D5.6 Demo of Transcode Mechanism

Grant Agreement nr	780470
Project acronym	SAUCE
Project start date (duration)	January 1st 2018 (36 months)
Document due:	31/03/2020
Actual delivery date	31/03/2020
Leader	Foundry
Reply to	Peri Friend
Document status	Submission Version

Project funded by H2020 from the European Commission

Project ref. no.	780470
Project acronym	Smart Asset re-Use in Creative Environments
Project full title	SAUCE
Document name	D5.6 Demo of Transcode Mechanism
Security (distribution level)	Public
Contractual date of delivery	31/03/2020
Actual date of delivery	31/03/2020
Deliverable name	Demo of Transcode Mechanism
Type	DEM
Status & version	Submission Version
Number of pages	14
WP / Task responsible	Foundry
Other contributors	None
Author(s)	Sam Hudson, Tom Mulvaney, Dan Ring, Peri Friend
EC Project Officer	Ms Adelina Cornelia Dinu - adelina-cornelia.dinu@ec.europa.eu
Abstract	One of the key aims of SAUCE is to create multiple versions of the same asset that are appropriate to its (re)use scenario. To do this requires a transcoding mechanism. This task will develop a scheduling system to trigger the appropriate transcoding mechanism. It will also allow the different types of the same asset to be linked via their associated descriptors. The development will allow efficient generation of assets via multiple transcode applications for use globally and for re-purpose in different sectors (e.g. a low-polygon model for VR or games and a high-polygon model for film)
Keywords	Plugin interface, USD transcode
Sent to peer reviewer	Yes
Peer review completed	Yes
Circulated to partners	No
Read by partners	No
Mgt. Board approval	No

Document History

Version and date	Reason for Change
1.0 02-0-12020	Document created by Peri Friend
1.1 20-03-2020	Version for internal review
1.2 31-03-2020	Version for submission

Table of Contents

1 EXECUTIVE SUMMARY

This document describes the demonstration of our transcode mechanism which enables the transformation of Assets within the Flix system. It demonstrates the plugin mechanism itself; how to upload and execute a plugin, as well as an example of using a “USD import” plug-in to render a thumbnail of the 3D stage contained in the uploaded USD file. It concludes that consortium partners are now able to write their own transcode plugins to suit their asset pipeline requirements.

2 BACKGROUND

The Flix plugin system is a flexible system which can utilise some built in transformations provided ‘out of the box’ with Flix, or can be extended via our Plugin system to allow for custom transformations. This extensibility allows for the system to meet the needs of any Studio, or consortium partner to transform any asset desired, given they have the appropriate transformation plugins.

2.1 Relationship to other work packages

This work will be integrated into WP7, creating a fully replicable, asset agnostic Flix toolset in D7.2 and D7.3. Additionally, partners from all work packages are now enabled to write plugins for all asset types that can be supported in the Flix framework. Benchmarking testing will take place in WP8T3

2.2 Relationship to Self Assessment

Flix previously only supported image and movie formats, the new transcode system allows plugins to define transcode mechanisms enabling extension to include new smart assets types, as described in the self assessment plan.

The document below outlines USD as an example of a plugin which extends Flix to work with 3D assets, showing the ability to extend and reuse the asset store for emerging formats as they are developed. Enabling the reuse of all older assets on the system via transcoding.

3 INTRODUCTION

3.1 Main objectives and goals

For assets to be reusable within a VFX pipeline they need to be readily utilisable in different scenarios. To facilitate as many of these scenarios as possible, assets need to be easily transformable into new formats. This allows for future proofing assets to be used with newer software versions, or to change an asset to meet new demands. An example of this is to convert one 3D model format to another, eg. from OBJ to FBX. We also want to demonstrate the ability to generate thumbnail images from a variety of image and non-image input formats, including PSD, USD, and OBJ.

In order for us to achieve this aim, we have built a plug-in architecture which enables transcoding on an asset from one to another. This process is managed across multiple servers, allowing for a scalable and fault tolerant service. Most importantly, users can enable new formats or compatibility with older formats by writing simple plugins to execute the transformation upon request.

3.2 Methodology

The plugin system was created using a process called *dialectic driven design* (a branch of *domain driven design*), in which the development team discusses the problem, and how a solution might be implemented, over an extended period of time, regularly including relevant stakeholders (client, users etc). Dialectic proceeds from the assumption that at any given time, each person is probably wrong about something, but that a better solution emerges through the process of disagreement and persuasion.

One issue that emerged during dialect was the need to balance the need for change with the need for predictability. Users must be able to update plugins because their code will inevitably have bugs, moreover their requirements may change. But, if users can change arbitrary plugin attributes then the risk of human error increases; for example, were the name of a plugin to change then anyone not aware of that fact might think that the plugin had been deleted. In VFX and animation, where mistakes are costly, the potential for human error must be minimized. Therefore, the system must resolve the competing demands of the need for change and the need for predictability.

