Maya-Houdini USD Rigging Schema for Streamlined Animation Workflow

By:

Yuqian(Ashley) He

N.C.C.A Bournemouth University

MSc Computer Animation and Visual Effects 2022-23

August 4, 2023

Abstract

The paper describes an innovative method for migrating mesh and skeletal data from Autodesk Maya to SideFX Houdini in the 3D animation domain. It provides an alternative to FBX and Alembic (ABC) data transfer processes by leveraging the robust nature of the Universal Scene Description (USD) format. In the first phase of the project, two bespoke plugins were developed using the C++ Maya API. These plugins were specifically engineered to extract the mesh and skeletal data from Maya. Subsequently, this extracted data was transmuted through the USD API into a compatible USD format. Using the C++ Houdini Development Kit (HDK), two further plugins were crafted to translate USD data into a form that Houdini can interpret and render. In essence, this project reveals a new paradigm for data interchange between Maya and Houdini, thereby enriching the range of data transfer methods available in the animation industry.

Contents

List of Figures

List of Tables

1 Introduction

In the ever-evolving field of 3D animation, the constant push for efficiency and improved compatibility between different software platforms is an enduring challenge. Autodesk Maya and Houdini, leaders in their respective domains of 3D animation and procedural visual effects, often serve as central pillars in a production pipeline. However, transferring complex data such as mesh and skeleton structures between these two platforms can present hurdles. These challenges often arise from the intrinsic complexities of 3D data and the varied data structures that different software systems employ.

Against this background, this project aims to provide a new way for mesh and skeleton data to be transferred from Maya to Houdini. By developing a new method, we seek to enhance the flexibility of data interchange options between these two software. Providing artists and technicians with another data transfer method allows them to address unique and demanding project requirements, increasing overall production pipeline efficiency.

In addition, it is important to clarify that this project does not seek to replace existing, time-tested methods of data transfer like FBX and Alembic (ABC). These formats have served the industry reliably for many years, allowing for the efficient exchange of 3D graphics and associated data across various platforms. Instead of replacing existing methods, this project aims to expand 3D artists' range of options. In doing so, it recognizes the value of having multiple robust solutions in a field as dynamic and demanding as 3D animation, where a one-size-fits-all approach often falls short.

2 Background

2.1 Traditional Data Transfer Methods

2.1.1 Filmbox(FBX)

Since its introduction in 1996 by Kaydara and subsequent acquisition by Autodesk, the Filmbox (FBX) format has become a recognized standard in the animation and 3D graphics industry. The ubiquity of this proprietary format can be attributed to its ability to support diverse data types, including meshes, skeletons, animations, and textures(McHenry and Bajcsy, 2008). The robust nature of FBX is further emphasized by Autodesk's provision of SDK bindings in both C++ and Python, coupled with a format description available through the FBX Extensions SDK(Lazor et al., 2019). Notable tools such as Blender and The OpenEnded Group's Field have exploited these SDKs to craft their FBX importers and exporters, attesting to the format's adaptability(Ardolino et al., 2014).

FBX is instrumental within the animation workflow, serving as a conduit for the uninterrupted data transition between disparate 3D software applications. A salient feature of the FBX format is its provision for both ASCII and binary on-disk representations, offering a balance between human readability and storage efficiency. The ASCII variant showcases a tree-structured document delineated by clear identifiers, whereas the binary counterpart, though undocumented officially, is explicated through an informal specification by the Blender Foundation(Lee et al., 2019).

However, the FBX format does exhibit certain constraints. Its proprietary nature means that its evolution is exclusively governed by Autodesk, which can spawn compatibility challenges with software external to Autodesk's ambit. The absence of a formalized binary specification can also pose dilemmas for developers aiming to integrate this format. Additionally, while FBX accommodates an extensive spectrum of 3D data types, it isn't all-encompassing, potentially instigating data losses or the need for alternative strategies during data transmission. Notwithstanding these impediments, the multifaceted nature and holistic support offered by the FBX SDK emphasize the format's paramount role within the 3D production trajectory, solidifying its position as a cornerstone in animation processes.

2.1.2 Alembic (ABC)

Originating from a collaborative endeavor between Sony Pictures Imageworks and Industrial Light Magic in 2011, Alembic (ABC) stands as an open-source computer graphics interchange framework. After its unveiling at SIGGRAPH 2011, its adoption skyrocketed across the animation and visual effects realms. At its core, Alembic aims to facilitate

the seamless exchange of animated geometric data among disparate teams and software utilities. This interoperability becomes paramount when multiple departments within an entity or even different studios collaboratively engage on projects.

Alembic's versatility emerges as a cardinal strength. It embraces prevalent geometric representations dominant in the industry, including polygon meshes, subdivision surfaces, parametric curves, NURBS patches, and particles. Beyond these geometric facets, Alembic deftly manages the conveyance of transform hierarchies and camera parameters. Recent iterations have augmented its capabilities, now endorsing materials and lights (Laforge et al., 2018). Distinct from archiving intricate dependency graphs of procedural instruments, Alembic utilizes a 'baking' paradigm. Herein, outcomes from intricate procedural motion generators and simulations are chronicled as distinct geometric configurations for individual frames (Lazor et al., 2019). This methodology situates Alembic as an invaluable asset in 3D productions, given its proficiency in tackling elevated complexity while preserving tractable file dimensions, especially in projects permeated with elaborate dynamics and animations.

Yet, Alembic isn't devoid of constraints. Foremost is its ineptitude in encapsulating intricate procedural dependencies within scene delineations, an inherent byproduct of its 'baking' strategy. Such a system implies that post-baking procedural modifications remain absent from archived content. Moreover, the Alembic paradigm can engender voluminous file sizes for exceptionally intricate scenes. And, albeit Alembic encompasses myriad geometric representations, it isn't exhaustive, potentially inducing obstacles in specific operational flows. Nevertheless, Alembic's adeptness at managing data of prodigious complexity without sacrificing file manageability cements its esteemed status within the 3D domain.

2.2 Universal Scene Description (USD)

Conceived by Pixar, the Universal Scene Description (USD) emerges as a pioneering framework dedicated to optimizing 3D graphics data interchange. Since being open-sourced in 2016, USD has consistently championed collaborative endeavors and non-destructive editing, securing its foothold in diverse sectors, ranging from visual effects to robotics and CAD.

In the realms of animation and VFX, USD is effecting a paradigm shift, facilitating enhanced workflows and symbiotic collaborations. It furnishes a holistic representation of scene components—from intricate geometries to fluid animations—and ensures continuous updates without jeopardizing active engagements. Its robust versioning mechanism encourages non-destructive modifications and the concurrent existence of myriad scene iterations, streamlining review cycles. Owing to its open-source lineage and software-agnostic stance, USD optimally bridges data transitions across disparate production utilities.

Contrasting with contemporaries like FBX and ABC, USD surpasses the limited designation of a typical file format, carving its niche as an avant-garde scene description protocol. It diligently preserves the elaborate semantics inherent to a scene graph, facilitating a granular and sophisticated data representation. Contrary to FBX and ABC, USD's stratified methodology for scene assembly bestows unparalleled flexibility in data structuring, modification, and integration. Its software-agnostic design ethos, envisioned to operate as a ubiquitous interchange medium, attenuates the complexities and potential data degradation typically associated with inter-software scene migrations. While the expansive scope of USD's competencies coupled with the necessity for pipeline overhauls might present initial hurdles, its indisputable merits are progressively garnering attention in the animation and VFX sectors. As studios increasingly resonate with the promise of robust, scalable, and efficient workflows that USD proffers, it is evident that a revolutionary epoch in the animation and VFX domain's data interchange arena is on the horizon.

2.3 Tool Selection

2.3.1 Maya API

Autodesk's Maya Application Programming Interface (API) consists of an ensemble of C++ classes, granting developers deep-seated access to Maya's foundational data structures and utilities. This intricate access propels the development of bespoke tools, plugins, and enhancements to extant functionalities. The utility continuum of the

Maya API ranges from streamlining repetitive tasks to intricate bridging with external software and databases. By providing profound control and versatility, the Maya API augments Maya's inherent capabilities, introducing innovative geometric frameworks, shader configurations, and animation controllers, while also refining traditional workflows(Schierenbeck, 2022).

Outside the ambit of the Maya API, the Maya Script Editor presents developers with a complementary vector for enhancing Maya's potential. This apparatus supports scripting languages, notably the Maya Embedded Language (MEL) and Python, facilitating task automation, crafting tailored user interfaces, and rapid tool prototyping. With their comparatively straightforward syntax, especially when set against the C++ constructs of the Maya API, these scripting languages often become the go-to for artists aiming to finetune their creative pipelines..

However, it's imperative to recognize the limitations of scripting. While undeniably versatile, scripts may fall short of the performance prowess exhibited by C++ plugins, particularly when navigating data-intensive operations or managing expansive datasets. Scripting paradigms may occasionally find themselves at odds with the realization of certain complex functionalities and custom nodes, a consequence of their intrinsic limitations. In contrast, the Maya API delves deeper into Maya's intricate architecture, paving the way for the creation of custom nodes, trailblazing geometric articulations, and other advanced enhancements that surpass the purview of the Script Editor. Furthermore, plugins developed via the Maya API frequently eclipse scripts in terms of performance efficiency—a quintessential trait in high-stakes production environments where promptness and optimal resource utilization reign supreme.

