Streamlined USD Integration for Rigging and Animation in Houdini

by

Ding Jia

Abstract

This paper introduces a method to streamline the transfer of character animation data between Houdini and the USD (Universal Scene Description) format. By developing a specialized Python application, the project aimed to make the conversion process between Houdini and USD formats more efficient for artists and engineers. The results show that the tool effectively simplifies the importing and exporting of USD animation data in Houdini, while ensuring the accuracy and quality of the data. This project contributes to the ongoing efforts to standardize 3D data workflows and enhances the adaptability of modern visual effects and character animation pipelines.

Contents

List of Figures

1 Introduction

In the dynamic realm of digital media and game creation, effectively handling intricate 3D elements—such as animations, meshes, and skeletal data—is crucial. As USD (Universal Scene Description) emerges as the new standard for modern 3D data interchange, integrating it into traditional production tools like Houdini poses significant challenges. Ensuring interoperability between USD and older systems, particularly in managing USD data, presents numerous technical obstacles.

While Houdini has made notable progress in managing static USD data, handling dynamic data like skeletal animation remains complex and time-consuming, requiring substantial effort from art professionals. To address these issues, this project introduces a methodology that seamlessly integrates USD's advanced functionalities with conventional animation and visual effects software such as Houdini. This integration allows artists and technicians to adopt the new data format without disrupting their existing workflows or decreasing productivity.

This project develops user-friendly Houdini tools that convert outdated file formats such as FBX into USD, and then import the USD data into Houdini's native format. These tools ensure optimal productivity while preserving the accuracy and integrity of the data throughout the conversion process. The objective is to expand the range of options available to artists and technicians in 3D material development, simplify the learning curve for new technologies, and enhance overall creative and production efficiency.

The subsequent sections provide an overview of the latest advancements in data interchange technology, examine the technical challenges addressed by this project, and outline the solutions designed to enhance interoperability between USD and Houdini. This thesis contributes to the field of digital content creation by offering valuable insights and tools that can be applied in various production scenarios.

2 Background

2.1 Introduction to Data Interchange in 3D Animation

Comprehensive knowledge of various 3D model file formats is crucial in the field of 3D modeling, as it significantly influences workflows, compatibility, and project success.

2.1.1 Traditional Methods

GLTF/GLB (Graphics Library Transmission Format/Binary) is widely regarded as the optimal solution for efficiently transferring 3D models over the internet. It is based on an open industry standard created by the KHRONOS Group. According to Library of Congress (2019), glTF assets can include scenes, models, geometry, skins, shaders, texture files, and animation data. gLTF/GLB files are well-suited for web-based apps, Virtual Reality (VR), and Augmented Reality (AR) experiences due to their small file size and compatibility with WebGL.

Originally developed by Kaydara for the Filmbox program, FBX has become a widely accepted format for exchanging 3D data in various industries. This format supports complex features such as animation and texturing, making it extensively used in gaming and entertainment sectors. Its comprehensive support for different 3D content types makes it ideal for transferring data between various software applications.

However, just as modelo (2024) said, the proprietary nature of the FBX format can disadvantage some users due to its limited interoperability with non-Autodesk software.

2.1.2 Why USD

Just as Pixar (2024a) said, Pipelines capable of producing computer graphics films and games typically generate, store, and transmit great quantities of 3D data, which we call ”scene description”. Each of many cooperating applications in the pipeline (modeling, shading, animation, lighting, fx, rendering) typically has its own special form of scene description tailored to the specific needs and workflows of the application, and neither readable nor editable by any other application. Universal Scene Description (USD) is the first publicly available software that addresses the need to robustly and scalably inter-

change and augment arbitrary 3D scenes that may be composed from many elemental assets.

The film and animation industry frequently utilises USD for exchanging assets between different software packages, facilitating collaborative processes, and rendering. The software support is currently limited, however there is an increasing trend towards integration. May require more time and effort to learn compared to other formats.

USD offers specialised modules for managing animation data, including UsdSkel, which is designed for handling the skeletal structure and skinning data of character animations. UsdSkel enables users to define intricate skeleton animations as USD files, facilitating smooth interchangeability across many 3D software platforms such as Houdini, Maya, Blender, and others. The hierarchical nature of USD enables users to effectively manage animation data at several levels, offering a significant level of flexibility and expandability.