In the early stages, code churn is high because refactoring is a regular occurrence. Therefore the early testing strategy emphasizes blackbox API tests and acceptance tests, with lightweight unit testing. In this way early tests do not make assumptions that cause them to break later. As code churn decreases, the team increases unit test coverage, but acceptance tests remain an important part of the testing strategy (because the team should know, not only that they built the system right, but that they built the right system).

Domain Concepts

The challenge was to create a system that requires minimal domain knowledge to use, while providing enough power for predictable and flexible plugin versioning.

As discussed above, plugins must be open to change over time, but should also have some attributes that remain the same in order to minimize human error. Therefore, we created two models: *plugin* and *plugin version* (or *version)*, with the former containing all immutable attributes and one mutable attribute, and the latter containing only mutable attributes.

The plugin model is simpler because there are fewer immutable attributes. It has a *name*, a *type* and a set of *events* on which it executes. It also has exactly one mutable attribute: *status*, because this allows users to temporarily disable plugins.

The version model contains all other plugin information, including a description, whether the plugin can execute in parallel, and information needed to execute the plugin. Plugin versions also have a *status*, which varies independently of plugin status, and is set by Flix Server to disable invalid plugin versions.

The above is required for predictable and flexible change over time, but is too complex for end-users. Therefore, we abstracted these concepts away, presenting users with a unified "plugin". To achieve this, we created a new domain concept: *plugin pair* (or *pair*), which has both a plugin and a version. Pairs only exist in the frontend and the API that the frontend uses to communicate with the backend, they are a domain concept with no backend implementation.

To users, a pair is a "plugin". Users see data from both plugin and version as belonging to the same object, presented in the same place. They can only see the status of the pair, which is inferred from the plugin and version status where both must be enabled for the pair to be "On".

At any given time, a plugin has an *active version* which is the most recently created version that has not been marked deleted. It is possible for the active version to be disabled, in which case the plugin does not execute.

So far, we have described only the domain models required for plugin management operations, but we also need to track plugin execution so that users can see that the plugin has executed and track the outcome. Therefore, we created the concept *plugin execution* (or *execution)* which contains information related to runtime execution (*event trigger*, *user*, timestamps etc). Importantly, the *status* and *message* are needed for detecting bugs in the plugin source code.

Finally, we describe *plugin types* (mentioned as an attribute of plugin above). There are three types of plugin: *Fire & Forget* (FaF), *modifier* and *override*. FaF is the simplest use case, because we track execution status but do not expect any output or return value (hence "fire and forget"). Modifier plugins are used to modify data generated by Flix, data is expected to return in the same format but modified (e.g. enforce a naming convention for sequence titles). Override plugins override some part of the default Flix pipeline, for example, users are able to override the default process for creating AAF files, instead creating the AAF files using their own systems.

Implementation

Flix Client is implemented in Typescript using the Angular framework. So all user-facing code in the plugin system is composed of these technologies.

Flix Server is implemented in Go, but has pre-existing capabilities to run Python. The plugin management system is written in Go, while plugins are executed in Python. Python is an appropriate language choice for plugins, due to its widespread use as a scripting language, particularly among VFX and animation pipelines.

3.3 Terminology

Term	Description
Plugin	A plugin is an arbitrary bundle of Python code which can be written by any third party which can be executed by the Flix Server in order to perform custom tasks within the Flix pipeline.
Transformation	Transformation is the `conversion` or `transcoding` of an asset from one form to another. This could mean converting an FBX to OBJ, or generating a graphical representation of a non-graphical format, eg. USD to PNG.

4 PLUGIN INTERFACE

The plugin interface is designed to allow users to execute code that augments, modifies or overrides the default Flix pipeline.

There are three types of plugin:

1. *Fire & Forget* - When execution is complete, no further action is required by Flix.
2. *Modifier* - Modifies data generated by Flix.
3. *Override* - Overrides part of the Flix pipeline.

Below we demonstrate how to upload and execute a plugin.

4.1 Upload plugin

Using the Flix client, upload a plugin through the Management Console.

