTER 3

Latest Presentation

OpenTimelineIO FMX April 2018 Presentation Slides

CHAPTER 4

Quick Start

4.1 Quickstart

This is for users who wish to get started using the "OTIOView" application to inspect the contents of editorial timelines.

4.1.1 Install Prerequisites

OTIOView has an additional prerequisite to OTIO:

- Try `pip install PySide2`
- If that doesn't work, try downloading PySide2 here: https://wiki.qt.io/Qt_for_Python

You probably want the prebuilt binary for your platform. PySide2 generally includes a link to the appropriate version of Qt as well.

4.1.2 Install OTIO

- `pip install opentimelineio`

4.1.3 Configure Environment Variables for extra adapters

By default, when you install OTIO you will only get the "Core" adapters, which include CMX EDL, Final Cut Pro 7 XML, and the built in JSON format. In order to get access to the "contrib" adapters (which includes the maya sequencer, rv and others) you'll need to set some environment variables. If you need support for these formats, please consult the Adapters documentation page for details

4.1.4 Run OTIOView

Once you have pip installed OpenTimelineIO, you should be able to run:

```
otioview path/to/your/file.edl
```

CHAPTER 5

Tutorials

5.1 Adapters

OpenTimelineIO supports, or plans to support, conversion adapters for many existing file formats.

5.1.1 Final Cut Pro XML

Final Cut 7 XML Format

- Status: Supported via the `fcp_xml` adapter
- Reference

Final Cut Pro X XML Format:

- Status
- Intro to FCP X XML

5.1.2 Adobe Premiere Project

- Based on guidance from Adobe, we support interchange with Adobe Premiere via the FCP 7 XML format (see above).

5.1.3 CMX3600 EDL

- Status: Supported via the `cmx_3600` adapter
- Includes support for ASC_CDL color correction metadata
- Full specification: SMPTE 258M-2004 "For Television Transfer of Edit Decision Lists"
- http://xmil.biz/EDL-X/CMX3600.pdf

- Reference

5.1.4 Avid AAF

- Status: Supports reading simple AAF compositions
  - Reading
  - Writing
- Spec
- Protocol
- set `${OTIO_AAF_PYTHON_LIB}` to point the location of the PyAAF module.

5.1.5 Contrib Adapters

The contrib area hosts adapters which come from the community (*not* supported by the core-otio team) and may require extra dependencies.

RV Session File

- Status: write-only adapter supported via the `rv_session` adapter.
- need to set environment variables to locate `py-interp` and `rvSession.py` from within the RV distribution
- set `${OTIO_RV_PYTHON_BIN}` to point at `py-interp` from within rv, for example: `setenv OTIO_RV_PYTHON_BIN /Applications/RV64.app/Contents/MacOS/py-interp`
- set `${OTIO_RV_PYTHON_LIB}` to point at the parent directory of `rvSession.py`: `setenv OTIO_RV_PYTHON_LIB /Applications/RV64.app/Contents/src/python`

Maya Sequencer

- Status: supported via the `maya_sequencer` adapter.
- set `${OTIO_MAYA_PYTHON_BIN}` to point the location of `mayapy` within the maya installation.

HLS Playlist

- Status: supported via the `hls_playlist` adapter.

Avid Log Exchange (ALE)

- Status: supported via the `ale` adapter.

Text Burn-in Adapter

Uses FFmpeg to burn text overlays into video media.

- Status: supported via the `burnins` adapter.

5.2 Architecture

5.2.1 Overview

OpenTimelineIO is an open source library for the interchange of editorial information. This document describes the structure of the python library.

To import the library into python: `import opentimelineio as otio`

5.2.2 Canonical Structure

Although you can compose your OTIO files differently if you wish, the canonical OTIO structure is as follows:

- root: `otio.schema.Timeline`. This file contains information about the root of a timeline, including a global start offset, and a top level container, `.tracks`
- `timeline.tracks`: This member is a `otio.schema.Stack` which contains `otio.schema.Track` objects
- `timeline.tracks[i]`: The `otio.schema.Track` contained by a `timeline.tracks` contain the clips, transitions and subcontainers that compose the rest of the editorial data.

5.2.3 Modules

The most interesting pieces of OTIO to a developer integrating OTIO into another application or workflow are:

- `otio.schema`: the classes that describe the in-memory OTIO representation
- `otio.opentime` Classes and utility functions for representing time, time ranges and time transforms.
- `otio.adapters`: modules that can read and or write to or from an on-disk format and the in-memory OTIO representation.

Additionally, for developers integrating OTIO into a studio pipeline:

- `otio.media_linker`: plugin system for writing studio or workflow specific media linkers that run after adapters read files

5.2.4 otio.schema

The in-memory OTIO representation data model is rooted at an `otio.schema.Timeline` which has a member `.tracks` which is a `otio.schema.Stack` of `otio.schema.Sequences`, which contain either `otio.schema.Clip` or `otio.schema.Gap`. The `otio.schema.Clip` objects can reference media through a `otio.media_reference.External` or indicate that they are missing a reference to real media with a `otio.media_reference.MissingReference`. All objects have a metadata dictionary for blind data.

Schema composition objects (`otio.schema.Stack` and `otio.schema.Sequence`) implement the python mutable sequence API. A simple script that prints out each shot might look like:

```python
import opentimelineio as otio

# read the timeline into memory
tl = otio.adapters.read_from_file("my_file.otio")

for each_seq in tl.tracks:
```

(continues on next page)

(continued from previous page)

```
  for each_item in each_seq:
   if isinstance(each_item, otio.schema.Clip):
    print each_item.media_reference
```

or, in the case of any nested composition, like this:

```
import opentimelineio as otio

# read the timeline into memory
tl = otio.adapters.read_from_file("my_file.otio")

for clip in tl.each_clip():
  print clip.media_reference
```

5.2.5 Time on otio.schema.Clip

A clip may set its timing information (which is used to compute its `duration()` or its `trimmed_range()`) by configuring either its:

- `.media_reference.available_range` This is the range of the available media that can be cut in. So for example, frames 10-100 have been rendered and prepared for editorial.
- `.source_range` The range of media that is cut into the sequence, in the space of the available range (if it is set). In other words, it further truncates the available_range.

A clip must have at least one set or else its duration is not computable.

```
cl.duration()
CannotComputeAvailableRangeError: No available_range set on media reference on clip:␣
↪Clip("example", External("file:///example.mov"), None, {})
```

You may query the `available_range` and `trimmed_range` via accessors on the `Clip()` itself, for example:

```
cl.trimmed_range()
cl.available_range() # == cl.media_reference.available_range
```

Generally, if you want to know the range of a clip, we recommend using the `.trimmed_range()` method, since this takes both the `media_reference.available_range` and the `.source_range` into consideration.

5.2.6 Time On Clips in Containers

Additionally, if you want to know the time of a clip in the context of a container, you can use the local: `.trimmed_range_in_parent()` method, or a parent's `.trimmed_range_of_child()`. These will additionally take into consideration the `.source_range` of the parent container, if it is set. They return a range in the space of the specified parent container.

5.2.7 otio.opentime

Opentime encodes timing related information.

RationalTime

A point in time at `rt.value*(1/rt.rate)` seconds. Can be rescaled into another RationalTime's rate.

TimeRange

A range in time. Encodes the start time and the duration, meaning that end_time_inclusive (last portion of a sample in the time range) and end_time_exclusive can be computed.

5.2.8 otio.adapters

OpenTimelineIO includes several adapters for reading and writing from other file formats. The `otio.adapters` module has convenience functions that will auto-detect which adapter to use, or you can specify the one you want.

Adapters can be added to the system (outside of the distribution) via JSON files that can be placed on the `OTIO_PLUGIN_MANIFEST_PATH` environment variable to be made available to OTIO.

Most common usage only cares about:

- `timeline = otio.adapters.read_from_file(filepath)`
- `timeline = otio.adapters.read_from_string(rawtext, adapter_name)`
- `otio.adapters.write_to_file(timeline, filepath)`
- `rawtext = otio.adapters.write_to_string(timeline, adapter_name)`

The native format serialization (`.otio` files) is handled via the "otio_json" adapter, `otio.adapters.otio_json`.

In most cases you don't need to worry about adapter names, just use `otio.adapters.read_from_file` and `otio.adapters.write_to_file` and it will figure out which one to use based on the filename extension.

For more information, see How To Write An OpenTimelineIO Adapter

5.2.9 otio.media_linkers

Media linkers run on the otio file after an adapter calls `.read_from_file` or `.read_from_string`. They are intended to replace media references that exist after the adapter runs (which depending on the adapter are likely to be `MissingReference`) with ones that point to valid files in the local system. Since media linkers are plugins, they can use proprietary knowledge or context and do not need to be part of OTIO itself.

You may also specify a media linker to be run after the adapter, either via the `media_linker_name` argument to `.read_from_file` or `.read_from_string` or via the `OTIO_DEFAULT_MEDIA_LINKER` environment variable. You can also turn the media linker off completely by setting the `media_linker_name` argument to `otio.media_linker.MediaLinkingPolicy.DoNotLinkMedia`.

For more information about writing media linkers, see How To Write An OpenTimelineIO Media Linker

5.2.10 Example Scripts

Example scripts are located in the examples subdirectory.

5.3 Contributing

We're excited to collaborate with the community and look forward to the many improvements you can make to OpenTimelineIO!

5.3.1 Contributor License Agreement

Before contributing code to OpenTimelineIO, we ask that you sign a Contributor License Agreement (CLA). At the root of the repo you can find the two possible CLAs:

- OTIO_CLA_Corporate.pdf: please sign this one for corporate use
- OTIO_CLA_Individual.pdf: please sign this one if you're an individual contributor

Once your CLA is signed, send it to `opentimelineio-cla@pixar.com` (please make sure to include your github username) and wait for confirmation that we've received it. After that, you can submit pull requests.

5.3.2 Coding Conventions

Please follow the coding convention and style in each file and in each library when adding new files.

5.3.3 Git Workflow

Here is the workflow we recommend for working on OpenTimelineIO if you intend on contributing changes back:

Post an issue on github to let folks know about the feature or bug that you found, and mention that you intend to work on it. That way, if someone else is working on a similar project, you can collaborate, or you can get early feedback which can sometimes save time.

Use the github website to fork your own private repository.

Clone your fork to your local machine, like this:

```
git clone https://github.com/you/OpenTimelineIO.git
```

Add Pixar's OpenTimelineIO repo as upstream to make it easier to update your remote and local repos with the latest changes:

```
cd OpenTimelineIO
git remote add upstream https://github.com/PixarAnimationStudios/OpenTimelineIO.git
```

Now you fetch the latest changes from Pixar's OpenTimelineIO repo like this:

```
git fetch upstream
```

All the development should happen against the `master` branch. We recommend you create a new branch for each feature or fix that you'd like to make and give it a descriptive name so that you can remember it later. You can checkout a new branch and create it simultaneously like this:

```
git checkout -b mybugfix upstream/master
```

Now you can work in your branch locally.

Once you are happy with your change, you can verify that the change didn't cause tests failures by running tests like this:

```
make test
make lint
```

If all the tests pass and you'd like to send your change in for consideration, push it to your remote repo:

```
git push origin mybugfix
```

Now your remote branch will have your `mybugfix` branch, which you can now pull request (to OpenTimelineIO's `master` branch) using the github UI.

Please make sure that your pull requests are clean. Use the rebase and squash git facilities as needed to ensure that the pull request is as clean as possible.

5.4 Feature Matrix

Adapters may or may not support all of the features of OpenTimelineIO or the format they convert to/from. Here is a list of features and which adapters do/don't support those features.

Feature	OTIO	EDL	FCPXML	AAF	RV	ALE
Single Track of Clips	yes	yes	yes	read-only	write-only	yes
Multiple Video Tracks	yes	no	yes	read-only	write-only	flattened
Audio Tracks & Clips	yes	needs testing	yes	read-only	?	yes
Gap/Filler	yes	yes	yes	read-only	yes	skipped
Markers	yes	yes	yes	planned	planned	no
Nesting	yes	no	yes	read-only	write-only	flattened
Transitions	yes	yes	planned	read-only	write-only	no
Audio/Video Effects	planned	needs research	planned	planned	planned	no
Speed Effects	yes	yes	planned	read-only	planned	no
Color Decision List (CDL)	metadata	yes	planned	planned	planned	planned

5.5 File Format Specification

5.5.1 Version

```
This may be out of date.
```

This DRAFT describes the OpenTimelineIO JSON File Format as of OTIO Alpha 5.

5.5.2 Note

We strongly recommend that you use the OpenTimelineIO library to read and write OTIO files instead of implementing your own parser or writer. However, there will undoubtedly be cases where this is not practical, so we document the format here for clarity.

5.5.3 Naming

OpenTimelineIO files should have a `.otio` path extension. Please do not use `.json` to name OTIO files.

5.5.4 MIME Type/UTI

TODO: Should we specify a MIME Type `application/vnd.pixar.opentimelineio+json` and UTI `com.pixar.opentimelineio` for OTIO, or just use MIME: `application/json` and UTI: `public.json`?

5.5.5 Contents

OpenTimelineIO files are serialized as JSON (http://www.json.org)

5.5.6 Structure

An OTIO file is a tree structure of nested OTIO objects. Each OTIO object is stored as a JSON dictionary with member fields, each of which may contain simple data types or nested OTIO objects.

OTIO does not support instancing, so you cannot reference the same object multiple times in the tree structure. If the same clip or media appears multiple times in your timeline, it will appear as identical copies of the Clip or MediaReference object.

The top level object in an OTIO file can be any OTIO data type, but is typically a Timeline. This means that most use cases will assume that the top level object is a Timeline, but in specific workflows, you can read and write otio files that contain just a Clip, Sequence, RationalTime, or any other OTIO data type. Due to the nature of JSON, you could also read/write an array of objects, but we recommend that you use the OTIO SerializableCollection data type in this case so that you can attach metadata to the container itself. Code that reads an OTIO file should guard against unexpected top level types and fail gracefully. Note also, that this is the reason that there is no top level file format version in OTIO. Each data type has a version instead to allow for more granular versioning.

Each OTIO object has an `"OTIO_SCHEMA"` key/value pair that identifies the OTIO data type and version of that type. For example `"OTIO_SCHEMA": "Timeline.1"` or `"OTIO_SCHEMA": "Clip.5"`. This allows future versions of OTIO to change the serialization details of each data type independently and introduce new data types over time. (TODO: Link to discussion on schema versioning.)

Member fields of each data type are encoded as key/value pairs in the containing object's dictionary. The value of each key can be a JSON string, number, list, or dictionary. If the value is a dictionary, then it will often be an OTIO data type. In some cases (specifically metadata) it can be a regular JSON dictionary.

OTIO JSON files are typically formatted with indentation to make them easier to read. This makes the files slightly larger, but dramatically improves human readability which makes debugging much easier. Since human readablility and ease of use are explicit goals of the OpenTimelineIO project, we recommend that you do not minify OTIO JSON unless absolutely necessary. If file size is really important, you should probably gzip them instead.

5.5.7 Nesting

A Timeline has one child, called "tracks" which is a Stack. Each of that Stack's children is a Sequence. From there on down each child can be any of these types: Clip, Filler, Stack, Sequence.

In a simple case with one track of 3 clips, you have:

```
Timeline "my timeline"
  Stack "tracks"
    Sequence "video track"
      Clip "intro"
      Clip "main"
      Clip "credits"
```

In order to make the tree structure easy to traverse, we use the name "children" for the list of child objects in each parent (except for Timeline's "tracks").

5.5.8 Metadata

TODO: Explain how metadata works and why we do it that way.

5.5.9 Example:

```
{
    "OTIO_SCHEMA": "Timeline.1",
    "metadata": {},
    "name": "transition_test",
    "tracks": {
        "OTIO_SCHEMA": "Stack.1",
        "children": [
            {
                "OTIO_SCHEMA": "Sequence.1",
                "children": [
                    {
                        "OTIO_SCHEMA": "Transition.1",
                        "metadata": {},
                        "name": "t0",
                        "transition_type": "SMPTE_Dissolve",
                        "parameters": {},
                        "in_offset": {
                            "OTIO_SCHEMA" : "RationalTime.1",
                            "rate" : 24,
                            "value" : 10
                        },
                        "out_offset": {
                            "OTIO_SCHEMA" : "RationalTime.1",
                            "rate" : 24,
                            "value" : 10
                        }
                    },
                    {
                        "OTIO_SCHEMA": "Clip.1",
                        "effects": [],
                        "markers": [],
                        "media_reference": null,
                        "metadata": {},
                        "name": "A",
                        "source_range": {
                            "OTIO_SCHEMA": "TimeRange.1",
                            "duration": {
                                "OTIO_SCHEMA": "RationalTime.1",
                                "rate": 24,
                                "value": 50
                            },
                            "start_time": {
                                "OTIO_SCHEMA": "RationalTime.1",
                                "rate": 24,
                                "value": 0.0
                            }
                        }

                    },
                    {
                        "OTIO_SCHEMA": "Transition.1",
                        "metadata": {},
                        "name": "t1",
                        "transition_type": "SMPTE_Dissolve",
                        "parameters": {},
```

(continues on next page)

(continued from previous page)

```
                        "in_offset": {
                            "OTIO_SCHEMA" : "RationalTime.1",
                            "rate" : 24,
                            "value" : 10
                        },
                        "out_offset": {
                            "OTIO_SCHEMA" : "RationalTime.1",
                            "rate" : 24,
                            "value" : 10
                        }
                    },
                    {
                        "OTIO_SCHEMA": "Clip.1",
                        "effects": [],
                        "markers": [],
                        "media_reference": null,
                        "metadata": {},
                        "name": "B",
                        "source_range": {
                            "OTIO_SCHEMA": "TimeRange.1",
                            "duration": {
                                "OTIO_SCHEMA": "RationalTime.1",
                                "rate": 24,
                                "value": 50
                            },
                            "start_time": {
                                "OTIO_SCHEMA": "RationalTime.1",
                                "rate": 24,
                                "value": 0.0
                            }
                        }

                    },
                    {
                        "OTIO_SCHEMA": "Clip.1",
                        "effects": [],
                        "markers": [],
                        "media_reference": null,
                        "metadata": {},
                        "name": "C",
                        "source_range": {
                            "OTIO_SCHEMA": "TimeRange.1",
                            "duration": {
                                "OTIO_SCHEMA": "RationalTime.1",
                                "rate": 24,
                                "value": 50
                            },
                            "start_time": {
                                "OTIO_SCHEMA": "RationalTime.1",
                                "rate": 24,
                                "value": 0.0
                            }
                        }

                    },
                    {
                        "OTIO_SCHEMA": "Transition.1",
```

(continues on next page)

(continued from previous page)

```
                        "metadata": {},
                        "name": "t3",
                        "transition_type": "SMPTE_Dissolve",
                        "parameters": {},
                        "in_offset": {
                            "OTIO_SCHEMA" : "RationalTime.1",
                            "rate" : 24,
                            "value" : 10
                        },
                        "out_offset": {
                            "OTIO_SCHEMA" : "RationalTime.1",
                            "rate" : 24,
                            "value" : 10
                        }
                    }

                ],
                "effects": [],
                "kind": "Video",
                "markers": [],
                "metadata": {},
                "name": "Sequence1",
                "source_range": null
            }
        ],
        "effects": [],
        "markers": [],
        "metadata": {},
        "name": "tracks",
        "source_range": null
    }
}
```

5.5.10 Schema Specification

Each OTIO data type (schema) defines its own key/value pairs. Here are the details of each.