2.2 Houdini and USD Integration

Solaris, introduced in sidefx (2024c), represents Houdini's comprehensive tooling and support for USD. It includes enhanced USD support within the viewport and introduces a new network type known as LOPs (Light Operators), which function similarly to SOPs. In LOP networks, each node processes an incoming USD scene, modifies it, and outputs a new scene.

The USD export capability in Houdini is extensively utilised to store intricate animated scenes as USD files, which can be employed in alternative platforms or render pipelines. The produced USD files include both geometry and comprehensive animation data, including skeletal animation, keyframes, and deformation information. This enables seamless collaboration across many teams in expansive production environments, facilitating the use of diverse software applications without the need to be concerned about potential issues arising from incompatible data formats. As an illustration, Houdini users have the ability to generate character animations and then export them in the USD format. This allows for additional manipulation or rendering in other tools that support USD.

Regarding the workflow for character animation. Houdini utilises LOP to import USD resources, including animation. The imported LOP node is then used to import and show the character model in the SOP node called USD Character Import.

To facilitate the export process, begin by utilising the SOP Character Import LOP,

which is encapsulated within the SOP Import. Then, employ USD export to carry out the actual exporting. However, it is worth noting that users frequently encounter numerous challenges throughout this procedure.

2.2.1 Current Challenges with USD in Houdini

Although USD has emerged as a widely accepted format for exchanging 3D data across different platforms, many users still encounter substantial difficulties when using it in practice.The USD export tool in Houdini enables users to export intricate animated scenarios to the USD format, facilitating its use on many platforms. Nevertheless, certain users have discovered that these tools may not consistently perform as anticipated throughout usage.

As an illustration in sidefx (2024b), a user's attempt to export an animated scene using Houdini's USD export node resulted in exporting solely the mesh data, excluding the animation data. This may be attributed to mismanagement, inherent limitations of the tool, or a deficiency in comprehending the tool's functionality.The user reported that the Geometry is OK, but the Animation is not producing any results.

This problem implies that although USD has strong capabilities, its usability and dependability still have significant room for improvement in real-world scenarios. This study aims to enhance the intuitiveness and reliability of exporting animation data by upgrading and refining the tools and workflows involved.

2.2.2 FBX Character Import

To create a new data exchange pipeline, one can utilise Houdini's existing technological assistance and construct the framework of a new data exchange tool by comprehending Houdini's approach to conventional character animation. FBX is a widely used format among traditional data interchange formats, as discussed above. Users commonly utilise the FBX format when they desire to import external resources from other platforms.

Houdini offers the FBXCharacterImport advanced tool node for accessing FBX resources. Upon further examination of this intricately structured node, it is evident that it consists of three components: skin, skeleton, and animation. These components align with the FBX Skin Import Sop, the FBX Animation Import Sop (outputting the rest pose), and the FBX Animation Import Sop (outputting the animation) respectively. After undergoing certain procedures, the node will be linked to boneDeform for its ultimate implementation.

The FBXCharacterImport node is a high-level component that merges the FBX Skin Import and FBX Animation Import SOPs. It creates a complete character with geometry, which can be easily deformed using the Joint Deform SOP.

The FBX Skin Import node is used to import skin geometry from an FBX file. This geometry can be deformed along with an animated skeleton, which can be imported using the FBX Animation Import SOP. Usually, FBX Character Import is utilised instead.

The FBX Animation Import node is used to import animation from an FBX file using a skeleton based on geometry.

Upon further examination of the FBXCharacterImport high-level node, it becomes apparent that it utilises two distinct types of Outputs from FBX Animation Import as ports for the purpose of importing collected skeleton data and animation data. The former yields skeletal data for a T-pose, representing the skeleton in a motionless state without any animated effects. The bone information will pertain to the essential data in FBX Skin Import, namely for the purpose of applying skinning deformation. The second output consists of the animation's data, namely the bone data obtained from the bind pose.

In accordance with the concept of how Houdini manages the import of FBX character animation resources that constitute the project's structure, all imports and exports will individually handle the three components: skeleton, skin, and animation data.

2.3 UsdSkel

Pixar (2024d) said UsdSkel defines schemas and API that form a basis for interchanging both skeletally skinned meshes and joint animations between DCC tools in a graphics pipeline. The composition comprises Joint and Skinbind. The Joint is composed of bones from USDSkeleton and Prim from UsdSkelRoot, while animations are controlled by Prim from USDSkelAnimation. Animations are considered to be generally autonomous, indicating that various animations can be transferred to the same skeletal structure.