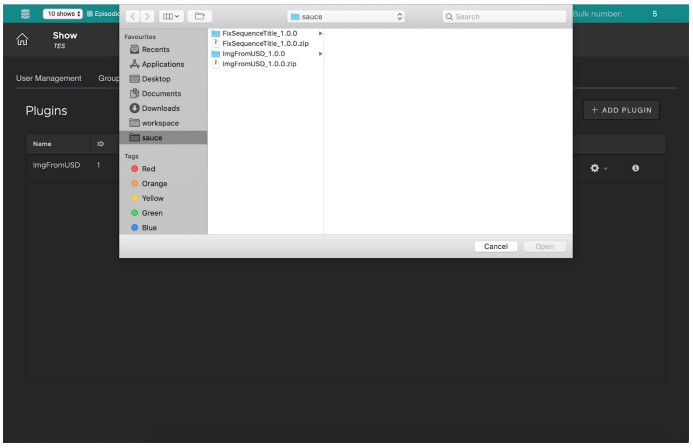

Fig. 1: Adding a new plugin in the management console.

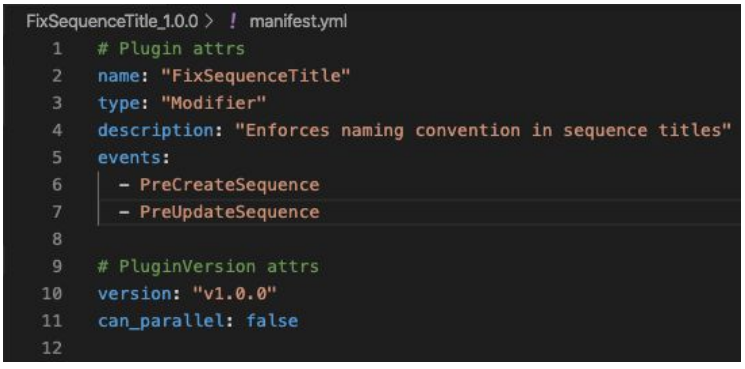

Fig. 2: Plugin manifest describes plugin. This plugin enforces naming convention for sequence titles, and it fires before a sequence is created or updated.

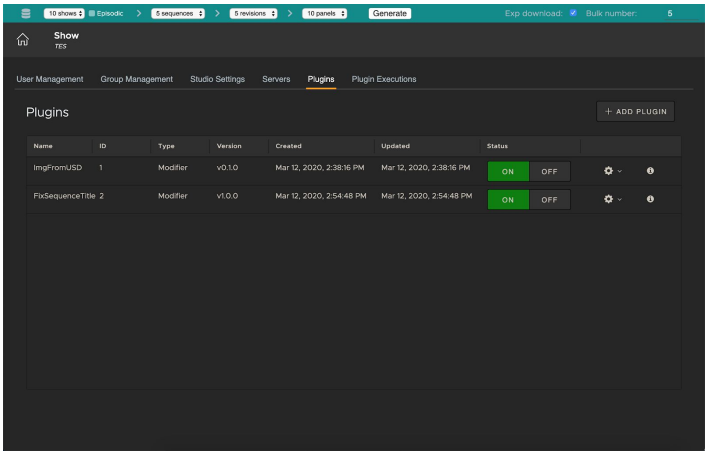

Fig. 3: The example plugin (called "FixSequenceTitle") is switched on, and will therefore execute when a relevant event fires (PreCreateSequence or PreUpdateSequence)

4.2 Execute plugin

Plugins are executed when the relevant event is fired. In this example, the plugin fires before a new sequence is created or an existing sequence is updated.