This list has the expanded fields of each concrete object type that you are likely to encounter in an OTIO file. It does not have the intermediate parent classes (Composition, Item, SerializableObject, etc.)

Timeline.1

- name (string)
- tracks (Stack)
- global_start_time (RationalTime)
- metadata (dict)

Sequence.1

- name (string)
- children (any Composable objects, like Clip, Transition, Filler, Stack, Sequence)

- metadata (dict)
- kind (string enumeration SequenceKind.Video, etc.)
- source_range (TimeRange, may be null)
- effects (list of Effect)
- markers (list of Markers)

Stack.1

- name (string)
- children (any Composable objects, like Clip, Transition, Filler, Stack, Sequence)
- metadata (dict)
- source_range (TimeRange, may be null)
- effects (list of Effect)
- markers (list of Markers)

Clip.1

- name (string)
- metadata (dict)
- source_range (TimeRange, may be null)
- effects (list of Effect)
- markers (list of Markers)
- media_reference (MediaReference, may be null)

MediaReference.1

- name (string)
- available_range (TimeRange, may be null)
- metadata (dict)

Filler.1

- name (string)
- metadata (dict)
- source_range (TimeRange, may be null)
- effects (list of Effect)
- markers (list of Markers)

Transition.1

- name (string)
- metadata (dict)
- transition_type (string)
- in_offset (TimeRange)
- out_offset (TimeRange)

RationalTime.1

- value (double)
- rate (double)

TimeRange.1

- start_time (RationalTime)
- duration (RationalTime)

5.6 Timeline Structure

An OpenTimelineIO `Timeline` object can contain many tracks, nested stacks, clips, gaps, and transitions. This document is meant to clarify how these objects nest within each other, and how they work together to represent an audio/video timeline.

5.6.1 Simple Cut List

Let's start with a simple cut list of a few clips. This is stored as a single `Timeline` with a single `Track` which contains several `Clip` children, spliced end-to-end.

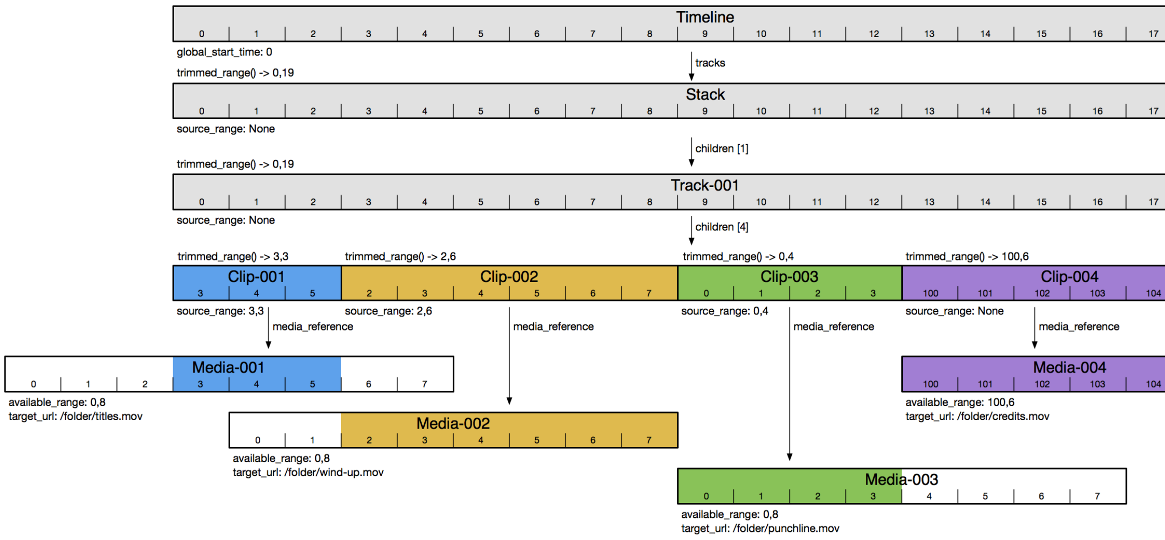

Figure 1 - Simple Cut List
1 - Simple Cut List

Since a `Timeline` can hold multiple tracks, it always has a top-level `Stack` object to hold its `Track` children. In this case, that `Stack` has just one `Track`, named "Track-001".

Within "Track-001", there are four `Clip` objects, named "Clip-001", "Clip-002", "Clip-003", and "Clip-004". Each `Clip` has a corresponding media reference, "Media-001", "Media-002", etc.

At the bottom level, we see that each media reference has a target_url and an available_range. The target_url tells us where to find the media (e.g. a file path or network URL, etc.) The available_range specifies the range of media that is available in the file that it points to. An available_range is a `TimeRange` object which specifies a start_time and duration. The start_time and duration are each `RationalTime` objects, which store a value and rate. Thus we can use `RationalTime(7,24)` to mean frame 7 at 24 frames per second. In the diagram we write this as just 7 for brevity.

In this case most of our media references have an available_range that starts at 0 for some number of frames. One of the media references starts at 100. Assuming the media is 24 frames per second, this means that the media file contains media that starts at 4 seconds and 4 frames (timecode 00:00:04:04).

In many cases you might not know the available_range because the media is missing, or points to a file path or URL which might be expensive to query. If that's the case, then the available_range of a media_reference will be None.

Above the media references, we see that each `Clip` has a source_range, which specifies a trimmed segment of media. In cases where we only want a portion of the available media, the source_range will start at a higher start_time, and/or have a shorter duration. The colored segments of "Media-001", "Media-002" and "Media-003" show the portion that each clip's source_range references.

In the case of "Clip-004", the source_range is None, so it exposes its entire media reference. In the OTIO API, you can query the trimmed_range() of a clip to get the range of media used regardless of whether it has a source_range, available_range or both - but it needs at least one of the two.

Also note that a clip's source_range could refer to a segment outside the available_range of its media reference. That is fine, and comes up in practice often (e.g. I only rendered the first half of my shot). OTIO itself does no snapping or verification of this, but downstream applications may handle this in a variety of ways.

The single `Track` in this example contains all four clips in order. You can ask the `Track` or `Stack` for its trimmed_range() or duration() and it will sum up the trimmed lengths of its children. In later examples, we will see cases where a `Track` or `Stack` is trimmed by setting a source_range, but in this example they are not trimmed.

5.6.2 Transitions

A `Transition` is a visual effect, like a cross dissolve or wipe, that blends two adjacent items on the same track. The most common case is a fade or cross-dissolve between two clips, but OTIO supports transitions between any two `Composable` items (`Clips`, `Gaps`, or nested `Tracks` or `Stacks`).

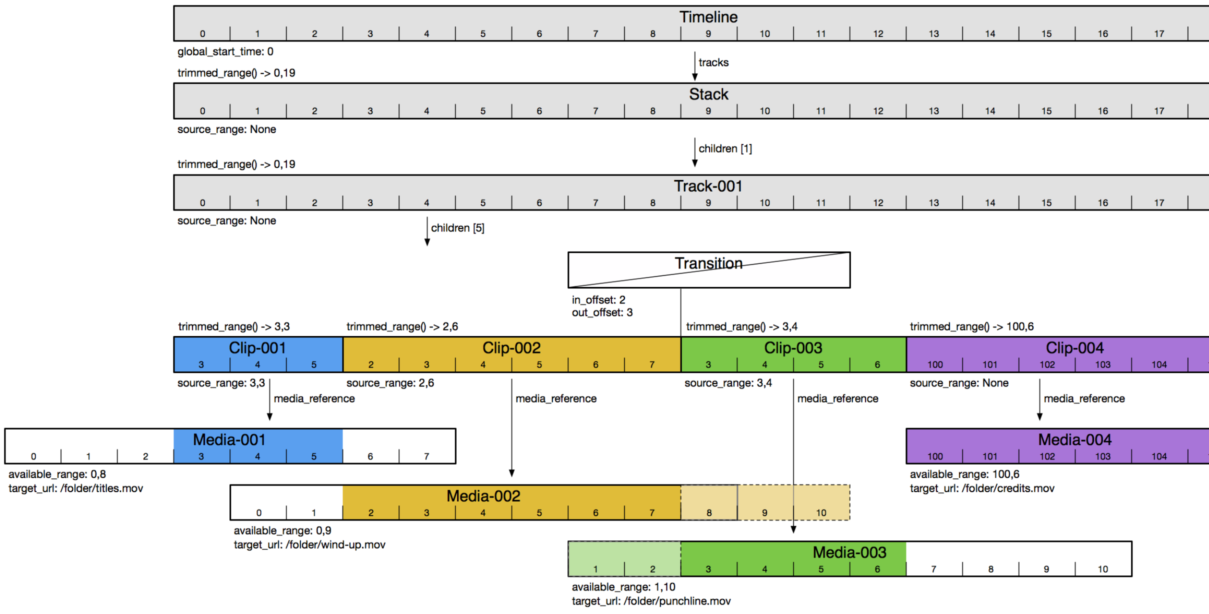

Figure 2 - Transitions
2 - Transitions

In Figure 2, there is a `Transition` between "Clip-002" and "Clip-003". The in_offset and out_offset of the `Transition` specify how much media from the adjacent clips is used by the transition.

Notice that the `Transition` itself does not make "Track-001" any shorter or longer. If a playback tool is not able to render a transition, it may simply ignore transitions and the overall length of the timeline will not be affected.

In Figure 2, the `Transition`'s in_offset of 2 frames means that frames 1 and 2 of "Media-003" are used in the cross dissolve. The out_offset of 3 frames means that frames 8, 9, 10 of "Media-002" are used. Notice that "Media-002"'s available_range is 2 frames too short to satisfy the desired length of the cross-dissolve. OTIO does not prevent you from doing this, as it may be important for some use cases. OTIO also does not specify what a playback tool might display in this case.

A `Transition`'s in_offset and out_offset are not allowed to extend beyond the duration of the adjacent clips. If a clip has transitions at both ends, the two transitions are not allowed to overlap. Also, you cannot place two transitions next to each other in a track; there must be a composable item between them.

A fade to or from black will often be represented as a transition to or from a `Gap`, which can be 0 duration. If multiple tracks are present note that a `Gap` is meant to be transparent, so you may need to consider using a `Clip` with a `GeneratorReference` if you require solid black or any other solid color.

5.6.3 Multiple Tracks

A more typical timeline will include multiple video tracks. In Figure 3, the top-level `Stack` now contains "Track-001", "Track-002", and "Track-003" which contain some `Clip` and `Gap` children. Figure 3 also shows a flattened copy of the timeline to illustrate how multitrack composition works.

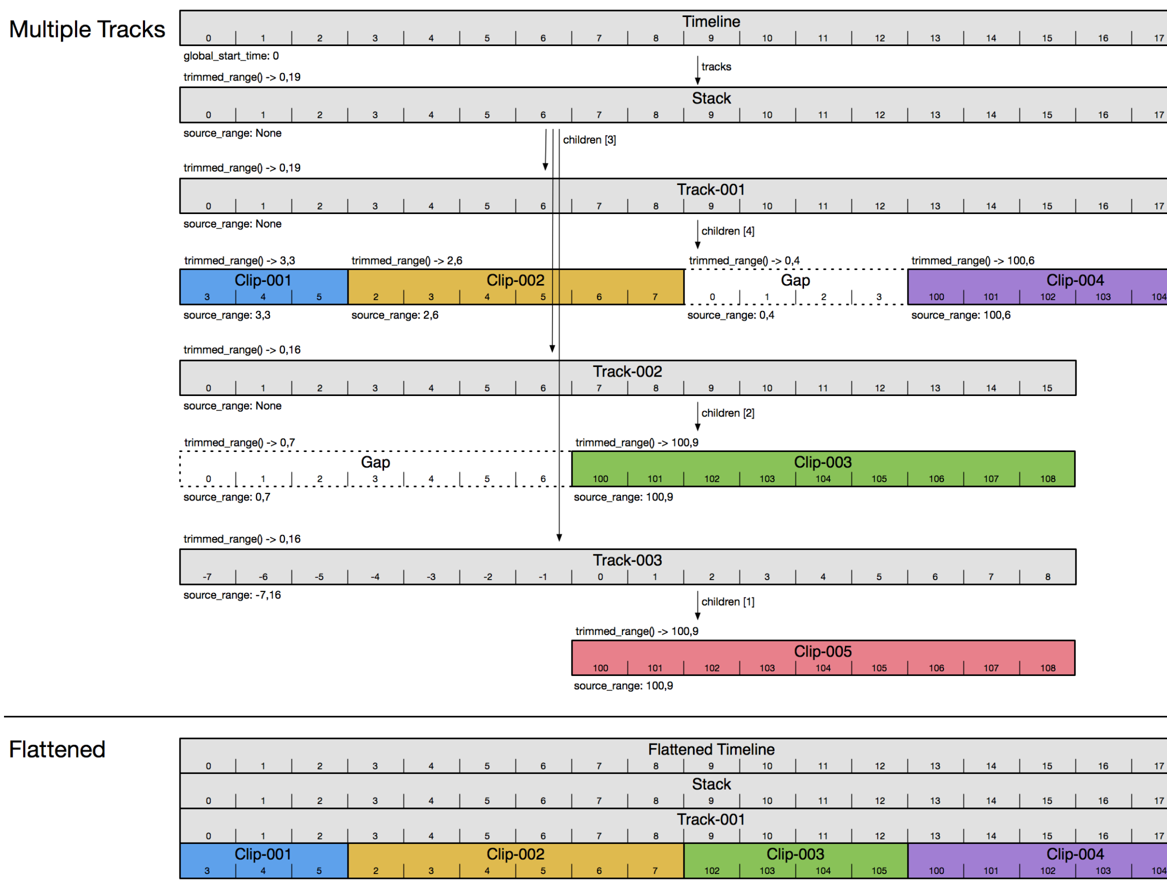

Figure 3 - Multiple Tracks
3 - Multiple Tracks

The Gap in “Track-001” is 4 frames long, and the track below, “Track-002”, has frames 102-105 of “Clip-003” aligned with the Gap above, so those frames show through in the resulting flattened Track.

Note that the Gap at the front of “Track-002” is used just to offset “Clip-003”. This is a common way to shift clips around on a track, but you may also use the Track’s source_range to do this, as illustrated in “Track-003”.

“Clip-005” is completely obscured by “Clip-003” above it, so “Clip-005” does not appear in the flattened timeline at all.

You might also notice that “Track-001” is longer than the other two. If you wanted “Track-002” to be the same length, then you would need to append a Gap at the end. If you wanted “Track-003” to be the same length, then you could extend the duration of its source_range to the desired length. In both cases, the trimmed_range() will be the same.

5.6.4 Nested Compositions

The children of a Track can be any Composable object, which includes Clips, Gaps, Tracks, Stacks, and Transitions. In Figure 4 we see an example of a Stack nested within a Track.