UsdSKelRoot is established as the primary node of Skeleton, serving to specify the structure of the Joint and store the BindPose. In USD, BindPose data will display as the variable named bindTransform which means the transform data in world space. In contrast to Maya's Joint or Houdini's Rig system, the bones in USD are represented as a single Prim that contains the joint's topology. The animations of UsdSkel are defined as AnimationPrims, which are independent entities. In animation part, the transform data of joint belong to the local space.

Lastly, we have the BindSkin. The SkinWeight information for USD is stored in the MeshPrim, which serves as the container for the geometric information of the mesh or skin. There are two crucial bits of information: Variables:The properties ”skel:jointIndices” and ”primvars”.

The Weight property differs from the other properties as it is classified as ”primvars”. Primvar is an abbreviation for ”primitive variable” and is employed to modify the value of the surface and volume (such as UV) of a primitive.

In Pixar (2024b), there is a simple illustration of a fundamental animation involving a three-segmented basic shape, which is formally provided by USD in the context of USDSkel.

2.4 Development Language Selection

2.4.1 Houdini Python API

This project opted to utilise Houdini's Python API instead of the HDK (Houdini Development Kit). Opting for Houdini's Python API over the HDK (Houdini Development Kit) offers more accessible system-level interactions. While the HDK provides more efficient, low-level access, the Python API supports the majority of necessary data import and export procedures. It facilitates script execution and automation within Houdini's UI, aligning well with the project's simplicity and workflow requirements.

Python is a cross-platform programming language that allows code to be executed on several operating systems, such as Windows, macOS, and Linux, without any need for modification. C++, although it is cross-platform, often encounters challenges related to the compatibility of compilers and libraries specific to each platform, hence increasing the intricacy of the development process.

Python is highly capable of handling the animation and skeleton data in this project, including the animation data from Mixamo, which consists of 65 joints. Its performance is more than sufficient for this task. Although HDK may exhibit superior performance, this benefit is typically only notable when handling exceedingly big datasets. The Python API's ease of use, versatility, and rapid development capabilities make it a preferable choice for most real-world circumstances. Unless a project necessitates exceptionally high performance, the straightforwardness and effectiveness of Python typically surpasses the underlying management and intricacy of HDK.

In addition, Python has the capability to directly interface with Houdini's user inter-

face, facilitating the process of debugging and development. This feature is especially advantageous for quick and efficient iteration and adjustment.

2.4.2 USD Python API

The primary motivations for selecting USD's Python API over the C++ libraries were to enhance development productivity, streamline workflow, and leverage Python's strong interoperability with pre-existing tools and ecosystems. Although C++ offers performance advantages, Python offers adequate functionality and performance support for most projects that do not require extreme performance. Additionally, Python greatly reduces development complexity.

Integration with Houdini: Houdini's robust built-in Python support facilitates the seamless incorporation of USD's PythonAPI into preexisting Houdini processes. Python enables developers to easily produce, alter, and manipulate USD data within Houdini without the need for further plugins or compilation processes.

Interoperability with other Python libraries: The Python environment offers a wide range of robust libraries (such as NumPy, Pandas, Matplotlib, etc.) that may be utilised with the USD API to improve data processing and analysis. Such integration can be achieved in C++, albeit it entails somewhat greater complexity.

3 Implementation

The objective of this project is to design an efficient method for transferring skeletal animation and mesh data between Houdini and USD. Given that USD (Universal Scene Description) has become one of the prevailing standard for contemporary 3D data transmission, professionals require a streamlined and user-friendly solution to effortlessly export animation data from Houdini to the USD format, or import from USD to Houdini for subsequent manipulation. Nevertheless, current methods continue to pose numerous difficulties when handling intricate animation data, particularly in the context of exchanging data across different platforms and software. The objective of this project is to streamline the animation data transmission process and optimise data management by creating a Houdini tool using Python.

This initiative prioritises the optimisation and user-friendliness of data transfer. The Python API was selected over the more intricate HDK C++ interface due to its sufficient functionality in managing most character animation data. This decision guaranteed that the tool was rapidly developed and effortlessly maintained. Although C++ may offer better efficiency for handling extensive data processing, Python is sufficiently performant for typical animation operations within the confines of this project.

The import and export tools for project production are presented as HDA (Houdini Digital Assets).

In sidefx (2024a), the role of a Houdini Digital Asset (HDA) is to define a custom operator. Usually, an HDA encapsulates a node network that, as a whole, performs an operation intended for this custom operator. However, an HDA may specify its operation as script, for example, in case of a Python SOP or a GLSL shader asset. It turn the work into reusable custom nodes called digital assets.