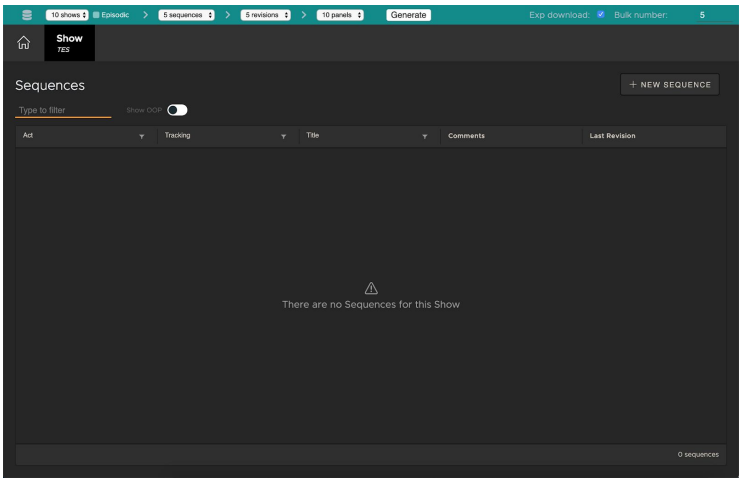

Fig. 4: There are no existing sequences for this show

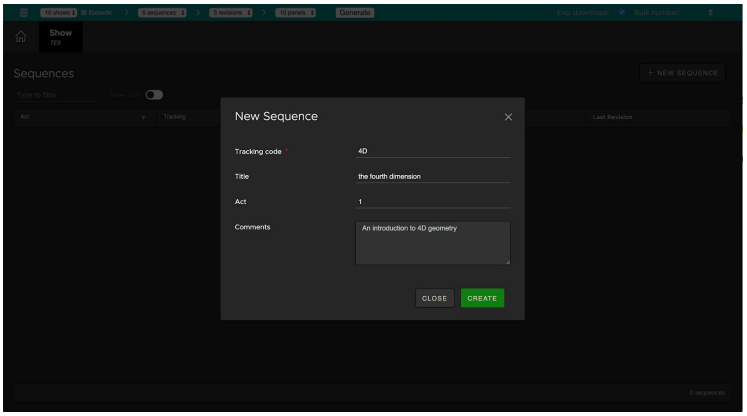

Fig. 5: Adding a sequence with an uncapitalized title (convention violation)

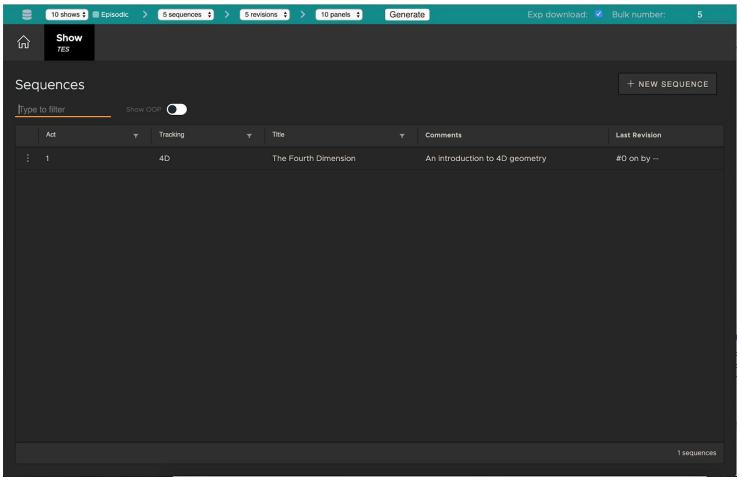

Fig. 6: Sequence has been added. We can see that the plugin has modified the title so that it conforms to convention

4.3 Plugin Management

Users can perform the following management operations:

Update - Update plugin to new version

Rollback - Rollback to the previous version

Delete - Delete the plugin and all associated versions

Download - Download the latest version from Flix server

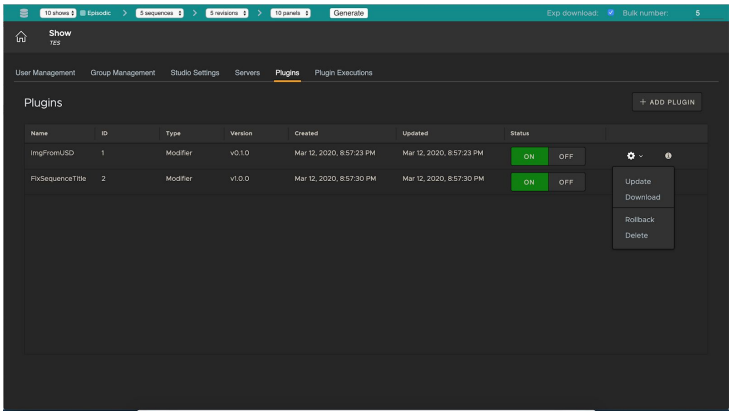

Fig. 7: Plugin management interface

5 USD TRANSCODE INTO IMAGE

USD is an open-source library from Pixar which aims to provide a unified way of capturing 3D scene data, such as geometry, meshes, cameras, lights, materials, animations, skeleton rigs, volumes and more. It's primary use at the moment is to provide a clear, well-defined format for passing 3D scene data through various stages of the visual effects and animation pipelines. Importantly the same USD data can be passed through the entire production, from storyboard & pre-viz, through post-production and delivery, having the relevant assets and complexity layered on as it moves through the production pipeline. Although relatively nascent, USD is being quickly adopted by visual effects and animation studios around the world.

One challenge with USD is its complexity. In addition to the non-trivial level of pipeline engineering to ensure that the USD libraries are built correctly, there is usually a substantial level of organizational complexity built-in, for example, capturing the show name, the sequence, the version of sequence, the version of each asset, purpose or attributes on the asset (such as high or low poly targets) and more. Essentially the organization of the assets which would normally be handled by the file system (i.e. different folders for the above would be created) is now also being handled by the USD file itself. This is great pipeline efficiency, as each USD can in theory define the entire pipeline. However for an artist or someone looking to access or pull data from the USD file, they will likely need the data transcoded into a more appropriate format for a given task.