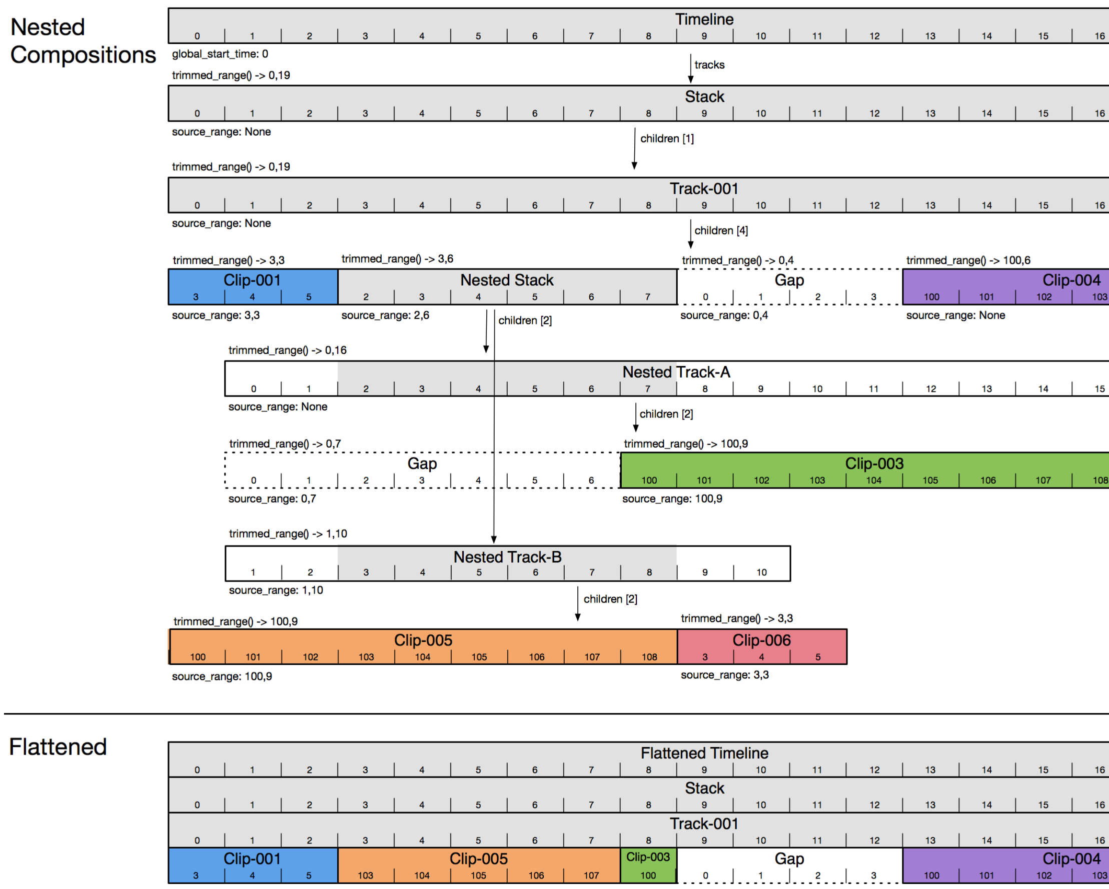

Figure 4 - Nested Compositions

4 - Nested Compositions

In this example, the top-level `Stack` contains only one `Track`. “Track-001” contains four children, “Clip-001”, “Nested Stack”, “Gap”, and “Clip-004”. By nesting a `Composition` (either `Track` or `Stack`) we can refer to a `Composition` as though it was just another `Clip` in the outer `Composition`. If a source_range is specified, then only a trimmed segment of the inner `Composition` is included. In this case that is frames 2 through 7 of “Nested Stack”. If no source_range is specified, then the full available_range of the nested composition is computed and included in the outer composition.

“Nested Stack” contains two tracks, with some clips, gaps, a track-level source_range on the lower track. This illustrates how the content of “Nested Stack” is composed upwards into “Track-001” so that a trimmed portion of “Clip-005” and “Clip-003” appear in the flattened composition.

Notice how the `Gap` in “Track-001” cannot see anything inside the nested composition (“Clip-003”, etc.) because those are not peers to “Track-001”, they are nested within “Nested Stack” and do not spill over into adjacent `Gaps`. In other words, “Nested Stack” behaves just like a `Clip` that happens to have complex contents rather than a simple media reference.

5.7 Time Ranges

5.7.1 Overview

A Timeline and all of the Tracks and Stacks it contains work together to place the Clips, Gaps and Transitions relative to each other in time. You can think of this as a 1-dimensional coordinate system. Simple cases of assembling Clips into a Track will lay the Clips end-to-end, but more complex cases involve nesting, cross-dissolves, trimming, and speed-up/slow-down effects which can lead to confusion. In an attempt to make this easy to work with OpenTimelineIO uses the following terminology and API for dealing with time ranges.

Note: You probably also want to refer to Timeline Structure.

5.7.2 Clips

There are several ranges you might want from a Clip. For each of these, it is important to note which time frame (the 1-dimensional coordinate system of time) the range is relative to. We call these the "Clip time frame" and the "parent time frame" (usually the Clip's parent Track).

Ranges within the Clip and its media:

`Clip.available_range()`

The `available_range()` method on Clip returns a TimeRange that tells you how much media is available via the Clip's `media_reference`. If a Clip points to a movie file on disk, then this should tell you how long that movie is and what timecode it starts at. For example: "wedding.mov" starts at timecode 01:00:00:00 and is 30 minutes long.

Note that `available_range()` may return `None` if the range is not known.

`Clip.source_range`

Setting the `source_range` property on a Clip will trim the Clip to only that range of media.

The `source_range` is specified in the Clip time frame.

Note that `source_range` may be `None` indicating that the Clip should use the full `available_range()` whatever that may be. If both `source_range` and `available_range()` are None, then the Clip is invalid. You need at least one of those.

Usually this will be a shorter segment than the `available_range()` but this is not a hard constraint. Some use cases will intentionally ask for a Clip that is longer (or starts earlier) than the available media as a way to request that new media (a newly animated take, or newly recorded audio) be made to fill the requested `source_range`.

`Clip.trimmed_range()`

This will return the Clip's `source_range` if it is set, otherwise it will return the `available_range()`. This tells you how long the Clip is meant to be in its parent Track or Stack.

The `trimmed_range()` is specified in the Clip time frame.

`Clip.visible_range()`

This will return the same thing as `trimmed_range()` but also takes any adjacent Transitions into account. For example, a Clip that is trimmed to end at frame 10, but is followed by a cross-dissolve with `out_offset` of 5 frames, will have a `visible_range()` that ends at frame 15.

The `visible_range()` is specified in the Clip time frame.

`Clip.duration()`

This is the way to ask for the Clip's "natural" duration. In `oitoview` or most common non-linear editors, this is the duration of the Clip you will see in the timeline user interface.

`Clip.duration()` is a convenience for `Clip.trimmed_range().duration()`. If you want a different duration, then you can ask for `Clip.available_range().duration()` or `Clip.visible_range().duration()` explicitly. This makes it clear in your code when you are asking for a different duration.

Ranges of the Clip in its parent Track or Stack:

`Clip.range_in_parent()`

The answer to this depends on what type of the Clip's parent. In the typical case, the Clip is inside a Track, so the `Clip.range_in_parent()` will give you the range within that Track where this Clip is visible. Each clip within the Track will have a start time that is directly after the previous clip's end. So, given a Track with clipA and clipB in it, this is always true:

The `range_in_parent()` is specified in the parent time frame.

`clipA.range_in_parent().end_time_exclusive() == clipB.range_in_parent().start_time`

If the parent is a Stack, then `range_in_parent()` is less interesting. The start time will always have `.value == 0` and the duration is the Clip's duration. This means that the start of each clip in a Stack is aligned. If you want to shift them around, then use a Stack of Tracks (like the top-level Timeline has by default) and then you can use Gaps to shift the contents of each Track around.

`Clip.trimmed_range_in_parent()`

This is the same as `Clip.range_in_parent()` but trimmed to the *parent* `source_range`. In most cases the parent has a `source_range` of None, so there is no trimming, but in cases where the parent is trimmed, you may want to ask where a Clip is relative to the trimmed parent. In cases where the Clip is completely trimmed by the parent, the `Clip.trimmed_range_in_parent()` will be None.

The `trimmed_range_in_parent()` is specified in the parent time frame.

5.7.3 Tracks

TODO.

5.7.4 Markers

Markers can be attached to any Item (Clips, Tracks, Stacks, Gaps, etc.)

Each Marker has a `marked_range` which specifies the position and duration of the Marker relative to the object it is attached to.

The `marked_range` of a Marker on a Clip is in the Clip's time frame (same as the Clip's `source_range`, `trimmed_range()`, etc.)

The `marked_range` of a Marker on a Track is in the Track's time frame (same as the Track's `source_range`, `trimmed_range()`, etc.)

The `marked_range.duration.value` may be 0 if the Marker is meant to be a instantaneous moment in time, or some other duration if it spans a length of time.

5.7.5 Transitions

TODO.

5.7.6 Gaps

TODO.

5.7.7 Stacks

TODO.

5.7.8 Timelines

TODO.

5.8 Writing an OTIO Adapter

OpenTimelineIO Adapters are plugins that allow OTIO to read and/or write other timeline formats.

Users of OTIO can read and write files like this:

```
#/usr/bin/env python
import opentimelineio as otio
mytimeline = otio.adapters.read_from_file("something.edl")
otio.adapters.write_to_file(mytimeline, "something.otio")
```

The otio.adapters module will look at the file extension (in this case ".edl" or ".otio") and pick the right adapter to convert to or from the appropriate format.

Note that the OTIO JSON format is treated like an adapter as well. The ".otio" format is the only format that is lossless. It can store and retrieve all of the objects, metadata and features available in OpenTimelineIO. Other formats are lossy - they will only store and retrieve features that are supported by that format (and by the adapter implementation). Some adapters may choose to put extra information, not supported by OTIO, into metadata on any OTIO object.

5.8.1 Registering Your Contrib Adapter

To create a new contrib OTIO Adapter, you need to create a file `myadapter.py` in the `opentimelineio_contrib/adapters` folder. Then add an entry to `opentimelineio_contrib/adapters/contrib_adapters.plugin_manifest.json` that registers your new adapter. Note that contrib adapters are community supported, and not supported by the OTIO team. They must still provide unit tests and pass both testing and linting before they are accepted into the repository.

Custom Adapters

Alternately, if you are creating a site specific adapter that you do *not* intend to share with the community, you can create your `myadapter.py` file anywhere. In this case, you must create a `mysite.plugin_manifest.json` (with an entry like the below example that points at `myadapter.py`) and then put the path to your `mysite.plugin_manifest.json` on your `$OTIO_PLUGIN_MANIFEST_PATH` environment variable, which is ":" separated.

For example, to register `myadapter.py` that supports files with a `.myext` file extension:

```
{
    "OTIO_SCHEMA" : "Adapter.1",
    "name" : "myadapter",
    "execution_scope" : "in process",
    "filepath" : "myadapter.py",
    "suffixes" : ["myext"]
}
```

Currently (as of OTIO Beta 8) only execution_scope "in process" is supported. If your adapter needs to execute non-Python code, then we intend to support execution_scope "external process" in the future.

Packaging and Sharing Custom Adapters

Adapters may also be organized into their own independent Python project for subsequent packaging, distribution and installation by `pip`. We recommend you organize your project like so:

```
.
├── setup.py
└── opentimelineio_mystudio
    ├── __init__.py
    ├── plugin_manifest.json
    ├── adapters
    │   ├── __init__.py
    │   ├── my_adapter_x.py
    │   └── my_adapter_y.py
    └── operations
        ├── __init__.py
        └── my_linker.py
```

With a `setup.py` containing this minimum entry set:

```
from setuptools import setup

setup(
    name='OpenTimelineMyStudioAdapters',
    entry_points={
        'opentimelineio.plugins': 'opentimelineio_mystudio = opentimelineio_mystudio'
    },
    package_data={
        'opentimelineio_mystudio': [
            'plugin_manifest.json',
        ],
    },
    version='0.0.1',
    packages=[
        'opentimelineio_mystudio',
```

(continues on next page)

(continued from previous page)

```
        'opentimelineio_mystudio.adapters',
        'opentimelineio_mystudio.operations',
    ],
)
```

And a `plugin_manifest.json` like:

```
{
    "OTIO_SCHEMA" : "PluginManifest.1",
    "adapters" : [
        {
            "OTIO_SCHEMA" : "Adapter.1",
            "name" : "adapter_x",
            "execution_scope" : "in process",
            "filepath" : "adapters/my_adapter_x.py",
            "suffixes" : ["xxx"]
        },
        {
            "OTIO_SCHEMA" : "Adapter.1",
            "name" : "adapter_y",
            "execution_scope" : "in process",
            "filepath" : "adapters/my_adapter_y.py",
            "suffixes" : ["yyy", "why"]
        }
    ],
    "media_linkers" : [
        {
            "OTIO_SCHEMA" : "MediaLinker.1",
            "name" : "my_studios_media_linker",
            "execution_scope" : "in process",
            "filepath" : "operations/my_linker.py"
        }
    ]
}
```

5.8.2 Required Functions

Each adapter must implement at least one of these functions:

```
def read_from_file(filepath):
    ...
    return timeline

def read_from_string(input_str):
    ...
    return timeline

def write_to_string(input_otio):
    ...
    return text

def write_to_file(input_otio, filepath):
    ...
    return
```

If your format is text-based, then we recommend that you implement `read_from_string` and `write_to_string`. The adapter module will automatically wrap these and allow users to call `read_from_file` and `write_to_file`.

5.8.3 Constructing a Timeline

To construct a Timeline in the `read_from_string` or `read_from_file` functions, you can use the API like this:

```
timeline = otio.schema.Timeline()
timeline.name = "Example Timeline"
track = otio.schema.Sequence()
track.name = "V1"
timeline.tracks.append(track)
clip = otio.schema.Clip()
clip.name = "Wedding Video"
track.append(clip)
```

Metadata

If your timeline, tracks, clips or other objects have format-specific, application-specific or studio-specific metadata, then you can add metadata to any of the OTIO schema objects like this:

```
timeline.metadata["mystudio"] = {
    "showID": "zz"
}
clip.metadata["mystudio"] = {
    "shotID": "zz1234",
    "takeNumber": 17,
    "department": "animation",
    "artist": "hanna"
}
```

Note that all metadata should be nested inside a sub-dictionary (in this example "mystudio") so that metadata from other applications, pipeline steps, etc. can be kept separate. OTIO carries this metadata along blindly, so you can put whatever you want in there (within reason). Very large data should probably not go in there.

Media References

Clip media (if known) should be linked like this:

```
clip.media_reference = otio.media_reference.External(
    target_url="file://example/movie.mov"
)
```

Some formats don't support direct links to media, but focus on metadata instead. It is fine to leave the media_reference empty (None) if your adapter doesn't know a real file path or URL for the media.

Source Range vs Available Range

To specify the range of media used in the Clip, you must set the Clip's source_range like this:

```
clip.source_range = otio.opentime.TimeRange(
    start_time=otio.opentime.RationalTime(150, 24), # frame 150 @ 24fps
    duration=otio.opentime.RationalTime(200, 24) # 200 frames @ 24fps
)
```

Note that the source_range of the clip is not necessarily the same as the available_range of the media reference. You may have a clip that uses only a portion of a longer piece of media, or you might have some media that is too short for the desired clip length. Both of these are fine in OTIO. Also, clips can be relinked to different media, in which case the source_range of the clip stays the same, but the media_reference (and its available_range) will change after the relink. For example, you might relink from an old render to a newer render which has been extended to cover the source_range references by the clip.

If you know the range of media available at that Media Reference's URL, then you can specify it like this:

```
clip.media_reference = otio.media_reference.External(
  target_url="file://example/movie.mov",
  available_range=otio.opentime.TimeRange(
      start_time=otio.opentime.RationalTime(100, 24), # frame 100 @ 24fps
      duration=otio.opentime.RationalTime(500, 24) # 500 frames @ 24fps
  )
)
```

It is fine to leave the Media Reference's available_range empty if you don't know it, but you should always specify a Clip's source_range.

5.8.4 Traversing a Timeline

When exporting a Timeline in the `write_to_string` or `write_to_file` functions, you will need to traverse the Timeline data structure. Some formats only support a single track, so a simple adapter might work like this:

```
def write_to_string(input_otio):
    """Turn a single track timeline into a very simple CSV."""
    result = "Clip,Start,Duration\n"
    if len(input_otio.tracks) != 1:
        raise Exception("This adapter does not support multiple tracks.")
    for item in input_otio.each_clip():
        start = otio.opentime.to_seconds(item.source_range.start_time)
        duration = otio.opentime.to_seconds(item.source_range.duration)
        result += ",".join([item.name, start, duration]) + "\n"
    return result
```

More complex timelines will contain multiple tracks and nested sequences. OTIO supports nesting via the abstract Composition class, with two concrete subclasses, Sequence and Stack. In general a Composition has children, each of which is an Item. Since Composition is also a subclass of Item, they can be nested arbitrarily.

In typical usage, you are likely to find that a Timeline has a Stack (the property called 'tracks'), and each item within that Stack is a Sequence. Each item within a Sequence will usually be a Clip, Transition or Gap. If you don't support Transitions, you can just skip them and the overall timing of the Sequence should still work.

If the format your adapter supports allows arbitrary nesting, then you should traverse the composition in a general way, like this:

```
def export_otio_item(item):
    result = MyThing(item)
    if isinstance(item, otio.core.Composition):
```

(continues on next page)

(continued from previous page)

```
        result.children = map(export_otio_item, item.children)
    return result
```

If the format your adapter supports has strict expectations about the structure, then you should validate that the input has the expected structure and then traverse it based on those expectations, like this:

```
def export_timeline(timeline):
    result = MyTimeline(timeline.name, ...)
    for track in timeline.tracks:
        if !isinstance(track, otio.schema.Sequence):
            raise Exception("This adapter requires each track to be a sequence, not a
↪"+typeof(track))
      t = result.AddTrack(track.name, ...)
      for clip in track.each_clip():
          c = result.AddClip(clip.name, ...)
    return result
```

5.8.5 Examples

OTIO includes a number of "core" (supported by the core team) adapters in `opentimelineio/adapters` as well as a number of community supported adapters in `opentimelineio_contrib/adapters`.

5.9 Writing an OTIO Media Linker

OpenTimelineIO Media Linkers are plugins that allow OTIO to replace MissingReferences with valid, site specific MediaReferences after an adapter reads a file.