The following are node diagrams of the imported and exported HDA resources.

In the import HDA node graph, use the mesh node to import skin information, which includes all vertices of the geometry and skinning information. Use the skeleton node to get the skeleton structure information, and then use captureproximity to create the skinning result. After obtaining the skinning information, edit the skeleton to adjust any misalignment of the joints. Use the anim node to import the animation. To speed up

the animation, bake the animation file in advance, and then import it into boneDeform together with the previously processed bones and skin information to visualize the animation effect.

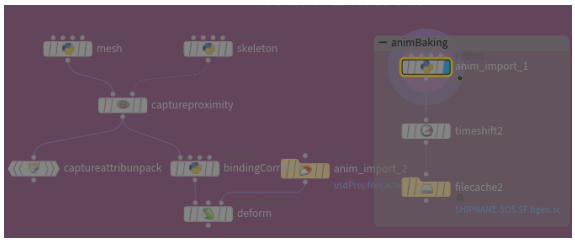

Figure 3.1: import HDA node graph

In export HDA node graph, it uses fbxSkinImport to get the skin information. Then it get the skeleton and animation information from two different type of fbxAnimImport node. To ensure that the animation resources are effective, three nodes are accessed using the boneDeform node as the final node.

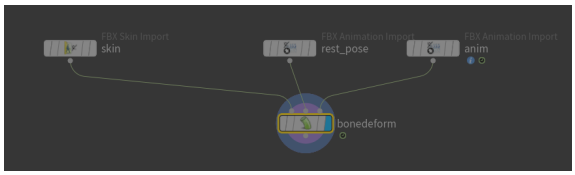

Figure 3.2: export HDA node graph

3.1 Mesh

Utilising the Python API of Houdini to generate a USD (Universal Scene Description) provides insight into the necessary steps for achieving data interchange. kiryha (2024) showcased a project that demonstrates a basic mesh structure. A geometric mesh necessitates the inclusion of three key components: the number of vertices that constitute a face (faceVertexCounts), the sequential arrangement of the points that create the face (faceVertexIndices), and data regarding the location of each individual point (points).

The skinBind operation requires two attributes: jointIndices and jointWeights.

The order of characteristics should align with the sequence of points. The former refers to the ordinal number assigned to the joints that impact a specific mesh vertex, while the latter represents the degree of influence these joints have on a single joint.

The BoneDeform node in Houdini is used for applying skinning to bones. It offers three options: LBS(Linear Blend Skinning), Dual Quaternion Skinning, and Blend Dual

Quaternion and Linear Skinning. These types of skinning are sufficient for the general needs of animation. This project will not include the creation of a skinning part. It will utilise the built-in skinning functionality with boneDeform.

3.1.1 Import

Begin by generating points and assigning the attribute ”P” to position datas in the USD, which includes the X, Y, and Z coordinates. Calculate the boneCaptures by considering the weights or numbers assigned to each point of the mesh in the USD. Then, retrieve the skin information for a point in this specified number of batches. The required number of BoneCaptures is equal to twice the number of weights or numbers. In other words, each point corresponds to many combinations of bone numbers and weights, and the total sum of all the weights of the bones affecting the point is 1.

Once all the points have been created, the createPolygon() method is utilised to connect the points and produce an unclosed polygon line with the predetermined number of points on each face.

The import process for the mesh's geometry and the skin information linked to the skeleton has been finished.

3.1.2 Export

Begin by processing the static geometry information exclusively.

Initially, obtain the geometric data. Retrieve the coordinates of all the points within the skin node. These positional coordinates are necessary for the mesh information in the USD file and are stored as the value of the variable point3f[] points.

To process the information for each face, switch to the primitive mode of the node and retrieve the vertices of each face. Store the ordinal information, number of vertices, and normals for each face.

Now proceed to establish the geometric data: The geometric data is stored on a Prim object of the UsdGeom.Mesh type. Utilise the Python API offered by USD to save the previously acquired point coordinates, vertex count per face, ordinal order of each face, and normal data. Please be aware that this project utilises the right-handed order, specifically referred to as UsdGeom.Tokens.rightHanded.

Subsequently, the skinning data will be processed. The process of reading all points in the fbxSkinImport is referred to as boneCapture. The Houdini interface presents this as ”boneCapture regn” and ”boneCapture w”. Values having a weight of -1.0 are treated

as 0 to avoid any unexpected outcomes or limitations. The two properties are linked to the mesh using USD's bindingAPI.