As mentioned previously, USD can be used at any stage of the production pipeline, often as early as pre-viz, which can lead into on-set virtual production pipelines. The challenge is then, given an asset library of USD files, or even just a single USD file for your entire production, how do you search, access & visualise its contents in a simple way? The approach we're taking here as a first step is to transcode the USD file into a more immediately accessible format: an image.

Let's look at how this works in Flix in practice. Flix lives at the beginning of the pipeline, and already has a view onto the production pipeline through it's Show, Sequence & Revision workflow. This means from a USD point of view, we have an access point to begin examining the pipeline. In its simplest use case, imagine a 3D artist mocking up a previz storyboard in a 3D layout and modelling tool such as Maya. The important things to block out at this stage are the various beats of the story, defined by character poses & expressions, camera positions and properties, as well as any location setting. This could be achieved by setting a handful of keyframes for the above scene, one keyframe per panel. The scene is then saved out to the pipeline compliant USD file. When it's time to review, the artist can import the USD file into Flix. The plug-in system then invokes the USD import plug-in, where for each keyframe the panel is rendered. The render happens on the server using Hydra, which is an open specification to allow arbitrary renderers to take a USD stage and deliver pixels. Once rendered, it's easy for the storyboard reviewer to discuss changes to timings and the story in the traditional way. Updates can then be made in 3D layout, and with the rendering taken care of, the round-tripping between update and review is drastically reduced.

The following is an end-to-end example of the the USD Import plug-in to create thumbnails for uploaded USD files.

5.1 USD File Example - 'usdview'

Included in the USD library are a set of tools for visualising USD files, as well as investigating the various Hydra renderers installed on the host system. These tools are written in Python which makes it both pipeline friendly, and easy to integrate into our plug-in system.

The image below is a screenshot of the `Kitchen & Robot' USD file used for testing (Fig. 1). It was assembled in our `Heist' USD-based layout tool, where a number of keyframes for the principal camera were set. The file is saved as a '.usdz' file, which is a USD-compatible 'zip' bundle of all of the scene assets, making it highly portable.