The current MediaLinker can be specified as an argument to `otio.adapters.read_from_file` or via an environment variable. If one is specified, then it will run after the adapter reads the contents of the file before it is returned back.

```
#/usr/bin/env python
import opentimelineio as otio
mytimeline = otio.adapters.read_from_file("something.edl", media_linker_name="awesome_
↪studios_media_linker")
```

After the EDL adapter reads something.edl, the media linker "awesome_studios_media_linker" will run and link the media in the file (if it can).

5.9.1 Registering Your Media Linker

To create a new OTIO Adapter, you need to create a file mymedialinker.py. Then add a manifest that points at that python file:

```
    {
        "OTIO_SCHEMA" : "PluginManifest.1",
        "media_linkers" : [
            {
                "OTIO_SCHEMA" : "MediaLinker.1",
                "name" : "awesome_studios_media_linker",
                "execution_scope" : "in process",
```

(continues on next page)

(continued from previous page)

```
                "filepath" : "mymedialinker.py"
            }
        ],
        "adapters" : [
        ]
    }
```

Then you need to add this manifest to your `$OTIO_PLUGIN_MANIFEST_PATH` environment variable (which is "`:`" separated).

Finally, to specify this linker as the default media linker, set `OTIO_DEFAULT_MEDIA_LINKER` to the name of the media linker:

```
setenv OTIO_DEFAULT_MEDIA_LINKER "awesome_studios_media_linker"
```

To package and share your media linker, follow these instructions.

5.9.2 Writing a Media Linker

To write a media linker, write a python file with a "link_media_reference" function in it that takes two arguments: "in_clip" (the clip to try and add a media reference to) and "media_linker_argument_map" (arguments passed from the calling function to the media linker.

For example:

```python
def link_media_reference(in_clip, media_linker_argument_map):
    d.update(media_linker_argument_map)
    # you'll probably want to set it to something other than a missing reference
    in_clip.media_reference = otio.media_reference.MissingReference(
        name=in_clip.name + "_tweaked",
        metadata=d
    )
```

5.9.3 For Testing

The otioconvert.py script has a –media-linker argument you can use to test out your media linker (once its on the path).

```
env OTIO_PLUGIN_MANIFEST_PATH=mymanifest.otio.json:$OTIO_PLUGIN_MANIFEST_PATH bin/
↪otioconvert.py -i somefile.edl --media-linker="awesome_studios_media_linker" -o /
↪var/tmp/test.otio
```

5.10 Writing a Hook Script

OpenTimelineIO Hook Scripts are plugins that run at predefined points during the execution of various OTIO functions, for example after an adapter has read a file into memory but before the media linker has run.

To write a new hook script, you create a python source file that defines a a function named `hook_function` with signature: `hook_function :: otio.schema.Timeline, Dict => otio.schema.Timeline`

The first argument is the timeline to process, and the second one is a dictionary of arguments that can be passed to it. Only one hook function can be defined per python file.

For example:

```python
def hook_function(tl, arg_dict):
    for cl in tl.each_clip():
        cl.metadata['example_hook_function_was_here'] = True
    return tl
```

This will insert some extra metadata into each clip.

This plugin can then be registered with the system by configuring a plugin manifest.

5.10.1 Registering Your Hook Script

To create a new OTIO hook script, you need to create a file myhooks.py. Then add a manifest that points at that python file:

```
{
    "OTIO_SCHEMA" : "PluginManifest.1",
    "hook_scripts" : [
        {
            "OTIO_SCHEMA" : "HookScript.1",
            "name" : "example hook",
            "execution_scope" : "in process",
            "filepath" : "example.py"
        }
    ],
    "hooks" : {
        "pre_adapter_write" : ["example hook"],
        "post_adapter_read" : []
    }
}
```

The `hook_scripts` section will register the plugin with the system, and the `hooks` section will attach the scripts to hooks.

Then you need to add this manifest to your `$OTIO_PLUGIN_MANIFEST_PATH` environment variable (which is "`:`" separated). You may also define media linkers and adapters via the same manifest.

5.10.2 Running a Hook Script

If you would like to call a hook script from a plugin, the hooks need not be one of the ones that otio pre-defines. You can have a plugin adapter or media linker, for example, that defines its own hooks and calls your own custom studio specific hook scripts. To run a hook script from your custom code, you can call:

`otio.hooks.run("some_hook", some_timeline, optional_argument_dict)`

This will call the `some_hook` hook script and pass in `some_timeline` and `optional_argument_dict`.

5.10.3 Order of Hook Scripts

To query which hook scripts are attached to a given hook, you can call:

```python
import opentimelineio as otio
hook_list = otio.hooks.scripts_attached_to("some_hook")
```

Note that `hook_list` will be in order of execution. You can rearrange this list, or edit it to change which scripts will run (or not run) and in which order.

To Edit the order, change the order in the list:

```
hook_list[0], hook_list[2] = hook_list[2], hook_list[0]
print hook_list # ['c','b','a']
```

Now c will run, then b, then a.

To delete a function the list:

```
del hook_list[1]
```

5.11 Writing an OTIO SchemaDef Plugin

OpenTimelineIO SchemaDef plugins are plugins that define new schemas within the otio type registry system. You might want to do this to add new schemas that are specific to your own internal studio workflow and shouldn't be part of the generic OpenTimelineIO package.

To write a new SchemaDef plugin, you create a Python source file that defines and registers one or more new classes subclassed from `otio.core.SerializeableObject`. Multiple schema classes can be defined and registered in one plugin, or you can use a separate plugin for each of them.

Here's an example of defining a very simple class called `MyThing`:

```
import opentimelineio as otio

@otio.core.register_type
class MyThing(otio.core.SerializableObject):
    """A schema for my thing."""

    _serializable_label = "MyThing.1"
    _name = "MyThing"

    def __init__(
        self,
        arg1=None,
        argN=None
    ):
        otio.core.SerializableObject.__init__(self)
        self.arg1 = arg1
        self.argN = argN

    arg1 = otio.core.serializable_field(
        "arg1",
        doc = ( "arg1's doc string")
    )

    argN = otio.core.serializable_field(
        "argN",
        doc = ( "argN's doc string")
    )

    def __str__(self):
        return "MyThing({}, {})".format(
```

(continues on next page)

(continued from previous page)

```
            repr(self.arg1),
            repr(self.argN)
        )

    def __repr__(self):
        return "otio.schema.MyThing(arg1={}, argN={})".format(
            repr(self.arg1),
            repr(self.argN)
        )
```

In the example, the `MyThing` class has two parameters `arg1` and `argN`, but your schema class could have any number of parameters as needed to contain the data fields you want to have in your class.

One or more class definitions like this one can be included in a plugin source file, which must then be added to the plugin manifest as shown below:

5.11.1 Registering Your SchemaDef Plugin

To create a new SchemaDef plugin, you need to create a Python source file as shown in the example above. Let's call it `mything.py`. Then you must add it to a plugin manifest:

```
{
    "OTIO_SCHEMA" : "PluginManifest.1",
    "schemadefs" : [
        {
            "OTIO_SCHEMA" : "SchemaDef.1",
            "name" : "mything",
            "execution_scope" : "in process",
            "filepath" : "mything.py"
         }
    ]
}
```

The same plugin manifest may also include adapters and media linkers, if desired.

Then you need to add this manifest to your `$OTIO_PLUGIN_MANIFEST_PATH` environment variable (which is "`:`" separated).

5.11.2 Using the New Schema in Your Code

Now that we've defined a new otio schema, how can we create an instance of the schema class in our code (for instance, in an adapter or media linker)? SchemaDef plugins are magically loaded into a namespace called `otio.schemadef`, so you can create a class instance just like this:

```
import opentimelineio as otio

mine = otio.schemadef.my_thing.MyThing(arg1, argN)
```

An alternative approach is to use the `instance_from_schema` mechanism, which requires that you create and provide a dict of the parameters:

```
    mything = otio.core.instance_from_schema("MyThing", 1, {
        "arg1": arg1,
```

(continues on next page)

(continued from previous page)

```
        "argN": argN
    })
```

This instance_from_schema approach has the added benefit of calling the schema upgrade function to upgrade the parameters in the case where the requested schema version is earlier than the current version defined by the schemadef plugin. This seems rather unlikely to occur in practice if you keep your code up-to-date, so the first technique of creating the class instance directly from otio.schemadef is usually preferred.

5.12 Versioning Schemas

During development, it is natural that the fields on objects in OTIO change. To accommodate this, OTIO has a system for handling version differences and upgrading older schemas to new ones. There are two components:

1. serializeable_label on the class has a name and version field: Foo.5 – Foo is the schema name and 5 is the version.
2. upgrade_function_for decorator

5.12.1 Changing a Field

For example, lets say you have class:

```
import opentimelineio as otio

@otio.core.register_type
class SimpleClass(otio.core.SerializeableObject):
  serializeable_label = "SimpleClass.1"
  my_field = otio.core.serializeable_field("my_field", int)
```

And you want to change my_field to new_field. To do this:

- Make the change in the class
- Bump the version number in the label
- add an upgrade function

So after the changes, you'll have:

```
@otio.core.register_type
class SimpleClass(otio.core.SerializeableObject):
  serializeable_label = "SimpleClass.2"
  my_field = otio.core.serializeable_field("new_field", int)

@otio.core.upgrade_function_for(SimpleClass, 2)
def upgrade_one_to_two(data):
  return {"new_field" : data["my_field"] }
```

Lets change it again, so that new_field becomes even_newer_field.

```
@otio.core.register_type
class SimpleClass(otio.core.SerializeableObject):
  serializeable_label = "SimpleClass.2"
  my_field = otio.core.serializeable_field("even_newer_field", int)

```

(continues on next page)

(continued from previous page)

```python
@otio.core.upgrade_function_for(SimpleClass, 2)
def upgrade_one_to_two(data):
  return {"new_field" : data["my_field"] }

# NOTE we now have a second upgrade function
@otio.core.upgrade_function_for(SimpleClass, 3)
def upgrade_two_to_three(data):
  return {"even_newer_field" : data["new_field"] }
```

Upgrade functions can be sparse - if version `3` to `4` doesn't require a function, for example, you don't need to write one.

5.12.2 Adding or Removing a Field

Starting from the same class:

```python
@otio.core.register_type
class SimpleClass(otio.core.SerializeableObject):
  serializeable_label = "SimpleClass.1"
  my_field = otio.core.serializeable_field("my_field", int)
```

Adding or Removing a field is simpler. In these cases, you don't need to write an upgrade function, since any new classes will be initialized through the constructor, and any removed fields will be ignored when reading from an older schema version.

So lets add a new field:

```python
@otio.core.register_type
class SimpleClass(otio.core.SerializeableObject):
  serializeable_label = "SimpleClass.2"
  my_field = otio.core.serializeable_field("my_field", int)
  other_field = otio.core.serializeable_field("other_field", int)
```

And then delete the original field:

```python
@otio.core.register_type
class SimpleClass(otio.core.SerializeableObject):
  serializeable_label = "SimpleClass.3"
  other_field = otio.core.serializeable_field("other_field", int)
```

CHAPTER 6

Use Cases

6.1 Animation Shot Frame Ranges Changed

Status: Planned

6.1.1 Summary

This case is very similar to the Shots Added/Removed from the Cut Use Case. The editorial and animation departments are working with a sequence of shots simultaneously over the course of a few weeks. The initial delivery of rendered video clips from animation to editorial provides enough footage for the editor(s) to work with, at least as a starting point. As the cut evolves, the editor(s) may need more frames at the head or tail of some shots, or they may trim frames from the head or tail that are no longer needed. Usually there is an agreement that some extra frames, called handles, should be present at the head and tail of each shot to give the editors some flexibility. In the case where the editors need more frames than the handles provide, they might use a freeze frame effect, or a slow down effect to stretch the clip, or simply repeat a segment of a clip to fill the gap. This is a sign that new revisions of those shots should be animated and rendered with more frames to fill the needs of the cut. Furthermore, as the sequence nears completion, the cut becomes more stable and the cost of rendering frames becomes higher, so there is a desire to trim unused handles from the shots on the animation side. In both cases, we can use OTIO to compare the frame range of each shot between the two departments.

6.1.2 Example

Animation delivers the first pass of 100 shots to editorial. Editorial makes an initial cut of the sequence. In the cut, several shots are trimmed down to less than half of the initial length, but 2 shots need to be extended. Editorial exports an EDL or AAF of the sequence from Avid Media Composer and gives this cut to the animation department. Animation runs a Python script which compares the frame range of each shot used in the cut to the frame range of the most recent take of each shot being animated. Any shot that is too short must be extended and any shot that is more than 12 frames too long can be trimmed down. The revised shots are animated, re-rendered and re-delivered to editorial. Upon receiving these new deliveries, editorial will cut them into the sequence (see also Use Case: Conform New Renders into the Cut). For shots that used timing effects to temporarily extend them, those effects can be removed, since the new version of those shots is now longer.

6.1.3 Features Needed in OTIO

- EDL reading
  - Clip names for video track
  - Source frame range for each clip
  - Timing effects
- AAF reading
  - Clip names across all video tracks, subclips, etc.
  - Source frame range for each clip
  - Timing effects
- Timeline should include (done)
  - a Stack of tracks, each of which is a Sequence
- Sequence should include (done)
  - a list of Clips
- Clips should include (done)
  - Name
  - Metadata
  - Timing effects
- Timing effects
  - Source frame range of each clip as effected by timing effects.
- Composition
  - Clips in lower tracks that are obscured (totally or partially) by overlapping clips in higher tracks are considered trimmed or hidden.
  - Visible frame range for each clip.

6.1.4 Features of Python Script

- Use OTIO to read the EDL or AAF
- Iterate through every Clip in the Timeline, printing its name and visible frame range

6.2 Conform New Renders Into The Cut

Status: Done This use case is in active use at Pixar (Coco, Incredibles 2, etc.)

6.2.1 Summary

Artists working on the animation or visual effects for shots in a sequence often want to view their in-progress work in the context of a current cut of the film. This could be accomplished by importing their latest renders into the editing system, but that often involves many steps (e.g. transcoding, cutting the clips into the editing system, etc.) Instead, the artists would like to preview the cut with their latest work spliced in at their desk.

6.2.2 Workflow

- In Editorial:

1. Export an EDL from the editorial system (Media Composer, Adobe Premiere, Final Cut Pro X, etc.)
2. Export a QuickTime audio/video mixdown that matches that EDL
3. Send the EDL+ QuickTime to the animators or visual effects artists

- At the Artist's Desk:

1. Open the EDL+QuickTime in a video player tool (RV, etc.)
2. Either: 2a. Use OpenTimelineIO to convert the EDL to OTIO or 2b. A plugin in the video player tool uses OpenTimelineIO to read the EDL.
3. In either case, we link the shots in the timeline to segments of the supplied QuickTime movie.
4. The artist can now play the sequence and see exactly what the editor saw.
5. The artist can now relink any or all of the shots to the latest renders (either via OpenTimelineIO or features of the video player tool)

6.2.3 Features Needed in OTIO

- EDL reading (done)
  - Single video track (done)
  - Clip names (done)
  - Source range for each clip (done)
  - Transitions (done)
- RV support
  - OTIO to RV Session file via adapter (done)
  - RV Plugin that uses OTIO to read an EDL (Pixar has use-case specific code for this) (done)
- Relinking
  - Relinking the whole EDL to segments of the full sequence QuickTime movie. (done)
  - Relinking of individual clips to renders of those shots.(done)

6.3 Shots Added or Removed From The Cut

Status: Planned

6.3.1 Summary

The creative process of editing often involves adding, removing or replacing shots in a sequence. Other groups of people working on the same project want to know about changes to the list of shots in use on a day to day basis. For example, animators working on a shot should be informed if the shot they are working on has been cut from the sequence, so they can stop working on it. Similarly, if new shots are added, animation should start working on those shots. Since the creative decision about which shots are in or out of the cut comes from editorial, we can use OTIO to communicate these changes.

6.3.2 Example

Editorial is working on a short film in Avid Media Composer. They have several bins of media with live action footage, rendered animation clips, dialogue recordings, sound effects and music. The lead editor is actively working on the cut for the short film over the course of a few weeks. At the same time, the animation department is actively working on the animated shots for the film. As revisions are made to the animated shots, rendered clips are delivered to editorial with a well established naming convention. On a daily basis, an EDL or AAF is exported from Media Composer and passed to the animation department so they can stay up to date with the current cut.

In each revision of the cut, Animation wants to know which shots have been added or removed. They run a Python script which uses OpenTimelineIO to read an EDL or AAF from editorial and produces a list of video clip names found in the cut. Some of these names match the animation department's shot naming convention - or contain shot tracking metadata - and can be compared to existing shots that the animators are working on.

If there are shots being animated that are not in this cut, then animation can stop working on those shots, as they are no longer needed.

When the editor wants to request a new shot with a new camera angle or new animation, he or she can duplicate an existing clip and give it a new name, or insert a placeholder with a previously unused name, or otherwise flag the new clip as a request for a new shot. When animation sees this newly requested shot in the cut, they can respond as appropriate and deliver the new shot to editorial when it is ready.

6.3.3 Features Needed in OTIO

- EDL reading (done)
  - Clip names across all tracks
- AAF reading
  - Clip names across all tracks, subclips, etc.
- Timeline should include (done)
  - a Stack of tracks, each of which is a Sequence
- Sequence should include (done)
  - a list of Clips
- Clips should include (done)
  - Name
  - Metadata

6.3.4 Features of Python Script

- Use OTIO to read the EDL or AAF. (done)
- Iterate through every Clip in the Timeline, printing its name. (done)
- Compare these names to the shots in a production tracking system.