Finally, the geomBindTransform refers to the transformation matrix of the mesh when it is attached to the bone. Therefore, it is enough to calculate the transformation matrix of the bone's root node when it is attached. In other words, the result is obtained by multiplying the bindTransform of the root node by the inverse of its local spatial transformation matrix.

3.2 Skeleton

Intended for the skeletal structure. The fundamental framework consists of skelRoot and skeleton. Skeletons include joints, bindTransforms, and restTransforms.

Joints refer to the whole pathways of each joint, and the USD mandates that joints must be organised in a manner where the parent joints are positioned before the child joints.

This project utilises a sorting method to get all joints and arrange them in ascending order based on the length of their respective paths. However, for additional use, it also identifies the root joint and retains it.

The bindTransforms property of a Skeleton returns the transformation of each joint in world space when the skeleton is initially bound.

The restTransforms property of a Skeleton contains the local space rest transforms for each joint.

As stated by the USD official Pixar (2024b),The restTransforms may be considered optional, but if left unspecified, the Skeleton must be bound to a complete, non-sparse animation. It is common for the restTransforms to refer to the same pose as the bindTransforms, only in joint-local space, rather than world space, but this is not required.

The FBXAnimImport node in Houdini will calculate and export both the restTransform and animation data. This project aims to incorporate and export the restTransform as the fundamental component of the skeleton to facilitate multiple potential uses in the future.

3.2.1 Import

Retrieve skeleton data: extract the complete paths of the joints from the USD file and construct one dictionary, a dictionary that represents the parent-child relationships between the joints based on their pathways. The matrix of the joint can be exported directly

using the API to obtain the necessary components. For example, the function ExtractTranslation() yields the P value as the resulting value. It is also feasible to directly obtain the value of a specific position to retrieve the appropriate component. Once the positions of the points have been determined, it is necessary to establish connections between them in order to create a skeletal structure. Each pair of parent-child relationships should be used to connect the appropriate lines.

Establishing the framework: To successfully input the relevant data, it is imperative to generate point attributes for the geometry, namely 'name', 'path', and 'transform'. These points, also known as joints, are assigned both absolute position values and customised values for the three attributes. Furthermore, these points are combined together as polygon line.

3.2.2 Export

To begin, the initial action is to acquire the skeleton data, which is derived from the rest pose output of fbxanimimport node. It is essential to obtain the complete path of all the joints in the skeleton, following a specific order. These joints, represented by the uniform token[] joints in the USD file, store this information in the path.

When users download FBX character resources or other FBX resources, the imported animation nodes may sometimes wrongly keep mesh information. This issue can be resolved by identifying the path without parent joint which is also not the root node after splitted by '/'. When it comes to obtaining the root node, there is a more straightforward and efficient approach. Since the character skeleton rigidly constructs its structure starting from the root node, it is enough to obtain the name of the first node on the longest path.

The bindTransforms and restTransforms, which are both arrays of matrix4d, mean the transform matrices in world space and local space. In the FBXAnimImport function, the bindTransform operation solely requires retrieving the P attribute and transform attribute of the point. As the matrix is defined in world space, the point's absolute location is equivalent to the translation component. The transform's nine values represent the combined rotation and scaling components.

Create one dictionary to handle intricate data, and assign the name attribute of the point as the key, and the merged bindTransforms as the value.

In order to properly handle parent-child connections in restTransforms, it is necessary to first create a dictionary to represent the parent-child relationship of the connection

point. Paths provide a practical way to create such connections. After splitting the path using the slash notation, the last element in the sequence represents the name of the subordinate joint, while the penultimate element in the sequence represents the name of the superior joint.

With a dictionary of parent-child relationships keyed to the child joint's name and a dictionary of bindTransforms keyed to the joint's name ready, it is time to process rest-Transforms.

If the joint is a root joint, then the restTransform of the joint is equivalent to its bind-Transform. Typically, the restTransform of each joint is calculated by taking the inverse of the current joint's bindTransform and multiplying it by the bindTransform of the parent joint.

Next, establish the skeleton information by setting up the skeleton using the joint routes that were previously produced, utilising UsdSkel.Topology is used to construct the structure of the skeleton. The USD system should utilise Tokens as the input characters. It is important to carefully manage any illegal characters that may arise. while using the mixamo resource, it is possible for the resource to have a namespace. This namespace is commonly used for animation resource management. However, while managing the resource in USD, if the namespace contains invalid characters, it can cause errors. Therefore, it is necessary to replace the unlawful characters.