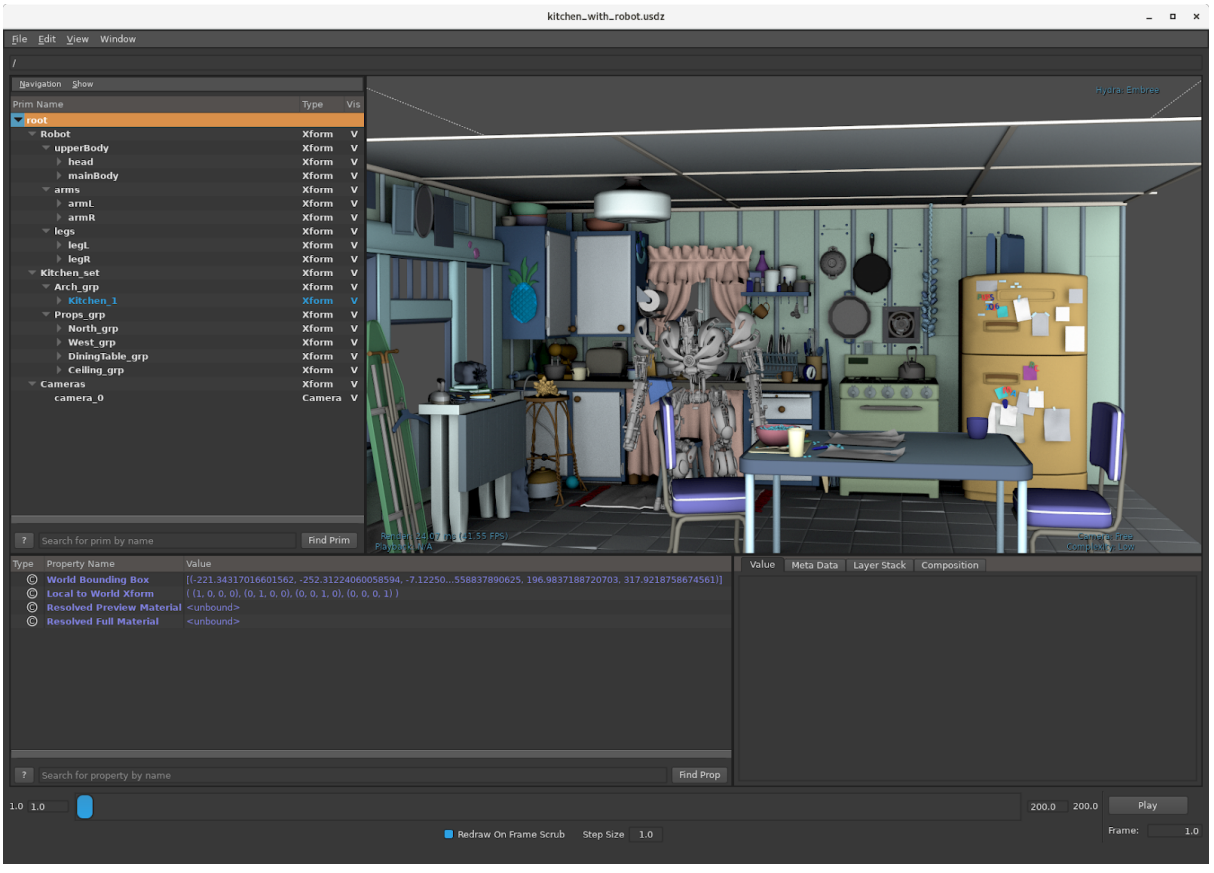

Fig. 8: USD's 'usdview' tool showing the 'Kitchen & Robot' test stage.

5.2 USD Import Plug-in Example

The following is an end-to-end workflow demonstrating how the USD Import plug-in is used. The first step is to add the 'Create Thumbnail' plug-in, by following the steps above. Once installed you can see it's been added to the list of active plug-ins (Fig. 2).

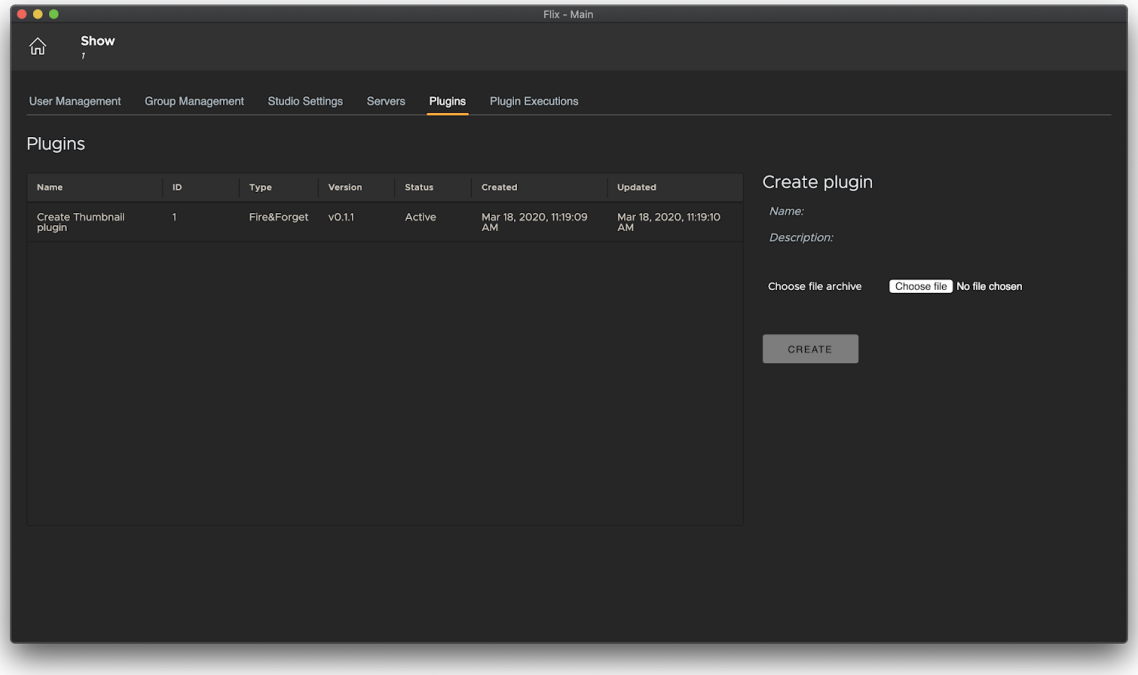

Fig. 9: The 'Create Thumbnail' plug-in is installed

Next, we navigate through the relevant 'Show', `Sequence' and to the board `Revision' we want to build out. Then we import the USD file by dragging it onto the Flix client. Identically to how an image file would be imported, this tags the USD file as an asset for this show (Fig. 3).