CHAPTER 7

API Reference

7.1 opentimelineio package

An editorial interchange format and library.

see: http://opentimeline.io

7.1.1 Subpackages

opentimelineio.adapters package

Expose the adapter interface to developers.

To read from an existing representation, use the read_from_string and read_from_file functions. To query the list of adapters, use the available_adapter_names function.

The otio_json adapter is provided as a the canonical, lossless, serialization of the in-memory otio schema. Other adapters are to varying degrees lossy. For more information, consult the documentation in the individual adapter modules.

`opentimelineio.adapters.`**`available_adapter_names`**`()`

Return a string list of the available adapters.

`opentimelineio.adapters.`**`from_filepath`**`(`*filepath*`)`

Guess the adapter object to use for a given filepath.

example: "foo.otio" returns the "otio_json" adapter.

`opentimelineio.adapters.`**`from_name`**`(`*name*`)`

Fetch the adapter object by the name of the adapter directly.

`opentimelineio.adapters.`**`read_from_file`**`(`*filepath, adapter_name=None, media_linker_name='__default', media_linker_argument_map=None, **adapter_argument_map*`)`

Read filepath using adapter_name.

If adapter_name is None, try and infer the adapter name from the filepath.

For example: timeline = read_from_file("example_trailer.otio") timeline = read_from_file("file_with_no_extension", "cmx_3600")

`opentimelineio.adapters.`**`read_from_string`**(*input_str*, *adapter_name='otio_json'*, *media_linker_name='__default'*, *media_linker_argument_map=None*, ***adapter_argument_map*)

Read a timeline from input_str using adapter_name.

This is useful if you obtain a timeline from someplace other than the filesystem.

Example: raw_text = urlopen(my_url).read() timeline = read_from_string(raw_text, "otio_json")

`opentimelineio.adapters.`**`suffixes_with_defined_adapters`**(*read=False*, *write=False*)

Return a set of all the suffixes that have adapters defined for them.

`opentimelineio.adapters.`**`write_to_file`**(*input_otio*, *filepath*, *adapter_name=None*, ***adapter_argument_map*)

Write input_otio to filepath using adapter_name.

If adapter_name is None, infer the adapter_name to use based on the filepath.

Example: otio.adapters.write_to_file(my_timeline, "output.otio")

`opentimelineio.adapters.`**`write_to_string`**(*input_otio*, *adapter_name='otio_json'*, ***adapter_argument_map*)

Return input_otio written to a string using adapter_name.

Example: raw_text = otio.adapters.write_to_string(my_timeline, "otio_json")

Submodules

opentimelineio.adapters.adapter module

Implementation of the OTIO internal *Adapter* system.

For information on writing adapters, please consult: https://opentimelineio.readthedocs.io/en/latest/tutorials/write-an-adapter.html# # noqa

`class` `opentimelineio.adapters.adapter.`**`Adapter`**(*name=None*, *execution_scope=None*, *filepath=None*, *suffixes=None*)

Bases: *`opentimelineio.plugins.python_plugin.PythonPlugin`*

Adapters convert between OTIO and other formats.

Note that this class is not subclassed by adapters. Rather, an adapter is a python module that implements at least one of the following functions:

> write_to_string(input_otio) write_to_file(input_otio, filepath) (optionally inferred) read_from_string(input_str) read_from_file(filepath) (optionally inferred)

. . . as well as a small json file that advertises the features of the adapter to OTIO. This class serves as the wrapper around these modules internal to OTIO. You should not need to extend this class to create new adapters for OTIO.

For more information: https://opentimelineio.readthedocs.io/en/latest/tutorials/write-an-adapter.html# # noqa

has_feature(*feature_string*)
: return true if adapter supports feature_string, which must be a key of the _FEATURE_MAP dictionary.

  Will trigger a call to self.module(), which imports the plugin.

read_from_file(*filepath*, *media_linker_name='__default'*, *media_linker_argument_map=None*, ***adapter_argument_map*)
: Execute the read_from_file function on this adapter.

  If read_from_string exists, but not read_from_file, execute that with a trivial file object wrapper.

read_from_string(*input_str*, *media_linker_name='__default'*, *media_linker_argument_map=None*, ***adapter_argument_map*)
: Call the read_from_string function on this adapter.

suffixes
: File suffixes associated with this adapter.

write_to_file(*input_otio*, *filepath*, ***adapter_argument_map*)
: Execute the write_to_file function on this adapter.

  If write_to_string exists, but not write_to_file, execute that with a trivial file object wrapper.

write_to_string(*input_otio*, ***adapter_argument_map*)
: Call the write_to_string function on this adapter.

opentimelineio.adapters.cmx_3600 module

OpenTimelineIO CMX 3600 EDL Adapter

class `opentimelineio.adapters.cmx_3600.`**ClipHandler**(*line*, *comment_data*, *rate=24*)
: Bases: `object`

  make_clip(*comment_data*)

  parse(*line*)

class `opentimelineio.adapters.cmx_3600.`**CommentHandler**(*comments*)
: Bases: `object`

  comment_id_map = {'ASC_SAT': 'asc_sat', 'ASC_SOP': 'asc_sop', 'FROM CLIP': 'media_refe

  parse(*comment*)

  regex_template = '\\*?\\s*{id}:?\\s*(?P<comment_body>.*)'

class `opentimelineio.adapters.cmx_3600.`**DissolveEvent**(*a_side_event*, *transition*, *b_side_clip*, *tracks*, *kind*, *rate*, *style*)
: Bases: `object`

  to_edl_format()
  : Example output:

    Cross dissolve… 002 Clip1 V C 00:00:07:08 00:00:07:08 00:00:01:21 00:00:01:21 002 Clip2 V D 100 00:00:09:07 00:00:17:15 00:00:01:21 00:00:10:05 * FROM CLIP NAME: Clip1 * FROM CLIP: /var/tmp/clip1.001.exr * TO CLIP NAME: Clip2 * TO CLIP: /var/tmp/clip2.001.exr

    Fade in… 001 BL V C 00:00:00:00 00:00:00:00 00:00:00:00 00:00:00:00 001 My_Clip V D 012 00:00:02:02 00:00:03:04 00:00:00:00 00:00:01:02 * TO CLIP NAME: My Clip * TO FILE: /var/tmp/clip.001.exr

Fade out… 002 My_Clip V C 00:00:01:12 00:00:01:12 00:00:00:12 00:00:00:12 002 BL V D 012 00:00:00:00 00:00:00:12 00:00:00:12 00:00:01:00 * FROM CLIP NAME: My Clip * FROM FILE: /var/tmp/clip.001.exr

exception opentimelineio.adapters.cmx_3600.**EDLParseError**
Bases: *opentimelineio.exceptions.OTIOError*

class opentimelineio.adapters.cmx_3600.**EDLParser**(*edl_string*, *rate=24*, *ignore_timecode_mismatch=False*)
Bases: object

add_clip(*line*, *comments*, *rate=24*)

add_transition(*clip_handler*, *transition*, *data*)

guess_kind_for_track_name(*name*)

parse_edl(*edl_string*, *rate=24*)

tracks_for_channel(*channel_code*)

class opentimelineio.adapters.cmx_3600.**EDLWriter**(*tracks*, *rate*, *style*)
Bases: object

get_content_for_track_at_index(*idx*, *title*)

class opentimelineio.adapters.cmx_3600.**Event**(*clip*, *tracks*, *kind*, *rate*, *style*)
Bases: object

to_edl_format()

Example output: 002 AX V C 00:00:00:00 00:00:00:05 00:00:00:05 00:00:00:10 * FROM CLIP NAME: test clip2 * FROM FILE: S:vartmptest.exr

class opentimelineio.adapters.cmx_3600.**EventLine**(*kind*, *rate*)
Bases: object

is_dissolve()

to_edl_format(*edit_number*)

opentimelineio.adapters.cmx_3600.**read_from_string**(*input_str*, *rate=24*, *ignore_timecode_mismatch=False*)
Reads a CMX Edit Decision List (EDL) from a string. Since EDLs don't contain metadata specifying the rate they are meant for, you may need to specify the rate parameter (default is 24). By default, read_from_string will throw an exception if it discovers invalid timecode in the EDL. For example, if a clip's record timecode overlaps with the previous cut. Since this is a common mistake in many EDLs, you can specify ignore_timecode_mismatch=True, which will supress these errors and attempt to guess at the correct record timecode based on the source timecode and adjacent cuts. For best results, you may wish to do something like this:

Example:

```
>>> try:
...      timeline = otio.adapters.read_from_string("mymovie.edl", rate=30)
... except EDLParseError:
...    print('Log a warning here')
...    try:
...        timeline = otio.adapters.read_from_string(
...            "mymovie.edl",
...            rate=30,
...            ignore_timecode_mismatch=True)
...    except EDLParseError:
...        print('Log an error here')
```

`opentimelineio.adapters.cmx_3600.`**`write_to_string`**(*input_otio*, *rate=None*, *style='avid'*)

opentimelineio.adapters.fcp_xml module

OpenTimelineIO Final Cut Pro 7 XML Adapter.

`opentimelineio.adapters.fcp_xml.`**`read_from_string`**(*input_str*)

`opentimelineio.adapters.fcp_xml.`**`write_to_string`**(*input_otio*)

opentimelineio.adapters.otio_json module

This adapter lets you read and write native .otio files

`opentimelineio.adapters.otio_json.`**`read_from_file`**(*filepath*)

`opentimelineio.adapters.otio_json.`**`read_from_string`**(*input_str*)

`opentimelineio.adapters.otio_json.`**`write_to_file`**(*input_otio*, *filepath*)

`opentimelineio.adapters.otio_json.`**`write_to_string`**(*input_otio*)

opentimelineio.algorithms package

Algorithms for OTIO objects.

Submodules

opentimelineio.algorithms.filter module

Algorithms for filtering OTIO files.

`opentimelineio.algorithms.filter.`**`filtered_composition`**(*root*, *unary_filter_fn*, *types_to_prune=None*)

Filter a deep copy of root (and children) with unary_filter_fn.

types_to_prune:: tuple of types, example: (otio.schema.Gap,…)

1. Make a deep copy of root
2. Starting with root, perform a depth first traversal
3. **For each item (including root):**
   a. **if types_to_prune is not None and item is an instance of a type** in types_to_prune, prune it from the copy, continue.
   b. **Otherwise, pass the copy to unary_filter_fn. If unary_filter_fn:**
      I. returns an object: add it to the copy, replacing original
      II. returns a tuple: insert it into the list, replacing original
      III. returns None: prune it
4. If an item is pruned, do not traverse its children
5. Return the new deep copy.

EXAMPLE 1 (filter):

If your unary function is:

def fn(thing):

if thing.name == B: return thing' # some transformation of B

else: return thing

If you have a track: [A,B,C]

filtered_composition(track, fn) => [A,B',C]

EXAMPLE 2 (prune):

If your unary function is:

def fn(thing):

if thing.name == B: return None

else: return thing

filtered_composition(track, fn) => [A,C]

EXAMPLE 3 (expand):

If your unary function is:

def fn(thing):

if thing.name == B: return tuple(B_1,B_2,B_3)

else: return thing

filtered_composition(track, fn) => [A,B_1,B_2,B_3,C]

EXAMPLE 4 (prune gaps):

track :: [Gap, A, Gap]

filtered_composition(track, lambda _:_, types_to_prune=(otio.schema.Gap,)) => [A]

`opentimelineio.algorithms.filter.`**`filtered_with_sequence_context`**(*root*, *reduce_fn*, *types_to_prune=None*)

Filter a deep copy of root (and children) with reduce_fn.

reduce_fn::function(previous_item, current, next_item) (see below) types_to_prune:: tuple of types, example: (otio.schema.Gap,. . .)

1. Make a deep copy of root
2. Starting with root, perform a depth first traversal
3. **For each item (including root):**

   a. **if types_to_prune is not None and item is an instance of a type** in types_to_prune, prune it from the copy, continue.

   b. **Otherwise, pass (prev, copy, and next) to reduce_fn. If reduce_fn:**

      I. returns an object: add it to the copy, replacing original

      II. returns a tuple: insert it into the list, replacing original

      III. returns None: prune it

**** note that reduce_fn is always passed objects from the original** deep copy, not what prior calls return. See below for examples

4. If an item is pruned, do not traverse its children
5. Return the new deep copy.

EXAMPLE 1 (filter):

```
>>> track = [A,B,C]
>>> def fn(prev_item, thing, next_item):
...     if prev_item.name == A:
...         return D # some new clip
...     else:
...         return thing
>>> filtered_with_sequence_context(track, fn) => [A,D,C]
```

order of calls to fn: fn(None, A, B) => A fn(A, B, C) => D fn(B, C, D) => C # !! note that it was passed B instead of D.

EXAMPLE 2 (prune):

```
>>> track = [A,B,C]
>>> def fn(prev_item, thing, next_item):
...    if prev_item.name == A:
...        return None # prune the clip
...   else:
...        return thing
>>> filtered_with_sequence_context(track, fn) => [A,C]
```

order of calls to fn: fn(None, A, B) => A fn(A, B, C) => None fn(B, C, D) => C # !! note that it was passed B instead of D.

EXAMPLE 3 (expand):

```
>>> def fn(prev_item, thing, next_item):
...     if prev_item.name == A:
...         return (D, E) # tuple of new clips
...     else:
...         return thing
>>> filtered_with_sequence_context(track, fn) => [A, D, E, C]
```

the order of calls to fn will be: fn(None, A, B) => A fn(A, B, C) => (D, E) fn(B, C, D) => C # !! note that it was passed B instead of D.

opentimelineio.algorithms.stack_algo module

Algorithms for stack objects.

`opentimelineio.algorithms.stack_algo.`**`flatten_stack`**`(`*in_stack*`)`

Flatten a Stack, or a list of Tracks, into a single Track. Note that the 1st Track is the bottom one, and the last is the top.

opentimelineio.algorithms.track_algo module

Algorithms for track objects.

`opentimelineio.algorithms.track_algo.`**`track_trimmed_to_range`**`(`*in_track*, *trim_range*`)`

Returns a new track that is a copy of the in_track, but with items outside the trim_range removed and items on the ends trimmed to the trim_range. Note that the track is never expanded, only shortened. Please note that you could do nearly the same thing non-destructively by just setting the Track's source_range but sometimes you want to really cut away the stuff outside and that's what this function is meant for.

`opentimelineio.algorithms.track_algo.`**`track_with_expanded_transitions`**`(`*in_track*`)`

Expands transitions such that neighboring clips are trimmed into regions of overlap.

For example, if your track is: Clip1, T, Clip2

will return: Clip1', Clip1_t, T, Clip2_t, Clip2'

Where Clip1' is the part of Clip1 not in the transition, Clip1_t is the part inside the transition and so on.

opentimelineio.console package

Console scripts for OpenTimelineIO

Submodules

opentimelineio.console.otiocat module

Print the contents of an OTIO file to stdout.

`opentimelineio.console.otiocat.`**`main`**`()`

Parse arguments and call _otio_compatible_file_to_json_string.

opentimelineio.console.otioconvert module

Python wrapper around OTIO to convert timeline files between formats.

Available adapters: ['fcp_xml', 'otio_json', 'cmx_3600', 'fcpx_xml', 'hls_playlist', 'rv_session', 'maya_sequencer', 'ale', 'burnins', 'AAF']

`opentimelineio.console.otioconvert.`**`main`**`()`

Parse arguments and convert the files.

opentimelineio.console.otiostat module

Print statistics about the otio file, including validation information.

`opentimelineio.console.otiostat.`**`main`**`()`

main entry point

`opentimelineio.console.otiostat.`**`stat_check`**`(`*name*`)`

opentimelineio.core package

Internal implementation details of OpenTimelineIO.

Submodules

opentimelineio.core.composable module

Composable class definition.

An object that can be composed by tracks.

class opentimelineio.core.composable.**Composable**(*name=None*, *metadata=None*)
Bases: *opentimelineio.core.serializable_object.SerializableObject*

An object that can be composed by tracks.

Base class of: Item Transition

is_parent_of(*other*)
Returns true if self is a parent or ancestor of other.

metadata
Metadata dictionary for this Composable.

name
Composable name.

static overlapping()
Return whether an Item is overlapping. By default False.

parent()
Return the parent Composable, or None if self has no parent.

static visible()
Return the visibility of the Composable. By default True.

opentimelineio.core.composition module

Composition base class. An object that contains *Items*.

class opentimelineio.core.composition.**Composition**(*name=None*, *children=None*, *source_range=None*, *markers=None*, *effects=None*, *metadata=None*)
Bases: *opentimelineio.core.item.Item*, collections.abc.MutableSequence

Base class for an OTIO Item that contains other Items.

Should be subclassed (for example by Track and Stack), not used directly.

children_at_time(*t*)
Which children overlap time t?

composition_kind
Returns a label specifying the kind of composition.

each_child(*search_range=None*, *descended_from_type=<class 'opentimelineio.core.composable.Composable'>*)

handles_of_child(*child*)
If media beyond the ends of this child are visible due to adjacent Transitions (only applicable in a Track) then this will return the head and tail offsets as a tuple of RationalTime objects. If no handles are present on either side, then None is returned instead of a RationalTime.

Example usage: >>> head, tail = track.handles_of_child(clip) >>> if head: … print('Do something') >>> if tail: … print('Do something else')

insert(*index*, *item*)
Insert an item into the composition at location *index*.

range_of_child(*child*, *reference_space=None*)
The range of the child in relation to another item (reference_space), not trimmed based on this composition's source_range.

Note that reference_space must be in the same timeline as self.

For example:

[—–] | seq

[—————–] Clip A

If ClipA has duration 17, and seq has source_range: 5, duration 15, seq.range_of_child(Clip A) will return (0, 17) ignoring the source range of seq.