Lastly,store bindTransform and restTransform datas in USD.

3.3 Animation

3.3.1 Import

Within the realm of animation data. There is a strong correlation between skeleton and joint movement data.

Firstly, the data on the skeleton and animation must be retrieved: Retrieve the skeletal data: Retrieve the full path of the joints using the topology. Extract all the bindTransforms from USD and obtain the translation components as the positional data of the points. Determine the serial numbers of the parent-child joints through the paths and store them in the list of skeletal lines connecting the joints.

Retrieve animation data: Retrieve the rotational, translational, and scaling elements of the animation primitive. Due to the combination of scaling and rotation components in the transform, Houdini lacks an API to directly specify their values. Therefore, we utilise

the Gf library's API to establish the three components and merge them into a 4x4 matrix in Gf. Subsequently, we convert this matrix into the matrix4 format employed by Houdini.

Begin the traversal process from the root node. Initially, obtain the bindTransform of the root joint. Then, proceed to compute the bindTransform of each joint. This value is determined by multiplying the bindTransform of the parent joint with the restTransform of the current joint. Finally, store all the computed values in a dictionary, using the frame value as the key.

The procedure for computing the restTransform of all joints for all frames is as follows:

Algorithm 1 Compute Global Transformation Matrices from USD Animation Data

Input: USD Animation data and joint hierarchy
Output: Dictionary of global transformation matrices for all frames and joints
Initialize an empty dictionary `global_matrices`
for each TimeSample t **do**
    Initialize an empty list `world_matrices`
    Get rotation, scale and translation values of this frame
    for each joint j **do**
        Extract the rotation, scale and translation components for joint j at TimeSample t
        Combine three components to a 4×4 matrix `_matrix`
        Convert the Gf Matrix `_matrix` to Houdini Matrix `matrix`
        if j is root joint **then**
            Append `matrix` to `world_matrices`
        else
            Find the parent joint `[parent_index]` of j
            `parent_matrix` $\leftarrow$ `world_matrices[parent_index]`
            `global_matrix` $\leftarrow$ `parent_matrix` $\times$ `matrix`
            Append `global_matrix` to `world_matrices`
        end if
    end for
    Store `world_matrices` in `global_matrices` dictionary at TimeSample t
end for
return `global_matrices`

To set up all datas, generate point attributes 'name', 'path', and 'transform'. Generate

points (joints) and apply custom values to two attributes (name and path). Concatenate the points.

The process of determining the current frame in Houdini involves extracting the current frame information from the transformation matrix data dictionary, which contains the joint data for all frames. Traverse all bone points to extract the matrix data of the current joint from the joint data of the current frame. Separate the translation component and the rotation scaling component, and assign these two components to the P attribute and the custom transform attribute.

3.3.2 Export

Algorithm 2 Compute Rest Transform Matrices and Store in USD

Input: List of parent-child joint relationships: `parent_indices`,all transform data of all joints in world space in all frames
Output: USD file with transformation matrices in local space for all joints and frames
Get the frame range from the animation data
Get the Animation output node of fbxAnimImport
Create a list `parent_child_list` of parent-child relationships based on joint topology, setting the parent of the root node to -1
for each frame f in the frame range **do**
    Initialize empty list `matrix_joints_frame`, `translations_frame`, `rotations_frame`, `scales_frame`
    `joint_index` ← 0
    for each joint $point$ and the index of joint $index$ in the joint topology order **do**
        if $index$ is not the index of mesh **then**
            Get the transform, position (`P`), and name attributes of current joint
            Construct the world space transform matrix `bindTransform` for joint $point$ using Houdini's `matrix4`
            `matrix_joints_frame[joint_index]` ← `matrix4`
            `joint_index` ← `joint_index` + 1
        end if
    end for
    for `index`, `global_matrix` **in** `enumerate(matrix_joints_frame)` **do**
        if current joint's parent joint is -1 **then**
            `local_matrix` ← `global_matrix`
        else
            `parent_global_matrix` ← `matrix_joints_frame[parent_indices[index]]`
            `local_matrix` ← `global_matrix` × `parent_global_matrix.inverted()`
        end if
        Store the rotation, scale and translation in `translations_frame`, `rotations_frame`, `scales_frame`
    end for
    Set all rotation, scale and translation data to USD
end for

The method retrieves the transformation matrix data for each joint in every frame and stores this data in the Prim of the Animation. Each frame is assigned as a key for the corresponding data.