2.3.2 USD API

The Universal Scene Description (USD) API, an innovative brainchild of Pixar, serves as a multifaceted Application Programming Interface (API) dedicated to facilitating interactions with and manipulations of USD data—a robust and scalable framework for exchanging diverse 3D scenes and associated data. Encompassing a suite of libraries coupled with Python bindings, the USD API streamlines the processes of creating, amending, and probing USD files. This encompasses tasks such as introducing, modifying, or excising objects, attributes, and affiliations within a USD scene. Through this API, software developers gain the leeway to embed USD functionalities within their proprietary tools, including but not limited to 3D modeling applications, animation platforms, and rendering engines. Concurrently, the API underpins a sturdy infrastructure adept at deciphering intricate hierarchical scene data, orchestrating concurrent animations and variances, interfacing seamlessly with a plethora of rendering solutions, and proficiently administering voluminous data sets.

2.3.3 Houdini HDK

The Houdini Development Kit (HDK) embodies a comprehensive suite of Application Programming Interfaces (APIs), primarily underpinned by C++, empowering developers to extend Houdini's functionality through interfaces with its core systems. With this control over Houdini's capabilities, the HDK fosters the creation and modification of parameters, nodes, and geometry, among others. This flexibility affords developers the freedom to conceive custom nodes, automate tasks via scripting, engineer plugins for augmenting functionality, and even refine Houdini's existing operational protocols, thus enabling software customization to align with specific project requirements. The HDK's depth and breadth demand a significant level of technical proficiency, making it a preferred instrument for veteran developers or technical directors in a production milieu.

Custom Surface Operators (SOPs), pivotal to Houdini's procedural philosophy, are a prime means of extending its functionality through the HDK. SOPs are responsible for the conception and manipulation of geometry within the 3D space. The development of custom SOPs involves the creation of a new node, establishing its parameters, and governing behaviors. The HDK allows developers to architect the node's user interface, specify its inputs and outputs, and dictate its execution process. For instance, a developer might employ a custom SOP to generate a distinct procedural pattern not included in Houdini's inherent tools or to automate a sophisticated series of operations consistently utilized within a pipeline.

While there are alternative methods to expand Houdini's functions, each comes with its advantages and constraints. Houdini supports Python scripting throughout the software, making it a flexible choice for automating

tasks, crafting user interfaces, or generating geometry. However, in comparison to HDK, Python may exhibit slower performance, a potential limitation when handling intricate and resource-heavy 3D data. In addition to Python, Houdini supports Vector Expression (VEX), a high-performance language used extensively, including shader writing(Piriya and Mahendran, 2014). VEX, while user-friendly and ideal for tasks such as manipulating geometry, does not offer the same level of control or versatility as HDK in creating custom nodes or tools. HDK, being a potent and low-level interface to Houdini, facilitates the creation of deeply integrated custom nodes and tools. Built on C++, HDK offers performance advantages over Python and deeper access to Houdini's inner mechanisms than VEX. Hence, despite existing alternatives, HDK stands as a leading choice for extensive plugin development in Houdini owing to its superior control and efficiency(Schierenbeck, 2022).

3 Implementation

3.1 Design Rationale

3.1.1 Mesh Information Design

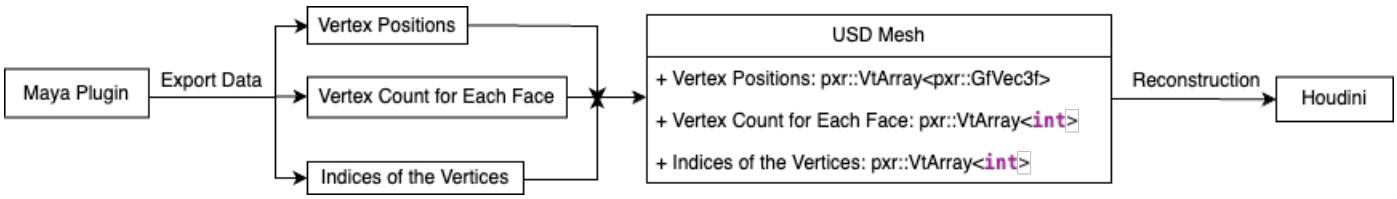

Figure 1: A visual flowchart illustrating the design process of mesh data conversion

- Vertex Positions: In the realm of 3D graphics, Vertex Positions, characterized by their three-dimensional coordinates (x, y, z), serve as an elemental foundation in delineating the form and architecture of a mesh. Their significance transcends mere spatial representation, profoundly influencing the mesh's rendering behavior and its intricate interactions with various scene facets such as lighting and shadows.
- Vertex Count for Each Face: This list tells us how many vertices make up each face in a mesh. Triangles have 3 vertices, quads have 4. Sometimes faces have more than 4 vertices, called n-gons. Keeping track of these counts is essential for accurately reconstructing the mesh in software.
- Indices of the Vertices: This structured array defines the order of connecting vertices to create faces. It's like a roadmap for constructing polygons. This indexed approach is efficient for storing complex meshes. If a vertex is shared by multiple polygons, its position is saved only once, reducing redundancy and improving processing speed. This indexing is crucial for optimizing 3D mesh storage and computation.

3.1.2 Skeleton Information Design

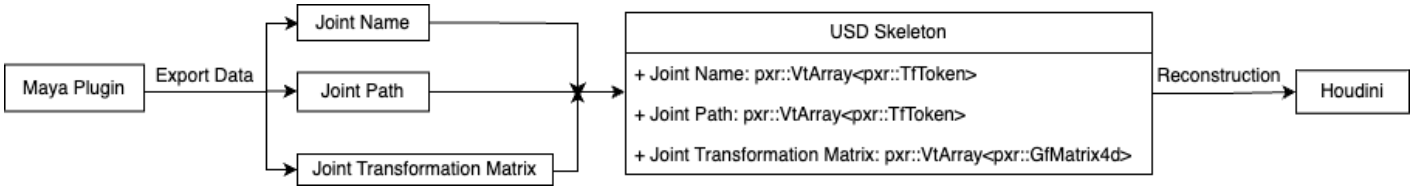

Figure 2: A visual flowchart illustrating the design process of skeleton data conversion

- Joint Name: Every joint (or bone) in a rig typically has a unique name, which allows it to be uniquely identified. This is essential for scripting, animation, and most importantly, for ensuring that data is matched correctly when transferred between software.

- Joint Path: In skeletal animation and deformation systems, the hierarchical structure of joints plays an indispensable role. A joint's full path elucidates its specific position within this hierarchical framework. This detail is not just informational; it carries significant functional implications. Take, for instance, the hierarchy represented as "arm/forearm/wrist/hand". In this structure, each subsequent joint is a child of the former. Such a hierarchical delineation ensures that transformations applied to a higher-level joint, such as the "arm", are accurately propagated down the chain, influencing all its descendant joints. In the given example, a movement in the arm joint would have a cascading effect, ultimately reaching and affecting the "hand". Thus, by preserving and understanding the full path hierarchy, one can ensure the integrity of deformations and animations, maintaining the intended behaviors and relationships established in the original rigging process.

- Joint Transformation Matrix: In 3D graphics, particularly within software platforms like Autodesk Maya, the transformation matrix of a joint encapsulates its position, rotation, and scale in relation to its parent joint. When engaging in data transfer operations, it's of paramount importance to ensure that the matrix is preserved to retain the joint's spatial orientation and behavior accurately. Directly conveying this matrix is often more computationally efficient and offers greater accuracy compared to decomposing and transmitting its constituent transformation components—namely, translation, rotation, and scale—individually. Maya delineates this combined transformation using the formula:

$$matrix = [S] \cdot [RO] \cdot [R] \cdot [JO] \cdot [IS] \cdot [T]$$

  [T]: The translation component positions the joint in the 3D space relative to its parent. This operation is foundational because it sets the initial spatial coordinates for the joint. Translation moves the joint without altering its orientation or scale. [IS]: This transformation inversely scales the joint based on its parent's scale, ensuring that any parent scaling doesn't undesirably distort the child joint. It effectively compensates for the parent joint's scaling, allowing child joints to maintain their intended size even if the parent joint has been scaled. [JO]: The jointOrient attribute specifies the joint's default orientation. This is crucial for defining the initial rotational alignment of the joint, especially important for things like limb orientation in a character rig. It establishes the joint's local rotation axes before any animated rotations are applied. [R]: The rotation component imparts the joint's animated rotation. When animators keyframe a joint's rotation in Maya, they're essentially manipulating this component. It represents the dynamic aspect of the joint's orientation, which changes over time based on the animation. [RO]: RotateOrient, also known as RotateAxis, serves as a pre-rotation adjustment to the joint's orientation. It enables riggers and animators to adjust the joint's rotational axes without affecting the actual rotation animation. It's like a foundational tweak to the joint's orientation, upon which other rotations are layered. [S]: The scale transformation resizes the joint along its local axes. It's typically used less frequently for joints than translation or rotation but can be crucial for certain rigging scenarios, especially for non-uniform scaling effects.

3.2 Development of Maya Plugins

3.2.1 Establishing the Plugin Framework with Maya API

In the initial stages of the plugin development process, an intricate C++ environment was methodically configured. This pivotal configuration is not a mere procedural step, but rather the linchpin that ensures an immaculate interface with Maya's multifaceted Application Programming Interface (API). In this step, it's noteworthy to mention that every indispensable USD library, situated within Maya's native USD directory, is explicitly linked in the "CMakeLists.txt" file. Culminating the configuration phase, the directive was established to produce an output of the ".bundle" format, rooted in macOS's operational framework. These bundles act as a comprehensive repository that encapsulates the compiled plugin code while being augmented with critical resources and metadata. This sophisticated packaging mechanism not only optimizes the distribution of plugins but also augments Maya's capabilities to facilitate efficient interfacing and manipulation of plugin functions.