To get the range of the child with the source_range applied, use the trimmed_range_of_child() method.

range_of_child_at_index(*index*)
Return the range of a child item in the time range of this composition.

For example, with a track: [ClipA][ClipB][ClipC]

The self.range_of_child_at_index(2) will return: TimeRange(ClipA.duration + ClipB.duration, ClipC.duration)

To be implemented by subclass of Composition.

top_clip_at_time(*t*)
Return the first visible child that overlaps with time t.

transform
Deprecated field, do not use.

trim_child_range(*child_range*)

trimmed_range_of_child(*child*, *reference_space=None*)
Get range of the child in reference_space coordinates, after the self.source_range is applied.

Example | [—–] | seq [—————–] Clip A

If ClipA has duration 17, and seq has source_range: 5, duration 10, seq.trimmed_range_of_child(Clip A) will return (5, 10) Which is trimming the range according to the source_range of seq.

To get the range of the child without the source_range applied, use the range_of_child() method.

Another example | [—–] | seq source range starts on frame 4 and goes to frame 8 [ClipA][ClipB] (each 6 frames long)

```
>>> seq.range_of_child(CLipA)
0, duration 6
>>> seq.trimmed_range_of_child(ClipA):
4, duration 2
```

trimmed_range_of_child_at_index(*index*)

Return the trimmed range of the child item at index in the time range of this composition.

For example, with a track:

[ ]

[ClipA][ClipB][ClipC]

The range of index 2 (ClipC) will be just like range_of_child_at_index() but trimmed based on this Composition's source_range.

To be implemented by child.

opentimelineio.core.item module

Implementation of the Item base class. OTIO Objects that contain media.

class opentimelineio.core.item.**Item**(*name=None*, *source_range=None*, *effects=None*, *markers=None*, *metadata=None*)

Bases: `opentimelineio.core.composable.Composable`

An Item is a Composable that can be part of a Composition or Timeline.

More specifically, it is a Composable that has meaningful duration.

Can also hold effects and markers.

Base class of:

- Composition (and children)
- Clip
- Gap

available_range()

Implemented by child classes, available range of media.

duration()

Convience wrapper for the trimmed_range.duration of the item.

effects

List of effects on this item.

markers

List of markers on this item.

metadata

Metadata dictionary for this item.

name

Item name.

range_in_parent()

Find and return the untrimmed range of this item in the parent.

source_range

Range of source to trim to. Can be None or a TimeRange.

transformed_time(*t*, *to_item*)

Converts time t in the coordinate system of self to coordinate system of to_item.

Note that self and to_item must be part of the same timeline (they must have a common ancestor).

Example:

0 20 [——t—D———] [–A-][t—-B—][–C–] 100 101 110 101 in B = 6 in D

t = t argument

transformed_time_range(*tr*, *to_item*)
Transforms the timerange tr to the range of child or self to_item.

trimmed_range()
The range after applying the source range.

trimmed_range_in_parent()
Find and return the trimmed range of this item in the parent.

static visible()
Return the visibility of the Item. By default True.

visible_range()
The range of this item's media visible to its parent. Includes handles revealed by adjacent transitions (if any). This will always be larger or equal to trimmed_range().

opentimelineio.core.json_serializer module

Serializer for SerializableObjects to JSON

Used for the otio_json adapter as well as for plugins and manifests.

opentimelineio.core.json_serializer.**deserialize_json_from_file**(*otio_filepath*)
Deserialize the file at otio_filepath containing JSON to OTIO.

opentimelineio.core.json_serializer.**deserialize_json_from_string**(*otio_string*)
Deserialize a string containing JSON to OTIO objects.

opentimelineio.core.json_serializer.**serialize_json_to_file**(*root*, *to_file*)
Serialize a tree of SerializableObject to JSON.

Writes the result to the given file path.

opentimelineio.core.json_serializer.**serialize_json_to_string**(*root*, *indent=4*)
Serialize a tree of SerializableObject to JSON.

Returns a JSON string.

opentimelineio.core.media_reference module

Media Reference Classes and Functions.

class opentimelineio.core.media_reference.**MediaReference**(*name=None*, *available_range=None*, *metadata=None*)
Bases: *opentimelineio.core.serializable_object.SerializableObject*

Base Media Reference Class.

Currently handles string printing the child classes, which expose interface into its data dictionary.

The requirement is that the schema is named so that external systems can fetch the required information correctly.

available_range
Available range of media in this media reference.

is_missing_reference

metadata
Metadata dictionary.

name
Name of this media reference.

opentimelineio.core.serializable_object module

Implements the otio.core.SerializableObject

class opentimelineio.core.serializable_object.**SerializableObject**
Bases: `object`

Base object for things that can be [de]serialized to/from .otio files.

To define a new child class of this, you inherit from it and also use the register_type decorator. Then you use the serializable_field function above to create attributes that can be serialized/deserialized.

You can use the upgrade_function_for decorator to upgrade older schemas to newer ones.

Finally, if you're in the process of upgrading schemas and you want to catch code that refers to old attribute names, you can use the deprecated_field function. This raises an exception if code attempts to read or write to that attribute. After testing and before pushing, please remove references to deprecated_field.

For example

```
>>>     import opentimelineio as otio
```

```
>>>     @otio.core.register_type
...     class ExampleChild(otio.core.SerializableObject):
...         _serializable_label = "ExampleChild.7"
...         child_data = otio.core.serializable_field("child_data", int)
```

# @TODO: delete once testing shows nothing is referencing this. >>> old_child_data_name = otio.core.deprecated_field()

```
>>>     @otio.core.upgrade_function_for(ExampleChild, 3)
...     def upgrade_child_to_three(_data):
...         return {"child_data" : _data["old_child_data_name"]}
```

copy()

deepcopy()

is_equivalent_to(*other*)
Returns true if the contents of self and other match.

is_unknown_schema

classmethod schema_name()

classmethod schema_version()

opentimelineio.core.serializable_object.**deprecated_field**()
For marking attributes on a SerializableObject deprecated.

opentimelineio.core.serializable_object.**serializable_field**(*name*, *required_type=None*, *doc=None*)
Create a serializable_field for child classes of SerializableObject.

Convienence function for adding attributes to child classes of SerializableObject in such a way that they will be serialized/deserialized automatically.

Use it like this:

class foo(SerializableObject): bar = serializable_field("bar", required_type=int, doc="example")

This would indicate that class "foo" has a serializable field "bar". So: f = foo() f.bar = "stuff"

# serialize & deserialize otio_json = otio.adapters.from_name("otio") f2 = otio_json.read_from_string(otio_json.write_to_string(f))

# fields should be equal f.bar == f2.bar

Additionally, the "doc" field will become the documentation for the property.

opentimelineio.core.type_registry module

Core type registry system for registering OTIO types for serialization.

`opentimelineio.core.type_registry.`**`instance_from_schema`**(*schema_name*, *schema_version*, *data_dict*)
Return an instance, of the schema from data in the data_dict.

`opentimelineio.core.type_registry.`**`register_type`**(*classobj*, *schemaname=None*)
Register a class to a Schema Label.

Normally this is used as a decorator. However, in special cases where a type has been renamed, you might need to register the new type to multiple schema names. To do this:

```
>>>     @core.register_type
...     class MyNewClass(...):
...         pass
```

```
>>>     core.register_type(MyNewClass, "MyOldName")
```

This will parse the old schema name into the new class type. You may also need to write an upgrade function if the schema itself has changed.

`opentimelineio.core.type_registry.`**`schema_label_from_name_version`**(*schema_name*, *schema_version*)
Return the serializeable object schema label given the name and version.

`opentimelineio.core.type_registry.`**`schema_name_from_label`**(*label*)
Return the schema name from the label name.

`opentimelineio.core.type_registry.`**`schema_version_from_label`**(*label*)
Return the schema version from the label name.

`opentimelineio.core.type_registry.`**`upgrade_function_for`**(*cls*, *version_to_upgrade_to*)
Decorator for identifying schema class upgrade functions.

Example >>> @upgrade_function_for(MyClass, 5) … def upgrade_to_version_five(data): … pass

This will get called to upgrade a schema of MyClass to version 5. My class must be a class deriving from otio.core.SerializableObject.

The upgrade function should take a single argument - the dictionary to upgrade, and return a dictionary with the fields upgraded.

Remember that you don't need to provide an upgrade function for upgrades that add or remove fields, only for schema versions that change the field names.

opentimelineio.core.unknown_schema module

Implementation of the UnknownSchema schema.

class opentimelineio.core.unknown_schema.**UnknownSchema**

Bases: *opentimelineio.core.serializable_object.SerializableObject*

Represents an object whose schema is unknown to us.

data

Exposes the data dictionary of the underlying SerializableObject directly.

is_unknown_schema

opentimelineio.plugins package

Plugin system for OTIO

Submodules

opentimelineio.plugins.manifest module

Implementation of an adapter registry system for OTIO.

opentimelineio.plugins.manifest.**ActiveManifest**(*force_reload=False*)

class opentimelineio.plugins.manifest.**Manifest**

Bases: *opentimelineio.core.serializable_object.SerializableObject*

Defines an OTIO plugin Manifest.

This is an internal OTIO implementation detail. A manifest tracks a collection of adapters and allows finding specific adapters by suffix

For writing your own adapters, consult: https://opentimelineio.readthedocs.io/en/latest/tutorials/write-an-adapter.html#

adapter_module_from_name(*name*)

Return the adapter module associated with a given adapter name.

adapter_module_from_suffix(*suffix*)

Return the adapter module associated with a given file suffix.

adapters

Adapters this manifest describes.

extend(*another_manifest*)

Extend the adapters, schemadefs, and media_linkers lists of this manifest by appending the contents of the corresponding lists of another_manifest.

from_filepath(*suffix*)

Return the adapter object associated with a given file suffix.

from_name(*name*, *kind_list='adapters'*)

Return the adapter object associated with a given adapter name.

hook_scripts

Scripts that can be attached to hooks.

hooks
Hooks that hooks scripts can be attached to.

media_linkers
Media Linkers this manifest describes.

schemadef_module_from_name(*name*)
Return the schemadef module associated with a given schemadef name.

schemadefs
Schemadefs this manifest describes.

`opentimelineio.plugins.manifest.`**load_manifest**()

`opentimelineio.plugins.manifest.`**manifest_from_file**(*filepath*)
Read the .json file at filepath into a Manifest object.

`opentimelineio.plugins.manifest.`**manifest_from_string**(*input_string*)
Deserialize the json string into a manifest object.

opentimelineio.plugins.python_plugin module

Base class for OTIO plugins that are exposed by manifests.

class `opentimelineio.plugins.python_plugin.`**PythonPlugin**(*name=None, execution_scope=None, filepath=None*)
Bases: *opentimelineio.core.serializable_object.SerializableObject*

A class of plugin that is encoded in a python module, exposed via a manifest.

execution_scope
Describes whether this adapter is executed in the current python process or in a subshell. Options are: ['in process', 'out of process'].

filepath
Absolute path or relative path to adapter module from location of json.

module()
Return the module object for this adapter.

module_abs_path()
Return an absolute path to the module implementing this adapter.

name
Adapter name.

opentimelineio.schema package

User facing classes.

Submodules

opentimelineio.schema.clip module

Implementation of the Clip class, for pointing at media.

class `opentimelineio.schema.clip.`**`Clip`**(*name=None*, *media_reference=None*, *source_range=None*, *markers=[]*, *effects=[]*, *metadata=None*)

Bases: *`opentimelineio.core.item.Item`*

The base editable object in OTIO.

Contains a media reference and a trim on that media reference.

`available_range`()

Implemented by child classes, available range of media.

`each_clip`(*search_range=None*)

Yields self.

`media_reference`

`name`

Name of this clip.

`transform`

Deprecated field, do not use.

opentimelineio.schema.effect module

Implementation of Effect OTIO class.

class `opentimelineio.schema.effect.`**`Effect`**(*name=None*, *effect_name=None*, *metadata=None*)

Bases: *`opentimelineio.core.serializable_object.SerializableObject`*

`effect_name`

Name of the kind of effect (example: ‘Blur’, ‘Crop’, ‘Flip’).

`metadata`

Metadata dictionary.

`name`

Name of this effect object. Example: ‘BlurByHalfEffect’.

class `opentimelineio.schema.effect.`**`FreezeFrame`**(*name=None*, *metadata=None*)

Bases: *`opentimelineio.schema.effect.LinearTimeWarp`*

Hold the first frame of the clip for the duration of the clip.

class `opentimelineio.schema.effect.`**`LinearTimeWarp`**(*name=None*, *time_scalar=1*, *metadata=None*)

Bases: *`opentimelineio.schema.effect.TimeEffect`*

A time warp that applies a linear scale across the entire clip

`time_scalar`

Linear time scalar applied to clip. 2.0 = double speed, 0.5 = half speed.

class `opentimelineio.schema.effect.`**`TimeEffect`**(*name=None*, *effect_name=None*, *metadata=None*)

Bases: *`opentimelineio.schema.effect.Effect`*

Base Time Effect Class

opentimelineio.schema.external_reference module

Implementation of the ExternalReference media reference schema.

class opentimelineio.schema.external_reference.**ExternalReference**(*target_url=None, available_range=None, metadata=None*)

Bases: *opentimelineio.core.media_reference.MediaReference*

Reference to media via a url, for example "file:///var/tmp/foo.mov"

target_url

URL at which this media lives. For local references, use the 'file://' format.

opentimelineio.schema.gap module

class opentimelineio.schema.gap.**Gap**(*name=None, duration=None, source_range=None, effects=None, markers=None, metadata=None*)

Bases: *opentimelineio.core.item.Item*

static visible()

Return the visibility of the Item. By default True.

opentimelineio.schema.generator_reference module

Generators are media references that _produce_ media rather than refer to it.

class opentimelineio.schema.generator_reference.**GeneratorReference**(*name=None, generator_kind=None, available_range=None, parameters=None, metadata=None*)

Bases: *opentimelineio.core.media_reference.MediaReference*

Base class for Generators.

Generators are media references that become "generators" in editorial systems. For example, color bars or a solid color.

generator_kind

Kind of generator reference, as defined by the schema.generator_reference.GeneratorReferenceTypes enum.

parameters

Dictionary of parameters for generator.

opentimelineio.schema.marker module

Marker class. Holds metadata over regions of time.

class `opentimelineio.schema.marker.`**`Marker`**(*name=None*, *marked_range=None*, *color='RED'*, *metadata=None*)

Bases: `opentimelineio.core.serializable_object.SerializableObject`

Holds metadata over time on a timeline

`color`
Color string for this marker (for example: 'RED'), based on the otio.schema.marker.MarkerColor enum.

`marked_range`
Range this marker applies to, relative to the Item this marker is attached to (e.g. the Clip or Track that owns this marker).

`metadata`
Metadata dictionary.

`name`
Name of this marker.

`range`
Deprecated field, do not use.

class `opentimelineio.schema.marker.`**`MarkerColor`**

Bases: `object`

Enum encoding colors of markers as strings.

`BLACK = 'BLACK'`

`BLUE = 'BLUE'`

`CYAN = 'CYAN'`

`GREEN = 'GREEN'`

`MAGENTA = 'MAGENTA'`

`ORANGE = 'ORANGE'`

`PINK = 'PINK'`

`PURPLE = 'PURPLE'`

`RED = 'RED'`

`WHITE = 'WHITE'`

`YELLOW = 'YELLOW'`

opentimelineio.schema.missing_reference module

Implementation of the MissingReference media reference schema.

class `opentimelineio.schema.missing_reference.`**`MissingReference`**(*name=None*, *available_range=None*, *metadata=None*)

Bases: `opentimelineio.core.media_reference.MediaReference`

Represents media for which a concrete reference is missing.

`is_missing_reference`

opentimelineio.schema.schemadef module

class `opentimelineio.schema.schemadef.`**`SchemaDef`**(*name=None*, *execution_scope=None*, *filepath=None*)

Bases: *`opentimelineio.plugins.python_plugin.PythonPlugin`*

`module`()

Return the module object for this schemadef plugin. If the module hasn't already been imported, it is imported and injected into the otio.schemadefs namespace as a side-effect. (redefines PythonPlugin.module())

`opentimelineio.schema.schemadef.`**`available_schemadef_names`**()

Return a string list of the available schemadefs.

`opentimelineio.schema.schemadef.`**`from_name`**(*name*)

Fetch the schemadef object by the name of the schema directly.

opentimelineio.schema.serializable_collection module

A serializable collection of SerializableObjects.

class `opentimelineio.schema.serializable_collection.`**`SerializableCollection`**(*name=None*, *children=None*, *metadata=None*)

Bases: *`opentimelineio.core.serializable_object.SerializableObject`*, `collections.abc.MutableSequence`

A kind of composition which can hold any serializable object.

This composition approximates the concept of a *bin* - a collection of SerializableObjects that do not have any compositing meaning, but can serialize to/from OTIO correctly, with metadata and a named collection.

`each_child`(*search_range=None*, *descended_from_type=<class 'opentimelineio.core.composable.Composable'>*)

`each_clip`(*search_range=None*)

`insert`(*index*, *item*)

S.insert(index, value) – insert value before index

`metadata`

Metadata dictionary for this SerializableCollection.

`name`

SerializableCollection name.

opentimelineio.schema.stack module

Implement Track and Stack.

class `opentimelineio.schema.stack.`**`Stack`**(*name=None*, *children=None*, *source_range=None*, *markers=None*, *effects=None*, *metadata=None*)

Bases: *`opentimelineio.core.composition.Composition`*

`available_range`()

Implemented by child classes, available range of media.

each_clip(*search_range=None*)

range_of_child_at_index(*index*)

Return the range of a child item in the time range of this composition.