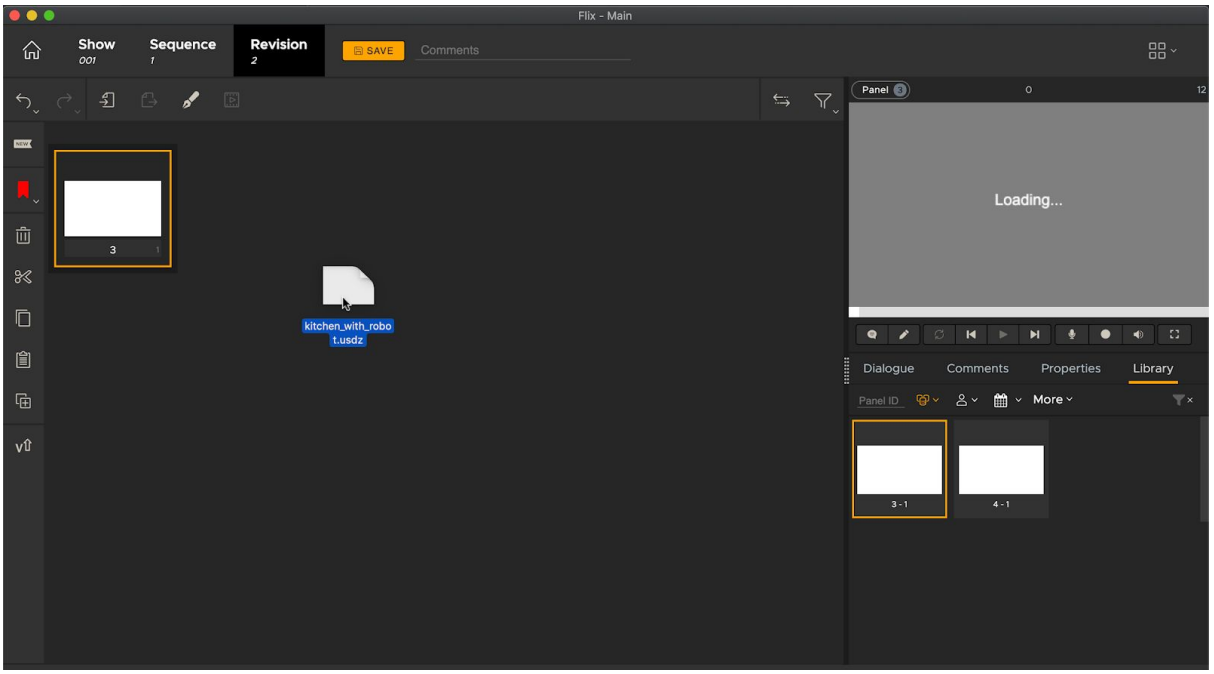

Fig. 10: Importing the USD file by dragging it onto the Flix client.

In the background on the server, the `Create Thumbnail' plug-in works to take the USD file and render it using a Hydra delegate. In this case, we've selected Intel's Embree renderer, however it's possible to switch this out for other commercial renderers that release their products as Hydra delegates, such as Pixar's Renderman, Chaos Group's V-Ray, OTOY's Octane and more. The render is then used as a thumbnail. Finally once the render is complete, the thumbnail is created and returned to the user as a panel (Fig. 4).

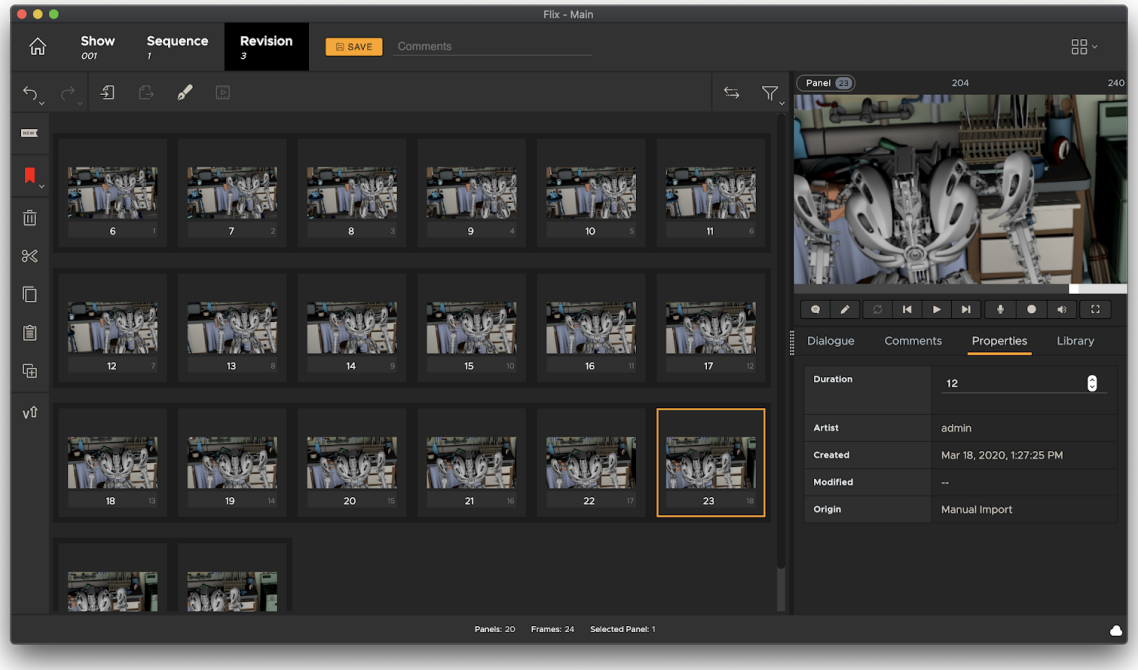

Fig. 11: Example of panels created by story beats created by different keyframes created in a layout tool.