In the codebase's structure, both Maya and USD should be included in the headers. The epicenter of this structural framework resides a bespoke command, explicitly for the confines of Maya's MEL scripting milieu. Furthermore, every plugin's operational logic is housed within the 'doIt' function, thereby epitomizing a paradigm of focused and streamlined coding strategies.

In the subsequent compilation phase, Visual Studio Code is utilized, converting the project into its designated .bundle format, which will be used to load into Maya. The .bundle file is then assimilated, thereby introducing a broad range of new capabilities. Within this plugin setting in Maya, a specific MEL command is executed, serving to activate the plugin. This process epitomizes a technical endeavor, where mesh data is systematically retrieved and subsequently encoded in the USD format.

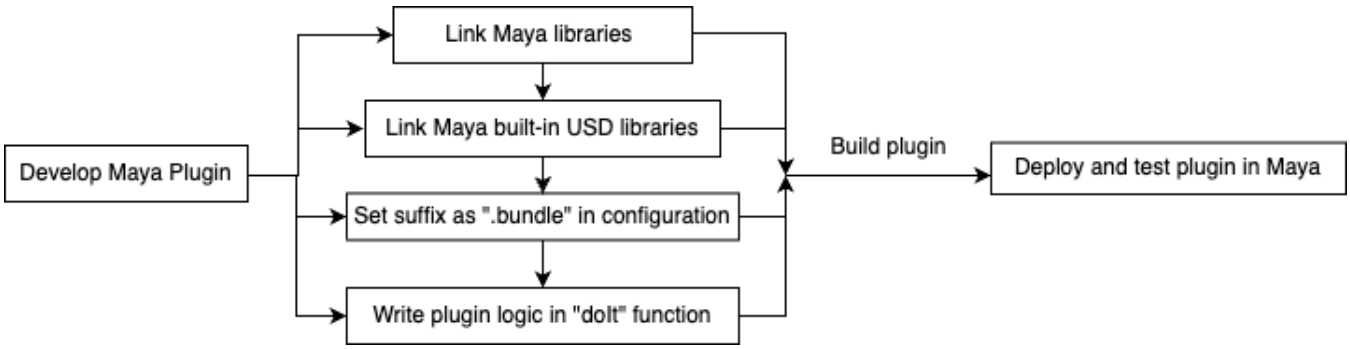

Figure 3: A visual flowchart illustrating the design process of the Maya plugin

3.2.2 Mesh Plugin Development

This plugin allows the user to select a mesh in Maya and use the getMeshInfo command to export the selected mesh's data to a USD format file. The exported USD file will contain the mesh's vertex positions, vertex count for each face, and indices of the vertices that form each face.

Algorithm 1 Get Mesh Information and Export to USD

```
1:  procedure GETMESHINFO
2:      Get the active selection list from Maya
3:      MDagPath dagPat = first selected object
4:      while polygon do
5:          Store vertex positions
6:          Store face vertex count
7:          Store face vertex indices
8:      end while
9:      Save to USD stage
10: end procedure
```

Giving an overview of the steps in Algorithm 1:

- Line 2-3: Upon initiating the data extraction process from Maya, the algorithm consolidated all selected meshes into the "MSelectionList". This selection list then facilitated the utilization of "MDagPath" to ascertain the Directed Acyclic Graph (DAG) structure of all the chosen meshes. In Maya's intricate scene architecture, the DAG serves as a pivotal data structure, encapsulating a plethora of scene entities including, but not limited to, transformations, shapes, lights, and cameras. This structure, by its design, ensures a unique identification mechanism for a node, predicated upon its hierarchical positioning within the DAG.

  However, it's crucial to note that "MDagPath" does not intrinsically contain the node's data. Instead, it proffers a pathway to both access the node in question and retrieves its corresponding data. To capitalize on this pathway and garner intricate mesh details such as points and vertex information, the algorithm employed the

"MFnMesh" function set, attaching it to the aforementioned "MDagPath". This strategy ensured a systematic and efficient extraction of the desired mesh attributes.

- Line 4-8: Before the execution of the main loop, the algorithm initially records the points' positions, face vertex counts, and face vertex indices, storing them in "MPoint", "int", and "MIntArray" respectively. This preparatory step is essential to enable the subsequent conversion of Maya's native format to a format compatible with the Universal Scene Description (USD) API.

  The loop is structured to iterate over each polygon in the model. Within this loop, The position of each point is recalibrated using the "pxr::GfVec3f" data structure to ensure compatibility with USD's expected format. The face vertex count and face vertex indices are meticulously stored using "pxr::VtArray", facilitating an organized representation of this data for future processing within the USD API.

3.2.3 Skeleton Plugin Development

The Maya skeleton plugin, developed as a key part of this project, leverages the C++ Maya API to offer a unique capability. This code represents a Maya plugin called "getSkeletonInfo" that exports skeleton data from selected joints in Maya to a USD file. The plugin defines a "Skeleton" structure to store joint names, paths, and transforms. It uses recursive functions to traverse the joint hierarchy, extracting joint information and transforming it into USD-compatible data structures. Finally, the plugin creates a new USD stage, defines a skeleton in USD, and exports the joint names, paths, and transforms to the USD file. The resulting USD file can be used in other applications that support USD format, such as Houdini, for further manipulation.

Here is the pseudocode of the "visitJointHierarchy" function recursively traverses a Maya joint hierarchy, extracting the joint's name, transformation matrix, and hierarchical path, and stores this information in a provided "Skeleton" data structure.

Algorithm 2 Visit Joint Hierarchy

```
1:  procedure VISITJOINTHIERARCHY(jointObject, skeleton, depth=0, parentName="")
2:
3:      status ← CREATESTATUS
4:      jointFn ← CREATEJOINTFUNCTIONSET(jointObject, status)
5:      if not status then
6:          DISPLAYERROR("MFnIkJoint constructor")
7:          return
8:      end if
9:
10:     skeleton.jointTransforms ← jointFn.TRANSFORMATIONMATRIX
11:     skeleton.jointNames ← jointFn.NAME
12:     skeleton.jointPath ← jointFn.CURRENTJOINTPATH
13:
14:     for i from 0 to jointFn.CHILDCOUNT - 1 do
15:         childObject ← jointFn.CHILD(i, status)
16:         if not status then
17:             DISPLAYERROR("child")
18:             continue
19:         end if
20:         if childObject.HASFUNCTION(MFn::kJoint) then
21:             VISITJOINTHIERARCHY(childObject, skeleton, depth + 1, currentJointName)
22:         end if
23:     end for
24:
25: end procedure
```

Giving an overview of the steps in Algorithm 2:

- Line 3-8: Within Maya's C++ API, "MStatus" is a class specifically designed to capture and convey the status of operations. It provides a way to understand whether an API call was successful or if it encountered issues. "MFnIkJoint" is a function set in the Maya API tailored to handle and manipulate IK joints. Function sets in Maya are essentially classes that wrap around specific types of objects in Maya, giving you a set of methods to query and modify those objects. This code is about creating a functional interface ("MFnIkJoint") for a given Maya object ("jointObject"), and ensuring that this creation process was successful before proceeding.

- Line 10-12: This is designed to extract vital transformational information from a Maya IK joint ("jointFn") and store it in a custom-defined structure named skeleton. This process is essential for translating the intricacies of Maya's joint attributes into a more streamlined and custom format, suitable for specific applications or further processing. Specifically, it retrieves rotation, translation, scale orientation, and overall orientation from the Maya joint. Moreover, it captures the joint's transformation matrix, its name, and its hierarchical relationship within the skeleton. By systematically extracting these attributes and pushing them to the skeleton structure, ensuring that essential joint data is captured in a format that's both intuitive and amenable to further manipulation outside of the Maya environment.

- Line 14-23: This is the main loop of this algorithm, visitJointHierarchy, designed to navigate through the intricate joint hierarchy of an animated skeleton or rig. By leveraging the Maya API, the function iterates over each joint's children, extracting relevant information such as joint transforms and relationships. Central to its efficiency and simplicity is the application of a recursive algorithm. Recursion, in this scenario, offers an elegant and straightforward solution for depth-first traversal, ensuring that the code delves as deep as possible into one branch of the hierarchy before backtracking and progressing to the next. This recursive approach not only reduces code complexity compared to other iterative methods but also dynamically accommodates varying depths of joint hierarchies. Moreover, the inherent isolation of states in each recursive call makes the code both more readable and less susceptible to errors. Thus, the adoption of recursion in this function exemplifies a method that is both effective and naturally suited to handle Maya's hierarchical joint systems

Algorithm 3 Process Maya Skeleton and Export to USD

```
1: procedure MAIN
2:
3:     initialize an empty selection list named selectionList
4:     fetch all active selections from Maya and add to selectionList
5:     initialize an empty skeleton structure named skeleton
6:
7:     for each object in selectionList do
8:         try to get the dependency node of object
9:         if error occurred then
10:            display error message "getDependNode"
11:            continue to next object
12:        end if
13:        if object is a joint then
14:            call visitJointHierarchy with object and skeleton
15:        end if
16:    end for
17:
18:    call exportUSD with skeleton
19:    return success
20: end procedure
```

Giving an overview of the steps in Algorithm 3:

- Line 7-16: Within the loop that iterates over the "selectionList", each item's dependency node is fetched using "getDependNode" and stored in the "object" variable; if there's a failure in retrieval, an error message is displayed, and if the object is identified as a Maya joint ("MFn::kJoint"), its hierarchy is processed using the "visitJointHierarchy" function which mentioned above to populate the "skeleton" object with the joint's data.
- Line 18: The "exportUSD" function takes a skeleton's joint data, converts it to a format compatible with Universal Scene Description (USD), and then saves it as a USD file at a specified location. It initializes a new USD stage, translates a given skeleton's joint names, paths, and transforms data into USD-compatible formats, defines these details within a USD skeleton structure, and finally saves this structured data into a ".usda" file at a designated path.