For example, with a track: [ClipA][ClipB][ClipC]

The self.range_of_child_at_index(2) will return: TimeRange(ClipA.duration + ClipB.duration, ClipC.duration)

To be implemented by subclass of Composition.

trimmed_range_of_child_at_index(*index*, *reference_space=None*)

Return the trimmed range of the child item at index in the time range of this composition.

For example, with a track:

[ ]

[ClipA][ClipB][ClipC]

The range of index 2 (ClipC) will be just like range_of_child_at_index() but trimmed based on this Composition's source_range.

To be implemented by child.

opentimelineio.schema.timeline module

Implementation of the OTIO built in schema, Timeline object.

class opentimelineio.schema.timeline.**Timeline**(*name=None*, *tracks=None*, *global_start_time=None*, *metadata=None*)

Bases: *opentimelineio.core.serializable_object.SerializableObject*

audio_tracks()

This convenience method returns a list of the top-level audio

duration()

Duration of this timeline.

each_child(*search_range=None*, *descended_from_type=<class 'opentimelineio.core.composable.Composable'>*)

each_clip(*search_range=None*)

Return a flat list of each clip, limited to the search_range.

metadata

Metadata dictionary.

name

Name of this timeline.

range_of_child(*child*)

Range of the child object contained in this timeline.

tracks

Stack of tracks containing items.

video_tracks()

This convenience method returns a list of the top-level video

opentimelineio.schema.timeline.**timeline_from_clips**(*clips*)

Convenience for making a single track timeline from a list of clips.

opentimelineio.schema.track module

Implement Track sublcass of composition.

class opentimelineio.schema.track.**NeighborGapPolicy**

Bases: `object`

enum for deciding how to add gap when asking for neighbors

around_transitions = 1

never = 0

class opentimelineio.schema.track.**Track**(*name=None*, *children=None*, *kind='Video'*, *source_range=None*, *markers=None*, *effects=None*, *metadata=None*)

Bases: `opentimelineio.core.composition.Composition`

available_range()

Implemented by child classes, available range of media.

each_clip(*search_range=None*)

handles_of_child(*child*)

If media beyond the ends of this child are visible due to adjacent Transitions (only applicable in a Track) then this will return the head and tail offsets as a tuple of RationalTime objects. If no handles are present on either side, then None is returned instead of a RationalTime.

Example usage

```
>>> head, tail = track.handles_of_child(clip)
>>> if head:
...     print('do something')
>>> if tail:
...     print('do something else')
```

kind

Composition kind (Stack, Track)

neighbors_of(*item*, *insert_gap=0*)

Returns the neighbors of the item as a namedtuple, (previous, next).

Can optionally fill in gaps when transitions have no gaps next to them.

with insert_gap == NeighborGapPolicy.never: [A, B, C] :: neighbors_of(B) -> (A, C) [A, B, C] :: neighbors_of(A) -> (None, B) [A, B, C] :: neighbors_of(C) -> (B, None) [A] :: neighbors_of(A) -> (None, None)

with insert_gap == NeighborGapPolicy.around_transitions: (assuming A and C are transitions) [A, B, C] :: neighbors_of(B) -> (A, C) [A, B, C] :: neighbors_of(A) -> (Gap, B) [A, B, C] :: neighbors_of(C) -> (B, Gap) [A] :: neighbors_of(A) -> (Gap, Gap)

range_of_all_children()

Return a dict mapping children to their range in this track.

range_of_child_at_index(*index*)

Return the range of a child item in the time range of this composition.

For example, with a track: [ClipA][ClipB][ClipC]

The self.range_of_child_at_index(2) will return: TimeRange(ClipA.duration + ClipB.duration, ClipC.duration)

To be implemented by subclass of Composition.

trimmed_range_of_child_at_index(*index*, *reference_space=None*)

Return the trimmed range of the child item at index in the time range of this composition.

For example, with a track:

[ ]

[ClipA][ClipB][ClipC]

The range of index 2 (ClipC) will be just like range_of_child_at_index() but trimmed based on this Composition's source_range.

To be implemented by child.

class opentimelineio.schema.track.**TrackKind**

Bases: `object`

Audio = 'Audio'

Video = 'Video'

opentimelineio.schema.transition module

Transition base class

class opentimelineio.schema.transition.**Transition**(*name=None*, *transition_type=None*, *in_offset=None*, *out_offset=None*, *metadata=None*)

Bases: *opentimelineio.core.composable.Composable*

Represents a transition between two items.

duration()

in_offset

Amount of the previous clip this transition overlaps, exclusive.

out_offset

Amount of the next clip this transition overlaps, exclusive.

static overlapping()

Return whether an Item is overlapping. By default False.

range_in_parent()

Find and return the range of this item in the parent.

transition_type

Kind of transition, as defined by the schema.transition.TransitionTypes enum.

trimmed_range_in_parent()

Find and return the timmed range of this item in the parent.

class opentimelineio.schema.transition.**TransitionTypes**

Bases: `object`

Enum encoding types of transitions.

This is for representing "Dissolves" and "Wipes" defined by the multi-source effect as defined by SMPTE 258M-2004 7.6.3.2

Other effects are handled by the *schema.Effect* class.

Custom = 'Custom_Transition'

SMPTE_Dissolve = 'SMPTE_Dissolve'

opentimelineio.schemadef package

7.1.2 Submodules

7.1.3 opentimelineio.exceptions module

Exception classes for OpenTimelineIO

exception opentimelineio.exceptions.**AdapterDoesntSupportFunctionError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**CannotComputeAvailableRangeError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**CannotTrimTransitionsError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**CouldNotReadFileError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**InstancingNotAllowedError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**InvalidSerializableLabelError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**MisconfiguredPluginError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**NoDefaultMediaLinkerError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**NoKnownAdapterForExtensionError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**NotAChildError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**NotSupportedError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**OTIOError**
Bases: Exception

exception opentimelineio.exceptions.**ReadingNotSupportedError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**TransitionFollowingATransitionError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**UnsupportedSchemaError**
Bases: *opentimelineio.exceptions.OTIOError*

exception opentimelineio.exceptions.**WritingNotSupportedError**
Bases: *opentimelineio.exceptions.OTIOError*

7.1.4 opentimelineio.hooks module

HookScripts are plugins that run at defined points ("Hooks").

They expose a hook_function with signature: hook_function :: otio.schema.Timeline, Dict -> otio.schema.Timeline

Both hook scripts and the hooks they attach to are defined in the plugin manifest.

You can attach multiple hook scripts to a hook. They will be executed in list order, first to last.

They are defined by the manifests HookScripts and hooks areas.

```
>>>
{
    "OTIO_SCHEMA" : "PluginManifest.1",
    "hook_scripts" : [
        {
            "OTIO_SCHEMA" : "HookScript.1",
            "name" : "example hook",
            "execution_scope" : "in process",
            "filepath" : "example.py"
        }
    ],
    "hooks" : {
        "pre_adapter_write" : ["example hook"],
        "post_adapter_read" : []
    }
}
```

The 'hook_scripts' area loads the python modules with the 'hook_function's to call in them. The 'hooks' area defines the hooks (and any associated scripts). You can further query and modify these from python.

```
>>> import opentimelineio as otio
... hook_list = otio.hooks.scripts_attached_to("some_hook") # -> ['a','b','c']
...
... # to run the hook scripts:
... otio.hooks.run("some_hook", some_timeline, optional_argument_dict)
```

This will pass (some_timeline, optional_argument_dict) to 'a', which will a new timeline that will get passed into 'b' with optional_argument_dict, etc.

To Edit the order, change the order in the list:

```
>>> hook_list[0], hook_list[2] = hook_list[2], hook_list[0]
... print hook_list # ['c','b','a']
```

Now c will run, then b, then a.

To delete a function the list:

```
>>> del hook_list[1]
```

class opentimelineio.hooks.**HookScript**(*name=None*, *execution_scope=None*, *filepath=None*)
Bases: *opentimelineio.plugins.python_plugin.PythonPlugin*

run(*in_timeline*, *argument_map={}*)
Run the hook_function associated with this plugin.

opentimelineio.hooks.**available_hookscript_names**()
Return the names of HookScripts that have been registered.

`opentimelineio.hooks.`**`available_hookscripts`**`()`

Return the HookScripts objects that have been registered.

`opentimelineio.hooks.`**`names`**`()`

Return a list of all the registered hooks.

`opentimelineio.hooks.`**`run`**`(`*hook*, *tl*, *extra_args=None*`)`

Run all the scripts associated with hook, passing in tl and extra_args.

Will return the return value of the last hook script.

If no hookscripts are defined, returns tl.

`opentimelineio.hooks.`**`scripts_attached_to`**`(`*hook*`)`

Return an editable list of all the hook scriptss that are attached to the specified hook, in execution order. Changing this list will change the order that scripts run in, and deleting a script will remove it from executing

7.1.5 opentimelineio.media_linker module

MediaLinker plugins fire after an adapter has read a file in oder to produce MediaReferences that point at valid, site specific media.

They expose a "link_media_reference" function with the signature: link_media_reference :: otio.schema.Clip -> otio.core.MediaReference

or:

def linked_media_reference(from_clip): result = otio.core.MediaReference() # whichever subclass # do stuff return result

To get context information, they can inspect the metadata on the clip and on the media reference. The .parent() method can be used to find the containing track if metadata is stored there.

Please raise an instance (or child instance) of otio.exceptions.CannotLinkMediaError() if there is a problem linking the media.

For example:

for clip in timeline.each_clip():

try: new_mr = otio.media_linker.linked_media_reference(clip) clip.media_reference = new_mr

except otio.exceptions.CannotLinkMediaError: # or report the error pass

`class` `opentimelineio.media_linker.`**`MediaLinker`**`(`*name=None*, *execution_scope=None*, *filepath=None*`)`

Bases: *`opentimelineio.plugins.python_plugin.PythonPlugin`*

`link_media_reference``(`*in_clip*, *media_linker_argument_map=None*`)`

`class` `opentimelineio.media_linker.`**`MediaLinkingPolicy`**

Bases: `object`

`DoNotLinkMedia = '__do_not_link_media'`

`ForceDefaultLinker = '__default'`

`opentimelineio.media_linker.`**`available_media_linker_names`**`()`

Return a string list of the available media linker plugins.

`opentimelineio.media_linker.`**`default_media_linker`**`()`

`opentimelineio.media_linker.`**`from_name`**`(`*name*`)`

Fetch the media linker object by the name of the adapter directly.

`opentimelineio.media_linker.`**`linked_media_reference`**(*target_clip*, *media_linker_name='__default'*, *media_linker_argument_map=None*)

7.1.6 opentimelineio.opentime module

Library for expressing and transforming time.

NOTE: This module is written specifically with a future port to C in mind. When ported to C, Time will be a struct and these functions should be very simple.

`class` `opentimelineio.opentime.`**`BoundStrategy`**

Bases: `object`

Different bounding strategies for TimeRange

`Clamp = 2`

`Free = 1`

`class` `opentimelineio.opentime.`**`RationalTime`**(*value=0*, *rate=1*)

Bases: `object`

Represents an instantaneous point in time, value * (1/rate) seconds from time 0seconds.

`almost_equal`(*other*, *delta=0.0*)

`rescaled_to`(*new_rate*)

Returns the time for this time converted to new_rate

`value_rescaled_to`(*new_rate*)

Returns the time value for self converted to new_rate

`class` `opentimelineio.opentime.`**`TimeRange`**(*start_time=otio.opentime.RationalTime(value=0, rate=1)*, *duration=otio.opentime.RationalTime(value=0, rate=1)*)

Bases: `object`

Contains a range of time, starting (and including) start_time and lasting duration.value * (1/duration.rate) seconds.

A 0 duration TimeRange is the same as a RationalTime, and contains only the start_time of the TimeRange.

`clamped`(*other*, *start_bound=1*, *end_bound=1*)

Apply the range to either a RationalTime or a TimeRange. If applied to a TimeRange, the resulting TimeRange will have the same boundary policy as other. (in other words, _not_ the same as self).

`contains`(*other*)

Return true if self completely contains other.

(RationalTime or TimeRange)

`duration`

`end_time_exclusive`()

“Time of the first sample outside the time range.

If Start Frame is 10 and duration is 5, then end_time_exclusive is 15, even though the last time with data in this range is 14.

If Start Frame is 10 and duration is 5.5, then end_time_exclusive is 15.5, even though the last time with data in this range is 15.

end_time_inclusive()

The time of the last sample that contains data in the TimeRange.

If the TimeRange goes from (0, 24) w/ duration (10, 24), this will be (9, 24)

If the TimeRange goes from (0, 24) w/ duration (10.5, 24): (10, 24)

In other words, the last frame with data (however fractional).

extended_by(*other*)

Construct a new TimeRange that is this one extended by another.

overlaps(*other*)

Return true if self overlaps any part of other.

(RationalTime or TimeRange)

class opentimelineio.opentime.**TimeTransform**(*offset=otio.opentime.RationalTime(value=0, rate=1)*, *scale=1.0*, *rate=None*)

Bases: `object`

1D Transform for RationalTime. Has offset and scale.

applied_to(*other*)

opentimelineio.opentime.**duration_from_start_end_time**(*start_time*, *end_time_exclusive*)

Compute duration of samples from first to last. This is not the same as distance. For example, the duration of a clip from frame 10 to frame 15 is 6 frames. Result in the rate of start_time.

opentimelineio.opentime.**from_footage**(*footage*)

opentimelineio.opentime.**from_frames**(*frame*, *fps*)

Turn a frame number and fps into a time object. :param frame: (`int`) Frame number. :param fps: (`float`) Frame-rate for the (*RationalTime*) instance.

Returns (*RationalTime*) Instance for the frame and fps provided.

opentimelineio.opentime.**from_seconds**(*seconds*)

Convert a number of seconds into RationalTime

opentimelineio.opentime.**from_time_string**(*time_str*, *rate*)

Convert a time with microseconds string into a RationalTime.

Parameters

- **time_str** – (`str`) A HH:MM:ss.ms time.
- **rate** – (`float`) The frame-rate to calculate timecode in terms of.

Returns (*RationalTime*) Instance for the timecode provided.

opentimelineio.opentime.**from_timecode**(*timecode_str*, *rate*)

Convert a timecode string into a RationalTime.

Parameters

- **timecode_str** – (`str`) A colon-delimited timecode.
- **rate** – (`float`) The frame-rate to calculate timecode in terms of.

Returns (*RationalTime*) Instance for the timecode provided.

opentimelineio.opentime.**range_from_start_end_time**(*start_time*, *end_time_exclusive*)

Create a TimeRange from start and end RationalTimes.

opentimelineio.opentime.**to_footage**(*time_obj*)

opentimelineio.opentime.**to_frames**(*time_obj*, *fps=None*)
    Turn a RationalTime into a frame number.

opentimelineio.opentime.**to_seconds**(*time_obj*)
    Convert a RationalTime into float seconds

opentimelineio.opentime.**to_time_string**(*time_obj*)
    Convert this timecode to time with microsecond, as formated in FFMPEG

        Returns Number formated string of time

opentimelineio.opentime.**to_timecode**(*time_obj*, *rate=None*)
    Convert a RationalTime into a timecode string.

        Parameters

        - **time_obj** – (*RationalTime*) instance to express as timecode.
        - **rate** – (float) The frame-rate to calculate timecode in terms of. (Default time_obj.rate)

        Returns (str) The timecode.

opentimelineio.opentime.**validate_timecode_rate**(*rate*)
    Check if rate is of valid type and value. Raises (TypeError for wrong type of rate. Raises (VaueError) for invalid rate value.

        Parameters **rate** – (int) or (float) The frame rate in question

7.1.7 opentimelineio.test_utils module

Utility assertions for OTIO Unit tests.

class opentimelineio.test_utils.**OTIOAssertions**
    Bases: object

    assertIsOTIOEquivalentTo(*known*, *test_result*)
        Test using the 'is equivalent to' method on SerializableObject

    assertJsonEqual(*known*, *test_result*)
        Convert to json and compare that (more readable).

CHAPTER 8

Forward Look: Proposed C++-based Implementation

8.1 Writing OTIO in C, C++ or Python (June 2018)

Here are some initial thoughts about the subject, from around June 2018, about providing languages other than Python. The actual current plan is here.

The current python implementation of OTIO has been super helpful for defining the library and getting studio needs settled, but in order to integrate the library into vendor tools, a C/C++ implementation is required. We don't want to give up the Python API, however, so the plan is to port the library to C/C++ with a Python wrapper that implements an interface to the library as it currently stands; existing Python code shouldn't notice the switch. We can use the existing unit tests to vet the implementation and make sure that it matches the Python API.

There are several options for how to wrap C/C++ in Python, the intent of this document is to discuss the options we see and their pros/cons.

8.1.1 Python C-API

link: Python C-API

Pros:

- No extra dependencies

Cons:

- Extremely boilerplate heavy
- Have to manually build every part of the binding
- For users of boost, the bindings won't be directly compatible with boost bindings.
- Error prone: less type-safe and the reference counting must be manually done

8.1.2 Boost-Python

link: Boost-python

Pros:

- High level binding
- Established, familiarity around the industry, reasonably popular

Cons:

- Heavy dependency to add to projects if they aren't already using boost

8.1.3 PyBind11

link: PyBind11 Github

Pros:

- High level binding
- Takes advantage of C++11/17 features to make wrapping even more terse (if they're available)
- Can be embedded in the project without requiring Boost

Cons:

- For users of boost, the bindings won't be directly compatible with boost bindings.
- Newer and less established than other options.