5.3 USD Plug-ins Future Work

The generic and reusable plug-in architecture means that you can further specify the unique elements of your production's pipeline to access or generate exactly what you need. For example, the lighting dept. might only care about shading quality, while the animation dept. might want the latest version of the animation rendered as a video instead of a thumbnail. Similarly the materials dept. might simply want a contact sheet of all of the materials for a given asset. The included USD import plug-in is the first step in exposing this level of functionality.

5.4 Asset linking for different descriptors

Assets are the core of the storage system. Assets are a metadata only representation which can describe something which is to be stored in the system. The reason for the distinction between Assets and Media Objects, is that an asset may be represented by a collection of media objects. For example a 3D model would be accompanied by textures, and animations. These can be present using the media objects.

Assets can also be versionable using this separation between media objects. A new version of an asset can simply be associated with different media objects. This gives the ability to track the history of an asset as it evolves over time, while the underlying media objects are always persisted and retained in the system. This gives greater flexibility with reusing data which might have otherwise been lost behind newer revisions. Actions which previously would have been destructive, become non-destructive.

The mechanism of linking media objects collectively within an asset container allows us to store the relationship between an asset and its different representations.


```
▼{asset_id: 42, show_id: 1, created_date: "2019-05-15T12:12:09Z",…}
   asset_id: 42
   created_date: "2019-05-15T12:12:09Z"
  ▼media_objects: {,…}
   ▶artwork: [{id: 196, name: "redtest.psd", content_type: "image/vnd.adobe.photoshop", content_length: 272294,…}]
   ▶fullres: [{id: 199, name: "d0c450da-3775-4553-9ef6-cd42b40c404d.png", content_type: "image/png",…}]
   ▶scaled: [{id: 198, name: "c0151383-4126-49c0-b1ed-71389971dd23.png", content_type: "image/png",…}]
   ▶thumbnail: [{id: 197, name: "afe75a19-4821-41be-ba32-ce20344dd3fe.png", content_type: "image/png",…}]
  ▶owner_id: {id: 1, username: "admin", email: "flix-admin@foundry.com", created_date: "2019-04-18T11:00:54Z",…}
   show_id: 1
```


Fig 5. This JSON document represents an Assets and its associated underlying media objects.

6 CONCLUSION

The goal of this work package is to develop a system architecture allowing transcoding between various media types and formats. The primary learning from this, through testing and discussion with the product team, is that the final transcoding mechanism needs to be generic, flexible and able to meet a wide variety of demands. That is, focusing on particular transcoding routes between a given format and another isn't enough. Modern pipelines rely on being able to quickly adopt new formats and technologies (such as USD) while also supporting older legacy formats. The developed plug-in system easily allows for this kind of interchange between formats, only requiring the pipeline engineer to write the relatively simple 'glue' code.

USD has become highly prolific in recent years, and the decision to build an example plug-in to transcode USD to images is the perfect demonstration of this plug-in and transcode system. Not only does it provide the immediate value of visualising complex 3D scenes quickly, it is a stepping stone into the vast functionality offered by the USD library framework. With additional work with the

product team, it is expected that previz and storyboard artists could start to work with this. Some possible next steps are to add additional plug-in triggers and context-specific menus to allow more control over the transcoding. For example, selecting the camera to be rendered, or the workflow around re-transcoding of previously uploaded assets. As mentioned previously, the main challenge with USD is the technical barrier around integrating it into a pipeline as well as its organisational complexities. Further work is needed here to both better understand customer pipelines and simplify working with them.

The choice to develop the plug-in system in Python was vital. Python is the current language of pipelines and makes it simple to take an off-the-shelf or template plug-in and tailor it to more specific needs. It also offers another insertion point into the pipeline beyond transcoding, enabling users to run arbitrary code with the Flix pipeline in a managed and controlled manner. For example, to trigger email alerts or share meta-data with another system in the facility.

In summary, the creation of a simple-to-use plug-in system enables any of the consortium members to build their own Flix plugins to run the transcoding mechanisms they have been developing. Flix' scheduling system ensures that code executing in the system is scalable and reliable, eliminating the complexity of distributing computing for the users.

Further steps to be completed after this work package would be for other partners to develop their own plugins for Flix to fully display the narrative of a fully integrated ecosystem of smart asset classification and generation.

7 ACRONYMS AND ABBREVIATIONS

- USD - Universal Scene Description