3.3 Development of Houdini Plugins

3.3.1 Establishing the Plugin Framework with Houdini HDK

In the development of a Houdini plugin leveraging the HDK, a comprehensive CMake configuration has been set up to ensure a seamless build and integration process. Essential paths, pertaining to Houdini's toolkit, libraries, and built-in USD libraries, are meticulously outlined, all directed towards the Houdini application path. As an enhancement, prototype header generation is based on an embedded DS file, which also acts as the parameter interface specifier, ensuring precise interfacing. A shared library has been defined, with rigorous linking to Houdini libraries and other essential dependencies, such as Python and an array of USD-related dynamic libraries. The culmination of this configuration guarantees that the library output properties resonate perfectly with Houdini's stipulations, making the integration process straightforward.

In the architectural design of the codebase, it's imperative that both HDK and USD are referenced within the header files. All logic pertaining to the plugin should reside within the "cook" function. Post the build process, which utilizes both cmake and xcode, the "hcustom" utility – a proprietary command-line tool from SideFX – is employed to generate the requisite library link. This utility serves the purpose of compiling custom plugins and extensions tailored for Houdini, effectively converting external C++ code into dynamic libraries. These libraries can subsequently be integrated into Houdini as custom nodes, operators, or other forms of extensions. Once executed, testing the functionality within Houdini's SOP environment is straightforward; the custom SOP becomes readily available in the SOP list for utilization.

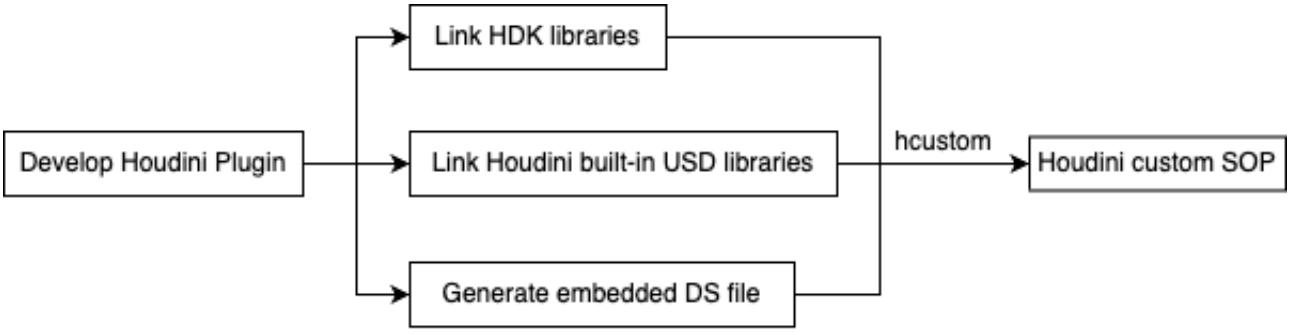

Figure 4: A visual flowchart illustrating the design process of the Houdini plugin

3.3.2 Mesh Plugin Development

The Houdini mesh plugin, developed as an integral component of this project, represents an innovative application of the C++ Houdini Development Kit (HDK). This tailored plugin is specifically engineered to import mesh data from a USD file that has been exported using the bespoke Maya mesh USD plugin. The Houdini mesh plugin efficiently interprets the mesh data from the USD file, transforming it into compatible Houdini geometry. It then skillfully generates corresponding points and polygons within the Houdini scene.

Algorithm 4 SOP_importUSD Process

```
1:
2: procedure newSopOperator(table)
3:     Create new operator for table with details like name, constructor, and templates
4: end procedure
5: Define parameter interface for the SOP as theDsFile
6: procedure SOP_importUSD::buildTemplates
7:     Build templates using theDsFile
8: end procedure
9:
10: procedure SOP_importUSDVerb::cook(cookparms)
11:     filePath ← get USD file path from cookparms
12:     if filePath is valid then
13:         Open the USD file using USD API
14:         Retrieve mesh from the opened file
15:         Get mesh attributes: points, vertexCounts, vertexIndices
16:         if number of points in current detail ≠ retrieved points then
17:             Clear and destroy current detail
18:             Create points in detail using retrieved positions
19:             Test if points are stored correctly
20:             Store points in Houdini, linking vertices to points based on indices
21:         end if
22:     end if
23: end procedure
```

Giving an overview of the steps in Algorithm 4:

- Line 2-8: The "newSopOperator" function serves to register a new SOP node within Houdini's internal framework. Upon the loading of a plugin, Houdini anticipates this function to relay information regarding any fresh nodes or tools introduced by the plugin. Subsequently, "theDsFile", a raw string literal in C++, delineates the parameter interface for this specific SOP. In essence, this is a Domain Specific Language (DSL) segment, laying out the design and functionality of your SOP's user interface. For this particular plugin, the user interface is succinct, merely comprising a label and a file selector to import the USD file.

  Transitioning to the "buildTemplates" function, it acts as a converter, transforming the content of "theDsFile" into a structure that Houdini can interpret for the construction of the node's user interface. Within this function, "PRM-TemplateBuilder" employs the DSL to generate an array of "PRM-Template" objects. Each object in this array elucidates specific details about a parameter, including its type, default values, and other pertinent metadata. To encapsulate, these code segments collectively orchestrate the creation of a bespoke Houdini node, endowed with a distinct user interface, and subsequently introduce this new node to Houdini.

- Line 10-23: The "cook" method serves as the cornerstone of a Houdini SOP (Surface Operator). This method delineates the procedure through which the node processes and subsequently outputs data. Within this method, the primary structure utilized is "GU-Detail". "GU-Detail" constitutes a fundamental data structure within Houdini's framework, specifically tailored to represent geometry. It is an integral component of Houdini's Geometry Utility Library. In the context of this plugin, "GU-Detail" is employed to facilitate the conversion of data.

  In the execution of this procedure, the algorithm proceeds to iterate over each face in the geometry. Utilizing the method "appendPrimitivesAndVertices", the algorithm dictates the manner in which polygons are integrated into the structure. The "GA-Offset" structure, meanwhile, serves a critical role, maintaining a record of the locations where vertices are incorporated within the Houdini geometry detail. Here is the pseudocode for how data be stored in Houdini structure.

Algorithm 5 Storing USD mesh data in Houdini

```
Initialize empty start_vtxoff                                  ▷ Starting offset for vertices
for each count in vertexCounts do
    Add a new polygon to detail with count vertices
    for j = 0 to count - 1 do
        vtxOff ← start_vtxoff + j
        ptOff ← vertexIndices[vtxOff]
        Associate vertex vtxOff with point ptOff
    end for
end for
```

3.3.3 Skeleton Plugin Development

The Houdini skeleton plugin, developed as an integral component of this project, represents an innovative application of the C++ Houdini Development Kit (HDK). This tailored plugin is specifically engineered to import skeleton data from a USD file that has been exported using the bespoke Maya skeleton USD plugin. It opens the USD file and retrieves the skeleton's data, including bind transforms, joint names, and joint paths. It then processes this data and creates bone objects in the Houdini scene based on the skeleton information. The bone objects are represented as polygons connecting joints and are positioned using the computed world transforms for each joint. Giving an

Algorithm 6 Processing a USD Skeleton in Houdini

```
1:
2: procedure NewSopOperator(table)
3:     Create new operator for table with details like name, constructor, and templates
4: end procedure
5: Define parameter interface for the SOP as theDsFile
6:
7: procedure Cook(cookparms)
8:     sopparms ← parameters from cookparms Get USD file path: sopparms.getUsdfile()
9:     objname_stdString ← sopparms.getUsdfile()
10:    if objname_stdString ≠ "0" then
11:        Open USD file using USD API
12:        Get the Skeleton prim from the opened stage
13:        Retrieve attributes: bindTransforms, jointNames, joints
14:        for each joint in skeleton do
15:            Get the transformation matrix, name and path of the joint
16:            Compute parent of the joint
17:        end for
18:        Test Houdini data
19:        for each joint do
20:            Compute world position and transform of the joint
21:            Add attributes to Houdini geometry: rest position, orientation, name
22:            If the joint has a parent, create a bone primitive connecting them
23:        end for
24:    end if
25: end procedure
26:
```

overview of the steps in Algorithm 6:

- Line 7-25 The "cook" method is the core function responsible for integrating USD skeletal data into Houdini. Beginning with the extraction of skeletal information from the USD file, the method retrieves vital attributes such as "bindTransforms", "jointNames", and "jointPaths". A crucial differentiation in this implementation from a typical Houdini Mesh Plugin lies in its skeletal data manipulation.