The list of joints will be saved in the Prim of the Animation of USD.

3.4 Result

This is a display of the results of interconverting any animation resource downloaded from mixamo into USD format in Houdini.

3.4.1 Import

This is the result of the import in houdini.

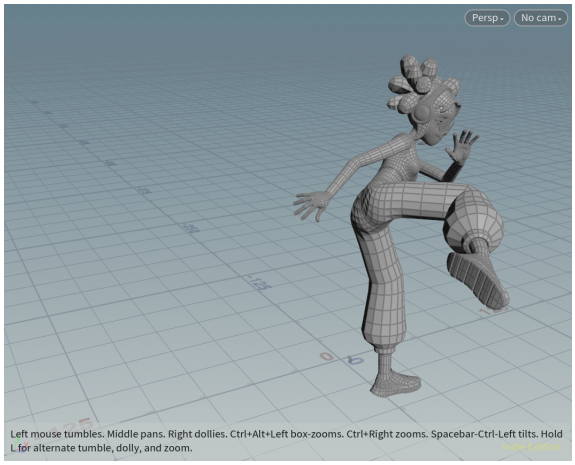

Figure 3.3: Import result

3.4.2 Export

This is the result of the export in usdview(usdview is the most fully-featured USD tool, combining interactive gl preview, scenegraph navigation and introspection, a (growing) set of diagnostic and debugging facilities, and an interactive python interpreter Pixar (2024c)).

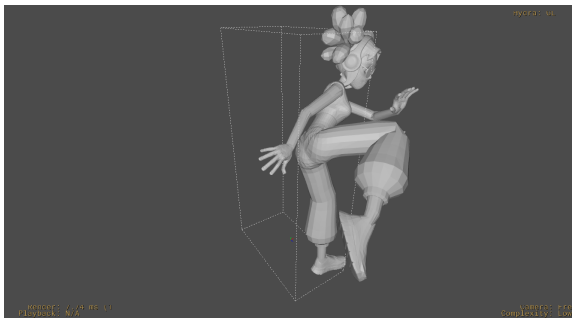

Figure 3.4: Export result

3.4.3 Integrity of data

The following set of images represents the integrity of the data imported and exported from the animation file.

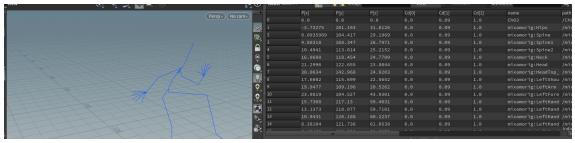

Figure 3.5: Skeletal animation source data

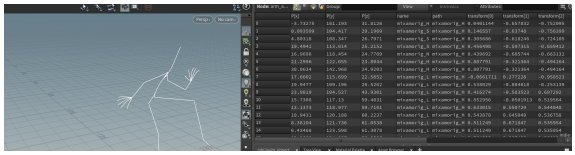

Figure 3.6: Imported skeletal animation data

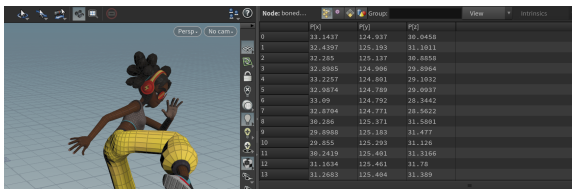

Figure 3.7: Final source data

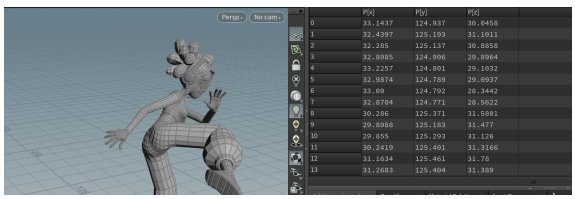

Figure 3.8: Final data

3.5 Advantage

3.5.1 Importing with FBX

FBX is a highly compatible format used for exchanging complicated animation data, such as skeletal animation and mesh information. FBX is widely compatible with leading 3D modelling and animation tools, making it an excellent choice for transferring data from other platforms (such as Maya, 3ds Max) to Houdini.

The solution enhances workflow flexibility by enabling the conversion from FBX to USD. Users have the ability to select the most suitable tools and formats for their project requirements, whether it be for rendering, compositing, or additional animation processing.

This strategy enables users to optimise the utilisation of current resources. For instance, it is possible that they already possess an extensive collection of assets stored in FBX format, and the solution simplifies the process of converting these assets to USD without the need for reconstruction.