8.1.4 Conclusion

After talking with several vendors, studios, and participants, we have concluded that we will make this:

- C++ Implementation of OTIO (following VFX Platform CY2017 standard C++11)
- Pybind11 Bindings
- To support other languages will make a `extern "C"` wrapper around the C++ API
- Support for Swift (with some bridging provided by NSObject derived classes written in Objective-C++)

This will replace the current pure-Python implementation, attempting to match the current Python API as much as possible, so that existing Python programs that use OTIO should not need to be modified to make the switch.

We will try to make this a smooth transition, by starting with `opentime` and working out to the rest of the API.

Also, in the future, we will likely provide Boost Python bindings to the C++ API for applications that already use Boost Python.

8.2 C++ Proposal Details

8.2.1 Caveats

Both the code examples and the sample header files are written without regard to either namespacing or how the *#include* structure will be implemented. For clarity, inline code definitions are omitted from header files though in many cases they will be present in the final product.

8.2.2 TL;DR

- Opentime C++ simplified header files
- OTIO C++ simplified hear files.

8.2.3 Dependencies

The C++ OpentimelineIO (OTIO) library implementation will have the following dependencies:

- rapidjson
- any (C++ class)
- optional (C++ class)
- The C++ Opentime library (see below)

The need for an "optional" (i.e. a container class that holds either no value or some specific value, for a given type T) is currently small, but does occur in one key place (schemas which need to hold either a `TimeRange` or indicate their time range is undefined).

In contrast, the need for an "any" (a C++ type-erased container) is pervasive, as it is the primary mechanism that serialization and deserialization rest upon. It is also the bridge to scripting systems like Python that are not strongly typed. The C++17 standard defines the types `std::optional` and `std::any` and these types are available in `std::experimental` in some other cases, and our implementation targets those. However, since many (probably most) sites are not yet compiling with C++17, our implementation makes available public domain C++11 compliant versions of these types:

- any (C++11 compliant)
- optional (C++11 compliant)

Support for Python will require pybind11.

The C++ Opentime library (i.e. `RationalTime`, `TimeRange` and `TimeTransform`) will have no outside dependencies. In fact, given the current Python specification, a C++ Opentime API (should) be fairly straightforward and uncontroversial. Reminder: these sample header files exist only to show the API; namespacing and other niceties are ommitted.

8.2.4 Starting Examples

Defining a Schema

Before jumping into specifics, let's provide some simple examples of what we anticipate code for defining and using schemas will look like. Consider the `Marker` schema, which adds a `TimeRange` and a color to a schema which already defines properties `name` and `metadata`:

```
class Marker : public SerializableObjectWithMetadata {
public:
    struct Schema {
        static auto constexpr name = "Marker";
        static int constexpr version = 1;
    };

    using Parent = SerializableObjectWithMetadata;

    Marker(std::string const& name = std::string(),
```

(continues on next page)

(continued from previous page)

```
            TimeRange const& marked_range = TimeRange(),
            std::string const& color = std::string("red"),
            AnyDictionary const& metadata = AnyDictionary());

    TimeRange marked_range() const;
    void set_time_range(TimeRange marked_range);

    std::string const& color() const;
    void set_color(std::string const& colir);

protected:
    virtual ~Marker;

    virtual bool read_from(Reader&);
    virtual void write_to(Writer&) const;

private:
    TimeRange _marked_range;
    std::color _color;
};
```

The contructor takes four properties, two of which (`marked_range` and `color`) are stored directly in `Marker`, with the remaining two (`name` and `metadata`) handled by the base class `SerializableObjectWithMetadata`.

For the OTIO API, we will write standard getters/setters to access properties; outside of OTIO, users could adopt this technique or provide other mechanisms (e.g. public access to member variables, if they like). The supplied Python binding code will allow users to define their own schemas in Python exactly as they do today, with no changes required.

The `Schema` structure exists so that this type can be added to the OTIO type registry with a simple call

```
TypeRegistry::instance()::register_type<Marker>();
```

The call to add a schema to the type registry would be done within the OTIO library itself for schemas known to OTIO; for schemas defined outside OTIO, the author of the schema would need to make the above call for their class early in a program's execution.

Reading/Writing Propeties

Code must also be written to read/write the new properties. This is simple as well:

```
bool Marker::read_from(Reader& reader) {
    return reader.read("color", &_color) &&
        reader.read("marked_range", &_marked_range) &&
        Parent::read_from(reader);
 }

void Marker::write_to(Writer& writer) const {
    Parent::write_to(writer);
    writer.write("color", _color);
    writer.write("marked_range", _marked_range);
}
```

Even when we define more complex properties, the reading/writing code is as simple as shown above, in almost all cases.

Note: Properties are written to the JSON file in the order they are written to from within `write_to()`. But the reading code need not be in the same order, and changes in the ordering of either the reading or writing code will not break compatability with previously written JSON files.

However, it is vital to invoke `Parent::read_from()` *after* reading all of the derived class properties, while for writing `Parent::write_to()` must be invoked *before* writing the derived class properties.

Note: Also note that the order of properties within a JSON file for data that is essentially an `std::map<>` (see `AnyDictionary` below) is always alphebetical by key. This ensures deterministic JSON file writing which is important for comparison and testing.

8.2.5 Using Schemas

Creating and manipulating schema objects is also simple:

```
Track* track = new Track();
Clip* clip1 = new Clip("clip1", new ExternalReference("/path/someFile.mov"));
Clip* clip2 = new Clip("clip2");

track->append_child(clip1);
track->append_child(clip2);

...

for (Item* item: track->children()) {
    for (Effect* effect: item->effects()) {
         std::cout << effect->effect_name();
         ...
    }
}
```

8.2.6 Serializable Data

Data in OTIO schemas must be read and written as JSON. Data must also be available to C++, in some cases as strongly typed data, while in other cases as untyped data (i.e. presented as an `any`).

For discussion purposes, let us consider that all data that is read and written to JSON is transported as a C++ `any`. What can that `any` hold?

First, the `any` can be empty, which corresponds with a `null` JSON value. The `any` could also hold any of the following "atomic" types: `bool`, `int`, `double`, `std::string`, `RationalTime`, `TimeRange` and `TimeTransform`. All but the last three are immediately expressable in JSON, while the three Opentime types are read/written as compound structures with the same format that the current Python implementation delivers. The final "atomic" type that an `any` can hold is a `SerializableObject*`, which represents the C++ base class for all schemas. (Note: it will not be valid for an `any` to hold a pointer to a derived class, for example, a `Clip*` value. The actual C++ static type in the `any` will always be a pointer to the base class `SerializableObject`.)

Next, for any of the above atomic types `T`, excepting for `SerializableObject*`, an `any` can store a type of `optional<T>`.

Finally, the `any` can hold two more types: an `AnyDictionary` and an `AnyVector`. For this discussion, consider an `AnyDictionary` to be the type `std::map<std::string, any>` and the type `AnyVector` to be the type `std::vector<any>`. The actual implementation is subtly different, but not to end-users: the API for both these types exactly mirrors the APIs of `std::vector<any>` and `std::map<std::string, any>` respectively. The `AnyVector` and `AnyDictionary` types are of course the JSON array and object types.

8.2.7 C++ Properties

In most cases, we expect C++ schemas to hold data as strongly-typed properties. The notable exception is that low in the inheritance hierarchy, a C++ property named `metadata` which is of type `AnyDictionary` is made available, which allows clients to story data of any type they want. Manipulating such data will be as simple as always, from an untyped language like Python, while in C++/Swift, the typical and necessary querying and casting would need to be written.

As we saw above, declaring and and handling read/writing for "atomic" property types e.g. `std::string`, `TimeRange`) is straightforward and requires little effort. Additionally, reading/writing support is automatically provided for the (recursively defined) types `std::vector<P>`, `std::list<P>` and `std::map<std::string, P>` where `P` is itself a serializable property type. Accordingly, one is free to declare a property of type `std::vector<std::map<std::string, std::list<TimeRange>>>` and it will serialize and deserialize properly. However, such a type might be hard to reflect/bind in a Python or Swift bridge. Our current implementation however bridges one-level deep types such as `std::vector<RationalTime>` or `std::map<std::string, double>` to Python (and later Swift) quite easily and idiomatically.

Finally, one can declare lists and dictionaries for schema objects, in as strongly typed fashion as required. That is, a property might be a list of schema objects of any type, or the property might specify a particular derived class the schema object must satisfy. Again, this is taken care of automatically:

```
class DerivedSchema : public SerializableObject {
    ...
private:
    std::vector<MediaReference*> _extra_references;    // (don't actually do this)
};
```

In this case, the derived schema could choose to store extra media references. The reading/writing code would simply call

```
reader.read("extra_references", &_extra_references)
```

and

```
writer.write("extra_references", _extra_references)
```

to read/write the property.

Note: The comment "don't actually do this" will be explained in the next section; the actual type of this property needs to be slightly different. The code for reading/writing the property however is correct.

8.2.8 Object Graphs and Serialization

The current Python implementation assumes that no schema object is referenced more than once, when it comes to serialization and deserialization. Specifically, the object "graph" is assumed to implicitly be a tree, although this is not always enforced. For example, the current Python implementation has this bug:

```
clip1 = otio.schema.Clip("clip1")
clip2 = otio.schema.Clip("clip2")
```

(continues on next page)

(continued from previous page)

```
ext_ref = otio.schema.ExternalReference("/path/someFile.mov")
clip1.media_reference = ext_ref
clip2.media_reference = ext_ref
```

As written, modifying `ext_ref` modifies the external media reference data for both `clip1` and `clip2`. However, if one serializes and then deserializes this data, the serialized data replicates the external references. Thus, upon reading back this object graph, the new clips no longer share the same media reference.

The C++ implementation for serialization will not have this limitation. That means that the object structure need no longer be a tree; it doesn't, strictly speaking, even need to be a DAG:

```
Clip* clip1 = new Clip();
Clip* clip2 = new Clip();

clip1->metadata()["other_clip"] = clip2;
clip2->metadata()["other_clip"] = clip1;
```

will work just fine: writing/reading or simply cloning `clip1` would yield a new `clip1` that pointed to a new `clip2` and vice versa.

Note: This really does work, except that it forms an unbreakable retain cycle in memory that is only broken by manually severing one of the links by removing, for example, the value under "other_clip" in one of the metadata dictionaries.

The above example shows what one could (but shouldn't do). More practical examples are that clips could now share media references, or that metadata could contain references to arbitrary schemas for convenience.

Most importantly, arbitrary serialization seperates lets us separate the concepts of "I am responsible for reading/writing you" from the "I am your (one and only) parent" from "I am responsible to deleting you when no longer needed." In the current Python implementation, these concepts are not explicitly defined, mostly because of the automatic nature of memory management in Python. In C++, we must be far more explicit though.

8.2.9 Memory Management

The final topic we must deal with is memory management. Languages like Python and Swift natually make use of reference counted class instances. We considered such a route in C++, by requiring that manipulations be done not in terms of `SerializableObject*` pointers, but rather using `std::shared_ptr<SerializableObject>` (and the corresponding `std::weak_ptr`). While some end users would find this a comfortable route, there are others who would not. Additionally (and this is a topic that is very deep, but that we are happy to discuss further) the `std::shared_ptr<>` route, coupled with the `pybind` binding system (or even with the older `boost` Python binding system) wouldn't provide an adequate end-user experience in Python. (And we would expect similar difficulties in Swift.)

Consider the following requirements from the perspective of an OTIO user in a Python framework. In Python, a user types

```
clip = otio.schema.Clip()
```

Behind the scenes, in C++, an actual `Clip` instance has been created. From the user's perspective, they "own" this clip, and if they immediate type

```
del clip
```

then they would expect the Python clip object to be destroyed (and the actual C++ `Clip` instance to be deleted). Anything less than this is a memory leak.

But what if before typing `del clip` the Python user puts that clip into a composition? Now neither the Python object corresponding to the clip *nor* the actual C++ `Clip` instance can be destroyed while the composition has that clip as a child.

The same situation applies if the end user does not create the objects directly from Python. Reading back a JSON file from Python creates all objects in C++ and hands back only the top-most object to Python. Yet that object (and any other objects subsequently exposed and held by Python) must remain undeletable from C++ while the Python interpreter has a reference to those objects.

It might seem like shared pointers would fix all this but in fact, they do not. The reason is that there are in reality two objects: the C++ instance, and the reflected object in Python. (While it might be feasible to "auto-create" the reflected Python object whenever it was needed, and really think of having one object, this choice makes it impossible to allow defining new schemas in Python. The same consequence applies to allowing for new schemas to be defined in Swift.) Ensuring a system that does not leak memory, and that also keeps both objects alive as long as either side (C++ or the bridged language) is, simply put, challenging.

With all that as a preamble, here is our proposed solution for C++.

- A new instance of a schema object is created by a call to `new`.
- All schema objects have protected destructors. Give a raw pointer to a schema object, client code may not directly invoke the `delete` operator, but may write

  ```
  Clip* c = new Clip();
  ...
  c->possibly_delete();    // returns true if c was deleted
  ```

- The OTIO C++ API uses raw pointers exclusively in all its function signatures (e.g. property access functions, property modifier functions, constructors, etc.).
- Schema objects prevent premature destruction of schema instances they are interested in by storing them in variables of type `SerializableObject::Retainer<T>` where `T` is of type `SerializableObject` (or derived from it).

For example:

```
 class ExtendedEffect : public Effect {
 public:
    ...
    MediaReference* best() const {
        return _best;
    }

    void set_best(MediaReference* best) {
        _best = best;
    }

    MediaReference* best_or_other() {
        return _best ? _best : some_other_reference();
    }

private:
  Retainer<MediaReference> _best;
};
```

In this example, the `ExtendedEffect` schema has a property named `best` that must be a `MediaReference`. To indicate that it needs to retain its instance, the schema stores the property not as a raw pointer, but using the `Retainer` structure.

Nothing special needs to be done for the reading/writing code, and there is automatic two-way conversion between `Retainer<MediaReference>` and `MediaReference*` which keeps the code simple. Even testing if the property is set (as `best_or_other()` does) is done as if we were using raw pointers.

The only rules that a developer needs to know is:

- Creating a new schema instance starts the instance with an internal count of 0.
- Putting a schema instance into a `Retainer` object increases the count by 1.
- Destroying the retainer object or reassigning a new instance to it decreases the count by 1 of the object if any in the retainer object. If this causes the count to reach zero, the schema instance is destroyed.
- The possibly_delete() member function of `SerializableObject*` checks that the count of the instance is zero, and if so deletes the object in question.
- An `any` instance holding a `SerializableObject*` actually holds a `Retainer<SerializableObject>`. That is, blind data safely retains schema instances.

In practice, these rules mean that only the "root" of the object graph needs to be held by a user in C++ to prevent destruction of the entire graph, and that calling `possibly_delete()` on the root of the graph will cause deletion of the entire structure (assuming no cyclic references) and/or assuming the root isn't currently sitting in the Python interpreter.

We have extensively tested this scheme with Python and written code for all the defined schema instances that exist so far. The code has proven to be lightweight and simple to read and write, with few surprises encountered. The Python experience has been unchanged from the original implementation.

Examples

This code shows when instances are actually deleted:

```
Track* t = new Track;

Clip* c1 = new Clip;
c1->possibly_delete();    // c1 is deleted

Clip* c2 = new Clip;
t->add_child(c2);
c2->possibly_delete();   // no effect
t->possibly_delete();   // deletes t and c2
```

The only time that using a `Retainer` object might be necessary in client code is as follows:

```
Track* t = get_some_track();
int index = find_child_named(t, "clip1");  // assume this returned a valid index

Clip* c = t->children()[index];
t->remove_child(index);   // deletes c if t holds the last reference to c
std::cout << c->name();   // <possible crash>
```

In this example, if the user tries to use `c` after the call to `remove_child()`, they will crash if the track held the last reference to `cc`. The last three lines should instead be written as:

```
SerializableObject::Retainer<Clip> rc = t->children()[index];
t->remove_child(index);   // child is NOT deleted because of the retainer

```

(continues on next page)

(continued from previous page)

```
Clip* c = rc.value;
std::cout << c->name();
```

In actual practice, we believe that the need to write code of the above form should occur extremely rarely.

8.2.10 Proposed OTIO C++ Header Files

Proposed stripped down OTIO C++ header files.

8.3 Proposed Language Bridges

8.3.1 Python

Since OTIO originated as Python (and has an extensive test suite, in Python), our starting position is that existing Python code (adapters, plugins, schemadefs) must continue to work, as currently written. Python code in the `core` or `schema` directories will of course be rewritten, but Python code outside those modules should not be aware of any change.

The bridge from C++ to Python (and back) will be `pybind11`. Given that existing code needs to work, clearly, the bridge will be implemented so as to make the reflection of the C++ datastructures, back to Python, utterly "Pythonic." (It has to be, since we don't want to break existing code.)

Given the above, there's no point giving Python illustrations of how things will work going forward: they'll work as they already have.

8.3.2 Swift

The intention is to expose OTIO in Swift with the same care we take with Python: we want everything to feel utterly Swift-like. Because Swift can gain automatic API access to non-member functions written in Objective-C++, and Objective-C++ can directly use the proposed OTIO C++ API, we believe that a bridge to swift will not require writing an explicit explicit `extern "C"` wrapper around OTIO C++. We believe that like Python, Swift should be capable of defining new schemas, and that access to existing and new schemas and their properties should be done in terms of Swift API's that conform Swift's sequence/collection protocols, just as Python interfaces do with respect to Python.

8.3.3 Bridging to C (and other languages)

Briding to C (and by extension other languages) would presumably be accomplished by writing an `extern "C"` wrapper around the OTIO C++ API. This is of relatively low priority, given that we will have three languages (C++ itself, Python, and Swift) that do not need this.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index