  To achieve a representation in Houdini, it is designed to calculate the world position for each joint and store them in an array. Once these world positions are established, the joints are systematically iterated over. For

every joint, the method visualizes its skeletal connections using the "GU-PrimPoly", a primitive type in Houdini that allows the representation of bones as line segments. These line segments, defined by two vertex points, represent bones by connecting a joint to its parent joint based on their world positions. This approach not only ensures the visual representation of the skeletal hierarchy but also seamlessly integrates the skeletal structure into Houdini's geometry space.

Here is the detailed pseudocode about how this plugin manipulates skeleton data and visualizes them in Houdini.

Algorithm 7 Bone SOP Node Creation

```
Initialize:
    a new GU_Detail called 'gdp'
    vectors for storing joint-related data
if P and xform attributes exist in 'gdp' then

    for each joint in 'jointNames' do
        Extract rotation matrix from 'bindTransform'
        Compute the transpose of the rotation matrix
        Extract local translation from 'bindTransform'
        if this joint is the root then
            Compute world transform for root joint
            Extract translation from the world transform
            Store translation as world position for root joint
        else
            Determine the parent joint's index
            Compute concatenated rotation matrix for this joint
            Compute final world position for this joint
            Store computed world position for this joint
        end if
    end for

    for each joint in 'jointNames' do
        Determine parent joint's index
        Create a new point in 'gdp' and set its position
        if the joint has a parent then
            Create a new bone primitive linking the joint and its parent
        end if
        Set the bind transform for the joint point
    end for

end if
```

4 Results

The primary goal of the project was to ensure a seamless and accurate transfer of mesh and skeleton data from Maya to Houdini. Achieving this not only entails a successful transfer but also ensures data integrity and the preservation of fine details inherent to the 3D modeling and animation process.

4.1 Output Details: Maya into USD

Upon initiating the transfer process via the developed plugins, the Maya meshes and skeletons were successfully extracted and converted to the USD format. This USD formatted data, when opened directly in a USD viewer, retained its structural integrity, showing that the conversion process in the Maya end was devoid of data loss.

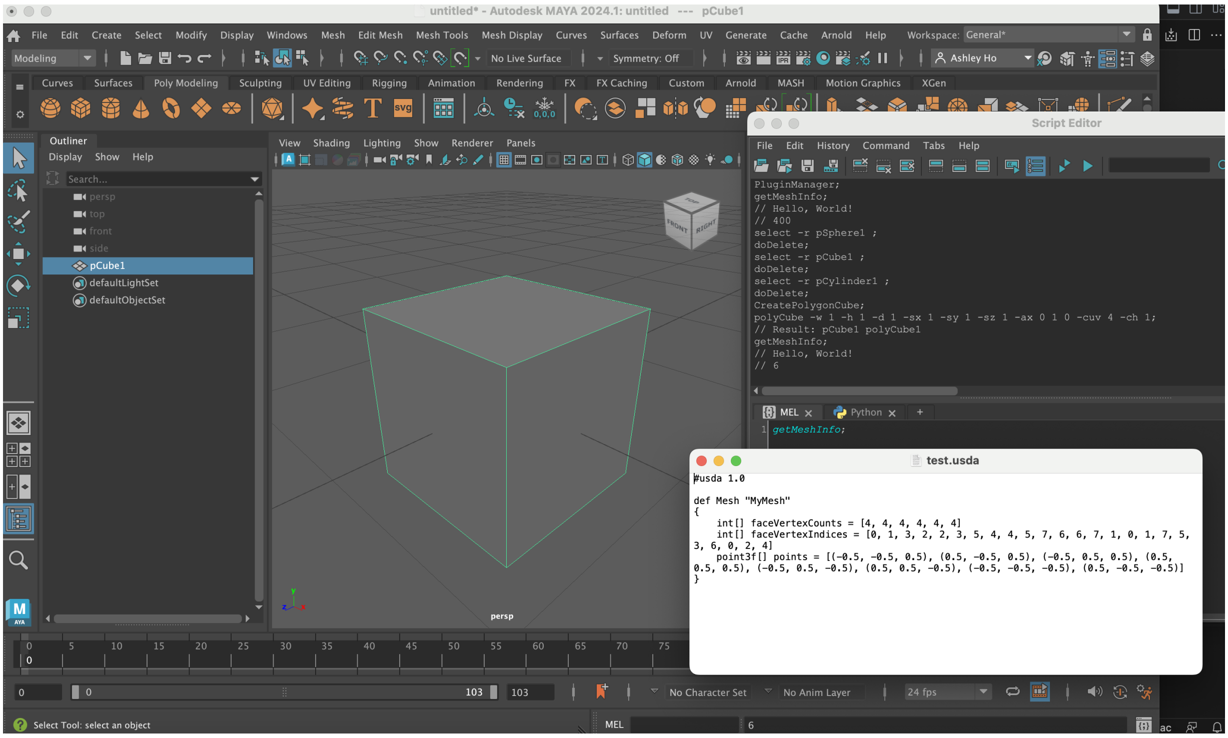

Figure 5: Test basic polygon cube (left) and after converting USD file (right)

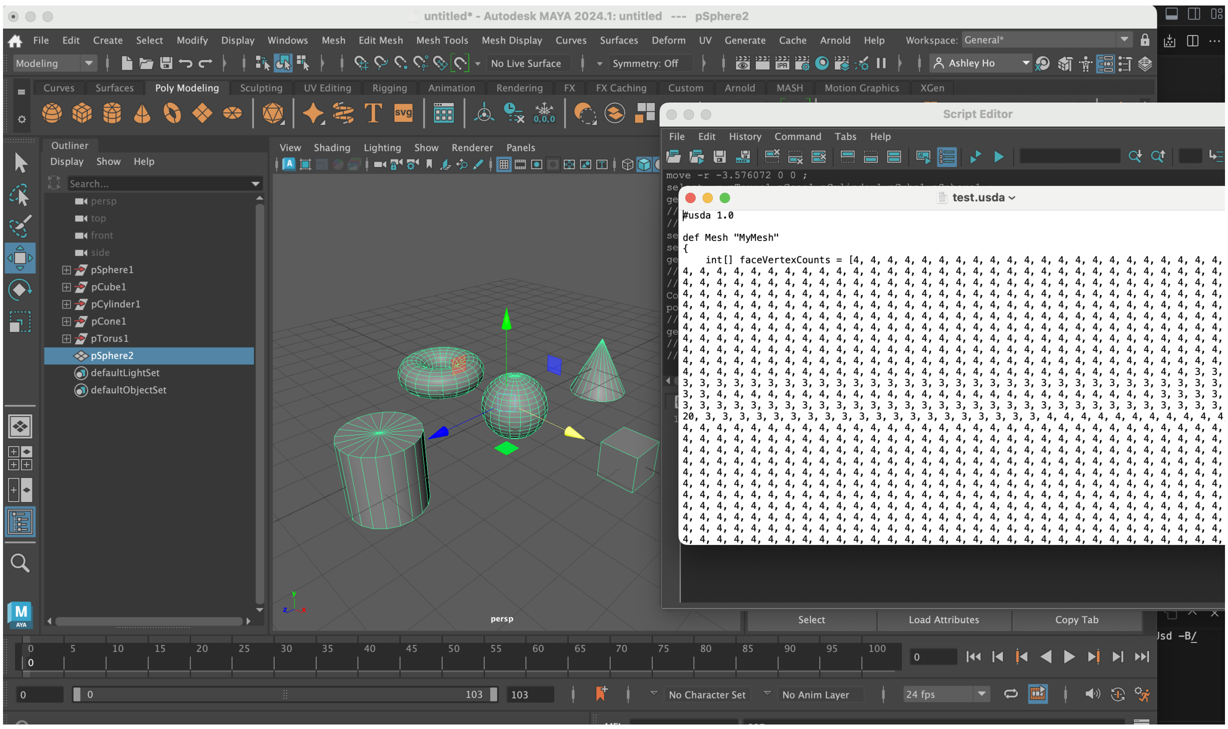

Figure 6: Test multiple polygons (left) and after converting USD file (right)