This project offers a progressive transition option for teams who are already experienced with FBX workflows. It helps them to gradually adjust to working with USD, instead of being immediately faced with alien tools and formats.

3.5.2 An agent is unnecessary

In order to utilise Solaris, a software specifically developed for USD (Universal Scene Description), within Houdini, it is necessary to employ the LOP (Lighting Operator) node. However, when integrating with pre-existing SOP (Surface Operator) processes, it is typically need to utilise a SOP proxy, such as importing SOPs, to access USD's LOP Network before proceeding. This project facilitates the direct import and export of USD animations in the SOP node, bypassing the requirement to initially utilise LOP and SOP agent.

3.5.3 Enhancing user experience and optimising process

The solution enhances workflow efficiency by offering users a more streamlined and user-friendly interface, enabling them to carry out all tasks inside a familiar setting. This minimises the necessity to acquire new skills or equipment.

Alleviate user workload: Through the automation of intricate data processing tasks, the project lessens the workload on users in terms of data translation and management.

This enables users to dedicate more time and energy to engaging in creative and productive activities.

3.6 Drawback

It is necessary to incorporate the capability to modify the animation source in order to showcase an additional benefit of USD.

Currently, the original animation import sets the value of each frame by verifying the current frame for the animation portion. In order to increase the speed of the animation, the project opted to pre-render the animation upon import.

It is necessary to devise a method for configuring the attribute of the point's keyframe.

4 Conclusion

This project successfully developed a suite of tools designed to facilitate the transfer of 3D character animation data between Houdini and USD. These solutions effectively manage the complexity of data exchange across multiple platforms, significantly streamlining the workflow for artists working with USD files in Houdini. By leveraging the Python API, the project achieved efficient data conversion and processing while maintaining a high degree of user-friendliness.

The project met key objectives, including the seamless transfer of mesh, bone, and animation data between systems, and the preservation of data integrity across different platforms.

However, certain challenges were encountered. The process of importing animations remains somewhat laborious. To accelerate the use of pre-baked animations, it is recommended to improve this method so that animations can be displayed directly and quickly without requiring baking. Additionally, there is room for further optimization in developing skin binding data. Specifically, the current approach of using the boneCapture property in the captureproximity node to create the attribute needed by the boneDeform node should be refined. Future efforts should explore ways to customize this property in advance, eliminating the need for subsequent skin data editing.

In addition to this, the handling of frames is also rough, currently only the frame range in Houdini is used as the starting and stopping frames of the USD animation, which needs to be set in further detail.

Future research could focus on enhancing the efficiency of handling large-scale animation data. Furthermore, expanding the tool's capabilities to support transitions between multiple animations and more complex animation scenes would be highly beneficial. In conclusion, this project provides a solid foundation for data interchange in 3D animation production and sets the stage for future advancements in this area.

References

kiryha, 2024. Pixar-usd-python-api. URL https://github.com/kiryha/Houdini/wiki/Pixar-USD-Python-API, [Accessed: 2024-7-22].

Library of Congress, 2019. gltf family. URL https://www.loc.gov/preservation/digital/formats/fdd/fdd000498.shtml, [Accessed: 2024-7-26].

modelo, 2024. Fbx review: Pros and cons. URL https://www.modelo.io/damf/article/2024/06/23/1436/fbx-review--pros-and-cons?hl=en, [Accessed: 2024-08-01].

Pixar, 2024a. Introduction to usd. URL https://openusd.org/release/intro.html, [Accessed: 2024-7-22].

Pixar, 2024b. Schema intro by example. URL https://openusd.org/release/api/_usd_skel__schema_overview.html, [Accessed: 2024-7-12].

Pixar, 2024c. Usd toolset. URL https://openusd.org/release/toolset.html, [Accessed: 2024-6-20].

Pixar, 2024d. Usdskel : Usd skeleton schema and api. URL https://openusd.org/release/api/usd_skel_page_front.html, [Accessed: 2024-7-12].

sidefx, 2024a. Houdini digital assets. URL https://www.sidefx.com/docs/hdk/_h_d_k_h_d_a_intro.html, [Accessed: 2024-07-04].

sidefx, 2024b. Usd animation export. URL https://www.sidefx.com/forum/topic/95553/?page=1#post-420216, [Accessed: 2024-6-26].

sidefx, 2024c. Usd basics. URL https://www.sidefx.com/docs/houdini/solaris/usd.html, [Accessed: 2024-7-13].