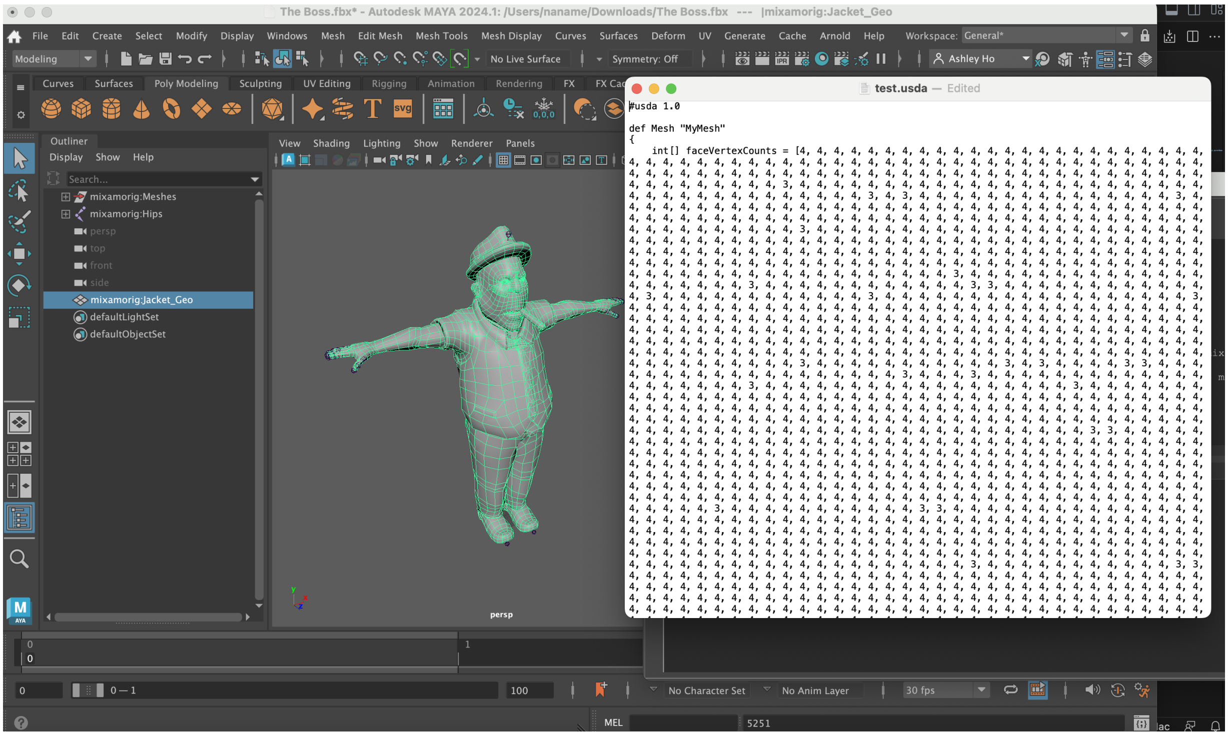

Figure 7: Test complex polygon (left) and after converting USD file (right)

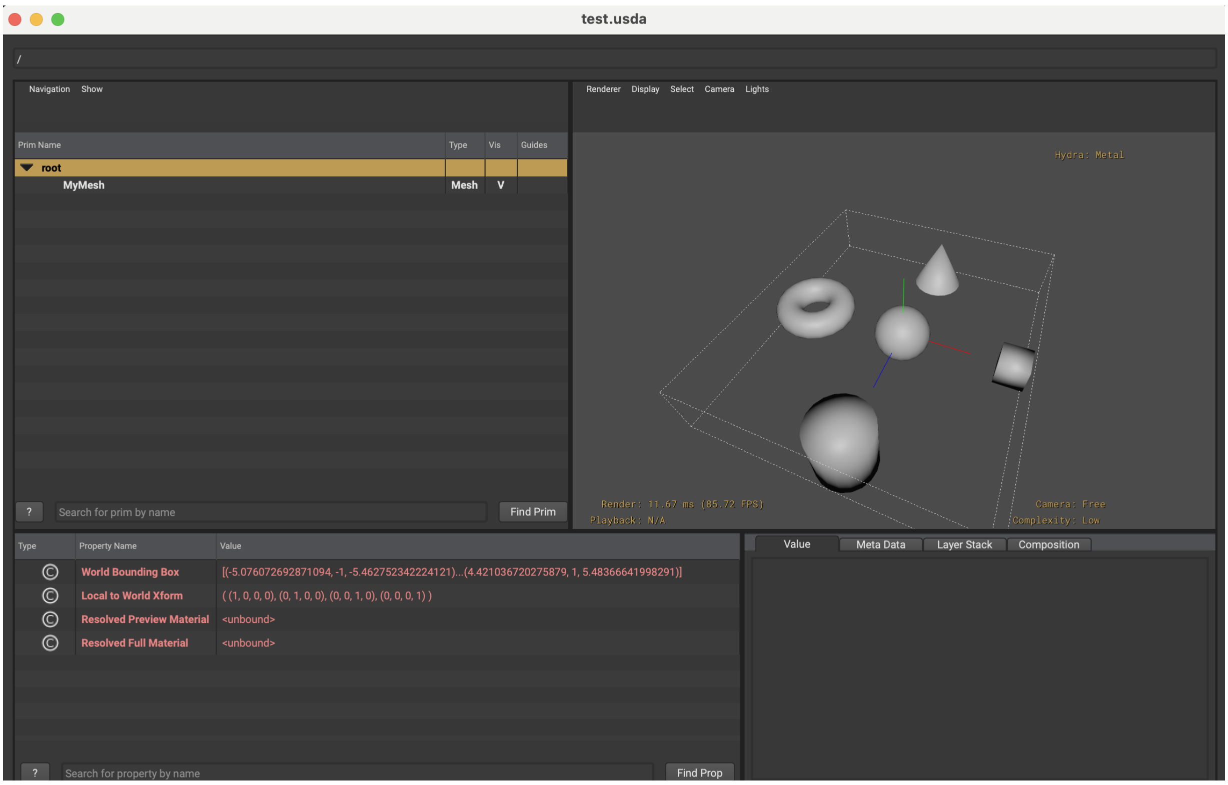

Figure 8: USD view with multiple polygons

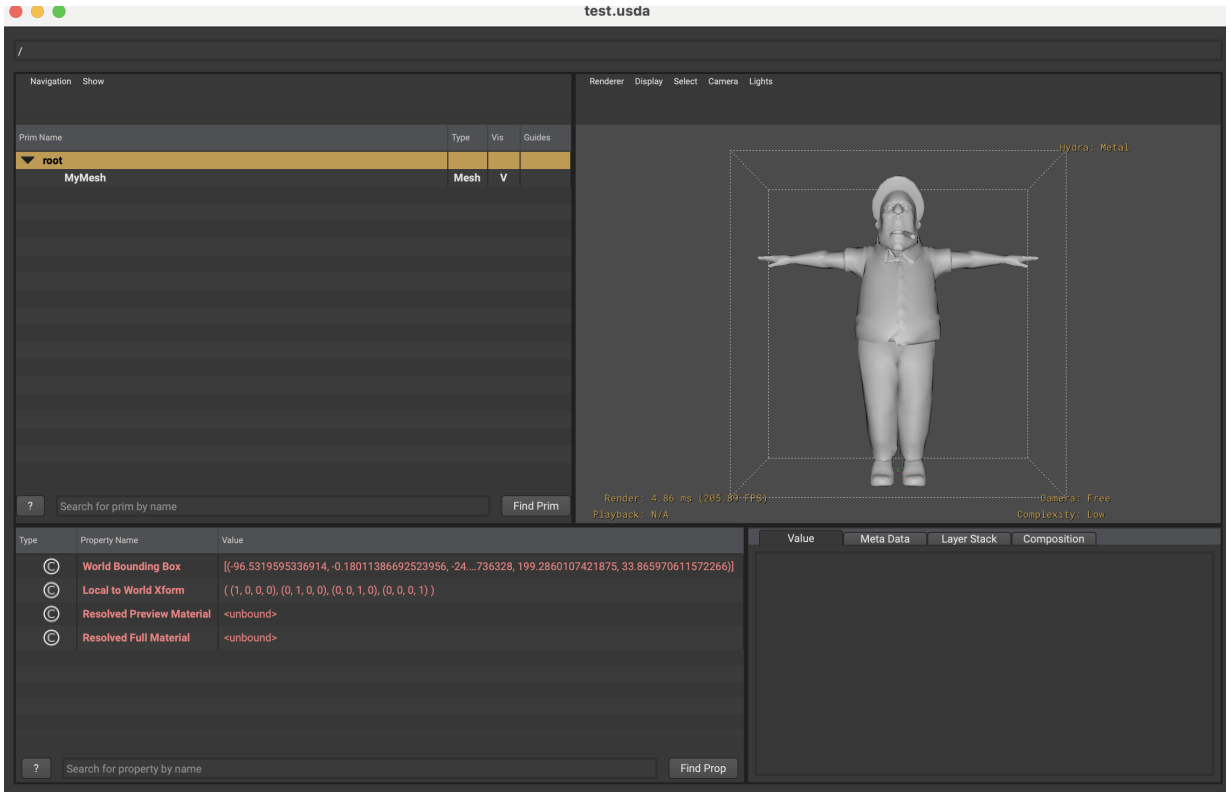

Figure 9: USD view with complex polygon

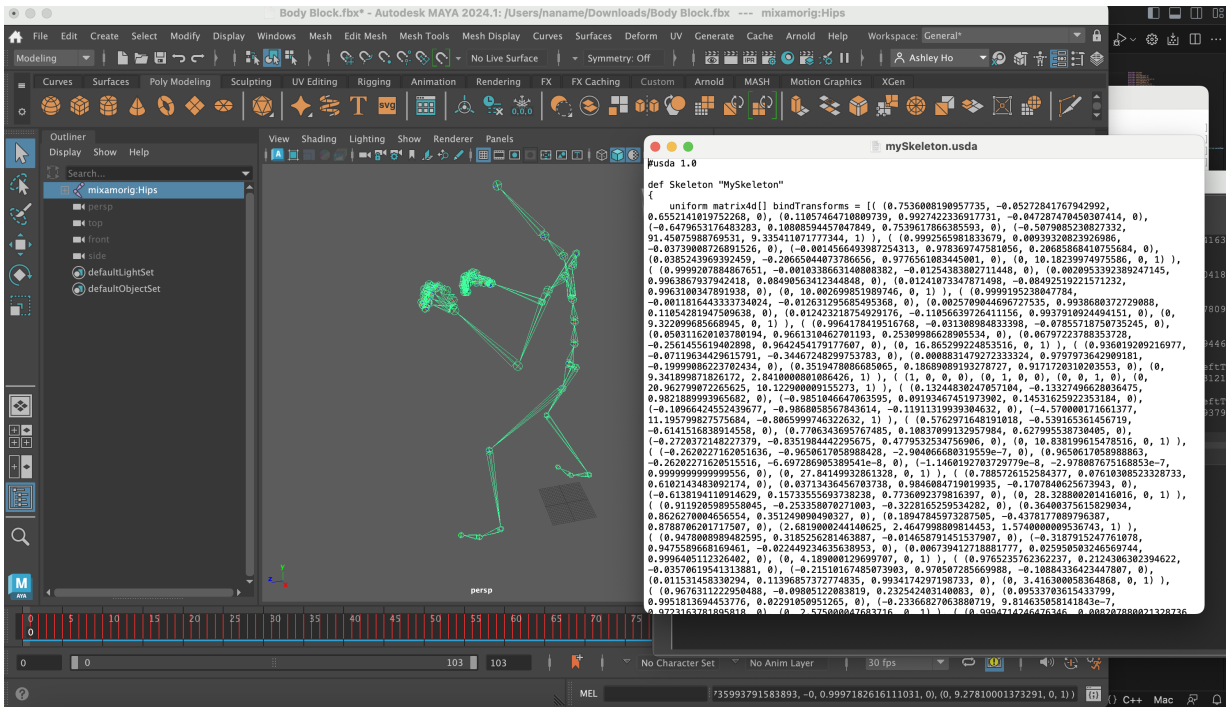

Figure 10: Test complex skeleton (left) and after converting USD file (right)

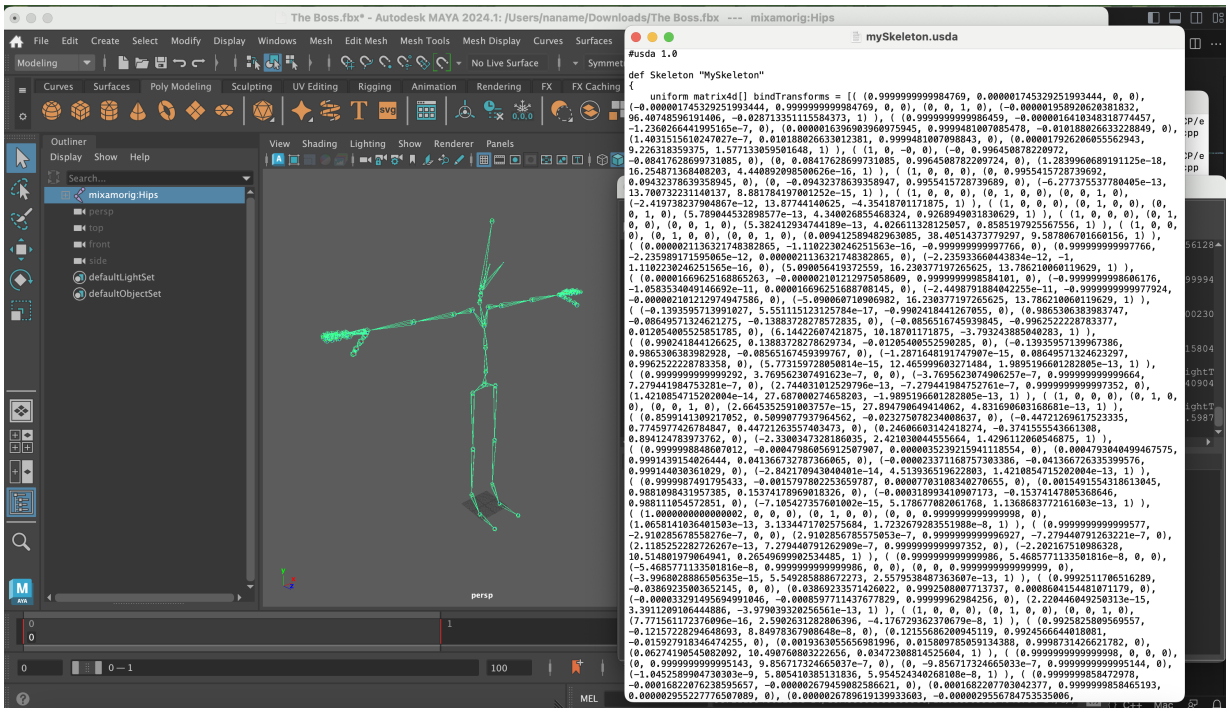

Figure 11: Test complex skeleton (left) and after converting USD file (right)

4.2 Output Details: USD into Houdini

On the Houdini side, upon activating the developed plugins, the USD data was read and translated back into a format Houdini recognizes. The resulting visualization in the Houdini interface was near-identical to the original Maya setup. Both the mesh structures, as well as the intricate skeleton data – which often includes joints, bones, and influence weights – were faithfully reproduced.

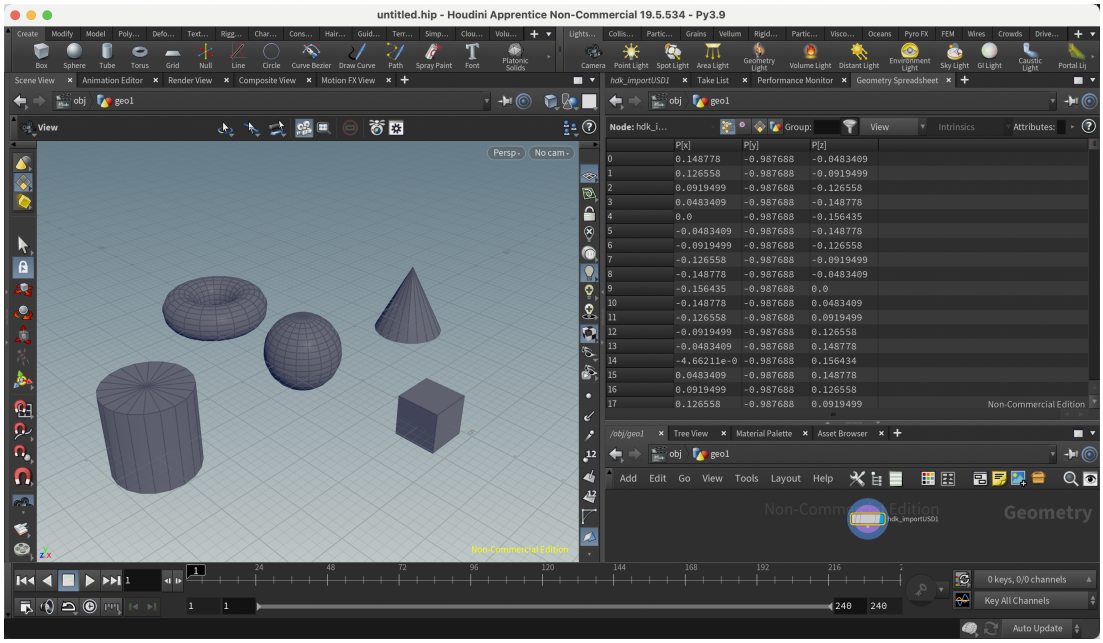

Figure 12: Test multiple polygons shown in Houdini interface using USD import plugin

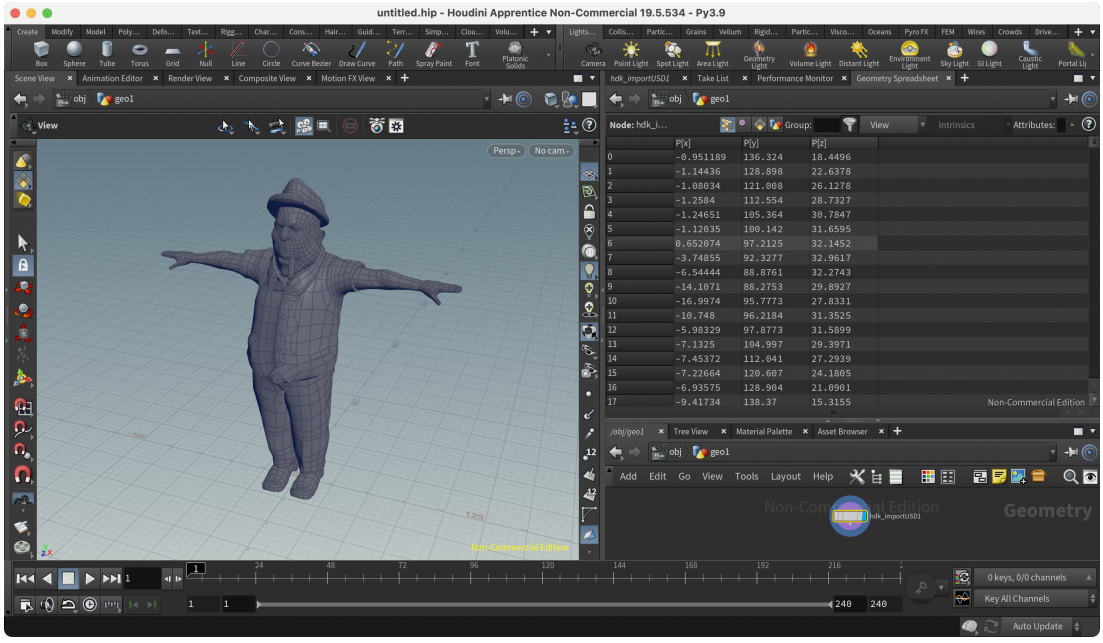

Figure 13: Test complex polygons shown in Houdini interface using USD import plugin

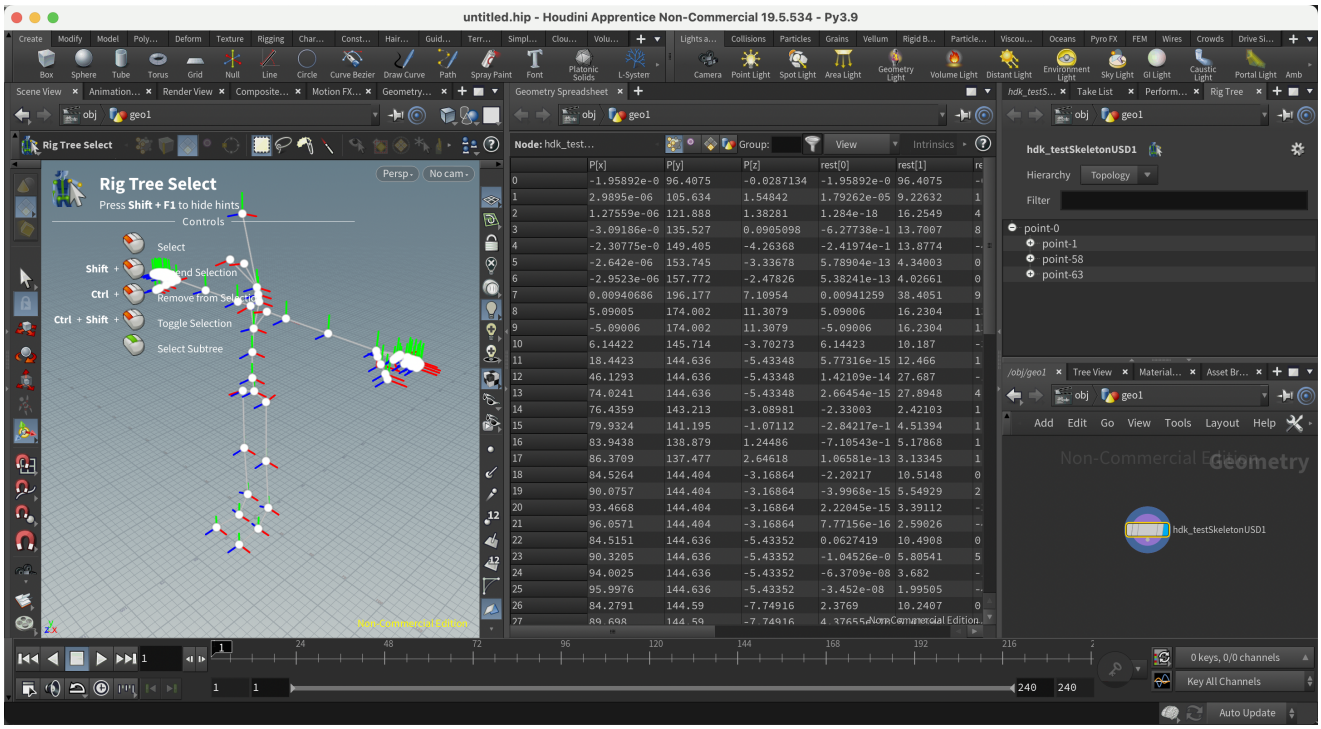

Figure 14: Test complex skeleton shown in Houdini interface using USD import plugin

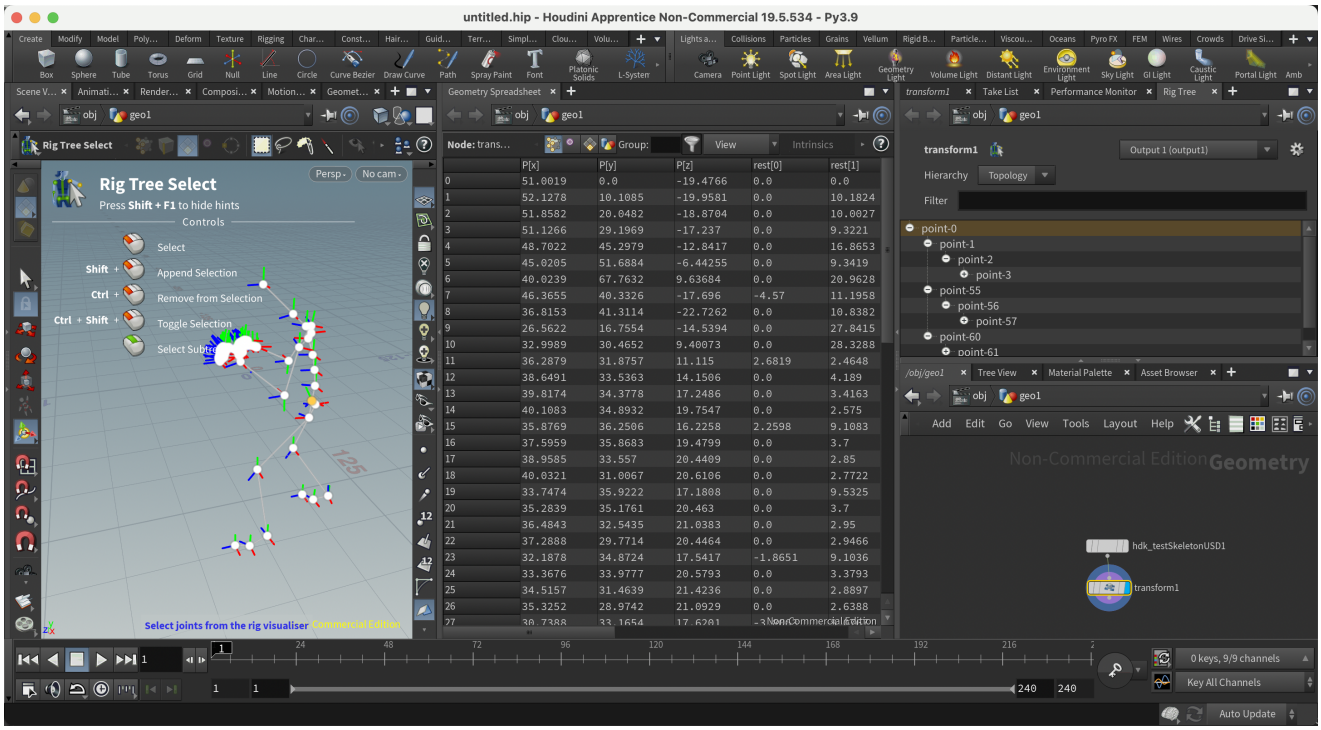

Figure 15: Test complex skeleton shown in Houdini interface using USD import plugin

5 Conclusion

The project embarked on an ambitious journey to bridge the gap between Maya and Houdini, two giants in the 3D animation industry. By developing innovative plugins, I achieved a streamlined transfer of mesh and skeleton data from Maya to Houdini using the Universal Scene Description (USD) format. This accomplishment not only outpaces traditional methods like FBX and ABC in terms of efficiency but also ensures data fidelity, a crucial component in 3D animation.

The implications of this innovation are profound. The 3D animation industry thrives on efficiency, precision, and the constant exchange of digital assets between multiple platforms. This new process significantly simplifies the workflow, potentially reducing the time and resources spent on asset transfers and troubleshooting. Studios, both big and small, can benefit from this tool, making complex projects more manageable and fostering greater collaboration between artists using different software.

However, the journey is far from complete. Given the constraints of time, our project's scope was predominantly limited to the transfer of mesh and skeleton data. The future trajectory of our research aims to expand into more intricate territories. Foremost among these is the study of animation conversion, a critical aspect in the continuum of 3D design processes. Additionally, the complex dynamics of bend transformations, particularly when intertwining both mesh and skeleton, present another avenue demanding comprehensive exploration. While significant strides have been achieved, vast realms remain untapped. These uncharted domains, encompassing the conversion of animations and the amalgamation of mesh with skeleton-in-bend transformations, will be pivotal in our forthcoming research endeavors.

References

Ardolino, A., Arnaud, R., Berinstein, P., Franco, S., Herubel, A., McCutchan, J., Nedelcu, N., Nitschke, B., Robinet, F., Ronchi, C., et al. (2014). Geometry and models: 3d format conversion (fbx, collada). *Game Development Tool Essentials*, 19–37.

Laforge, G., Yudin, V., Caillaud, L., & Couet, J. (2018). Walter: An open source vfx framework for usd and alembic. In *Acm siggraph 2018 talks* (pp. 1–2).

Lazor, M., Gajić, D. B., Dragan, D., & Duta, A. (2019). Automation of the avatar animation process in fbx file format. *FME Transactions*, *47*(2), 398–403.

Lee, G.-h., Choi, P.-h., Nam, J.-h., Han, H.-s., Lee, S.-h., & Kwon, S.-c. (2019). A study on the performance comparison of 3d file formats on the web. *International journal of advanced smart convergence*, *8*(1), 65–74.

McHenry, K., & Bajcsy, P. (2008). An overview of 3d data content, file formats and viewers. *National Center for Supercomputing Applications*, *1205*, 22.

Piriya, S. L., & Mahendran, S. (2014). Procedural function based modelling of three dimensional objects modelling three dimensional primitives using node-based architecture. *Int. J. Innov. Res. Adv. Eng*, *1*, 181–186.

Schierenbeck, N. (2022). Collaborative topology exchange in autodesk maya: Enabling peer-to-peer online topology editing through the maya c++ and python api.